### Code Generation

- **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 tiles.

- 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.

### Target 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)

### MIPS

<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>
<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>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>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>--------------------------------------------------</td>
</tr>
<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)

### Registers

<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>$esp</td>
<td>$sp</td>
<td>Expression Stack Pointer – The expression stack holds temporary values for expression evaluations</td>
</tr>
<tr>
<td>$v0</td>
<td>$v0</td>
<td>Result Register – Holds the return value for functions</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Mnemonic Name</th>
<th>SPIM Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ra</td>
<td>$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>$t0</td>
<td>$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 – \( a * b + c * d * \text{foo}(x) \) – 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

### Expression Stack

- 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

### 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

08-13: **Simple Tiling**

• 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?

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

08-17: **Expression Trees**

• Arithmetic Binary Operations

```
+  
\( \text{Constant(3)} \) \hspace{1cm} \( \text{Constant(4)} \)
```

• Instead of using a single tile to cover this tree, we will use three.
08-18: **Expression Trees**

- Arithmetic Binary Operations

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

- 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

08-20: **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
```

08-22: **Expression Trees**

```
+ \\
Constant(3)  Constant(4)
```
• Complete Assembly:

08-23: Expression Trees

![Expression Tree Diagram]

```assembly
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 4 into the register $t1
sw $t1, 0($ESP)    # Store $t1 on the top of the expression stack
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
```

08-24: Expression Trees

• 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)

```assembly
sw $FP, 0($ESP)    # Store frame pointer
addi $ESP, $ESP, -4 # Update the ESP
```

08-26: Expression Trees

• Tiling the tree:

![Tiled Tree Diagram]

08-27: Expression Trees

• Tiling the tree:

![Tiled Tree Diagram]
08-28: Expression Trees

- Store frame pointer on the top of the expression stack
- Update the expression stack pointer
- Load the constant value 8 into the register $t1$
- Store $t1$ on the top of the expression stack
- Update the expression stack pointer
- Load the first operand into temporary $t2$
- Load the second operand into temporary $t2$
- Do the subtraction, storing result in $t1$
- Update the expression stack pointer

08-29: 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

- 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

08-31: Expression Trees

- 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

08-32: Expression Trees

• 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
  ```

08-34: Expression Trees

• Relational Operations
  • <=, >=
    • SimpleJava uses all integer operands
    • x <= y iff (x-1) < y

08-35: Expression Trees

• Relational Operations
  • ==, !=
    • Can’t use slt easily
    • Use beq instead

08-36: 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

08-39: **Expression Trees**

- 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 slt 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 slt for NOT

08-40: **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**
08-43: Expression Trees

- Memory Example

```plaintext
aw $FP, 0($ESP)  % Store frame pointer on the top of the expression stack
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
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
```

08-44: Expression Trees

- Memory Example

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

08-46: Expression Trees

- Function calls

```
Call("Foo")
```

08-47: Expression Trees
- Function calls

```
Call("Foo")
```

```
Constant(9)  Constant(3)
```

08-48: **Expression Trees**

```
Call("Foo")
```

```
Constant(9)  Constant(3)
```

- Code for the Call tile

```assembly
lw  $t1  4($ESP)
sw  $t1  0($ESP)
lw  $t1  8($ESP)
sw  $t1  -4($ESP)
addi $SP, $SP, -8
addi $ESP, $ESP, -8
jal foo
addi $SP, $SP, 8
sw  $result, 0($ESP)
addi $ESP, $ESP, -4
```

08-49: **Simple Tiling**

- 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:

```
Label1:
```

08-51: **Statement Trees**

- 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)
08-52: Statement Trees

- Move Trees (Moving into Registers)

```
  Move
  Register(r1)  Constant(8)
```

08-53: Statement Trees

- 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

```
  lw  $r1, 4($ESP)
  addi $ESP, $ESP, 4
```

08-54: Statement Trees

- 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
```

**08-56: Statement Trees**

- Move Trees (Moving into MEMORY locations)

```
Move
Memory Constant(4)
Register(FP)
```

**08-57: Statement Trees**

- Move Trees (Moving into MEMORY locations)

```
Move
Memory Constant(4)
Register(FP)
```

- 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)

