

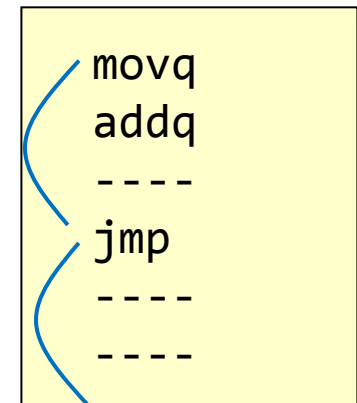
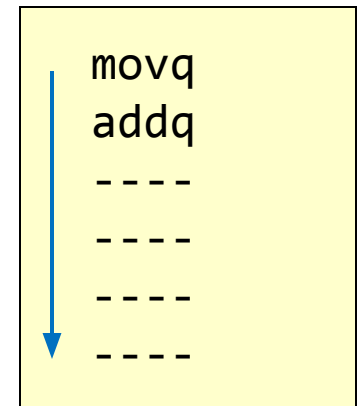
CS356 Unit 5

x86 Control Flow

JUMP/BRANCHING OVERVIEW

Concept of Jumps/Branches

- Assembly is executed in _____ order by default
- Jump instruction (aka "_____") cause execution to skip ahead or back to some other location
- Jumps are used to implement control structures like ___ statements & _____



```

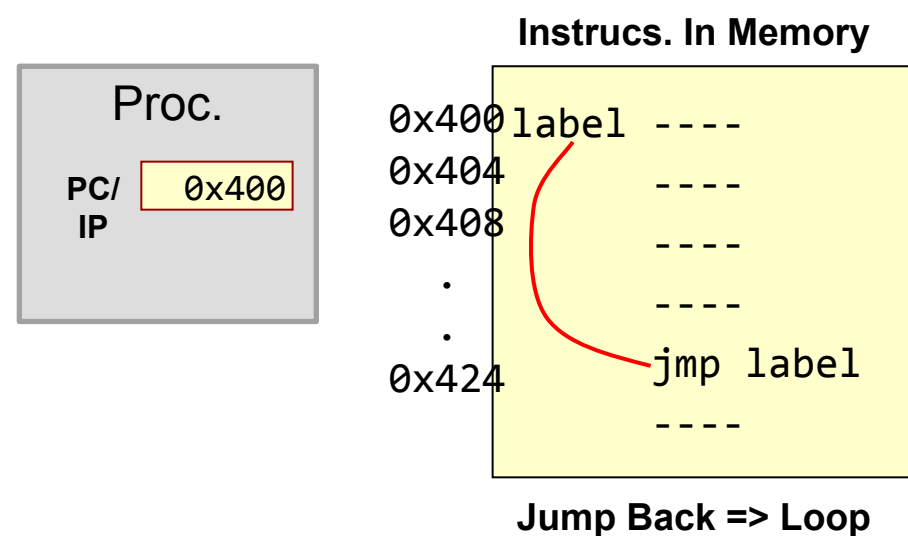
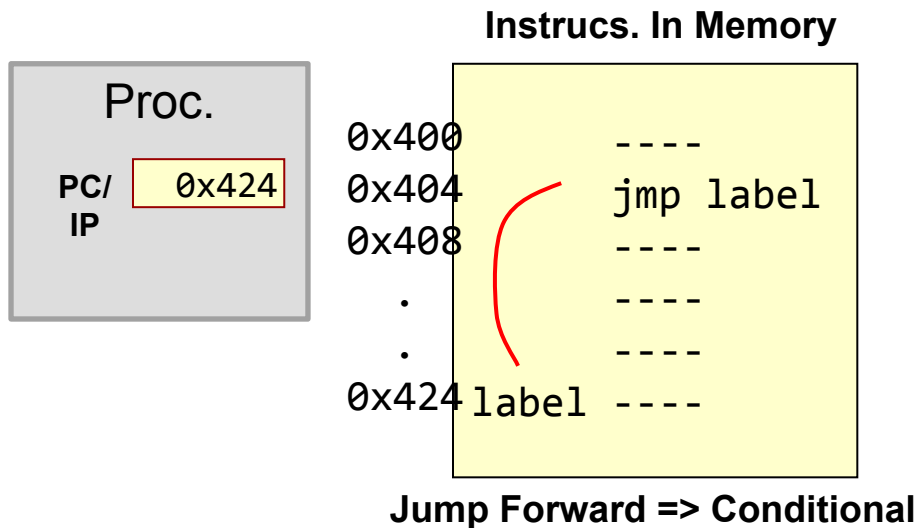
if( x < 0 ){
}
else {
}
    
```

```

while ( x > 0 ){
}
    
```

