COMP 520 - Compilers

Lecture 15 – PA4 Details

Midterm 2 Postponed to 4/11

- WA3 is out
  - Quick questions on REX bit flags, stack framing, and x64
- PA3 – If you haven't submitted yet, make sure to submit to Partial only. Your first submission to the "Full" tests will be your grade, minus a late penalty.

Midterm 2

- Cumulative
- Know your:
  - Identification
  - Type-Checking
  - Visitor Traversal
  - Assembly Generation

PA4 – A Unique Opportunity

- Have the ability to claim that you wrote a REAL compiler.

PA4 – A Unique Opportunity

- Have the ability to claim that you wrote a REAL compiler.

- ***Starter code*** does the mundane parts of code generation (make sure bytes are in the right order, make sure you can set the REX prefix after-the-fact).

Course Registration Open

- For our seniors: Show them what UNC students are capable of. You wrote an x64 compiler, a feat not many are capable of.

Course Registration Open

- For our seniors: Show them what UNC students are capable of. You wrote an x64 compiler, a feat not many are capable of.

- For those who will continue with their coursework: if you didn't let this class beat you, **there shouldn't be any other class that can.** Treat each as a challenge.

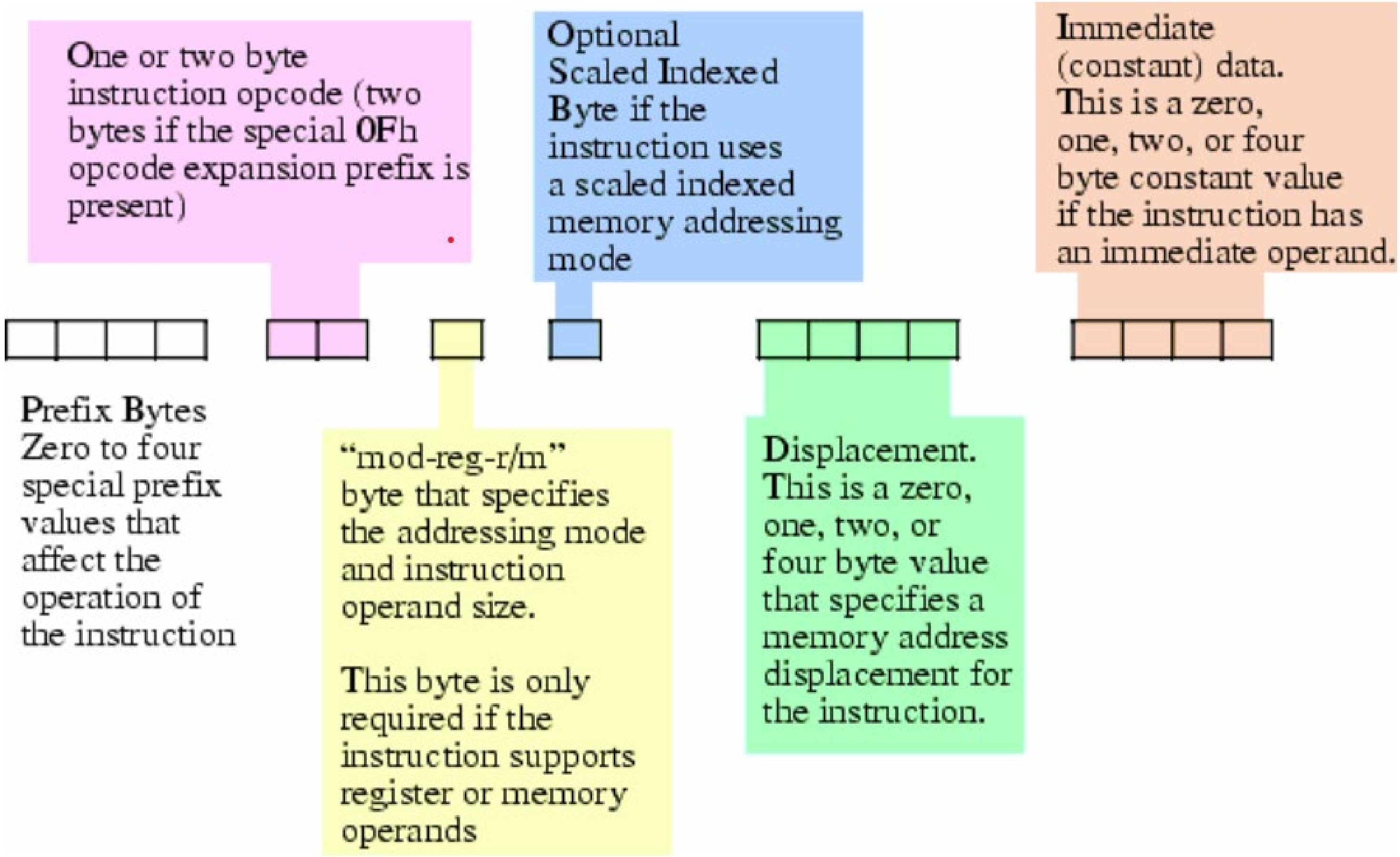
Highly Suggested Courses

1. **COMP 750- Grad Algorithms.** An essential course if you want to call yourself a computer scientist. Difficult, but you will look back on it fondly.

2. **COMP 630/541.** Get to write an OS / get to construct a processor, nearly from scratch.

Let’s get coding!

Instruction Encoding (c-jump.com)

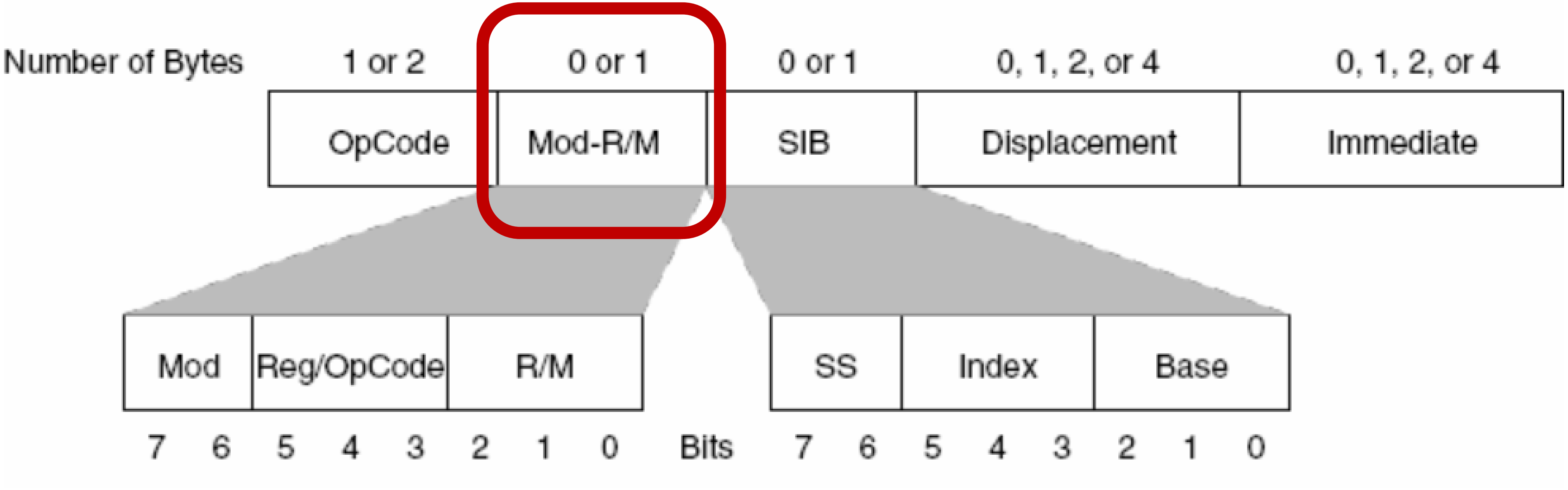


Quick Review of ModRM/SIB

- ModRM: http://ref.x86asm.net/coder64.html#modrm_byte_32_64
- SIB: http://ref.x86asm.net/coder64.html#sib_byte_32_64

Instruction Encoding (c-jump.com)

STEP ONE



Mod RM

- Used to allow operands: `rm, r` or `[rdisp+disp],r`
- Register operands are encoded in the ModRM byte.