```
Move
Mem
```

```assembly
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)
Move
Memory
Constant(4)
Register(FP)

`sw $FP, 0($ESP)` % Store the frame pointer on the expression stack
`addi $ESP, $ESP, -4` % Update the expression stack pointer
`addi $t1, $ZER0, 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
```

08-62: **Statement Trees**

- Conditional Jump Trees
  - Evaluate the expression
  - Jump if the expression ! = 0

08-63: **Statement Trees**

- Conditional Jump Trees

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

08-64: **Statement Trees**

- Conditional Jump Trees
08-65: **Statement Trees**

- Conditional Jump Trees

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

08-66: **Statement Trees**

- Conditional Jump Trees

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

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

08-67: **Statement Trees**

```
CondJump("jumplab")
<
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 $t1, 4($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 %--
```

08-68: **Statement Trees**

- Sequential Trees

```
addi $t1, $zero, 3 %--
sw $t1, 0($ESP) % Tile for const(3)
addi $ESP, $ESP, -4 %--
addi $t1, $zero, 4 %--
sw $t1, 4($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 %--
```

08-69: **Statement Trees**

- Sequential Trees

```
addi $t1, $zero, 3 %--
sw $t1, 0($ESP) % Tile for const(3)
addi $ESP, $ESP, -4 %--
addi $t1, $zero, 4 %--
sw $t1, 4($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 %--
```

08-70: **Statement Trees**

- Sequential Trees

```
addi $t1, $zero, 3 %--
sw $t1, 0($ESP) % Tile for const(3)
addi $ESP, $ESP, -4 %--
addi $t1, $zero, 4 %--
sw $t1, 4($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 %--
```

After we have emitted code for the left and right subtrees, what do we need to do?
• Nothing!
  • Sequential trees have no associated code

08-71: **Statement Trees**
  • Empty Statement Trees
    • No action is required
    • No code associated with tile for empty trees

08-72: **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

08-74: **Improved Tiling**
  • 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

08-75: **Accumulator Register**
  • 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

08-76: **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:

```assembly
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

08-80: **Accumulator Register**

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

```assembly
addi $ACC, $r1, 0
```

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!

08-83: **Accumulator Register**

• 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)
```

08-85: **Accumulator Register**

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

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

- 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)

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

<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

08-87: **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

```
```

08-91: **Accumulator Register**

- Memory Example

```
```

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

08-93: Accumulator Register

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

08-94: Accumulator Register

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

addi $r1, $ACC, 0

08-95: Accumulator Register

- 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
<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

```
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
```

08-99: **Accumulator Register**

- Memory Move Example

08-100: **Accumulator Register**

- Memory Move Example

08-101: **Accumulator Register**
- 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

  ```
  sw $ACC, -(4n-4)(SP)  # Store first argument on the call stack
  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 $ACC, $result, 0 # Store result of function in accumulator
  ```

08-103: Accumulator Register

- Procedure Calls

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

08-104: Accumulator Register

- Conditional Jumps

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

08-105: Accumulator Register

- Conditional Jumps

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

  <Code for conditional expression>

  ```
  bgtz $ACC, jumplab
  ```

08-106: Accumulator Register

- Jumps, labels, sequential statements

  - Do not have subtrees with values
  - Code is the same as previously defined

08-107: Larger Tiles

- 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)
```

08-109: Larger Tiles Example

- 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
lw $t1, 4($ESP) \% code for tile 3
addi $ESP, $ESP, 4 \% code for tile 3
sub $ACC, $t1, $ACC \% code for tile 3
lw $t1, 4($ESP) \% code for tile 4
addi $ESP, $ESP, -4 \% code for tile 4
sw $ACC, 0($t1) \% code for tile 5
addi $ESP, $ESP, 4 \% code for tile 5
lw $t1, 4($ESP) \% code for tile 5
sw $ACC, 0($t1) \% code for tile 5
```

08-110: Larger Tiles Example

- 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
lw $t1, 4($ESP) \% code for tile 3
addi $ESP, $ESP, 4 \% code for tile 3
sub $ACC, $t1, $ACC \% code for tile 3
lw $t1, 4($ESP) \% code for tile 4
addi $ESP, $ESP, -4 \% code for tile 4
sw $ACC, 0($t1) \% 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
08-112: Larger Tiles Example

- Using Larger Tiles

08-113: Larger Tiles Example

- Code for Tile 2?
Move Memory Constant(7)

• Code for Tile 2?

sw $ACC, -4($FP)

08-115: Larger Tiles Example

addi $ACC, $zero 7 % tile 1
sw $ACC, -4($FP) % tile 2

08-116: Larger Tiles

• 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

