CS356 Unit 5

x86 Control Flow

An overview for BombLab

ASSEMBLY PROCEDURES

Assembly Procedures

- Used to implement function calls:
  - Prepare arguments (in **rdi**, **rsi**, **rdx**, **rcx**, **r8**, **r9**)
  - Jump to the procedure (**callq** instruction)
  - The procedure saves the return value in **rax**
  - The procedure uses the **retq** instruction to jump back to the instruction after **callq**

C code:

Call

```
...
res = avg(x,4);
...
```

Definition

```
// a in %edi, b in %esi
// return value in %eax
int avg(int a, int b) {
  return (a+b)/2;
}
```

Assembly:

```
113b      callq avg   # save a link, jump
1140      next inst. # use result %eax
```

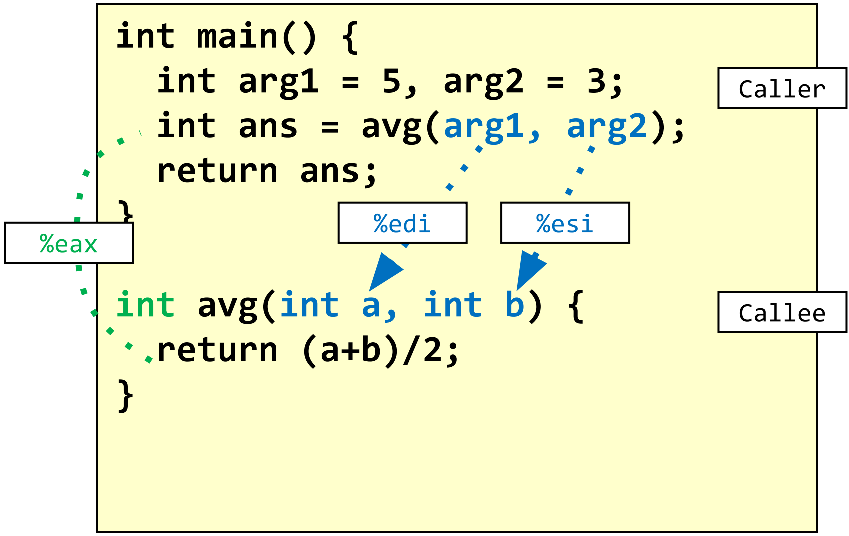
Desired return location

```
avg:
1125      addl %edi,%esi
1127      movl %esi,%eax
1129      shrl $31,%eax   // 1 if negative
112c      addl %esi,%eax // ... biasing!
112e      sarl %eax
1130      ret
```

Arguments and Return Values

CS:APP 3.7.3

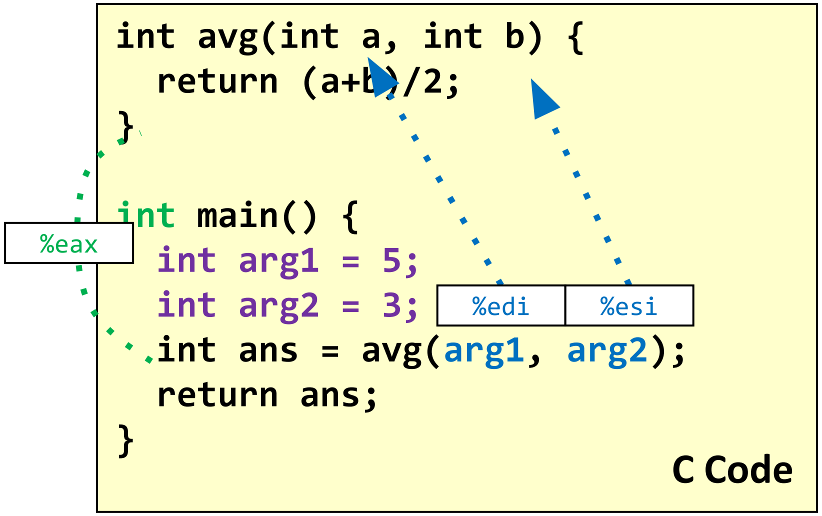
- Most procedure calls pass arguments/parameters to the procedure and it often produces return values
- To implement this, there must be **locations agreed upon** by caller and callee for where this information will be found
- x86-64 convention is to use certain registers for this task (**see table**)



```
int main() {
  int arg1 = 5, arg2 = 3;
  int ans = avg(arg1, arg2);
  return ans;
}

int avg(int a, int b) {
  return (a+b)/2;
}
```

1st Argument	%rdi
2nd Argument	%rsi
3rd Argument	%rdx
4th Argument	%rcx
5th Argument	%r8
6th Argument	%r9
Additional arguments	Pass on stack *(future lecture)*
Return value	%rax

