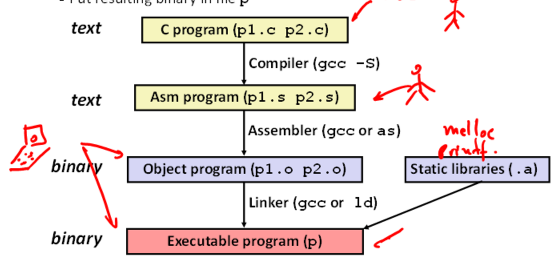
X86 Assembly, and C-to-assembly

- Move instructions, registers, and operands
- Complete addressing mode, address computation (`leal`)
- Arithmetic operations (including some x86-64 instructions)
- Condition codes
- Control, unconditional and conditional branches
- While loops

Turning C into Object Code

- Code in files `p1.c p2.c`
- Compile with command: `gcc -O p1.c p2.c -o p`
  - Use optimizations (`-O`)
  - Put resulting binary in file `p`



Compiling Into Assembly

C Code

```
int sum(int x, int y)
{
  int t = x+y;
  return t;
}
```

Generated IA32 Assembly

```
sum:
    pushl %ebp
    movl %esp,%ebp
    movl 12(%ebp),%eax
    addl 8(%ebp),%eax
    movl %ebp,%esp
    popl %ebp
    ret
```

Obtain with command

`gcc -O -S code.c`

Produces file `code.s`

Three Kinds of Instructions

- Perform arithmetic function on register or memory data
  - c = a + b;
- Transfer data between memory and register
  - Load data from memory into register
    - %reg = Mem[address]
  - Store register data into memory
    - Mem[address] = %reg
- Transfer control (control flow)
  - Unconditional jumps to/from procedures
  - Conditional branches

Assembly Characteristics: Data Types

- "Integer" data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- How about arrays, structs, etc?

Object Code

Code for `sum`

```
0x401040 <sum>:
    0x55
    0x89
    0xe5
    0x8b
    0x45
    0x0c
    0x03
    0x45
    0x08
    0x89
    0xec
    0x5d
    0xc3
```

- Total of 13 bytes
- Each instruction 1, 2, or 3 bytes
- Starts at address `0x401040`

- Assembler
  - Translates `.s` into `.o`
  - Binary encoding of each instruction
  - Nearly-complete image of executable code
  - Missing linkages between code in different files
- Linker
  - Resolves references between files
  - Combines with static run-time libraries
    - E.g., code for `malloc`, `printf`
  - Some libraries are *dynamically linked*
    - Linking occurs when program begins execution

Example

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

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

Similar to expression:

```
x += y
```

More precisely:

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

```
0x401046:    03 45 08
```

- **C Code**
  - Add two signed integers
- **Assembly**
  - Add 2 4-byte integers
    - "Long" words in GCC speak
    - 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 0x401046

Disassembling Object Code

Disassembled

```
00401040 <_sum>:
   0:    55              push   %ebp
   1:    89 e5           mov    %esp,%ebp
   3:    8b 45 0c        mov    0xc(%ebp),%eax
   6:    03 45 08        add    0x8(%ebp),%eax
   9:    89 ec           mov    %ebp,%esp
   b:    5d              pop    %ebp
   c:    c3              ret
```

- **Disassembler**
  `objdump -d p`
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces approximate rendition of assembly code
  - Can be run on either a.out (complete executable) or .o file

Integer Registers (IA32)

general purpose
%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

Integer Registers (IA32)

	32-bit	16-bit	high byte	low byte	Origin (mostly obsolete)
general purpose	%eax	%ax	%ah	%al	accumulate
general purpose	%ecx	%cx	%ch	%cl	counter
general purpose	%edx	%dx	%dh	%dl	data
general purpose	%ebx	%bx	%bh	%bl	base
general purpose	%esi	%si			source index
general purpose	%edi	%di			destination index
	%esp	%sp			stack pointer
	%ebp	%bp			base pointer

16-bit virtual registers (backwards compatibility)

Moving Data: IA32

- **Moving Data**
  - movx *Source*, *Dest*
  - x is one of {b, w, l}
  - movl *Source*, *Dest*:
    Move 4-byte "long word"
  - movw *Source*, *Dest*:
    Move 2-byte "word"
  - movb *Source*, *Dest*:
    Move 1-byte "byte"
- **Lots of these in typical code**

%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

Moving Data: IA32

- **Moving Data**
  movl *Source*, *Dest*:
- **Operand Types**
  - *Immediate:* Constant integer data
    - Example: $0x400, $-533
    - Like C constant, but prefixed with '$'
    - Encoded with 1, 2, or 4 bytes
  - *Register:* One of 8 integer registers
    - Example: %eax, %edx
    - But %esp and %ebp reserved for special use
    - Others have special uses for particular instructions
  - *Memory:* 4 consecutive bytes of memory at address given by register
    - Simplest example: (%eax)
    - Various other "address modes"

%eax
%ecx
%edx
%ebx
%esi
%edi
%esp
%ebp

movl Operand Combinations

	Source	Dest	Src,Dest	C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	
		Mem	`movl $-147,(%eax)`	
	Reg	*Reg*	`movl %eax,%edx`	
		Mem	`movl %eax,(%edx)`	
	Mem	*Reg*	`movl (%eax),%edx`	

Cannot do memory-memory transfer with a single instruction.

