# 50.054 Code Generation (Stack Machine)

ISTD, SUTD

# Learning Outcomes

1. Name the difference among the target code platforms
2. Implement the target code generation to JVM bytecode given a Pseudo Assembly Program

# Recap Compiler Pipeline

```
[Lexing] -->[Parsing] --> [Semantic Analysis] --> [Optimization]
                                                         |
                                                         ↓
                                                [Target Code Generation]
```

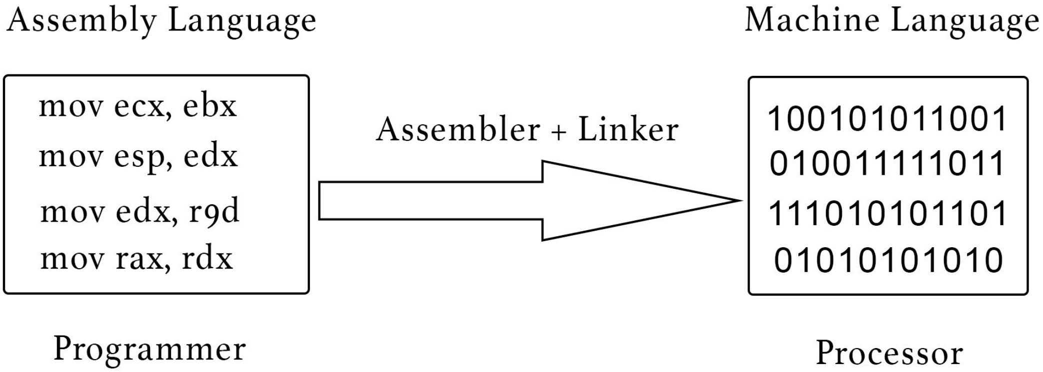
- Target Code Generation
  - Input: some IR as input
  - Output: the target code (executable)

# Instruction Selection

- 3-address instruction
  - RISC (Reduced Instruction Set Computer) architecture. E.g. Apple PowerPC, ARM, Pseudo Assembly
- 2-address instruction
  - CISC (Complex Instruction Set Computer) architecture. E.g. Intel x86
- 1-address instruction
  - Stack machine. E.g. JVM

# Assembly code vs Machine code

- The actual target codes are in binary (Machine code).
- The assembly codes in human readable represntation of the Machine code.



# 3-address instruction

- `dst <- src1 op src2`
- Simple instruction set
- More registers (and temp variables)

```
x <- 1
y <- 2
r <- x + y
```

or

```
load x 1
load y 2
add r x y
```

# 2-address instruction

- `op dst src`
- More complex instruction set
- Fewer registers

```
load x 1
load y 2
add x y
```

- `r` is the same as `x`

# 1-address instruction

- AKA P-code, stack machine code
- More complex instruction set with stack for computation (not to confuse with stack for function call)
- Minimum registers, e.g. JVM has only 3 registers.

```
push 1
push 2
add
store r
```

# JVM bytecode (reduced set)

$$
\begin{array}{rrcl}
(\texttt{JVM Instructions}) & \mathit{jis} & ::= & [\,] \mid \mathit{ji}\ \mathit{jis} \\
(\texttt{JVM Instruction}) & \mathit{ji} & ::= & \mathit{ilabel}\ l \mid \mathit{iload}\ n \mid \mathit{istore}\ n \mid \mathit{iadd} \mid \mathit{isub} \mid \mathit{imul} \\
 & & & \mid \mathit{if\_icmpge}\ l \mid \mathit{if\_icmpne}\ l \mid \mathit{igoto}\ l \mid \mathit{sipush}\ c \mid \mathit{ireturn} \\
(\texttt{JVM local vars}) & n & ::= & 1 \mid 2 \mid \ldots \\
(\texttt{constant}) & c & ::= & -32768 \mid \ldots \mid 0 \mid \ldots \mid 32767
\end{array}
$$

1. a register for the first operand and result
2. a register for the second operand
3. a register for controlling the state of the stack operation (we can't used.)

