# COMP 520 - Compilers

## Lecture 13 – Branch Construction

# PA3 Extension

Due on Friday 3/22 11:59pm

(Late penalty afterwards)

# Finally, PA4!

- **PA1:** Large step into an unknown project
- **PA2:** Comply with someone else's code
- **PA3:** Organize and finish the project (your code and theirs) to accomplish the goal of validating input source code

- **PA4:** ?

# Finally, PA4!

- **PA1:** Large step into an unknown project
- **PA2:** Comply with someone else's code
- **PA3:** Organize and finish the project (your code and theirs) to accomplish the goal of validating input source code
- **PA4:** Research the unknown, **read documentation**, and write the bytecode output!

# Today's Goal

Implement:

```
if( ( a >= b ) == ( c < d ) ) {
    a = 3;
    b = 4;
}
```

# Today's Goal

Implement:

```
if( ( a >= b ) == ( c < d ) ) {
    a = 3;
    b = 4;
}
```

Using Comparisons

Storing Comparisons (for comparing later)

Conditional Execution

# Intermediate Goal

- What does code generation look like?

# Intermediate Goal (2)

- What does code generation look like?
- General idea: Work on one section at a time.

# Intermediate Goal (3)

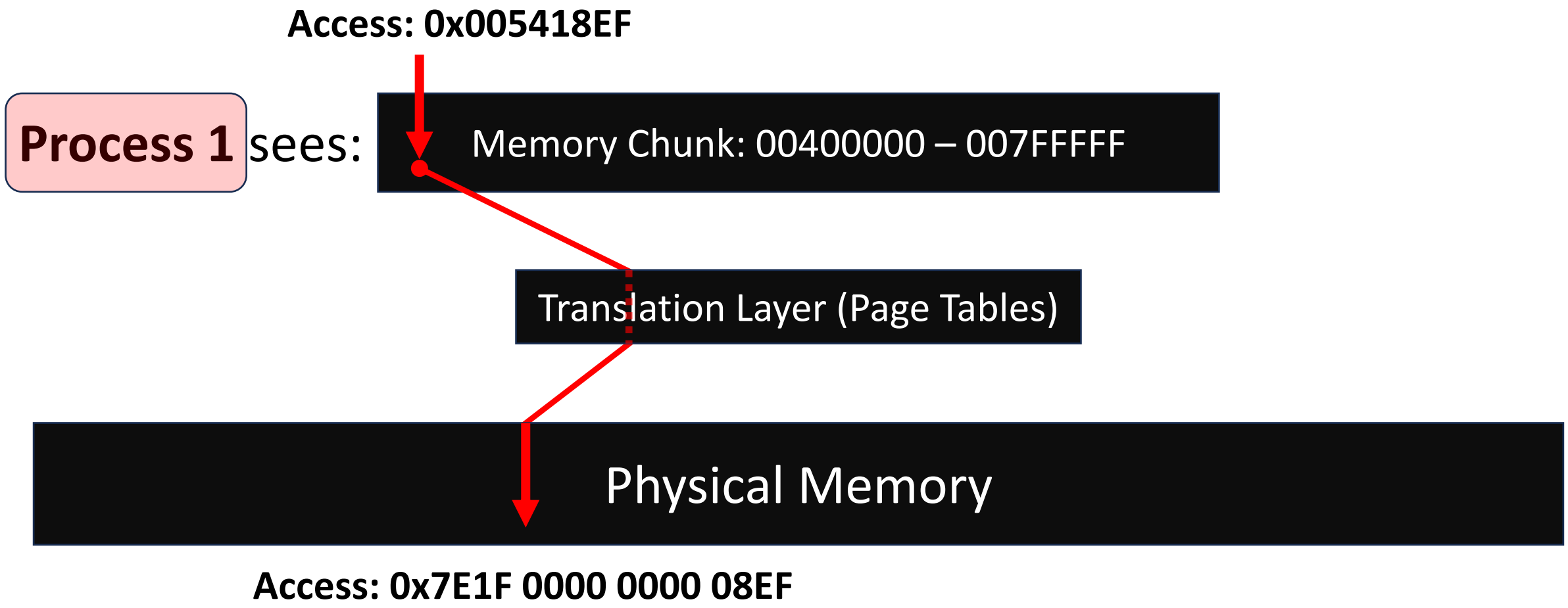
- What does code generation look like?

- General idea: Work on one section at a time.

- Where is our code? Where is our data?
- Don't know, don't need to know yet, first section we work on is generating "oblivious" incomplete code.

# Virtual Memory

What you could spend days/weeks on, gets two slides.