Jump/Branch Instructions

- Jump (aka "branch") instructions allow us to jump backward or forward in our code
- How? By manipulating the _____
- Operation: _____
 - Compiler/programmer specifies a "label" for the instruction to branch to; then the assembler will determine the displacement



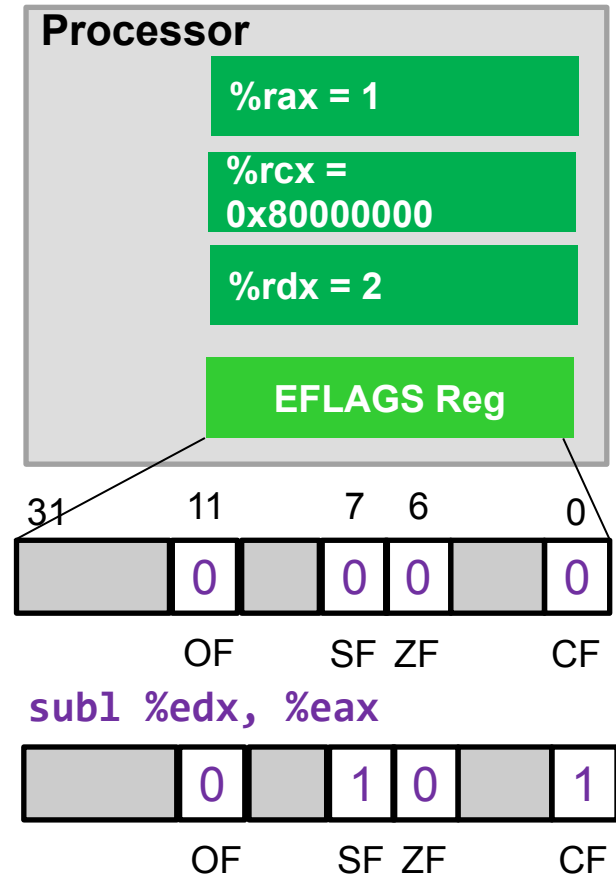
Condition Codes

MAKING A DECISION

Condition Codes (Flags)

CS:APP 3.6.1

- The processor hardware performs several tests on the result of most instructions
- Each test generates a True/False (1 or 0) outcome which are recorded in various bits of the FLAGS register in the process
- The tests and associated bits are:
 - SF = _____
 - Tests if the result is negative (just a copy of the _____ of the result of the instruction)
 - ZF = _____
 - Tests if the result is equal to ____
 - OF = 2's complement _____ Flag
 - Set if signed _____ has occurred
 - CF = ~~Carry Flag~~ _____
 - Not just the carry-out, 1 if unsigned overflow
 - Unsigned Overflow: carry out in addition, or borrow out in subtraction



cmp and test Instructions

- **cmp[bwql] src1, src2**
 - Compares src2 to src1 (e.g. `src2 < src1`, `src2 == src1`)
 - Performs `(src2 - src1)` and **sets the condition codes** based on the result
 - src1 and src2 are _____ (subtraction result is only used for condition codes and then discarded)
- **test[bwql] src1, src2**
 - Performs `(src1 & src2)` and **sets condition codes**
 - src1 and src2 are not changed, OF and CF always set to 0
 - Often used with the `src1 = src2` (i.e., `test %eax, %eax`) to check if a value is _____

