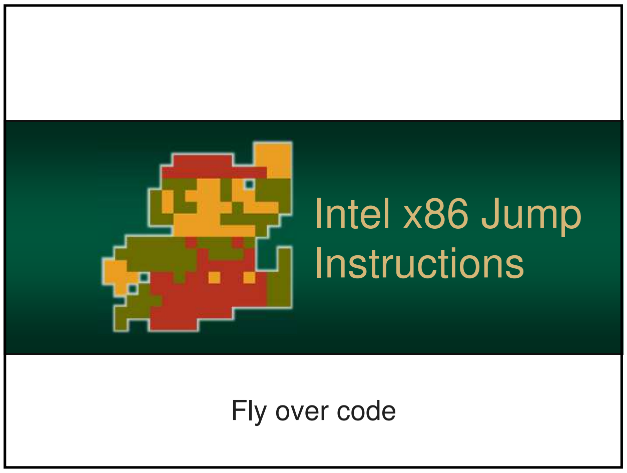
Control Logic

Part 6

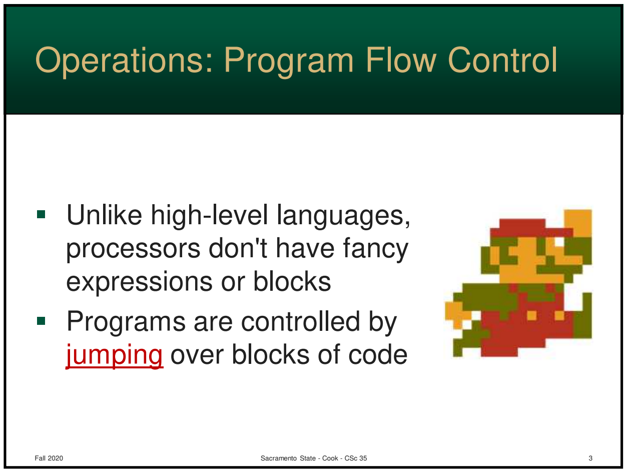
1



Intel x86 Jump Instructions

Fly over code

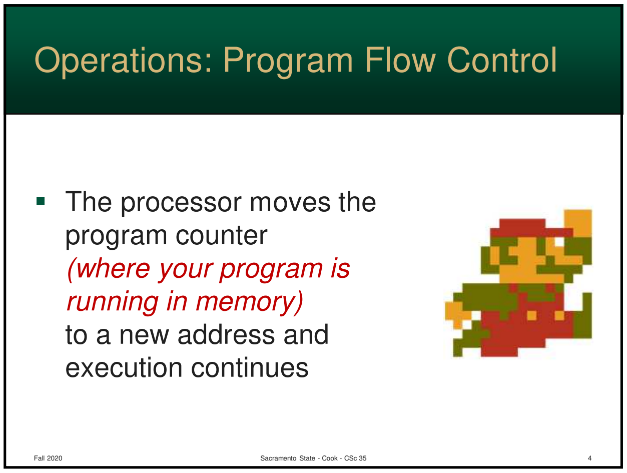
2



Operations: Program Flow Control

- Unlike high-level languages, processors don't have fancy expressions or blocks
- Programs are controlled by jumping over blocks of code

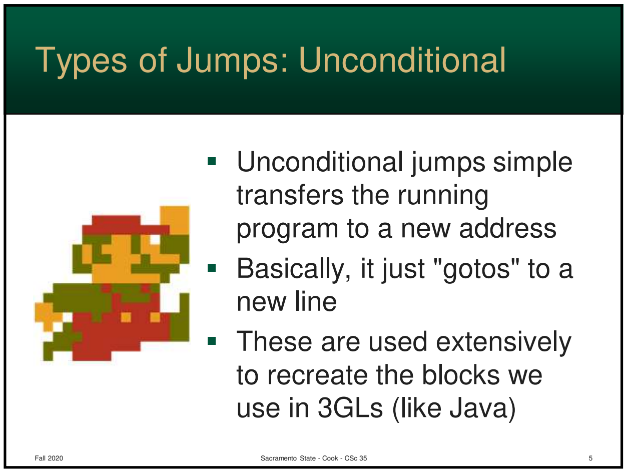
3



Operations: Program Flow Control

- The processor moves the program counter *(where your program is running in memory)* to a new address and execution continues

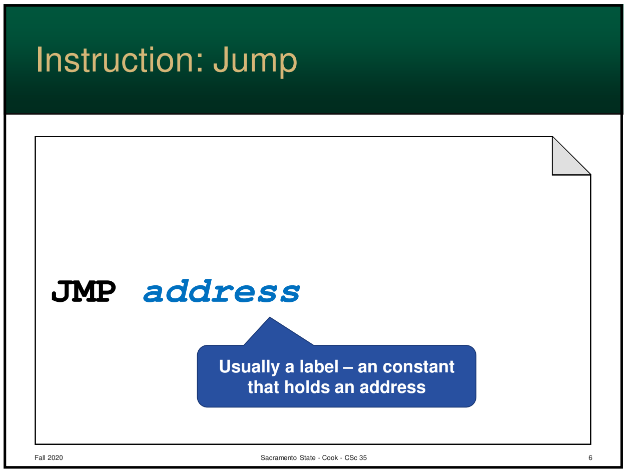
4



Types of Jumps: Unconditional

- Unconditional jumps simple transfers the running program to a new address
- Basically, it just "gotos" to a new line
- These are used extensively to recreate the blocks we use in 3GLs (like Java)

5

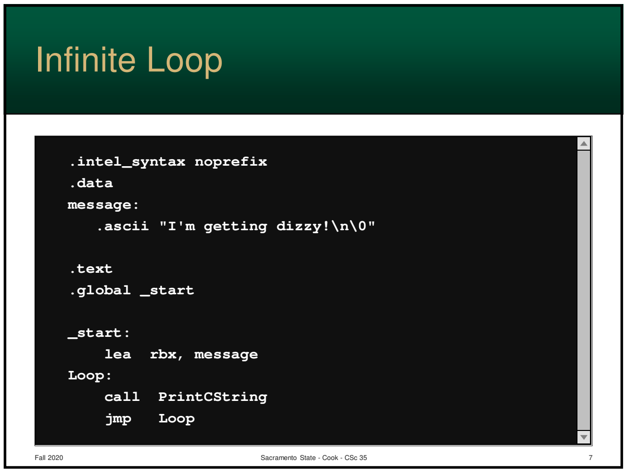


Instruction: Jump

```
JMP    address
```

Usually a label – an constant that holds an address

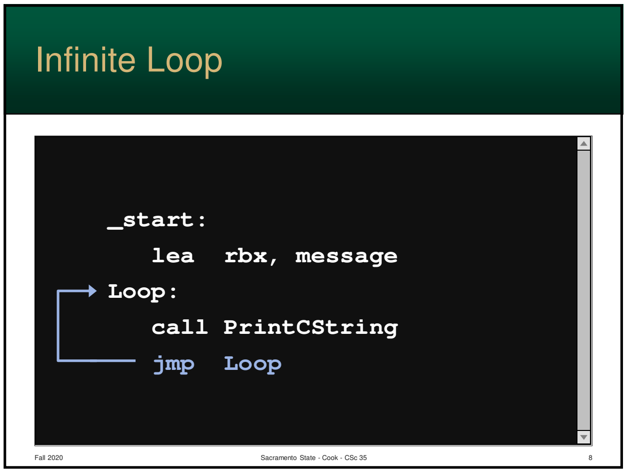
6



Infinite Loop

```
.intel_syntax noprefix
.data
message:
    .ascii "I'm getting dizzy!\n\0"

.text
.global _start

_start:
    lea  rbx, message
Loop:
    call  PrintCString
    jmp   Loop
```

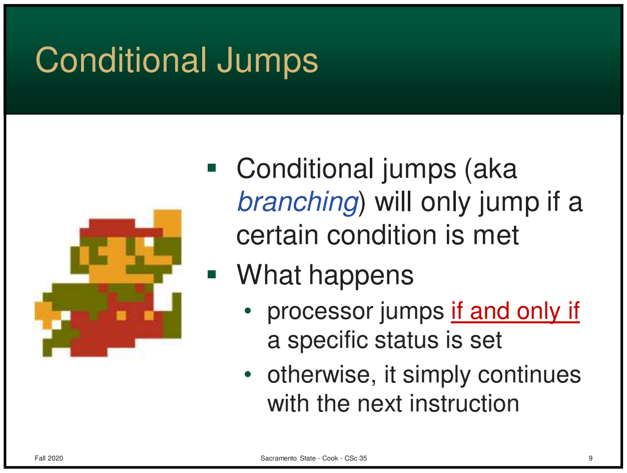
7



Infinite Loop

```
_start:
     lea  rbx, message
Loop:
     call PrintCString
     jmp  Loop
```

