




# 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




Spring 2024 Berkeley CS61C - CS61C

3

## Operations: Program Flow Control


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



Spring 2024 Berkeley CS61C - CS61C

4

## Types of Jumps: Unconditional



- *Unconditional jumps* simply 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)

Spring 2024 Berkeley CS61C - CS61C

5

## Instruction: Jump

```
JMP address
```

Usually a label – a constant that holds an address

Spring 2024 Berkeley CS61C - CS61C

6

## Infinite Loop

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

.text
.global _start

_start:
lea rax, message
Loop:
call PrintString
jmp Loop
```

7

## Infinite Loop

```
_start:
lea rax, message
Loop:
call PrintString
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

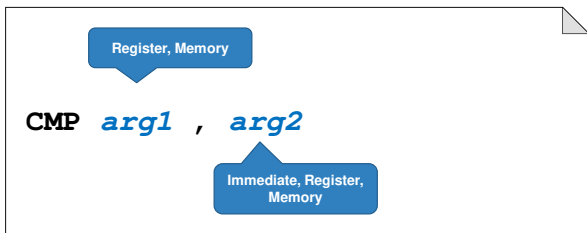
## Instruction: Compare

- 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



11

## Conditional Jumps

- x86 contains a large number of conditional jump instructions
- 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



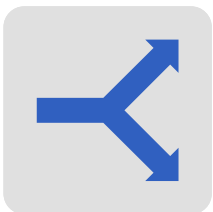
If Statements on the x86

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

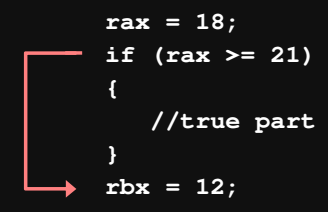
- Converting from an If Statement to assembly is easy
- Let's look at If Statements...
  - block is only executed if the expression is true
  - so, if the expression is false your program will skip over the block
  - this is a jump...

18

## If Statement jumps over code

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

False



19

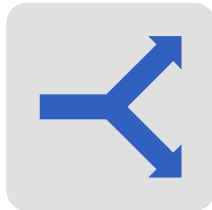
## Converting an If Statement

- 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
  - $a < b \rightarrow \text{not } (a \geq b)$
  - $a \geq b \rightarrow \text{not } (a < b)$

20

## 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

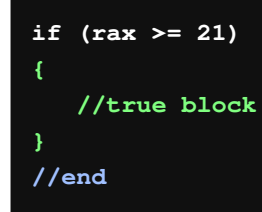


21

## Converting an If Statement

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

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



22

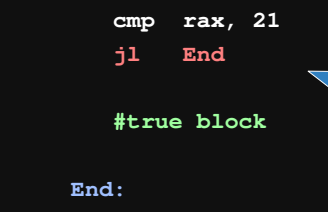
## Jump over true part

```
cmp rax, 21
j1 End

#true block

End:
```

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



23

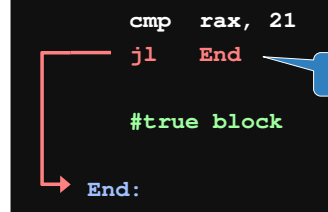
## Jump over true part

```
cmp rax, 21
j1 End

#true block

End:
```