How do you copy from a memory location to another then?

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;`

Simple 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

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

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

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

   movl -4(%ebp),%ebx
   movl %ebp,%esp         Finish
   popl %ebp
   ret
```


Understanding Swap

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

Stack (in memory)

Offset	
	• • •
12	yp
8	xp
4	Rtn adr
0	Old %ebp ← %ebp
-4	Old %ebx

Register	Value
%ecx	yp
%edx	xp
%eax	t1
%ebx	t0

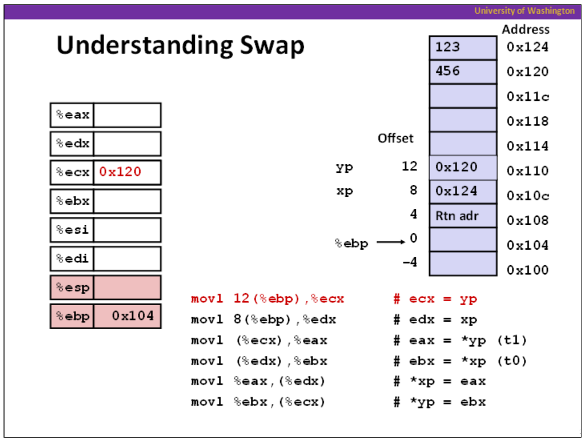
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

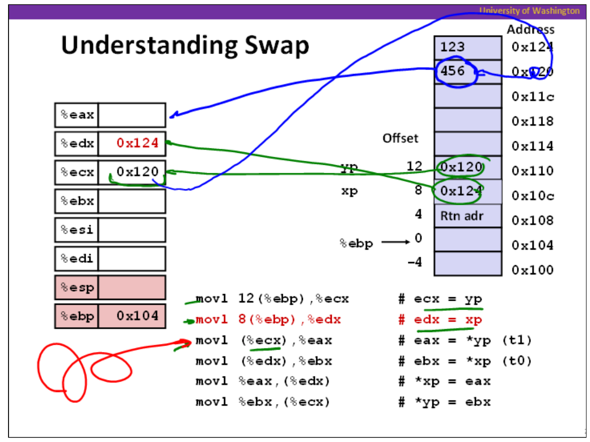

Understanding Swap

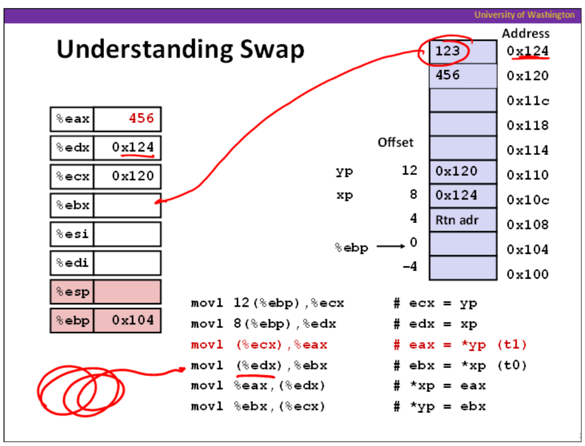
Register	Value
%eax	
%edx	
%ecx	
%ebx	
%esi	
%edi	
%esp	
%ebp	0x104

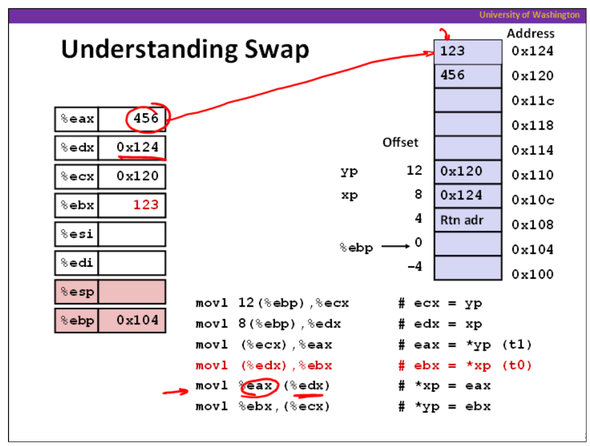
	Offset	Value	Address
		123	0x124
		456	0x120
			0x11c
			0x118
			0x114
yp	12	0x120	0x110
xp	8	0x124	0x10c
	4	Rtn adr	0x108
%ebp →	0		0x104
	-4		0x100

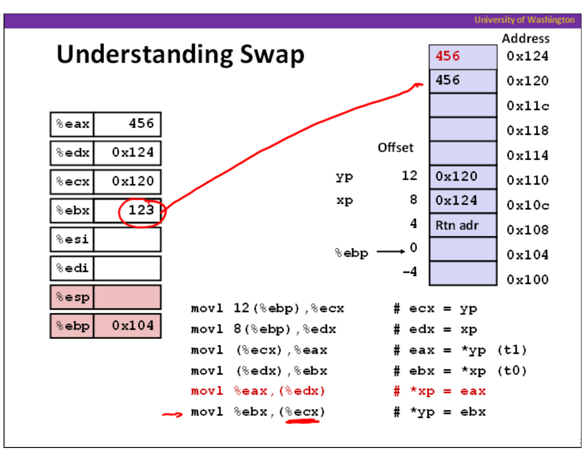
```
movl 12(%ebp),%ecx   # ecx = yp
movl 8(%ebp),%edx    # edx = xp
movl (%ecx),%eax     # eax = *yp (t1)
movl (%edx),%ebx     # ebx = *xp (t0)
movl %eax,(%edx)     # *xp = eax
movl %ebx,(%ecx)     # *yp = ebx
```

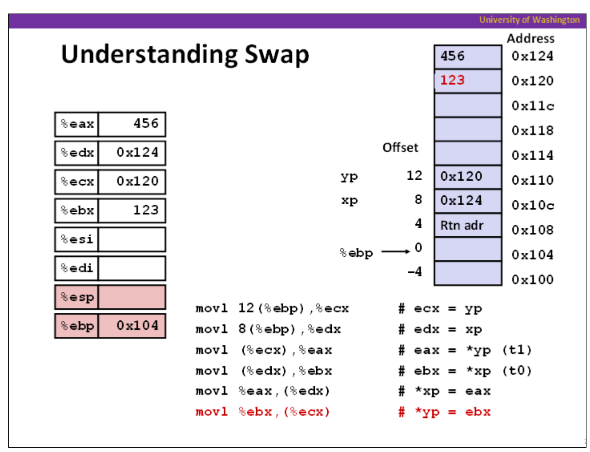

University of Washington
Understanding Swap
%eax
%edx
%ecx 0x120
%ebx
%esi
%edi
%esp
%ebp 0x104
Address
123 0x124
456 0x120
0x11c
0x118
Offset
0x114
yp 12 0x120 0x110
xp 8 0x124 0x10c
4 Rtn adr 0x108
%ebp 0 0x104
-4 0x100
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx


University of Washington
Understanding Swap
%eax
%edx 0x124
%ecx 0x120
%ebx
%esi
%edi
%esp
%ebp 0x104
Address
123 0x124
456 0x120
0x11c
0x118
Offset
0x114
yp 12 0x120 0x110
xp 8 0x124 0x10c
4 Rtn adr 0x108
%ebp 0 0x104
-4 0x100
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx


University of Washington
Understanding Swap
%eax 456
%edx 0x124
%ecx 0x120
%ebx
%esi
%edi
%esp
%ebp 0x104
Address
123 0x124
456 0x120
0x11c
0x118
Offset
0x114
yp 12 0x120 0x110
xp 8 0x124 0x10c
4 Rtn adr 0x108
%ebp 0 0x104
-4 0x100
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx


University of Washington
Understanding Swap
%eax 456
%edx 0x124
