

Control

Young W. Lim

2022-06-02 Thr

- 1 Based on
- 2 Control
 - TOC: Control
- 3 Condition Codes
- 4 Accessing the Condition Codes
- 5 Jump Instructions
- 6 PC-relative Addressing Example
- 7 Translating Conditional Branches
- 8 Loop Instructions
- 9 Switch

- 1 "Self-service Linux: Mastering the Art of Problem Determination",

Mark Wilding

- 1 "Computer Architecture: A Programmer's Perspective", Bryant & O'Hallaron

I, the copyright holder of this work, hereby publish it under the following licenses: GNU head Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled GNU Free Documentation License.

CC BY SA This file is licensed under the Creative Commons Attribution ShareAlike 3.0 Unported License. In short: you are free to share and make derivative works of the file under the conditions that you appropriately attribute it, and that you distribute it only under a license compatible with this one.

Compiling 32-bit program on 64-bit gcc

- `gcc -v`
- `gcc -m32 t.c`
- `sudo apt-get install gcc-multilib`
- `sudo apt-get install g++-multilib`
- `gcc-multilib`
- `g++-multilib`
- `gcc -m32`
- `objdump -m i386`

- Condition codes
- Accessing the condition codes
- Jump instructions
- PC-relative addressing example
- Translating conditional branches
- Loop instruction
- Switch

TOC: Conditional codes

Essential flags

Z	Zero flag	destination equals zero
S	Sign flag	destination is negative
C	Carry flag	unsigned value out of range
O	Overflow flag	signed value out of range

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_1

- Whenever the destination operand equals Zero, the **Z**ero flag is set

```
movw $1, %cx
subw $1, %cx          ; %cx = 0, ZF = 1
movw $0xFFFF, %ax
incw %ax              ; AX = 0, ZF = 1
incw %ax              ; AX = 1, ZF = 0
```

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_1

- the Sign flag is set when the destination operand is negative
- the Sign flag is clear when the destination operand is positive

```
movw $0, %cx
subw $1, %cx      ; %cx = -1, SF = 1
addw $2, %cx      ; %cx = 1, SF = 0
```

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_1

- Addition : copy carry out of MSB to CF
- Subtraction : copy inverted carry out of MSB to CF
- INC / DEC : not affect CF
- Applying NEG to a nonzero operand sets CF

```
movw $0x00ff, %cx
addw $1,      %ax      ; %ax = 0x0100, SF = 0, ZF = 0, CF = 0
subw $1,      %ax      ; %cx = 0x00ff, SF = 0, ZF = 0, CF = 0
addb %1,      %al      ; %al = 0x00, SF = 0, ZF = 1, CF = 1
movb $0x6c,   %bh
addb %0x95,   %bh      ; %bh = 0x01, SF = 0, ZF = 0, CF = 1

movb $2,      %al
subb $3,      %al      ; %al = 0xff, SF = 1, ZF = 0, CF = 1
```

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_1

Overflow flag **OF**

- the overflow flag is set when the **signed** result of an operation is invalid or out of range
 - case 1: adding two positive operands produces a negative number
 - case 2: adding two negative operands produces a positive number

```
movb $+127, %al
addb $1,    %al      ; %al = -128,  OF = 1
```

```
movb $0x7F, %al
addb $1,    %al      ; %al = 0x80,  OF = 1
```

https://www.csie.ntu.edu.tw/~cyy/courses/assembly/12fall/lectures/handouts/lec14_1

Condition Codes (1)

- condition code registers describe attributes of the most recent arithmetic or logical operation
- these registers can be tested to perform conditional branches
- the most useful condition codes are as follows

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Condition Codes (2)

- as a result of the most recent operation

CF a carry was generated out of the msb
used to detect overflow for unsigned operations

ZF a zero was yielded

SF a negative value was yielded

OF a 2's complement overflow was happened
either neagive or positive

Condition Codes and `c = a+b` (1)

- assume `addl` is used to perform `t = a + b`
and `a`, `b`, `t` are of type `int`

