# Compiler Project
# PA4 – Code Generation

Due: 2024-04-17 11:59pm

The final programming checkpoint requires generating code that targets x86/x64 processors and the Linux operating system. In PA4, a miniJava program that passes syntactic and contextual analysis should be able to generate an executable binary with appropriate file headers.

The code generator can be implemented as an AST visitor with a single traversal. These assignment instructions will assume you will be using the starter code and doing code generation through the AST visitor.

1. Helpful Resources

https://defuse.ca/online-x86-assembler.htm#disassembly2

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

https://www.felixcloutier.com/x86/

http://ref.x86asm.net/coder64.html / http://ref.x86asm.net/coder64-abc.html

http://www.sunshine2k.de/coding/javascript/onlineelfviewer/onlineelfviewer.html

https://en.wikipedia.org/wiki/Executable_and_Linkable_Format

These resources will help ensure that your generated code is accurate. The difficulty of PA4 largely comes from proper planning and difficulty in testing. As is the case when working with hardware, getting feedback on code correctness is not easy. To prevent headaches, make sure to read the documentation thoroughly.

2. New miniJava language requirements.
   a. Every program needs exactly one `public static void main` method with a single `String[]` parameter. If the main method is not correctly provided in the input source code, then that is a code generation error.

   b. Ensure that every non-`void` method's last Statement is a return statement. Do not overcomplicate this check by checking block statements and validating code paths.

3. PA4 Starter code is available on the course website/GitHub repo.

Several files will give you an error or are marked with TODOs. Such files need to be completed in order for proper functionality.

Inside CodeGeneration.x64.ISA are common instructions that will help complete the assignment. Consider "Push.java": It is missing the constructor for "push imm32" where a 32-bit immediate is pushed on the stack. Use the resources given on the first page to find the opcode for pushing a 32-bit immediate on the stack. Additionally, look at other instruction examples to see how immediate bytes are written for an x64 instruction.

You should spend some time reading the starter code to see how x64 code is formed.

4. Quick overview of registers

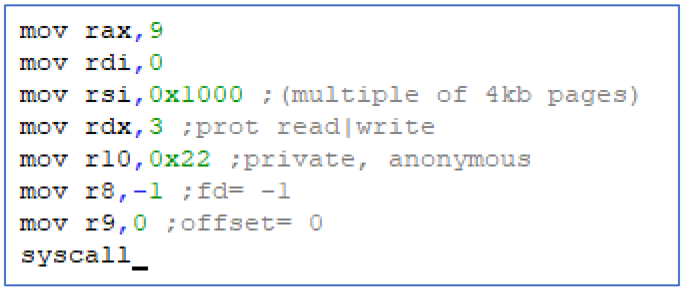
**RAX, RCX, RDX, RBX**: general purpose registers. Often, RCX refers to counters, and RAX is an accumulator, but registers can be used for anything.

**RSP, RBP**: Respectively, your stack pointer and stack base pointer. RSP is set before the program can execute. You can use these as general-purpose registers, but it would not be advisable.

**RSI, RDI**: Respectively, a source and destination index. These can be used just like any other general-purpose register.

**R8-R15**: Similar to the above, but can only be accessed by setting the appropriate REX prefix byte.

System calls require setting some registers to specific values.
For example, the "mmap" system call is given to you:



```
mov rax,9
mov rdi,0
mov rsi,0x1000 ;(multiple of 4kb pages)
mov rdx,3 ;prot read|write
mov r10,0x22 ;private, anonymous
mov r8,-1 ;fd= -1
mov r9,0 ;offset= 0
syscall
```

You can use this system call to create usable memory on the heap. Don't worry about cleaning up heap data when no longer referenced.

5. ModRMSIB

The operands of an instruction are specified by an instance of this class. You need to complete the "Make" methods in this class. Recall that you can have memory-related instructions in the "rm" operand, and plain registers appear in the "r" operand.

Register encoding table: http://ref.x86asm.net/coder64.html#modrm_byte_32_64

SIB encoding table: http://ref.x86asm.net/coder64.html#sib_byte_32_64

**You need to be able to read such tables** and decipher how to properly encode data using them. They are compact and not always intuitive, but treat it as a challenge. In fact, most documentation is often troublesome to read, so consider this good practice.

To get a register's encoding index, use `getIdx(registerObject)`. Implement:

"`rm, r`": For two plain registers, e.g., "`add rcx, rdx`." This is done for you.

"`[rdisp+disp],r`": The "rm" operand dereferences the memory pointed to by the register "rdisp" and with some displacement "disp". E.g., "`[rsi+3]`" will provide an operand that dereferences the memory located at `rsi+3`. Whether it is for writing or reading is determined by the instruction opcode.

"`[ridx*mult+disp],r`": This is similar to the above example except there is no displacement register. E.g., "`[rcx*4+0x80000000]`" might reference the "`rcx`-th" 4-byte integer offset from some starting memory address.

