Machine-Level Programming: Control

CSCI 237: Computer Organization
9th Lecture, Feb 28, 2025

Jeannie Albrecht

Administrative Details

- Lab 2: first three phases due Tue/Wed
  - Intermediate "results" automatically collected (no need to submit)
  - Submit defuser.txt and short writeup on Glow when finished all 5 phases
  - Any questions so far??
- HW 2 due today, HW 3 due next Friday
- Finally, congrats to winners! ☺

1	2	3	4	5	6	7	8	9	10	11	12	Winner?	Score	Nickname
3	4	7	9	11	4	6	3	8	5	2	14	Winner!	20	Nathan
3	4	7	9	11	4	6	3	10	5	2	16	Winner!	16	ben
3	**4**	**7**	**16**	**11**	**5**	**7**	**3**	**12**	**5**	**4**	**19**		0	**jeannie**
3	4	7	9	11	*7*	6	3	*14*	5	—	—		—	nickname2
3	*6*	7	9	—	4	—	3	—	—	—	—		—	Saul Goodman

Last Time:
Machine Programming: Ops & Control

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
- Intro to data-dependent control

Recap:
Complete Memory Addressing Modes

- Most General Form

  D(Rb,Ri,S) → Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant "displacement" **(no restrictions!)**
  - Rb: Base register Reg[Rb]: Any of 16 integer registers
  - Ri: Index register Reg[Ri]: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8

- Special Cases

  (Rb,Ri) → Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) → Mem[Reg[Rb]+Reg[Ri]+D]

  (Rb,Ri,S) → Mem[Reg[Rb]+S*Reg[Ri]]

Today:
Machine Programming: Ops & Control

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
- Intro to data-dependent control

Arithmetic Operations

- Two Operand Instructions:

Format		Computation	
`addq`	Src, Dest	Dest = Dest + Src	
`subq`	Src, Dest	Dest = Dest – Src	
`imulq`	Src, Dest	Dest = Dest * Src	
`salq`	Src, Dest	Dest = Dest << Src	(Also called `shlq`)
`sarq`	Src, Dest	Dest = Dest >> Src	Arithmetic right shift
`shrq`	Src, Dest	Dest = Dest >> Src	Logical right shift
`xorq`	Src, Dest	Dest = Dest ^ Src	
`andq`	Src, Dest	Dest = Dest & Src	
`orq`	Src, Dest	Dest = Dest | Src	

- Remember argument order! *Src, Dest*
  (Warning (again): Intel docs use "op *Dest,Src*")
- No distinction between signed and unsigned int in x86

Arithmetic Operations

- One Operand Instructions

`incq`	Dest	Dest = Dest + 1
`decq`	Dest	Dest = Dest – 1
`negq`	Dest	Dest = – Dest
`notq`	Dest	Dest = ~Dest

- See book for a more complete list!

Arithmetic Expression Example

```
long arith (long x,
            long y,
            long z){
  long t1 = x + y;
  long t2 = z + t1;
  long t3 = x + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax
  addq    %rdx, %rax
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx
  leaq    4(%rdi,%rdx), %rcx
  imulq   %rcx, %rax
  ret
```

Instructions

- **`leaq`**: address computation
- **`salq`**: left shift
- **`imulq`**: multiplication
  - Only used once

Arithmetic Expression Example

```
long arith (long x,
            long y,
            long z){
  long t1 = x + y;
  long t2 = z + t1;
  long t3 = x + 4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith:
  leaq    (%rdi,%rsi), %rax    # t1
  addq    %rdx, %rax           # t2
  leaq    (%rsi,%rsi,2), %rdx
  salq    $4, %rdx             # t4
  leaq    4(%rdi,%rdx), %rcx   # t5
  imulq   %rcx, %rax           # rval
  ret
```