CF	unsigned overflow	<code>(unsigned t) < (unsigned a)</code>
ZF	zero	<code>(t == 0)</code>
SF	negative	<code>(t < 0)</code>
OF	signed overflow	<code>(a < 0 == b < 0) && (t < 0 != a < 0)</code>

Condition Codes and $c = a+b$ (2)

CF	$(\text{unsigned } t) < (\text{unsigned } a)$	$\text{mag}(t) < \text{mag}(a)$ if $C=1$
ZF	$(t == 0)$	zero t
SF	$(t < 0)$	negative t
OF	$(a < 0 == b < 0) \ \&\& \ (t < 0 \ ! \ a < 0)$	$\text{sign}(a) = \text{sign}(b) \ ! \ \text{sign}(t)$

Setting condition codes without altering registers (1)

- Compare and test

<code>cmpb S2, S1</code>	<code>S1 - S2</code>	Compare bytes
<code>cmpw S2, S1</code>	<code>S1 - S2</code>	Compare words
<code>cmpd S2, S1</code>	<code>S1 - S2</code>	Compare double words
<code>testb S2, S1</code>	<code>S1 & S2</code>	Test bytes
<code>testw S2, S1</code>	<code>S1 & S2</code>	Test words
<code>testl S2, S1</code>	<code>S1 & S2</code>	Test double words

Setting condition codes without altering registers (2)

- Compare and test

<code>cmpb S2, S1</code>	<code>-S2 + S1</code>	Compare bytes
<code>cmpw S2, S1</code>	<code>-S2 + S1</code>	Compare words
<code>cmpq S2, S1</code>	<code>-S2 + S1</code>	Compare double words
<code>testb S2, S1</code>	<code>S2 & S1</code>	Test bytes
<code>testw S2, S1</code>	<code>S2 & S1</code>	Test words
<code>testq S2, S1</code>	<code>S2 & S1</code>	Test double words

CMP instruction (1)

- `cmpb op1, op2`
- `cmpw op1, op2`
- `cmpl op1, op2`

- NULL $\$ \leftarrow op2 - op1$
 - subtracts the contents of the *src* operand *op1* from the *dest* operand *op2*
 - discard the results, only the flag register is affected

CMP instruction (2)

- `cmpb op1, op2`
- `cmpw op1, op2`
- `cmpl op1, op2`

Condition	Signed Compare	Unsigned Compare
<code>op1 < op2</code>	<code>ZF == 0 && SF == 0F</code>	<code>CF == 0 && ZF == 0</code>
<code>op1 < op2=</code>	<code>SF == 0F</code>	<code>CF == 0</code>
<code>op1 = op2=</code>	<code>ZF == 1</code>	<code>ZF == 1</code>
<code>op1 > op2=</code>	<code>ZF == 1 or SF != 0F</code>	<code>CF == 1 or ZF == 1</code>
<code>op1 > op2</code>	<code>SF != 0F</code>	<code>CF == 1</code>

- `testb src, dest`
- `testw src, dest`
- `testl src, dest`

- `NULL ← dest & src`
 - ands the contents of the `src` operand with the `dest` operand
 - discard the results, only the flag register is affected

TOC: accessing the condition codes

Set (1)

set(e, z)	D	(equal / zero)	$D \leftarrow ZF$
set(ne, nz)	D	(not equal/ not zero)	$D \leftarrow \sim ZF$
set(s)	D	(negative)	$D \leftarrow SF$
set(ns)	D	(non-negative)	$D \leftarrow \sim SF$
set(g, le)	D	(greater, signed >)	$D \leftarrow \sim (SF \wedge OF) \& \sim ZF$
set(ge, nl)	D	(greater or equal, signed >=)	$D \leftarrow \sim (SF \wedge OF)$
set(l, nge)	D	(less, signed <)	$D \leftarrow SF \wedge OF$
set(le, ng)	D	(less or equal, signed <=)	$D \leftarrow (SF \wedge OF) ZF$
set(a, nbe)	D	(above, unsigned >)	$D \leftarrow \sim CF \& \sim ZF$
set(ae, nb)	D	(above or euqal, unsigned >=)	$D \leftarrow \sim CF$
set(b, nae)	D	(below, unsigned <)	$D \leftarrow CF$
set(be, na)	D	(below or equal, unsigned <=)	$D \leftarrow CF \& \sim ZF$