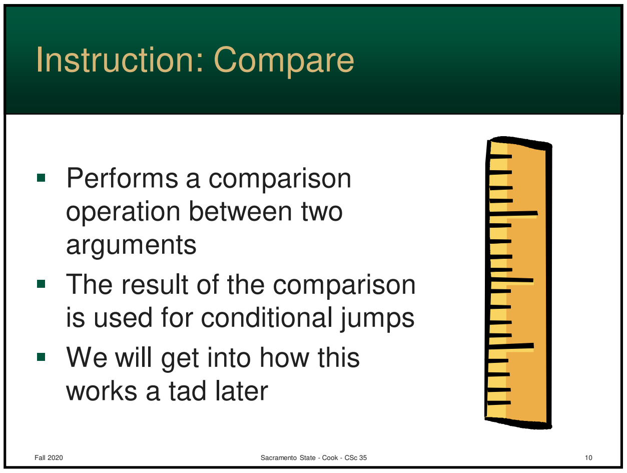
8



Conditional Jumps

- Conditional jumps (aka *branching*) will only jump if a certain condition is met
- What happens
  - processor jumps if and only if a specific status is set
  - otherwise, it simply continues with the next instruction

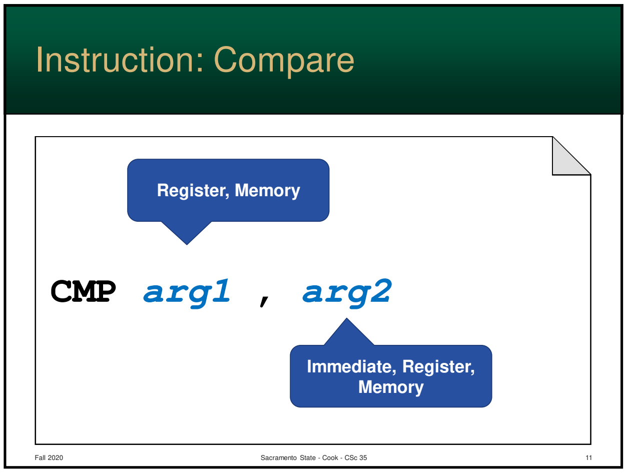
9



Instruction: Compare

- Performs a comparison operation between two arguments
- The result of the comparison is used for conditional jumps
- We will get into how this works a tad later

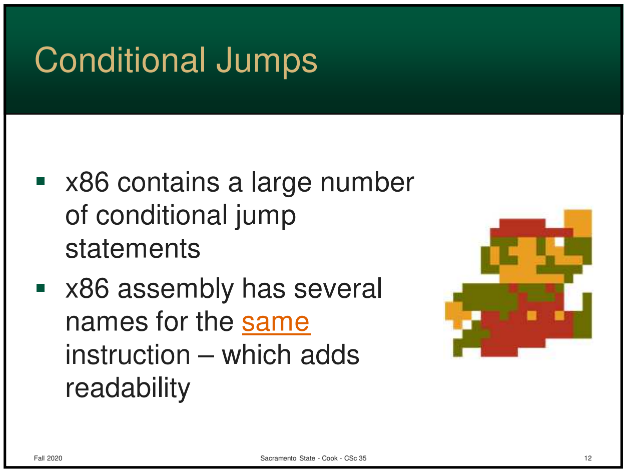
10



Instruction: Compare

CMP *arg1* , *arg2*

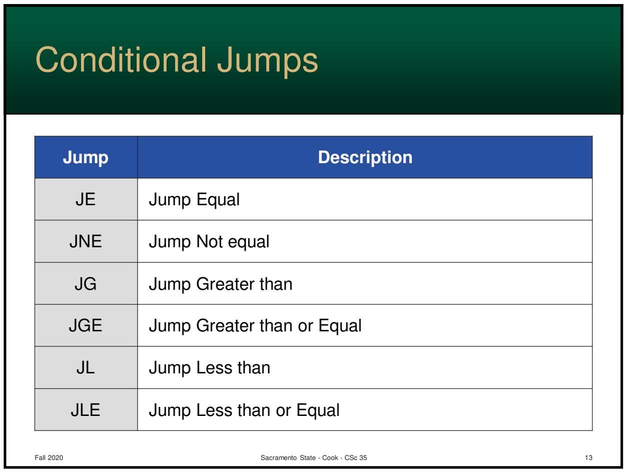
11



Conditional Jumps

- x86 contains a large number of conditional jump statements
- x86 assembly has several names for the same instruction – which adds readability

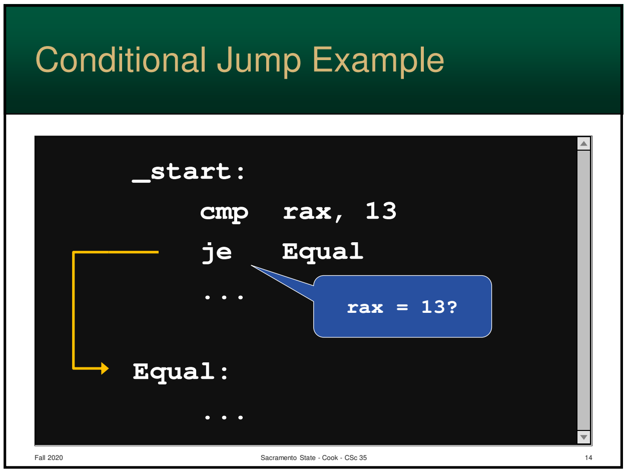
12



Conditional Jumps

Jump	Description
JE	Jump Equal
JNE	Jump Not equal
JG	Jump Greater than
JGE	Jump Greater than or Equal
JL	Jump Less than
JLE	Jump Less than or Equal

13

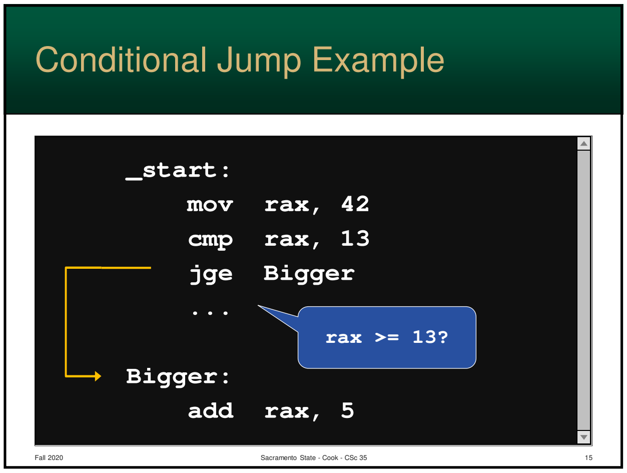


Conditional Jump Example

```
_start:
        cmp   rax, 13
        je    Equal
        ...
Equal:
        ...
```

rax = 13?

14

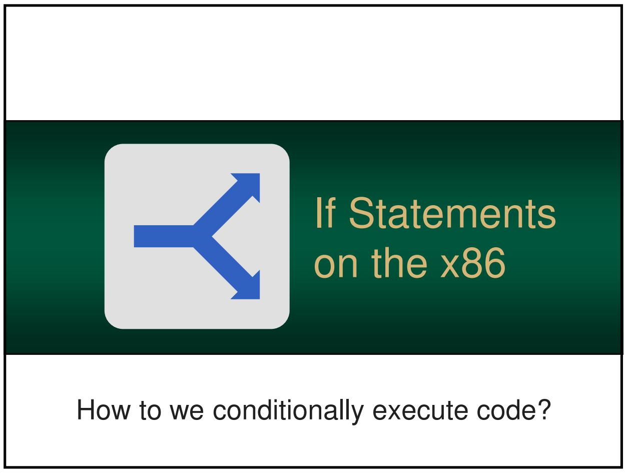


Conditional Jump Example

```
_start:
       mov   rax, 42
       cmp   rax, 13
       jge   Bigger
       ...
Bigger:
       add   rax, 5
```

rax >= 13?

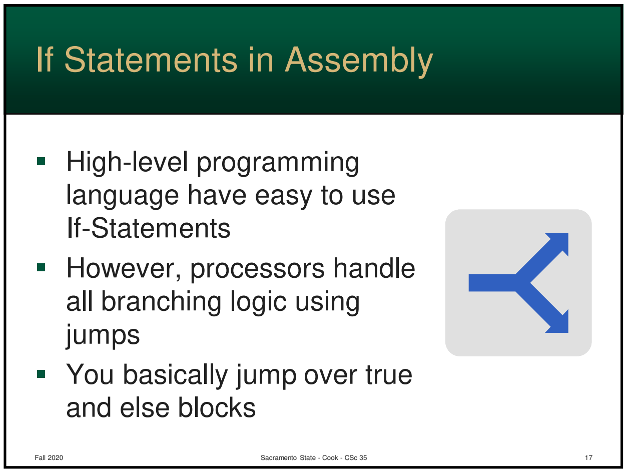
15



If Statements on the x86

How to we conditionally execute code?