Condition Code Exercises

Processor Registers

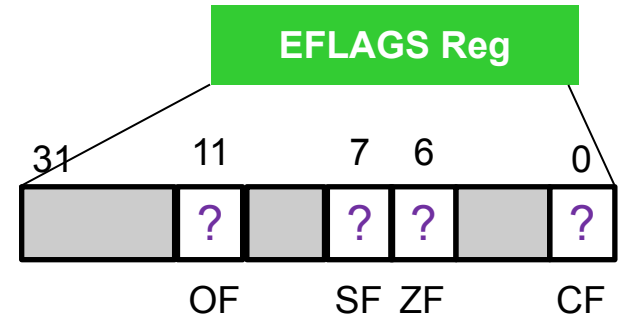
0000 0000 0000 0001
0000 0000 0000 0000
0000 0000 0000 8801
0000 0000 0000 0002

rax

rbx

rcx

rdx



– `addl $0x7fffffff,%edx`

0000 0000 _____

rdx



– `andb %al, %bl`

0000 0000 0000 00__

rbx



– `addb $0xff, %al`

0000 0000 0000 00__

rax



– `cmpw $0x7000, %cx`

0000 0000 0000 _____

result

0000 0000 0000 _____

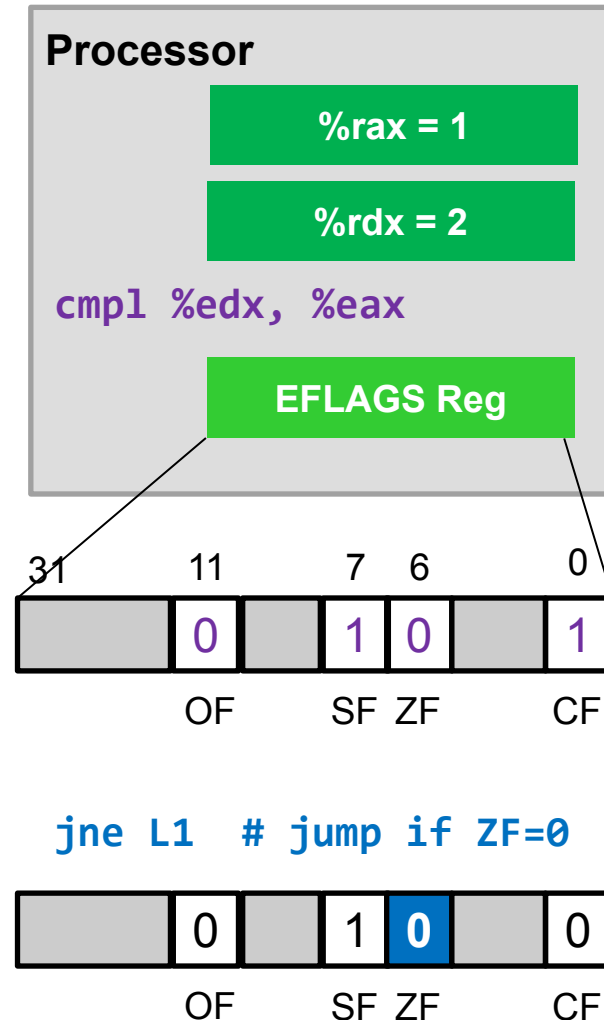
rcx



OF SF ZF CF

Conditional Branches

- Comparison in x86 is *usually* a 2-step (2-instruction) process
- **Step 1:**
 - Execute an instruction that will compare or examine the data (e.g. `cmp`, `test`, etc.)
 - Results of comparison will be saved in the EFLAGS register via the condition codes
- **Step 2:**
 - Use a conditional jump (`je`, `jne`, `jl`, etc.) that will check for a certain comparison result of the previous instruction



Conditional Jump Instructions

CS:APP 3.6.3

- Figure 3.15 from CS:APP, 3e

Instruction	Synonym	Jump Condition	Description
<code>jmp label</code>			
<code>jmp *(Operand)</code>			
<code>je label</code>	<code>jz</code>	ZF	Equal / zero
<code>jne label</code>	<code>jnz</code>	\sim ZF	Not equal / not zero
<code>js label</code>		SF	Negative
<code>jns label</code>		\sim SF	Non-negative
<code>jg label</code>	<code>jnle</code>	\sim (SF ^ OF) & \sim ZF	Greater (signed >)
<code>jge label</code>	<code>jnl</code>	\sim (SF ^ OF)	Greater or Equal (signed >=)
<code>jl label</code>	<code>jnge</code>	(SF ^ OF)	Less (signed <)
<code>jle label</code>	<code>jng</code>	(SF ^ OF) ZF	Less of equal (signed <=)
<code>ja label</code>	<code>jnbe</code>	\sim CF & \sim ZF	Above (unsigned >)
<code>jae label</code>	<code>jnb</code>	\sim CF	Above or equal (unsigned >=)
<code>jb label</code>	<code>jnae</code>	CF	Below (unsigned <)
<code>jbe label</code>	<code>jna</code>	CF ZF	Below or equal (unsigned <=)

