1 Introduction

This documentation will cover the design choices we have made and the challenges we have encountered for the code generation stage.

In the following sections, we will discuss *Labeling*, *Code Generation Order*, *Data Layout*, *Calling Convention* in detail.

2 Labeling

We use labels for static fields and methods, class and interface non-static methods, constructors, virtual table, SIT, and Subtype Table. We have the following conventions:

For static field and methods we use:
${className}#Static#Field#${fieldName}
${className}#Static#Method#${methodName}#${params.mkString("#")}

For class non static fields we use:
Class#${className}#Field#$fieldName

For class and interface non static methods we use:
Interface#${interfaceName}#Method#${methodName}#${params.mkString("#")}
Class#${className}#Method#${methodName}#${params.mkString("#")}

For constructors we use:
Class#${className}#Constructor#${params.mkString("#")}

For VTable, SIT and Subtype Table we use:
vtable#${className}
vtable#SIT#${className}
vtable#${className}#SubTypeTable

2.1 Special Labels

We also have built in some labels for special implementation purposes.

A primitive array vtable label will be assigned to all arrays during object creation as a pointer. This label is pointing to an array vtable that contains *java.lang.Object*'s method, because all arrays can be casted to a *java.lang.Object*. We call this label **vtable#Primitive#Array**

A native method call label is hardcoded for PrintStream's native write:
NATIVEjava.io.OutputStream.nativeWrite
This label will point to a native method call in the `runtime.s` file.

3 Code Generation Order

We generate our code in the following order:

1. Copy and Output `runtime.s` code
2. Generate all method labels and implementations
3. Generate all non-static field labels and implementations
4. Generate all constructor labels and implementations
5. Generate VTable, SIT and Subtype Table for all classes
6. Call the global `__start` function
7. Call and Implement all Static Fields
8. Call the static test method as the entry function
9. Call `__debexit`

4 Data Layout

The *Data Layout* for objects contain the same *Data Layout* for array types. This was determined half way through our implementation because of a special case — all arrays can be casted to *java.lang.Object*. Since array type has the same data layout as any class type, nothing needs to be modified when casting.

4.1 Object

Creating an Object of n fields will require $4 \times (n + 3)$ bytes in our structure.
The first four bytes of an object contains a pointer to its vtable. The second four bytes of an object contains 0 which is to match up with the data layout of an array type, more details will be discussed in the array section. The third four bytes is used to indicate that this object is not an array. During run-time, if there exists a cast expression to cast an *java.lang.Object* to an array, the first check will be made on the *java.lang.Object*'s length field, a -1 indicates a bad casting.

Fields are collected from all parent classes of the current class (interface does not have fields in JOOS1W and can be ignored), the collection process begins from *java.lang.Object* down to the current class, all fields with duplicated names will be overridden, the result is called as **Field Map**. It is guaranteed that for all parents of the current class, the order of fields will be the same, since any class can be extended to at most one class in Java.

When trying to get a field offset for the object, it will first add 3×4 bytes, and then use the **Field Map** to get the offset of the field.

VTable Pointer	`vtable_curclass`
Placeholder	`0`
length	`-1`

Field 1
...
Field n
	`Field 1 value`
	`...`
	`Field n value`

4.2 Array

An array has the same data layout as any other objects. With three main differences.

1. The first index of array contains a pointer to a *General Array Vtable*
2. The second index of array's value varies from primitive type array to a reference type array.
   For a primitive type array, its value is determined by its primitive type:
   - Primitive Boolean 1
   - Primitive Byte 2
   - Primitive Short 3
   - Primitive Char 4
   - Primitive Int 5

   For a reference type array, its value will be a pointer to its corresponding reference type's Vtable.

   When performing array access, casting or instance of expressions, a check will first be made on the **VTable Type Pointer** field to determine the **Type** of the array (Primitive Or Reference).

General Array VTable Pointer	`vtable_curclass`
VTable Type Pointer	`???`
length	`n`

Field 1
...
Field n
	`Field 1 value`
	`...`
	`Field n value`

4.3 *Java.Lang.String* Object

A *java.lang.String* Object is treated the same as any other objects. A *java.lang.String* is created for each **String Literals** in the program. We first allocate space and create a character array for the **String Literal**, next we allocate space to create **java.lang.String** object, its only field will contain the address of the newly created character array.

4.4 VTable

Our VTable contains a SIT pointer to the current class's SIT and a SubType Table pointer to the current class's SubType Table. Each class contains their own unique Vtable, SubType table and SIT. No interface should have Vtable, SubType table nor SIT because VTable is assigned during class instance creation and it is impossible to allocate and to create an interface object.

Implementation details of SIT and SubType Table will be listed in the following sections. All pointers are referenced by their unique label name.

For constructing the method implementations in the VTable, we first collect all methods signatures from the parent classese and interfaces of the current class into a map and call it as **Implementation Map**. Method overriding rules will be applied during the method collection process. Next, assign implementations pointers to the vtable follow from the **Implementation Map**'s entry order.

During compile time, the offset of the invoked method will be determined from the ordered entry in the **Implementation Map**.