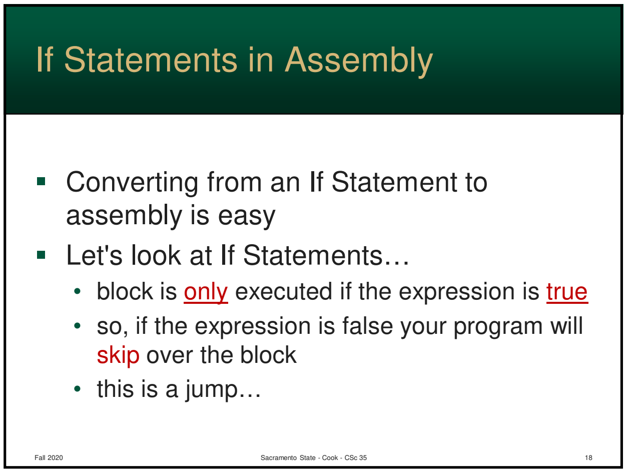
16



If Statements in Assembly

- High-level programming language have easy to use If-Statements
- However, processors handle all branching logic using jumps
- You basically jump over true and else blocks

17

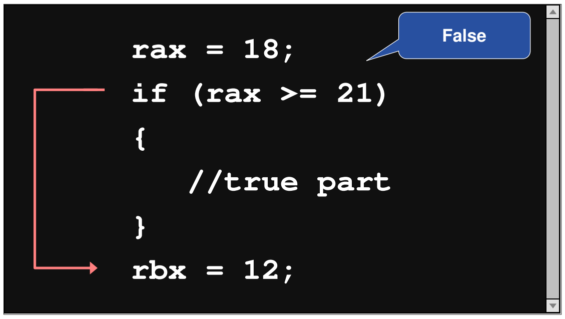


If Statements in Assembly

- Converting from an If Statement to assembly is easy
- Let's look at If Statements…
  - block is only executed if the expression is true
  - so, if the expression is false your program will skip over the block
  - this is a jump…

18

If Statement jumps over code



```
rax = 18;
if (rax >= 21)
{
   //true part
}
rbx = 12;
```

False

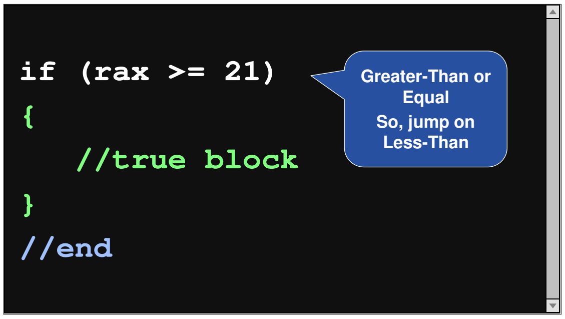
Converting an If Statement

- Compare the two values
- If the result is *false* …
  - then jump over the true block
  - you will need label to jump to
- To jump on false, reverse your logic
  - `a < b` → `not (a >= b)`
  - `a >= b` → `not (a < b)`

Please Note…

- Following examples use *very* generic label names
- In your program, each label you create must be unique
- So, please don't think that each label (as it is typed) is "the" label you need to use

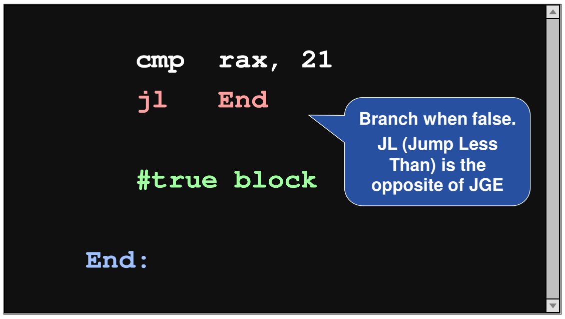
Converting an If Statement



```
if (rax >= 21)
{
   //true block
}
//end
```

Greater-Than or Equal
So, jump on Less-Than

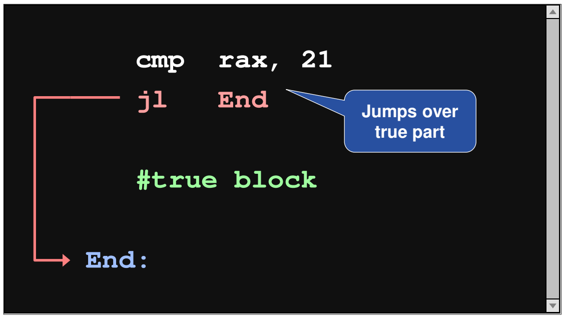
Jump over true part



```
      cmp   rax, 21
      jl    End

      #true block

   End:
```

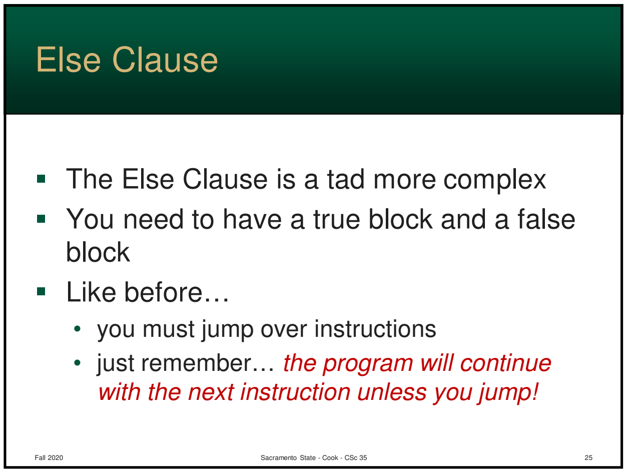
Branch when false. JL (Jump Less Than) is the opposite of JGE

Jump over true part



```
      cmp   rax, 21
      jl    End

      #true block

   End:
```

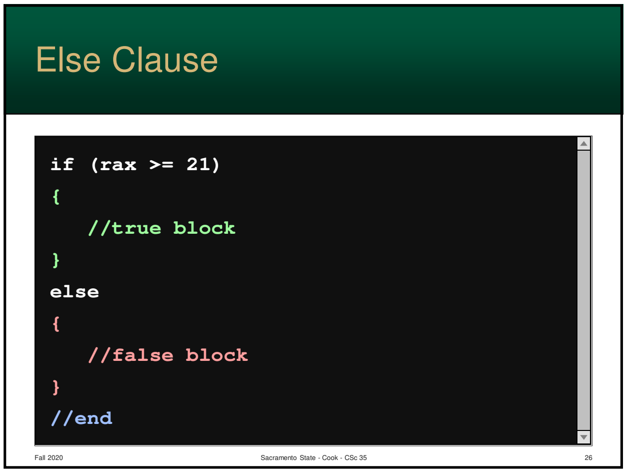
Jumps over true part



Else Clause

- The Else Clause is a tad more complex
- You need to have a true block and a false block
- Like before…
  - you must jump over instructions
  - just remember… *the program will continue with the next instruction unless you jump!*

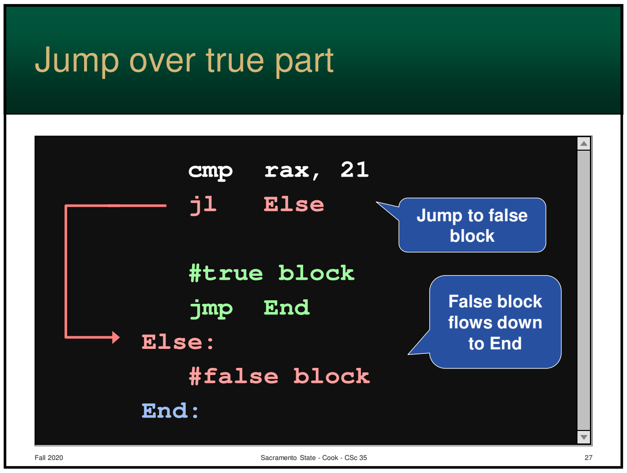
25



Else Clause

```
if (rax >= 21)
{
   //true block
}
else
{
   //false block
}
//end
```

26



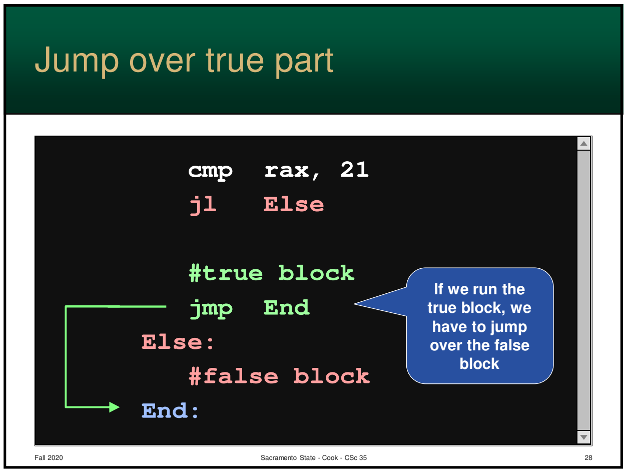
Jump over true part

```
      cmp   rax, 21
      jl    Else

      #true block
      jmp   End
   Else:
      #false block
   End:
```

Jump to false block

False block flows down to End

27

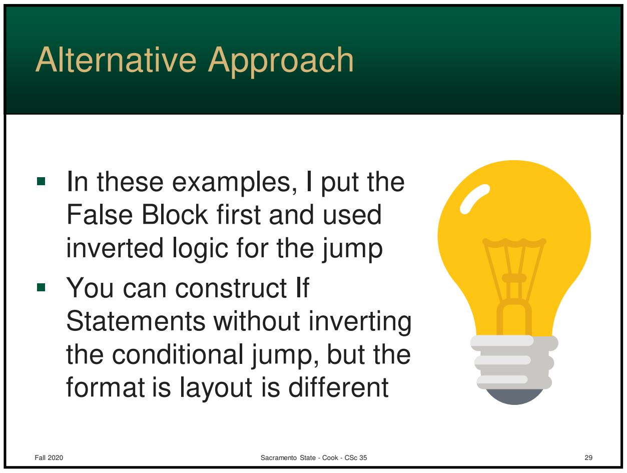


Jump over true part

```
      cmp   rax, 21
      jl    Else

      #true block
      jmp   End
   Else:
      #false block
   End:
```

