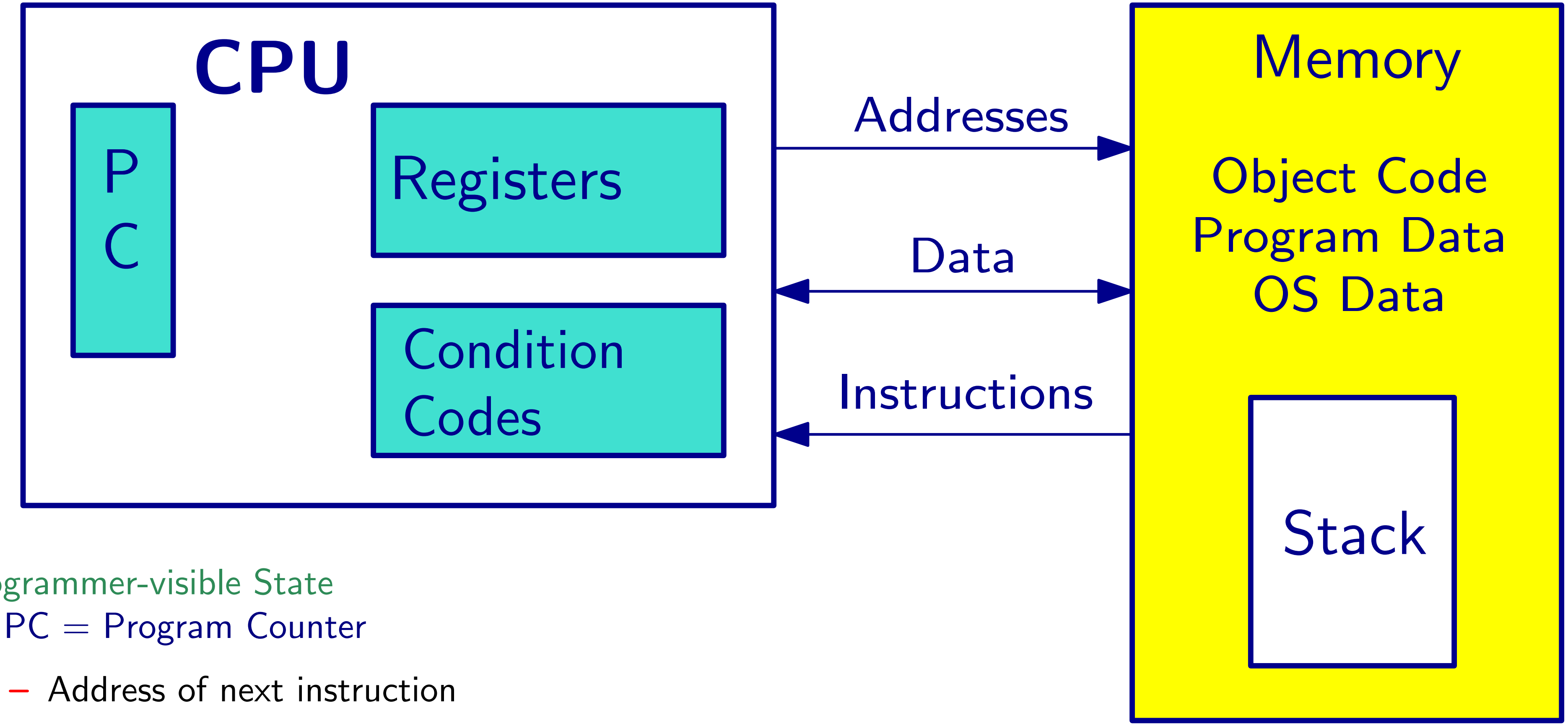
Overview of GCC and the IA-32 instruction set