Passing Arguments and Return Values

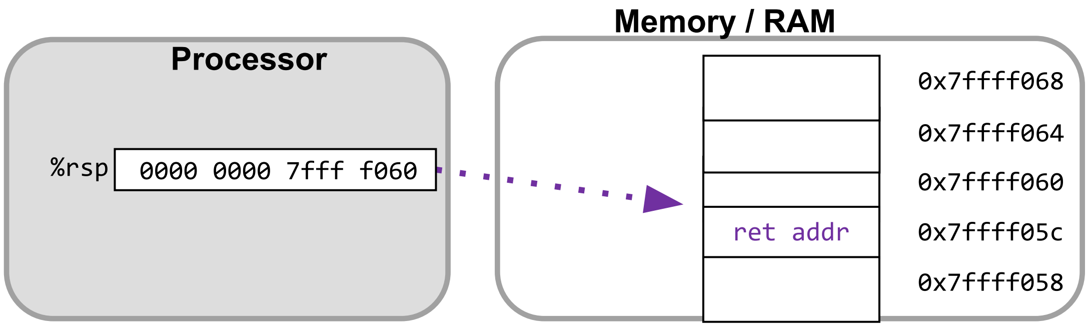


```
int avg(int a, int b) {
  return (a+b)/2;
}

int main() {
  int arg1 = 5;
  int arg2 = 3;
  int ans = avg(arg1, arg2);
  return ans;
}
```

```
        .text
        .globl  avg
avg:
        addl    %edi, %esi
        movl    %esi, %eax
        shrl    $31, %eax
        addl    %esi, %eax
        sarl    %eax
        ret

        .globl  main
main:
        movl    $3, %esi
        movl    $5, %edi
        call    avg
        ret
```

Assembly

After returning to main():



Example

```c
#include <stdio.h>
#include <string.h>

int phase1(char *input) {
  if (strcmp(input, "gandalf\n"))
    return 1;  // wrong input
  else
    return 0;  // right input
}

void explode_bomb() {
  // notifies our server
}

int main() {
  char input[200];
  fgets(input, 200, stdin);
  if (!phase1(input)) {
    puts("Success!");
  } else {
    explode_bomb();
  }
}
```

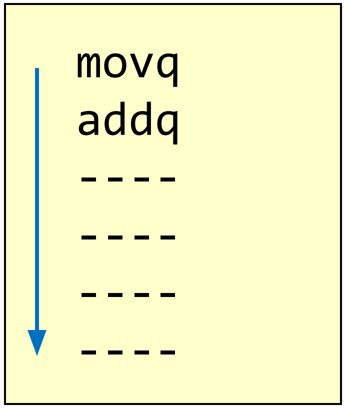
```asm
.text
.LC0: .string "gandalf\n"
.LC1: .string "Success!"
phase1:
  leaq .LC0(%rip), %rsi
  callq strcmp
  testl %eax, %eax
  je .L1
  movl $1, %eax
.L1:
  ret

explode_bomb:  // ADD BREAKPOINT HERE!
  // notifies our server
  ret

.globl main
main:
  // read string from stdin, save its addr in %rdi
  call phase1
  testl %eax, %eax
  jne .L6
  leaq .LC1(%rip), %rdi
  call puts
  movl $0, %eax
  ret
.L6:
  call explode_bomb
  movl $0, %eax
  ret
```

Just focus on the procedure calls/returns for now :-)

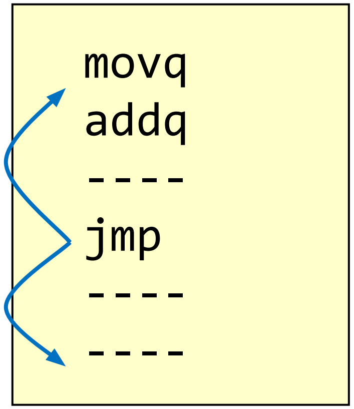
JUMP/BRANCHING OVERVIEW

Concept of Jumps/Branches

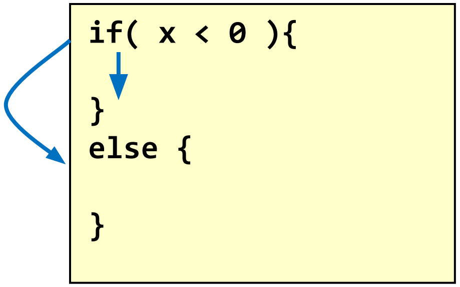
- Assembly is executed in sequential order by default
- Jump instruction (aka "branches") cause execution to skip ahead or back to some other location
- Jumps are used to implement control structures like if statements & loops



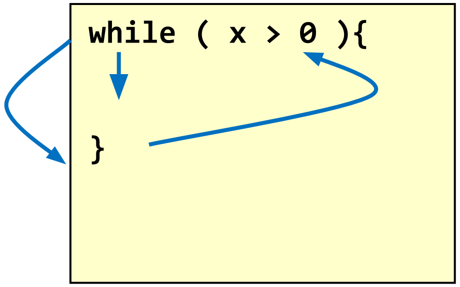
```
movq
addq
----
----
----
----
```



```
movq
addq
----
jmp
----
----
```