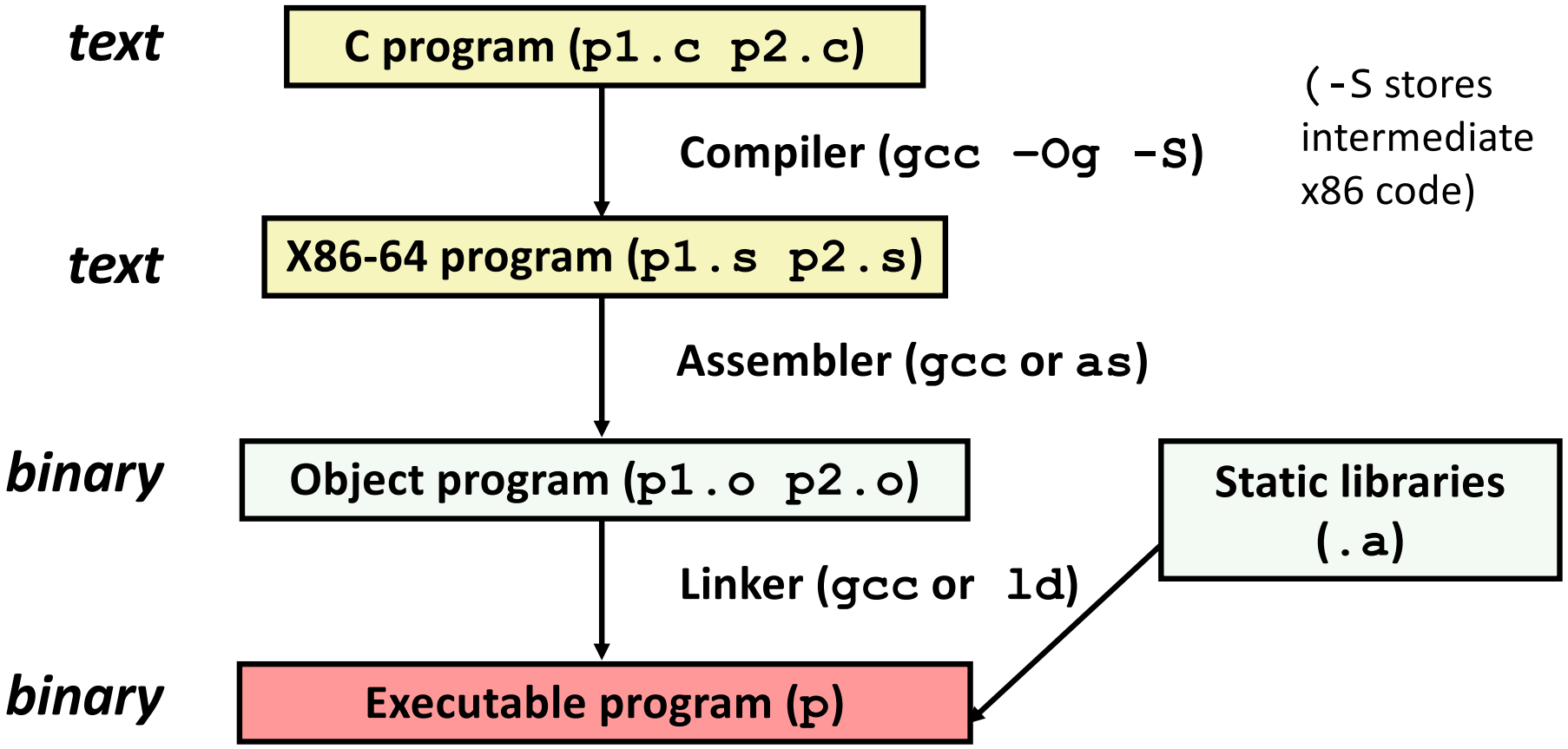
Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rdx	Argument **z, t4**
%rax	**t1**, **t2**, **rval**
%rcx	**t5**

Today:
Machine Programming: Ops & Control

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
- Intro to data-dependent control

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc –Og p1.c p2.c -o p`
  - `–Og` : Use "general optimizations" (but nothing too crazy)
  - `-o p` : Put resulting output binary in file `./p` (`a.out` by default)



Compiling Into Assembly

C Code (sum.c)

```
long plus(long x, long y);

void sumstore(long x, long y,
              long *dest) {
    long t = plus(x, y);
    *dest = t;
}
```

Generated x86-64 Assembly

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

Obtain (on lab machines) with command

```
gcc -Og -S sum.c
```

Produces file `sum.s`

Warning: May get slightly different results on other (non-lab) machines due to different versions of gcc and different compiler settings.