If we run the true block, we have to jump over the false block

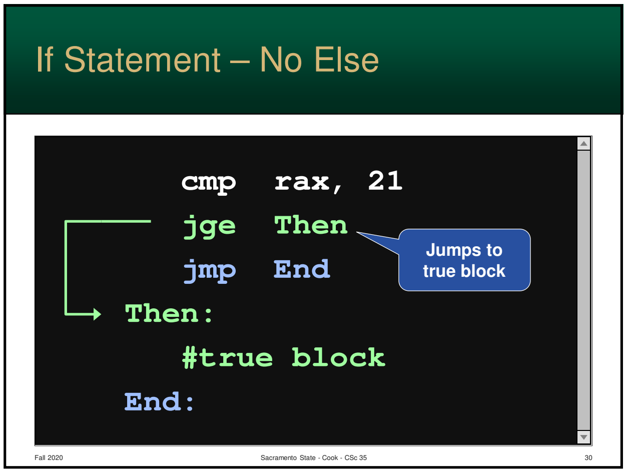
28



Alternative Approach

- In these examples, I put the False Block first and used inverted logic for the jump
- You can construct If Statements without inverting the conditional jump, but the format is layout is different

29

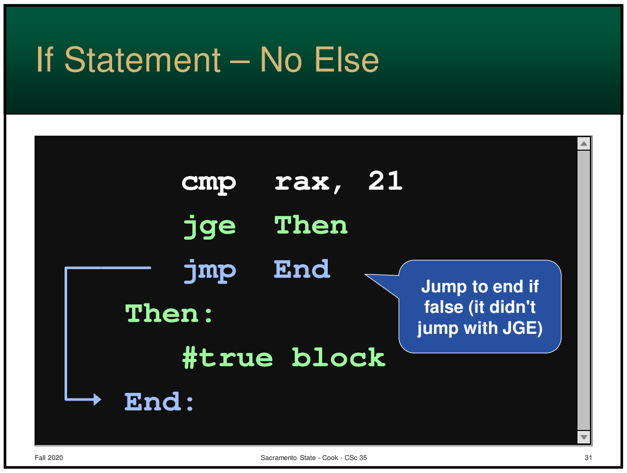


If Statement – No Else

```
      cmp   rax, 21
      jge   Then
      jmp   End
   Then:
      #true block
   End:
```

Jumps to true block

30

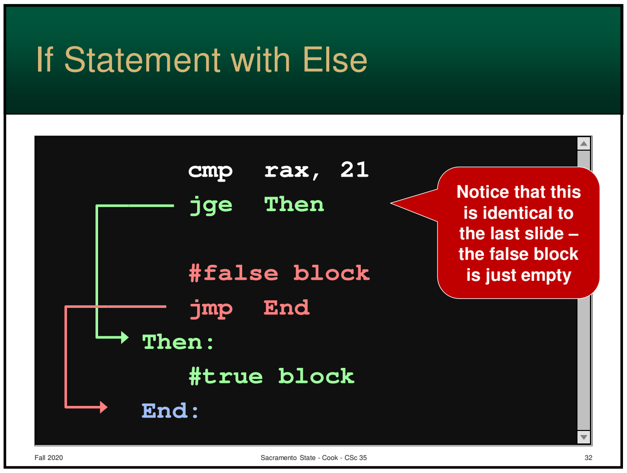


If Statement – No Else

```
        cmp   rax, 21
        jge   Then
        jmp   End
    Then:
        #true block
    End:
```

Jump to end if false (it didn't jump with JGE)

31

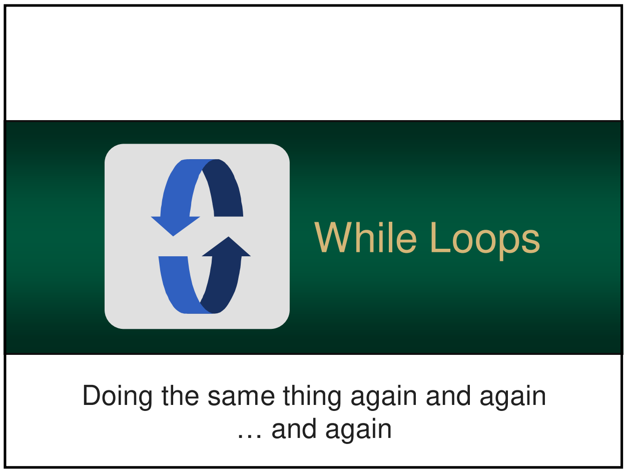


If Statement with Else

```
        cmp   rax, 21
        jge   Then

        #false block
        jmp   End
    Then:
        #true block
    End:
```

Notice that this is identical to the last slide – the false block is just empty

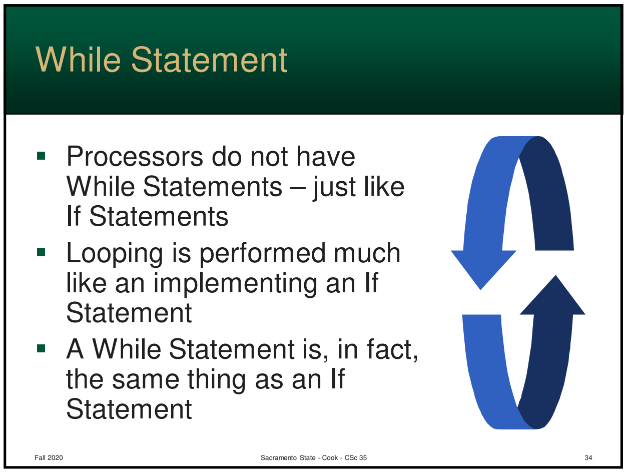
32



While Loops

Doing the same thing again and again … and again

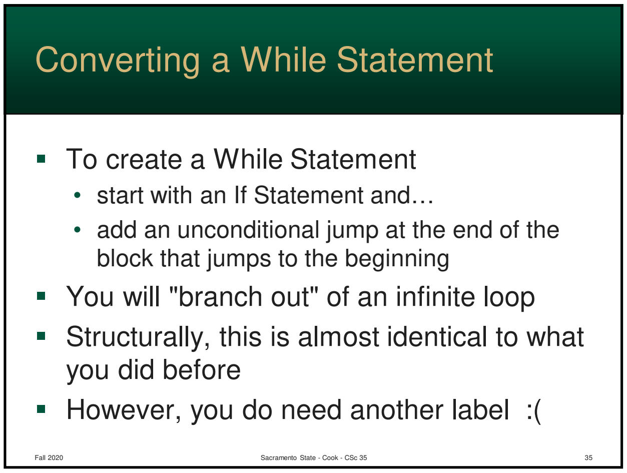
33



While Statement

- Processors do not have While Statements – just like If Statements
- Looping is performed much like an implementing an If Statement
- A While Statement is, in fact, the same thing as an If Statement

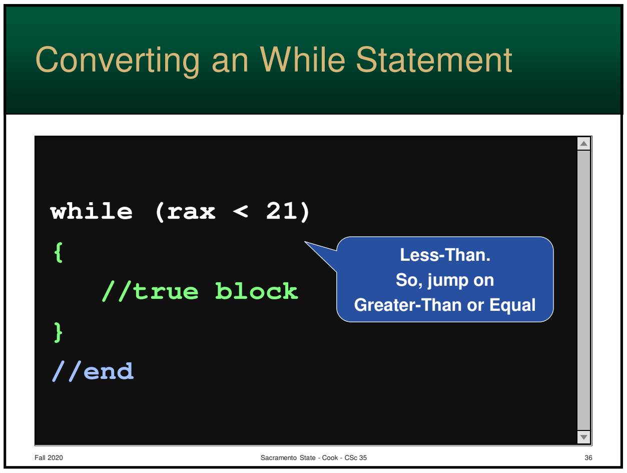
34



Converting a While Statement

- To create a While Statement
  - start with an If Statement and…
  - add an unconditional jump at the end of the block that jumps to the beginning
- You will "branch out" of an infinite loop
- Structurally, this is almost identical to what you did before
- However, you do need another label :(

35

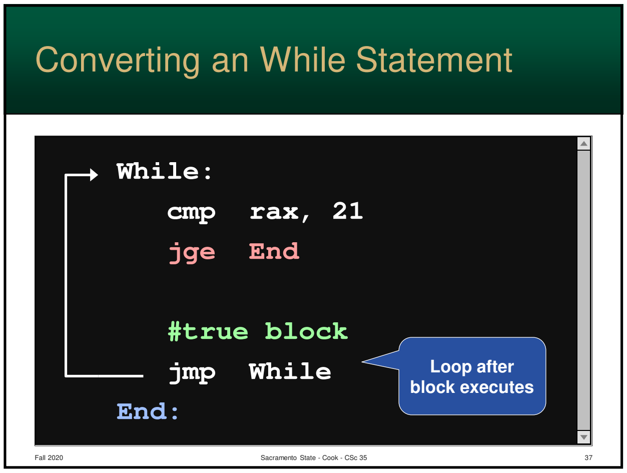


Converting an While Statement

```
while (rax < 21)
{
   //true block
}
//end
```