# Virtual Memory

## What you could spend days/weeks on, gets two slides.

**Access: 0x005418EF (same address as Proc1)**

**Process 2** sees:

Memory Chunk: 00400000 – 007FFFFF

Translation Layer (Page Tables)

Physical Memory

**Access: 0x8E6F 0230 0000 08EF**

# Variable References- Where?

- If it is static, it is in either .bss or .data

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

- **Strategy:** Pick a size for .bss, and assign each static variable an offset from the start of .bss
- Then, pick a start address for .bss
  - E.g. ".bss starts in memory at 0x00400000"

# Where is .bss?

- We can set it if we are an executable file
- We don't get to set it if we are a **shared object**.

- A shared object is loaded "somewhere" in a parent process.

# Where is .bss?

- We can set it if we are an executable file
- We don't get to set it if we are a **shared object**.

- In the latter case, where is .bss?
- How can I refer to **0x40408080C0C00000** + bssOffset when I don't actually know where .bss starts from?
- To test your researching capabilities, this will be a WA3 question (coming soon!)

# Executable Section

- We won't know the size of the executable section until we're done.
- We do need a start address though (unless *.so*)

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

- **Strategy:** Pick ".text starts at 0x00800000"
  - Where we know the size of any previous sections.

# Code Generation

- Strategy: Evaluate simple expressions with RAX
- We will need additional complexity for PA4

# Code Generation Example

Visit Method

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

# Code Generation Example

Visit Stmt

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

# Code Generation Example

| Variables | | |
|---|---|---|
| x | rbp−0 | ? |
| | | |

```
sub rsp, 8
```

Create "x"

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

# Code Generation Example

```
sub rsp, 8
mov rax, 4
```

| Variables | | |
|---|---|---|
| x | rbp−0 | ? |
| | | |

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

Visit Expr

# Code Generation Example

| Variables | | |
|---|---|---|
| x | rbp−0 | 4 |
| | | |

```
sub rsp, 8
mov rax, 4
mov [rbp], rax
```

Store

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

# Code Generation Example

```
sub rsp, 8
mov rax, 4
mov [rbp], rax
```

Visit Stmt

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

| Variables | | |
|---|---|---|
| x | rbp−0 | 4 |
| | | |

# Code Generation Example

| Variables | | |
|---|---|---|
| x | rbp−0 | 4 |
| | | |

```
sub rsp, 8
mov rax, 4
mov [rbp], rax
```

```
mov rax, [rbp]
add rax, 3
```

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

Visit Expr

# Code Generation Example

| Variables | | |
|---|---|---|
| x | rbp−0 | 7 |
| | | |

```
sub rsp, 8
mov rax, 4
mov [rbp], rax
mov rax, [rbp]
add rax, 3
```

Store

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

```
mov [rbp], rax
```

# Code Generation Example

| Variables | | |
|---|---|---|
| x | rbp−0 | 7 |
| | | |

```
sub rsp, 8
mov rax, 4
mov [rbp], rax
mov rax, [rbp]
add rax, 3
mov [rbp], rax
```

Visit Stmt

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

# Code Generation Example

| Variables | | |
|---|---|---|
| x | rbp−0 | 7 |
| y | rbp−8 | ? |

```
sub  rsp, 8
mov  rax, 4
mov  [rbp], rax
mov  rax, [rbp]
add  rax, 3
mov  [rbp], rax
```

Create "y"

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

```
sub  rsp, 8
```

# Code Generation Example

| Variables | | |
|---|---|---|
| x | rbp−0 | 7 |
| y | rbp−8 | ? |

```
sub rsp, 8
mov rax, 4
mov [rbp], rax
mov rax, [rbp]
add rax, 3
mov [rbp], rax
sub rsp, 8
```

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

Visit Expr

```
mov rax, 2
```

# Code Generation Example

| Variables | | |
|---|---|---|
| x | rbp−0 | 7 |
| y | rbp−8 | 2 |

```
sub  rsp, 8
mov  rax, 4
mov  [rbp], rax
mov  rax, [rbp]
add  rax, 3
mov  [rbp], rax
sub  rsp, 8
mov  rax, 2
```

