CS207: File Processing Term I/2018-19

Lecture 15:

A Pragmatic View of Computer Systems

x64, x86-64 and friends

Sunsern Cheamanunkul

Mahidol University
International College

Today's Topic

- C to Assembly
- Why?
  - We can understand why some implementation is faster than others, despite having the same big-*O*.
  - Get yourself ready for OS.
- Goal: Be able to read assembly code

RECAP:
Everything is bits!
Different types just view it differently.

Everything is bits!

- Seen many data types so far:
  - Integers:
    - char/short/int/long (encoding as unsigned or two's complement signed)
  - Letters/punctutation/etc:
    - char (ASCII encoding)
  - Real numbers:
    - float/double (IEEE floating point encoding, didn't discuss)
  - Memory addresses:
    - pointer types (unsigned long encoding)
  - **Now a new one…..the code itself!**
    - **Instructions**

WHAT HAPPENS WHEN WE COMPILE CODE?

anatomy of an executable file

What happens when we compile code?

```
int sum_array(int arr[], int nelems) {
    int sum = 0;
    for (int i = 0; i < nelems; i++)
{
        sum += arr[i];
    }
    return sum;
}
```

```
gcc sum.c
```


Using objdump

Name of the function (same as in C) and the memory address where the code for this function begins.

Assembly code: "human-readable" version of instructions

```
00000000004004ed <sum_array>:
4004ed:   85 f6                  test   %esi,%esi
4004ef:   7e 17                  jle    400508 <sum_array+0x1b>
4004f1:   ba 00 00 00 00         mov    $0x0,%edx
4004f6:   b8 00 00 00 00         mov    $0x0,%eax
4004fb:   03 04 97               add    (%rdi,%rdx,4),%eax
4004fe:   48 83 c2 01            add    $0x1,%rdx
400502:   39 d6                  cmp    %edx,%esi
400504:   7f f5                  jg     4004fb <sum_array+0xe>
400506:   f3 c3                  repz retq
400508:   b8 00 00 00 00         mov    $0x0,%eax
40050d:   c3                     retq
```

Mem address where each of line of instruction is found---laid out sequentially in memory.

Machine code: rendered as raw hexadecimal, as read by the computer.

Let's focus on the instructions

$[number] means a constant value (this is the number 1)

```
4004fb:   03 04 97               add    (%rdi,%rdx,4),%eax
4004fe:   48 83 c2 01            add    $0x1,%rdx
400502:   39 d6                  cmp    %edx,%esi
400504:   7f f5                  jg     4004fb
```

Operation name (sometimes called "opcode")

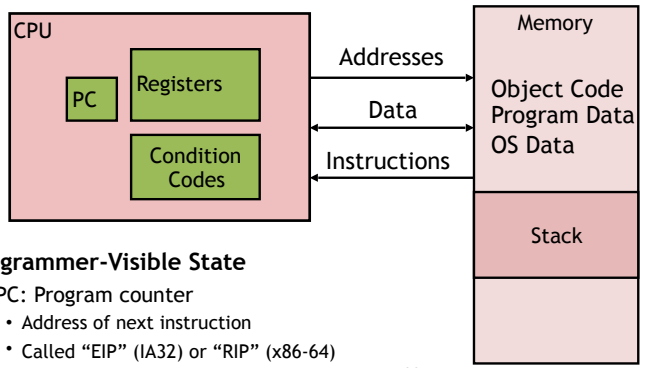
Operands (like... function arguments)

%[name] names a register— these are a small collection of memory slots right on the CPU that can hold variables' values

```
00000000004004ed <sum_array>:
4004ed:   85 f6                  test   %esi,%esi
4004ef:   7e 17                  jle    400508 <sum_array+0x1b>
4004f1:   ba 00 00 00 00         mov    $0x0,%edx
4004f6:   b8 00 00 00 00         mov    $0x0,%eax
4004fb:   03 04 97               add    (%rdi,%rdx,4),%eax
4004fe:   48 83 c2 01            add    $0x1,%rdx  <-- %rdx = %rdx + 1
400502:   39 d6                  cmp    %edx,%esi
400504:   7f f5                  jg     4004fb <sum_array+0xe>
400506:   f3 c3                  repz retq
400508:   b8 00 00 00 00         mov    $0x0,%eax
40050d:   c3                     retq
```

```
int sum_array(int arr[], int nelems) {   Guess?
      int sum = 0;                       // (A)
      for (int i = 0; i < nelems; i++) {  // (B)
            sum += arr[i];                // (C)
      }
      return sum;
}
```


Remember: Registers and Memory?