Reminder: For all jump instructions other than `jmp` (which is unconditional), some previous instruction (`cmp`, `test`, etc.) is needed to set the condition codes to be examined by the `jmp`

Condition Code Exercises

Processor Registers

0000 0000 0000 0001	rax
0000 0000 0000 0002	rbx
0000 0000 ffff fffe	rcx
0000 0000 0000 0000	rdx

Order:

```

f1:
    testl %edx, %edx
    je    L2
L1:  cmpw  %bx, %ax
    jge  L3
L2:  addl $1,%ecx
    js   L1
L3:  ret
    
```

OF SF ZF CF

<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Control Structure Examples 1

CS:APP 3.6.5

```
// x = %edi, y = %esi, res = %rdx
void func1(int x, int y, int *res)
{
    if (x < y)
        *res = x;
    else
        *res = y;
}
```



```
gcc -S -Og func1.c
```

```
func1:
    cmpl    %esi, %edi
    jge     .L2
    movl    %edi, (%rdx)
    ret

.L2:
    movl    %esi, (%rdx)
    ret
```

```
// x = %edi, y = %esi, res = %rdx
void func2(int x, int y, int *res)
{
    if(x == -1 || y == -1)
        *res = y-1;
    else if(x > 0 && y < x)
        *res = x+1;
    else
        *res = 0;
}
```



```
gcc -S -O3 func2.c
```

```
func2:
    cmpl    $-1, %edi
    je      .L6
    cmpl    $-1, %esi
    je      .L6
    testl   %edi, %edi
    jle     .L5
    cmpl    %esi, %edi
    jle     .L5
    addl    $1, %edi
    movl    %edi, (%rdx)
    ret

.L5:
    movl    $0, (%rdx)
    ret

.L6:
    subl    $1, %esi
    movl    %esi, (%rdx)
    ret
```

Control Structure Examples 2

CS:APP 3.6.7

```
// str = %rdi
int func3(char str[])
{
    int i = 0;
    while(str[i] != 0){
        i++;
    }
    return i;
}
```



gcc -S -Og func3.c

```
func3:
    movl    $0, %eax
    jmp     .L2
.L3:
    addl    $1, %eax
.L2:
    movslq  %eax, %rdx
    cmpb   $0, (%rdi,%rdx)
    jne    .L3
    ret
```

```
// dat = %rdi, len = %esi
int func4(int dat[], int len)
{
    int min = dat[0];
    for (int i=1; i < len; i++) {
        if (dat[i] < min) {
            min = dat[i];
        }
    }
    return min;
}
```



gcc -S -Og func4.c

```
func4:
    movl    (%rdi), %eax
    movl    $1, %edx
    jmp     .L2
.L4:
    movslq  %edx, %rcx
    movl    (%rdi,%rcx,4), %ecx
    cmpl   %ecx, %eax
    jle    .L3
    movl    %ecx, %eax
.L3:
    addl    $1, %edx
.L2:
    cmpl   %esi, %edx
    jl     .L4
    ret
```

Branch Displacements

CS:APP 3.6.4

- **Recall:** Jumps perform _____ = _____
- Assembler converts jumps and labels to appropriate **displacements**
- Examine the disassembled output (below) especially the machine code in the left column
 - Displacements are in the 2nd byte of the instruction
 - Recall: PC increments to point at next instruction while jump is fetched and _____ **the jump is executed**

```

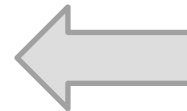
// dat = %rdi, len = %esi
int func4(int dat[], int len)
{
    int i, min = dat[0];
    for(i=1; i < len; i++){
        if(dat[i] < min){
            min = dat[i];
        }
    }
    return min;
}
```