Set (2)

set(e, z)	D	(equal / zero)	$D \leftarrow ZF$
set(s)	D	(negative)	$D \leftarrow SF$
set(g, le)	D	(greater, signed >)	$D \leftarrow \sim(SF \wedge OF) \& \sim ZF$
set(l, ge)	D	(less, signed <)	$D \leftarrow SF \wedge OF$
set(a, nbe)	D	(above, unsigned >)	$D \leftarrow \sim CF \& \sim ZF$
set(b, nae)	D	(below, unsigned <)	$D \leftarrow CF$
set(ne, nz)	D	(not equal / not zero)	$D \leftarrow \sim ZF$
set(ns)	D	(non-negative)	$D \leftarrow \sim SF$
set(ge, nl)	D	(greater or equal, signed \geq)	$D \leftarrow \sim(SF \wedge OF)$
set(le, ng)	D	(less or equal, signed \leq)	$D \leftarrow (SF \wedge OF) \mid ZF$
set(ae, nb)	D	(above or equal, unsigned \geq)	$D \leftarrow \sim CF$
set(be, na)	D	(below or equal, unsigned \leq)	$D \leftarrow CF \& \sim ZF$

Flag registers (1) - Z, O, S, P

E, Z	Equal, Zero	ZF == 1
NE, NZ	Not Equal, Not Zero	ZF == 0
O	Overflow	OF == 1
NO	No Overflow	OF == 0
S	Signed	SF == 1
NS	Not Signed	SF == 0
P	Parity	PF == 1
NP	No Parity	PF == 0

<https://riptutorial.com/x86/example/6976/flags-register>

Flag registers (2) - unsigned arithmetic

C, B	Carry, Below,	CF == 1
NAE	Not Above or Equal	
NC, NB	No Carry, Not Below,	CF == 0
AE	Above or Equal	
A, NBE	Above, Not Below or Equal	CF==0 and ZF==0
NA, BE	Not Above, Below or Equal	CF==1 or ZF==1

<https://riptutorial.com/x86/example/6976/flags-register>

Flag registers (3) - signed arithmetic

GE, NL	Greater or Equal, Not Less	SF==0F
NGE, L	Not Greater or Equal, Less	SF!=0F
G, NLE	Greater, Not Less or Equal	ZF==0 and SF==0F
NG, LE	Not Greater, Less or Equal	ZF==1 or SF!=0F

<https://riptutorial.com/x86/example/6976/flags-register>

Flag registers (4)

- The condition codes are grouped into three blocks :

Z, O, S, P	Zero Overflow Sign Parity
unsigned arithmetic	Above Below
signed arithmetic	Greater Less

- JB would be "Jump if Below" (**unsigned**)
- JL would be "Jump if Less" (**signed**)

<https://riptutorial.com/x86/example/6976/flags-register>

Flag registers (3)

- In 16 bits, subtracting 1 from 0

from	to	
0	65,535	unsigned arithmetic
0	-1	signed arithmetic
0x0000	0xFFFF	bit representation

- It's only by interpreting the condition codes that the meaning is clear.
- 1 is subtracted from 0x8000:

from	to	
32,768	32,767	unsigned arithmetic
-32,768	32,767	signed arithmetic
0x8000	0x7FFF	bit representation

(0111 1111 1111 1111 + 1 = 1000 0000 0000 0000)

<https://riptutorial.com/x86/example/6976/flags-register>

- accessing the condition codes
 - to read the condition codes directly
 - to set an integer register
 - to perform a conditional branch

based on some combination of condition codes

Set (4)