Assembly Programmer's View

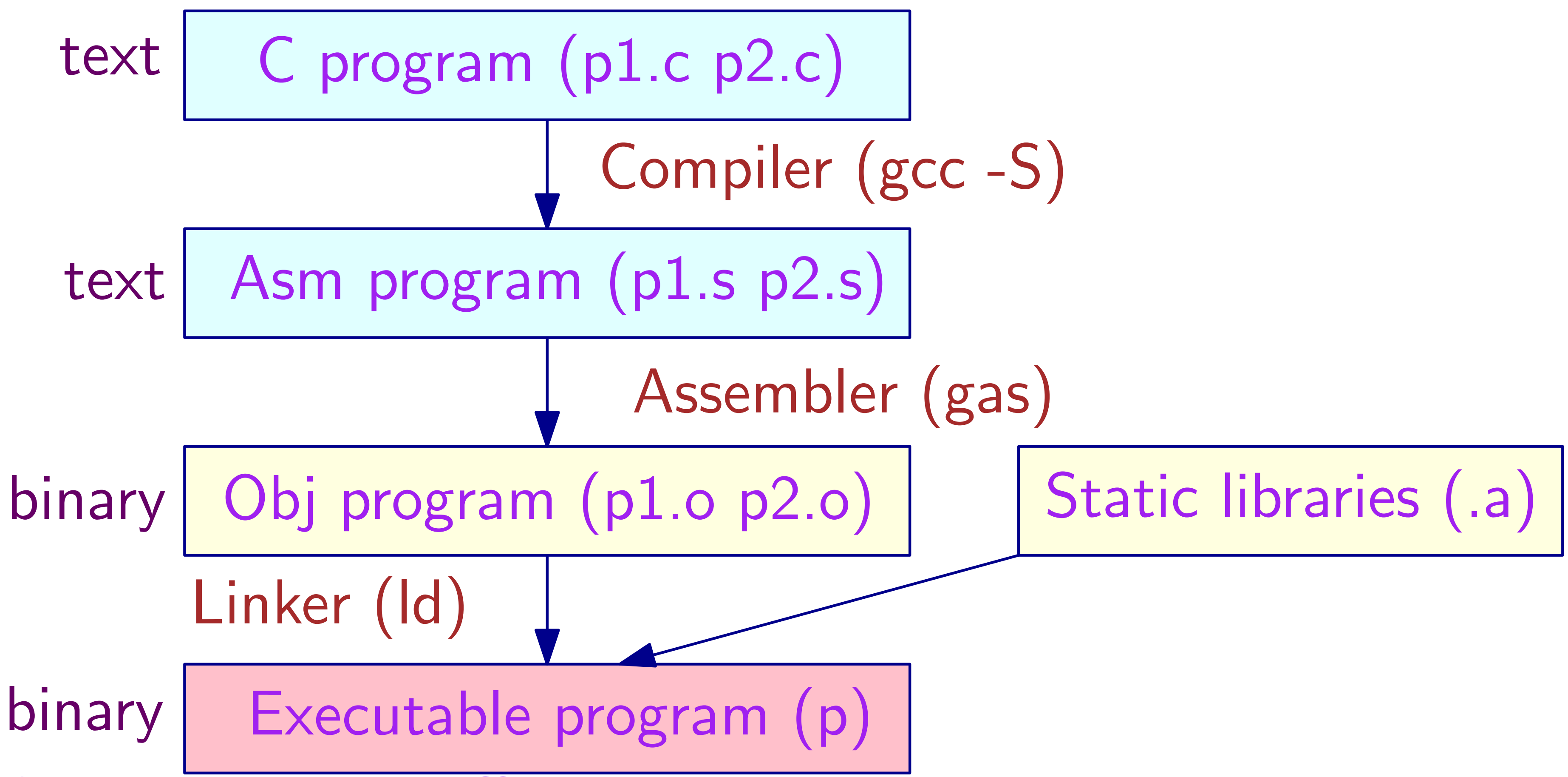


Programmer-visible State

- PC = Program Counter
  - Address of next instruction
  - Called `EIP` (IA32) or `RIP` (x86-64)
- Register File
  - Heavily used program data
- Condition Codes
  - Store status information about most recent arithmetic operation
  - Used for conditional branching
- Memory
  - Byte-addressable array
  - Code, user data, OS data
  - Includes stack used to support procedures

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 the resulting binary in `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 -fno-asynchronous-unwind-tables \
    -mpreferred-stack-boundary=2 -O -S code.c
```

Produces file `code.S`

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 -fno-asynchronous-unwind-tables \
    -mpreferred-stack-boundary=2 -O -S code.c
```

