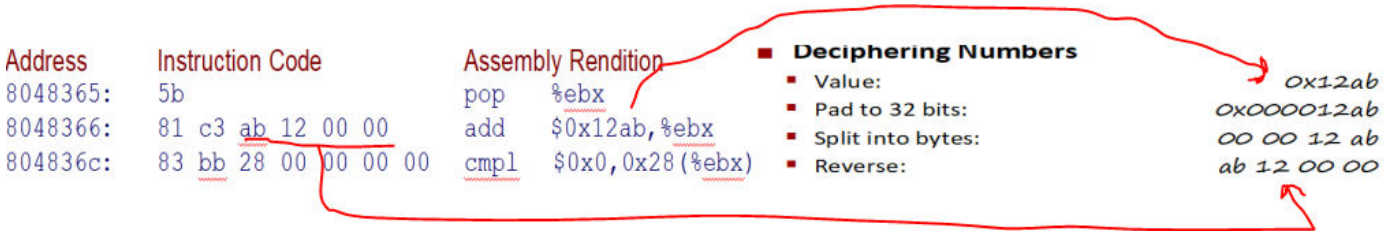
Ders01 - BITS, BYTES & INTEGERS:

- Byte = 8 bits.

ADDRESS	0000	0001	0002	0003	0004	0005	0006	0007	0008	0009	0010	0011	0012	0013	0014	0015
32-BIT WORDS	Address == 0000				Address == 0004				Address == 0008				Address == 0012			
64-BIT WORDS	Address == 0000								Address == 0008							

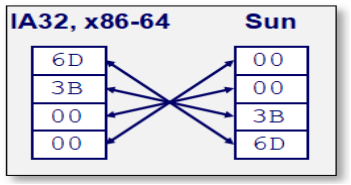
C Data Type	Typical 32-bit	Intel IA32	x86-64
char	1	1	1
short	2	2	2
int	4	4	4
long	4	4	8
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/12	10/16
pointer	4	4	8

- Binary $(00000000)_2$ to $(11111111)_2$
- Decimal: $(0)_{10}$ to $(255)_{10}$
- Hexadecimal $(00)_{16}$ to $(FF)_{16}$
- **Byte Sıralama**
  - ✓ Big Endian; En büyük adres en sonda. 4 byte'lı bir 0x01234567 sayısı için >>> 01 23 45 67
  - ✓ Little Endian; En küçük adres en sonda. 4 byte'lı bir 0x01234567 sayısı için >>> 67 45 23 01
- **Disassembly**
  - ✓ Makine kodunun yazdığımız dile geri çevirilmesi.
- **Reading Byte-Reversed Listings**

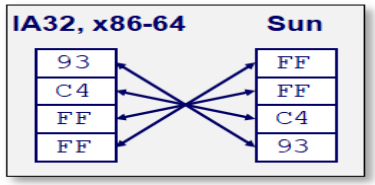


Decimal:	15213
Binary:	0011 1011 0110 1101
Hex:	3 B 6 D

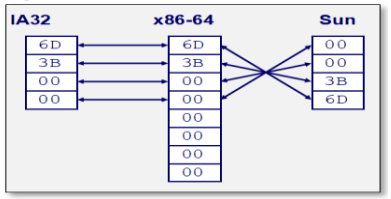
int A = 15213;



int B = -15213;



long int C = 15213;



➢ Boolean Algebra

And

- A&B = 1 when both A=1 and B=1

&	0	1
0	0	0
1	0	1

Or

- A|B = 1 when either A=1 or B=1

|	0	1
0	0	1
1	1	1

Not

- ~A = 1 when A=0

~	
0	1
1	0

Exclusive-Or (Xor)

- A^B = 1 when either A=1 or B=1, but not both

^	0	1
0	0	1
1	1	0

➢ Bitwise Operations

x	y	x|y	x&y	x^y
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	1	1	0

Logic Operations

Operatör	Sembol	Kullanılışı	İşlem sonucu
Mantıksal AND	&&	a && b	a ve b'nin her ikisi sıfırdan farklı ise sonuç 1 aksi takdirde 0
Mantıksal OR	||	a || b	a veya b'den biri sıfırdan farklı ise sonuç 1 aksi takdirde sonuç 0
Mantıksal değil	!	! a	a sıfırsa sonuç 1 aksi takdirde sonuç 0

Shift Operations (Aritmetik kaydırma; en yüksek anlamlı biti kaydırır.)

Argument x	01100010
<< 3	00010000
Log. >> 2	00011000
Arith. >> 2	00011000

Argument x	10100010
<< 3	00010000
Log. >> 2	00101000
Arith. >> 2	11101000

➢ Unsigned & Signed Integers

Two's Complement (Ters Çevir, 1 Arttır!)

