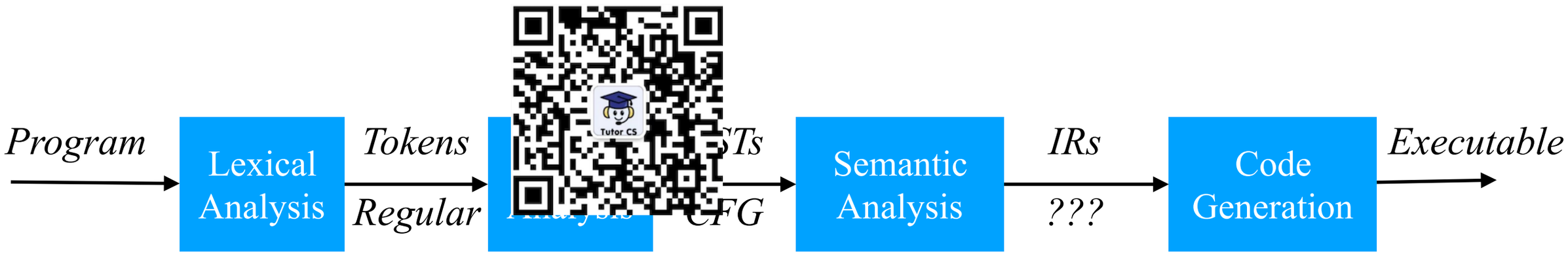
CS 160 Compilers

Lecture 14: Code Generation

Yu Feng

Fall 2021

A typical flow of a compiler



Code Generation

- To generate actual code that can run on a processor (such as gcc) or on a virtual machine (such as javac) we need to understand what code for each of these machines looks like.

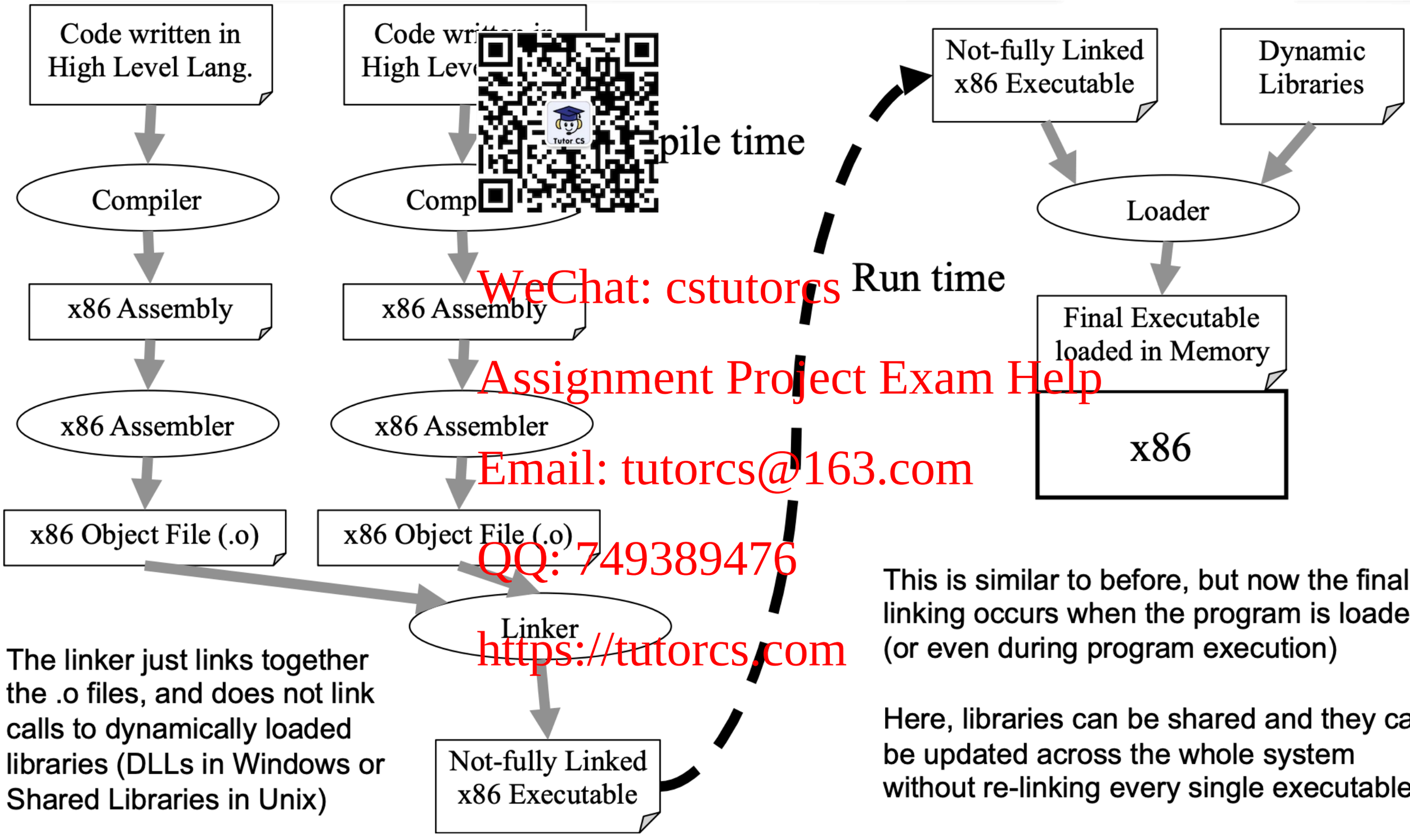
- Rather than worry about the exact syntax of a given assembly language, we instead use a type of pseudo-assembly that is close to the underlying machine.