- the set instructions set a single byte to 0 or 1 depending on some combination of the **condition codes**
- the destination operand D is
 - either one of the eight single byte register elements
 - or a memory location where the single byte is to be stored
- to generate a 32-bit result, the high-order 24-bits must be *cleared*

a typical assembly for a c predicate

```
; a is in %edx  
; b is in %eax  
  
cml     %eax, %edx      ; compare a and b ; (a - b)  
setl    %al             ; set low order byte of %eax to 0 or 1  
movzbl  %al, %eax      ; set remaining bytes of %eax to 0
```

- movzbl instruction is used to clear the high-order three bytes
- | set(1, ge) | D | (less, signed <) | $D \leftarrow SF \oplus OF$ |

movz instrucion (1)

- Purpose: To convert an unsigned integer to a wider unsigned integer
- opcode `src.rx, dst.wy`
- `dst <- zero extended src;`

- MOVZBW (Move Zero-extended Byte to Word) 8-bit zero **BW**
- MOVZBL (Move Zero-extended Byte to Long) 24-bit zero **BL**
- MOVZWL (Move Zero-extended Word to Long) 16-bit zero **WL**

movz instrucion (2)

- MOVZ **BW** (Move Zero-extended Byte to Word) 8-bit zero
 - the low 8 bits of the destination are replaced by the source operand
 - the top 8 bits are set to 0.
- MOVZ **BL** (Move Zero-extended Byte to Long) 24-bit zero
 - the low 8 bits of the destination are replaced by the source operand.
 - the top 24 bits are set to 0.
- MOVZ **WL** (Move Zero-extended Word to Long) 16-bit zero
 - the low 16 bits of the destination are replaced by the source operand.
 - the top 16 bits are set to 0.
- The source operand is unaffected.

register operand types (1)

byte 3	byte 2	byte 1	byte 0
		%ah	%al
		%ax_1	%ax_0
%eax_3	%eax_2	%eax_1	%eax_0
		%ch	%cl
		%cx_1	%cx_0
%ecx_3	%ecx_2	%ecx_1	%ecx_0
		%dh	%dl
		%dx_1	%dx_0
%edx_3	%edx_2	%edx_1	%edx_0
		%bh	%bl
		%bx_1	%bx_0
%ebx_3	%ebx_2	%ebx_1	%ebx_0

register operand types (2)

byte 3	byte 2	byte 1	byte 0
		%si_1	%si_0
%esi_3	%esi_2	%esi_1	%esi_0
		%di_1	%di_0
%edi_3	%edi_2	%edi_1	%edi_0
		%sp_1	%sp_0
%esp_3	%esp_2	%esp_1	%esp_0
		%bp_1	%bp_0
%ebp_3	%ebp_2	%ebp_1	%ebp_0

register operand types (3)

byte 3	byte 2	byte 1	byte 0
		%ah	%al
		%ch	%cl
		%dh	%dl
		%bh	%bl
<hr/>			
		%ax_1	%ax_0
		%cx_1	%cx_0
		%dx_1	%dx_0
		%bx_1	%bx_0
<hr/>			
		%si_1	%si_0
		%di_1	%di_0
		%sp_1	%sp_0
		%bp_1	%bp_0

register operand types (4)

byte 3	byte 2	byte 1	byte 0
%eax_3	%eax_2	%eax_1	%eax_0
%ecx_3	%ecx_2	%ecx_1	%ecx_0
%edx_3	%edx_2	%edx_1	%edx_0
%ebx_3	%ebx_2	%ebx_1	%ebx_0
%esi_3	%esi_2	%esi_1	%esi_0
%edi_3	%edi_2	%edi_1	%edi_0
%esp_3	%esp_2	%esp_1	%esp_0
%ebp_3	%ebp_2	%ebp_1	%ebp_0

Set (6)

- for some of the underlying machine instructions, there are multiple possible names (synonyms),
 - `setg` (set greater)
 - `setnle` (set not less or equal)
- compilers and disassemblers make arbitrary choices of which names to use

Set (7)

- although all arithmetic operations set the condition codes, the descriptions of the different set commands apply to the case where a comparison instruction has been executed, setting the condition codes according to the computation $t = a - b$
- for example, consider the `sete`, or "Set when equal" instruction
- when $a = b$, we will have $t = 0$, and hence the zero flag indicates equality