What it really looks like

```
 .file   "sum.c"
        .text
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB0:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   sumstore, .-sumstore
        .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

What it really looks like

```
 .file   "sum.c"
        .text
        .globl  sumstore
        .type   sumstore, @function
sumstore:
.LFB0:
        .cfi_startproc
        pushq   %rbx
        .cfi_def_cfa_offset 16
        .cfi_offset 3, -16
        movq    %rdx, %rbx
        call    plus
        movq    %rax, (%rbx)
        popq    %rbx
        .cfi_def_cfa_offset 8
        ret
        .cfi_endproc
.LFE0:
        .size   sumstore, .-sumstore
        .ident  "GCC: (Ubuntu 4.8.4-2ubuntu1~14.04.3) 4.8.4"
        .section        .note.GNU-stack,"",@progbits
```

Things that look weird and are preceded by a '.' are generally directives.
(We can usually ignore them!)

```
sumstore:
    pushq   %rbx
    movq    %rdx, %rbx
    call    plus
    movq    %rax, (%rbx)
    popq    %rbx
    ret
```

Object Code

Code for `sumstore`

```
0x04004ed:
    0x53
    0x48
    0x89
    0xd3
    0xe8
    0x05
    0x00
    0x00
    0x00
    0x48
    0x89
    0x03
    0x5b
    0xc3
```

- **Total of 14 bytes**
- **Each instruction 1, 3, or 5 bytes**
- **Starts at address `0x04004ed`**

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
  - Can use –c flag to gcc to generate .o files
- Linker
  - Resolves references between files
  - Combines with *static* run-time libraries
    - E.g., code for **`malloc, printf`**
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Machine Instruction Example

- C Code

```
*dest = t;
```

  - Store value **t** where designated by **dest**

- Assembly

```
movq %rax, (%rbx)
```

  - Move 8-byte value to memory
    - Quad word in x86-64 terms
  - Operands:

    t: Register **%rax**

    dest: Register **%rbx**

    ***dest:** Memory **M[%rbx]**

- Object Code

```
0x4004ee:  48 89 d3
```

  - 3-byte instruction
  - Stored at address **0x4004ee**

Disassembling Object Code

Disassembled