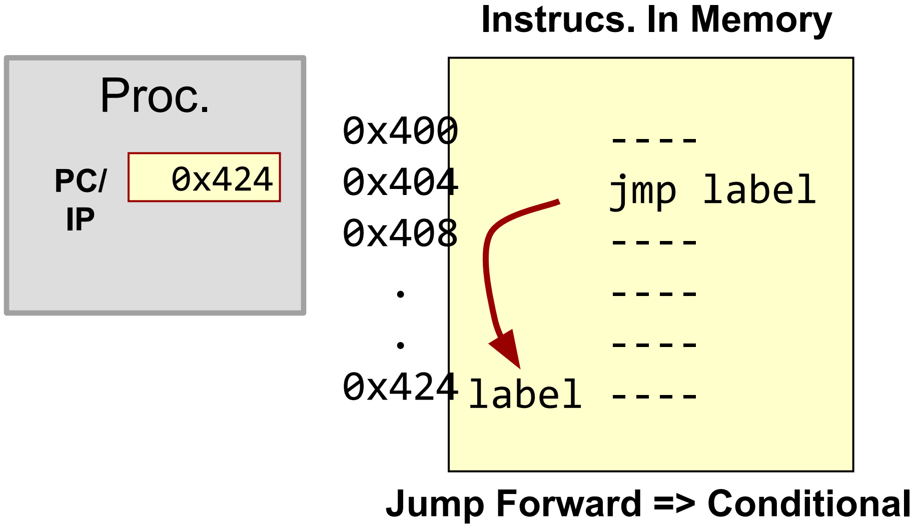
```
if( x < 0 ){

}
else {

}
```



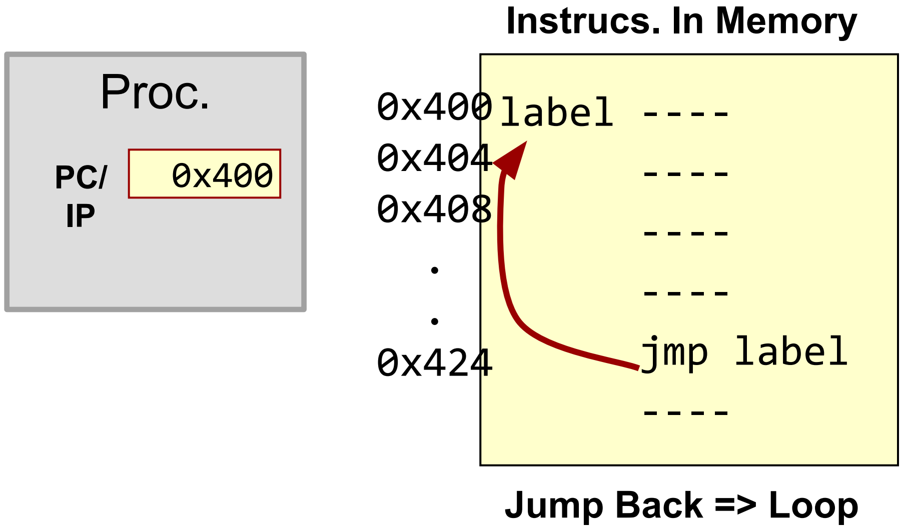
```
while ( x > 0 ){


}
```

Jump/Branch Instructions

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



Jump Forward => Conditional

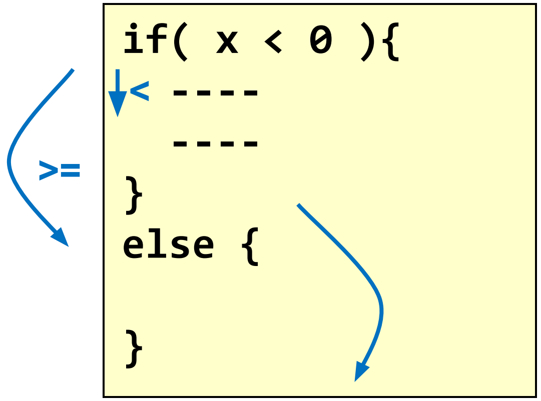


Jump Back => Loop

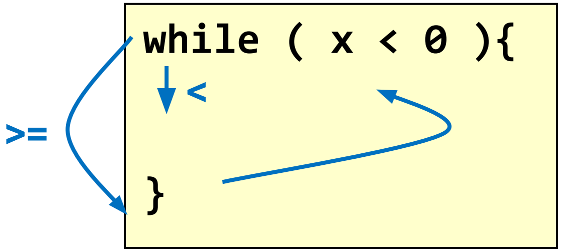
Conditional vs. Unconditional Jumps

Two kinds of jumps/branches

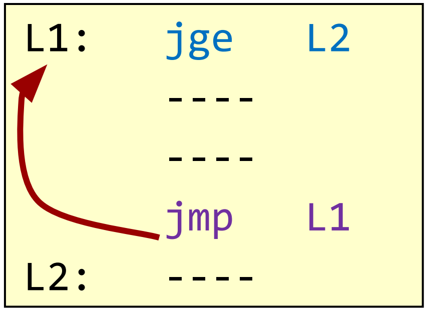
- **Conditional**
  - Jump only if a condition is true, otherwise continue sequentially
  - x86 instructions: `je, jne, jge, …` (see next slides)
    - **Need a way to compare and check conditions**
    - Needed for if, while, for
- **Unconditional**
  - Always jump to a new location
  - x86 instruction: `jmp label`