%ecx 0x120
%ebx 123
%esi
%edi
%esp
%ebp 0x104
Address
123 0x124
456 0x120
0x11c
0x118
Offset
0x114
yp 12 0x120 0x110
xp 8 0x124 0x10c
4 Rtn adr 0x108
%ebp 0 0x104
-4 0x100
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx


University of Washington
Understanding Swap
%eax 456
%edx 0x124
%ecx 0x120
%ebx 123
%esi
%edi
%esp
%ebp 0x104
Address
456 0x124
456 0x120
0x11c
0x118
Offset
0x114
yp 12 0x120 0x110
xp 8 0x124 0x10c
4 Rtn adr 0x108
%ebp 0 0x104
-4 0x100
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx


University of Washington
Understanding Swap
%eax 456
%edx 0x124
%ecx 0x120
%ebx 123
%esi
%edi
%esp
%ebp 0x104
Address
456 0x124
123 0x120
0x11c
0x118
Offset
0x114
yp 12 0x120 0x110
xp 8 0x124 0x10c
4 Rtn adr 0x108
%ebp 0 0x104
-4 0x100
movl 12(%ebp),%ecx # ecx = yp
movl 8(%ebp),%edx # edx = xp
movl (%ecx),%eax # eax = *yp (t1)
movl (%edx),%ebx # ebx = *xp (t0)
movl %eax,(%edx) # *xp = eax
movl %ebx,(%ecx) # *yp = ebx

Complete Memory Addressing Modes

- Most General Form
  - D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]
  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for `%esp`
    - Unlikely you'd use `%ebp`, either
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)
- Special Cases
  - (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]
  - D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]
  - (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

University of Washington

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

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

Expression	Address Computation	Address
`0x8(%edx)`		
`(%edx,%ecx)`		
`(%edx,%ecx,4)`		
`0x80(%edx,2)`		

University of Washington

Address Computation Examples

`%edx`	`0xf000`
`%ecx`	`0x100`

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

University of Washington

Address Computation Instruction

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

University of Washington

Some Arithmetic Operations

- Two Operand Instructions:

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*	

University of Washington

Some Arithmetic Operations

- Two Operand Instructions:

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*	

- No distinction between signed and unsigned int (why?)

Some Arithmetic Operations

- One Operand Instructions