Mod RM (2)

- Used to allow operands: `rm, r` or `[rdisp+disp],r`
- Register operands are encoded in this byte.

- **Do you have plain registers?**
  - Yes: mod=11
  - No: mod=??

Mod RM (3)

- Memory operation, like [rdisp+disp]
- **Do you have a zero displacement?**
  - Mod=00
- **Do you have a 1-byte displacement?**
  - Mod=01
- **4-byte displacement?**
  - Mod=10

01 010 110

Mod Value

“R” Index

“Rdisp” Index

01 010 110

Mod Value	"R" Index	"Rdisp" Index
=1 byte disp	**RDX=2**	**RSI=6**

Combined: [rsi+XX], rdx

01 010 110

Mod Value
=1 byte disp

"R" Index
RDX=2

"Rdisp" Index
RSI=6

Combined: `[rsi+XXYYZZWW], rdx`

10 010 110

Mod Value
=4 byte disp

“R” Index
RDX=2

“Rdisp” Index
RSI=6

ModRM

- When **mod**$\neq$11, is the displacement register RSP?

- Then you are forced to output an **SIB**, even if you don’t have an index.

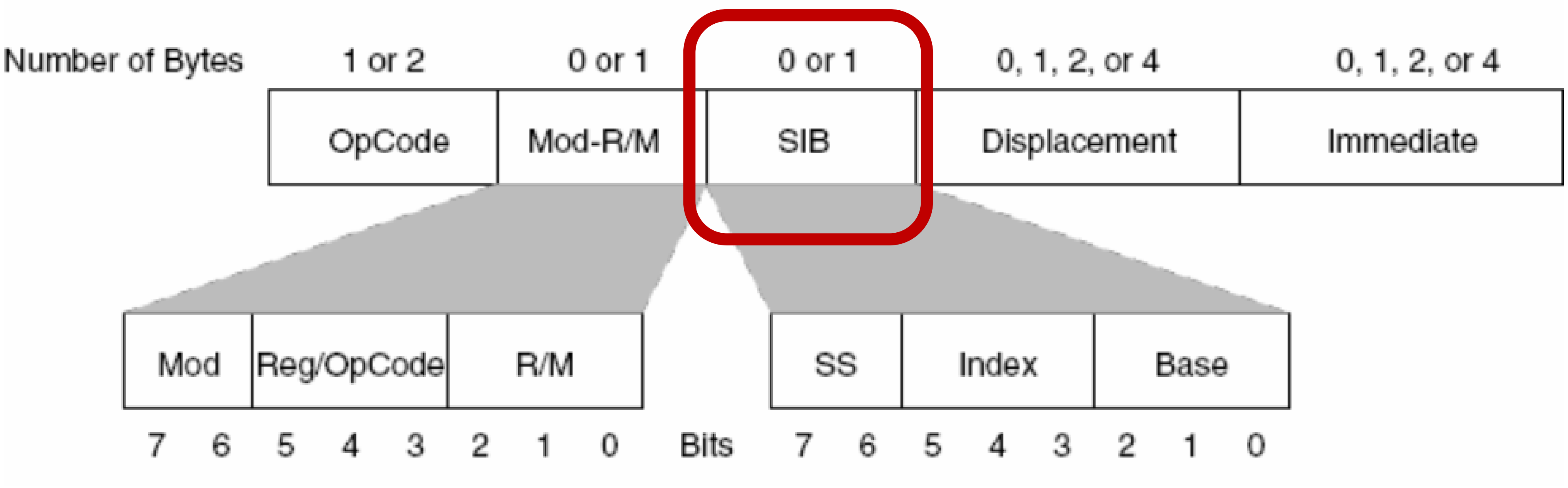
ModRM

- When **mod**$\neq$11, is the displacement register RSP?

- Then you are forced to output an **SIB**, even if you don't have an index.
- Note: there is an entry in the SIB table to just ignore the index anyway.

Instruction Encoding (c-jump.com)

STEP TWO



Scaled Index Byte: ridx*mult

- Is your index multiplier x1, x2, x4, x8?
  - Then, **ss**= 00, 01, 10, 11 respectively

- Note, for SIB, a displacement register has not yet been picked.
- SIB forces the ModRM "**rdisp**" register to be **RSP**

Combined: `[rsi+rcx*2+XX],r`

01 001 110

SS Value
=multiply by 2

"RIDX" Index
RCX=1

"RDISP" Index
RSI=6

Take care for exceptions

- Note, RSP cannot be used as an index register.
- Question: what other intricacies are there, and why?

Take care for exceptions

- Note, RSP cannot be used as an index register.
- Question: what other intricacies are there, and why?

- If getting stuck, it may be useful to draw out a decision tree.

PA4 x64 Code Generation

Step 1: Finish ModRMSIB Class

- There is starter code on the course website.
- The ModRMSIB class decides which registers are used in an instruction (it doesn't care about order, but just the operands themselves).

Step 1: Finish ModRMSIB Class

- There is starter code on the course website.
- The ModRMSIB class decides which registers are used in an instruction (it doesn't care about order, but just the operands themselves).

- The "Make" methods need to be completed. An example for "rm,r" is done.

ModRMSIB Strategy

- Test instances of this class.
- Output the bytes, and make sure your implementation matches what is on the table.

Make Methods: rm, r

```
// rm,r
private void Make(Reg rm, Reg r) {
        int mod = 3;

        int regByte = ( mod << 6 ) | ( getIdx(r) << 3 ) | getIdx(rm);
        _b.write( regByte );
}
```

Make: [rdisp+disp],r

Implement this inside:

```
private void Make(
            Reg64 rdisp, int disp, Reg r
    )
```

Make: [ridx*mult+disp],r

Implement this inside:

```
private void Make(
        Reg64 ridx, int mult, int disp, Reg r
    )
```

Make: [rdisp+ridx*mult+disp],r

Implement this inside:

```
private void Make(
 Reg64 rdisp, Reg64 ridx, int mult, int disp, Reg r
    )
```

Once you have encoded the table in these methods…

- You are done with the most difficult part of encoding assembly into bytecode.

- The operands are the hardest part, and the rest is material you have seen before.

Once you have encoded the table in these methods…

- You are done with the most difficult part of encoding assembly into bytecode.

- The operands are the hardest part, and the rest is material you have seen before.
- Opcodes get some bytes, Prefixes get some bytes, and immediates get some bytes.

Unsure about how something is encoded?

- Check the tool: https://defuse.ca/online-x86-assembler.htm#disassembly2
- Test things such as "mov r11,[r10+r8*8+2222]" to make sure your implementation gets the correct ModRM and SIB bytes

Unsure about how something is encoded?

- Check the tool: https://defuse.ca/online-x86-assembler.htm#disassembly2
- Test things such as “mov r11,[r10+r8*8+2222]” to make sure your implementation gets the correct ModRM and SIB bytes

Disassembly:

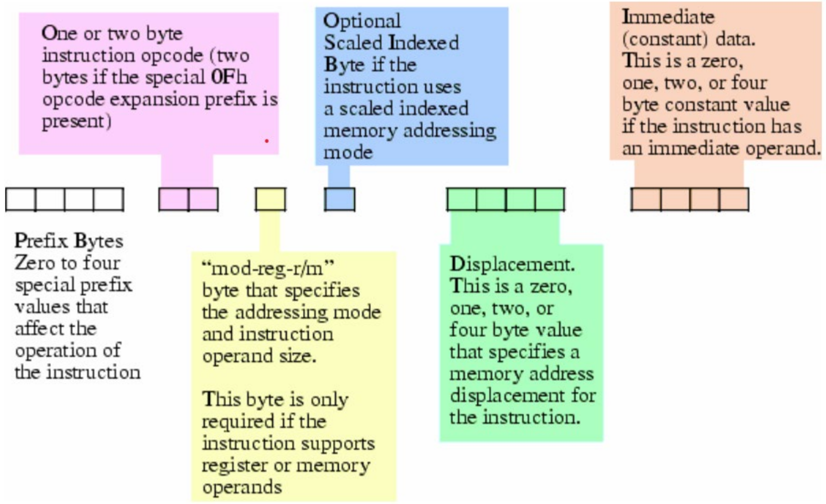
```
0:   4f 8b 9c c2 ae 08 00    mov    r11,QWORD PTR [r10+r8*8+0x8ae]
7:   00
```

Code Generation and Visitor Traversal

Instruction

- Instruction is an abstract class that has some associated bytecode with it.
- Includes prefix bytes, immediate bytes, …

All of these:



Instruction (2)

- The first instruction has a start address of zero.
- …
- The next instruction has a start address of the previous instruction start address + size.