"`[rdisp+ridx*mult+disp],r`": This is a combination of the previous two.

The index register is useful for dereferencing arrays. For example, if you decide that integers are 4-byte values, then if `rcx` contains the evaluated index expression, then `rbx+rcx*4` would contain the location of the `rcx`-th integer in an array starting at `rbx`.

Note there are some intricacies. For example, the index register cannot be `RSP`. Given that `RSP` has a special purpose for manipulating the stack, it is not likely that `RSP` is ever used as an index anyway. You have two ways of dealing with the intricacies in the reference ModRM/SIB tables: detect them as an error, or never use such combinations when generating x64 code. Either works, but the latter may save you some time if you accidentally use such a combination.

## 6. Runtime Entities

Once you are done with ModRMSIB and completing any unfinished instructions, you can move on to code generation by traversing your program's AST.

The starter code targets the more difficult "position-independent bytecode" format for executable binaries. This means that we will not know where other segments like ".bss" are located. This makes it difficult, consider:

```
static int a; // Store at .bss+0
static int b; // Store at .bss+4
static SomeClass someObj; // Store at .bss+12
```

In the above code, we store uninitialized static data by assigning some offsets in the ".bss" section. However, we do not know what ".bss" resolves to. As such, for PA4, it is suggested that you create stack space for each static variable. This stack space can be created at your program's entrypoint before the actual main method code.

In miniJava, an int is 4 bytes long, and any object pointer is 8 bytes. For object pointers allocated by mmap, you will need all 64 bits. Stack space is limited, so dynamically allocating objects on the stack will result in many crashes. Thus, the heap must be utilized. Booleans can be allocated however few bytes you wish, but 1 byte is ideal.

Local variables should be created on the stack as an offset from RBP. This means the runtime entity for some local variable will be "RBP-X" for some X. You can make sure this stack space is not utilized by pushing some placeholder memory when you encounter a VarDecl, then storing some data in the VarDecl AST about where it would be stored as an offset from RBP or RSP. This offset describes how such a runtime entity can be referred to.

You will want to assign non-static field variables some offset from a base pointer. For example, if the non-static variables `x`, `b`, and `c` are in some class, an acceptable runtime entity assignment would be "object base`+0`", "`+8`", and "`+16`" for `x`, `b`, and `c`, respectively (Note that this assumes each variable is 8 bytes long). To resolve such variables, one strategy would be:

I. Local variable "`A a`"
II. Allocate memory for "`new A`", and store via "`a = new A`".
 Call your mmap function, and the result is in `rax`.
III. Assume `a`'s runtime entity is `rbp-4`. Then, `mov [rbp-4],rax`.
IV. To access the variable "`b`", we would need `mov rbx,[rbp-4]`.
V. Then, `[rbx+8]` would refer to the location of "`a.b`".

In the above example, "`a.b`" qualified reference is first resolved by getting the "object base" in step IV, where "`a`" is the object base. Then the runtime entity "`a.b`" can resolve as "object base+offset", where in this case, it would be `+16`. As such, a qualified reference's right-hand-side runtime entity is resolved in the context of the left-hand-side reference.

In summary, static variables are offset from some initial location on the stack. Local variables are offset from the stack frame's base pointer (or stack pointer). Instance variables are offset from some base pointer loadable from elsewhere.

7. Call / Return

When you call a method, you should create a new stack frame for local variables to be offset from. When you return, you should clean up the stack so that you do not overutilize the stack. There is a heavy reduction in points if you do not clean up the stack after a function returns. This is something you have learned in other classes but covered here briefly.

The calling convention for static methods is to push evaluated expressions in reverse order. This means that `SomeClass.MyStaticMethod( a+b, c );` would push "`c`" onto the stack first, then the result of "`a+b`" second. Then, inside the method, you would refer to parameter variables offset from your current stack frame (e.g., `rbp+8`).

Non-static methods have a hidden extra parameter, "`this`." This means that `a.b(x)` is actually: `b(a,x)`, where "`a`" becomes "`this`" for the method body. Similarly, `c(x)` would actually be `c(this,x)`.

Do not forget to clean up "this" on the stack when returning. Cleanup of the stack entails reclaiming rsp by increasing its value.

You should be generating code sequentially. This means that when you begin a method, then _asm.size() will contain the address of the next instruction. When calling a method, you might not know where that method is located in the assembly. As such, you should keep track of such "Unknown Runtime Entities" and patch them when you know the start address of a method whose code has not yet been generated.

You can decorate your AST with a patch list for each MethodDecl such that when that MethodDecl is visited, you can patch instructions that call that method.

Additionally, you need to keep track of where your main method is (and initialization code if any) so that you can create the ELF file appropriately later. This entrypoint is an offset from the start of the code (offset from zero), not the virtual address it will be loaded at.