No C++ exceptions

Produces file `code.S`

Compiling into Assembly

[C]

C Code

```c
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 -fno-asynchronous-unwind-tables \
    -mpreferred-stack-boundary=2 -O -S code.c
```

Align on 4-byte boundary

Produces file `code.S`

Compiling into Assembly

[C]

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 -fno-asynchronous-unwind-tables \
    -mpreferred-stack-boundary=2 -O -S code.c
```

Produces file `code.S`

Easier to understand asm code

Assembly Characteristics

[B]

Minimal data types

- "Integer" data of 1, 2, or 4 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 10 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Primitive operations

- Perform arithmetic function on register or memory data
- Transfer data between memory and register
  - Load data from memory into register
  - Store register data into memory
- Transfer control
  - Unconditional jumps to/from procedures
  - Conditional branches

Object Code

Code for `sum`

```
0x40140 <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**

Machine Instruction Example


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

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

Similar to expression

```
x += y
```

Or

```
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 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 `0x401046`

Disassembling Object Code

[C]

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
  d:        8d 76 00        lea 0x0(%esi),%esi
```

Disassembler

`objdump -d p`

- Useful tool for examining data
- Analyzes bit pattern of series of instructions
- Produces approximate rendition of assembly code
- Can be run on either executables or object files

Alternate Disassembly

[C]

Object

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

Disassembled

```
0x401040 <sum>:      push %ebp
0x401041 <sum+1>:    mov %esp,%ebp
0x401043 <sum+3>:    mov 0xc(%ebp),%eax
0x401046 <sum+6>:    add 0x8(%ebp),%eax
0x401049 <sum+9>:    mov %ebp,%esp
0x40104b <sum+11>:   pop %ebp
0x40104c <sum+12>:   ret
0x40104d <sum+13>:   lea 0x0(%esi),%esi
```

Within gdb debugger

```
gdb p

disassemble sum

x/13b sum
```

Moving Data: IA32

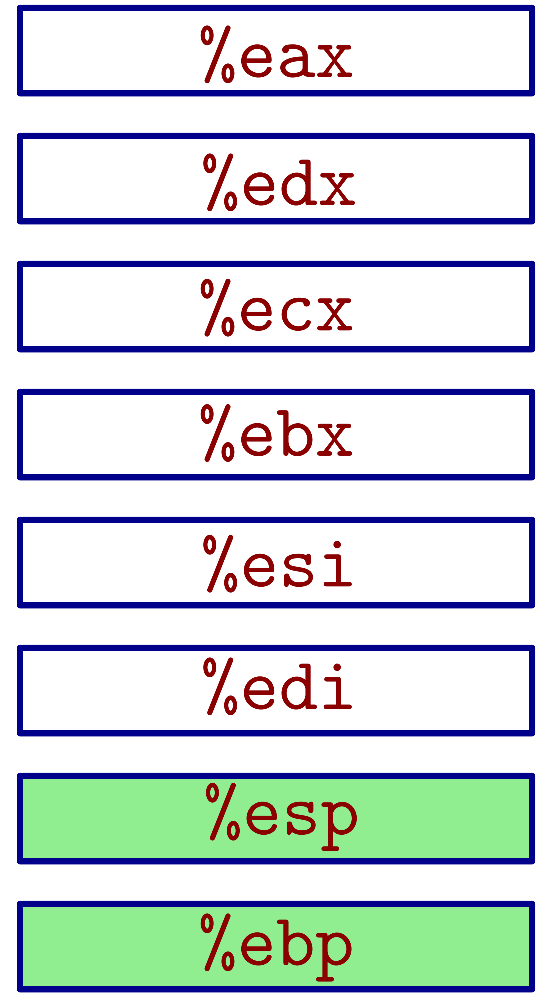
Moving Data

`movl` *Source,Dest*

- Move 4-byte ("long") word
- Lots of these in typical code

Operand Types

- Immediate: Constant integer data
  - Like C constant, but prefixed with `'$'`
  - E.g. `$0x400, $-533`
  - Encoded with 1, 2, or 4 bytes
- Register: one of 8 integer registers
  - But `%esp` and `%ebp` reserved for special use
  - Others have special uses for particular instructions
- Memory: 4 consecutive bytes of memory
  - Various *"address modes"*