```
incl  Dest        Dest = Dest + 1
decl  Dest        Dest = Dest - 1
negl  Dest        Dest = -Dest
notl  Dest        Dest = ~Dest
```

- See book for more instructions

Using `leal` for Arithmetic Expressions

```
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                    } Set
    movl  %esp,%ebp               } Up

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

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


Understanding `arith`

```
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;
}
```

Stack

Offset	
	• • •
16	z
12	y
8	x
4	Rtn adr
0	Old %ebp ← %ebp

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

What does each of these instructions mean?

Understanding `arith`

```
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;
}
```

Stack

Offset	
	• • •
16	z
12	y
8	x
4	Rtn adr
0	Old %ebp ← %ebp

```
movl  8(%ebp),%eax            # eax = x
movl  12(%ebp),%edx           # edx = y
leal  (%edx,%eax),%ecx        # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx      # edx = 3*y
sall  $4,%edx                 # edx = 48*y  (t4)
addl  16(%ebp),%ecx           # ecx = z+t1  (t2)
leal  4(%edx,%eax),%eax       # eax = 4+t4+x  (t5)
imull %ecx,%eax               # eax = t5*t2  (rval)
```


Understanding `arith`

```
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;
}
```

Stack

Offset	
	• • •
16	z
12	y
8	x
4	Rtn adr
0	Old %ebp ← %ebp

```
movl  8(%ebp),%eax            # eax = x
movl  12(%ebp),%edx           # edx = y
leal  (%edx,%eax),%ecx        # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx      # edx = 3*y
sall  $4,%edx                 # edx = 48*y  (t4)
addl  16(%ebp),%ecx           # ecx = z+t1  (t2)
leal  4(%edx,%eax),%eax       # eax = 4+t4+x  (t5)
imull %ecx,%eax               # eax = t5*t2  (rval)
```


Understanding `arith`

```
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;
}
```

Stack

Offset	
	• • •
16	z
12	y
8	x
4	Rtn adr
0	Old %ebp ← %ebp

```
movl  8(%ebp),%eax            # eax = x
movl  12(%ebp),%edx           # edx = y
leal  (%edx,%eax),%ecx        # ecx = x+y   (t1)
leal  (%edx,%edx,2),%edx      # edx = 3*y
sall  $4,%edx                 # edx = 48*y  (t4)
addl  16(%ebp),%ecx           # ecx = z+t1  (t2)
leal  4(%edx,%eax),%eax       # eax = 4+t4+x  (t5)
imull %ecx,%eax               # eax = t5*t2  (rval)
```

Understanding `arith`

```
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;
}
```

Offset	Stack	
	• • •	
16	z	
12	y	
8	x	
4	Rtn adr	
0	Old %ebp	← %ebp

```
movl 8(%ebp),%eax        # eax = x
movl 12(%ebp),%edx       # edx = y
leal (%edx,%eax),%ecx    # ecx = x+y  (t1)
leal (%edx,%edx,2),%edx  # edx = 3*y
sall $4,%edx             # edx = 48*y (t4)
addl 16(%ebp),%ecx       # ecx = z+t1 (t2)
leal 4(%edx,%eax),%eax   # eax = 4+t4+x (t5)
imull %ecx,%eax          # eax = t5*t2 (rval)
```

University of Washington

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp             } Set
    movl %esp,%ebp         } Up

    movl 8(%ebp),%eax      }
    xorl 12(%ebp),%eax     }
    sarl $17,%eax          }
    andl $8185,%eax        } Body

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

```
movl 8(%ebp),%eax    # eax = x
xorl 12(%ebp),%eax   # eax = x^y
sarl $17,%eax        # eax = t1>>17
andl $8185,%eax      # eax = t2 & 8185
```

University of Washington

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp             } Set
    movl %esp,%ebp         } Up

    movl 8(%ebp),%eax      }
    xorl 12(%ebp),%eax     }
    sarl $17,%eax          }
    andl $8185,%eax        } Body

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

```
movl 8(%ebp),%eax    eax = x
xorl 12(%ebp),%eax   eax = x^y      (t1)
sarl $17,%eax        eax = t1>>17   (t2)
andl $8185,%eax      eax = t2 & 8185
```

University of Washington

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp             } Set
    movl %esp,%ebp         } Up

    movl 8(%ebp),%eax      }
    xorl 12(%ebp),%eax     }
    sarl $17,%eax          }
    andl $8185,%eax        } Body

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

```
movl 8(%ebp),%eax    eax = x
xorl 12(%ebp),%eax   eax = x^y      (t1)
sarl $17,%eax        eax = t1>>17   (t2)
andl $8185,%eax      eax = t2 & 8185
```

University of Washington

Another Example

```
int logical(int x, int y)
{
  int t1 = x^y;
  int t2 = t1 >> 17;
  int mask = (1<<13) - 7;
  int rval = t2 & mask;
  return rval;
}
```

```
logical:
    pushl %ebp             } Set
    movl %esp,%ebp         } Up

    movl 8(%ebp),%eax      }
    xorl 12(%ebp),%eax     }
    sarl $17,%eax          }
    andl $8185,%eax        } Body

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

$2^{13} = 8192$, $2^{13} - 7 = 8185$

```
movl 8(%ebp),%eax    eax = x
xorl 12(%ebp),%eax   eax = x^y      (t1)
sarl $17,%eax        eax = t1>>17   (t2)
andl $8185,%eax      eax = t2 & 8185
```

University of Washington

Control-Flow/Conditionals

- Unconditional