# An Example

```
// PA1
1: x <- input
2: s <- 0
3: c <- 0
4: b <- c < x
5: ifn b goto 9
6: s <- c + s
7: c <- c + 1
8: goto 4
9: _ret_r <- s
10: ret
```

- PA variables to JVM variables
  - input to 1,
  - x to 2,
  - s to 3,
  - c to 4
  - and b to 5
- PA labels to JVM labels
  - 4 to l1
  - 9 to l2

```
iload 1         // push the content of input to r0
istore 2        // pop r0's content to x,
sipush 0        // push the value 0 to r0
istore 3        // pop r0 to s
sipush 0        // push the value 0 to r0
istore 4        // pop r0 to c
ilabel l1       // mark label l1
iload 4         // push the content of c to r0
iload 2         // push the content of x to r1
if_icmpge l2    // if r0 >= r1 jump,
                // pop both r0 r1
iload 4         // push the content of c to r0
iload 3         // push the content of s to r1
iadd            // sum up r0 and r1 and result in r0
istore 3        // pop r0 to s
iload 4         // push the content of c to r0
sipush 1        // push a constant 1 to r1
iadd
istore 4        // pop r0 to c
igoto l1
ilabel l2
iload 3         // push the content of s to r0
ireturn
```

# Operational Semantics of JVM

$$\begin{array}{rrcl} (\texttt{JVM Program}) & J & \subseteq & jis \\ (\texttt{JVM Environment}) & \Delta & \subseteq & n \times c \\ (\texttt{JVM Stack}) & S & = & \_,\_ \mid c,\_ \mid c,c \end{array}$$

Small step operational semantics

$$J \vdash (\Delta, S, jis) \longrightarrow (\Delta', S', jis')$$

# Operational Semantics of JVM

$$\text{(sjLoad1)} \qquad J \vdash (\Delta, \_, \_, \mathit{iload}\ \ n; \mathit{jis}) \longrightarrow (\Delta, \Delta(n), \_, \mathit{jis})$$

$$\text{(sjLoad2)} \qquad J \vdash (\Delta, c, \_, \mathit{iload}\ \ n; \mathit{jis}) \longrightarrow (\Delta, c, \Delta(n), \mathit{jis})$$

$$\text{(sjPush1)} \qquad J \vdash (\Delta, \_, \_, \mathit{sipush}\ \ c; \mathit{jis}) \longrightarrow (\Delta, c, \_, \mathit{jis})$$

$$\text{(sjPush2)} \qquad J \vdash (\Delta, c_0, \_, \mathit{sipush}\ \ c_1; \mathit{jis}) \longrightarrow (\Delta, c_0, c_1, \mathit{jis})$$

$$\text{(sjLabel)} \qquad J \vdash (\Delta, r_0, r_1, \mathit{ilabel}\ \ l; \mathit{jis}) \longrightarrow (\Delta, r_0, r_1, \mathit{jis})$$

$$\text{(sjStore)} \qquad J \vdash (\Delta, c, \_, \mathit{istore}\ \ n; \mathit{jis}) \longrightarrow (\Delta \oplus (n, c), \_, \_, \mathit{jis})$$

$$\text{(sjAdd)} \qquad J \vdash (\Delta, c_0, c_1, \mathit{iadd}; \mathit{jis}) \longrightarrow (\Delta, c_0 + c_1, \_, \mathit{jis})$$

$$\text{(sjGoto)} \qquad J \vdash (\Delta, r_0, r_1, \mathit{igoto}\ \ l'; \mathit{jis}) \longrightarrow (\Delta, r_0, r_1, \mathit{codeAfterLabel}(J, l'))$$

$$\text{(sjCmpNE1)} \qquad \frac{c_0 \neq c_1 \qquad \mathit{jis}' = \mathit{codeAfterLabel}(J, l')}{J \vdash (\Delta, c_0, c_1, \mathit{if\_icmpne}\ \ l'; \mathit{jis}) \longrightarrow (\Delta, \_, \_, \mathit{jis}')}$$

$$\text{(sjCmpNE2)} \qquad \frac{c_0 = c_1}{J \vdash (\Delta, c_0, c_1, \mathit{if\_icmpne}\ \ l'; \mathit{jis}) \longrightarrow (\Delta, \_, \_, \mathit{jis})}$$