Set (8)

- Similarly, consider testing a signed comparison with the set1 or "Set when less"
- when a and b are in two's complement form, then for $a < b$ we will have $a - b < 0$ if the true difference were computed
- when there is no overflow, this would be indicated by having the sign flag set

Set (9)

- when there is positive overflow, because $a - b$ is a large positive number, however, we will have $t < 0$
- when there is negative overflow, because $a - b$ is a small negative number, we will have $t > 0$
- in either case, the sign flag will indicate the opposite of the sign of the true difference

Set (10)

- in either case, the sign flag will indicate the opposite of the sign of the true difference
- hence, the Exclusive-Or of the overflow and sign bits provides a test for whether $a < b$
- the other signed comparison tests are based on other combinations of $SF \oplus OF$ and ZF

Set (11)

- for the testing of unsigned comparisons, the carry flag will be set by the `cmpl` instruction when the integer difference $a - b$ of the unsigned arguments a and b would be negative, that is when $(\text{unsigned}) a < (\text{unsigned}) b$
- thus, these tests use combinations of the carry and zero flags

TOC: Jump instructions

Jump instruction encoding

- a jump instruction can cause the execution to switch to a completely new position in the program
- these jump destinations are generally indicated by a **label**
- in generating the object code file
 - the **assembler** determines the addresses of all **labeled** instructions
 - and **encodes** the jump targets as part of the jump instruction

jmp instruction

- the `jmp` instruction jumps unconditionally
- **direct** jump
 - the jump target is encoded as part of instruction
 - give a label as the jump target
- **indirect** jump
 - the jump target is read from a register or a memory location
 - using `*` followed by an operand specifier
 - `jmp %eax` uses the value in register `%eax` as the jump target
 - `jmp *(%eax)` reads the jump target from memory using the value in `%eax` as the read address

Conditional jump instructions

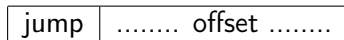
- the other jump instructions either jump or continue executing at the next instruction in the code sequence depending on some combination of the **condition codes**
- like set instruction
- the underlying machine instructions have multiple names
- **conditional** jumps can only be **direct**

Jump instructions

jmp Label		(1)	direct
jmp *Operand		(1)	indirect
je Label	jz	(ZF)	equal/zero
jne Label	jnz	(~ZF)	not equal / non-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))	less or 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 <=

PC-relative addressing

- jump relative



- **effective** PC address = next instruction address + offset
(offset may be negative)
- particularly useful in connection with jumps,
because typical jumps are to nearby instructions
- most `if` or `while` statements are reasonably short
- another advantage is `+position-independent_code`

https://en.wikipedia.org/wiki/Addressing_mode#PC-relative

Encoding format of object code

- the format of object code
- understanding how the targets of jump instructions are encoded will be important
 - when studying linking process
 - interpreting the output of a disassembler
- In assembly code, jump targets are written using symbolic labels
- the assembler, and later the linker, generate the proper encodings of the jump targets
- there are several different encodings of for jumps, but some of the most commonly used ones are **PC-relative**

Computer Architecture : A Programmer's Perspective

Encoding jump instructions

- **PC-relative**
 - encodes the difference between the address of the target instruction the address of the instruction immediately following the jump
 - these offsets can be encoded using 1, 2, or 4 bytes
- **Absolute**
 - directly specify the target address using 4 bytes
- the assembler and linker select the appropriate encodings