`movl` Operand Combinations

[C]

	Source	Dest	Example	C Analog
movl	*Imm*	*Reg*	`movl $0x4,%eax`	`temp = 0x4 ;`
		Mem	`movl $-147,12(%ebx)`	`*(p+3) = -147;`
	Reg	*Reg*	`movl %eax,%edx`	`temp2 = temp1;`
		Mem	`movl %eax,(%edx)`	`*p = temp ;`
	Mem	*Reg*	`movl (%eax),%edx`	`temp = *p`

Cannot perform memory-memory transfers in a single instruction

Simple Addressing Modes Example

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

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

Pro-
logue

Body

Epi-
logue

Variable	Register	Memory
yp	%ecx	12(%ebp)
xp	%edx	8(%ebp)
t1	%eax	
t0	%ebx	

Simple Addressing Modes Example

[C]

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

ecx = yp

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

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

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

Prologue

Body

Epilogue

Variable	**Register**	**Memory**
yp	%ecx	12(%ebp)
xp	%edx	8(%ebp)
t1	%eax	
t0	%ebx	

[C]

Simple Addressing Modes Example

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

ecx = yp

edx = xp

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

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

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

Pro-logue

Body

Epi-logue

Variable	Register	Memory
yp	%ecx	12(%ebp)
xp	%edx	8(%ebp)
t1	%eax	
t0	%ebx	

Simple Addressing Modes Example

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

```
ecx = yp
edx = xp
eax = *yp
```

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

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

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

Pro-logue

Body

Epi-logue

Variable	**Register**	**Memory**
yp	%ecx	12(%ebp)
xp	%edx	8(%ebp)
t1	%eax	
t0	%ebx	

Simple Addressing Modes Example

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

```
ebx = *xp
*xp = eax
*yp = ebx
```

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

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

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

Pro-logue

Body

Epi-logue

Variable	Register	Memory
yp	%ecx	12(%ebp)
xp	%edx	8(%ebp)
t1	%eax	
t0	%ebx	

Address Computation Example

Assume:

%edx	0xf000
%ecx	0x100

Expression	Computation	Address	Example	Effect
0x8(%edx)	0xf000+0x8	0xf008	movl 0x08(%edx),%eax leal 0x08(%edx),%eax	%eax = M[0xf008] %eax = 0xf008
(%edx,%ecx)	0xf000 + 0x100	0xf100	movl (%edx,%ecx),%eax leal (%edx,%ecx),%eax	%eax = M[0xf100] %eax = 0xf100
(%edx,%ecx,4)	0xf000 + 4*0x100	0xf400	movl (%edx,%ecx,4),%eax leal (%edx,%ecx,4),%eax	%eax = M[0xf400] %eax = 0xf400
0x80(,%edx,2)	2*0xf000 + 0x80	0x1e080	movl 0x80(,%edx,2),%eax leal 0x80(,%edx,2),%eax	%eax = M[0x1e080] %eax = 0x0x1e080

`leal` *Src*, *Dest*

- *Src* is an address-mode expression
- Set *Dest* to address denoted by expression