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

}
```



```
while ( x < 0 ){


}
```



```
L1:   jge   L2
      ----
      ----
      jmp   L1
L2:   ----
```

x86 View

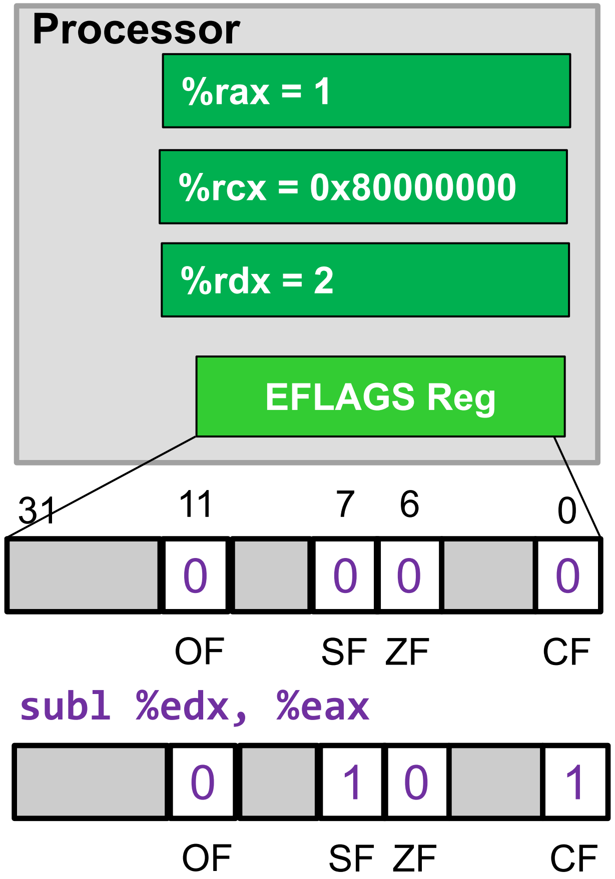
Condition Codes

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** = Sign Flag
    - Tests if the result is negative (just a copy of the MSB of the result of the instruction)
  - **ZF** = Zero Flag
    - Tests if the result is equal to 0
  - **OF** = 2's complement Overflow Flag
    - Set if signed overflow has occurred
  - **CF** = ~~Carry Flag~~ Unsigned Overflow
    - Not just the carry-out, 1 if unsigned overflow
    - Unsigned Overflow: carry out in addition, or borrow out in subtraction



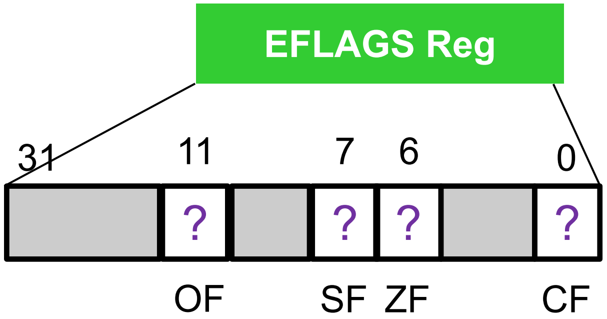
`cmp` and **`test`** Instructions

- **`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 not changed** (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 **0 or negative** (through ZF and SF)

Condition Code Exercises 1

Processor Registers

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



- addl $0x7fffffff,%edx

0000 0000 8000 0001	rdx

OF	SF	ZF	CF
1	1	0	0

- andb %al, %bl

0000 0000 0000 0000	rbx

OF	SF	ZF	CF
0	0	1	0

- addb $0xff, %al

0000 0000 0000 0000	rax

OF	SF	ZF	CF
0	0	1	1

- cmpw $0x7000, %cx

0000 0000 0000 1801	result
0000 0000 0000 8801	rcx

OF	SF	ZF	CF
1	0	0	0

Move and Logic Ops and Condition Codes

- **`mov`** and **`lea`** instructions ***leave the condition codes unaffected***
- **Logical instructions**
  - **and**, **or**, **xor** update only **SF** and **ZF** based on the result and ***clear CF and OF to 0***
