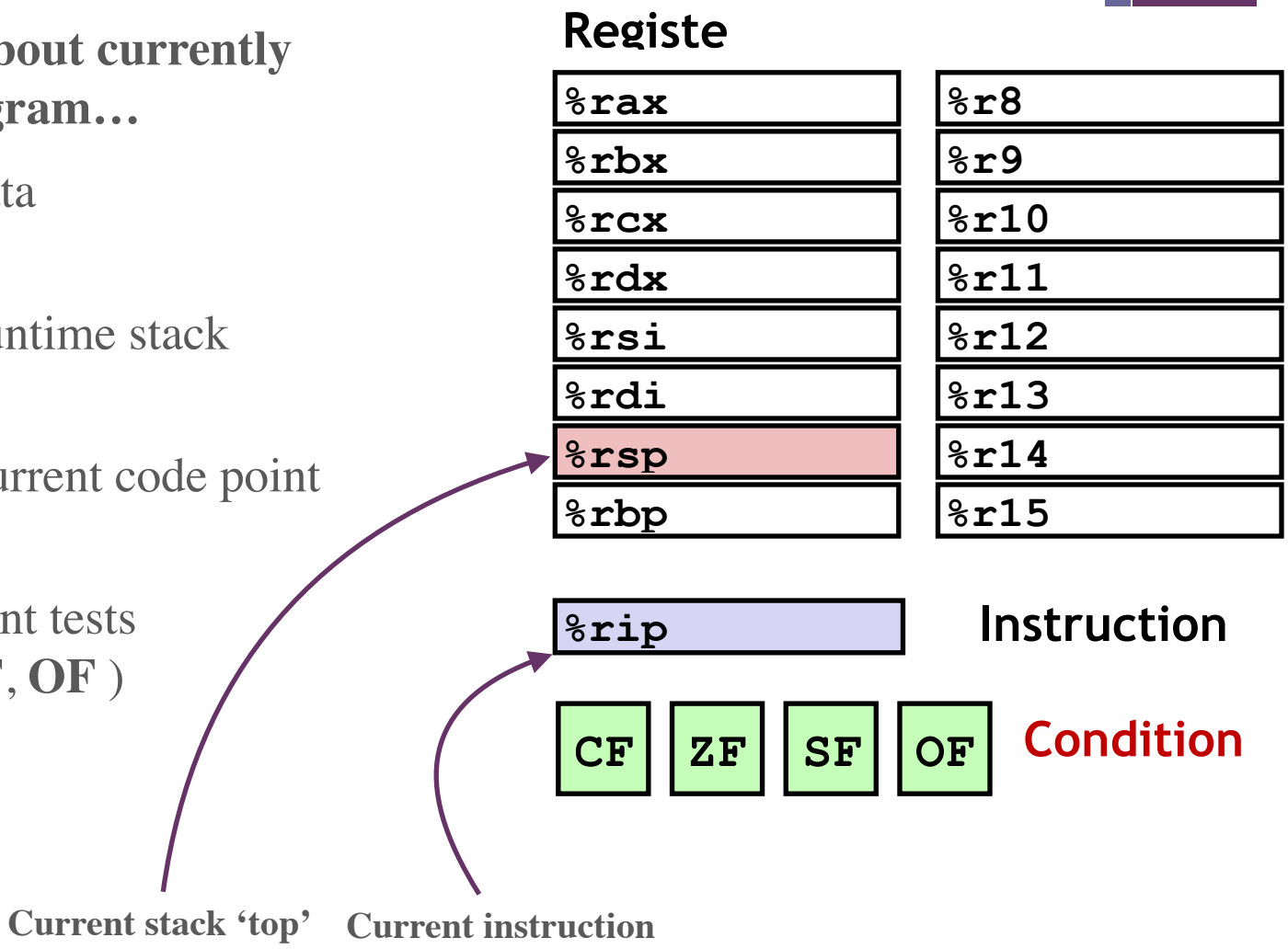
+

Control & Condition Codes

Processor State (x86-64, Partial)

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



Condition Codes (Implicit Setting)

- **Single bit registers**
  - **CF** Carry Flag (for unsigned)
  - **ZF** Zero Flag
  - **SF** Sign Flag (for signed)
  - **OF** Overflow Flag (for signed)
- **Implicitly set (think of it as a side effect) by arithmetic operations**
  - Example: **addq** ***src*****,** ***dest*** ↔ b = 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 (!!!)**

Condition Codes (Explicit Setting)

- **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)