Store

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

```
mov  [rbp−8], rax
```

# Code Generation Example

```
sub rsp, 8
mov rax, 4
mov [rbp], rax
mov rax, [rbp]
add rax, 3
mov [rbp], rax
sub rsp, 8
mov rax, 2
mov [rbp-8], rax
```

| Variables | | |
|---|---|---|
| x | rbp-0 | 7 |
| y | rbp-8 | 2 |

Visit Stmt

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

# Code Generation Example

```
sub rsp, 8
mov rax, 4
mov [rbp], rax
mov rax, [rbp]
add rax, 3
mov [rbp], rax
sub rsp, 8
mov rax, 2
mov [rbp-8], rax
```

| Variables | | |
|---|---|---|
| x | rbp-0 | 7 |
| y | rbp-8 | 2 |

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

Visit Expr

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

# Code Generation Example

```
sub  rsp, 8
mov  rax, 4
mov  [rbp], rax
mov  rax, [rbp]
add  rax, 3
mov  [rbp], rax
sub  rsp, 8
mov  rax, 2
mov  [rbp-8], rax
mov  rax, [rbp]
add  rax, [rbp-8]
```

| Variables | | |
|---|---|---|
| x | rbp-0 | 9 |
| y | rbp-8 | 2 |

Store

```
public static void main(String[] args) {
    int x = 4;
    x = x + 3;
    int y = 2;
    x = x + y;
    y = y + x;
}
```

```
mov  [rbp], rax
```

**Flat assembler file is on the course website if you want it.**

# Let’s see this in action.

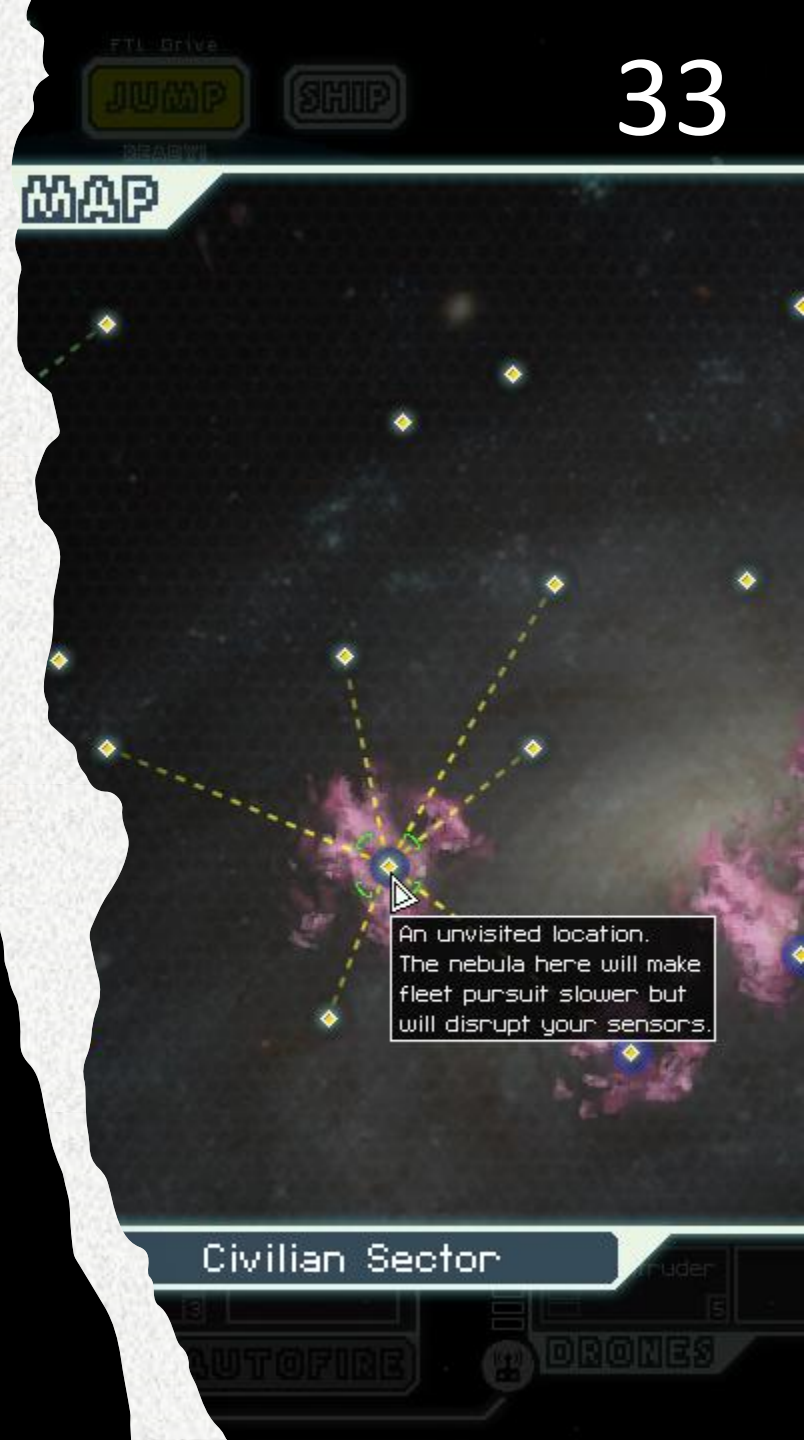
Additionally, see how branching is done!