Instruction List

- A list of instructions with a few extra items.
- If you add an Instruction to an InstructionList, it will populate the startAddress and listIdx fields.

- (Where listIdx is the index of the instruction, e.g., the 0^{th} instruction has a listIdx of 0).

Instruction Implementations

- Several files in the starter code are marked with TODOs.
- Find these and implement them just like with ModRMSIB.

Example: JMP Imm32

- Consider unconditional jump: jmp imm32

- In our instruction list, find it, opcode 0xE9.

```
// jmp imm32 (offset from next instruction)
public Jmp(int offset) {
        opcodeBytes.write(0xE9);
        x64.writeInt(immBytes,offset);
}
```

Example: JMP Imm32

- Consider unconditional jump: jmp imm32

- In our instruction list, find it, opcode 0xE9.
- Then implement it:
- Take notes: the imm32 is an offset from the start of the next ins.

```
// jmp imm32 (offset from next instruction)
public Jmp(int offset) {
        opcodeBytes.write(0xE9);
        x64.writeInt(immBytes,offset);
}
```

Read other implementations as well!

- IDIV: Divide `RDX:RAX` / rm
- `RDX`: 00000001, `RAX`: 00000001

Read other implementations as well!

- **IDIV**: Divide `RDX:RAX` / rm
- `RDX`: 00000001, `RAX`: 00000001
- Thus, the operand becomes:
    - **4294967297** / rm

- E.g.: 4294967297 / [rbp-16]

Operands created with ModRMSIB

- How can we actually generate x64?

Operands created with ModRMSIB

```
new Push(new
        ModRMSIB(Reg64.RBP,16)
        );
```

Analogous to:

```
push [rbp+16]
```

Operands created with ModRMSIB

```
new Push(new
        ModRMSIB(Reg64.RBP,16)
        );
```

Analogous to:

```
push [rbp+16]
```

```
new Mov_rrm(new ModRMSIB(
  Reg64.RSI,Reg64.RCX,4,0x1000,Reg64.RDX
        );
```

```
mov rdx, [rsi+rcx*4+0x1000]
```

Operands created with ModRMSIB

```
new Push(new
        ModRMSIB(Reg64.RBP,16)
        );
```

Analogous to:

```
push [rbp+16]
```

R operand, then RM operand (so move from memory to register)

```
new Mov_rrm(new ModRMSIB(
  Reg64.RSI,Reg64.RCX,4,0x1000,Reg64.RDX
        );
```

```
mov rdx, [rsi+rcx*4+0x1000]
```

mov rm,r

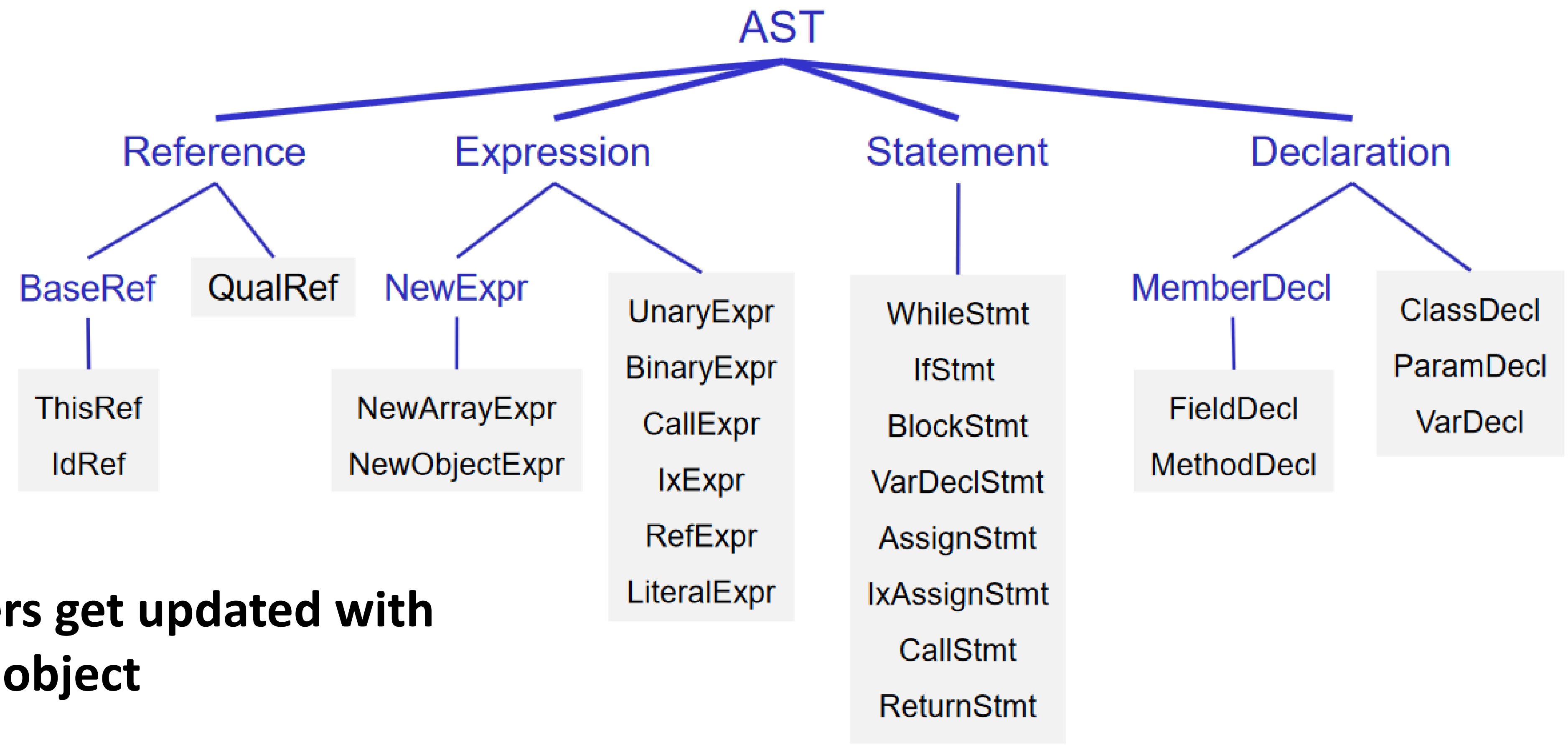
Don’t always need to dereference memory locations.