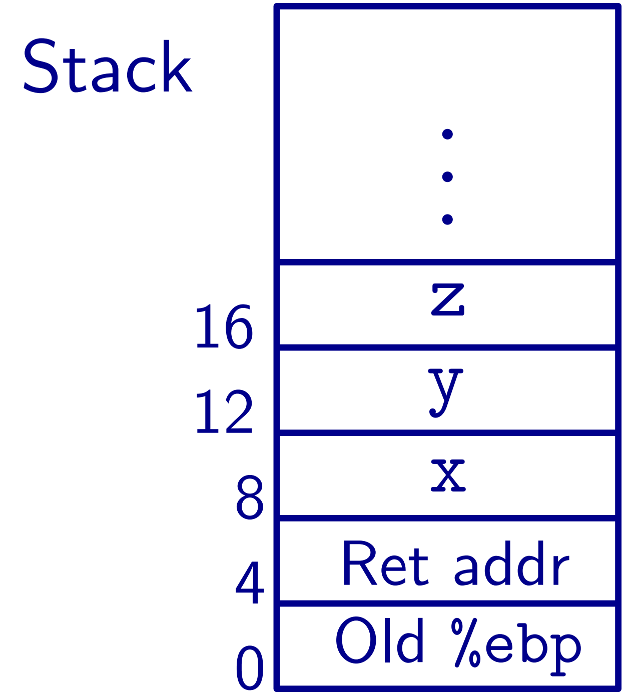
Uses:
- Translation of statements of the form `p = &x[i]`
- Computing arithmetic expressions of the form `x+k*y`, where `k = 1, 2, 4, 8`

Some Arithmetic Operations

<table>
<tr><th>Format</th><th>Computation</th></tr>
<tr><td><code>addl S,D</code></td><td><code>D += S</code></td></tr>
<tr><td><code>subl S,D</code></td><td><code>D -= S</code></td></tr>
<tr><td><code>imull S,D</code></td><td><code>D *= S</code></td></tr>
<tr><td><code>sall S,D</code></td><td><code>D <<= S</code></td></tr>
<tr><td><code>sarl S,D</code></td><td><code>D >>= S</code> (arithmetic)</td></tr>
<tr><td><code>shrl S,D</code></td><td><code>D >>= S</code> (logical)</td></tr>
<tr><td><code>xorl S,D</code></td><td><code>D ^= S</code></td></tr>
<tr><td><code>andl S,D</code></td><td><code>D &= S</code></td></tr>
<tr><td><code>orl S,D</code></td><td><code>D |= S</code></td></tr>
<tr><td><code>incl D</code></td><td><code>D ++</code></td></tr>
<tr><td><code>decl D</code></td><td><code>D --</code></td></tr>
<tr><td><code>negl D</code></td><td><code>D = - D</code></td></tr>
<tr><td><code>notl D</code></td><td><code>D =   D</code></td></tr>
</table>

Arithmetic Example

```
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 ;
}
```




```
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)
```

Data Representations

Sizes of C object in bytes

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

x86-64 General Purpose Registers

%rax	%eax	%r8	%r8d
%rdx	%edx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rbx	%ebx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14d	%r14d
%rbp	%ebp	%r15	%r15d

Extend existing registers (which are still accessible with double word-size instructions). Add 8 new registers (whose lower halves become accessible via word-size instructions too).

64-bit Example: `swap`

[A]

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

```
swap:
    movl (%rdi), %edx
    movl (%rsi), %eax
    movl %eax, (%rdi)
    movl %edx, (%rsi)
    ret
```

- Operands passed in registers
  - First (xp) in %rdi, second (yp) in %rsi
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers %eax and %edx
  - `movl` (long) operation

64-bit Example: `swap` with long ints

[A]

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

```
swap_l:
    movq (%rdi), %rdx
    movq (%rsi), %rax
    movq %rax, (%rdi)
    movq %rdx, (%rsi)
    ret
```

- 64-bit data
  - Data held in registers %rax and %rdx
  - `movq` operations (quad-word size)

Condition Codes

[C]

Single Bit Registers

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

Not set by `lea`, `inc`, or `dec` instructions

Implicitly Set by Arithmetic Operations

`addl` *Src,Dest* `addq` *Src,Dest*

C analog: `t = a + b` (`a` $=$ *Src*, `b` $=$ *Dest*)