  - **not** does not affect the condition codes in any way

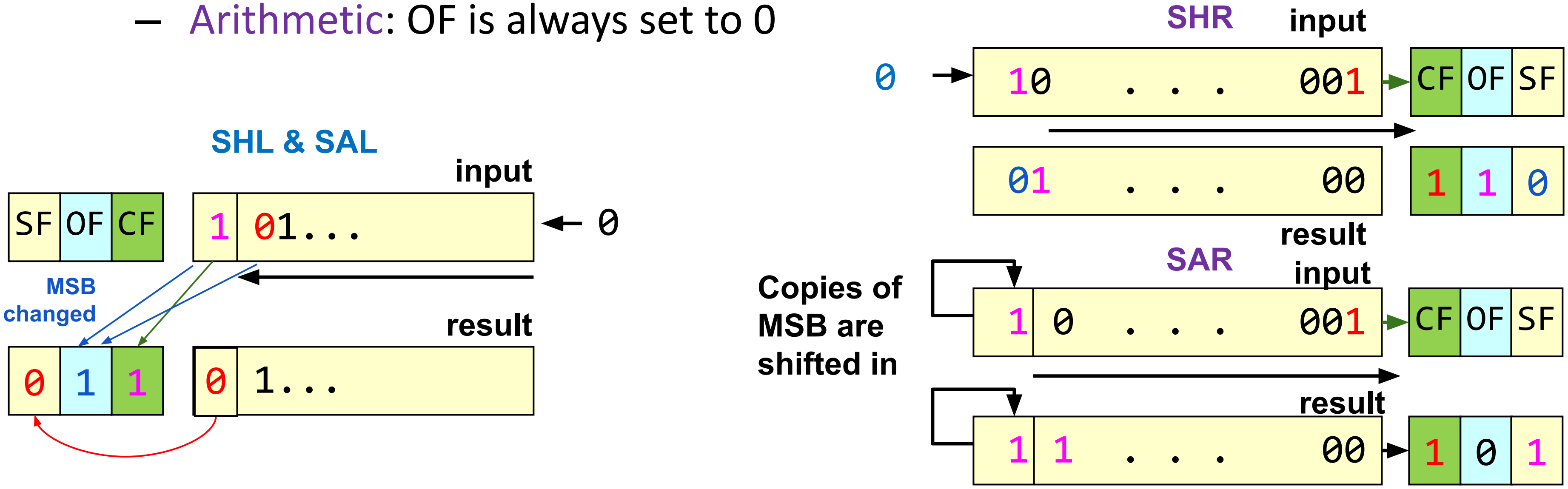
EFLAGS

			SF	ZF	OF
	0000 0000 8000 0001	rdx	1	0	0
movw $0,%dx	0000 0000 8000 0000	rdx	1	0	0
leaq -1(%rdx),%rdx	0000 0000 7FFF FFFF	rdx	1	0	0

			SF	ZF
	0000 0000 8000 FF33	rdx	1	0
andb $0xcc,%dl	0000 0000 8000 FF00	rdx	0	1
orb $0x80,%dl	0000 0000 8000 FF80	rdx	1	0
notb %dl	0000 0000 8000 FF7F	rdx	1	0

Shifts and Condition Codes

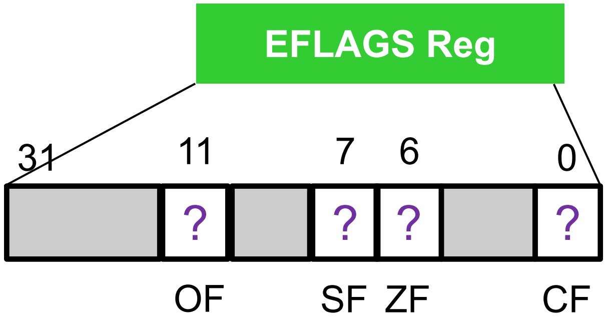
- **All shift instructions**
  - Set SF (copy of MSB) and ZF (true if result is 0)
  - CF is always set with the **last bit shifted out** of the input
  - **OF = undef** for **shifts of more than 1 bit**; shifts by 1-bit work as follows…
- **Left shifts (Logical or Arithmetic) by 1-bit**
  - OF = 1 if MSB (sign bit) changed (i.e. **`CF ^ MSB(result)`**); 0, otherwise.
- **Right shifts by 1-bit**
  - Logical: OF is set with the ORIGINAL MSB of the input value
  - Arithmetic: OF is always set to 0



Condition Code Exercises 2

Processor Registers

0000 0000 ff00 f0f6	rax
0000 0000 0000 018a	rbx
0000 0000 0000 0002	rcx
0000 0000 1234 8000	rdx



- `shlw $1,%dx`

		OF	SF	ZF	CF
0000 0000 1234 0000	rdx	1	0	1	1

- `shrb $1, %bl`

		OF	SF	ZF	CF
0000 0000 0000 0145	rbx	1	0	0	0

- `sarb %cl, %al`

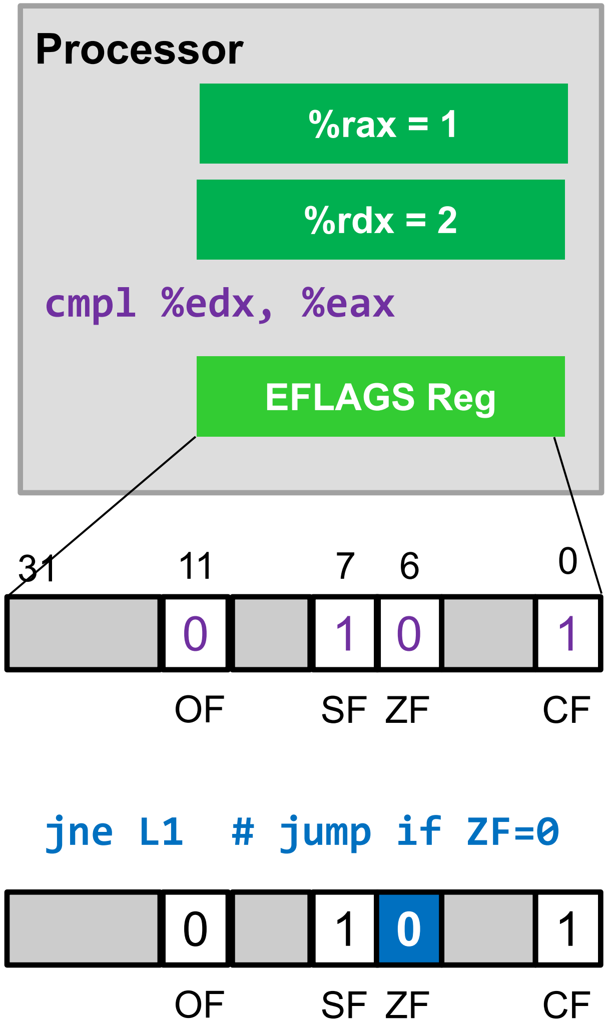
		OF	SF	ZF	CF
0000 0000 ff00 f0fd	rax	?	1	0	1

- `shrb %cl, %bl`

		OF	SF	ZF	CF
0000 0000 0000 0122	rbx	?	0	0	1

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**)
  - 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