- **Programmer-Visible State**
  - PC: Program counter
    - Address of next instruction
    - Called "EIP" (IA32) or "RIP" (x86-64)
  - Register file
    - Heavily used program data
  - **Memory**
    - Byte addressable array
    - Code, user data, (some) OS data
    - Includes stack used to support procedures

Copying Data: The *Mov* Instructions

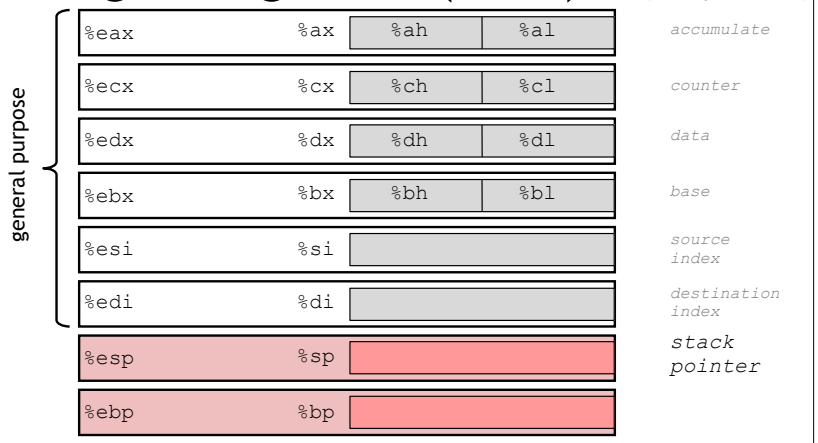
- A main job of assembly language is to manage data:
  - Data can be on the CPU (in registers) or in memory (at an address)
    - Turns out this distinction REALLY MATTERS for performance
  - Often want to move data:
    - Move from one place in memory to another
    - Move from one register to another
    - Move from memory to register
    - Move from register to memory
  - Introducing the "mov" family of instructions

Let's welcome all the x86-64 Registers

Super fast, but there are so few!

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

Integer Registers (IA32)



HOW TO TRANSFER DATA?

Data Copying (despite the name)

- **Moving Data**
  - `movq` *Source, Dest*
  - Other suffix (b,w,l,q): movb, movw, movl, movq
- **Operand Types**
  - *Immediate:* Constant integer data
    - Example: `$0x400`, `$-533`
    - Like C constant, but prefixed with `'$'`
    - Encoded with 1, 2, or 4 bytes
  - *Register:* One of 16 integer registers
    - Example: `%rax`, `%r13`
    - But `%rsp` reserved for special use
    - Others have special uses for particular instructions
  - *Memory:* 8 consecutive bytes of memory at address given by register
    - Simplest example: `(%rax)`
    - Various other "addressing modes"

`%rax` `%rcx` `%rdx` `%rbx` `%rsi` `%rdi` `%rsp` `%rbp` `%rN`

Warning: Intel docs use mov *Dest, Source*

`movq` Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Basic Addressing Mode

- One major difference between high-level code and assembly instructions: ***the absence of programmer-chosen, descriptive variable names:***

```
int total = sugar + coffee;
Vs. addl 8(%rbp), %eax
```

- We don't get to choose variable names; we have to talk directly about places in hardware
- **"Addressing modes" are allowable ways of naming these places**

Simple Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

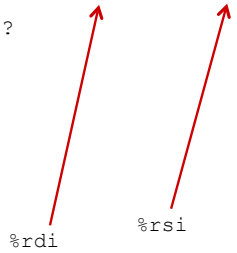
```
movq (%rcx),%rax
```

- Displacement D(R) Mem[Reg[R]+D]
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```


Example of Simple Addressing Modes

```
void
whatAmI(<type> a, <type> b)
{
      ????
}
```




```
whatAmI:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

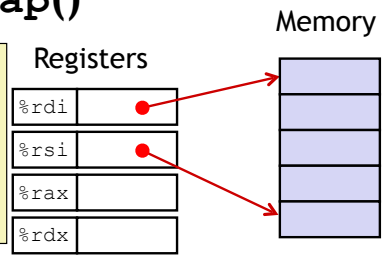

Example of Simple Addressing Modes

```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

```
swap:
    movq    (%rdi), %rax
    movq    (%rsi), %rdx
    movq    %rdx, (%rdi)
    movq    %rax, (%rsi)
    ret
```