- 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)
```

Setting Condition Codes

Explicit Setting by Compare Instruction

`cmpl` *Src1,Src2*   `cmpq` *Src1,Src2*

- `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

Setting Condition Codes

Explicit Setting by Test Instruction

`testl` *Src2,Src1*

`testq` *Src2,Src1*

- Sets condition codes based on value of *Src1* and *Src2*
  - ◇ Useful to have one of the operands be a mask
- `testl b,a` like computing `a&b` without setting destination
- ZF set when `a&b == 0`
- SF set when `a&b < 0`

Reading Condition Codes

SetX Instructions

- Set single byte based on combinations of condition codes

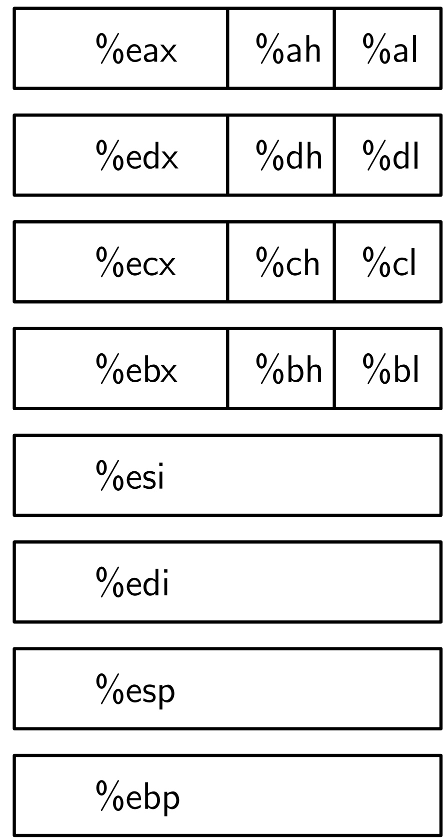
SetX	**Condition**	Description
`sete`	`ZF`	Equal/Zero
`setne`	`~ZF`	Not Equal/Not Zero
`sets`	`SF`	Negative
`setns`	`~SF`	Non-negative
`setg`	`~(SF^OF)&~ZF`	Greater (signed)
`setge`	`~(SF^OF)`	Greater or Equal (signed)
`setl`	`(SF^OF)`	Less (Signed)
`setle`	`(SF^OF)|ZF`	Less or Equal (signed)
`seta`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

Reading Condition Codes

[A]

SetX Instructions

- Set single byte based on combinations of condition codes
- One of 8 addressable byte registers
  - Embedded within first 4 integer registers
  - Does not alter remaining 3 bytes
  - Typically use `movzbl` to finish job




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

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

Note inverted ordering

Reading Condition Codes: x86-64

[A]

SetX Instructions

- Set single byte based on combinations of condition codes
  - Does not alter remaining 7 bytes

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

```
long lgt (long x, long y) {
    return x > y;
}
```

- x86-64 arguments
  - x in `%rdi`
  - y in `%rsi`

- Translation (same for both)

```
xorl %eax, %eax    # eax = 0
cmpq %rsi, %rdi    # Compare x : y
setg %al           # al = x > y
```

32-bit instructions set high order 32 bits to 0

Jumping

jX Instructions

- jump to different parts 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	Non-negative
`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

[C]

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