```
Move
  Memory
  Register(r1)
   -
Register(r2) Costant(x)
```

08-119: Larger Tiles – Examples

```
Move
  Memory
  Register(r1)
   -
Register(r2) Costant(x)
```

```
sw r1 -x(r2)
```

08-120: Larger Tiles – Examples

```
Move
  Memory
  Subtree
   -
Register(r2) Costant(x)
```

08-121: Larger Tiles – Examples

```
Move
  Memory
  Subtree
   -
Register(r2) Costant(x)
```

```
<code for Subtree>
sw $ACC, -x(r2)
```

08-122: Larger Tiles – Examples
08-123: Larger Tiles – Examples

Move
Memory
Subtree2
Costant(x)
Subtree1
<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

08-125: Larger Tiles – Examples

Move
Memory
register(r1)
Subtree

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

```
sw $ACC, 0($ESP)
addi $ESP, $ESP
```

08-127: Larger Tiles – Examples

```
lw $t1, 4($ESP)
addi $ESP, $ESP, 4
sw $ACC, 0($t1)
```

08-128: 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

08-130: 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

08-131: Larger Tiles

- Conditional Jump Trees

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

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

08-132: Larger Tiles

- Conditional Jump Trees

```
Tile 4
```

```
CondJump("jumplab")

Tile 3
```

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

Tile 1
```

```
Tile 2
```

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:
addi $ACC, $zero, 1  % Tile 3
endlab1:
```

```
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

08-135: Larger Tiles

- Conditional Jump Trees
08-136: **Larger Tiles**

- Conditional Jump Trees

![Diagram of a conditional jump tree with labels](Image)

\[
\begin{align*}
\text{addi} & \qquad \text{ACC, zero} 3 \quad \% \text{Tile 1} \\
\text{sw} & \qquad \text{ACC, 0(ESP)} \quad \% \text{Tile 3} \\
\text{addi} & \qquad \text{ESP, ESP, -4} \quad \% \text{Tile 3} \\
\text{addi} & \qquad \text{ACC, zero, 4} \quad \% \text{Tile 2} \\
\text{lw} & \qquad \text{t1, 4(ESP)} \quad \% \text{Tile 3} \\
\text{addi} & \qquad \text{ESP, ESP, 4} \quad \% \text{Tile 3} \\
\text{beq} & \qquad \text{t1, ACC, jumplab} \quad \% \text{Tile 3}
\end{align*}
\]

08-137: **Larger Tiles**

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

![Diagram of a conditional jump tree with labels](Image)

08-138: **Larger Tiles**

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

![Diagram of a conditional jump tree with labels](Image)

\[
\text{bltz} \quad \text{ACC, jumplab}
\]

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?)

08-141: **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?)
  • We can always find a tiling this way – provided we include all unit-sized tiles in our tile set

08-142: **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

08-144: **Constant Stack Offsets**

• Pushing an expression:

  \[
  \begin{align*}
  &\text{sw} \quad \$\text{ACC}, \quad 0(\$\text{ESP}) \\
  &\text{addi} \quad \$\text{ESP}, \quad \$\text{ESP}, \quad -4
  \end{align*}
  \]

• Popping an expression:

  \[
  \begin{align*}
  &\text{addi} \quad \$\text{ESP}, \quad \$\text{ESP}, \quad 4 \\
  &\text{lw} \quad \$\text{ACC}, \quad 0(\$\text{ESP})
  \end{align*}
  \]
08-145: **Constant Stack Offsets**

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

  (decrement <offset> by 4)

- Popping an expression:
  