```
cmpl %edx, %eax
```

```
jne L1   # jump if ZF=0
```

Conditional Jump Instructions

CS:APP 3.6.3

- Figure 3.15 from CS:APP, 3e

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

How to interpret (quickly)

```
// For: jg, jge, jle, jl   (signed comparison  >, >=, <=, <)
//      ja, jae, jbe, jb (unsigned comparison  >, >=, <=, <)
//      je, jne   (signed/unsigned comparison ==, !=)

// jump if %rbx >= %rax
cmpq %rax, %rbx
jge .L1

// For: jz/je, jnz/jne (signed/unsigned ==, != 0)
//      jg, jge/jns, jle, jl/js (signed  >, >=, <=, < 0)

// jump if %rbx >= 0
testq %rbx, %rbx
jge .L1
```

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:
__1__          testl %edx, %edx
__2__          je    L2
__5___   L1:   cmpw  %bx, %ax
__6___         jge   L3
__3,7_   L2:   addl  $1,%ecx
__4,8_         js    L1
____9_   L3:   ret
```

OF	SF	ZF	CF				
0	0	1	0				
0	1	0	1				
0	1	0	0	0	0	1	1

Reminder: je jumps if ZF, jge jumps if ~(SF ^ OF), js jumps if SF

Example

```
#include <stdio.h>
#include <string.h>

int phase1(char *input) {
  if (strcmp(input, "gandalf\n"))
    return 1;  // wrong input
  else
    return 0;  // right input
}

void explode_bomb() {
  // notifies our server
}

int main() {
  char input[200];
  fgets(input, 200, stdin);
  if (!phase1(input)) {
    puts("Success!");
  } else {
    explode_bomb();
  }
}
```

```
.text
.LC0: .string "gandalf\n"
.LC1: .string "Success!"
phase1:
  leaq .LC0(%rip), %rsi
  callq strcmp
  testl %eax, %eax
  je .L1
  movl $1, %eax
.L1:
  ret

explode_bomb:  // ADD BREAKPOINT HERE!
  // notifies our server
  ret

.globl main
main:
  // read string from stdin, save its addr in %rdi
  call phase1
  testl %eax, %eax
  jne .L6
  leaq .LC1(%rip), %rdi
  call puts
  movl $0, %eax
  ret
.L6:
  call explode_bomb
  movl $0, %eax
  ret
```

Skip next instruction if strcmp returned 0

Go to .L6 if phase1's returned nonzero value

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

```c
// 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
```

```c
// 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 PC = PC + displacement
- 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 **BEFORE 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

```
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

```
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

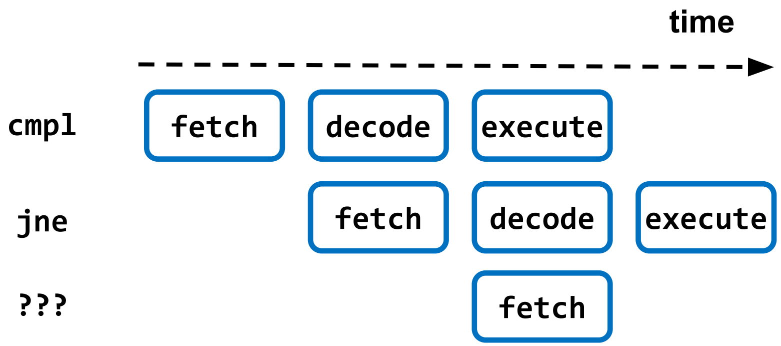
CONDITIONAL MOVES

Cost of Jumps

CS:APP 3.6.6

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