# Operational Semantics of JVM

$$
\begin{array}{rcl}
\mathit{codeAfterLabel}(\mathit{ireturn}, l) & = & \mathit{error} \\
\mathit{codeAfterLabel}(\mathit{ilabel}\ \ l; \mathit{jis}, l') & = & \begin{cases} \mathit{jis} & l == l' \\ \mathit{codeAfterLabel}(\mathit{jis}, l') & \texttt{otherwise} \end{cases} \\
\mathit{codeAfterLabel}(\mathit{ji}; \mathit{jis}, l) & = & \mathit{codeAfterLabel}(\mathit{jis}, l)
\end{array}
$$

# From PA to JVM

- $M$ - a mapping from PA temporary variables to JVM local variables.
- $L$ - a mapping from PA labels (which are used as the targets in some jump instructions) to JVM labels.
- We have three types of rules.
  - $M, L \vdash lis \Rightarrow jis$
  - $M \vdash s \Rightarrow jis$
  - $L \vdash l \Rightarrow jis$

# From PA to JVM

## Converting PA operands

$$(\texttt{jConst}) \quad M \vdash c \Rightarrow [\mathit{sipush}\ c]$$

$$(\texttt{jVar}) \quad M \vdash t \Rightarrow [\mathit{iload}\ M(t)]$$

## Converting PA Labels

$$(\texttt{jLabel1}) \quad \frac{l \notin L}{L \vdash l \Rightarrow []}$$

$$(\texttt{jLabel2}) \quad \frac{l \in L}{L \vdash l \Rightarrow [\mathit{ilabel}\ l]}$$

# From PA to JVM

$$(\texttt{jMove}) \quad \frac{L \vdash l \Rightarrow jis_0 \quad M \vdash s \Rightarrow jis_1 \quad M, L \vdash lis \Rightarrow jis_2}{M, L \vdash l : t \leftarrow s; lis \Rightarrow jis_0 + jis_1 + [istore\ M(t)] + jis_2}$$

```
// PA1
1: x <- input
2: s <- 0
3: c <- 0
...
```

- $M = \{(input, 1), (x, 2), (s, 3), (c, 4), (b, 5)\}$
- $L = \{(4, l1), (9, l2)\}$

```
iload 1       // push the content of input to r0
istore 2      // pop r0's content to x,
sipush 0      // push the value 0 to r0
istore 3      // pop r0 to s
sipush 0      // push the value 0 to r0
istore 4      // pop r0 to c
...
```

# From PA to JVM

$$(\texttt{jEq})\quad \frac{L \vdash l_1 \Rightarrow jis_0 \quad M \vdash s_1 \Rightarrow jis_1 \quad M \vdash s_2 \Rightarrow jis_2 \quad M, L \vdash lis \Rightarrow jis_3}{M, L \vdash l_1 : t \leftarrow s_1 == s_2; l_2 : ifn\ t\ goto\ l_3; lis \Rightarrow jis_0 + jis_1 + jis_2 + [if\_icmpne\ L(l_3)] + jis_3}$$

$$(\texttt{jLThan})\quad \frac{L \vdash l_1 \Rightarrow jis_0 \quad M \vdash s_1 \Rightarrow jis_1 \quad M \vdash s_2 \Rightarrow jis_2 \quad M, L \vdash lis \Rightarrow jis_3}{M, L \vdash l_1 : t \leftarrow s_1 < s_2; l_2 : ifn\ t\ goto\ l_3; lis \Rightarrow jis_0 + jis_1 + jis_2 + [if\_icmpge\ L(l_3)] + jis_3}$$

```
// PA1
...
4: b <- c < x
5: ifn b goto 9
...
```