Jumps over true part



24

## 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!*

Spring 2024

Backwards Step - CS4 - CS131

25

25

## Else Clause

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

Spring 2024

Backwards Step - CS4 - CS131

26

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

Spring 2024

Backwards Step - CS4 - CS131

27

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

Spring 2024

Backwards Step - CS4 - CS131

28

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



Spring 2024

Backwards Step - CS4 - CS131

29

29

## If Statement – No Else

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

Jumps to true block

Spring 2024

Backwards Step - CS4 - CS131

30

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

## 3 Rules of Engineering

1. If it works... *it works!*
2. If it ain't broke... *don't fix it!*
3. Reread rules 1 and 2 you moron!



33



## While Loops

Doing the same thing again and again  
... and again

34

## 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



35

## 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 :(

36

## Converting an While Statement

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

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

37

## Converting an While Statement

```
While:
    cmp rax, 21
    jge End

    #true block
    jmp While
End:
```

Loop after  
block executes

38

## Converting an While Statement

```
While:
    cmp rax, 21
    jge End

    #true block
    jmp While
End:
```

Escape infinite  
loop

39

## 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



40

## Alternative Approach

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

41

## Alternative Approach

```
While:
    cmp rax, 21
    jl Do
    jmp End

Do:
    #true block
    jmp While
End:
```

Jumps to Do  
Block

42

### Alternative Approach

```

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

```

!l didn't jump, so jump out of the loop

43

### Alternative Approach


```

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

```

Repeat the loop

44



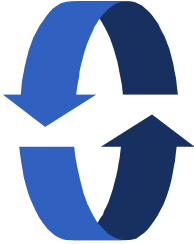
## Do Loops

Post-Test While Loops

45

### Do Loops

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



46

### Converting Do Loops

```

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

```

We jump UP when TRUE

47

### Converting Do Loops

```

Do:
  #true block

  cmp rax, 21
  jl  Do

```


Positive logic

48



## Alternative Approach

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



Spring 2024 Sacramento State - CS&E - CSJ 35 49

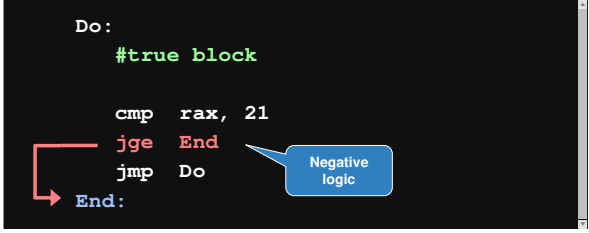
49

## Alternative Approach

```

Do:
  #true block

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



Spring 2024 Sacramento State - CS&E - CSJ 35 50

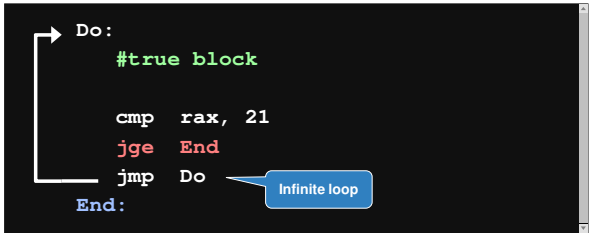
50

## Alternative Approach

```


Do:
  #true block

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



Spring 2024 Sacramento State - CS&E - CSJ 35 51

51




## Addressing & Loops

They were made for each other ... *literally*

52

## Addressing & Loops

- When you use arrays in Java, often the index is a variable
- This allows you to use a For Loop to analyze very element in the array
- This is more common than you think in assembly




Spring 2024 Sacramento State - CS&E - CSJ 35 53

53

## Addressing & Loops

- So, processors allow a register to be used as an index
- This allows you to:
  - copy strings (copying arrays)
  - search through a list
  - and much more...



Spring 2024 Sacramento State - CS&E - CSJ 35 54

54

## For Loop: 0 to 4 - Before

```

.intel_syntax noprefix
.data
Greet:
    .ascii "HELLO"
.text
.global _start
_start:
    
```

Greet	H	0
	E	1
	L	2
	L	3
	O	4

55

## For Loop: 0 to 4

```

mov rax, 0
Loop:
    cmp rax, 4
    jg End
    movb [Greet + rax], 33
    add rax, 1
    jmp Loop
End:
    
```

! character

56

## For Loop: 0 to 4 - After

```

mov rax, 0
Loop:
    cmp rax, 4
    jg End
    movb [Greet + rax], 33
    add rax, 1
    jmp Loop
End:
    
```

Greet	!	0
	!	1
	!	2
	!	3
	!	4

57



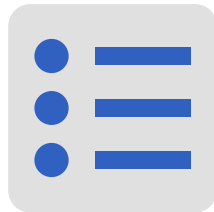
## Switch Statements on the x86

Reason for the C, Java, and C# design

58

## 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



59

## 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



60

## 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

Spring 2024

Beckwith, Stein - C&A - CSU 35

61

61

## 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

Spring 2024

Beckwith, Stein - C&A - CSU 35

62

62

## C/Java Code

```
switch (month)
{
    case 10:
        Halloween();
    case 11:
        Thanksgiving();
    default:
        Christmas();
}
```

Spring 2024

Beckwith, Stein - C&A - CSU 35

63

63

## Assembly Code

```
mov rax, month
cmp rax, 10
je case_10
cmp rax, 11
je case_11
jmp default

case_10:
    call Halloween
case_11:
    call Thanksgiving
default:
    call Christmas
```

Spring 2024

Beckwith, Stein - C&A - CSU 35

64

64

## Assembly Code

```
mov rax, month
cmp rax, 10
je case_10
cmp rax, 11
je case_11
jmp default

case_10:
    call Halloween
case_11:
    call Thanksgiving
default:
    call Christmas
```

Jump header

Spring 2024

Beckwith, Stein - C&A - CSU 35

65

65

## Assembly Code: Jump Header

```
mov rax, month
cmp rax, 10
je case_10
cmp rax, 11
je case_11
jmp default
```

case 10:

case 11:

default:

Spring 2024

Beckwith, Stein - C&A - CSU 35

66

66

## Assembly Code

```
mov rax, month
cmp rax, 10
je case_10
cmp rax, 11
je case_11
jmp default

case_10:
call Halloween
case_11:
call Thanksgiving
default:
call Christmas
```

Case Body

67

## Assembly Code: The Case Body

```
case_10:
    call Halloween
case_11:
    call Thanksgiving
default:
    call Christmas
```

Each "falls through". They are just labels!

68

## Fall-Through Labels

```
10
Halloween
Thanksgiving
Christmas
```

69

## 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

70

## Java Code

```
switch (month)
{
    case 10:
        Halloween();
        break;
    case 11:
        Thanksgiving();
        break;
    default:
        Christmas();
}
```

Let's jump to the end

71

## Assembly Code: The Cases

```
case_10:
    call Halloween
    jmp End
case_11:
    call Thanksgiving
    jmp End
default:
    call Christmas
End:
```

Break jumps to the end

72

## 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

Spring 2024

Benjamin Stein - C&A - CSU 35

73

73

## 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

Spring 2024

Benjamin Stein - C&A - CSU 35

74

74

## 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

Spring 2024

Benjamin Stein - C&A - CSU 35

75

75

## 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.

Spring 2024

Benjamin Stein - C&A - CSU 35

76

76