C Code

```

0000000000000000 <func4>:
 0:  8b 07                mov    (%rdi),%eax
 2:  ba 01 00 00 00      mov    $0x1,%edx
 7:  eb 0f                jmp    18 <func4+0x18>
 9:  48 63 ca            movslq %edx,%rcx
 c:  8b 0c 8f            mov    (%rdi,%rcx,4),%ecx
 f:  39 c8                cmp    %ecx,%eax
11:  7e 02                jle   15 <func4+0x15>
13:  89 c8                mov    %ecx,%eax
15:  83 c2 01            add    $0x1,%edx
18:  39 f2                cmp    %esi,%edx
1a:  7c ed                jl    9 <func4+0x9>
1c:  f3 c3                retq
```

x86 Disassembled Output



```

func4:
    movl    (%rdi), %eax
    movl    $1, %edx
    jmp     .L2
.L4:
    movslq %edx, %rcx
    movl    (%rdi,%rcx,4), %ecx
    cmpl   %ecx, %eax
    jle    .L3
    movl   %ecx, %eax
.L3:
    addl   $1, %edx
.L2:
    cmpl   %esi, %edx
    jl    .L4
    ret
```

x86 Assembler

CONDITIONAL MOVES

Cost of Jumps

CS:APP 3.6.6

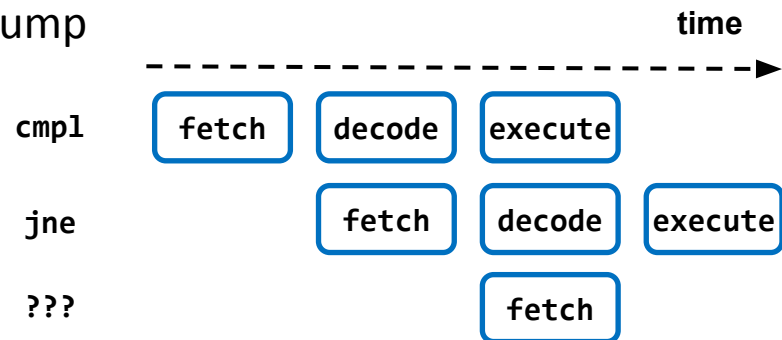
- Fact: Modern processors execute _____ instructions at one time
 - While earlier instructions are executing the processor can be _____ and _____ later instructions
 - This overlapped execution is known as _____ and is key to obtaining good performance
- Problem: **Conditional jumps** _____ **pipelining** because when we reach a jump, the comparison results it relies on may not be computed yet
 - It is _____ which instruction to fetch next
 - To be safe we have to stop and wait for the jump condition to be known

```

func1:
    cmp1    $-1, %edi
    je      .L6
    cmp1    $-1, %esi
    je      .L6
    test1   %edi, %edi
    jle     .L5
    cmp1    %esi, %edi
    jl      .L5
    add1    $1, %edi
    mov1    %edi, (%rdx)
    ret

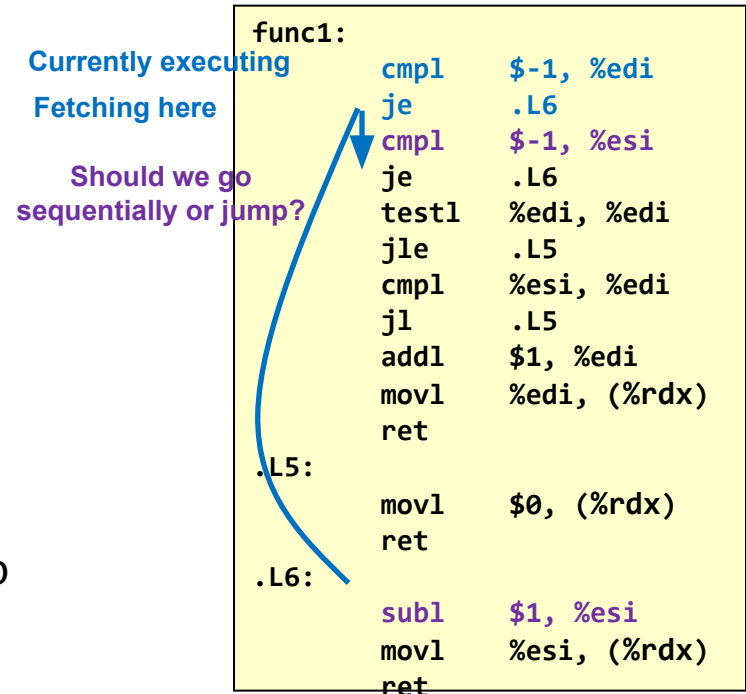
.L5:
    mov1    $0, (%rdx)
    ret

.L6:
    subl    $1, %esi
    mov1    %esi, (%rdx)
    ret
    
```