Understanding Swap()

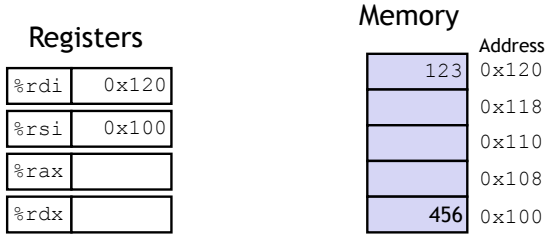
```
void swap
   (long *xp, long *yp)
{
  long t0 = *xp;
  long t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```



Register	Value
%rdi	xp
%rsi	yp
%rax	t0
%rdx	t1

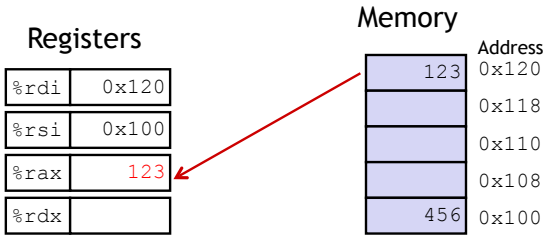
```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```


Understanding Swap()



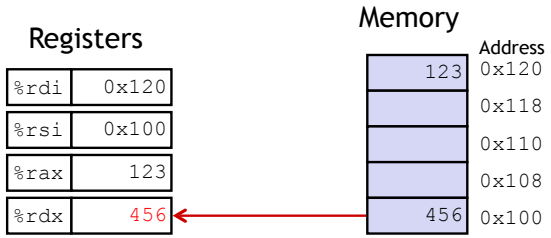

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```


Understanding Swap()



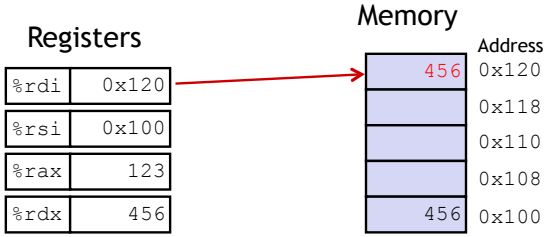

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```


Understanding Swap()



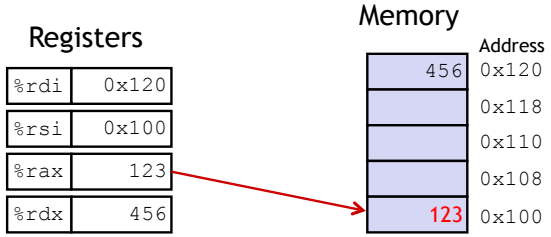

```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```

Understanding Swap()




```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```


Understanding Swap()




```
swap:
   movq    (%rdi), %rax  # t0 = *xp
   movq    (%rsi), %rdx  # t1 = *yp
   movq    %rdx, (%rdi)  # *xp = t1
   movq    %rax, (%rsi)  # *yp = t0
   ret
```


Complete Memory Addressing Modes

Most General Form

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

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

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]]

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

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

- D: Constant "displacement" 1, 2, or 4 bytes
- Rb: Base register: Any of 16 integer registers
- Ri: Index register: Any, except for `%rsp`
- S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

Address Computation Instruction

- `leaq` *Src*, *Dst*
  - *Src* is address mode expression
  - Set *Dst* to address denoted by expression
- Uses
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
- Example