# In-Class Demo

FTL: Faster Than Light

Goals: observe register and branch behavior



# Generating Branches

# Condition

- **Recall:** `cmp a, b` ≡ **set** RFLAGS **after** "a−b"

# Conditions (S1)

```
( a >= b ) == ( c < d )
```

- **Strategy 1:** Is there a way to get data out of the FLAGS and store them somewhere?

```
temp0 = a >= b
temp1 = c < d
temp0 = temp0 == temp1
cmp temp0,0
jnz …
```

# Conditions (S2)

```
( a >= b ) == ( c < d )
```

- **Strategy 2:** Figure out a way to leave the last binary comparison, and utilize `je` at the very end

```
temp0 = a >= b
temp1 = c < d
cmp temp0, temp1
je …
```

# Conditions (S3)

```
( a >= b ) == ( c < d )
```

- **Strategy 3:** use the conditional jumps themselves:

```
        cmp  a, b
        jge  isEqual
        mov  [rbp-04], 0
        jmp  End
isEqual: mov  [rbp-04], 1
    End: …
```

```
        cmp  c, d
        jl  isEq2
        mov  [rbp-08], 0
        jmp  End2
isEq2: mov  [rbp-08], 1
  End2: …
```

# Conditions (S3)

```
( a >= b ) == ( c < d )
```

- Strategy 3 (continued):

```
…
mov rax, [rbp-04]
cmp rax, [rbp-08]
je finalIsEqual
```

# Strategy 3

- This strategy isn't wrong, just inelegant
- Optimization is actually quite difficult

# What do you think?

- What does the VisualC compiler do?
- Strategy 1, 2, 3, or something else?
- Or is it something so optimized that it would take an hour to analyze?

- Taking guesses!

# What strategy is this?

```
    11:     if ((a >= b) == (c < d)) {
00007FF6CDA24217 8B 45 24            mov         eax,dword ptr [b]
00007FF6CDA2421A 39 45 04            cmp         dword ptr [a],eax
00007FF6CDA2421D 7C 0C               jl          main+4Bh (07FF6CDA2422Bh)
00007FF6CDA2421F C7 85 34 01 00 00 01 00 00 00 mov         dword ptr [rbp+134h],1
00007FF6CDA24229 EB 0A               jmp         main+55h (07FF6CDA24235h)
00007FF6CDA2422B C7 85 34 01 00 00 00 00 00 00 mov         dword ptr [rbp+134h],0
00007FF6CDA24235 8B 45 64            mov         eax,dword ptr [d]
00007FF6CDA24238 39 45 44            cmp         dword ptr [c],eax
00007FF6CDA2423B 7D 0C               jge         main+69h (07FF6CDA24249h)
00007FF6CDA2423D C7 85 38 01 00 00 01 00 00 00 mov         dword ptr [rbp+138h],1
00007FF6CDA24247 EB 0A               jmp         main+73h (07FF6CDA24253h)
00007FF6CDA24249 C7 85 38 01 00 00 00 00 00 00 mov         dword ptr [rbp+138h],0
00007FF6CDA24253 8B 85 38 01 00 00   mov         eax,dword ptr [rbp+138h]
00007FF6CDA24259 39 85 34 01 00 00   cmp         dword ptr [rbp+134h],eax
00007FF6CDA2425F 75 0C               jne         main+8Dh (07FF6CDA2426Dh)
    12:         printf("true\n");
00007FF6CDA24261 48 8D 0D 60 69 00 00 lea         rcx,[string "true\n" (07FF6CDA2ABC8h)]
00007FF6CDA24268 E8 E9 D1 FF FF      call        printf (07FF6CDA21456h)
    13:     }
```

```
    11:     if ((a >= b) == (c < d)) {
00007FF6CDA24217 8B 45 24              mov         eax,dword ptr [b]
00007FF6CDA2421A 39 45 04              cmp         dword ptr [a],eax
00007FF6CDA2421D 7C 0C                 jl          main+4Bh (07FF6CDA2422Bh)
00007FF6CDA2421F C7 85 34 01 00 00 01 00 00 00 mov         dword ptr [rbp+134h],1
00007FF6CDA24229 EB 0A                 jmp         main+55h (07FF6CDA24235h)
00007FF6CDA2422B C7 85 34 01 00 00 00 00 00 00 mov         dword ptr [rbp+134h],0
00007FF6CDA24235 8B 45 64              mov         eax,dword ptr [d]
00007FF6CDA24238 39 45 44              cmp         dword ptr [c],eax
00007FF6CDA2423B 7D 0C                 jge         main+69h (07FF6CDA24249h)
00007FF6CDA2423D C7 85 38 01 00 00 01 00 00 00 mov         dword ptr [rbp+138h],1
00007FF6CDA24247 EB 0A                 jmp         main+73h (07FF6CDA24253h)
00007FF6CDA24249 C7 85 38 01 00 00 00 00 00 00 mov         dword ptr [rbp+138h],0
00007FF6CDA24253 8B 85 38 01 00 00     mov         eax,dword ptr [rbp+138h]
00007FF6CDA24259 39 85 34 01 00 00     cmp         dword ptr [rbp+134h],eax
00007FF6CDA2425F 75 0C                 jne         main+8Dh (07FF6CDA2426Dh)
    12:         printf("true\n");
00007FF6CDA24261 48 8D 0D 60 69 00 00 lea         rcx,[string "true\n" (07FF6CDA2ABC8h)]
00007FF6CDA24268 E8 E9 D1 FF FF        call        printf (07FF6CDA21456h)
    13:     }
```