Cost of Jumps

- Solution: When modern processors reach a jump before the comparison condition is known, it will _____ whether the jump condition will be true (aka "branch prediction") and "_____ " execute down the chosen path
 - If the guess is right...we win and get good performance
 - If the guess is wrong...we lose and will have to _____ the wrongly fetched/decoded instructions once we realize the jump was mispredicted



Conditional Move Concept

- Potential better solution: Be more **pipelining friendly** and compute _____ results and only _____ the correct result when the condition is known
- Allows for pure sequential execution
 - With jumps, we had to choose which instruction to fetch next
 - With conditional moves, we only need to choose whether to _____ a computed result

```
int cmove1(int x, int* res)
{
    if(x > 5) *res = x+1;
    else *res = x-1;
}
```

C Code

```
cmove1:
    cmpl    $5, %edi
    jle     .L2
    addl    $1, %edi
    movl    %edi, (%rsi)
    ret
.L2:
    subl    $1, %edi
    movl    %edi, (%rsi)
    ret
```

With Jumps (-Og Optimization)

```
int cmove1(int x)
{
    int then_val = x+1;
    int temp = x-1;
    if(x > 5) temp = then_val;
    *res = temp;
}
```

Equivalent C code

```
cmove1:
    leal    1(%rdi), %edx
    leal    -1(%rdi), %eax
    cmpl    $6, %edi
    cmovge  %edx, %eax
    movl    %eax, (%rsi)
    ret
```

With Conditional Moves
(-O3 Optimization)

Conditional Move Instruction

- Similar to (cond) ? x : y
- Syntax: `cmov[cond] src, reg`
 - Cond = Same conditions as jumps (e, ne, l, le, g, ge)
 - Destination must be a register
 - If condition is true, _____
 - If condition is false, _____
 - **Transfer size inferred from register name**

```
if(test-expr)
  res = then-expr
else
  res = else-expr
```

```
Let v = then-expr
Let res = else-expr
Let t = test-expr
if(t) res = v // cmov in assembly
```

Conditional Move Instructions

- Figure 3.18 from CS:APP, 3e

Instruction	Synonym	Jump Condition	Description
<code>cmove reg1, reg2</code>	<code>cmovz</code>	ZF	Equal / zero
<code>cmovne reg1, reg2</code>	<code>cmovnz</code>	\sim ZF	Not equal / not zero
<code>cmovs reg1, reg2</code>		SF	Negative
<code>cmovns reg1, reg2</code>		\sim SF	Non-negative
<code>cmovg reg1, reg2</code>	<code>cmovnl</code>	\sim (SF ^ OF) & \sim ZF	Greater (signed >)
<code>cmovge reg1, reg2</code>	<code>cmovnl</code>	\sim (SF ^ OF)	Greater or Equal (signed >=)
<code>cmovl reg1, reg2</code>	<code>cmovnge</code>	(SF ^ OF)	Less (signed <)
<code>cmovle reg1, reg2</code>	<code>cmovng</code>	(SF ^ OF) ZF	Less of equal (signed <=)
<code>cmova reg1, reg2</code>	<code>cmovnbe</code>	\sim CF & \sim ZF	Above (unsigned >)
<code>cmovae reg1, reg2</code>	<code>cmovnb</code>	\sim CF	Above or equal (unsigned >=)
<code>cmovb reg1, reg2</code>	<code>cmovnae</code>	CF	Below (unsigned <)
<code>cmovbe reg1, reg2</code>	<code>cmovna</code>	CF ZF	Below or equal (unsigned <=)

Reminder: Some previous instruction (`cmp`, `test`, etc.) is needed to set the condition codes to be examined by the `cmov`