Mov_rmr can also take a plain:

new **ModRMSIB**(Reg64.RCX, Reg64.RDX)

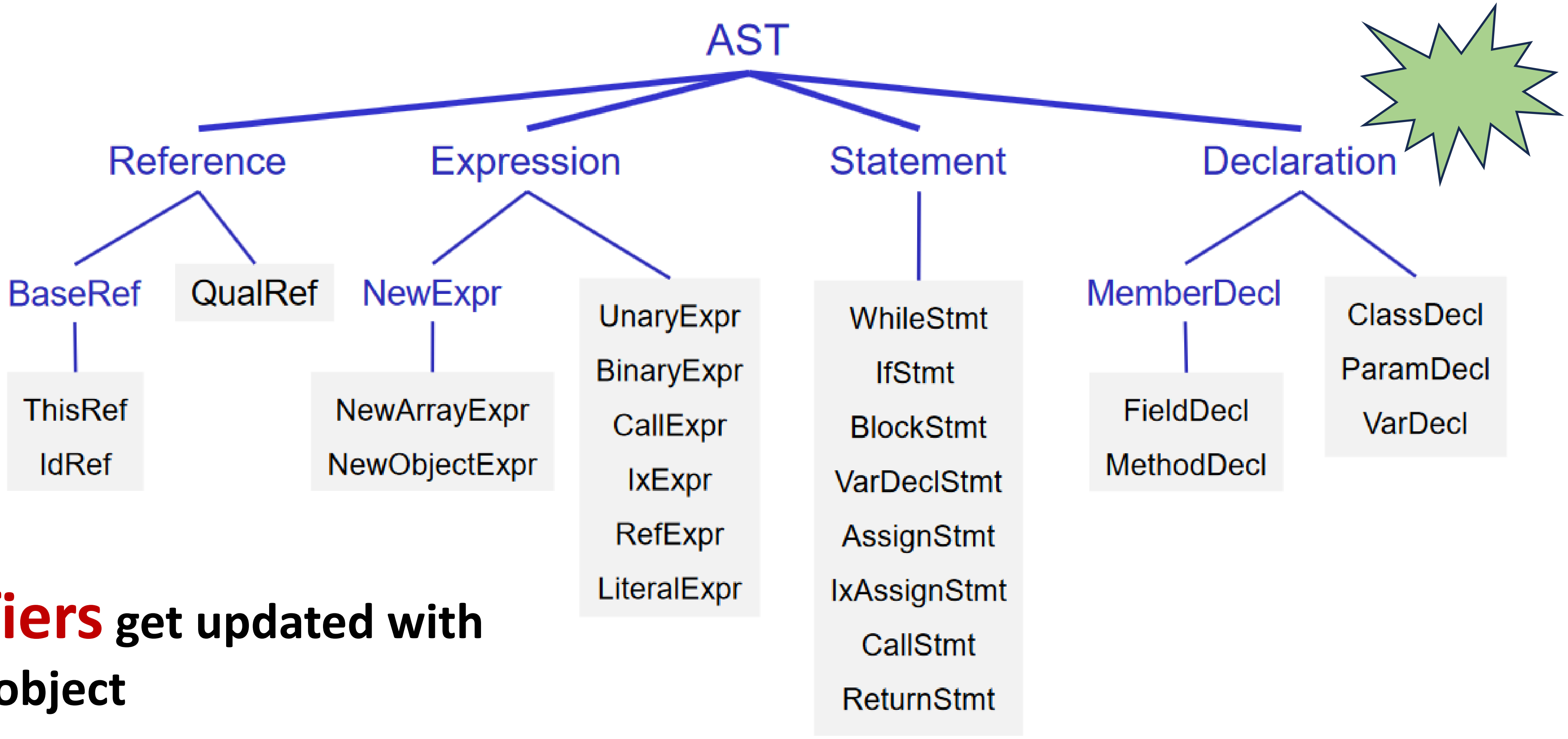
Results in: **mov rcx,rdx**

AST Decoration



Identifiers get updated with a “decl” object

This is now a decorated AST



Identifiers get updated with a “decl” object

We need additional decorations

- Where is a variable in memory?
- Let’s store that in the AST!

- We will need a point(er) of reference, and map out our memory appropriately.
- We call this data the “RuntimeEntity” object, for each variable.

Runtime Entity

LocalDecl	Type	Reference	Offset
int x = 3;	int ≡ 4 bytes	rbp	-8
int y = x + 7;	int ≡ 4 bytes	rbp	-16 ???

Stack

- Push and Pop always operate 64-bits at a time
- This is because we are in 64-bit mode (long mode)
- Thus, storing data on the stack will always be 8 bytes long.

Runtime Entity

LocalDecl	Type	Reference	Offset	Code
int x = 3;	int ≡ 4 bytes	rbp	-8	push 0 ??
int y = x + 7;	int ≡ 4 bytes	rbp	-16	push 0 ??
SomeObj A = new A();	ClassType ≡ 8 bytes	rbp	-24	push 0 ??

Push

- We will create space on the stack by initializing data to zero.

- Push will decrement RSP by 8, THEN store the data being pushed
- After evaluating the expression, store something in the appropriate location.

Static Variables

- We are targeting position-independent code
- This means that .bss is not easy to resolve.
- In PA5- optional extra credit to properly implement .bss

- **For PA4- where can we store static data? (Note, if it stays on the stack, doesn't have to be 8 bytes)**

What about recursion?

- Consider:

```
class A {
    private void fn(int x) {
        int y = 48 + x;
        if( x > 0 ) {
            fn(x-1);
            System.out.println(y);
        }
    }
}
```

Every call of this function needs a separate location for the local variable y and parameter x

Stack Frame

- Every function call, move up your point of reference (which can be RBP or RSP).

Stack Frame

- Every function call, move up your point of reference (which can be RBP or RSP).
- Consider a static method: fn(x)

x	[return addr]	??	??	??
RSP+8	↑ RSP When method starts	RSP-8	RSP-16	

Stack Frame

- Consider after local variables declared:

New Local about to be declared

x	[return addr]	*Old RBP*	Local var1	??
RBP+16	RBP+8	RBP	RBP-8	RBP-16
RSP+24	**RSP+16**	**RSP+8**	**RSP**	**RSP-8**

Stack Frame

- Consider after local variables declared:
- Thus, easier to reference data from **rbp** (assuming you set up your stack frame correctly)

x	[return addr]	*Old RBP*	Local var1	Local var2
RBP+16	RBP+8	RBP	RBP-8	RBP-16
RSP+32	**RSP+24**	**RSP+16**	**RSP+8**	**RSP**

Member Variables (FieldDecl)

- Where are field variables?
- First, let's figure out the size of a class.

Member Variables (FieldDecl)

- Where are field variables?
- First, let's figure out the size of a class.

- Objects should be allocated on the heap. Meaning their data should not be on the stack.
- Instead, an object is just an 8 byte pointer

Map out the data

FieldDecl	Runtime Entity	Size	ASM
int x	Base + 0	**4**	**dword** [r+0]
int y	Base + 4	**4**	**dword** [r+4]
A z	Base + 8	8	qword [r+8]

How was the data allocated?

push 0 **call mmap** **mov [rbp-8],rax**	**A a = new a();**
??	**a.y = 3;**

How was the data allocated?

push 0 call mmap mov [rbp-8],rax	A a = new a();
mov rsi,[rbp-8] mov [rsi+4],3	a.y = 3;

FieldDecl

- Thus, the runtime entity for a FieldDecl is, once again, some offset

- The base address can be loaded during runtime (unlike being an offset from RBP)

What about this.x?

- What if we are in an instance method?
- Where is the “current object: this”?

