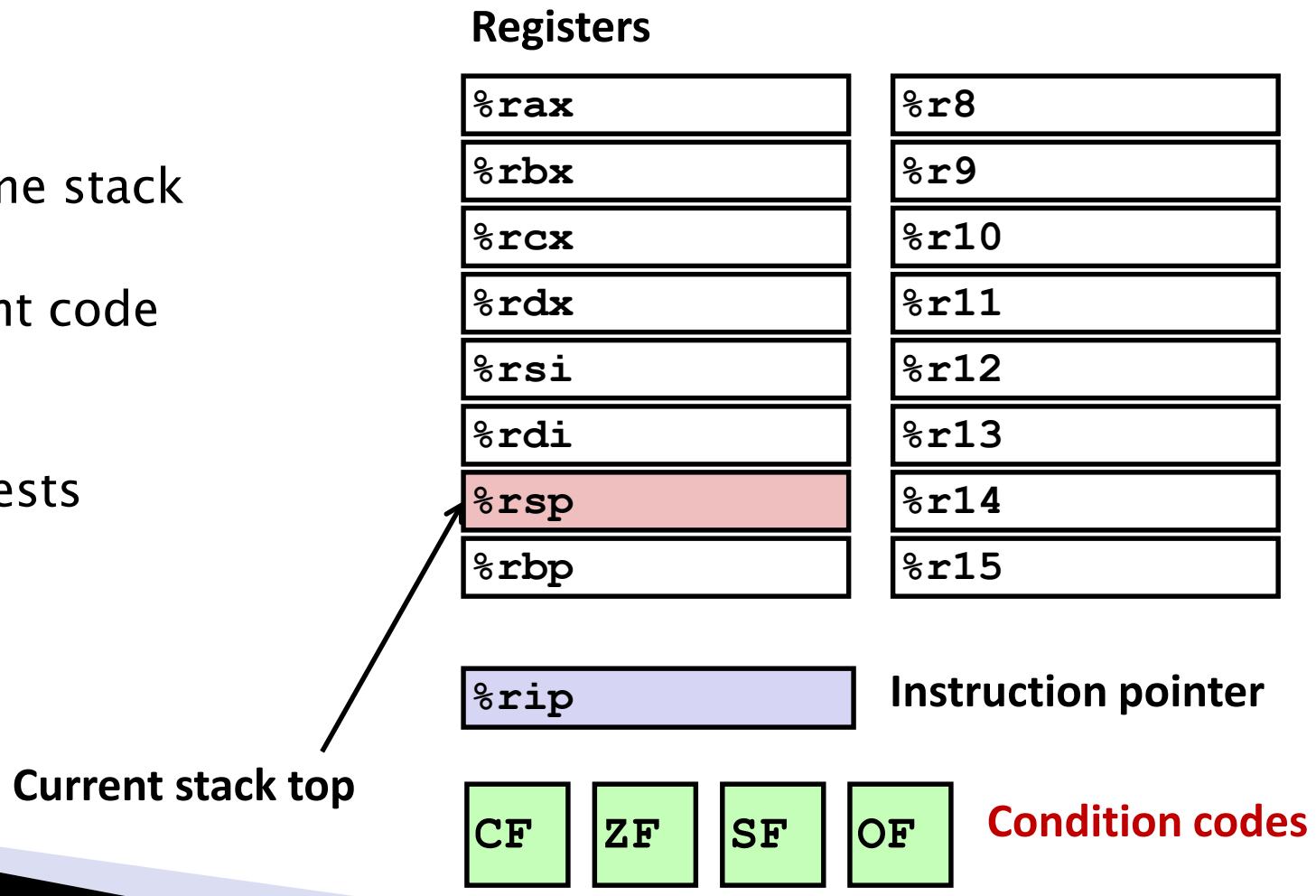
Section 5: Condition codes

Processor State (x86-64, Partial)

- Information about currently executing program
  - Temporary data (**%rax**, …)
  - Location of runtime stack (**%rsp**)
  - Location of current code control point (**%rip**, …)
  - Status of recent tests (**CF, ZF, SF, OF**)



Condition Codes (Implicit Setting)

- Single bit registers
  - **CF** Carry Flag (for unsigned)
  - **SF** Sign Flag (for signed)
  - **ZF** Zero Flag
  - **OF** Overflow Flag (for signed)

- Implicitly set (think of it as side effect) by arithmetic operations

  Example: **`addq`** *Src*,*Dest* ↔ **`t = a+b`**

  CF set if carry out from most significant bit (unsigned overflow)

  ZF set if **`t == 0`**

  SF set if **`t < 0`** (as signed)

  OF set if two's-complement (signed) overflow

  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

- Not set by **`leaq`** instruction – AGU doesn't set codes
- Not set by mov – the data doesn't go through the ALU

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - **`cmpq`** *Src2*, *Src1*
  - **`cmpq b,a`** like computing **`a-b`** without setting destination

  - **CF set** if carry out from most significant bit (used for unsigned comparisons)
  - **ZF set** if **`a == b`**
  - **SF set** if **`(a-b) < 0`** (as signed)
  - **OF set** if two's-complement (signed) overflow

```
(a>0 && b<0 && (a-b)<0) ||
                    (a<0 && b>0 && (a-b)>0)
```