Conditional Move Exercises

Processor Registers

0000	0000	0000	0001	rax
0000	0000	0000	0000	rbx
0000	0000	0000	8801	rcx
0000	0000	0000	0002	rdx

- `cmpl $8,%edx`
- `cmovl %ecx,%edx`

- `testq %rax,%rax`
- `cmove %rcx,%rax`

0000	0000	0000	_____	rdx	OF	SF	ZF	CF

0000	0000	0000	_____	rax				

Important Notes:

- No size modifier is added to `cmov`, but instead the register names specify the size
- Byte-size conditional moves are not supported (only 16-, 32- or 64-bit conditional moves)

Limitations of Conditional Moves

- If code in then and else have side effects then executing both would _____ the original intent
- If _____ of code in then or else branches, then doing both may be more time consuming

```
int badcmove1(int x, int y)
{
    int z;
    if(x > 5) z = x++; // side effect
    else z = y;
    return z+1;
}

void badcmove2(int x, int y)
{
    int z;
    if(x > 5) {
        /* Lots of code */
    }
    else {
        /* Lots of code */
    }
}
```

C Code

ASIDE: ASSEMBLER DIRECTIVES

Labels and Instructions

- The optional label in front of an instruction evaluates to the _____ where the instruction or data _____ in memory and can be used in other instructions

```

        .text
func4:  movl  %eax,8(%rdx)
.L1:    add   $1,%eax
        jne   .L1
        jmp   func4
    
```

Assembly Source File

movl	0x400000 = func4
add	0x400003 = .L1
jne	0x400006
jmp	0x400008

Assembler finds what address each instruction starts at...

```

        .text
0:      movl  %eax,8(%rdx)
3:      add   $1,%eax
6:      jne   0x400003 (-5)
8:      jmp   0x400000 (-10)
    
```

...and replaces the labels with their corresponding address

Assembler Directives

- Start with `.` (e.g. `.text`, `.quad`, `.long`)
- Similar to pre-processor statements (`#include`, `#define`, etc.) and global variable declarations in C/C++
 - Text and data segments
 - Reserving & initializing global variables and constants
 - Compiler and linker status
- Direct the assembler in how to assemble the actual instructions and how to _____ when the program is loaded

An Example

- Directives specify
 - Where to place the information (.text, .data, etc.)
 - What names (symbols) are visible to other files in the program (.globl)
 - Global data variables & their size (.byte, .long, .quad, .string)
 - Alignment requirements (.align)

```
int x[4] = {1,2,3,4};
char* str = "Hello";
unsigned char z = 10;
double grades[10];

int func()
{
    return 1;
}
```



```
.text
.globl func
func:
    movl    $1, %eax
    ret

.globl z
.data
z:
    .byte   10

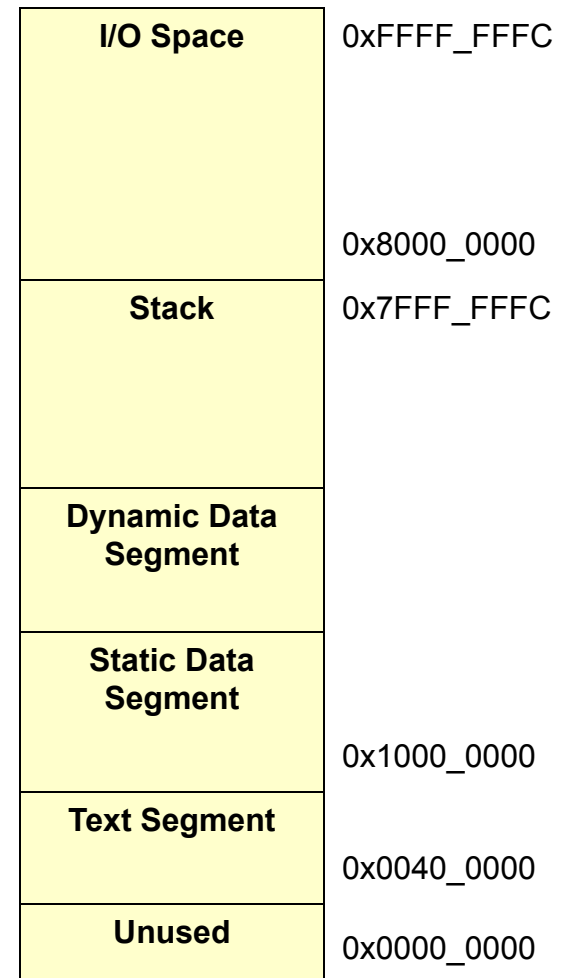
.globl str
.string "Hello"

.data
.align 8
str:
    .quad  .LC0

.globl x
.align 16
x:
    .long  1
    .long  2
    .long  3
    .long  4
```