# MSVC Used Strategy 3!!

- Yes, that's right
- Throw out preconceived notions of elegant/inelegant

- In the world of assembly, you do what works!
- (And if in the business of optimization, you find the fastest operations, even if it is horribly inelegant)

# Organizing Assembly

- We could spend an entire semester on optimization strategies where we just build the compiler more and more and more…
- In the interest of time, you may want to take shortcuts in PA4. If it increases execution time by 40ns, the autograder won't likely notice.

# Strategy 1 & 2 (Easy)

- There are instructions that you can use

```
setz, setl, setle, setg, setge
```

# Strategy 1 & 2 (Easy)

- There are instructions that you can use

```
setz, setl, setle, setg, setge
```

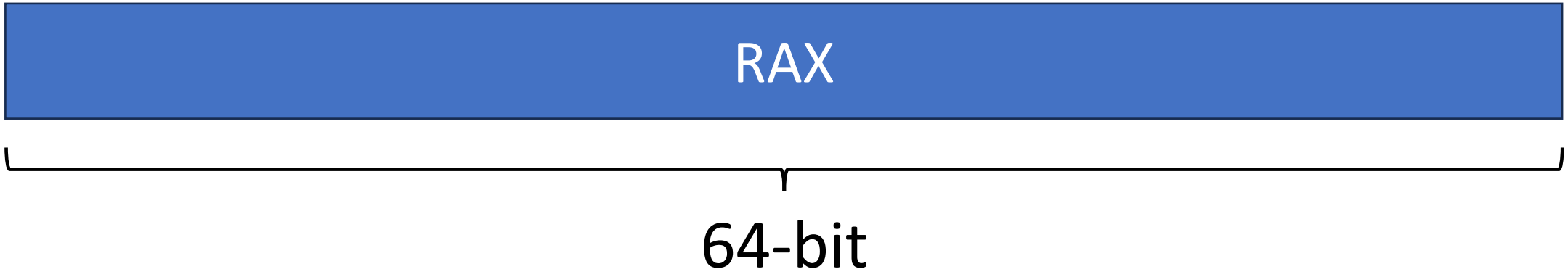
- They will set a byte (al/cl/dl/bl/ah/ch/dh/bh) depending on if the flags are zero, less, …, etc.
- But what if you didn't see that documentation?

# More Registers?

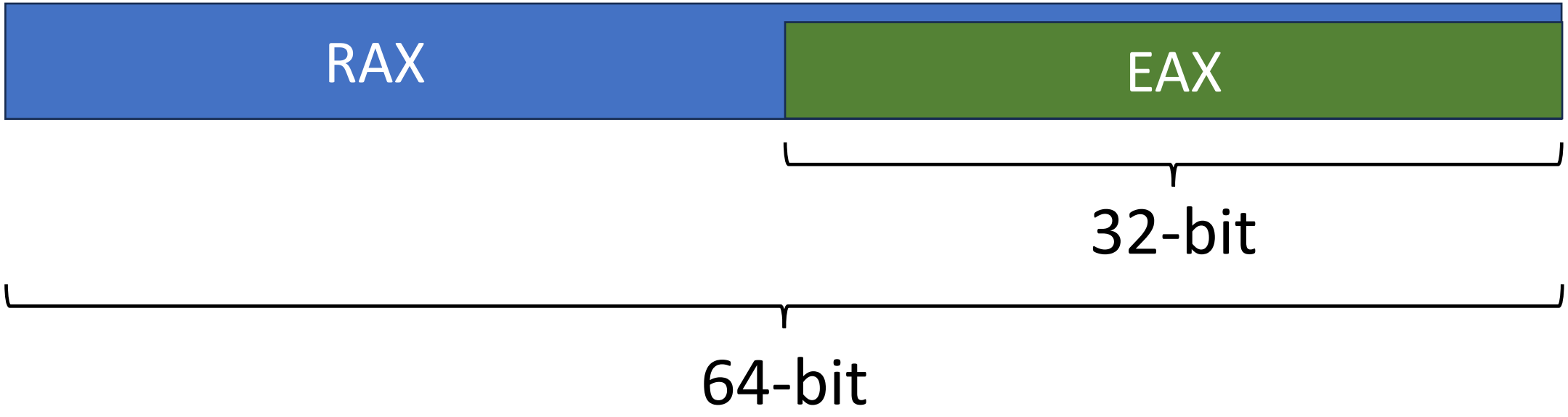
- Wait a minute, al/cl/dl/bl/ah/ch/dh/bh are not registers we have seen before!