```
0000000004004ed <sumstore>:
  4004ed:  53                          push   %rbx
  4004ee:  48 89 d3                    mov    %rdx,%rbx
  4004f1:  e8 05 00 00 00              callq  4004fb <plus>
  4004f6:  48 89 03                    mov    %rax,(%rbx)
  4004f9:  5b                          pop    %rbx
  4004fa:  c3                          retq
```

- Disassembler

  `objdump -d sum`

  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either `a.out` (complete executable) or `.o` file

Operations Summary

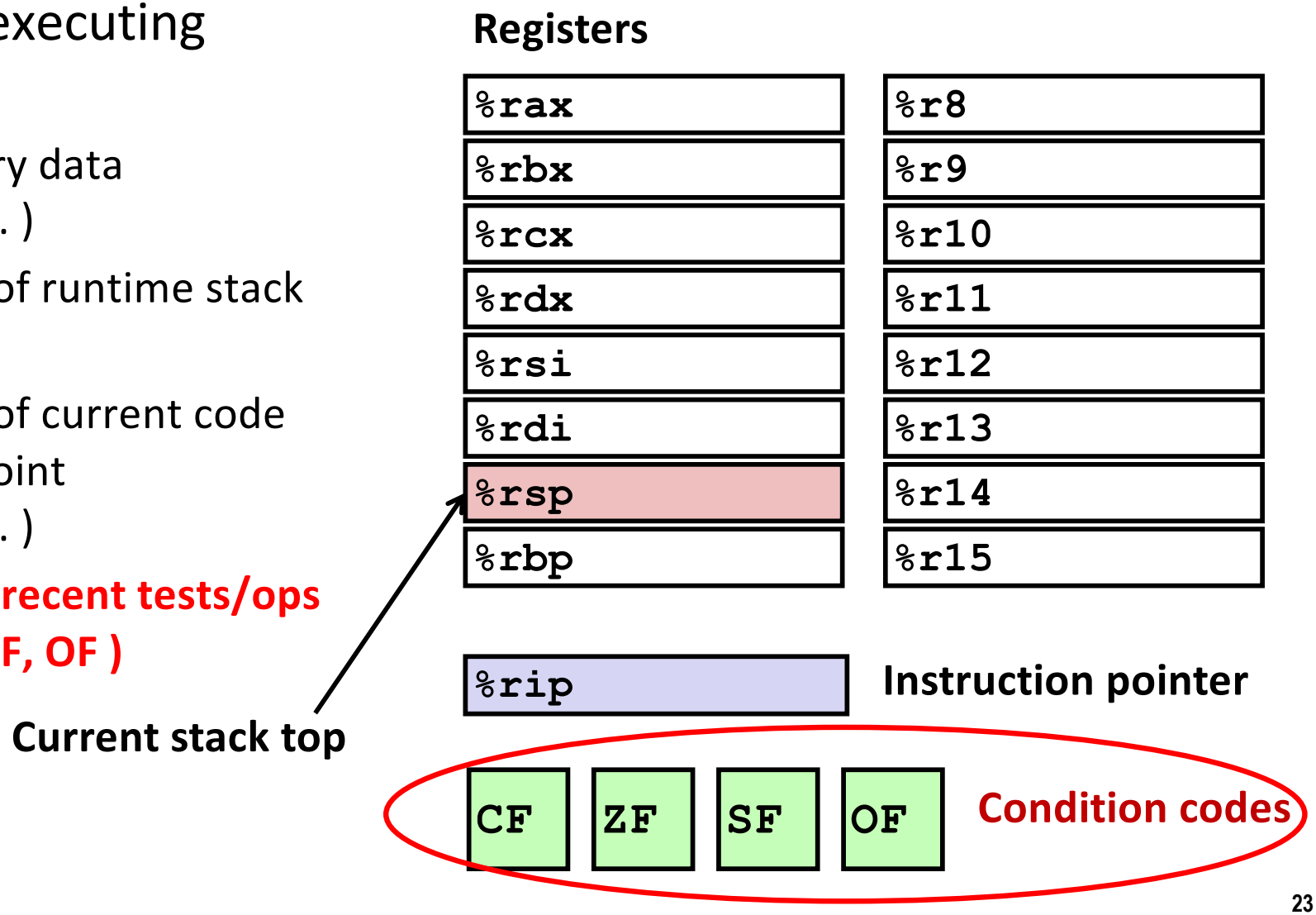
- History of Intel processors and architectures
  - Evolutionary design leads to many quirks and artifacts
- Assembly Basics: Registers, operands, move
  - The x86-64 move instructions cover wide range of data movement forms
- Arithmetic
  - C compiler will figure out different instruction combinations to carry out computation
- C, assembly, machine code
  - New forms of visible state: program counter, registers, ...
  - Compiler must transform statements, expressions, procedures into low-level instruction sequences

Today:
Machine Programming: Ops & Control

- History of Intel processors and architectures
- Assembly Basics: Registers, operands, move
- Arithmetic & logical operations
- C, assembly, machine code
- Intro to data-dependent control

Data dependent control (Ch 3.6): 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/ops (CF, ZF, SF, OF)**