Less-Than. So, jump on Greater-Than or Equal

36

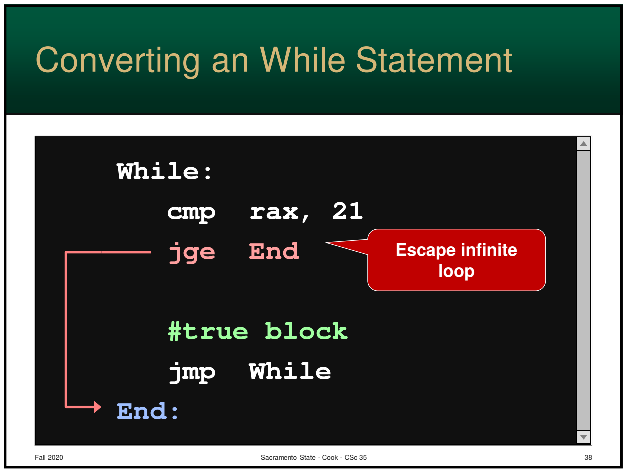


Converting an While Statement

```
 While:
     cmp   rax, 21
     jge   End

     #true block
     jmp   While
 End:
```

Loop after block executes

37

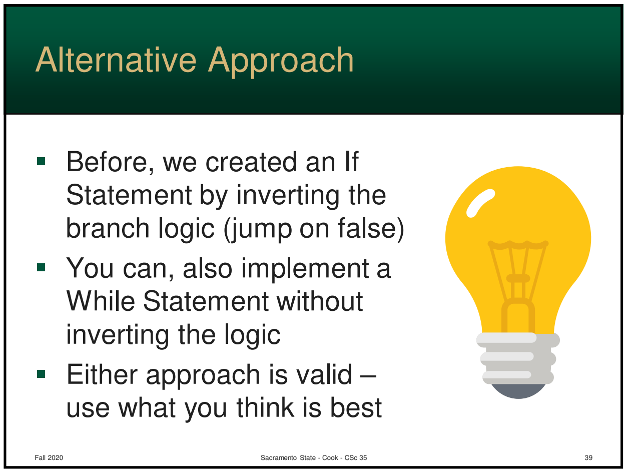


Converting an While Statement

```
 While:
     cmp   rax, 21
     jge   End

     #true block
     jmp   While
 End:
```

Escape infinite loop

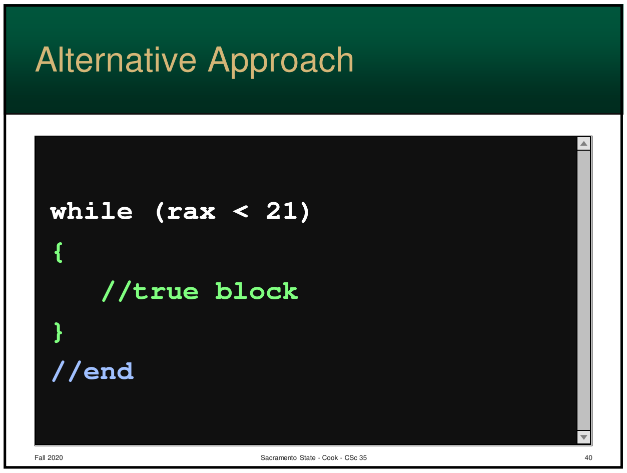
38



Alternative Approach

- Before, we created an If Statement by inverting the branch logic (jump on false)
- You can, also implement a While Statement without inverting the logic
- Either approach is valid – use what you think is best

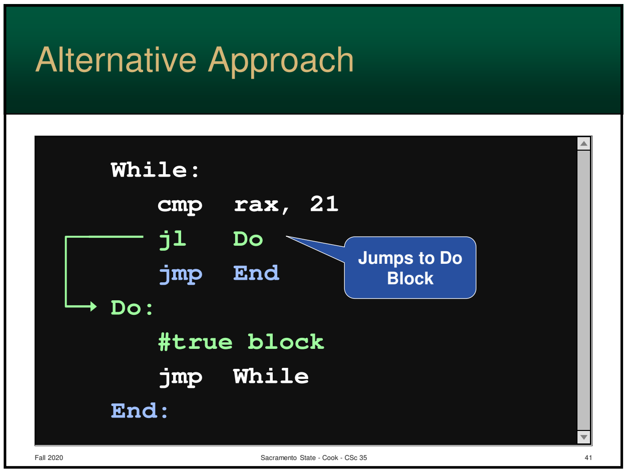
39



Alternative Approach

```
while (rax < 21)
{
   //true block
}
//end
```

40

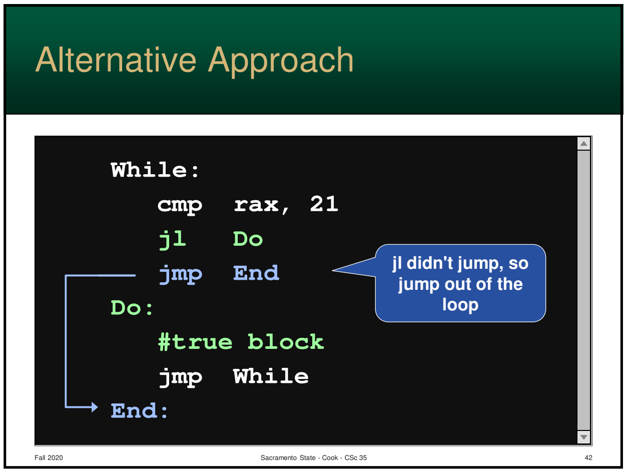


Alternative Approach

```
 While:
     cmp   rax, 21
     jl    Do
     jmp   End
 Do:
     #true block
     jmp   While
 End:
```

Jumps to Do Block

41

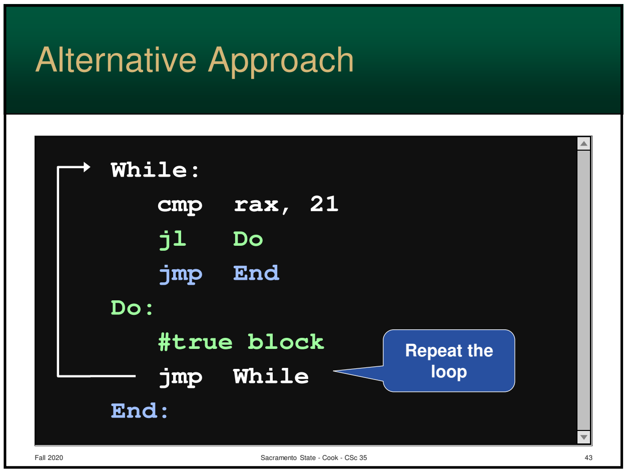


Alternative Approach

```
 While:
     cmp   rax, 21
     jl    Do
     jmp   End
 Do:
     #true block
     jmp   While
 End:
```

jl didn't jump, so jump out of the loop

42

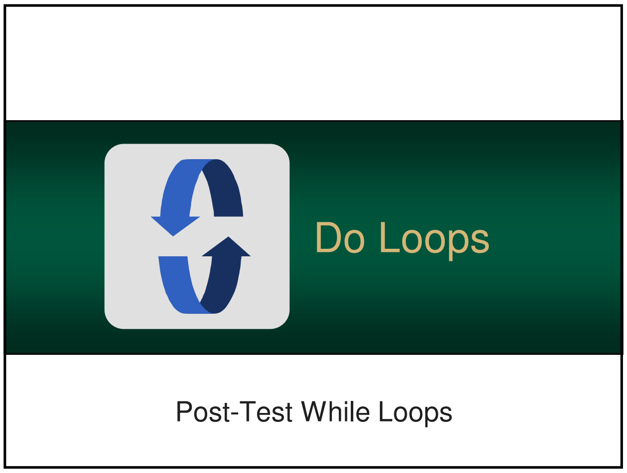


Alternative Approach

```
While:
    cmp   rax, 21
    jl    Do
    jmp   End
Do:
    #true block
    jmp   While
End:
```

Repeat the loop

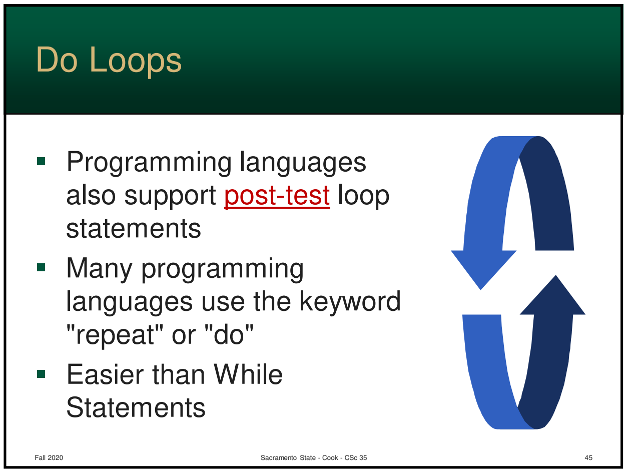
43



Do Loops

Post-Test While Loops

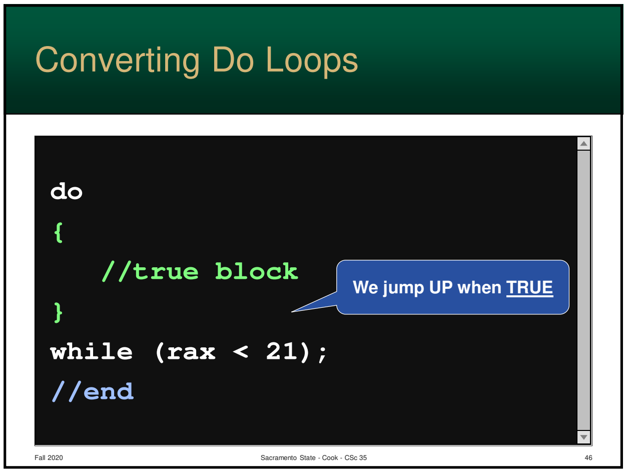
44



Do Loops

- Programming languages also support post-test loop statements
- Many programming languages use the keyword "repeat" or "do"
- Easier than While Statements