# x64 General Purpose Registers Overview

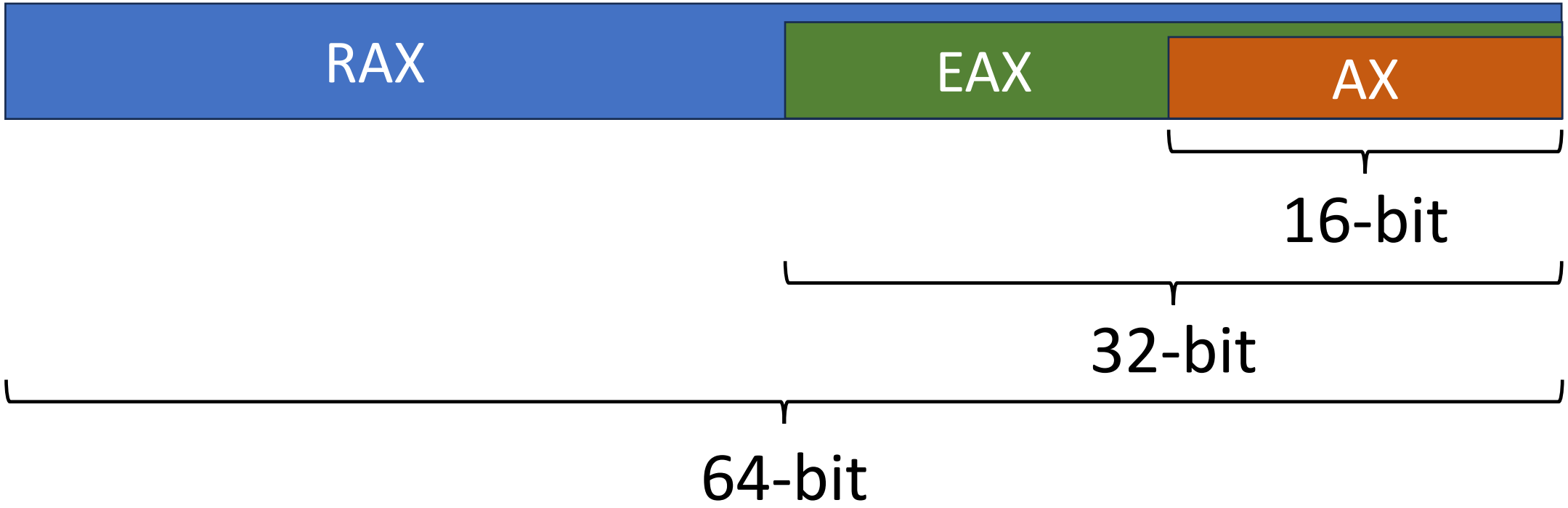
- Wait a minute, al/cl/dl/bl/ah/ch/dh/bh are not registers we have seen before!