Reading Condition Codes

- Condition codes are extra bits that summarize the results of operations and affect the execution of later instructions
  - Set automatically during execution
- Three ways to "access" condition codes in assembly
  1. Operations that set a byte to 0/1 based on some combination of the condition codes

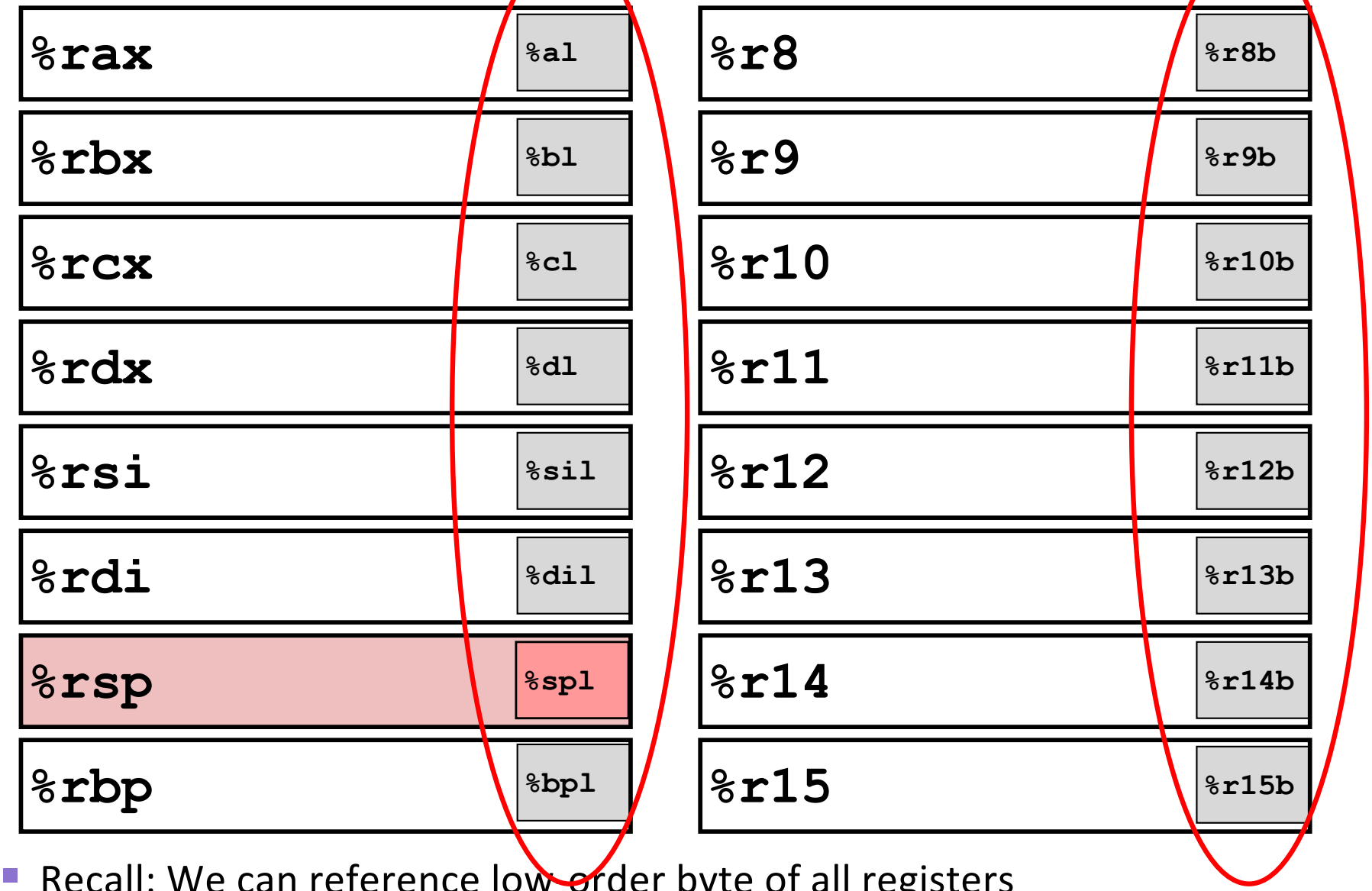
     `setX Dest` instructions
  1. Operations that "jump" to some part of program based on condition codes
  2. Operations that transfer (or move) data only if some condition codes are set

Reading Condition Codes

- SetX Instructions
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Set instructions do not alter remaining 7 bytes in register!

SetX	Condition	Description
sete	**ZF** (zero flag)	Equal / Zero
setne	**~ZF**	Not Equal / Not Zero
sets	**SF** (sign flag)	Negative
setns	**~SF**	Nonnegative
setg	**~(SF^OF) & ~ZF** (overflow flag)	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** (carry flag)	Below (unsigned)

x86-64 Integer Registers



- Recall: We can reference low-order byte of all registers

Reading Condition Codes (Cont.)

- Set Instructions: **`setX dest`**
  - Set single byte based on combination of condition codes
  - Often used with `cmp` instruction (compare)
- Dest is one of addressable *single byte* registers
  - Does not alter remaining bytes of register
  - 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;
}
```