Instance Methods

- SomeClass.myStaticMethod(x) – pretty normal
- a.nonStaticMethod(x,y) – slightly different

Stack Frame

- a.nonStaticMethod(x,y) – Hidden “this” parameter
- Will be used to find FieldDecl variables in the current object instance.

y	x	this	[return]	Old rbp	??
RBP+32	RBP+24	RBP+16	RBP+8	RBP	
RSP+32	**RSP+24**	**RSP+16**	**RSP+8**	**RSP**	

System.out.println

- Recall for PA4, miniJava is incomplete because String has not yet been implemented (optional for PA5)
- We take in a parameter, and need to output it on the screen

System Calls

- We will be using the SYS_write system call to output data to the console.

```
long sys_write(unsigned int fd, const char __user *buf,
               size_t count);
```

- Where fd=stdout=1

Read the mmap system call in starter files

- Find out how sys_write is called, it is very similar to sys_mmap

- Implement sys_write where the int parameter to System.out.println(int n) is the output byte.
- **The printable character ‘0’ is 48, take care for this when testing.**

System.out.println

- Outputs one byte (null terminated, so two bytes)
- Do not output a line break (can do this in PA5)

- System.out.println(53)
- System.out.println(50)
- System.out.println(48)
- ***What will this output?* Consult a hex-ascii table.**

Actual Code Generation Step

Case study: Flat Assembler

- A very simple assembler, outputs x86_64 bytecode
- It compiles the code 3 times (known as passes)

- Only then does it reach "optimization level zero"

Quick Note: Optimization Level

- Compilers can optimize your code by rewriting it efficiently (restructuring your ASTs is one method)

- -O0 (letter O, number zero) does no optimization

Quick Note: Optimization Level

- Compilers can optimize your code by rewriting it efficiently (restructuring your ASTs is one method)

- -O0 (letter O, number zero) does no optimization

- We do not require multiple passes to reach level 0, you can instead output inefficient “less than O0” code

Strategy: Stack-based evaluation

- Everything should be evaluated on the stack, and loaded into registers for only short code portions

Goal: Expressions

- Evaluate: 3+4*5-6

Goal: Expressions

- Evaluate: 3+4*5

- visitBinExpr: (visit LHS, visit RHS expressions)

Goal: Expressions

Stack
3

- Evaluate: 3+4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 3
  - RHS: visitBinExpr: ?

Goal: Expressions

Stack
5
4
3

- Evaluate: 4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 4
  - RHS: visitBinExpr: push 5

Goal: Expressions

Stack
5 -> X
4 -> X
3

- Evaluate: 4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 4
  - RHS: visitBinExpr: push 5
- Operator is multiply:
  - pop rcx, pop rax # Get two operands

Registers	
rax	4
rcx	5

Goal: Expressions

- Evaluate: 4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 4
  - RHS: visitBinExpr: push 5
- Operator is multiply:
  - pop rcx, pop rax # Get two operands
  - imul rcx # Multiply

Stack
-
-
-
3

Registers	
rax	20
rcx	5

Goal: Expressions

Stack
-
-
20
3

- Evaluate: 4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 4
  - RHS: visitBinExpr: push 5
- Operator is multiply:
  - pop rcx, pop rax # Get two operands
  - imul rcx # Multiply
  - push rax

Registers	
rax	20
rcx	5

Stack
-
-
20
3

Goal: Expressions

- Evaluate: 3+4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 3
  - RHS: visitBinExpr: (done earlier)

Goal: Expressions

Stack
-
-
20 -> X
3 -> X

- Evaluate: 3+4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 3
  - RHS: visitBinExpr: (done earlier)
- Operator is addition:
  - pop rcx, pop rax # get two operands

Registers	
rax	3
rcx	20

Goal: Expressions

- Evaluate: 3+4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 3
  - RHS: visitBinExpr: (done earlier)
- Operator is addition:
  - pop rcx, pop rax # get two operands
  - add rax, rcx # do the addition

Stack
-
-
-
-

Registers	
rax	23
rcx	20

Goal: Expressions

Stack
-
-
-
23

- Evaluate: 3+4*5
- visitBinExpr: (visit LHS, visit RHS expressions)
  - LHS: visitLiteralExpr: push 3
  - RHS: visitBinExpr: (done earlier)
- Operator is addition:
  - pop rcx, pop rax # get two operands
  - add rax, rcx # do the addition
  - push rax # store

Registers	
rax	3
rcx	20

Finally:

Stack
-
-
-
23 -> X

- int x = 3+4*5;
- 3+4*5: resolves to 23 on the stack
- Thus:

```
pop rax
mov [rbp-8],rax
```

- And our local variable now has a value!

Registers	
rax	23
rcx	20

int x = 3 + 4 * 5;

```
push 0 # create int x
push 3 # visitLiteralExpr
push 4 # visitLiteralExpr
push 5 # visitLiteralExpr
pop rcx
pop rax
imul rcx
push rax # do 4*5
```

int x = 3 + 4 * 5;

```
push 0 # create int x
push 3 # visitLiteralExpr
push 4 # visitLiteralExpr
push 5 # visitLiteralExpr
pop rcx
pop rax
imul rcx
push rax # do 4*5
```

```
pop rcx # load the evaluated 4*5
pop rax # load the earlier 3 (LHS)
add rax,rcx # do LHS+RHS
push rax # store result on stack
pop rax # get result
mov [rbp-8],rax # store in x
```

Is it really necessary?

```
push 4 # visitLiteralExpr
push 5 # visitLiteralExpr
pop rcx
pop rax
```

Why not: mov rcx,5
mov rax,4

That would actually be the second pass

- We will go more in-depth about optimization soon, but for now, unoptimized code is fine.

- Idea: “condense X pushes, and Y pops (where X=Y) into move operations”
  - Can wait until PA5

Visit Identifier?

Goal: 3 + y

Visit Identifier?

Goal: 3 + y

- push 3
- push [rbp-16] **# Push it on the stack**
- pop rcx
- pop rax
- add rax,rcx

Second Pass (Opcode size reduction)

- Recall cache slides from earlier
- Why do I want to reduce the size of my code?

Third Pass (instruction size reduction)

- Recall cache slides from earlier
- **In your first pass, you greedily pick the largest instructions for jumps and calls**
- Recall: jump and call is (mostly) relative from your current position

Third Pass (instruction size reduction)

- Recall: jump and call is (mostly) relative from your current position

- If we have to patch the instruction, then we want to patch it with the same instruction size.

Consider:

je 0 (1 byte offset) (Address=0)

```
if( x ) {
        …
}
```

Consider:

je 0 (1 byte offset) (Address=0)	if(x) {
A bunch of code is generated	…
(Address=0x30C)	}

Cannot jump to this address using a single byte!!

Consider:

je 0 **(4 byte offset)** (Address=0) if(x) {

A bunch of code is generated …

(Address=0x30C) }

Cannot jump to this address using a single byte!!

Correct solution: greedily pick 4 byte offset instructions because we do not know the offset yet!

Instruction Patching

Starter Code

- Recall: InstructionList populates the startAddress and listIdx data of any added Instruction object.
- With this listIdx, we can patch it later.

Consider:

```
if( y ) {
        …
        …
}
```

visitExpression()

_asm.add(new “cmp [rsp],0”);

Instruction jmp = new CondJmp(Cond.E, 0);

_asm.add(jmp);

ifStmt.thenStmt.visit(this); # Generate Code

Consider:

```
if( y ) {
        ...
        ...
    }
```

```
...
Instruction jmp = new CondJmp(Cond.E, 0);
_asm.add( jmp );
thenStmt.visit(...);
_asm.patch( jmp.listIdx,
    new CondJmp( Cond.E,
        ... Evaluate offset here ...
    )
);
```

Consider:

```
if( y ) {
        …
        …
    }
```

```
…
Instruction jmp = new CondJmp(Cond.E, 0);
_asm.add( jmp );
thenStmt.visit(…);
```

_asm.patch(jmp.listIdx,
new CondJmp(Cond.E,
jmp.startAddress, _asm.size()
)
);

Current Address

Old Address

This constructor will calculate the relative offset of these two addresses.

One more thing!

- Under the previous code, visiting a reference loads its value on the stack.

- But what about:

a.x = 3;

I need to load it's ADDRESS when visiting something on the left-hand-side of an assignment operation.

Resolving Assignment Destination

- First: QualRef is resolved exactly like you think it should be.

- a.b.c.x

"Load pointer a" -> "ClassA.b is offset +16"
-> "ClassB.c is offset +8"
-> "ClassC.x is offset +0"

Resolving QualRef Destination

- $a.b.c.x = ?$

"Load pointer a" -> "a.b is offset +16" -> "b.c is offset +8" -> "c.x is offset +0"

mov rax,[rbp-8] # Load local "a"

mov rax,[rax+16] # Load a.b pointer

Resolving Assignment Destination

- $a.b.c.x = ?$

"Load pointer a" -> "a.b is offset +16" -> "b.c is offset +8" -> "c.x is offset +0"

mov rax,[rbp-8] # Load local "a"

mov rax,[rax+16] # Load a.b pointer

mov rax,[rax+8] # Load b.c pointer

mov rax,[rax] # Load data at c.x ?????