- ✓ B2U(x) = Binary to Unsigned
- ✓ B2T(x) = Binary to Two's Complement
- ✓ UMAX = Unsigned Max [0-255]
- ✓ TMAX = Two's Complement Max. [128]
- ✓ TMIN = Two's Complement Min. [-127]

X	B2U(X)	B2T(X)
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	–8
1001	9	–7
1010	10	–6
1011	11	–5
1100	12	–4
1101	13	–3
1110	14	–2
1111	15	–1

x = 15213

	Decimal	Hex	Binary
x	15213	3B 6D	00111011 01101101
~x	-15214	C4 92	11000100 10010010
~x+1	-15213	C4 93	11000100 10010011
y	-15213	C4 93	11000100 10010011

x = 0

	Decimal	Hex	Binary
0	0	00 00	00000000 00000000
~0	-1	FF FF	11111111 11111111
~0+1	0	00 00	00000000 00000000

- **Binary Arithmetic** (https://www.youtube.com/watch?v=mZE_w5L-hyU)
- **Two's complement Arithmetic** (https://www.youtube.com/watch?v=-46X79rX9B4)

Ders02 - FLOTING POINTS:

Value	Representation
5 3/4	101.11_2
2 7/8	10.111_2
63/64	0.111111_2

Value	Representation
1/3	$0.0101010101[01]..._2$
1/5	$0.001100110011[0011]..._2$
1/10	$0.0001100110011[0011]..._2$

Numerical Form:

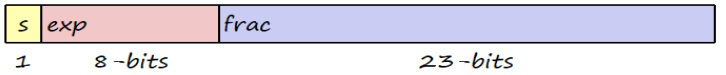
$$(-1)^s M\ 2^E$$

- Sign bit s determines whether number is negative or positive
- Significand M normally a fractional value in range [1.0,2.0).
- Exponent E weights value by power of two

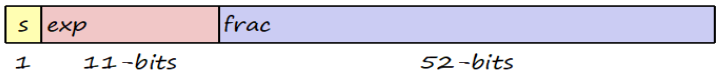
Encoding

- MSB s is sign bit s
- *exp* field encodes E (but is not equal to E)
- *frac* field encodes M (but is not equal to M)

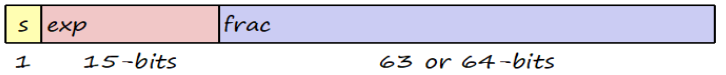
- Single precision: 32 bits



- Double precision: 64 bits



- Extended precision: 80 bits (Intel only)



- **Normalized Encoding:** https://www.youtube.com/watch?v=8afbTaA-gOQ

Ders03 - MACHINE BASICS:

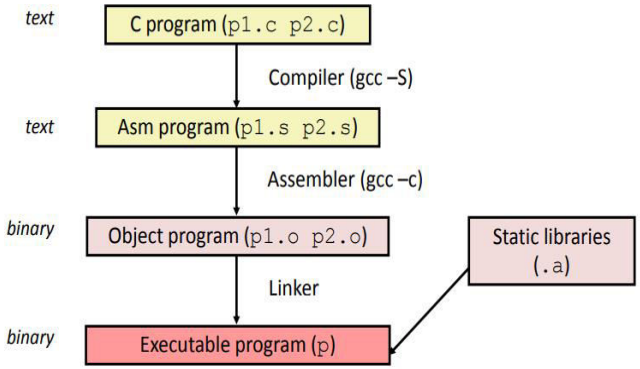
➢ Turning C into Object Code

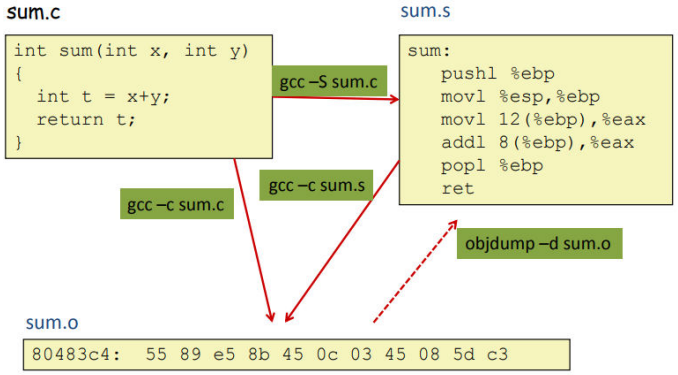
– Code in files `p1.c p2.c`

– Compile with command: `gcc -O1 p1.c p2.c -o p`

Optimization level

Output file is p





Note: If your platform is 64-bit, you may want to force it to generate 32-bit assembly by **gcc –m32 –S sum.c** to get the above output.