45

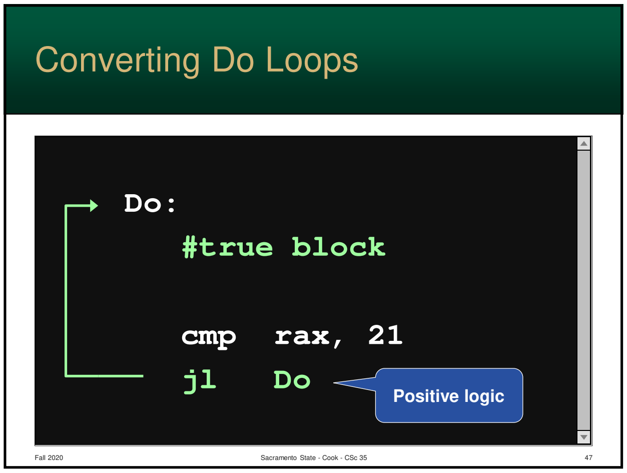


Converting Do Loops

```
do
{
   //true block
}
while (rax < 21);
//end
```

We jump UP when TRUE

46

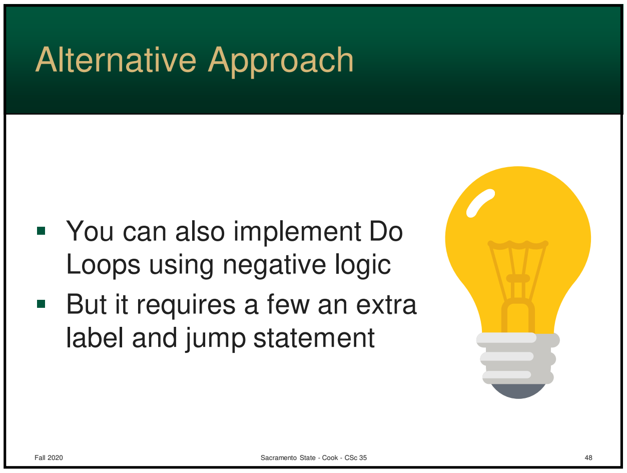


Converting Do Loops

```
Do:
    #true block

    cmp   rax, 21
    jl    Do
```

Positive logic

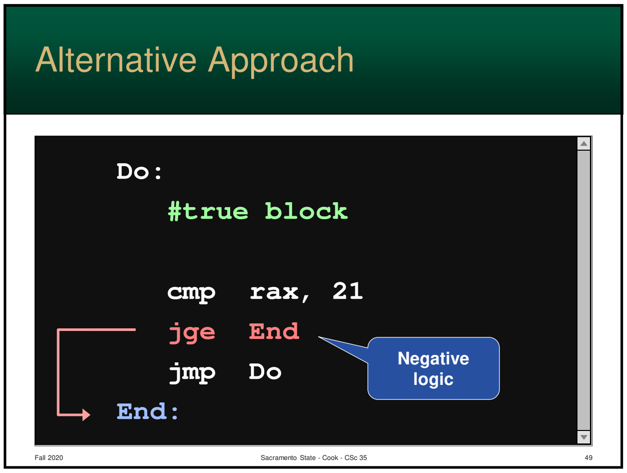
47



Alternative Approach

- You can also implement Do Loops using negative logic
- But it requires a few an extra label and jump statement

48



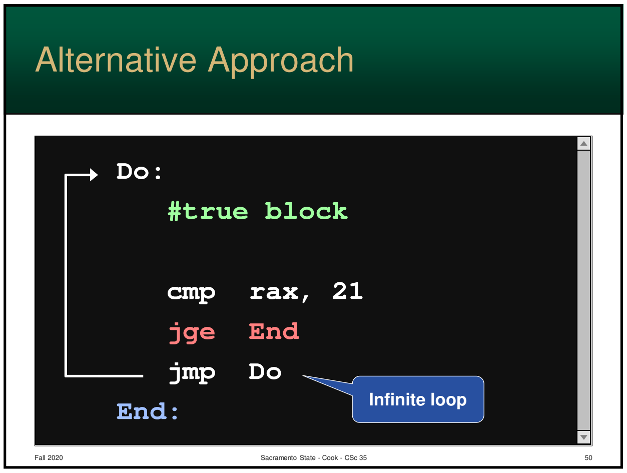
Alternative Approach

```
    Do:
        #true block

        cmp   rax, 21
        jge   End
        jmp   Do
    End:
```

Negative logic

Fall 2020 Sacramento State - Cook - CSc 35 49

49



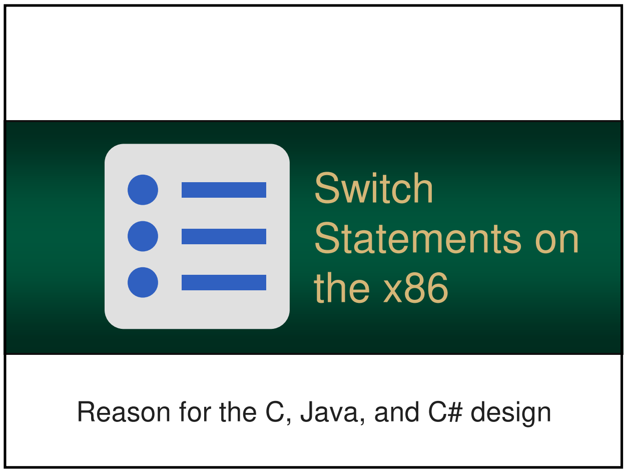
Alternative Approach

```
    Do:
        #true block

        cmp   rax, 21
        jge   End
        jmp   Do
    End:
```

Infinite loop

Fall 2020 Sacramento State - Cook - CSc 35 50

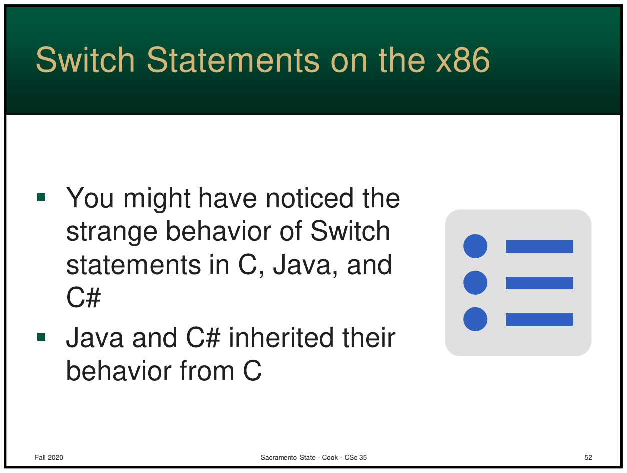
50



Switch Statements on the x86

Reason for the C, Java, and C# design

51

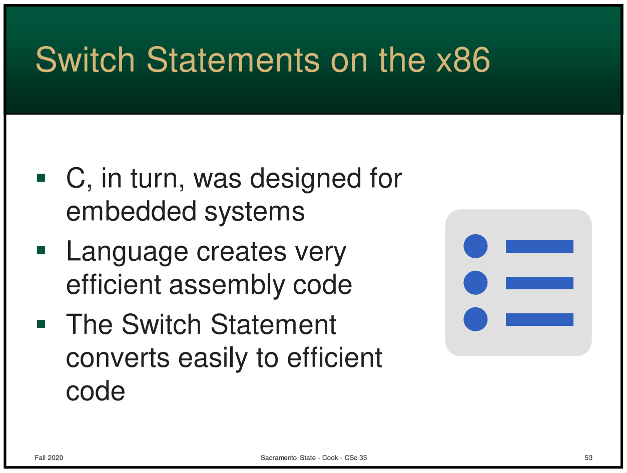


Switch Statements on the x86

- You might have noticed the strange behavior of Switch statements in C, Java, and C#
- Java and C# inherited their behavior from C

Fall 2020 Sacramento State - Cook - CSc 35 52

52

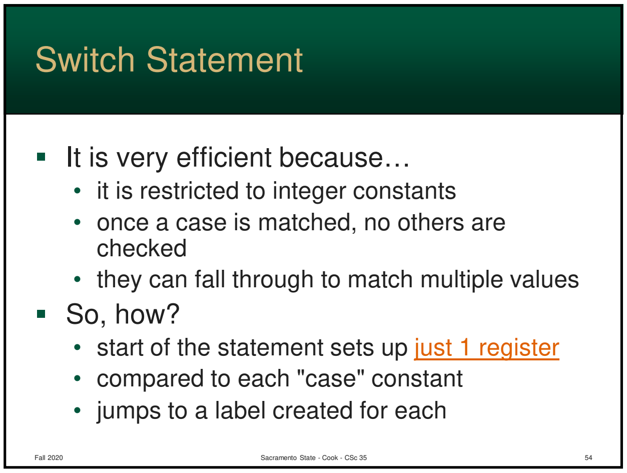


Switch Statements on the x86

- C, in turn, was designed for embedded systems
- Language creates very efficient assembly code
- The Switch Statement converts easily to efficient code