  ```
  lw $ACC, <offset>($ESP)
  ```

08-146: **Constant Stack Offsets**

- Example:

  ```
  addi $ACC, $zero, 9  # Tile 1
  sw $ACC, 0($ESP)     # Tile 7 -- pushing on expression stack
  addi $ESP, $ESP, -4  # Tile 7  
  addi $ACC, $zero, 8  # Tile 2
  sw $ACC, 0($ESP)     # Tile 6 -- pushing on expression stack
  addi $ESP, $ESP, -4  # Tile 6  
  addi $ACC, $zero, 7  # Tile 3
  sw $ACC, 4($ESP)     # Tile 5 -- pushing on expression stack
  addi $ESP, $ESP, -4  # Tile 5  
  addi $ACC, $zero, 6  # Tile 4
  sw $ACC, 8($ESP)     # Tile 5  
  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
  ```

08-147: **Constant Stack Offsets**

- Example:

  ```
  addi $ACC, $zero, 9  # Tile 1
  sw $ACC, 0($ESP)     # Tile 7 -- pushing on expression stack  
  addi $ESP, $ESP, -4  # Tile 7  
  addi $ACC, $zero, 8  # Tile 2
  sw $ACC, 0($ESP)     # Tile 6 -- pushing on expression stack  
  addi $ESP, $ESP, -4  # Tile 6  
  addi $ACC, $zero, 7  # Tile 3
  sw $ACC, 4($ESP)     # Tile 5 -- pushing on expression stack  
  addi $ESP, $ESP, -4  # Tile 5  
  addi $ACC, $zero, 6  # Tile 4
  sw $ACC, 8($ESP)     # Tile 5  
  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
  ```

08-148: **Constant Stack Offsets**

- Example:

  ```
  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
  ```
08-149: Constant Stack Offsets

- Using constant offsets off the $ESP words well most of the time
- There are problems with function calls, however
  - \( x = y + (z + \text{bar}(2)) \)

08-150: Constant Stack Offsets

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

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

- What’s wrong with this code?

08-151: Constant Stack Offsets

- 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 ..

08-152: Constant Stack Offsets

- 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 \text{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 \emph{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

**08-155: 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 stack
addi $ACC, $zero, 2 # Load actual parameter (2) into ACC
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 $x$
```

**08-156: Optimizing Expression Stack**

- 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

**08-158: 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
08-159: Using Registers

- Constant stack offsets (no registers)

```assembly
addi $ACC, $zero, 9  \% Tile 1  -- pushing on expression stack
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
lw $t1, 0($ESP)     \% Tile 7  -- popping from expression stack
lw $ACC, xoffset($FP) \% Tile 8
```

08-160: Using Registers

- Bottom of expression stack in registers

```assembly
addi $ACC, $zero, 9  \% Tile 1  -- pushing on expression stack
addi $t2, $ACC, 0    \% Tile 7  -- pushing on expression stack
addi $ACC, $zero, 8  \% Tile 2
addi $t3, $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
sub $ACC, $t3, $ACC \% Tile 6  -- popping from expression stack
lw $t1, 0($ESP)      \% Tile 6  -- popping from expression stack
sub $ACC, $t2, $ACC \% Tile 7  -- popping from expression stack
lw $ACC, xoffset($FP) \% Tile 8
```

08-161: 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?

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?

- 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

08-164: **Using Registers**

```assembly
addi $ACC, $zero, 9 % Tile 1 -- pushing on expression stack
addi $r2, $ACC, 0 % Tile 7 -- pushing on expression stack
addi $ACC, $zero, 8 % Tile 2
addi $r3, $ACC, 0 % Tile 6
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
sw $ESP, $ESP, -12 % Tile 4 -- Update ESP
jal foo % Tile 4
addi $ESP, $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
lw $t1, 0($ESP) % Tile 5 -- popping from expression stack
sub $ACC, $t1, $ACC % Tile 5
addi $t1, $t3, 0 % Tile 6
addi $t1, $t2, 0 % Tile 7
lw $ACC, xoffset($FP) % Tile 8
```

08-166: **Extended Example**

```c
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**
08-168: Extended Example – Basic

```
sw $FP, 0($ESP)  % Tile 1
addi $ESP, $ESP, -4  % Tile 1
sw $t1, $t2, $szero, 4  % Tile 2
addi $ESP, $ESP, -4  % Tile 2
lw $t1, 8($ESP)  % Tile 3
lw $t2, 4($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 $t2, 4($ESP)  % Tile 4
addi $ESP, $ESP, 4  % Tile 4
lw $t1, 0($t1)  % Tile 5
sw $t1, 4($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
```