```
while(true) {
  do_something;
}
...
```

- Conditional

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

Conditionals and Control Flow

- A test / conditional branch is sufficient to implement most control flow constructs offered in higher level languages
  - if (condition) then {...} else {...}
  - while(condition) {...}
  - do {...} while (condition)
  - for (initialization; condition;) {...}
- (Unconditional branches implemented some related control flow constructs
  - break, continue)

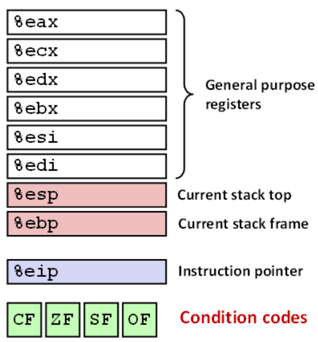
Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
`jmp`	1	Unconditional
`je`	ZF	Equal / Zero
`jne`	~ZF	Not Equal / Not Zero
`js`	SF	Negative
`jns`	~SF	Nonnegative
`jg`	~(SF^OF)&~ZF	Greater (Signed)
`jge`	~(SF^OF)	Greater or Equal (Signed)
`jl`	(SF^OF)	Less (Signed)
`jle`	(SF^OF)|ZF	Less or Equal (Signed)
`ja`	~CF&~ZF	Above (unsigned)
`jb`	CF	Below (unsigned)

Processor State (IA32, Partial)

- 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 (Implicit Setting)

- Single bit registers
  - `CF` Carry Flag (for unsigned) `SF` Sign Flag (for signed)
  - `ZF` Zero Flag `OF` Overflow Flag (for signed)
- Implicitly set (think of it as side effect) by arithmetic operations
  - Example: `addl/addq` *Src,Dest* ↔ `t = a+b`
  - CF set if carry out from most significant bit (unsigned overflow)
  - ZF set if `t == 0`
  - SF set if `t < 0` (as signed)
  - OF set if two's complement (signed) overflow
    `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`
- *Not* set by `lea` instruction (beware!)
- Full documentation (IA32)

Condition Codes (Explicit Setting: Compare)

- Explicit Setting by Compare Instruction
  - `cmpl/cmpq` *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` (as signed)
  - OF set if two's complement (signed) overflow
    `(a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)`

Condition Codes (Explicit Setting: Test)

- Explicit Setting by Test instruction
  - `testl/testq` *Src2,Src1*
  - `testl b,a` like computing `a&b` without setting destination
  - Sets condition codes based on value of *Src1* & *Src2*
  - Useful to have one of the operands be a mask
  - ZF set when `a&b == 0`
  - SF set when `a&b < 0`
  - `testl %eax, %eax`
    - Sets SF and ZF, check if eax is +,0,-

Reading Condition Codes

- SetX Instructions
  - Set a single byte based on combinations of condition codes

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

Reading Condition Codes (Cont.)

- SetX Instructions:
  Set single byte based on combination of condition codes
- One of 8 addressable byte registers
  - Does not alter remaining 3 bytes
  - Typically use **movzbl** to finish job

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

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

Body

```
movl 12(%ebp),%eax
cmpl %eax,8(%ebp)
setg %al
movzbl %al,%eax
```

What does each of these instructions do?

University of Washington

Reading Condition Codes (Cont.)

- SetX Instructions:
  Set single byte based on combination of condition codes
- One of 8 addressable byte registers
  - Does not alter remaining 3 bytes
  - Typically use **movzbl** to finish job

%eax	%ah	%al
%ecx	%ch	%cl
%edx	%dh	%dl
%ebx	%bh	%bl
%esi		
%edi		
%esp		
%ebp		

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

Body

```
movl 12(%ebp),%eax     # eax = y
cmpl %eax,8(%ebp)      # Compare x and y
setg %al               # al = x > y
movzbl %al,%eax        # Zero rest of %eax
```

Note inverted ordering!

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

jX	Condition	Description
jmp	1	Unconditional
je	ZF	Equal / Zero
jne	~ZF	Not Equal / Not Zero
js	SF	Negative
jns	~SF	Nonnegative
jg	~(SF^OF)&~ZF	Greater (Signed)
jge	~(SF^OF)	Greater or Equal (Signed)
jl	(SF^OF)	Less (Signed)
jle	(SF^OF)|ZF	Less or Equal (Signed)
ja	~CF&~ZF	Above (unsigned)
jb	CF	Below (unsigned)

Conditional Branch Example

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

```
absdiff:
    pushl   %ebp              } Setup
    movl    %esp, %ebp
    movl    8(%ebp), %edx     } Body1
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave                     } Finish
    ret
.L7:
    subl    %edx, %eax        } Body2
    jmp     .L8
```


Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

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

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```

Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```


Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```


Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```


Conditional Branch Example (Cont.)

```
int goto_ad(int x, int y)
{
  int result;
  if (x <= y) goto Else;
  result = x-y;
Exit:
  return result;
Else:
  result = y-x;
  goto Exit;
}
```

```
absdiff:
    pushl   %ebp
    movl    %esp, %ebp
    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L7
    subl    %eax, %edx
    movl    %edx, %eax
.L8:
    leave
    ret
.L7:
    subl    %edx, %eax
    jmp     .L8
```


General Conditional Expression Translation

C Code

```
val = Test ? Then-Expr : Else-Expr;
```

```
val = x>y ? x-y : y-x;
```

- *Test* is expression returning integer
  - = 0 interpreted as false
  - ≠0 interpreted as true
- Create separate code regions for then & else expressions
- Execute appropriate one
- How would you make this efficient?

Goto Version

```
  nt = !Test;
  if (nt) goto Else;
  val = Then-Expr;
Done:
  . . .
Else:
  val = Else-Expr;
  goto Done;
```


Conditionals: x86-64

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