Fall 2020 Sacramento State - Cook - CSc 35 53

53

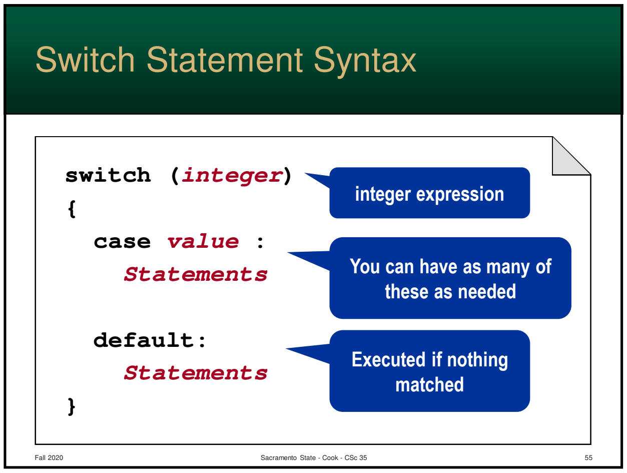


Switch Statement

- It is very efficient because…
  - it is restricted to integer constants
  - once a case is matched, no others are checked
  - they can fall through to match multiple values
- So, how?
  - start of the statement sets up just 1 register
  - compared to each "case" constant
  - jumps to a label created for each

Fall 2020 Sacramento State - Cook - CSc 35 54

54



Switch Statement Syntax

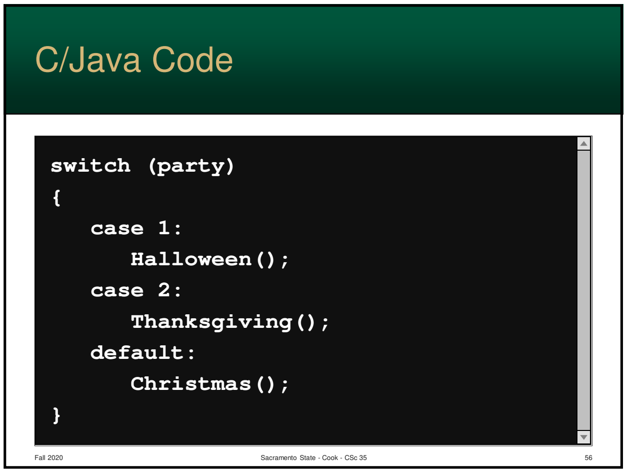
```
switch (integer)
{
   case value :
      Statements

   default:
      Statements
}
```

integer expression

You can have as many of these as needed

Executed if nothing matched

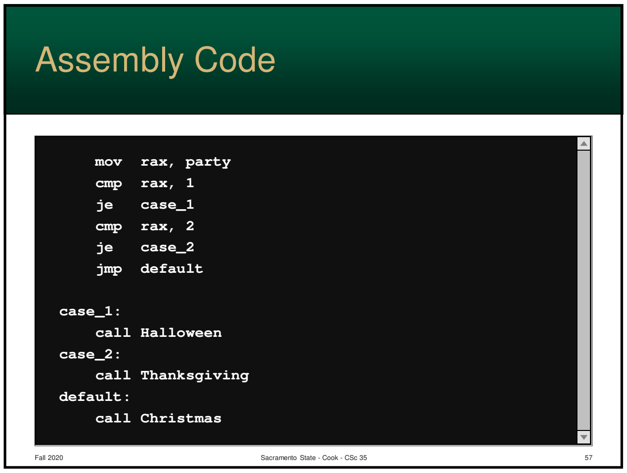
55



C/Java Code

```
switch (party)
{
   case 1:
      Halloween();
   case 2:
      Thanksgiving();
   default:
      Christmas();
}
```

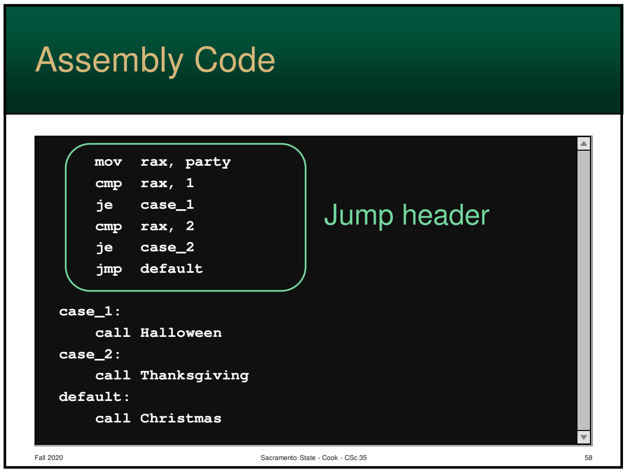
56



Assembly Code

```
    mov  rax, party
    cmp  rax, 1
    je   case_1
    cmp  rax, 2
    je   case_2
    jmp  default

case_1:
    call Halloween
case_2:
    call Thanksgiving
default:
    call Christmas
```

57

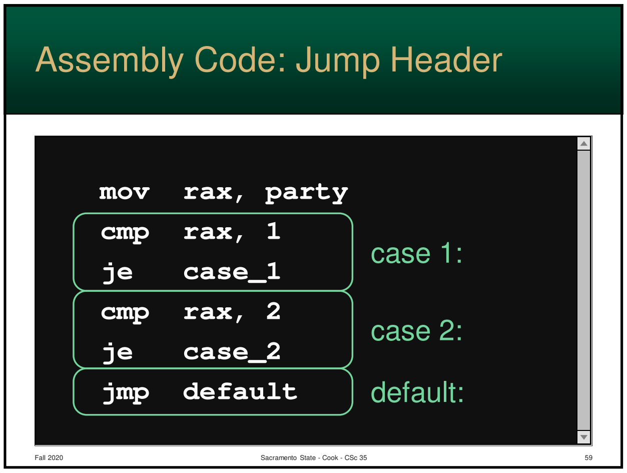


Assembly Code

```
    mov  rax, party
    cmp  rax, 1
    je   case_1
    cmp  rax, 2
    je   case_2
    jmp  default

case_1:
    call Halloween
case_2:
    call Thanksgiving
default:
    call Christmas
```

Jump header

58



Assembly Code: Jump Header

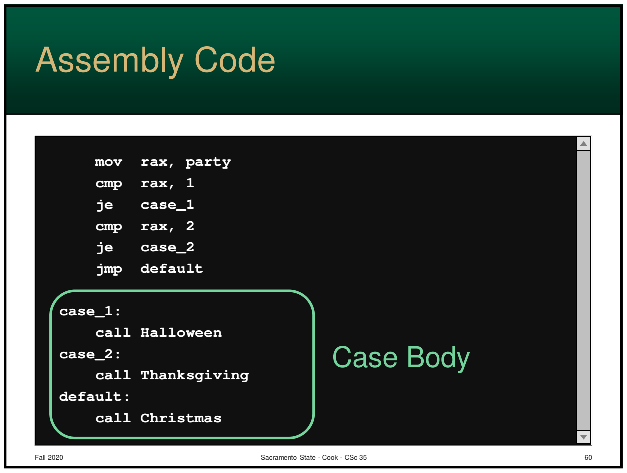
```
mov  rax, party
cmp  rax, 1
je   case_1
cmp  rax, 2
je   case_2
jmp  default
```

case 1:

case 2:

default:

59

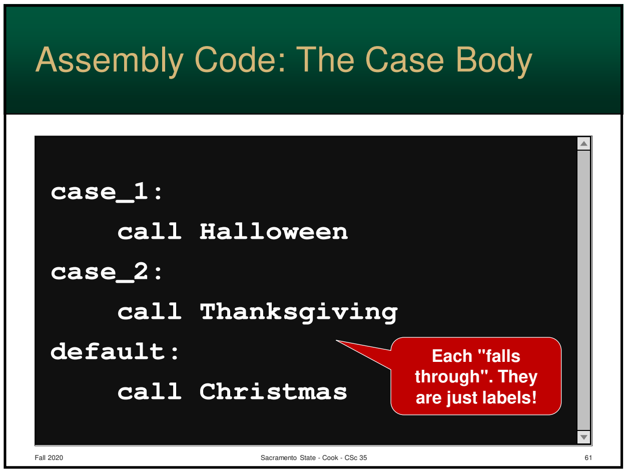


Assembly Code

```
    mov  rax, party
    cmp  rax, 1
    je   case_1
    cmp  rax, 2
    je   case_2
    jmp  default

case_1:
    call Halloween
case_2:
    call Thanksgiving
default:
    call Christmas
```

Case Body

60

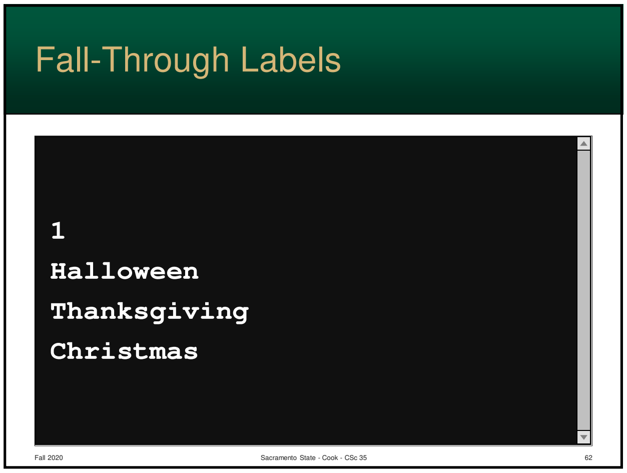


Assembly Code: The Case Body

```
case_1:
    call Halloween
case_2:
    call Thanksgiving
default:
    call Christmas
```

Each "falls through". They are just labels!