Text and Data Segments

- .text directive indicates the following instructions should be placed in the program area of memory
- .data directive indicates the following data declarations will be placed in the data memory segment



Static Data Directives

- Fills memory with specified data when program is loaded
- Format:

(Label:) *.type_id* *val_0, val_1, ..., val_n*

- `type_id = { .byte, .value, .long, .quad, .float, .double }`
- Each value in the comma separated list will be stored using the indicated size
 - Example: `myval: .long 1, 2, 3`
 - Each value 1, 2, 3 is stored as a word (i.e. 32-bits)
 - Label “myval” evaluates to the start address of the first word (i.e. of the value 1)

Indirect jumps with jump tables

SWITCH TABLES

Switch with Direct Jumps

CS:APP 3.6.8

```
void switch1(unsigned x, int* res)
{
    switch(x%8)
    {
        case 0:
            *res = x+5;
            break;
        case 1:
            *res = x-3;
            break;
        case 2:
            *res = x+12;
            break;
        default:
            *res = x+7;
            break;
    }
}
```



```
switch1:
    movl    %edi, %eax
    andl    $7, %eax
    cmpl    $1, %eax
    je      .L3
    cmpl    $1, %eax
    jb      .L4
    cmpl    $2, %eax
    je      .L5
    jmp     .L7
.L4:
    addl    $5, %edi
    movl    %edi, (%rsi)
    ret
.L3:
    subl    $3, %edi
    movl    %edi, (%rsi)
    ret
.L5:
    addl    $12, %edi
    movl    %edi, (%rsi)
    ret
.L7:
    addl    $7, %edi
    movl    %edi, (%rsi)
    ret
```

Switch w/ Indirect Jumps (Jump Tables)

```
// x = %edi, res = %rsi
void switch2(unsigned x, int* res)
{
    switch(x%8)
    {
        case 0:
            *res = x+5;
            break;
        case 1:
            *res = x-3;
            break;
        case 2:
            *res = x+12;
            break;
        case 3:
            *res = x+7;
            break;
        case 4:
            *res = x+5;
            break;
        case 5:
            *res = x-3;
            break;
        case 6:
            *res = x+12;
            break;
        case 7:
            *res = x+7;
            break;
    }
}
```

jump to
 *(table[x%8])

1000 0ef0	0040 008a
	0040 0090
	0040 0096
	0040 009c
	0040 00a2
	0040 00a8
	0040 00ae
	0040 00b4



```
switch2:
    movl    %edi, %eax
    andl   $7, %eax
    movl   %eax, %eax
    jmp    *.L4(,%rax,8)
    .section
    .rodata
    .align 8
    .align 4
.L4:    1000 0ef0
        .quad .L3
        .quad .L5
        .quad .L6
        .quad .L7
        .quad .L8
        .quad .L9
        .quad .L10
        .quad .L11
        .text
.L3:    0040 008a
        addl   $5, %edi
        movl   %edi, (%rsi)
        ret
.L5:    0040 0090
        subl   $3, %edi
        movl   %edi, (%rsi)
        ret
.L6:    0040 0096
        addl   $12, %edi
        movl   %edi, (%rsi)
        ret
.L7:    0040 009c
        addl   $7, %edi
        movl   %edi, (%rsi)
        ret
.L8:    0040 00a2
        addl   $5, %edi
        movl   %edi, (%rsi)
        ret
.L9:    0040 00a8
        subl   $3, %edi
        movl   %edi, (%rsi)
        ret
.L10:   0040 00ae
        addl   $12, %edi
        movl   %edi, (%rsi)
        ret
.L11:   0040 00b4
        addl   $7, %edi
        movl   %edi, (%rsi)
        ret
```