VTable Label	`vtable_curclass`
Selector Index Table Pointer	`vtable_SIT_curClass`
Subtype Table Pointer	`vtable_SubtypeTable_curClass`
Method Signature 1	`Method 1 Implementation Pointer`
...	`...`
Method Signature n	`Method n Implementation Pointer`

4.4.1 Selector Index Table

Our Selector Index Table is defined in the following table. Whenever an interface type has invoked a method, the method implementation will be looked up from the **Selector Index Table**. We first collect all interface methods in the program into a list and call it as the **Method List**. Methods with same signatures will only occur once in the **Method List**. Next, we collect non-static method implementations for the current class type into a map and call it as the **Implementation Map**. This map maps a method signature to a type name that contains the implementation for the method signature. Method overriding rules apply for the method implementation collection process. We then loop through the **Method List**'s method signatures, if there exist signatures within the **Implementation Map**, then a pointer to the implementation will be placed at the corresponding index. If no implementation can be found for some method signatures in the **Method List**, then a pointer to an exception call will be placed. This index is determined by the order of method signatures in the **Method List** and will be same for all SIT tables.

During compilation time, the method signature's offset will be determined from the **Method List**

Selector Index Table Label	`vtable_SIT_curClass`
Method 1 Signature	`Method 1 Implementation Pointer`
Method 2 Signature	`Method 2 Implementation Pointer`
...	`...`
Method n Signature	`Method n Implementation Pointer`

4.4.2 SubType Table

Our SubType Table has the following structure, the subtype table will be used to perform the subtype check during cast and instance of expression during run time. This table is pre-built during compile time. We first collect all classes and interfaces to a list and call it as the **type list**. Then we collect all parent interfaces and classes for the current SubType Table's class. If the current class's parent and the current class exists in the **type list**, then a 1 will be placed into the corresponding index, else a 0 will be placed. The index is determined by the order of the class or interface name listed in the **parent list** and will be same for all SubType Tables.

During compilation time, the subtype's offset in the Subtype Table will be determined by the index of the subtype in the **type list**.

SubType Table Label	`vtable_SubtypeTable_curClass`
Interface Or Class Name 1	`0 or 1`
Interface Or Class Name 2	`0 or 1`
...	`...`
Interface Or Class Name n	`0 or 1`

5 Calling Convention

When calling a method with two parameters, we will have a structure as of the following stack after the method call:

this
param a
param b
eip
ebx
esi
edi
ebp
this
param a
param b

We first push the object's address as **This** onto the stack, followed by 2 parameters of the method. We will store the address pointing to **This** to a register **Register esi** prior to any pushes. After calling the method, **Register eip** will be pushed onto the stack automatically. We then store **Register ebx, esi, edi** as callee save registers. After that, we store the current **Register ebp** value.
Lastly, we copy down **This, param a, param b**. We can do that because we have been holding the address of **This** in the **Register esi**.

5.1 Local Variable Offset

We've encountered a design issue for this part and have modified our scope implementation as following:
Every local variable is given new scope. Every **local Variable Declaration Statement** and **For Statement** will be surrounded by a **Block Statement** till the end of its scope. Every scope will take a previously built scope as its parent. For example:

```
int a;
int b;
```

For the above 2 local variables, b's scope will have a's scope as a parent, we will also convert the code to:

```
{
int a;
{int b;}
}
```

After encountering a **Block Statement** during code generation, we allocate 4 bytes if it contains a variable inside, and we deallocate 4 bytes during the end of the **Block Statement**. When looking for offset of a local variable, it will first check if the local variable exists in the current scope. If yes, then it will add up all the variables in its parents' scope to get the offset. If the current scope does not have the local variable, then it will request the offset from its parent scope.

To make this work, we have one invariant condition:
When allocating variables, we must move all local variables after the **Register ebp** register.

6 Challenges and Implementations

6.1 String and Concatenation

String concatenation is made either when the *concat* method is explicitly called or when a string is involved in an addition with some other primitive types or another string. We've found that the *java.lang.String* contains 2 methods *concat* and *valueOf*. If there exists an addition between two strings, then we perform a method call on the *concat* method. If there exists an addition between a primitive type and a string, then we first convert the primitive type into a string through the *valueOf* method and then apply the *concat* method. This approach simplifies the code generation process.

In future, an improvement that can be made is to write a separate java file containing java helper function codes. These codes can be used as implementations for parts of the program that is difficult to write in assembly.

6.2 Abstract methods and interface methods

It is possible to have a vtable point to an abstract method during runtime. Our solution is to make use of the exception label in runtime.s. All abstract and interface methods will be given an implementation of *call __exception* by default.

6.3 Type of a Expression AST Node

In cases such as casting expressions, it will be convenient to know what type will be cast beforehand. We've added a new modification to our existing AST tree to store a **Type** for all expression AST nodes. This information is already computed during the **TypeChecking** stage, and therefore type information is extracted from there.

6.4 InstanceOf and Casting

The logic behind **InstanceOf** and **Casting** are similar, they will require a type check with the **subtype table**. Instead of throwing an error, **InstanceOf** will return a false if type is not an instance of another type.

7 Testing

The given public tests covered most of the basic situations, but missed some cases. So we created several custom test cases regarding the following cases:

- **CastExpression** between **PrimitiveType**, **ClassType**, **InterfaceType**, and **ArrayType** (Subtype Matrix)
- **InstanceofExpression** between **PrimitiveType**, **ClassType**, **InterfaceType**, and **ArrayType** (Subtype Matrix)
- Type checking in **AssignmentExpression** when the lhs is an **ArrayAccessExpression**
- When the static type of lhs of **MethodInvocationExpression** is a **ClassType** (vtable) and a **InterfaceType** (SIT)
- Bounds check in **ArrayAccessExpression**
- Code generation for **ConstructorDecl**