```
absdiff: # x in %edi, y in %esi
  movl   %edi, %eax   # eax = x
  movl   %esi, %edx   # edx = y
  subl   %esi, %eax   # eax = x-y
  subl   %edi, %edx   # edx = y-x
  cmpl   %esi, %edi   # x:y
  cmovle %edx, %eax   # eax=edx if <=
  ret
```

- **Conditional move instruction**
  - `cmovC` src, dest
  - Move value from src to dest if condition *C* holds
  - More efficient than conditional branching (simple control flow)
  - But overhead: both branches are evaluated

PC Relative Addressing

```
0x100     cmp   r2, r3      0x1000
0x102     je    0x70        0x1002
0x104     …                 0x1004
…         …                 …
0x172     add   r3, r4      0x1072
```

- PC relative branches are relocatable
- Absolute branches are not

Compiling Loops

C/Java code

```
while ( sum != 0 ) {
  <loop body>
}
```

Machine code

```
loopTop:   cmp   r3, $0
           be    loopDone
               <loop body code>
           jmp  loopTop
loopDone:
```

- How to compile other loops should be clear to you
  - The only slightly tricky part is to be sure where the conditional branch occurs: top or bottom of the loop
- Q: How is `for (i=0; i<100; i++)` implemented?
- Q: How are `break` and `continue` implemented?

Machine Programming II: Instructions (cont'd)

- Move instructions, registers, and operands
- Complete addressing mode, address computation (`leal`)
- Arithmetic operations (including some x86-64 instructions)
- Condition codes
- Control, unconditional and conditional branches
- While loops
- For loops
- Switch statements

"Do-While" Loop Example

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

"Do-While" Loop Compilation

Registers:

%edx	x
%eax	result

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;

loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

Assembly

```
fact_goto:
  pushl %ebp
  movl %esp,%ebp
  movl $1,%eax
  movl 8(%ebp),%edx

.L11:
  imull %edx,%eax
  decl %edx
  cmpl $1,%edx
  jg .L11

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

Translation?

"Do-While" Loop Compilation

Registers:

%edx	x
%eax	result

Goto Version

```
int
fact_goto(int x)
{
  int result = 1;

loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;

  return result;
}
```

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
```

General "Do-While" Translation

C Code

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

Goto Version

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

- *Body:* {
  Statement$_1$;
  Statement$_2$;
  …
  Statement$_n$;
  }
- *Test* returns integer
  = 0 interpreted as false
  ≠0 interpreted as true

"While" Loop Example

C Code

```
int fact_while(int x)
{
  int result = 1;
  while (x > 1) {


    result *= x;
    x = x-1;
  };

  return result;
}
```

Goto Version #1

```
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;
}
```

- Is this code equivalent to the do-while version?
- Must jump out of loop if test fails

Alternative "While" Loop Translation

C Code

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

Goto Version #2

```
int fact_while_goto2(int x)
{
  int result = 1;
  if (!(x > 1))
    goto done;
loop:
  result *= x;
  x = x-1;
  if (x > 1)
    goto loop;
done:
  return result;
}
```

- Historically used by GCC
- Uses same inner loop as do-while version
- Guards loop entry with extra test

General "While" Translation

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:
```

New Style "While" Loop Translation

C Code

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

Goto Version

```
int fact_while_goto3(int x)
{
  int result = 1;
  goto middle;
loop:
  result *= x;
  x = x-1;
middle:
  if (x > 1)
    goto loop;
  return result;
}
```

- Recent technique for GCC
  - Both IA32 & x86-64
- First iteration jumps over body computation within loop

Jump-to-Middle While Translation

C Code

```
while (Test)
  Body
```

- Avoids duplicating test code
- Unconditional `goto` incurs no performance penalty
- `for` loops compiled in similar fashion

Goto Version

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

Goto (Previous) Version

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

Jump-to-Middle Example

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

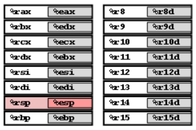
```
# x in %edx, result in %eax
  jmp    .L34          #   goto Middle
.L35:                  # Loop:
  imull %edx, %eax #   result *= x
  decl   %edx          #   x--
.L34:                  # Middle:
  cmpl   $1, %edx      #   x:1
  jg     .L35          #   if >, goto Loop
```


Quick Review

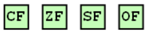
- Complete memory addressing mode
  - `(%eax), 17(%eax), 2(%ebx, %ecx, 8),` …
- Arithmetic operations that do set condition codes
  - `subl %eax, %ecx          # ecx = ecx + eax`
  - `sall $4,%edx             # edx = edx << 4`
  - `addl 16(%ebp),%ecx       # ecx = ecx + Mem[16+ebp]`
  - `imull %ecx,%eax          # eax = eax * ecx`
- Arithmetic operations that do NOT set condition codes
  - `leal 4(%edx,%eax),%eax   # eax = 4 + edx + eax`

Quick Review

- x86-64 vs. IA32
  - Integer registers: **16 x 64-bit** vs. **8 x 32-bit**
  - `movq, addq,` … vs. `movl, addl,` …
  - Better support for passing function arguments in registers



- Control
  - Condition code registers
  - Set as side effect or by `cmp, test`
  - Used:
    - Read out by setx instructions (`setg, setle`, …)
    - Or by conditional jumps (`jle .L4, je .L10,` …)



Quick Review

- Do-While loop

C Code

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

Goto Version

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

- While-Do loop

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:
```

or

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


"For" Loop Example: Square-and-Multiply

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
   int result;
     for (result = 1; p != 0; p = p>>1) {
       if (p & 0x1)
         result *= x;
       x = x*x;
     }
   return result;
}
```