Machine Instruction Example

```
int t = x+y;
```

```
addl 8(%ebp),%eax
```

Similar to expression:

```
x += y
```

More precisely:

```
int eax;
int *ebp;
eax += ebp[2]
```

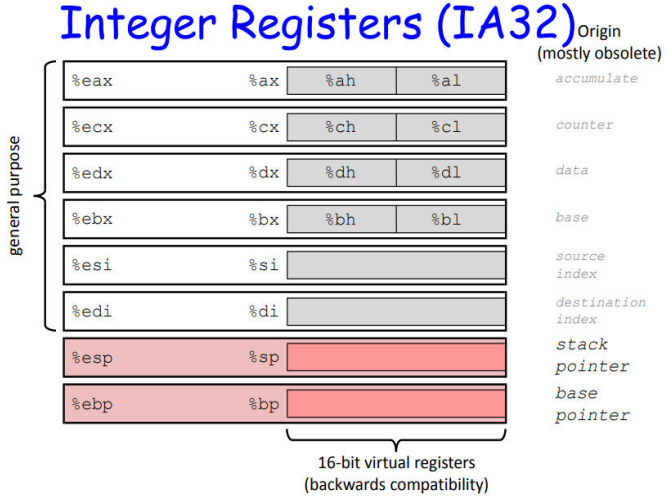
```
0x80483ca: 03 45 08
```

- **C Code**
  - Add two signed integers
- **Assembly**
  - Add 2 4-byte integers
    - "Long" words in GCC parlance
    - Same instruction whether signed or unsigned
  - Operands:
    - `x`: Register `%eax`
    - `y`: Memory `M[%ebp+8]`
    - `t`: Register `%eax`
      - Return function value in `%eax`
- **Object Code**
  - 3-byte instruction
  - Stored at address `0x80483ca`

➢ Integer Registers (IA32)

Moving Data (IA32)

Integer Registers (IA32)



- **`movl` *Source, Dest***
- Operand Types
  - ***Immediate:*** Integer constant
    - e.g. `$0x400`
  - ***Register:*** One of 8 integer registers
    - e.g. `%eax`
  - ***Memory:*** 4 consecutive bytes of memory at address given by register
    - Simplest example (`%eax`)

`movl` Operand Combinations

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

No memory-to-memory instruction

➢ Memory Addressing Modes

- Normal (R) Mem[Reg[R]]
  - Register R specifies memory address

```
movl (%ecx),%eax
```

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

```
movl 8(%ebp),%edx
```

➢ Using Simple Addressing Modes (Ders Slaytı: 30-38.sayfa)

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

```
swap:
    pushl %ebp
    movl  %esp,%ebp
    pushl %ebx

    movl  8(%ebp), %edx
    movl  12(%ebp), %ecx
    movl  (%edx), %ebx
    movl  (%ecx), %eax
    movl  %eax, (%edx)
    movl  %ebx, (%ecx)

    popl  %ebx
    popl  %ebp
    ret
```