08-169: Extended Example – Basic

```
sw $FP, 0($ESP)  % Tile 1
addi $ESP, $ESP, -4  % Tile 1
add $t1, $t2, $t3  % Tile 2
lw $t1, 8($ESP)  % Tile 3
lw $t2, 4($ESP)  % Tile 3
add $t1, $t2, $t1  % Tile 3
sw $t1, 8($ESP)  % Tile 3
addi $ESP, $ESP, 4  % Tile 3
lw $t1, 4($ESP)  % Tile 4
lw $t2, 4($ESP)  % Tile 4
addi $ESP, $ESP, 4  % Tile 4
lw $t1, 0($t1)  % Tile 5
sw $t1, 4($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
```

08-170: 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
lw $t1, 0($ESP) % Tile 9
addi $ESP, $ESP, -4 % Tile 9
lw $t1, 4($ESP) % Tile 10
lw $t2, 8($ESP) % Tile 10
sw $t1, 0($ESP) % Tile 10
addi $ESP, $ESP, -4 % Tile 10
lw $t1, 4($ESP) % Tile 11
lw $t2, 8($ESP) % Tile 11
sw $t1, 8($ESP) % Tile 11
lw $t1, 0($ESP) % Tile 11
addi $ESP, $ESP, -4 % Tile 11
lw $t1, 4($ESP) % Tile 12
lw $t2, 8($ESP) % 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
lw $t1, 0($ESP) % Tile 13
addi $ESP, $ESP, -4 % Tile 13
lw $t1, 4($ESP) % Tile 14
lw $t2, 8($ESP) % Tile 14
sw $t1, 0($ESP) % Tile 14
addi $ESP, $ESP, -4 % Tile 14
lw $t1, 4($ESP) % Tile 15
lw $t2, 8($ESP) % Tile 15
sw $t2, 0($t1) % Tile 15
addi $ESP, $ESP, -4 % Tile 15
sw $t1, 8($ESP) % Tile 15
addi $ESP, $ESP, 4 % Tile 15
iftrue1: % Tile 16
lw $t1, 4($ESP) % Tile 17
lw $t1, 0($t1) % Tile 17
sw $t1, 4($ESP) % Tile 17
lw $t1, 4($ESP) % Tile 18
lw $t2, 8($ESP) % Tile 18
sw $t1, 0($ESP) % Tile 18
addi $ESP, $ESP, -4 % Tile 18
lw $t1, 4($ESP) % Tile 19
lw $t2, 8($ESP) % 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 $t1, 0($ESP) % Tile 20
addi $ESP, $ESP, -4 % Tile 20
lw $t1, 4($ESP) % Tile 21
lw $t2, 8($ESP) % Tile 21
addi $ESP, $ESP, -4 % Tile 21
lw $t1, 0($t1) % Tile 21
sw $t1, 0($ESP) % Tile 21
addi $ESP, $ESP, -4 % Tile 21
lw $t1, 4($ESP) % Tile 22
lw $t2, 8($ESP) % Tile 22
addi $ESP, $ESP, -4 % Tile 22
lw $t1, 4($ESP) % Tile 23
lw $t2, 8($ESP) % Tile 23
sw $t1, 0($ESP) % Tile 23
addi $ESP, $ESP, -4 % Tile 23
lw $t1, 8($ESP) % Tile 23
lw $t1, 0($ESP) % Tile 23
addi $ESP, $ESP, 4 % Tile 23
lw $t1, 4($ESP) % Tile 24
lw $t2, 8($ESP) % Tile 24
sw $t1, 0($ESP) % Tile 24
addi $ESP, $ESP, -4 % Tile 24
lw $t1, 4($ESP) % Tile 25
lw $t2, 8($ESP) % Tile 25
addi $ESP, $ESP, -4 % Tile 25
lw $t1, 0($ESP) % Tile 25
lw $t1, 4($ESP) % Tile 26
lw $t2, 8($ESP) % Tile 26
addi $ESP, $ESP, -4 % Tile 26
lw $t1, 0($t1) % Tile 26
sw $t1, 0($ESP) % Tile 26
addi $ESP, $ESP, -4 % Tile 26
lw $t1, 4($ESP) % Tile 27
lw $t2, 8($ESP) % Tile 27
sw $t1, 0($t1) % Tile 27
lw $t1, 0($ESP) % Tile 27
addi $ESP, $ESP, 8 % Tile 27
Ifend1: % Tile 28