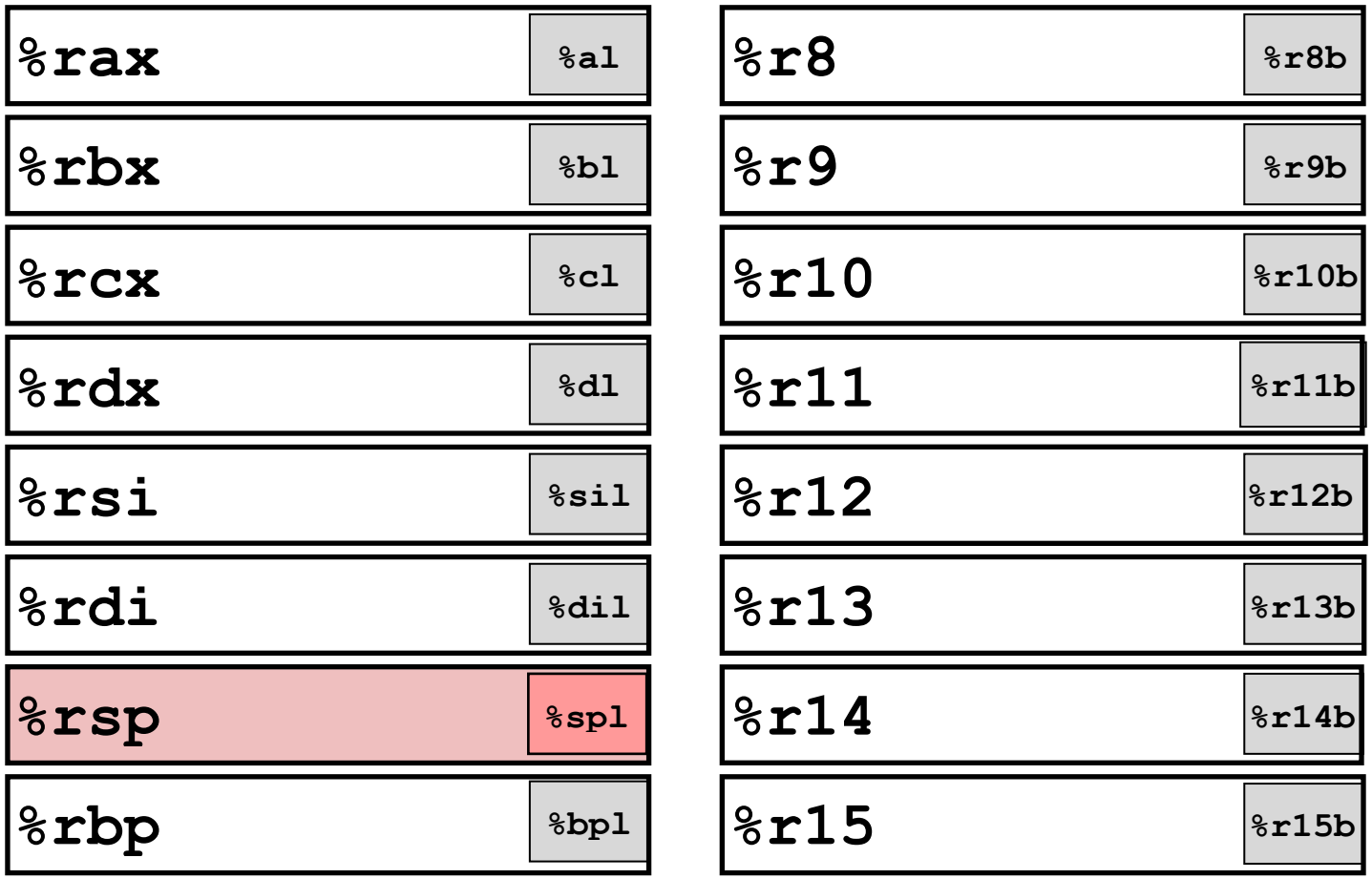
- Only purpose of this instruction is to set condition codes!
- There are other instructions like this.

Reading Condition Codes

- **SetX Instructions**
  - Set low-order byte of destination 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)

x86-64 Integer Registers



- Can reference low-order byte.

Reading Condition Codes *Con't*

- **SetX instructions:**
  - Set single byte based on combination of condition codes
- **One of addressable byte registers**
  - Does not alter remaining bytes
  - Typically use movzbl to finish job
    - 32-bit instructions also set upper 32 bits to 0

```
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 and y
    setg    %al           # Set %al 'on' when x > y
    movzbl  %al, %rax     # Copy and zero rest of %rax
    ret
```

+

Conditional Branches

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

Conditional Branching by Jumping

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x - y;
    else
        result = y - x;
    return result;
}
```

```
absdiff:
    cmpq    %rsi, %rdi # y, x
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:                   # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    ret
```

- Note: must use ***–fno-if-conversion*** argument to gcc, otherwise assembly will not use jumps in this program, we'll see why in a moment.

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

Rewriting C with `goto` Statements

- C allows **`goto`** statement
- Jump to position designated by label…

```
long absdiff(long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

```
long absdiff_j(long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    return result;
 Else:
    result = y-x;
    return result;
}
```

- **`goto`** form resembles assembly instructions using jumps

Rewriting C with `goto` Statements *con't*

- C code

```
val = test ? then_expr : else_expr;
```

- Example

```
val = x > y ? x - y : y - x;
```

- Goto version

```
  if (!test) goto Else;
  val = then_expr;
  goto Done:
Else:
  val = else_Expr;
Done:
  return val;
```

- Create separate code regions for then & else expressions
- Execute appropriate one
- This is how we can think about 'jumping' in assembly