8. System.out.println

This method takes an int parameter. You will need to write the assembly for the SYS_write system call (RAX=1). The output requires a null-terminated string of characters. You need to output the raw integer passed as a parameter. Note, this means that 48 = the printable number '0'. You do not need to do any fancy conversion from integers to ascii characters. Just make sure there exists a null terminator after whatever bytes you output. The autograder will not use values above 0xFF.

You will need to research how the write system call works. Use the mmap example.

9. Instruction Patching

After an Instruction has been added to the InstructionList, the listIdx field is filled. You can use this field to replace this instruction later. For example, in an IfStmt, you do not necessarily know where to jump to. Instead, create a placeholder ("jle 0" with new CondJmp(Condition.LE,0)). Then, when you know where to jump to, you can call _asm.patch with the proper listIdx of the conditional jump, and replace it with the proper value. Make sure you patch instructions with new instructions of the same size. An example is given in the comments in the starter code.

## 10. ELF Generation

There are three sections worth noting: .`text`, .`bss`, null, and .`shstrtab`. Set the `sh_type` and `sh_flags` for these sections appropriately. Similarly, set the `p_type` and `p_flags` for the program header segment and text segment. Review this code, it may come up.

You should output an "a.out" file (as shown in the starter code) if there were no other errors. If you do not output an "a.out" file, that implies your compiler thinks there was an error.

Unlike earlier assignments, this indicates whether the input source code passes or not. There is no need to output "Error" or "Success," and you may output whatever you like.

## 11. Autograder Information

Because the challenge of PA4 lies in your ability to research topics and implement what you find, the autograder is also done differently. Some simple autograder tests are made available for you, but you should test your solution more thoroughly to make sure your implementation is correct.

First, check to make sure your ELF is being read correctly by using the resources listed on the first page. Second, you can copy the raw data from your "a.out" file and check to make sure the bytecode corresponds to what assembly you were expecting. If this proves challenging, try outputting the bytecode to console in your compiler, and then pasting it into the website disassembler.

To test Linux x86_64 binaries, you can ssh into the following server:

```
ssh comp520@home.digital-haze.org -p 52025
```

The password is: comp520
Create a folder for yourself (`mkdir myonyen && cd myonyen`). Then, on subsequent connections, just `cd myonyen` to enter that directory.

You have access to the `objdump`, `readelf`, and `gcc/g++` if you need to test your output.

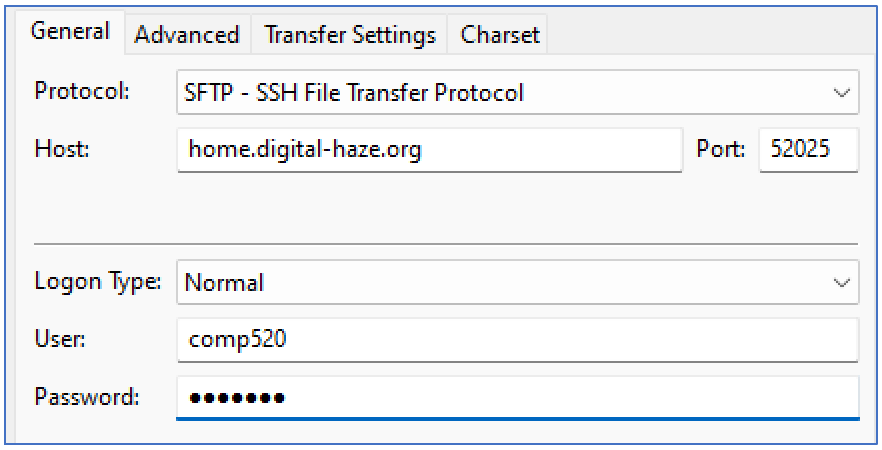
First, run your code locally:

```
java -cp bin miniJava.Compiler /your/test/file.java
```

Then, when you have generated the `a.out` file, upload it to your folder on the test server, then make sure to "`chmod u+x a.out`" then "`./a.out`" to make sure it outputs the expected output. DO NOT UPLOAD YOUR COMPILER SOURCE CODE.

To upload your files, you can use FileZilla for an easier drag-and-drop interface, but you will still need to ssh to actually execute the file.

In FileZilla, go to your server manager, and add the following:



Protocol: SFTP - SSH File Transfer Protocol
Host: home.digital-haze.org Port: 52025
Logon Type: Normal
User: comp520
Password: •••••••

Then you can drag your "a.out" file into your onyen folder, chmod and run it in ssh.

Consider what your compiler is capable of (arrays, objects, static variables, etc.) and create test java input files. Ensure that any output from System.out.println matches what you were expecting. Additionally, consult a hex-ascii table to make sure your output is actually readable. For example, System.out.println(10) is just a line break, and will not output the number "10." Instead, System.out.println(49) and System.out.println(48) will output the letters '10'.

Submit your completed project to the autograder. Ensure only relevant .java files are uploaded.