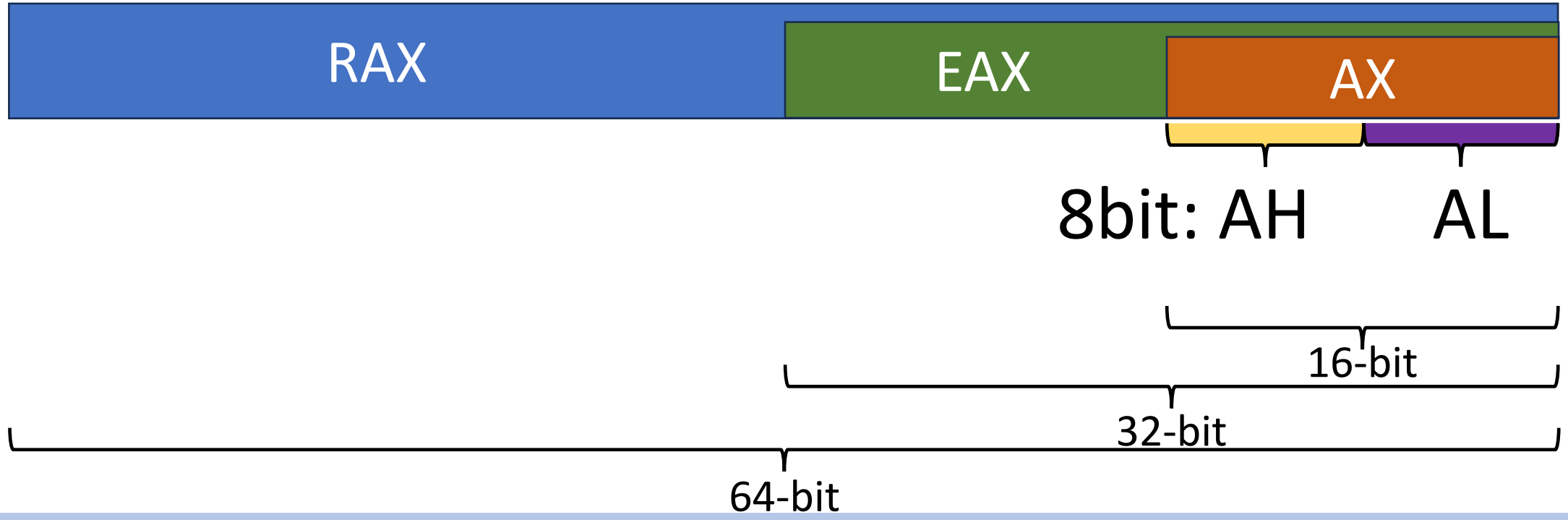
# x64 General Purpose Registers Overview



# x64 General Purpose Registers Overview



# x64 General Purpose Registers Overview



# Useful Register List

| **64-bit** | **RAX** | | **RCX** | | **RDX** | | **RBX** | | **RSP** | **RBP** | **RSI** | **RDI** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 32-bit | EAX | | ECX | | EDX | | EBX | | ESP | EBP | ESI | EDI |
| 16-bit | AX | | CX | | DX | | BX | | SP | BP | SI | DI |
| 8-bit | AH | AL | CH | CL | DH | DL | BH | BL | ? | ? | ? | ? |
| Honorable Mentions: MM0, ST(0), XMM0, YMM0, ZMM0, CS, SS, DS, ES, FS, GS, GDTR, IDTR, TR, LDTR, CR, DR, IP… | | | | | | | | | | | | |

# Useful Register List (2) (REX prefix)

| **64-bit** | **R8** | **R9** | **R10** | **R11** | **R12** | **R13** | **R14** | **R15** |
|---|---|---|---|---|---|---|---|---|
| 32-bit | R8D | R9D | R10D | R11D | R12D | R13D | R14D | R15D |
| If operand R: REX.R, if R/M base: REX.B, if index: REX.X | | | | | | | | |
| Note: Flag REX.W has to be set to use 64-bit *operands* (RAX-R15) (otherwise it defaults to 32-bit operand EAX-R15D) We will cover encoding later. | | | | | | | | |

# Back to the problem

- This emulates what your googling might look like:

- *"Oh, I see something I haven't seen before"*
- *"I have now learned the new thing"*
- *"How can I apply the new thing?"*

# Back to the problem

- We are currently after:
- “I want to resolve a == b” to a value

# Back to the problem

- We are currently after:

- "I want to resolve a == b" to a value
- I will XOR RAX,RAX (Why?)
- I will CMP a,b
- I will SETE al
- Now RAX contains 0 or 1, depending on a==b

# Strategy 1 & 2 (Easy)

- There are instructions that you can use

  `setz, setl, setle, setg, setge`

- They will set a byte (`al/cl/dl/bl/ah/ch/dh/bh`) depending on if the flags are zero, less, …, etc.
- **But what if you didn't see that documentation?**

# Strategy 1 & 2

- There is an instruction that you can use

```
pushfq
```

- It stores the **RFLAGS** data on the stack
- From this, we can determine >=, <=, <, >, ==, !=
- Example: `[rsp] & 0x00C0`

  Same as "<="

Why? (You will need to research **RFLAGS**)

# Strategy 1 & 2 (without SETLE)

```
if( [rsp] & 0x00C0 )
```

Any ideas on what the corresponding assembly looks like? Hint: `cmp` is not needed.

# Strategy 1 & 2 (without SETLE)

```
if( [rsp] & 0x00C0 )
```

Any ideas on what the corresponding assembly looks like? Hint: `cmp` is not needed.

```
and qword[rsp], 0xC0
jnz …
```

# Lastly: Conditional jump is misleading

```
if( cond )
    StmtA
StmtB
```

**Will see this in action in the next section**

More accurately: jump if the condition is NOT true.