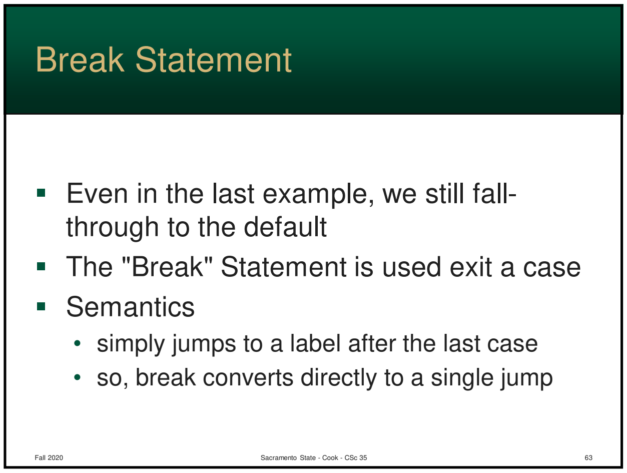
61



Fall-Through Labels

```
1
Halloween
Thanksgiving
Christmas
```

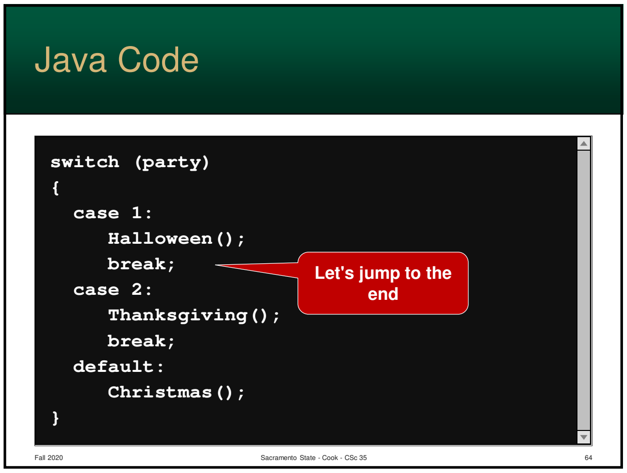
62



Break Statement

- Even in the last example, we still fall-through to the default
- The "Break" Statement is used exit a case
- Semantics
  - simply jumps to a label after the last case
  - so, break converts directly to a single jump

63

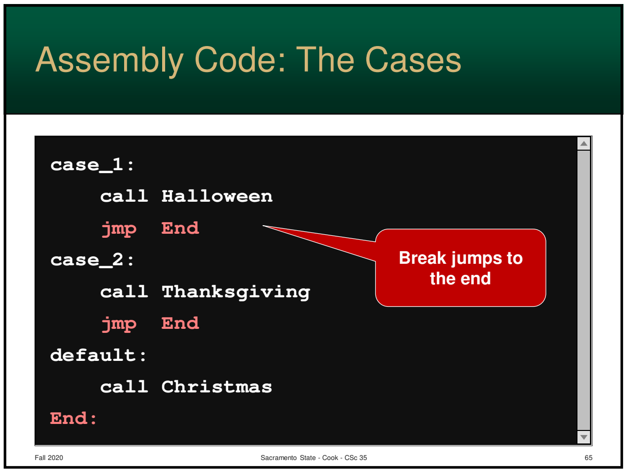


Java Code

```
switch (party)
{
  case 1:
     Halloween();
     break;
  case 2:
     Thanksgiving();
     break;
  default:
     Christmas();
}
```

Let's jump to the end

64

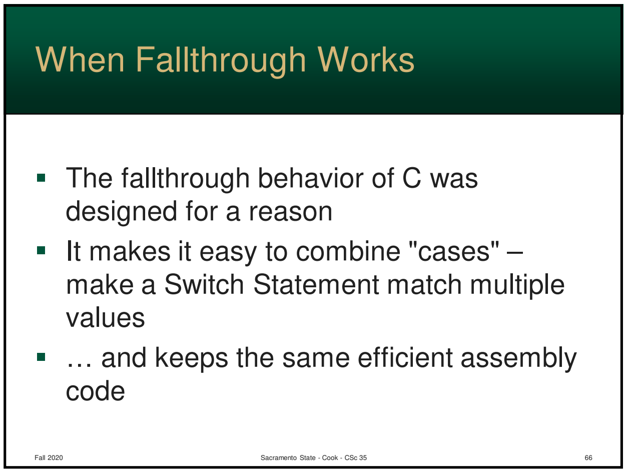


Assembly Code: The Cases

```
case_1:
    call Halloween
    jmp  End
case_2:
    call Thanksgiving
    jmp  End
default:
    call Christmas
End:
```

Break jumps to the end

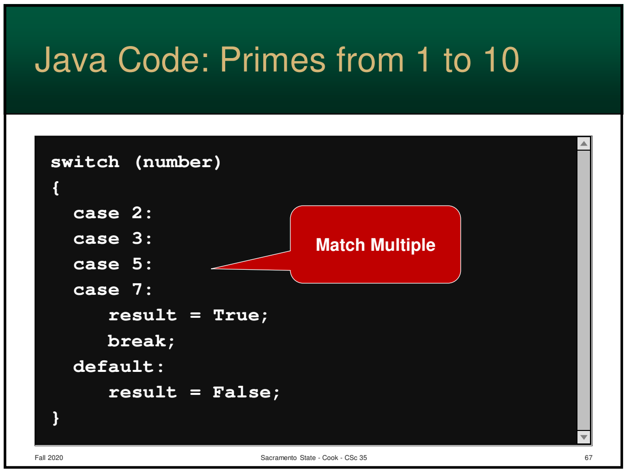
65



When Fallthrough Works

- The fallthrough behavior of C was designed for a reason
- It makes it easy to combine "cases" – make a Switch Statement match multiple values
- … and keeps the same efficient assembly code

66

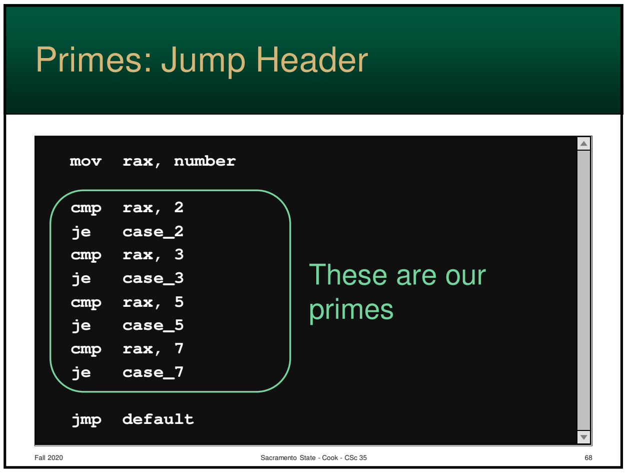


Java Code: Primes from 1 to 10

```
switch (number)
{
  case 2:
  case 3:
  case 5:
  case 7:
     result = True;
     break;
  default:
     result = False;
}
```

Match Multiple

67

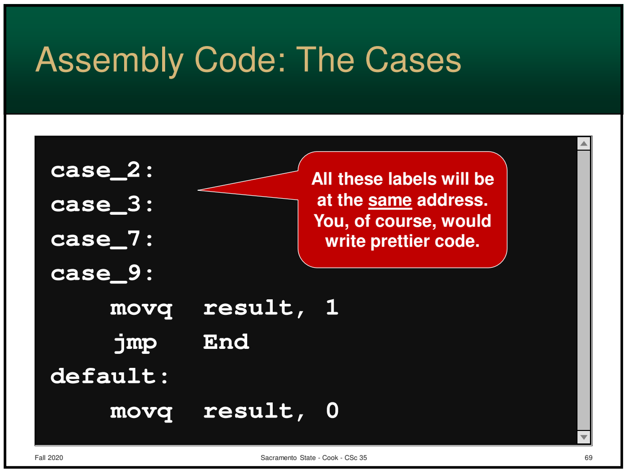


Primes: Jump Header

```
   mov   rax, number

   cmp   rax, 2
   je    case_2
   cmp   rax, 3
   je    case_3
   cmp   rax, 5
   je    case_5
   cmp   rax, 7
   je    case_7

   jmp   default
```

These are our primes

68



Assembly Code: The Cases

```
case_2:
case_3:
case_7:
case_9:
     movq  result, 1
     jmp   End
default:
     movq  result, 0
```

All these labels will be at the same address. You, of course, would write prettier code.

69