08-171: Extended Example – Basic

08-172: Extended Example – Basic

08-173: Extended Example – Basic

08-174: Extended Eg. – Accumulator
08-175: Extended Eg. – Accumulator

```assembly
addi $ACC, $FP, 0  % Tile 1
sw $ACC, 0($ESP)   % Tile 3
addi $DSP, $ESP, -4 % Tile 3
addi $ACC, $zero, 1 % Tile 3
lw $t1, 0($ESP)    % Tile 3
addi $ESP, $ESP, 4 % Tile 3
sw $ACC, 0($ACC)  % Tile 4
lw $ACC, 0($ACC)  % Tile 4
sw $ACC, 0($ESP)  % Tile 6
addi $ESP, $ESP, -4 % Tile 6
addi $ACC, $zero, 8 % Tile 6
lw $t1, 4($ESP)   % Tile 6
addi $ESP, $ESP, 4 % Tile 6
sw $ACC, 0($ACC)  % Tile 8
lw $ACC, 0($ACC)  % Tile 8
```

08-176: Extended Eg. – Accumulator

```assembly
sw $ACC, 0($ESP)   % Tile 10
addi $DSP, $ESP, -4 % Tile 10
addi $ACC, $zero, 7 % Tile 10
lw $t1, 0($ESP)    % Tile 10
addi $ESP, $ESP, 4 % Tile 10
sw $ACC, 0($ACC)  % Tile 12
lw $ACC, 0($ACC)  % 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 13
addi $ESP, $ESP, -4 % Tile 13
addi $ACC, $zero, 11 % Tile 13
lw $ACC, 0($ACC)  % Tile 13
lw $t1, 4($ESP)   % Tile 15
addi $ESP, $ESP, 4 % Tile 15
addi $DSP, $ESP, 4 % Tile 15
addi $ACC, $zero, 15 % Tile 15
```

08-177: Extended Eg. – Accumulator

```assembly
j ifend1  % Tile 16
iftrue:  % Tile 17
addi $ACC, $FP, 0  % Tile 17
sw $ACC, 0($ESP)   % Tile 18
addi $DSP, $ESP, -4 % Tile 18
addi $ACC, $zero, 0 % Tile 18
lw $t1, 0($ESP)    % Tile 18
addi $ESP, $ESP, 4 % Tile 18
addi $ACC, $zero, 4 % Tile 18
lw $ACC, 0($ACC)  % Tile 18
lw $ACC, 0($ACC)  % Tile 21
addi $ESP, $ESP, 4 % Tile 21
addi $DSP, $ESP, 4 % Tile 21
addi $ACC, $zero, 16 % Tile 21
lw $t1, 4($ESP)   % Tile 21
addi $ESP, $ESP, 4 % Tile 21
addi $ACC, $zero, 20 % Tile 21
lw $ACC, 0($ACC)  % Tile 21
lw $ACC, 0($ACC)  % Tile 22
addi $ESP, $ESP, 4 % Tile 22
addi $DSP, $ESP, 4 % Tile 22
addi $ACC, $zero, 23 % Tile 22
lw $t1, 4($ESP)   % Tile 22
addi $ESP, $ESP, 4 % Tile 22
addi $ACC, $zero, 27 % Tile 22
lw $ACC, 0($ACC)  % Tile 22
lw $ACC, 0($ACC)  % Tile 24
addi $ESP, $ESP, 4 % Tile 24
addi $DSP, $ESP, 4 % Tile 24
addi $ACC, $zero, 31 % Tile 24
lw $t1, 4($ESP)   % Tile 24
addi $ESP, $ESP, 4 % Tile 24
addi $DSP, $ESP, 4 % Tile 25
addi $ACC, $zero, 35 % Tile 25
lw $ACC, 0($ACC)  % Tile 25
lw $ACC, 0($ACC)  % Tile 26
```