So `IfStmt` will be: `jmp` if condition is false, otherwise continue to "`ThenStmt`", then `jmp` past "`ElseStmt`"

# What have we learned?

- Evaluating the condition parameter will be challenging
- But not intellectually difficult, just need to make sure your code “lines up” properly with what you want.

# Organizing Then/Else

# Two chunks of code

- What to execute when the condition is true (if cond)
- What to execute when the condition is false (else)

```
if( cond )
        ThenStmt
(else ElseStmt)?
```

# Recall Demo Example

```
.text:00518FC9 mov     eax, [eax+4D4h]
.text:00518FCF cmp     [ecx+0E8h], eax ; This looks interesting
.text:00518FD5 jg      short loc__text_518FF0 ; Some kind of comparison
```

```
.text:00518FD7 mov     eax, [ecx]
.text:00518FD9 call    dword ptr [eax+20h]
```

```
.text:00518FF0
.text:00518FF0 loc__text_518FF0:
.text:00518FF0 mov     ecx, offset dword__bss_9196E0
.text:00518FF5 mov     dword ptr [esp], 6
.text:00518FFC lea     ebx, [ebp+var_18]
.text:00518FFF call    sub_text_5BA3A0
```

# Recall Demo Example

```
.text:00518FC9 mov     eax, [eax+4D4h]
.text:00518FCF cmp     [ecx+0E8h], eax ; This looks interesting
.text:00518FD5 jg      short loc__text_518FF0 ; Some kind of comparison
```

```
.text:00518FD7 mov     eax, [ecx]
.text:00518FD9 call    dword ptr [eax+20h]
```

```
.text:00518FF0
.text:00518FF0 loc__text_518FF0:
.text:00518FF0 mov     ecx, offset dword__bss_9196E0
.text:00518FF5 mov     dword ptr [esp], 6
.text:00518FFC lea     ebx, [ebp+var_18]
.text:00518FFF call    sub_text_5BA3A0
```

The “**not true**” portion of code is immediately after the “**jump if true**”

# Recall Demo Example

```
.text:00518FC9 mov     eax, [eax+4D4h]
.text:00518FCF cmp     [ecx+0E8h], eax ; This looks interesting
.text:00518FD5 jg      short loc__text_518FF0 ; Some kind of comparison
```

```
.text:00518FD7 mov     eax, [ecx]
.text:00518FD9 call    dword ptr [eax+20h]
```

```
.text:00518FF0
.text:00518FF0 loc__text_518FF0:
.text:00518FF0 mov     ecx, offset dword__bss_9196E0
.text:00518FF5 mov     dword ptr [esp], 6
.text:00518FFC lea     ebx, [ebp+var_18]
.text:00518FFF call    sub_text_5BA3A0
```

But we don’t always have an else statement!

# Misleading Conditions Part 1

```
if( x == 3 )
    x = 4;
y = 3;
```

We have no else statement.

What if we branch on the same condition as our source code?

i.e., use "je" because we see "=="

# Misleading Conditions Part 1

```
if( x == 3 )
    x = 4;
y = 3;
```

We have no else statement.

So the code could be:

```
         cmp [x],3
         je IsEq
         jmp AfterEq
IsEq:    mov [x],4
AfterEq: mov [y],3
```

# Misleading Conditions Part 2

```
if( x == 3 )
    x = 4;
y = 3;
```

```
                cmp [x],3
                jne AfterIfStmt
                mov [x],4
AfterIfStmt:    mov [y],3
```

Less instructions is better!

# Misleading Conditions Part 3

```
if( x == 3 )
    x = 4;
else
    x = 5;
y = 3;
```

```
                    cmp [x],3
                    jne AfterIfStmt
                    mov [x],4
                    jmp AfterElseStmt
AfterIfStmt:        mov [x],5
AfterElseStmt:      mov [y],3
```

# Instruction Patching

- But we do not know where an "**AfterIfStmt**" is!
- Ideas?

# Instruction Patching (2)

- But we do not know where an "**AfterIfStmt**" is!
- Generate a placeholder conditional jump
  - E.g., `jge 0x00000000`
- Keep track of where you have this placeholder

# Instruction Patching (3)

- Generate a placeholder conditional jump
  - E.g., `jge 0x00000000`

- Keep track of where you have this placeholder
- Generate the "thenStmt" code
- Finally, go back and patch "placeholder"
  - `jge 0x00000000` -> `jge 0x00001CF0`

# PA3 Due soon, PA4 Next!

Don’t forget PA3!!

# End