```
func1:
        cmpl    $-1, %edi
        je      .L6
        cmpl    $-1, %esi
        je      .L6
        testl   %edi, %edi
        jle     .L5
        cmpl    %esi, %edi
        jl      .L5
        addl    $1, %edi
        movl    %edi, (%rdx)
        ret
.L5:
        movl    $0, (%rdx)
        ret
.L6:
        subl    $1, %esi
        movl    %esi, (%rdx)
        ret
```


time

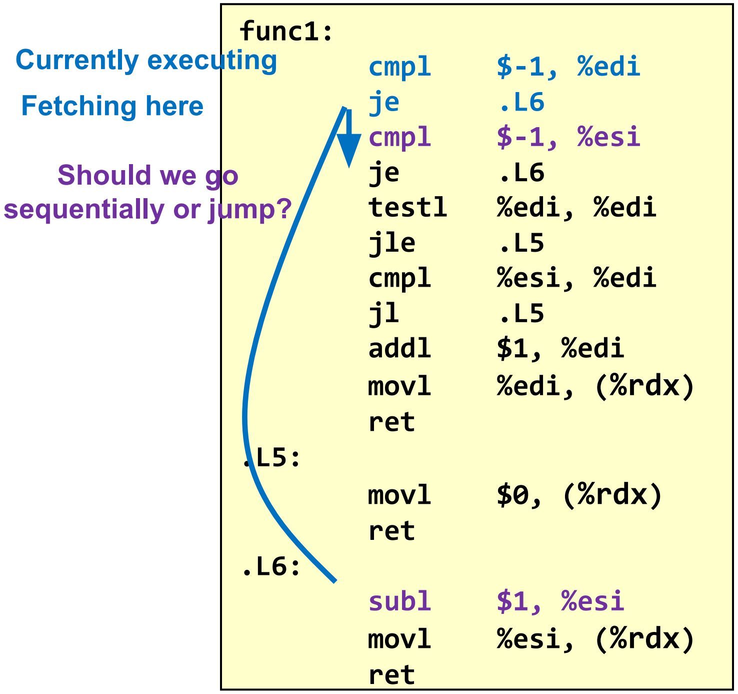
cmpl: fetch, decode, execute

jne: fetch, decode, execute

???: fetch

Cost of Jumps

- Solution: When modern processors reach a jump before the comparison condition is known, it will predict whether the jump condition will be true (aka "branch prediction") and "speculatively" 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 throw away the wrongly fetched/decoded instructions once we realize the jump was mispredicted



```
func1:
        cmpl    $-1, %edi
        je      .L6
        cmpl    $-1, %esi
        je      .L6
        testl   %edi, %edi
        jle     .L5
        cmpl    %esi, %edi
        jl      .L5
        addl    $1, %edi
        movl    %edi, (%rdx)
        ret
.L5:
        movl    $0, (%rdx)
        ret
.L6:
        subl    $1, %esi
        movl    %esi, (%rdx)
        ret
```

Conditional Move Concept

- Potential better solution: Be more **pipelining friendly** and compute both results and only store 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 save or discard 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, reg = src
  - If condition is false, reg is unchanged
  - **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
cmove reg1,reg2	cmovz	ZF	Equal / zero
cmovne reg1,reg2	cmovnz	~ZF	Not equal / not zero
cmovs reg1,reg2		SF	Negative
cmovns reg1,reg2		~SF	Non-negative
cmovg reg1,reg2	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
cmovge reg1,reg2	cmovnl	~(SF ^ OF)	Greater or Equal (signed >=)
cmovl reg1,reg2	cmovnge	(SF ^ OF)	Less (signed <)
cmovle reg1,reg2	cmovng	(SF ^ OF) | ZF	Less of equal (signed <=)
cmova reg1,reg2	cmovnbe	~CF & ~ZF	Above (unsigned >)
cmovae reg1,reg2	cmovnb	~CF	Above or equal (unsigned >=)
cmovb reg1,reg2	cmovnae	CF	Below (unsigned <)
cmovbe reg1,reg2	cmovna	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`

		OF	SF	ZF	CF
0000 0000 0000 8801	rdx	0	1	0	1

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

		OF	SF	ZF	CF
0000 0000 0000 0001	rax	0	0	0	0

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 violate the original intent
- If large amounts 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 address where the instruction or data starts 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 initialize memory when the program is loaded

An Example

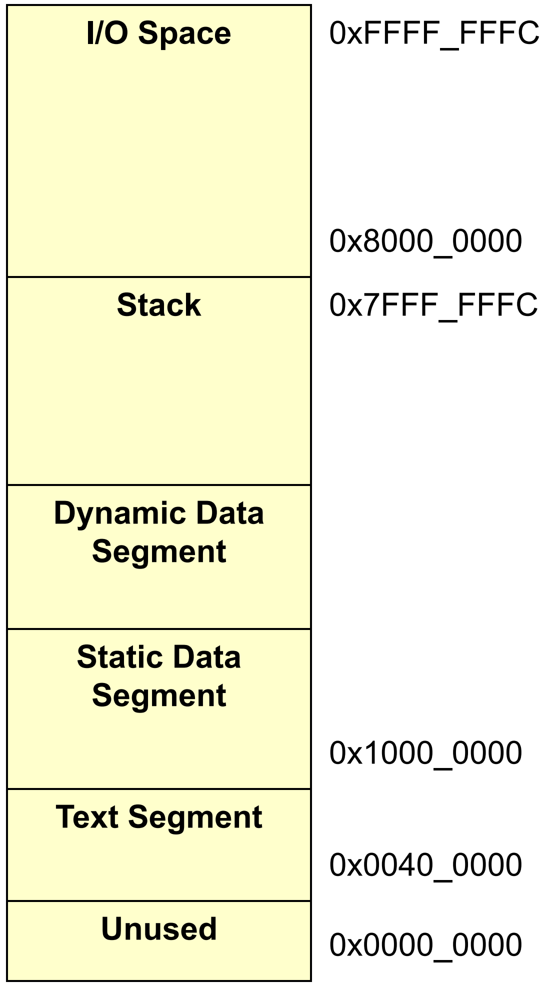
- Directives specify
  - Where to place the information (.text, .data, etc.)
  - What names (symbols) are visible to other files in the program (.globl)
  - Global data variables & their size (.byte, .long, .quad, .string)
  - Alignment requirements (.align)