- Algorithm
  - Exploit bit representation: $p = p_0 + 2p_1 + 2^2 p_2 + \ldots 2^{n-1} p_{n-1}$
  - Gives: $x^p = z_0 \cdot z_1{}^2 \cdot (z_2{}^2)^2 \cdot \ldots \cdot \underbrace{(\ldots((z_{n-1}{}^2)^2)\ldots)^2}_{n-1 \text{ times}}$

    $z_i = 1$ when $p_i = 0$

    $z_i = x$ when $p_i = 1$
  - Complexity $O(\log p)$

Example

$3^{10} = 3^2 * 3^8$

$= 3^2 * ((3^2)^2)^2$

`ipwr` Computation

```
/* Compute x raised to nonnegative power p */
int ipwr_for(int x, unsigned p)
{
   int result;
     for (result = 1; p != 0; p = p>>1) {
       if (p & 0x1)
         result *= x;
       x = x*x;
     }
   return result;
}
```

before iteration	result	x=3	p=10
1	1	3	$10=1010_2$
2	1	9	$5= 101_2$
3	9	81	$2= 10_2$
4	9	6561	$1= 1_2$
5	59049	43046721	0_2

"For" Loop Example

```
int result;
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

General Form

```
for (Init; Test; Update)
  Body
```

Test

```
p != 0
```

Init

```
result = 1
```

Update

```
p = p >> 1
```

Body

```
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```


"For"→ "While"→ "Do-While"

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

Do-While Version

```
Init;
if (!Test)
    goto done;
do {
    Body
    Update ;
} while (Test)
done:
```

Goto Version

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


For-Loop: Compilation #1

For Version

```
for (Init; Test; Update )
    Body
```

Goto Version

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

```
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

```
   result = 1;
   if (p == 0)
     goto done;
loop:
   if (p & 0x1)
     result *= x;
   x = x*x;
   p = p >> 1;
   if (p != 0)
     goto loop;
done:
```


"For"→ "While" (Jump-to-Middle)

For Version

```
for (Init; Test; Update )
    Body
```

While Version

```
Init;
while (Test ) {
    Body
    Update ;
}
```

Goto Version

```
    Init;
    goto middle;
loop:
    Body
    Update ;
middle:
    if (Test)
        goto loop;
done:
```


For-Loop: Compilation #2

For Version

```
for (Init; Test; Update )
    Body
```

Goto Version

```
    Init;
    goto middle;
loop:
    Body
    Update ;
middle:
    if (Test)
        goto loop;
done:
```

```
for (result = 1; p != 0; p = p>>1)
{
  if (p & 0x1)
    result *= x;
  x = x*x;
}
```

```
   result = 1;
goto middle;
loop:
   if (p & 0x1)
     result *= x;
   x = x*x;
   p = p >> 1;
middle:
   if (p != 0)
     goto loop;
done:
```



```
long switch_eg
    (long x, long y, long z)
{
    long w = 1;
    switch(x) {
    case 1:
        w = y*z;
        break;
    case 2:
        w = y/z;
        /* Fall Through */
    case 3:
        w += z;
        break;
    case 5:
    case 6:
        w -= z;
        break;
    default:
        w = 2;
    }
    return w;
}
```

Switch Statement Example

- Multiple case labels
  - Here: 5, 6
- Fall through cases
  - Here: 2
- Missing cases
  - Here: 4

Jump Table Structure

Switch Form

```
switch(x) {
  case val_0:
    Block 0
  case val_1:
    Block 1
    • • •
  case val_n-1:
    Block n–1
}
```

Jump Table

```
jtab: Targ0
      Targ1
      Targ2
        •
        •
        •
      Targn-1
```

Jump Targets

```
Targ0:   Code Block 0
Targ1:   Code Block 1
Targ2:   Code Block 2
  •
  •
  •
Targn-1: Code Block n–1
```

Approximate Translation

```
target = JTab[x];
goto *target;
```


Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Setup:

```
switch_eg:
    pushl %ebp              # Setup
    movl  %esp, %ebp        # Setup
    pushl %ebx              # Setup
    movl  $1, %ebx
    movl  8(%ebp), %edx
    movl  16(%ebp), %ecx
    cmpl  $6, %edx
    ja    .L61
    jmp   *.L62(,%edx,4)
```

Translation?

Switch Statement Example (IA32)

```
long switch_eg(long x, long y, long z)
{
    long w = 1;
    switch(x) {
      . . .
    }
    return w;
}
```

Jump table

```
.section .rodata
  .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

Setup:

```
switch_eg:
    pushl %ebp              # Setup
    movl  %esp, %ebp        # Setup
    pushl %ebx              # Setup
    movl  $1, %ebx          # w = 1
    movl  8(%ebp), %edx     # edx = x
    movl  16(%ebp), %ecx    # ecx = z
    cmpl  $6, %edx          # x:6
    ja    .L61              # if > goto default
    jmp   *.L62(,%edx,4)    # goto JTab[x]
```

Indirect jump → `jmp`

Assembly Setup Explanation

- **Table Structure**
  - Each target requires 4 bytes
  - Base address at `.L62`
- **Jumping**