- In this class, we need to worry about 2 different types of code

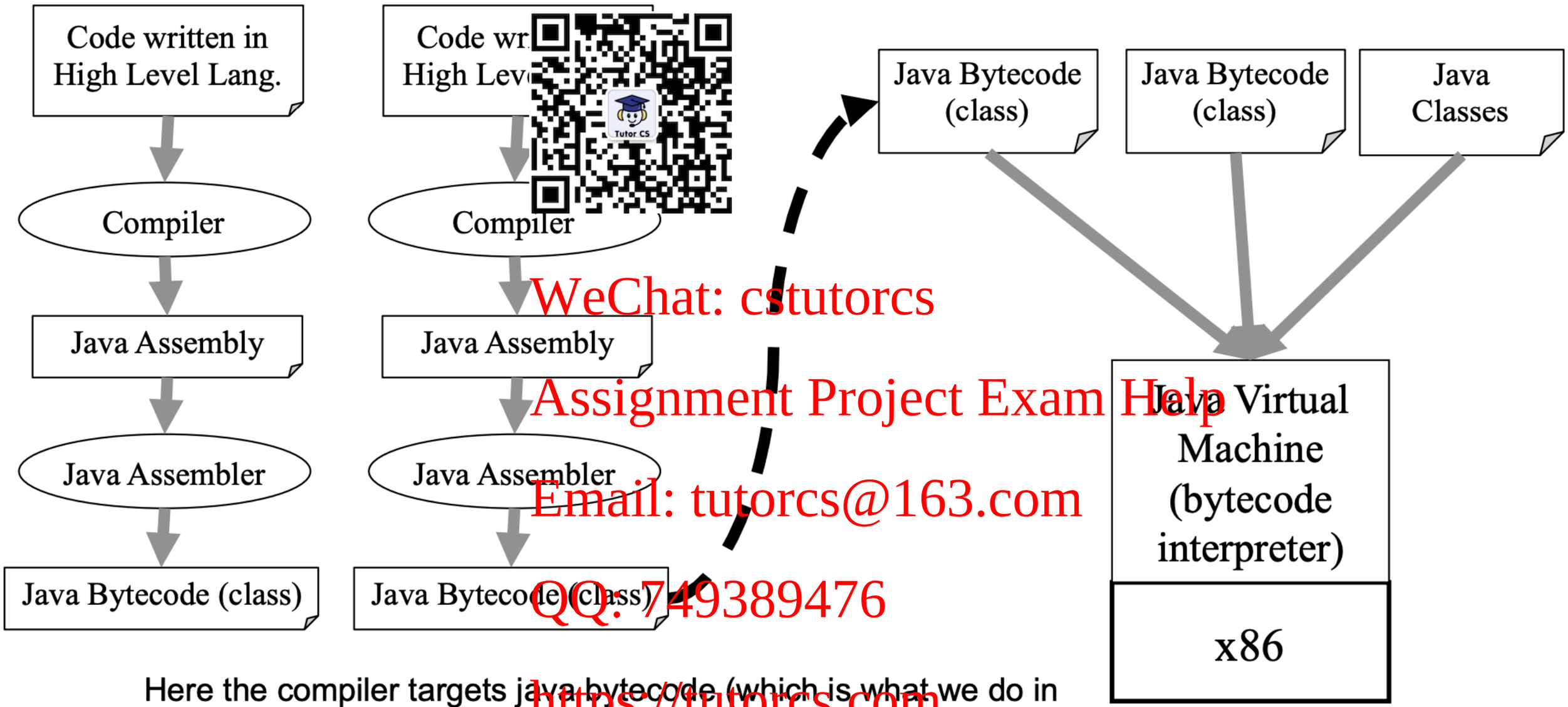
  - Stack based code: Similar to the Java Virtual Machine