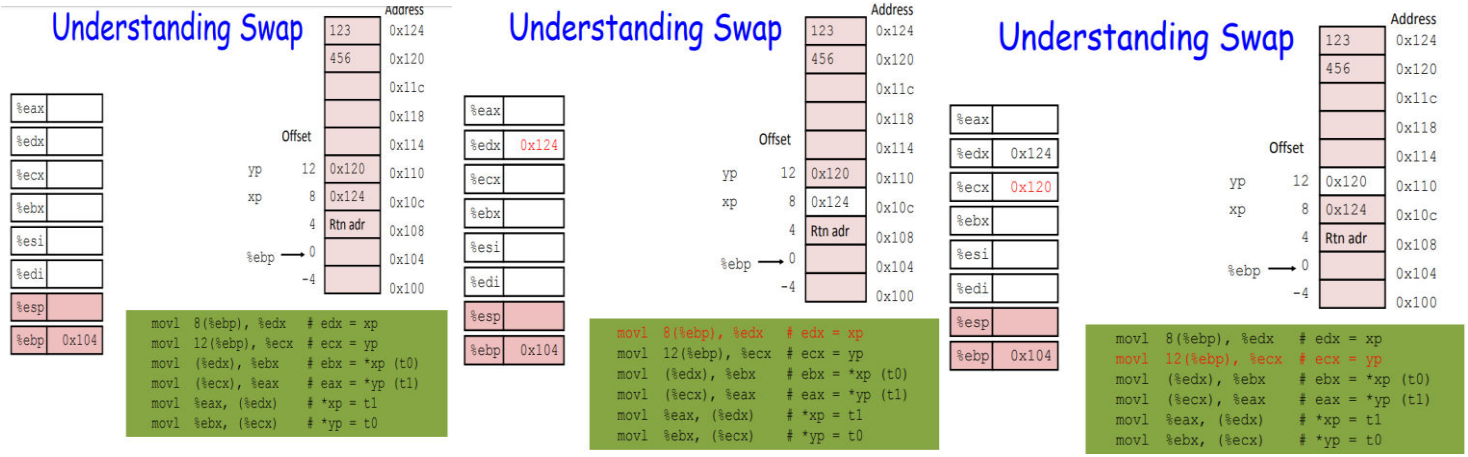
Set Up

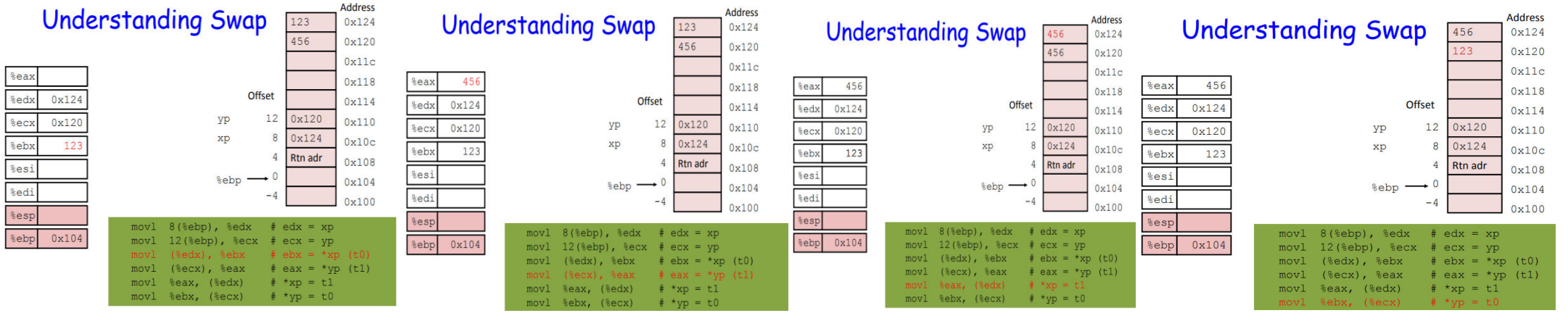
Point to first argument

Point to second argument

Body

Finish





➢ Data Representations (IA32 && x86-64)

C Data Type	Generic 32-bit	Intel IA32	x86-64
• unsigned	4	4	4
• int	4	4	4
• long int	4	4	8
• char	1	1	1
• short	2	2	2
• float	4	4	4
• double	8	8	8
• char *	4	4	8

— Or any other pointer

x86-64 Registers (Long word l (4 Bytes) ↔ Quad word q (8 Bytes))

%rax	%eax
%rbx	%ebx
%rcx	%ecx
%rdx	%edx
%rsi	%esi
%rdi	%edi
%rsp	%esp
%rbp	%ebp

%r8	%r8d
%r9	%r9d
%r10	%r10d
%r11	%r11d
%r12	%r12d
%r13	%r13d
%r14	%r14d
%r15	%r15d

- Extend existing registers. Add 8 new ones.
- Make %ebp/%rbp general purpose

➢ **Example: Swap Function (32Bit vs. 64Bit)**

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

```
swap:
  pushl %ebp
  movl  %esp,%ebp
  pushl %ebx

  movl  8(%ebp), %edx
  movl  12(%ebp), %ecx
  movl  (%edx), %ebx
  movl  (%ecx), %eax
  movl  %eax, (%edx)
  movl  %ebx, (%ecx)

  popl  %ebx
  popl  %ebp
  ret
```

Set Up / Body / Finish

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

```
swap_l:

  movq    (%rdi), %rdx
  movq    (%rsi), %rax
  movq    %rax, (%rdi)
  movq    %rdx, (%rsi)

  ret
```

Set Up / Body / Finish

- **64-bit data**
  - Data held in registers `%rax` and `%rdx`
  - `movq` operation
    - "q" stands for quad-word

Ders04 - ARITHMETIC CONTROL PROCEDURES:

➢ **"leal Src,Dest" (Leal fonksiyonu "x+k*y" ifadesini hesaplamada kullanılır.)**

- **Example**

```
int mul12(int x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leal  (%eax,%eax,2), %eax   ;t <- x+x*2
sall  $2, %eax              ;return t<<2
```

➢ **Arithmetic Operations**

Format	Computation		
addl	Src,Dest	Dest = Dest + Src	
subl	Src,Dest	Dest = Dest – Src	
imull	Src,Dest	Dest = Dest * Src	
sall	Src,Dest	Dest = Dest << Src	Also called shll
sarl	Src,Dest	Dest = Dest >> Src	Arithmetic
shrl	Src,Dest	Dest = Dest >> Src	Logical
xorl	Src,Dest	Dest = Dest ^ Src	
andl	Src,Dest	Dest = Dest & Src	
orl	Src,Dest	Dest = Dest | Src	

incl	Dest	Dest = Dest + 1
decl	Dest	Dest = Dest – 1
negl	Dest	Dest = – Dest
notl	Dest	Dest = ~Dest

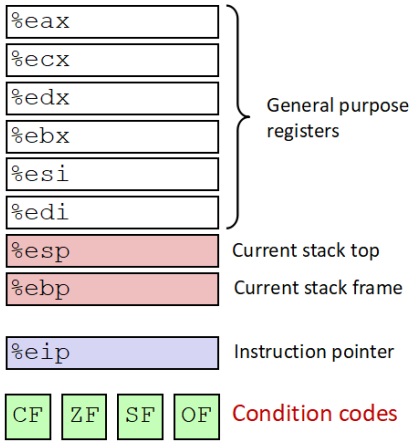
```
int arith(int x, int y, int z)
{
  int t1 = x+y;
  int t2 = z+t1;
  int t3 = x+4;
  int t4 = y * 48;
  int t5 = t3 + t4;
  int rval = t2 * t5;
  return rval;
}
```

```
arith:
  pushl %ebp
  movl  %esp, %ebp

  movl  8(%ebp), %ecx
  movl  12(%ebp), %edx
  leal  (%edx,%edx,2), %eax
  sall  $4, %eax
  leal  4(%ecx,%eax), %eax
  addl  %ecx, %edx
  addl  16(%ebp), %edx
  imull %edx, %eax

  popl  %ebp
  ret
```

Set Up / Body / Finish

➢ **Processor State (IA32, Partial)** **Condition Codes (Implicit)** **Condition Codes (Explicit)**