  Direct: `jmp .L61`
  - Jump target is denoted by label `.L61`

  Indirect: `jmp *.L62(,%edx,4)`
  - Start of jump table: `.L62`
  - Must scale by factor of 4 (labels have 32-bit = 4 Bytes on IA32)
  - Fetch target from effective Address `.L62 + edx*4`
    - Only for $0 \le x \le 6$

Jump table

```
.section .rodata
  .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```


Jump Table

Jump table

```
.section .rodata
  .align 4
.L62:
  .long   .L61  # x = 0
  .long   .L56  # x = 1
  .long   .L57  # x = 2
  .long   .L58  # x = 3
  .long   .L61  # x = 4
  .long   .L60  # x = 5
  .long   .L60  # x = 6
```

```
switch(x) {
case 1:       // .L56
    w = y*z;
    break;
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
case 5:
case 6:       // .L60
    w -= z;
    break;
default:      // .L61
    w = 2;
}
```


Code Blocks (Partial)

```
switch(x) {
  . . .
case 2:       // .L57
    w = y/z;
    /* Fall Through */
case 3:       // .L58
    w += z;
    break;
  . . .
default:      // .L61
    w = 2;
}
```

```
.L61:  // Default case
   movl  $2, %ebx      # w = 2
   movl  %ebx, %eax    # Return w
   popl  %ebx
   leave
   ret
.L57:  // Case 2:
   movl  12(%ebp), %eax  # y
   cltd                  # Div prep
   idivl %ecx            # y/z
   movl  %eax, %ebx # w = y/z
# Fall through
.L58:  // Case 3:
   addl  %ecx, %ebx # w+= z
   movl  %ebx, %eax # Return w
   popl  %ebx
   leave
   ret
```

Code Blocks (Rest)

```
switch(x) {
case 1:        // .L56
    w = y*z;
    break;
  . . .
case 5:
case 6:        // .L60
    w -= z;
    break;
  . . .
}
```

```
.L60: // Cases 5&6:
   subl  %ecx, %ebx  # w -= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
.L56: // Case 1:
   movl  12(%ebp), %ebx # w = y
   imull %ecx, %ebx     # w*= z
   movl  %ebx, %eax  # Return w
   popl  %ebx
   leave
   ret
```

IA32 Object Code

- Setup
  - Label `.L61` becomes address `0x08048630`
  - Label `.L62` becomes address `0x080488dc`

Assembly Code

```
switch_eg:
  . . .
  ja    .L61           # if > goto default
  jmp   *.L62(,%edx,4) # goto JTab[x]
```

Disassembled Object Code

```
08048610 <switch_eg>:
 . . .
08048622:  77 0c                 ja      8048630
08048624:  ff 24 95 dc 88 04 08  jmp     *0x80488dc(,%edx,4)
```

IA32 Object Code (cont.)

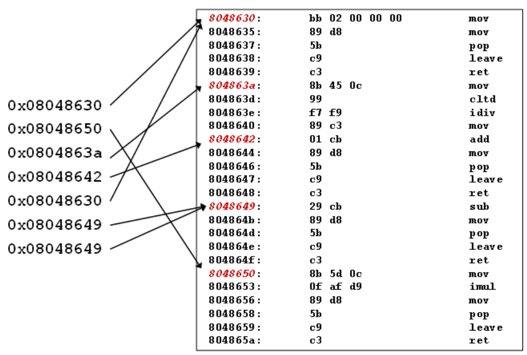
- Jump Table
  - Doesn't show up in disassembled code
  - Can inspect using GDB
    `gdb asm-cntl`
    `(gdb) x/7xw 0x080488dc`
    - Examine 7 hexadecimal format "words" (4-bytes each)
    - Use command "`help x`" to get format documentation

```
0x080488dc:
  0x08048630
  0x08048650
  0x0804863a
  0x08048642
  0x08048630
  0x08048649
  0x08048649
```

Disassembled Targets

```
8048630:   bb 02 00 00 00    mov    $0x2,%ebx
8048635:   89 d8             mov    %ebx,%eax
8048637:   5b                pop    %ebx
8048638:   c9                leave
8048639:   c3                ret
804863a:   8b 45 0c          mov    0xc(%ebp),%eax
804863d:   99                cltd
804863e:   f7 f9             idiv   %ecx
8048640:   89 c3             mov    %eax,%ebx
8048642:   01 cb             add    %ecx,%ebx
8048644:   89 d8             mov    %ebx,%eax
8048646:   5b                pop    %ebx
8048647:   c9                leave
8048648:   c3                ret
8048649:   29 cb             sub    %ecx,%ebx
804864b:   89 d8             mov    %ebx,%eax
804864d:   5b                pop    %ebx
804864e:   c9                leave
804864f:   c3                ret
8048650:   8b 5d 0c          mov    0xc(%ebp),%ebx
8048653:   0f af d9          imul   %ecx,%ebx
8048656:   89 d8             mov    %ebx,%eax
8048658:   5b                pop    %ebx
8048659:   c9                leave
804865a:   c3                ret
```

Matching Disassembled Targets



Summarizing

- C Control
  - if-then-else
  - do-while
  - while, for
  - switch
- Assembler Control
  - Conditional jump
  - Conditional move
  - Indirect jump
  - Compiler
  - Must generate assembly code to implement more complex control
- Standard Techniques
  - Loops converted to do-while form
  - Large switch statements use jump tables
  - Sparse switch statements may use decision trees (see text)
- Conditions in CISC
  - CISC machines generally have condition code registers