TOC: PC-relative addressing example

an example of PC-relative addressing

```
jle .L4           If <=, goto dest2
    .p2align 4,,7  Aligns next instruction to multiple of 8
.L5:             dest1:
    movl %edx, %eax
    sarl $1, %eax
    subl %eax, %edx
    jg .L5        If >, goto dest1
.L4:             dest2:
    movl %edx, %eax
```

```

    jle .L4
    .p2align 4,,7
.L5:      dest1:
    movl %edx, %eax    0111 -> 0111    0100 -> 0100    0010 -> 0010    0001 -> 0001
    sarl $1, %eax     0111 -> 0011    0100 -> 0010    0010 -> 0001    0001 -> 0000
    subl %eax, %edx   -> 0100          -> 0010          -> 0001          -> 0001
    jg .L5            If >, goto dest1
.L4:      dest2:
    movl %edx, %eax

    jle .L4
    .p2align 4,,7
.L5:      dest1:
    movl %edx, %eax    0100-> 0100    0010 -> 0010    0001 -> 0001    0001 -> 0001
    sarl $1, %eax     0100-> 0010    0010 -> 0001    0001 -> 0000    0001 -> 0000
    subl %eax, %edx   -> 0010          -> 0001          -> 0001          -> 0001
    jg .L5            If >, goto dest1
.L4:      dest2:
    movl %edx, %eax

```

```
1. 8: 7e 11          jle 1b <silly+0x1b>    Target = dest2
2. a: 8d b6 00 00 00 00 lea 0x0(%esi),%esi    Added nops
3. 10: 89 d0         mov %edx,%eax         dest1:
4. 12: c1 f8 01      sar $0x1,%eax
5. 15: 29 c2         sub %eax,%edx
6. 17: 85 d2         test %edx,%edx        %edx & %edx = %edx
7. 19: 7f f5         jg 10 <silly+0x10>    Target = dest1
8. 1b: 89 d0         mov %edx,%eax         dest2:
```

Computer Architecture : A Programmer's Perspective

disassembled version notes (1)

```
1. 8: 7e 11                jle 1b <silly+0x1b>    Target = dest2
```

- no real effects, serves as 6-byte nop
- to make the address of the next instruction a multiple of 16

Computer Architecture : A Programmer's Perspective

disassembled version notes (2)

```
1. 8: 7e 11                jle  1b <silly+0x1b>   Target = dest2
2. a: 8d b6 00 00 00 00    lea  0x0(%esi),%esi    Added nops
```

- jump target : 0x1b (27)
- jump target encoding : 0x11 + 0xa = 0x1b (17 + 10 =27)
- next instruction address : 0xa (10)

Computer Architecture : A Programmer's Perspective

disassembled version notes (3)

```
7. 19: 7f f5          jg    10 <silly+0x10>   Target = dest1
8. 1b: 89 d0          mov   %edx,%eax        dest2:
```

- jump target : 0x10 (16)
- jump target encoding : 0xf5 + 0x1b = 0xf5 (-11 + 27 =16)
- next instruction address : 0x1b (27)

Computer Architecture : A Programmer's Perspective

- the value of the program counter when performing PC-relative addressing is the address of the instruction following the jump not the address of the jump instruction
- the processor would update the program counter as its first step in executing an instruction

Computer Architecture : A Programmer's Perspective

disassembled version after linking

```
1. 80483c8: 7e 11          jle 79473db <silly+0x1b> Target = dest2
2. 80483ca: 8d b6 00 00 00 00 lea 0x0(%esi),%esi      Added nops
3. 80483d0: 89 d0          mov %edx,%eax          dest1:
4. 80483d2: c1 f8 01      sar $0x1,%eax
5. 80483d5: 29 c2          sub %eax,%edx
6. 80483d7: 85 d2          test %edx,%edx         %edx & %edx = %edx
7. 80483d9: 7f f5          jg 80483d0 <silly+0x10> Target = dest1
8. 80483db: 89 d0          mov %edx,%eax          dest2:
```

Computer Architecture : A Programmer's Perspective

disassembled version after linking notes

```
1. 8: 7e 11          jle  1b <silly+0x1b>   Target = dest2
7. 19: 7f f5         jg   10 <silly+0x10>   Target = dest1
1. 80483c8: 7e 11         jle  79473db <silly+0x1b> Target = dest2
7. 80483d9: 7f f5         jg   80483d0 <silly+0x10> Target = dest1
```

- the instructions have been relocated to different addresses, but the encodings of the jump targets in line 1 and line 7 remain unchanged
- by using PC-relative encoding of the jump targets, the instructions can be completely encoded (requiring just two bytes) and the object code can be shifted to different positions in memory without modification.