- **Information about currently executing program**
  - Temporary data (%eax, …)
  - Location of runtime stack (%ebp,%esp)
  - Location of current code control point (%eip, …)
  - Status of recent tests (CF, ZF, SF, OF)



Condition Codes

- Single Bit Registers
  - CF Carry Flag
  - SF Sign Flag
  - ZF Zero Flag
  - OF Overflow Flag
- Implicitly Set By Arithmetic Operations
  - addl *Src,Dest*
  - C analog: t = a + b
  - CF set if carry out from most significant bit
    - Used to detect unsigned overflow
  - ZF set if t == 0
  - SF set if t < 0
  - OF set if two's complement overflow
    (a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)
- *Not* Set by leal instruction

- Explicit Setting by Compare Instruction
  cmpl *Src2,Src1*
  - cmpl 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
  - OF set if two's complement overflow
    (a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)

➢ SetX Instruction

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)

jX (Jump) Instruction

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)

Example

```
int max(int x, int y)
{
  if (x > y)
    return x;
  else
    return y;
}
```

```
_max:
        pushl %ebp          } Set
        movl %esp,%ebp      } Up
        movl 8(%ebp),%edx   }
        movl 12(%ebp),%eax  }
        cmpl %eax,%edx      } Body
        jle L9              }
        movl %edx,%eax      }
L9:
        movl %ebp,%esp      }
        popl %ebp           } Finish
        ret                 }
```

➢ "goto"

```
int goto_max(int x, int y)
{
  int rval = y;
  int ok = (x <= y);
  if (ok)
    goto done;
  rval = x;
done:
  return rval;
}
```

- C allows "goto" as means of transferring control
  - Closer to machine-level programming style
- Generally considered bad coding style

```
    movl 8(%ebp),%edx    # edx = x
    movl 12(%ebp),%eax   # eax = y
    cmpl %eax,%edx       # x : y
    jle L9               # if <= goto L9
    movl %edx,%eax       # eax = x   } Skipped when x ≤ y
L9:                  # Done:
```

"Do-While" vs. "goto"

C Code

```
int fact_do
  (int x)
{
  int result = 1;
  do {
    result *= x;
    x = x-1;
  } while (x > 1);
  return result;
}
```

Goto Version

```
int fact_goto(int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Use backward branch to continue looping
- Only take branch when "while" condition holds

Conditional Move Operation

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

x in %edi

y in %esi

```
absdiff:
  movl   %edi, %edx
  subl   %esi, %edx  # tval = x-y
  movl   %esi, %eax
  subl   %edi, %eax  # result = y-x
  cmpl   %esi, %edi  # Compare x:y
  cmovg  %edx, %eax  # If >, result = tval
  ret
```

➢ **"Do-While"ı "goto" İle Tanımlama** **"goto" >>> "Do-While"**

Goto Version

```
int fact_goto
  (int x)
{
  int result = 1;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
  return result;
}
```

- Registers

  %edx x

Assembly

```
_fact_goto:
  pushl %ebp          # Setup
  movl %esp,%ebp      # Setup
  movl $1,%eax        # eax = 1
  movl 8(%ebp),%edx   # edx = x

L11:
  imull %edx,%eax     # result *= x
  decl %edx           # x--
  cmpl $1,%edx        # Compare x : 1
  jg L11              # if > goto loop

  movl %ebp,%esp      # Finish
  popl %ebp           # Finish
  ret                 # Finish
```

C Code

```
do
  Body
  while (Test);
```

Goto Version

```
loop:
  Body
  if (Test)
    goto loop
```

➢ **"While" Loop** **"While" >>> "goto"**

C Code

```
int fact_while
  (int x)
{
  int result = 1;
  while (x > 1) {
    result *= x;
    x = x-1;
  };
  return result;
}
```

First Goto Version

```
int fact_while_goto
  (int x)
{
  int result = 1;
loop:
  if (!(x > 1))
    goto done;
  result *= x;
  x = x-1;
  goto loop;
done:
  return result;
}
```

While version

```
while (Test)
  Body
```

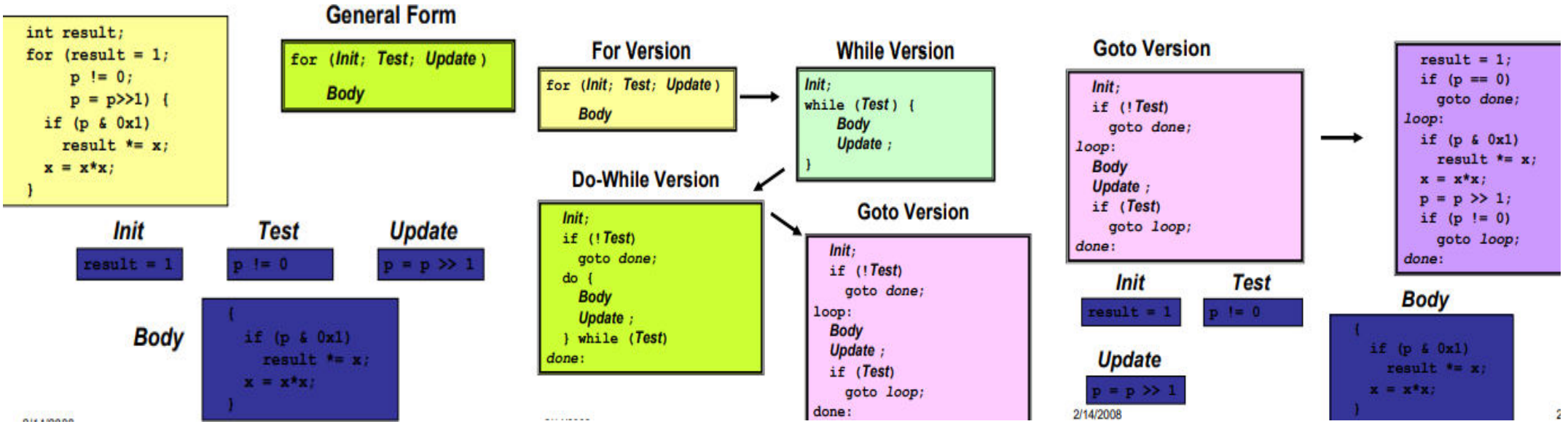
Do-While Version