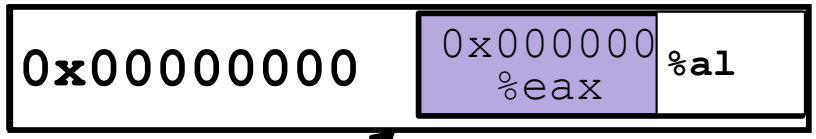
Register	Use(s)
`%rdi`	Argument **`x`**
`%rsi`	Argument **`y`**
`%rax`	Return value

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

Reading Condition Codes (Cont.)

Beware of movzbl (and others like it)

```
movzbl %al, %eax
```



Zapped to all 0's

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

Data-Dependent Control

- Condition codes
- Conditional branches and moves
- Loops
- Switch Statements

Jumping

- Jump Instructions: **`jX address`**
  - 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 Branch Example (with jumps)

- Generation

```
> gcc -Og -S -fno-if-conversion absdiff.c
```

I'll come back to this.

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

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

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

Conditional Branch Example (with jumps)

- Generation

```
> gcc -Og -S -fno-if-conversion absdiff.c
```

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

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

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

Expressing with Goto Code

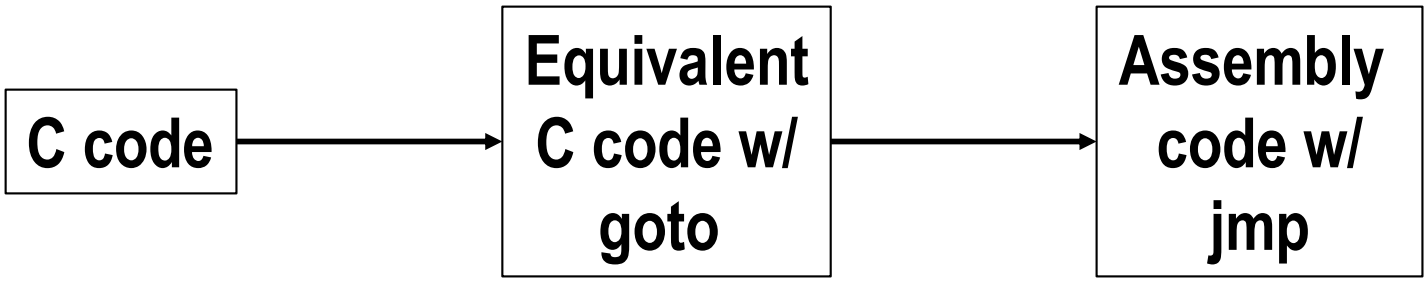
- C allows **`goto`** statements (typically considered very bad programming style!)
- Jump to (or goto) position designated by label
- Goto version of C code is similar to x86-64 with jumps

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

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

jX Is a Powerful Tool

- We can convert many C control flow constructs to equivalent C code that contains goto statements
- We can convert C code with goto statements into x86-64 with **jX**
- We will look closer at this mapping



- But first, we will talk about conditional moves

General Conditional Expression Translation (Using Branches/Jumps)

C Code (w/ ternary operator)

```
val = Test ? val_if_true : val_if_false;
```

```
val = Test ? Then_Expr : Else_Expr;
```

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

Goto Version

```
  ntest = !Test;
  if (ntest) goto Else;
  val = Then_Expr;
  goto Done;
Else:
  val = Else_Expr;
Done:
  . . .
```

- Create separate code regions for then & else expressions
- Execute appropriate one

General Conditional Expression Translation (Using Conditional Moves)

- Conditional Move Instructions: `cmovX src, dest`
  - Instruction supports:

    `if (Test) Dest ← Src`
  - Supported in post-1995 x86 processors
  - GCC tries to use them
    - But, only when known to be safe
- Benefits of cmov vs jump
  - Branches (jumps) are very disruptive to instruction flow through pipelines
  - Relies on good *prediction* for good performance
  - Conditional moves do not require control transfer

C Code

```
val = Test
      ? Then_Expr
      : Else_Expr;
```

"Goto" Version

```
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```

Conditional Move Example

Generation

```
> gcc -O1 -S absdiff.c
```

Notice the compile flags!

```
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

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

Bad Cases for Conditional Move

Expensive Computations

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

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

Bad Performance

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed
- **May have undesirable effects**

Unsafe

Computations with side effects

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

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

Illegal

Compile flags

- If-else statements executed using conditional branches or moves
  - `-Og -S -fno-if-conversion`
    - Used to generate **conditional branch** code (with jumps)
    - –fno-if-conversion not always needed, but often is
  - –O1
    - Used to generate **conditional move** code (no jumps)