Resolving QualRef Destination

- $a.b.c.x = ?$

"Load pointer a" -> "a.b is offset +16" -> "b.c is offset +8" -> "c.x is offset +0"

mov rax,[rbp-8] # Load local "a"

mov rax,[rax+16] # Load a.b pointer

mov rax,[rax+8] # Load b.c pointer

🤔**lea rax,[rax+0] # Load address at c.x**

Resolving QualRef Destination

- `visitExpression()` # 3 is on the stack
- `a.b.c.x = 3`

mov rax,[rbp-8] # Load local “a”

mov rax,[rax+16] # Load a.b pointer

mov rax,[rax+8] # Load b.c pointer

lea rax,[rax] # Load address at c.x

pop [rax] # Store top of stack at address rax

ELF Generation

ELF Generation

- Let’s be honest, this is not an interesting problem
- You already have seen ELF headers before.

ELF Generation

- Additionally: testing proper ELF files is a pain

- There is almost no way to tell if your ELF file is correctly generated without having a correctly generated file that can execute.

ELF Generation

- Additionally: testing proper ELF files is a pain

- There is almost no way to tell if your ELF file is correctly generated without having a correctly generated file that can execute.
- Lastly, this is just making sure “this byte goes to that spot, that byte here, that byte there”
  - You’ve already proven this capability
    (otherwise you wouldn’t be taking a 500 level class)

ELF Generation

- What IS important though: read through the starter code
- Fill out how segment and section flags are assigned

- That part is very important.

PIE

- What is a position-independent executable?

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

Section header tables
=====================

Nr	Name	Type	Address
0		SHT_NULL	0x0000000000000000
1	.interp	SHT_PROGBITS	0x0000000000000318
2	.note.gnu.property	SHT_NOTE	0x0000000000000338
3	.note.gnu.build-id	SHT_NOTE	0x0000000000000368
4	.note.ABI-tag	SHT_NOTE	0x000000000000038C
5	.gnu.hash	Unknown	0x00000000000003B0
6	.dynsym	SHT_DYNSYM	0x00000000000003D8
7	.dynstr	SHT_STRTAB	0x0000000000000480
8	.gnu.version	Unknown	0x000000000000051E
9	.gnu.version_r	Unknown	0x0000000000000530
10	.rela.dyn	SHT_RELA	0x0000000000000570
11	.rela.plt	SHT_RELA	0x0000000000000630
12	.init	SHT_PROGBITS	0x0000000000001000
13	.plt	SHT_PROGBITS	0x0000000000001020
14	.plt.got	SHT_PROGBITS	0x0000000000001040
15	.plt.sec	SHT_PROGBITS	0x0000000000001050
16	.text	SHT_PROGBITS	0x0000000000001060
17	.fini	SHT_PROGBITS	0x0000000000001184
18	.rodata	SHT_PROGBITS	0x0000000000002000
19	.eh_frame_hdr	SHT_PROGBITS	0x0000000000002004
20	.eh_frame	SHT_PROGBITS	0x0000000000002038
21	.init_array	SHT_INIT_ARRAY	0x0000000000003DB8
22	.fini_array	SHT_FINI_ARRAY	0x0000000000003DC0
23	.dynamic	SHT_DYNAMIC	0x0000000000003DC8
24	.got	SHT_PROGBITS	0x0000000000003FB8
25	.data	SHT_PROGBITS	0x0000000000004000
26	.bss	SHT_NOBITS	0x0000000000004010
27	.comment	SHT_PROGBITS	0x0000000000000000
28	.symtab	SHT_SYMTAB	0x0000000000000000
29	.strtab	SHT_STRTAB	0x0000000000000000
30	.shstrtab	SHT_STRTAB	0x0000000000000000

PIE

- Every section (and segment) is just sequentially laid out in the file.
- Linux realizes this, and loads sections at whatever memory address is convenient for it.

Section header tables
======================

Nr	Name	Type	Address
0		SHT_NULL	0x0000000000000000
1	.interp	SHT_PROGBITS	0x0000000000000318
2	.note.gnu.property	SHT_NOTE	0x0000000000000338
3	.note.gnu.build-id	SHT_NOTE	0x0000000000000368
4	.note.ABI-tag	SHT_NOTE	0x000000000000038C
5	.gnu.hash	Unknown	0x00000000000003B0
6	.dynsym	SHT_DYNSYM	0x00000000000003D8
7	.dynstr	SHT_STRTAB	0x0000000000000480
8	.gnu.version	Unknown	0x000000000000051E
9	.gnu.version_r	Unknown	0x0000000000000530
10	.rela.dyn	SHT_RELA	0x0000000000000570
11	.rela.plt	SHT_RELA	0x0000000000000630
12	.init	SHT_PROGBITS	0x0000000000001000
13	.plt	SHT_PROGBITS	0x0000000000001020
14	.plt.got	SHT_PROGBITS	0x0000000000001040
15	.plt.sec	SHT_PROGBITS	0x0000000000001050
16	.text	SHT_PROGBITS	0x0000000000001060
17	.fini	SHT_PROGBITS	0x0000000000001184
18	.rodata	SHT_PROGBITS	0x0000000000002000
19	.eh_frame_hdr	SHT_PROGBITS	0x0000000000002004
20	.eh_frame	SHT_PROGBITS	0x0000000000002038
21	.init_array	SHT_INIT_ARRAY	0x0000000000003DB8
22	.fini_array	SHT_FINI_ARRAY	0x0000000000003DC0
23	.dynamic	SHT_DYNAMIC	0x0000000000003DC8
24	.got	SHT_PROGBITS	0x0000000000003FB8
25	.data	SHT_PROGBITS	0x0000000000004000
26	.bss	SHT_NOBITS	0x0000000000004010
27	.comment	SHT_PROGBITS	0x0000000000000000
28	.symtab	SHT_SYMTAB	0x0000000000000000
29	.strtab	SHT_STRTAB	0x0000000000000000
30	.shstrtab	SHT_STRTAB	0x0000000000000000

PIE

- This is useful for loading shared objects into another executable.

Section header tables
======================

Nr	Name	Type	Address
0		SHT_NULL	0x0000000000000000
1	.interp	SHT_PROGBITS	0x0000000000000318
2	.note.gnu.property	SHT_NOTE	0x0000000000000338
3	.note.gnu.build-id	SHT_NOTE	0x0000000000000368
4	.note.ABI-tag	SHT_NOTE	0x000000000000038C
5	.gnu.hash	Unknown	0x00000000000003B0
6	.dynsym	SHT_DYNSYM	0x00000000000003D8
7	.dynstr	SHT_STRTAB	0x0000000000000480
8	.gnu.version	Unknown	0x000000000000051E
9	.gnu.version_r	Unknown	0x0000000000000530
10	.rela.dyn	SHT_RELA	0x0000000000000570
11	.rela.plt	SHT_RELA	0x0000000000000630
12	.init	SHT_PROGBITS	0x0000000000001000
13	.plt	SHT_PROGBITS	0x0000000000001020
14	.plt.got	SHT_PROGBITS	0x0000000000001040
15	.plt.sec	SHT_PROGBITS	0x0000000000001050
16	.text	SHT_PROGBITS	0x0000000000001060
17	.fini	SHT_PROGBITS	0x0000000000001184
18	.rodata	SHT_PROGBITS	0x0000000000002000
19	.eh_frame_hdr	SHT_PROGBITS	0x0000000000002004
20	.eh_frame	SHT_PROGBITS	0x0000000000002038
21	.init_array	SHT_INIT_ARRAY	0x0000000000003DB8
22	.fini_array	SHT_FINI_ARRAY	0x0000000000003DC0
23	.dynamic	SHT_DYNAMIC	0x0000000000003DC8
24	.got	SHT_PROGBITS	0x0000000000003FB8
25	.data	SHT_PROGBITS	0x0000000000004000
26	.bss	SHT_NOBITS	0x0000000000004010
27	.comment	SHT_PROGBITS	0x0000000000000000
28	.symtab	SHT_SYMTAB	0x0000000000000000
29	.strtab	SHT_STRTAB	0x0000000000000000
30	.shstrtab	SHT_STRTAB	0x0000000000000000

PIE

- This is useful for loading shared objects into another executable.
- Downside: we won't easily know where sections are in memory.