08-178: Extended Eg. – Accumulator

```assembly
```
08-178: Extended Eg. – Accumulator

\[
\begin{align*}
& \text{addi } \$\text{ESP}, \$\text{ESP}, -4 \quad \% \text{Tile 26} \\
& \text{addi } \$\text{ACC}, \$\text{zero}, 1 \quad \% \text{Tile 25} \\
& \text{lw } \$\text{t1}, 4(\$\text{ESP}) \quad \% \text{Tile 26} \\
& \text{addi } \$\text{ESP}, \$\text{ESP}, 4 \quad \% \text{Tile 26} \\
& \text{lw } \$\text{t1}, 4(\$\text{ESP}) \quad \% \text{Tile 26} \\
& \text{addi } \$\text{ESP}, \$\text{ESP}, 4 \quad \% \text{Tile 26} \\
& \text{sw } \$\text{ACC}, 0(\$\text{t1}) \quad \% \text{Tile 27} \\
& \text{ifend1} \quad \% \text{Tile 28} \\
\end{align*}
\]

% No code for tiles 29 -- 33

08-179: Extended Eg. – Large Tiles

\[
\begin{align*}
& \text{CondJump}(\text{"iftrue1"}) \\
& \text{Mem Constant(5)} \\
& \text{Register(FP) Constant(4)} \\
& \text{Mem} \\
& \text{Register(FP) Constant(4)} \\
& \text{Mem} \\
& \text{Register(FP) Constant(8)} \\
& \text{Jump}(\text{"ifend1"}) \\
& \text{Label}(\text{"iftrue1"}) \\
& \text{Mem} \\
& \text{Register(FP) Constant(4)} \\
& \text{Mem} \\
& \text{Register(FP) Constant(12)} \\
& \text{Label}(\text{"ifend1"}) + \text{Constant(1)}
\end{align*}
\]

08-180: Extended Eg. – Large Tiles

\[
\begin{align*}
& \text{Tile 16} \\
& \text{Tile 15} \\
& \text{Tile 14} \\
& \text{Tile 13} \\
& \text{Tile 12} \\
& \text{Tile 11} \\
& \text{Tile 10} \\
& \text{Tile 9} \\
& \text{Tile 8} \\
& \text{Tile 7} \\
& \text{Tile 6} \\
& \text{Tile 5} \\
& \text{Tile 4} \\
& \text{Tile 3} \\
& \text{Tile 2} \\
& \text{Tile 1}
\end{align*}
\]

08-181: Extended Eg. – Large Tiles

\[
\begin{align*}
& \text{lw } \$\text{ACC}, -4(\$\text{FP}) \quad \% \text{Tile 1} \\
& \text{sw } \$\text{ACC}, 0(\$\text{ESP}) \quad \% \text{Tile 3} \\
& \text{addi } \$\text{ESP}, \$\text{ESP}, -4 \quad \% \text{Tile 3} \\
& \text{addi } \$\text{ACC}, \$\text{zero}, 5 \quad \% \text{Tile 2} \\
& \text{lw } \$\text{t1}, 4(\$\text{ESP}) \quad \% \text{Tile 3}
\end{align*}
\]
addi $ESP, $ESP, 4 % Tile 3
sli $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:
lw $ACC, -12($FP) % Tile 8
addi $ACC, $ACC, 1 % Tile 9
sw $ACC, -4($FP) % Tile 10
ifend:
% 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
sli $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:
lw $ACC, -12($FP) % Tile 8
addi $ACC, $ACC, 1 % Tile 9
sw $ACC, -4($FP) % Tile 10
ifend:
% No code for tiles 12 -- 16

08-183: Further Optimizations

- Tiles 2 and 3 could be covered by a single tile:

\[ \text{CondJump("jumplab")} \]
\[ \text{Constant(x)} \]

addi $ACC, $ACC, -x
bgtz $ACC, jumplab

08-184: Further Optimizations

lw $ACC, -4($FP) % Tile 1
addi $ACC, $ACC, -5 % Tile 3
addi $ACC, $ACC, -5 % Tile 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:
% No code for tiles 12 -- 16

08-185: Implementation Details

- 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 “instanceof” (slightly more ugly than semantic analysis)
08-186: Implementation Details

- 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.