```
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



I/O Space 0xFFFF_FFFC

0x8000_0000

Stack 0x7FFF_FFFC

Dynamic Data Segment

Static Data Segment

0x1000_0000

Text Segment

0x0040_0000

Unused 0x0000_0000

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;
    }
}
```

gcc -Og -S switch.c

```
switch1:
      movl  %edi, %eax
      andl  $7, %eax     // same as x%8
      cmpl  $1, %eax
      je    .L2          // jumps if x%8==1
      cmpl  $2, %eax
      je    .L3          // jumps if x%8==2
      testl   %eax, %eax
      je    .L6          // jumps if x%8==0
      addl  $7, %edi
      movl  %edi, (%rsi)
      ret
.L6:
      addl  $5, %edi
      movl  %edi, (%rsi)
      ret
.L2:
      subl  $3, %edi
      movl  %edi, (%rsi)
      ret
.L3:
      addl  $12, %edi
      movl  %edi, (%rsi)
      ret
```

Default

Case 0

Case 1

Case 2

Switch with 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;
    }
}
```

gcc -Og -S switch.c

Jump Table

.L4	.L11-.L4
	.L10-.L4
	.L9-.L4
	.L8-.L4
	.L7-.L4
	.L6-.L4
	.L5-.L4
	.L3-.L4

```
.text   // start a code block
.globl  switch2

switch2:
  // save x%8 into eax
  movl  %edi, %eax
  andl  $7, %eax

  // save addr of table to rdx
  leaq  .L4(%rip), %rdx

  // use eax as an index to read
  // an entry (offset) from table
  movslq  (%rdx,%rax,4), %rax

  // add entry to addr of table
  addq  %rdx, %rax

  // jump to: table addr + entry
  jmp   *%rax

.section .rodata  // data block
.align 4
.align 4

// table of long words (4 bytes)
// each entry has an offset from
// .L4, the addr of the table
.L4:
  .long  .L11-.L4
  .long  .L10-.L4
  .long  .L9-.L4
  .long  .L8-.L4
  .long  .L7-.L4
  .long  .L6-.L4
  .long  .L5-.L4
  .long  .L3-.L4
```

```
.text // start a code block

.L11: // at .L4+table[0]
  addl  $5, %edi
  movl  %edi, (%rsi)
  ret
.L0:  // at .L4+table[1]
  subl  $3, %edi
  movl  %edi, (%rsi)
  ret
.L9:  // at .L4+table[2]
  addl  $12, %edi
  movl  %edi, (%rsi)
  ret
.L8:  // at .L4+table[3]
  addl  $7, %edi
  movl  %edi, (%rsi)
  ret
.L7:  // at .L4+table[4]
  addl  $5, %edi
  movl  %edi, (%rsi)
  ret
.L6:  // at .L4+table[5]
  subl  $3, %edi
  movl  %edi, (%rsi)
  ret
.L5:  // at .L4+table[6]
  addl  $12, %edi
  movl  %edi, (%rsi)
  ret
.L3:  // at .L4+table[7]
  addl  $7, %edi
  movl  %edi, (%rsi)
  ret
```

About switch(x%8) with signed x

```
// x = %edi, res = %rsi
void switch2(int 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;
    }
}
```

gcc -Og -S switch.c

```
.text  // start a code block
.globl  switch2

switch2:
  // save x%8 into eax
  movl  %edi, %edx
  sarl  $31, %edx
  shrl  $29, %edx
  leal  (%rdi,%rdx), %eax
  andl  $7, %eax
  subl  %edx, %eax
  // jump to .L1 if rax is not
  // one of 0, 1, .., 7
  cmpl  $7, %eax
  ja .L1
  movl  %eax, %eax

// rest is similar
...

// additional label at end
.L1:
  ret
```

Why all these instructions for x%8? **x%8 could be in -7,..,-1 if x < 0** !!!

- **sarl $31, %edx** replicates sign bit of edi all over edx
- **shrl $29, %edx** keeps sign bit only over the last 32-29=3 bits (**bias**)
- **leal (%rdi,%rdx), %eax** adds 7 to edi if edi was negative, 0 if positive
- **andl $7, %eax** keeps only the last three bits
- **subl %edx, %eax** subtracts the bias (7 if edi was negative, 0 if positive)

Example: for edi=9, the bias edx is 0, (9+0)&7-0 is 1 => correct! 1 == 9%8
Example: for edi=-9, the bias edx is 7, (-9+7)&7-7 is -1 => correct! -1 == -9%8