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax  # t = x+2*x
salq $2, %rax             # return t<<2
```


Some 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*
`shrq`	*Src*,*Dest*	Dest = Dest >> Src	*Logical*
`xorq`	*Src*,*Dest*	Dest = Dest ^ Src	
`andq`	*Src*,*Dest*	Dest = Dest & Src	
`orq`	*Src*,*Dest*	Dest = Dest | Src	

- Watch out for argument order! *Src*,*Dest*
  (Warning: Intel docs use "op *Dest*,*Src*")
- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

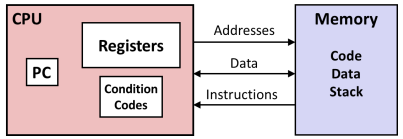
- One Operand Instructions

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

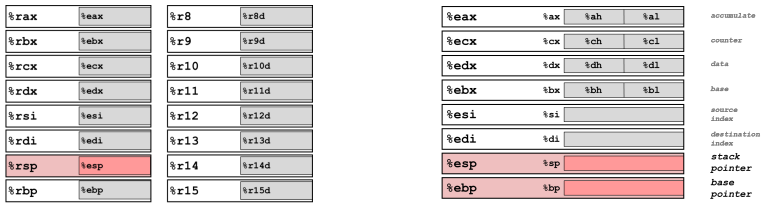
COMPARISON

Recap: x86-64 Architecture

Abstract view



X86-64 Integer Register File



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)

Registers

`%rax`	`%r8`
`%rbx`	`%r9`
`%rcx`	`%r10`
`%rdx`	`%r11`
`%rsi`	`%r12`
`%rdi`	`%r13`
`%rsp`	`%r14`
`%rbp`	`%r15`

`%rip` Instruction pointer

Current stack top

`CF` `ZF` `SF` `OF` Condition codes

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 (as side effect) of 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)

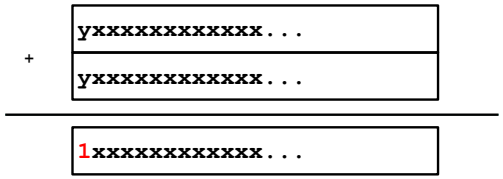
  OF set if two's-complement (signed) overflow

  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- Not set by `leaq` instruction

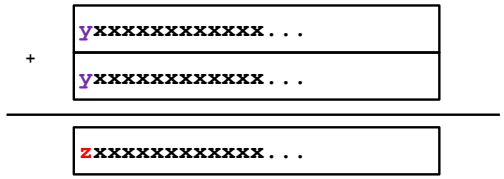
CF set when

```
  1xxxxxxxxxxxx...
+ 1xxxxxxxxxxxx...
------------------
1 xxxxxxxxxxxxx...
```

SF set when

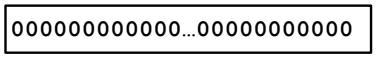


OF set when



z = ~y

ZF set when



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)`

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 to have one of the operands be a mask
- ZF set when `a&b == 0`
- SF set when `a&b < 0`

Very often:

```
testq   %rax,%rax
```


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)

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 `movzbl` to finish job
  - 32-bit instructions also set upper 32 bits to 0

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

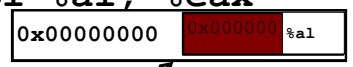
Register	Use(s)
`%rdi`	Argument `x`
`%rsi`	Argument `y`
`%rax`	Return value

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


Reading Condition Codes (Cont.)

Beware weirdness `movzbl` (and others)

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



Zapped to all 0's

Use(s)
Argument `x`
Argument `y`
Return value

```
setg    %al           # Set when >
movzbl  %al, %eax     # Zero rest of %rax
ret
```

Instruction Pointer (%rip)

- The instruction that follows after the current instruction
- %rip advances as the processor works through more instructions.

- How do we alter the course of execution? Example: If?, Loop? Etc. etc.
- The family of *jump* commands.

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 Branch Example (Old Style)

- Generation

```
hamachi0> gcc -Og -S -fno-if-conversion cor
```

will get to this shortly.

```
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` 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;
    goto Done;
 Else:
    result = y-x;
 Done:
    return result;
}
```

General Conditional Expression Translation (Using Branches)

C Code

```
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

Using Conditional Moves

Conditional Move Instructions

- Instruction supports:
  if (Test) Dest ← Src
- Supported in post-1995 x86 processors
- 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_Expr
    : Else_Expr;
```

Goto Version

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

```
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 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

Exercise

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)

- `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)`

```
xor     %rax, %rax
sub     $1, %rax
cmp     $2, %rax
setl    %al
movzbl %al, %eax
```

%rax	SF	CF	OF	ZF

In-Class Lab: Parity

- **Write a function which takes an integer and returns**
  - 1 if there are an odd number of '1' bits
  - 0 if there are an even number of '1' bits

```
int parity_check(int x){
    …
}
```

- **Any ideas?**

Parity Example

- **Inspiration:**
  - If we could XOR all of the bits in the argument… we would get the answer!

1101100101100011111001010010 1101

1101100101100011111001010010 1101

XOR 1101100101100011
1110010100101101

0011110001001110 (down to 16 bits)

Parity Example

- **Just keep going!**

0011110001001110

00111100 01001110

XOR 00111100
01001110

01110010 (down to 8 bits)

Parity Example

- **Just keep going!**

01110010

01110010

XOR 0111
0010

0101 (down to 4 bits)