```
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
```

Entry

Body 1

Exit

Body 2

Conditional Transfers: x86-64

[A]

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

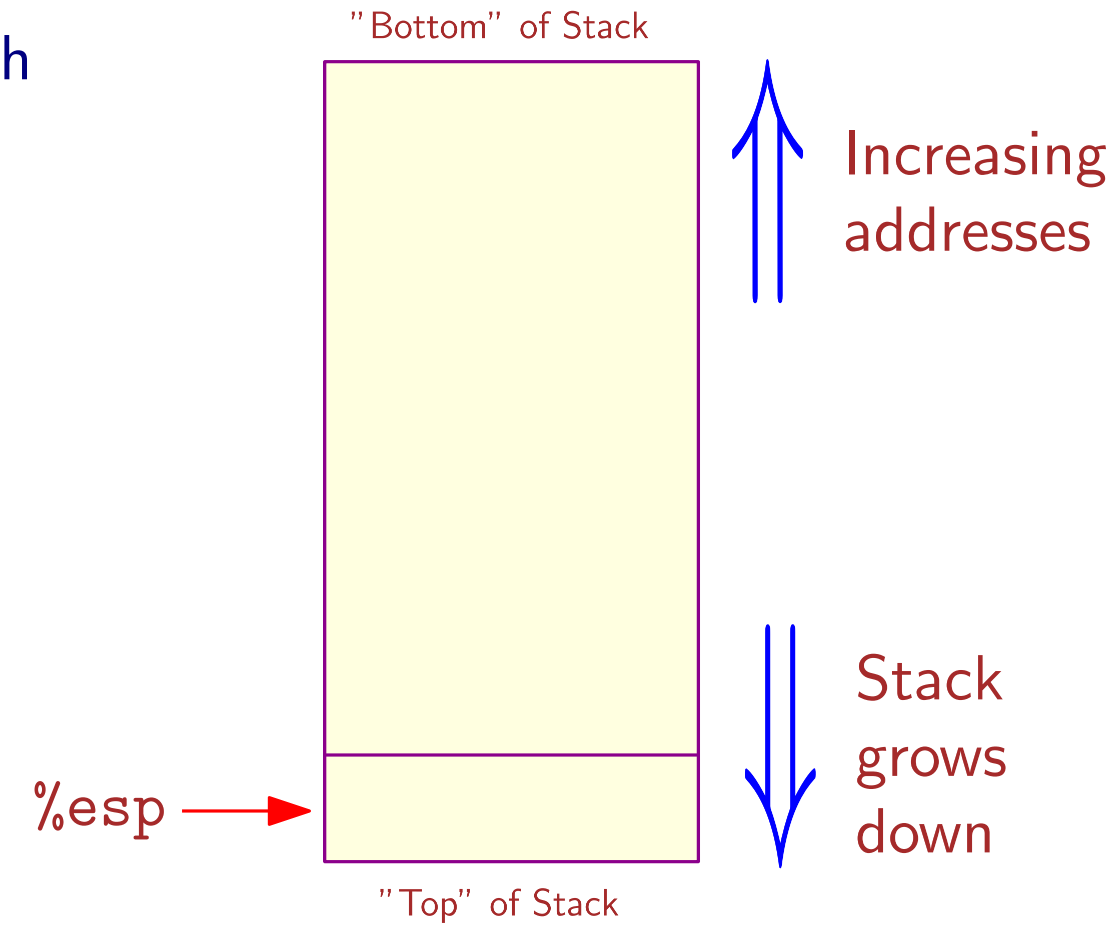
```
absdiff: # x in %edi, y in %esi
  movl %edi, %eax      # v = x
  movl %esi, %edx      # ve = y
  subl %esi, %eax      # v -= y
  subl %edi, %edx      # ve -= x
  cmpl %esi, %edi      # x:y
  cmovle %edx, %eax    # v=ve if <=
  ret
```

Conditional move instruction

- `cmov`*C Src*, *Dest*
- Move value from *Src* to *Dest* if condition holds
- More efficient than conditional branching
  - Simple and predictable control flow

IA32 Stack

- Region of memory managed with stack discipline
- Grows towards lower addresses
- Register %esp indicates lowest stack address
  - address of top element
- Operations
  - Pushing
    - `pushl` *Src*
    - Fetch operand at *Src*
    - Decrement %esp by 4
    - Write operand at address given by %esp
  - Popping
    - `popl` *Dest*
    - Reverse of push



Local variables and procedure arguments are stored on the stack to allow for recursion.

Procedure Control Flow

- Use stack to support procedure call and 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

Stack-Based Languages

[C]

Languages that support recursion

- e.g. C, Pascal, Java
- Code must be *reentrant*
  - Multiple simultaneous instantiations of single procedure
- Need some place to store state of each installation
  - Arguments
  - Local variables
  - Return pointer

Require *entry* and *exit* code for each procedure

More when we learn about procedures!

Stack discipline

- State for given procedure needed for limited time
  - From when called to when return occurs
- Callee returns before caller does

Stack allocated in *frames*

- State for single procedure instantiation

Tips

- Our toy compilers will generate Pentium assembly code
  - Need to be reasonably familiar with arithmetic/logic and branching instructions
- When looking for the instruction that implements a specific operator
  - Write a small C program that has the operation that you want to use
  - Translate into assembly, and check how GCC does it
  - Try using `-masm=intel` to get INTEL syntax for assembly language
- Full Pentium architecture manuals, including the complete set of instructions:
  - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html/