Section header tables
======================

Nr	Name	Type	Address
0		SHT_NULL	0x0000000000000000
1	.interp	SHT_PROGBITS	0x0000000000000318
2	.note.gnu.property	SHT_NOTE	0x0000000000000338
3	.note.gnu.build-id	SHT_NOTE	0x0000000000000368
4	.note.ABI-tag	SHT_NOTE	0x000000000000038C
5	.gnu.hash	Unknown	0x00000000000003B0
6	.dynsym	SHT_DYNSYM	0x00000000000003D8
7	.dynstr	SHT_STRTAB	0x0000000000000480
8	.gnu.version	Unknown	0x000000000000051E
9	.gnu.version_r	Unknown	0x0000000000000530
10	.rela.dyn	SHT_RELA	0x0000000000000570
11	.rela.plt	SHT_RELA	0x0000000000000630
12	.init	SHT_PROGBITS	0x0000000000001000
13	.plt	SHT_PROGBITS	0x0000000000001020
14	.plt.got	SHT_PROGBITS	0x0000000000001040
15	.plt.sec	SHT_PROGBITS	0x0000000000001050
16	.text	SHT_PROGBITS	0x0000000000001060
17	.fini	SHT_PROGBITS	0x0000000000001184
18	.rodata	SHT_PROGBITS	0x0000000000002000
19	.eh_frame_hdr	SHT_PROGBITS	0x0000000000002004
20	.eh_frame	SHT_PROGBITS	0x0000000000002038
21	.init_array	SHT_INIT_ARRAY	0x0000000000003DB8
22	.fini_array	SHT_FINI_ARRAY	0x0000000000003DC0
23	.dynamic	SHT_DYNAMIC	0x0000000000003DC8
24	.got	SHT_PROGBITS	0x0000000000003FB8
25	.data	SHT_PROGBITS	0x0000000000004000
26	.bss	SHT_NOBITS	0x0000000000004010
27	.comment	SHT_PROGBITS	0x0000000000000000
28	.symtab	SHT_SYMTAB	0x0000000000000000
29	.strtab	SHT_STRTAB	0x0000000000000000
30	.shstrtab	SHT_STRTAB	0x0000000000000000

What is helpful in PIE?

- Note, you must have a loaded readable SEGMENT that contains the program headers (data on the segments)

- It is helpful to have a section called .shstrtab

SHSTRTAB

- This section is just a list of names

3580	73	74	72	74	61	62	00	2E	73	68	73	74	72	74	61	62	00	2E	69	6E	74	65	72	70	00	2E	6E	6F	74	65	2E	67	strtab .shstrtab .interp .note.g
35A0	6E	75	2E	70	72	6F	70	65	72	74	79	00	2E	6E	6F	74	65	2E	67	6E	75	2E	62	75	69	6C	64	2D	69	64	00	2E	nu.property .note.gnu.build-id .
35C0	6E	6F	74	65	2E	41	42	49	2D	74	61	67	00	2E	67	6E	75	2E	68	61	73	68	00	2E	64	79	6E	73	79	6D	00	2E	note.ABI-tag .gnu.hash .dynsym .
35E0	64	79	6E	73	74	72	00	2E	67	6E	75	2E	76	65	72	73	69	6F	6E	00	2E	67	6E	75	2E	76	65	72	73	69	6F	6E	dynstr .gnu.version .gnu.version
3600	5F	72	00	2E	72	65	6C	61	2E	64	79	6E	00	2E	72	65	6C	61	2E	70	6C	74	00	2E	69	6E	69	74	00	2E	70	6C	_r .rela.dyn .rela.plt .init .pl
3620	74	2E	67	6F	74	00	2E	70	6C	74	2E	73	65	63	00	2E	74	65	78	74	00	2E	66	69	6E	69	00	2E	72	6F	64	61	t.got .plt.sec .text .fini .roda
3640	74	61	00	2E	65	68	5F	66	72	61	6D	65	5F	68	64	72	00	2E	65	68	5F	66	72	61	6D	65	00	2E	69	6E	69	74	ta .eh_frame_hdr .eh_frame .init
3660	5F	61	72	72	61	79	00	2E	66	69	6E	69	5F	61	72	72	61	79	00	2E	64	79	6E	61	6D	69	63	00	2E	64	61	74	_array .fini_array .dynamic .dat

ELF Generation

- Fill out the TODO parts in the ELFMaker class
- Once this is done, your code will be generating executable files

Code Testing

Compile some test file locally

- Assume you are in the folder where “bin” contains your class files (compiled Compiler)

```
java –cp bin miniJava.Compiler ../my/test/file.java
```

Compile some test file locally

- Assume you are in the folder where "bin" contains your class files (compiled Compiler)

java –cp bin miniJava.Compiler ../my/test/file.java

This will output your "a.out" file.

SSH to a server I set up

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

Username: comp520

Password: comp520

SSH to a server I set up

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

Username: comp520

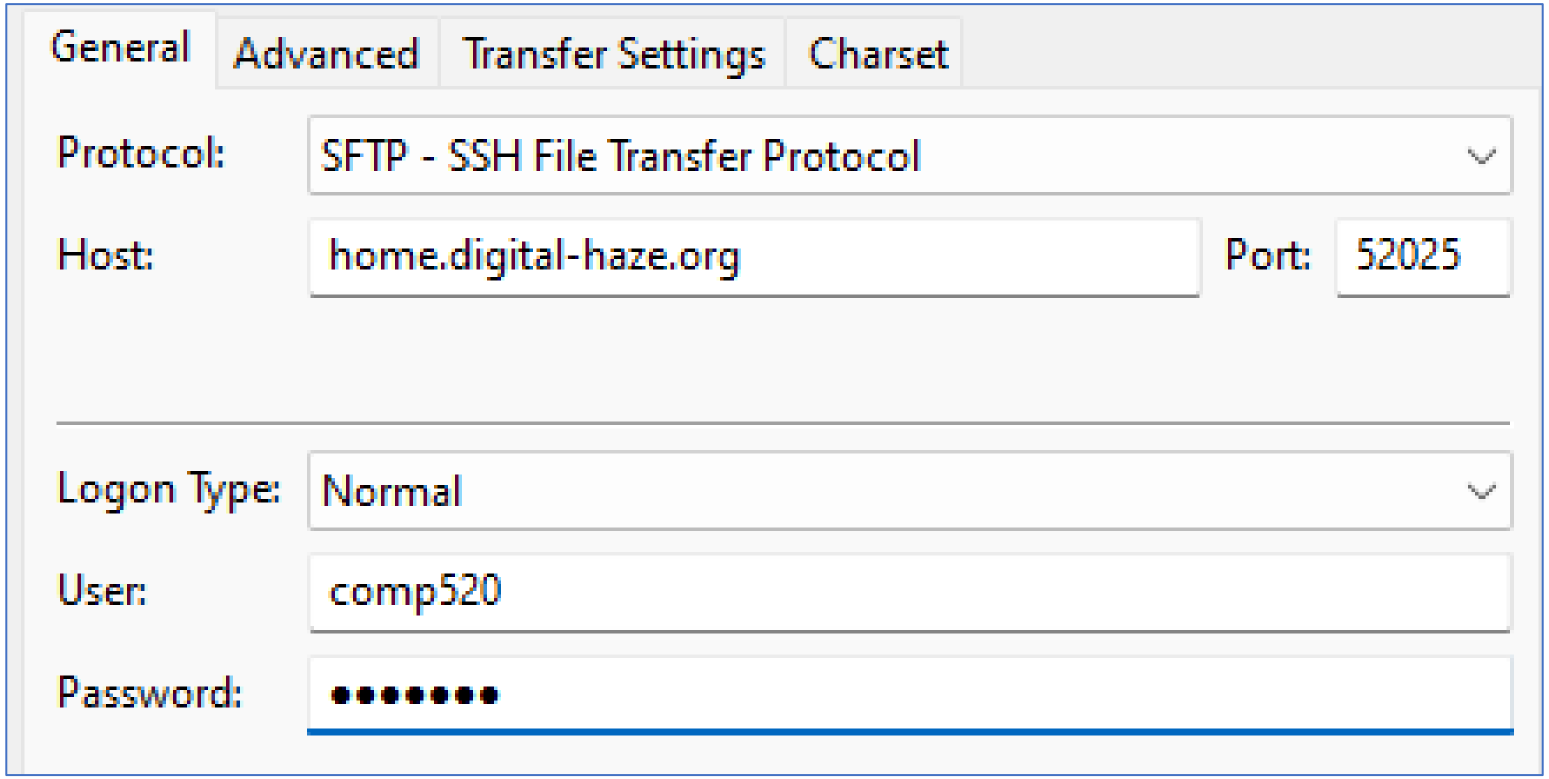
Password: comp520

Create a folder for yourself with your onyen:

mkdir swali

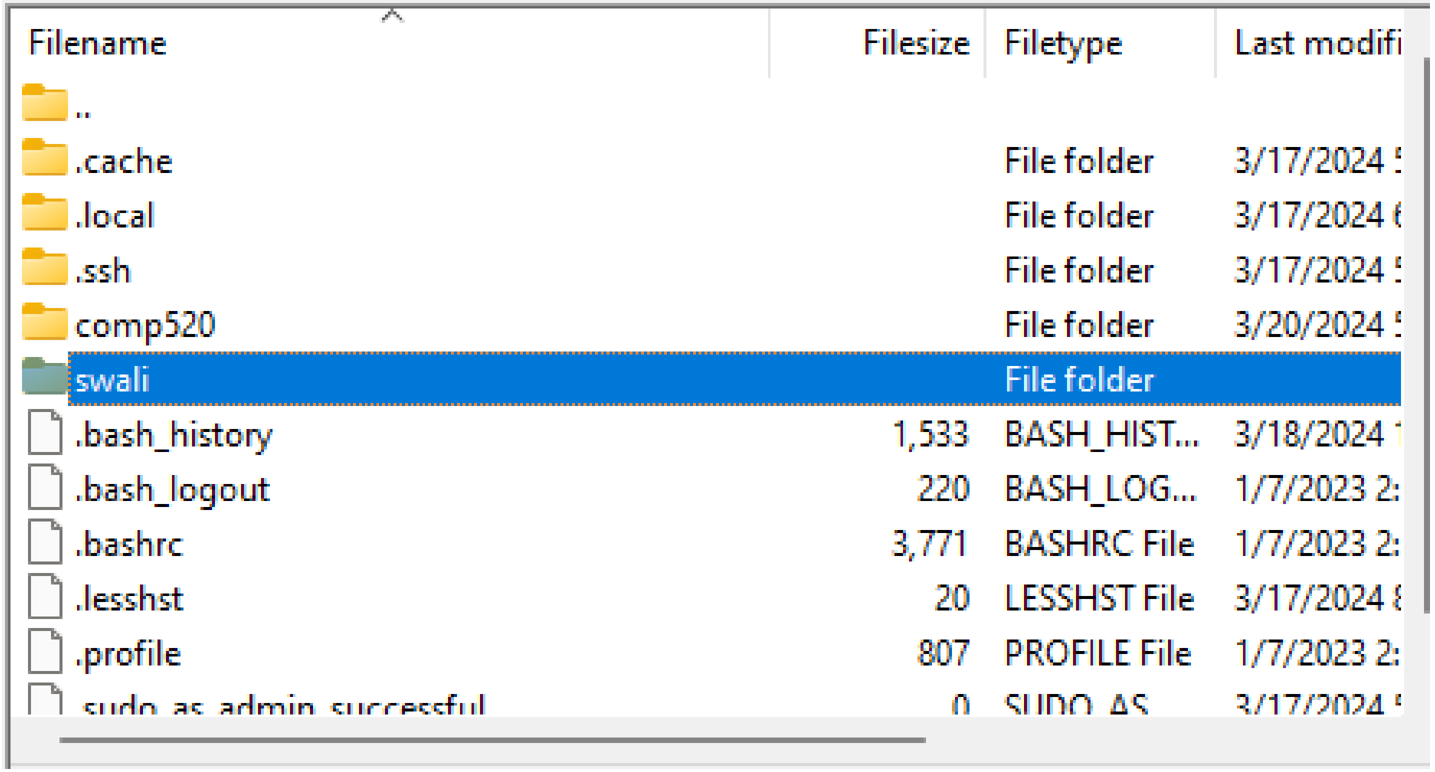
Use FileZilla (or any other method)

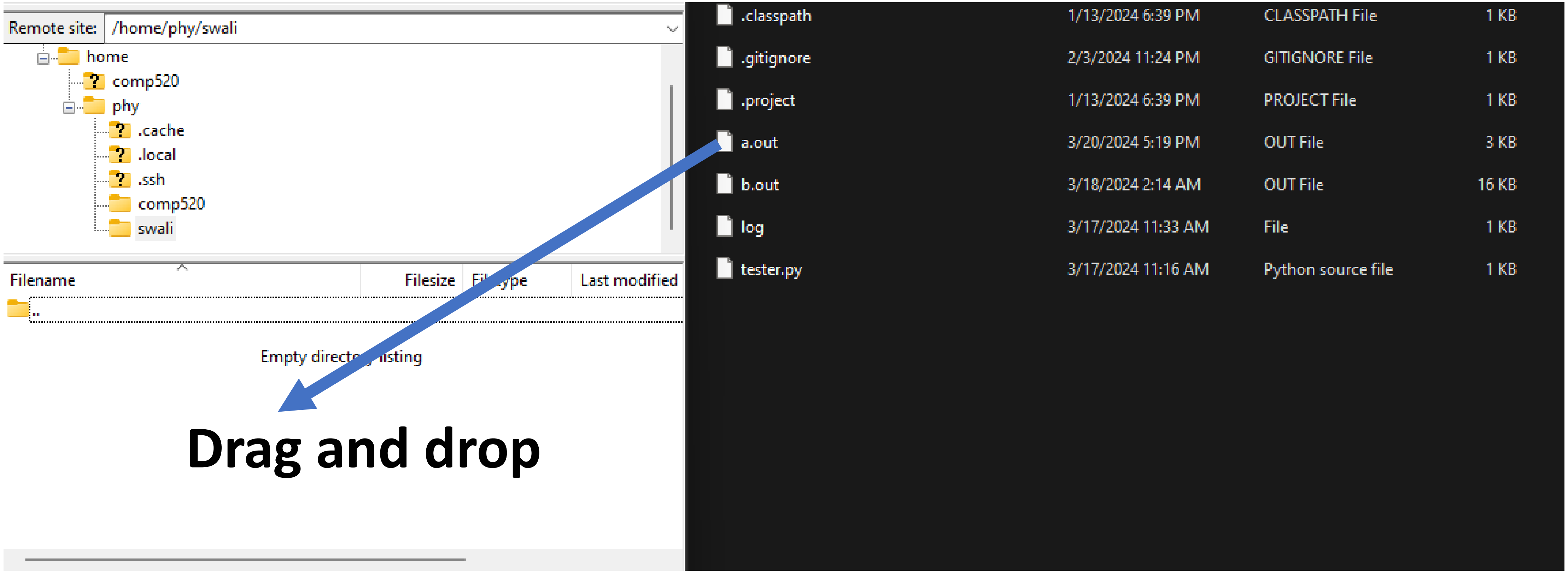
- Connect to the server:



Enter your folder

Double Click





Drag and drop

Run your a.out file

```
comp520@ubuntutest:~/swali$ ./a.out
```

- Make sure the output matches what you were expecting.

End