  - Register-based code: Similar to most processors (x86, Sparc, ARM)

x86 C Compiler



Java Compiler



Here the compiler targets java bytecode (which is what we do in this class) and the bytecode is then run on top of the Java Virtual Machine (JVM). The JVM really just interprets (simulates) the bytecode like any scripting language. Because of this, any java program compiled to bytecode is portable to any machine that someone has already ported the JVM too. No need to recompile.

Register-based Machine

- Each instruction can ha  ree operands
- We have to break large nto little operations that use temporary variables
  - X=(2+3)+4 turns 2+3; X=T1+4;
- Temporary variables store the results at the internal nodes in the AST
- Assignments
  - x := y
  - x := y op z *op: binary arithmetic or logical operators*
  - x := op y *op: unary operators (minus, negation, integer to float)*
- Branch
  - goto L *execute the statement with labeled L next*
- Conditional Branch
  - if x relop y goto L relop: <, =, <=, >=, ==, !=
    - if the condition holds, we execute statement labeled L next
    - if the condition does not hold, we execute the statement following this statement next

Register-based Machine


```
if (x < y)
  x = 5*y + 5*y/3;
else
  y = 5;
x = x + y;
```

Variables can be represented with their locations in the symbol table

```
if x < y goto L1
goto L2
L1:   t1 := 5 * y
      t2 := 5 * y
      t3 := t2 / 3
      x := t1 + t2
      goto L3
L2:   y := 5
L3:   x := x + y
```

Temporaries: temporaries correspond to the internal nodes of the syntax tree

- Three-address code instructions can be represented as an array of
  - quadruples: operation, argument1, argument2, result
  - triples: operation, argument1, argument2
    (each triple implicitly corresponds to a temporary)

Stack-based Machine

- Stack based code uses a stack to store temporary variables

- When we evaluate an expression *(E+E)*, it will take its arguments off the stack, add them together and put the result back on the stack.

- (2+3)+4 will *push 2; push 3; add; push 4; add*

- The machine code for this is a bit more ugly but the code is actually easier to generate because we do not need to handle temporary variables.

Stack-based Machine


```
if (x < y)
   x = 5*y + 5*y/3;
else
   y = 5;
x = x+y;
```


```
      load x
      load y
      iflt L1
      goto L2
L1:   push 5
      load y
      multiply
      push 5
      load y
      multiply
      push 3
      divide
      add
      store x
      goto L3
L2:   push 5
      store y
L3:   load x
      load y
      add
      store x
```

he value
cation x to
x

pops the top two elements and compares them

pops the top two elements, multiplies them, and pushes the result back to the stack

stores the value at the top of the stack to the location x

JVM: A stack machine

- JVM interpreter executes the bytecode on different machines
- JVM has an operand stack which we use to evaluate expressions
- JVM provides 65,535 local variables for each method The local variables are like registers so we do not have to worry about register allocation
- Each local variable in JVM is denoted by a number between 0 and 65535 (x and y in the example will be assigned unique numbers)