```
  if (!Test)
    goto done;
  do
    Body
    while(Test);
done:
```

Goto Version

```
  if (!Test)
    goto done;
loop:
  Body
  if (Test)
    goto loop;
done:
```

➢ **"For" Loop** **"For" >>>>> "goto"** **"For" Loop'u "goto" İle Tanımlama**



Ders05 - SWITCH STATEMENT & IA32 PROCEDURES

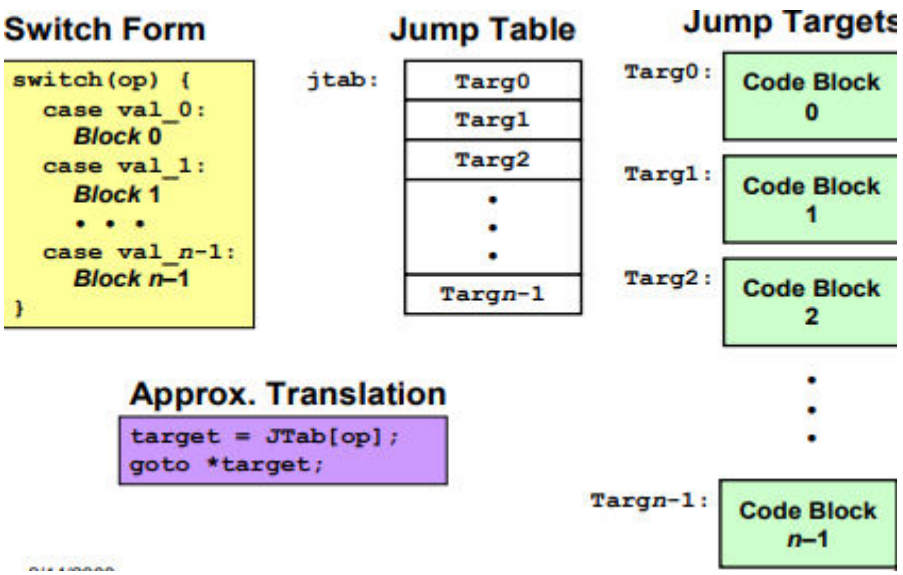
➢ **Switch Statement**

```
typedef enum
 {ADD, MULT, MINUS, DIV, MOD, BAD}
    op_type;

char unparse_symbol(op_type op)
{
  switch (op) {
  case ADD :
    return '+';
  case MULT:
    return '*';
  case MINUS:
    return '-';
  case DIV:
    return '/';
  case MOD:
    return '%';
  case BAD:
    return '?';
  }
}
```

Switch Statements

- Implementation Options
  - Series of conditionals
    - Good if few cases
    - Slow if many
  - Jump Table
    - Lookup branch target
    - Avoids conditionals
    - Possible when cases are small integer constants
  - GCC
    - Picks one based on case structure
  - Bug in example code
    - No default given

Jump Table Structure



➢ **Example ^^^^**

<<<<< Assembly Setup

Table Contents

```
.section .rodata
   .align 4
.L57:
  .long .L51 #Op = 0
  .long .L52 #Op = 1
  .long .L53 #Op = 2
  .long .L54 #Op = 3
  .long .L55 #Op = 4
  .long .L56 #Op = 5
```

Enumerated Values

```
ADD     0
MULT    1
MINUS   2
DIV     3
MOD     4
BAD     5
```

Targets & Completion

```
.L51:
    movl $43,%eax # '+'
    jmp .L49
.L52:
    movl $42,%eax # '*'
    jmp .L49
.L53:
    movl $45,%eax # '-'
    jmp .L49
.L54:
    movl $47,%eax # '/'
    jmp .L49
.L55:
    movl $37,%eax # '%'
    jmp .L49
.L56:
    movl $63,%eax # '?'
    # Fall Through to .L49
```

- Symbolic Labels
  - Labels of form `.LXX` translated into addresses by assembler
- Table Structure
  - Each target requires 4 bytes
  - Base address at `.L57`
- Jumping

  `jmp .L49`
  - **Jump target is denoted by label** `.L49`

  `jmp *.L57(,%eax,4)`
  - **Start of jump table denoted by label** `.L57`
  - **Register** `%eax` **holds** `op`
  - **Must scale by factor of 4 to get offset into table**
  - **Fetch target from effective Address** `.L57 + op*4`

➢ X86-64 Switch Implementation

- **Same general idea, adapted to 64-bit code**
- **Table entries 64 bits (pointers)**
- **Cases use revised code**