TOC: Translating conditional branches

if-else statement

```
if (expr)
  then-statement
else
  else-statement
```

```
t = exor1
if (t)
  goto true;
  // else-statement
  goto done;
true:
  // then-statement
done:
```

if-else exmple (1)

```
int abs(int x, int y)
{
    if (x<y)
        return y-x;
    else
        return x-y;
}
```

```
int abs_goto(int x, int y)
{
    int val;

    if (x < y)
        goto true;
    val = x - y;
    goto done;
true:
    val = y - x;
done:
    return val;
}
```


if-else exmple (2)

```
movl 8(%ebp), %edx
movl 12(%ebp), %eax
cmpl %eax, %edx
jl  .L3
subl %eax, %edx
movl %edx, %eax
jmp  .L5
.L3:
subl %edx, %eax
.L5:
```

```
int abs_goto(int x, int y)
{
    int val;

    if (x < y)
        goto true;
    val = x - y;
    goto done;
true:
    val = y - x;
done:
    return val;
}
```

TOC: Loop instructions

do-while statement

```
do
    // body-statement
while (expr);
```

```
loop:
    // body-statement
    t = expr;
    if (t)
        goto loop;
```

while statement (1)

```
while (expr)
{
    // body-statement
}
```

```
loop:
    t = expr;
    if (!t)
        goto done;
    // body-statement
    goto loop;
done:
```

while statement (2)

```
if (!expr)
    goto done;
do
    // body-statement
while (expr);
done:
```

```
t = expr;
if (!t)
    goto done;
loop:
    // body-statement
    t = expr;
    if (t)
        goto loop;
done;
```

for statement (1)

```
for (init; test; update)
  // body-statement
```

```
init_expr;
while (test) {
  // body-statement
  update;
}
```

for statement (2)

```
init_expr;  
if (!test)  
    goto done;  
do {  
    // body-statement  
    update;  
} while (test);  
done:
```

```
init_expr;  
t = test;  
if (!t)  
    goto done;  
loop:  
    // body-statement  
    update_expr;  
    t = test;  
    if (t)  
        goto loop;  
done;
```


Switch statement (1)

```
int switch_example(int x)
{
    int result = x;

    switch (x) {
        case 100: result *= 13; break;
        case 102: result += 10;
        case 103: result += 11; break;
        case 104:
        case 105: result *= result; break;
        default: result = 0;
    }

    return result;
}
```

Switch statement (2)

```
code *jt[7] =  
{loc_A, loc_def, loc_B, loc_C, loc_D, loc_def, loc_D };
```

```
int switch_tanslated(int x)  
{  
    unsigned xi = x - 100;  
    int result = x;  
    if (xi>6) goto loc_def;  
    goto jt[xi];  
loc_A: result *= 13; goto done;  
loc_B: result += 10; goto done;  
loc_C: result += 11; goto done;  
loc_D: result *= result; goto done;  
loc_def: result = 0;  
done: return result;  
}
```

Switch statement (3)

```
leal -100(%edx), %eax    ;; xi = x-100
cpl $6, %eax            ;; compare xi:6
ja .L9                  ;; if >, go to loc_def
jmp *.L10(,%eax,4)      ;; goto jt[xi]

;; Case 100
.L4:                    ;; loc_A
leal (%edx,%edx,2), %eax ;; 3*x
leal (%edx,%eax,4), %edx ;; x+4*3x
jmp .Le                 ;; goto done

;; Case 102
.L5:                    ;; loc_B
addl $10, %edx          ;; result += 10
```

Switch statement (4)

```
;; Case 103
.L6:                ;; loc_C
addl $11, %edx      ;; result += 11
jmp .L3             ;; goto done

;; Cases 104, 106
.L8:                ;; loc_D
imull %edx, %edx    ;; result *= result
jmp .L3             ;; goto done

;; Default Case
.L9:                ;; loc_defa
xorl %edx, %edx     ;; result = 0
```

Switch statement (5)

```
;; Return Result  
.L3:                ;; done  
movl %edx, %eax    ;; set return value
```