Stack-based vs. Register-based

- Register-Based code:
  - Good - Compact representation
  - Good - "Self contained" has inputs, outputs, and operation all in one "instruction"
  - Bad - Requires lots of temporary variables
  - Bad - Temporary variables have to be handled explicitly
- Stack Based Code:
  - Good – No temporaries, everything is kept on the stack
  - Good – It is easy to generate code for this
  - Bad – Requires more instructions to do the same thing

Expressions

- We are targeting a regi machine

- We need to evaluate expressions assuming a very limited set of available registers (No register allocation).

- To generate code for an expression we will do a recursive traversal in post-order (that is, visit the children first, then generate code for the parent).

Expressions

- Let's start with a simpl 

expression: `(1 + 2) * (3 - 4)`

```
call generate_aexp(* node, left):
  call generate_aexp(+ node, left):
    call generate_aexp(1 node, left):
      emit "mov 1 LEFT_REG"
    call generate_aexp(2 node, right):
      emit "mov 2 RIGHT_REG"
    emit "add RIGHT_REG LEFT_REG"
call generate_aexp(- node, right):
  call generate_aexp(3 node, left):
    emit "mov 3 LEFT_REG"
  call generate_aexp(4 node, right):
    emit "mov 4 RIGHT_REG"
  emit "sub RIGHT_REG LEFT_REG"
emit "mul RIGHT_REG LEFT_REG"
```

```
mov 1 LEFT_REG
mov 2 RIGHT_REG
add RIGHT_REG LEFT_REG
mov 3 LEFT_REG
mov 4 RIGHT_REG
sub RIGHT_REG LEFT_REG
mov LEFT_REG RIGHT_REG
mul RIGHT_REG LEFT_REG
```


What is the problem?

Expressions

- We have to create memory locations to hold temporary values during expression evaluation.

```
call generate_aexp(* node, tmp_num =
  call generate_aexp(+ node, tmp_num
    call generate_aexp(1 node, tmp_num = 2):
      emit "mov 1 RESULT_REG"
    insert _tmp1 into symbol table
    emit "store RESULT_REG [_tmp1]"
    call generate_aexp(2 node, tmp_num = 2):
      emit "mov 2 RESULT_REG"
    emit "ld [_tmp1] OTHER_REG" "add OTHER_REG RESULT_REG"
    remove _tmp1 from symbol table
  insert _tmp0 into symbol table
  emit "store RESULT_REG [_tmp0]"
  call generate_aexp(- node, tmp_num = 1):
    insert _tmp1 into symbol table
    call generate_aexp(3 node, tmp_num = 2):
      emit "mov 3 RESULT_REG"
    emit "store RESULT_REG [_tmp1]"
    call generate_aexp(4 node, tmp_num = 2):
      emit "mov 4 RESULT_REG"
    emit "ld [_tmp1] OTHER_REG" "sub RESULT_REG OTHER_REG" "mov OTHER_REG RESULT_REG"
    remove _tmp1 from symbol table
  emit "ld [_tmp0] OTHER_REG" "mul OTHER_REG RESULT_REG"
  remove _tmp0 from symbol table
```

```
mov 1 RESULT_REG
store RESULT_REG [_tmp1]
mov 2 RESULT_REG
ld [_tmp1] OTHER_REG
add OTHER_REG RESULT_REG
store RESULT_REG [_tmp0]
mov 3 RESULT_REG
store RESULT_REG [_tmp1]
mov 4 RESULT_REG
ld [_tmp1] OTHER_REG
sub RESULT_REG OTHER_REG
mov OTHER_REG RESULT_REG
ld [_tmp0] OTHER_REG
mul OTHER_REG RESULT_REG
```