```
switch(x) {
case 1:      // .L3
    w = y*z;
    break;
  . . .
}
```

```
.L3:
  movq    %rdx, %rax
  imulq   %rsi, %rax
  ret
```

Jump Table

```
  .section .rodata
  .align 8
.L7:
  .quad    .L2    # x = 0
  .quad    .L3    # x = 1
  .quad    .L4    # x = 2
  .quad    .L5    # x = 3
  .quad    .L2    # x = 4
  .quad    .L6    # X = 5
  .quad    .L6    # x = 6
```

➢ IA32 STACK

Stack Examples

IA32 Stack

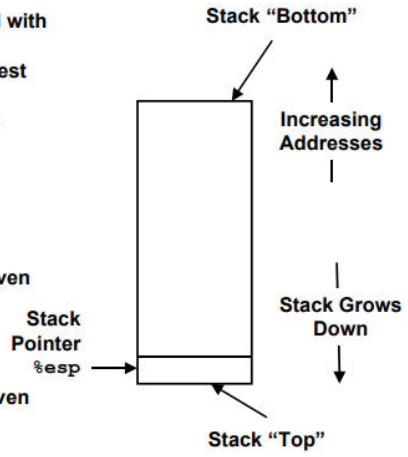
- Region of memory managed with stack discipline
- Register `%esp` indicates lowest allocated position in stack
  - i.e., address of top element

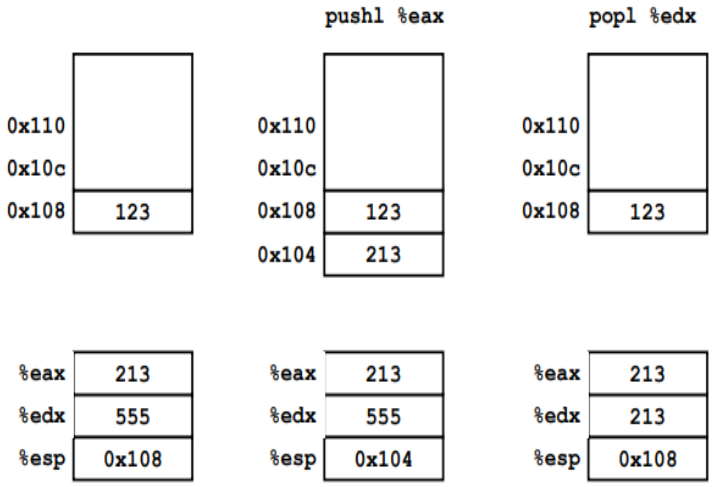
Pushing

- `pushl` *Src*
- Fetch operand at *Src*
- Decrement `%esp` by 4
- Write operand at address given by `%esp`

Popping

- `popl` *Dest*
- Read operand at address given by `%esp`
- Increment `%esp` by 4
- Write to *Dest*





➢ Procedure Call & Return

Procedure call:

`call label` Push return address on stack; Jump to *label*

Return address value

- Address of instruction beyond `call`
- Example from disassembly

```
804854e:   e8 3d 06 00 00   call   8048b90 <main>
8048553:   50               pushl  %eax
```

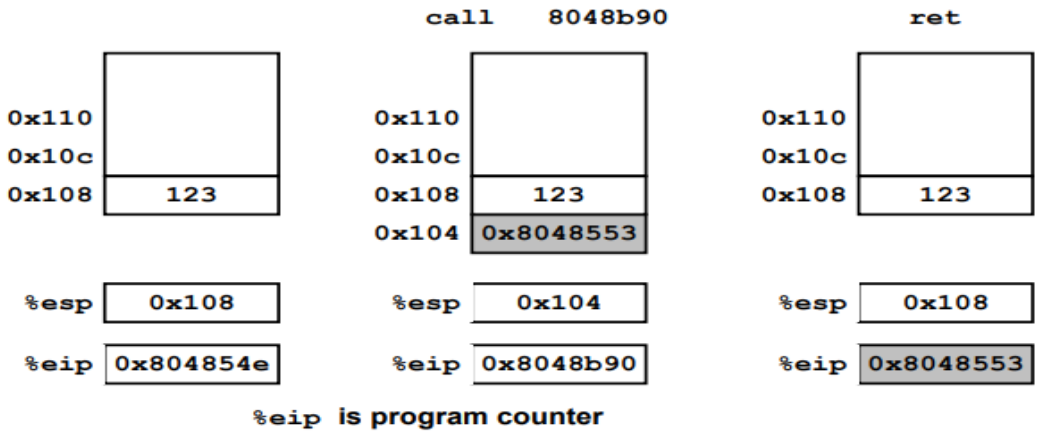
  - Return address = `0x8048553`

Procedure return:

- `ret` Pop address from stack; Jump to address

Example