AT&T Syntax, Compare, & Conditionals

- AT&T Syntax is “backwards” to intel syntax
- This is confusing when using the compare instruction and conditional operations:

```
                                # %rdi holds x, %rsi holds y
                                # jump if x > y
    cmpq        %rsi, %rdi      # compare x and y
    jg          some_label          # jump when %rdi > %rsi
```

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - **`testq`** *Src2*, *Src1*
    - **`testq b,a`** like computing **`a&b`** without setting destination
  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful for:
    - repeating the operand to determine if value is negative, zero or positive (e.g. testq %rax %rax)
    - to have one of the operands be a mask to test individual bits (e.g. testq %rax, 0x0100)
  - **ZF set** when **`a&b == 0`**
  - **SF set** when **`a&b < 0`**

Reading Condition Codes

- SetX Instructions (Figure 3.14 in Bryant/O'Hallaron)
  - Set low-order byte of destination(low order single-byte register or a single byte memory location) to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

SetX	Condition	Description
`sete`	`ZF`	Equal / Zero
`setne`	`~ZF`	Not Equal / Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Nonnegative
`setg`	`~(SF^OF)&~ZF`	Greater (Signed)
`setge`	`~(SF^OF)`	Greater or Equal (Signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`seta`	`~CF&~ZF`	Above (unsigned)
`setb`	`CF`	Below (unsigned)

Why? So that you can store a condition longer than one instruction

Reading Condition Codes (Cont.)

- SetX Instructions:
  - Set single byte based on combination of condition codes
- One of addressable byte registers
  - Does not alter remaining bytes
  - Typically use **`movzbq`** to finish job
    - (Figure 3.5 & last 4 paragraphs of 3.4.2)

```
int gt (long x, long y)
{
  return x > y;
}
```

Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

```
    cmpq    %rsi, %rdi    # Compare x:y
    setg    %al           # Set when >
    movzbq  %al, %rax     # Zero rest of %rax
    ret
```

In a nutshell

- Aritmetic
  - cmp <src1> <src2>
    - Computes <src2> – <src1>, but does not save result anywhere
    - Condition codes are set based on the computation
    - src1, and src2 must be of the same size
  - cmpb, cmpw, cmpl or cmpq
- Logical
  - Test <src1> <src2>
    - Computes <src2> & <src1>, but does not save result anywhere
    - Condition codes are set based on the computation
    - src1, and src2 must be of the same size
  - testb, testw, testl, testq

Conditional Moves

- cmovX Instructions
  - Move a value (or not) depending on condition codes

cmovX	Condition	Description
`cmove`	`ZF`	Equal / Zero
`cmovne`	`~ZF`	Not Equal / Not Zero
`cmovs`	`SF`	Negative
`cmovns`	`~SF`	Nonnegative
`cmovg`	`~(SF^OF)&~ZF`	Greater (Signed)
`cmovge`	`~(SF^OF)`	Greater or Equal (Signed)
`cmovl`	`(SF^OF)`	Less (Signed)
`cmovle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`cmova`	`~CF&~ZF`	Above (unsigned)
`cmovb`	`CF`	Below (unsigned)

Appendix: Looking at a simple C Program

- We'll see this again as our first assembler programming homework

Simple C program

- The simple C program below will be translated to assembly language in the following slides:

```
#include <stdio.h>

long x;            /* file scope variable – stored on the heap */

int main () {
    printf("Please enter an integer on the next line, followed by enter:\n“);
    scanf("%i", &x);     /* Get a value from the user */
    x = x + 5;           /* add 5 to the input value */
    printf("The value of x after adding 5 is: %i\n", x);
    return 0;
}
```

x86-64 program

```
.file	"scanPrint.s“		#optional directive
.section		.rodata			#required directives for rodata
.LC0:
	.string	"Please enter an integer on the next line, followed by enter:\n"
.LC1:
	.string	"%i"
.LC2:
	.string	"The value of x after adding 5 is: %i\n"

.data				#required for file scope data: read-write program data
	#of static storage class
x:
	.quad 0

.globl main			#required directive for every function
	.type	main, @function		#required directive
```

Code for main

```
.text                                   #required directive
main:
        pushq   %rbp                    #stack housekeeping #1
        movq    %rsp, %rbp                      #stack housekeeping #2
        movq    $.LC0, %rdi                     #address of string “Please enter…:\n“ to %rdi
                                        # %rdi is location of 1st parameter not pushing any caller saved
                                        # registers because there is no valuable data there
        movq    $0, %rax                # C library ABI says %rax should be zero b4 call to printf
        call    printf
        movq    $x, %rsi                #mov the address of x to %rsi (2nd parameter)
        movq    $.LC1, %rdi                     #address of string “%i” in %rdi (1st parameter)
        movq    $0, %rax                # to keep ABI happy
        call    scanf
        addq    $5, x                   #add the constant 5 to what is stored in variable x
        movq    x, %rsi                 #value of x to %rsi (2nd parameter)
        movq    $.LC2, %rdi                     #address of string “The value of…” to %rdi (1st param)
        movq    $0, %rax                # keep ABI happy
        call    printf
        movq    $0, %rax                #set return value to 0
        leave
        ret
        .size   main, .-main            #required directive
```