- $M = \{(input, 1), (x, 2), (s, 3), (c, 4), (b, 5)\}$
- $L = \{(4, l1), (9, l2)\}$

```
...
ilabel l1     // mark label l1
iload 4       // push the content of c to r0
iload 2       // push the content of x to r1
if_icmpge l2  // if r0 >= r1 jump,
              // pop both r0 r1

...
```

# From PA to JVM

$$(\texttt{jAdd})\ \frac{L \vdash l \Rightarrow jis_0 \quad M \vdash s_1 \Rightarrow jis_1 \quad M \vdash s_2 \Rightarrow jis_2 \quad M, L \vdash lis \Rightarrow jis_3}{M, L \vdash l : t \leftarrow s_1 + s_2; lis \Rightarrow jis_0 + jis_1 + jis_2 + [iadd, istore\ M(t)] + jis_3}$$

$$(\texttt{jSub})\ \frac{L \vdash l \Rightarrow jis_0 \quad M \vdash s_1 \Rightarrow jis_1 \quad M \vdash s_2 \Rightarrow jis_2 \quad M, L \vdash lis \Rightarrow jis_3}{M, L \vdash l : t \leftarrow s_1 - s_2; lis \Rightarrow jis_0 + jis_1 + jis_2 + [isub, istore\ M(t)] + jis_3}$$

$$(\texttt{jMul})\ \frac{L \vdash l \Rightarrow jis_0 \quad M \vdash s_1 \Rightarrow jis_1 \quad M \vdash s_2 \Rightarrow jis_2 \quad M, L \vdash lis \Rightarrow jis_3}{M, L \vdash l : t \leftarrow s_1 * s_2; lis \Rightarrow jis_0 + jis_1 + jis_2 + [imul, istore\ M(t)] + jis_3}$$

```
// PA1
...
6: s <- c + s
7: c <- c + 1
...
```

- $M = \{(input, 1), (x, 2), (s, 3), (c, 4), (b, 5)\}$
- $L = \{(4, l1), (9, l2)\}$

```
...
iload 4       // push the content of c to r0
iload 3       // push the content of s to r1
iadd          // sum up r0 and r1 and result in r0
istore 3      // pop r0 to s
iload 4       // push the content of c to r0
sipush 1      // push a constant 1 to r1
iadd
istore 4      // pop r0 to c
...
```

# From PA to JVM

$$(\texttt{jGoto})\quad \frac{L \vdash l_1 \Rightarrow jis_0 \quad M, L \vdash lis \Rightarrow jis_1}{M, L \vdash l_1 : goto\ l_2; lis \Rightarrow jis_0 + [igoto\ l_2] + jis_1}$$

$$(\texttt{jReturn})\quad \frac{L \vdash l_1 \Rightarrow jis_0 \quad M \vdash s \Rightarrow jis_1}{M, L \vdash l_1 : rret \leftarrow s; l_2 : ret \Rightarrow jis_0 + jis_1 + [ireturn]}$$

```
// PA1
...
8: goto 4
9: _ret_r <- s
10: ret
```

- $M = \{(input, 1), (x, 2), (s, 3), (c, 4), (b, 5)\}$
- $L = \{(4, l1), (9, l2)\}$

```
...
igoto l1
ilabel l2
iload 3        // push the content of s to r0
ireturn
```

# Bytecode Optimization

SIMP

```
r = (1 + 2) * 3
```

PA

```
1: t <- 1 + 2
2: r <- t * 3
```

```
sipush 1
sipush 2
iadd
istore 2 // 2 is t
iload 2
sipush 3
imul
istore 3 // 3 is r
```

# Bytecode Optimization

SIMP

```
r = (1 + 2) * 3
```

PA

```
1: t <- 1 + 2
2: r <- t * 3
```

We could apply

1. Liveness analysis on PA level or
2. Generate JVM byte code directly from SIMP.
   - This requires the expression of SIMP assignment to be left nested.

```
sipush 1
sipush 2
iadd
sipush 3
imul
istore 3 // 3 is r
```