```
804854e:   e8 3d 06 00 00   call   8048b90 <main>
8048553:   50               pushl  %eax
```



➢ Stack Structure

Stack Frame

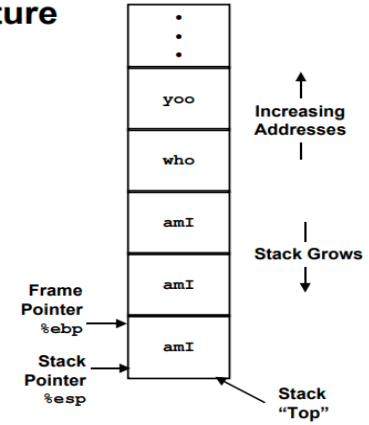
Register Usage

IA32 Stack Structure

Stack Growth
- Toward lower addresses

Stack Pointer
- Address of next available location in stack
- Use register `%esp`

Frame Pointer
- Start of current stack frame
- Use register `%ebp`

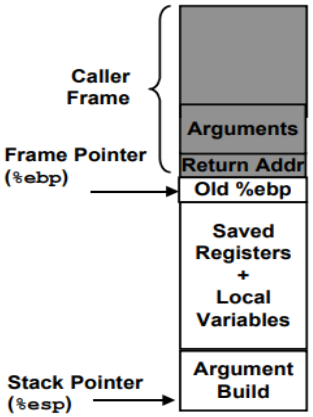


Callee Stack Frame ("Top" to Bottom)

- Parameters for called functions
- Local variables
  - If can't keep in registers
- Saved register context
- Old frame pointer

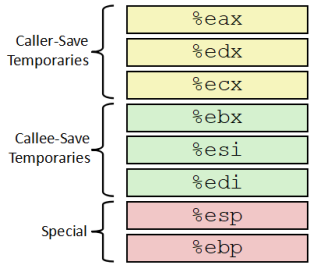
Caller Stack Frame

- Return address
  - Pushed by `call` instruction
- Arguments for this call



IA32/Linux+Windows Register Usage

- **`%eax, %edx, %ecx`**
  - Caller saves prior to call if values are used later
- **`%eax`**
  - also used to return integer value
- **`%ebx, %esi, %edi`**
  - Callee saves if wants to use them
- **`%esp, %ebp`**
  - special form of callee save
  - Restored to original values upon exit from procedure



➢ Recursive Function

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
    movl  $0, %eax
    testl %eax, %ebx
    je  .L3
    movl  %ebx, %eax
    shrl  %eax
    movl  %eax, (%esp)
    call  pcount_r
    movl  %ebx, %edx
    andl  $1, %edx
    leal  (%edx,%eax), %eax
.L3:
    addl  $4, %esp
    popl  %ebx
    popl %ebp
    ret
```

- **Registers**
  - `%eax`, `%edx` used without first saving
  - `%ebx` used, but saved at beginning & restored at end

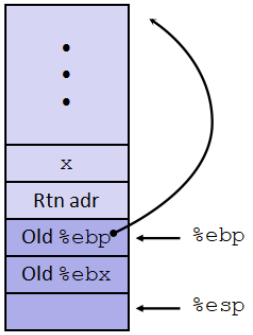
Recursive Call

```
/* Recursive popcount */
int pcount_r(unsigned x) {
  if (x == 0)
    return 0;
  else return
    (x & 1) + pcount_r(x >> 1);
}
```

```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
    • • •
```

- **Actions**
  - Save old value of `%ebx` on stack
  - Allocate space for argument to recursive call
  - Store x in `%ebx`

`%ebx` | x

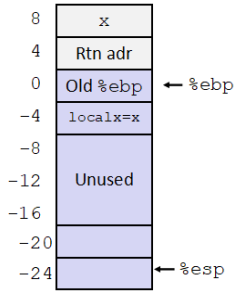


➢ Pointer Code

Creating and Initializing Local Variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Variable localx must be stored on stack
  - Because: Need to create pointer to it
- Compute pointer as -4(%ebp)



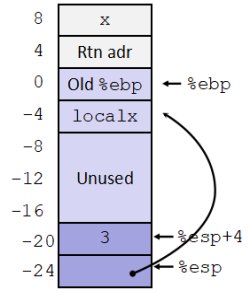
First part of `add3`

```
add3:
   pushl%ebp
   movl %esp, %ebp
   subl $24, %esp      # Alloc. 24 bytes
   movl 8(%ebp), %eax
   movl %eax, -4(%ebp)# Set localx to x
```

Creating Pointer as Argument

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Use leal instruction to compute address of localx



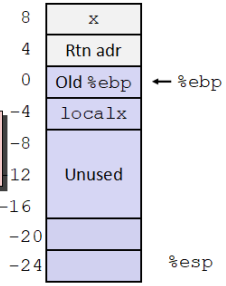
Middle part of `add3`

```
   movl $3, 4(%esp)    # 2nd arg = 3
   leal -4(%ebp), %eax# &localx
   movl %eax, (%esp)   # 1st arg = &localx
   call incrk
```

Retrieving local variable

```
int add3(int x) {
  int localx = x;
  incrk(&localx, 3);
  return localx;
}
```

- Retrieve localx from stack as return value



Final part of `add3`

```
   movl -4(%ebp), %eax # Return val= localx
   leave
   ret
```