Let's Generalize It

- Let's generalize the algorithm for arbitrary arithmetic expressions

```
generate_aexp(AST* node, int tmp_num = 0) {
  if (node is a constant <n>) { emit "mov <n> RESULT_REG";
return; }
  if (node is a variable <x>) { emit "ld [x] RESULT_REG"; return; }

  // node must be one of +,-,*
  generate_aexp(node->left, tmp_num+1);
  insert _tmp<tmp_num> into symbol table;
  emit "store RESULT_REG [_tmp<tmp_num>]";
  generate_aexp(node->right, tmp_num+1);
  emit "ld [_tmp<tmp_num>] OTHER_REG";

  // left-hand value is in OTHER_REG, right-hand value is in RESULT_REG
  if (node is +) { emit "add OTHER_REG RESULT_REG"; return; }
  if (node is -) { emit "sub RESULT_REG OTHER_REG"; emit "mov OTHER_REG
RESULT_REG"; return; }
  emit "mul OTHER_REG RESULT_REG";

  remove _tmp<tmp_num> from symbol table;
}
```


https://sites.cs.ucsb.edu/~emre/cs160/#orgfa5615b

Assignment

- We evaluate the right-hand s[illegible]nerate_aexp, which puts the result in RESULT_REG, then store th[illegible]he memory location for the left-hand side variable.

```
generate_assign(lhs, rhs) {
  generate_aexp(rhs);
  emit "store RR [lhs]";
}
```

Non-nested Conditionals

- Conditionals without a [illegible]ope

```
[illegible](x < 2) { x := 1; } else { x := 2; }
```

```
generate_if(node) {
  <n> = fresh index;
  generate_rexp(node->guard);
  emit "cmp 0 RESULT_REG";
  emit "jmpe IF_FALSE_<n>";
  generate_block(node->true_branch);
  emit "jmp IF_END_<n>";
  emit "IF_FALSE_<n>:";
  generate_block(node->false_branch);
  emit "IF_END_<n>:";
}
```

```
    ld [x] RR
    store RR [_tmp0]
    mov 2 RR
    ld [_tmp0] OR
    cmp RR OR
    setlt RR
    cmp 0 RR
    jmpe IF_FALSE_0
    mov 1 RR
    store RR [x]
    jmp IF_END_0
IF_FALSE_0:
    mov 2 RR
    store RR [x]
IF_END_0:
```

Nested Conditionals

- When the code generator en[illegible]v scope:
    - see how many declared v[illegible]re are
    - adjust the stack pointer accordingly
    - initialize the new memory locations to 0
    - update symbol table to map the newly declared variables to their offsets
- and when we leave the new scope we need to adjust things back the way they were:
    - reset the stack pointer to its old position
    - restore the symbol table to its old value

Nested Conditionals


```
generate_block(node) {
  old_symbol_table = s                    e;
  stack_size = node->r                    d_variables * 4; // because 4-byte
integers
  emit "sub <stack_size> STACK_REG";
  insert_in_symbol_table(symbol_table, node->declared_variables);
  for each var in node->declared_variables { emit "store 0 [var]"; }
  .
  .
  .
  emit "add <stack_size> STACK_REG";
  symbol_table = old_symbol_table;
}
```

While Loops

- Conditionals without a … ope

```
le (x < 3) { x := x + 1; }
```

```
generate_while(node) {
  <n> = fresh index;
  emit "WHILE_START_<n>:";
  generate_rexp(node->guard);
  emit "cmp 0 RESULT_REG";
  emit "jmpe WHILE_END_<n>";
  generate_block(node->body);
  emit "jmp WHILE_START_<n>";
  emit "WHILE_END_<n>:";
}
```

```
WHILE_START_0:
    ld [x] RR
    store RR [_tmp0]
    mov 3 RR
    ld [x] OR
    cmp RR OR
    setlt RR
    cmp 0 RR
    jmpe WHILE_END_0
    ld [x] RR
    store RR [_tmp0]
    mov 1 RR
    ld [_tmp0] OR
    add OR RR
    store RR [x]
    jmp WHILE_START_0
WHILE_END_0:
```

TODOs by next lecture

- Starting AS4

- Review x86 assembly https://en.wikibooks.org/wiki/X86_Assembly

- Move to optimizations next week (last topic)