Alternate Approach: Conditional Moves

- **Conditional Move Instructions**
  - Instruction supports:
    - if (test) dest <- src
  - GCC tries to use them, but, only when known to be safe
- **Why?**
  - Branches are very disruptive to instruction flow through pipelines
  - Conditional moves do not require control transfer

C Code

```
val = test ? then_exp : else_exp;
```

Goto Version

```
result = then_expr;
eval = else_expr;
neg_test = !test;
if (neg_test) result = eval;
return result;
```

Conditional Move Example

```
long absdiff(long x, long y){
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value
%rdx	Temp variable

```
absdiff:
   movq    %rdi, %rax   # x
   subq    %rsi, %rax   # if-val = x-y
   movq    %rsi, %rdx
   subq    %rdi, %rdx   # else-val = y-x
   cmpq    %rsi, %rdi   # %rsi = y, %rdi = x
   cmovle  %rdx, %rax   # if y <= x, result = else-val
ret
```

Bad Cases for Conditional Move

- **Expensive computations**

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

  - Both values get computed
  - Only makes sense when computations are very simple

- **Risky computations**

```
val = p == 0 ? 0 : 5 / p;
```

  - Both values get computed
  - May have undesirable effects

- **Computations with side effects**

```
val = x > 0 ? x *= 7 : x += 3;
```

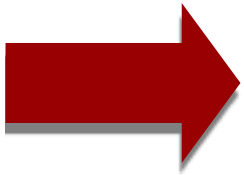
  - Both values get computed
  - Must be side-effect free

Loops

General "Do-While" Translation

C Code

```
do
   Body
   while (Test);
```



Goto Version

```
loop:
   Body
   if (Test)
      goto loop
```

Body

```
{
   statement1;
   statement2;
      …
   statementn;
}
```

"Do-While" Loop Example

C Code

```
long pcount_do(unsigned long x)
{
  long result = 0;
  do {
    result += x & 0x1;
    x >>= 1;
  } while (x);
  return result;
}
```

Goto Version

```
long pcount_goto(unsigned long x)
{
  long result = 0;
 loop:
  result += x & 0x1;
  x >>= 1;
  if(x) goto loop;
  return result;
}
```

- Count number of 1's in argument x ('popcount')
- Use conditional jump to either continue looping or to exit loop

"Do-While" Loop Compilation

```
long pcount_goto(unsigned long x){
   long result = 0;
 loop:
   result += x & 0x1;
   x >>= 1;
   if(x) goto loop;
   return result;
}
```

Register	Use(s)
%rdi	Argument x
%rax	result

```
   movl     $0, %rax          #  result = 0
.L2:                          # loop:
   movq     %rdi, %rdx
   andq     $1, %rdx          #  t = x & 0x1
   addq     %rdx, %rax        #  result += t
   shrq     %rdi              #  x >>= 1
   jne      .L2               #  if (x) goto loop
   rep; ret
```

- Note: some processors' branch predictors behave badly when a branch's target or fall-through is a **ret** instruction, and adding the **rep;** prefix avoids this.

General "While" Translation

- "Jump-to-middle" translation
- Used with **gcc -Og** (our setting)

While Version

```
while (Test)
  Body
```

Goto Version

```
  goto test;
loop:
  Body
test:
  if (Test)
    goto loop;
done:
```

While Loop Example

C Code

```
long pcount_while(unsigned long x)
{
  long result = 0;
  while (x) {
    result += x & 0x1;
    x >>= 1;
  }
  return result;
}
```

Jump to Middle

```
long pcount_goto_jtm(unsigned long x)
{
  long result = 0;
  goto test;
 loop:
  result += x & 0x1;
  x >>= 1;
 test:
  if(x) goto loop;
  return result;
}
```

- Compare to do-while version of function
- Initial goto starts loop at test

For Loop: Derived From While

For Version

```
for (Init; Test; Update )
   Body
```

While Version

```
Init;

while (Test) {
   Body
   Update;
}
```

For-While Conversion

```
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned long x)
{
  int i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

```
long pcount_for_while(unsigned long x)
{
  size_t i;
  long result = 0;
  i = 0;
  while (i < WSIZE)
  {
    unsigned bit = (x >> i) & 0x1;
    result += bit;
    i++;
  }
  return result;
}
```

Init

```
i = 0
```

Test

```
i < WSIZE
```

Update

```
i++
```

Body

```
{
  unsigned bit =
     (x >> i) & 0x1;
  result += bit;
}
```

"For" Loop Do-While Conversion

```
long pcount_for(unsigned long x)
{
  size_t i;
  long result = 0;
  for (i = 0; i < WSIZE; i++)
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  return result;
}
```

```
long pcount_for_goto_dw(unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;
  goto TEST
 LOOP:
  {
    unsigned bit =
      (x >> i) & 0x1;
    result += bit;
  }
  i++;
 TEST:
  if (i < WSIZE)
    goto LOOP;
 DONE:
  return result;
}
```

Init

Test *(jump to middle)*

Body

Update

Test

Initial test may be optimized away if compiler knows its safe