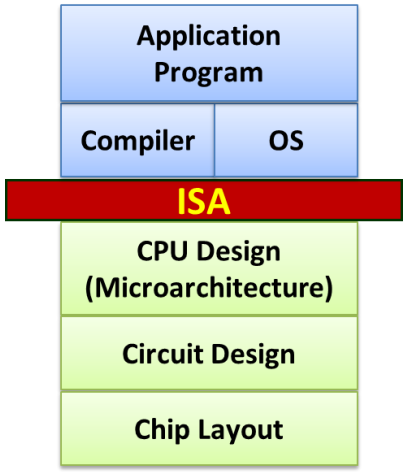
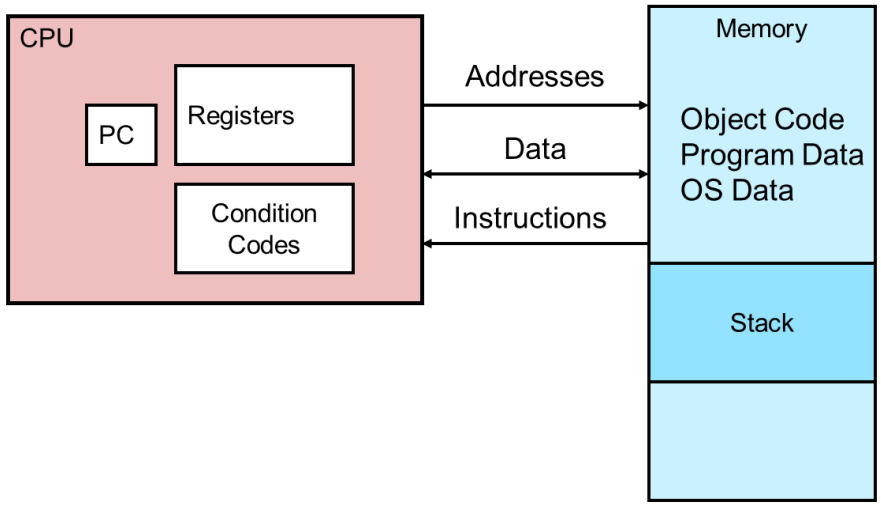
REVE1

Processor Architecture and the Instruction Set Architecture



Module Outline

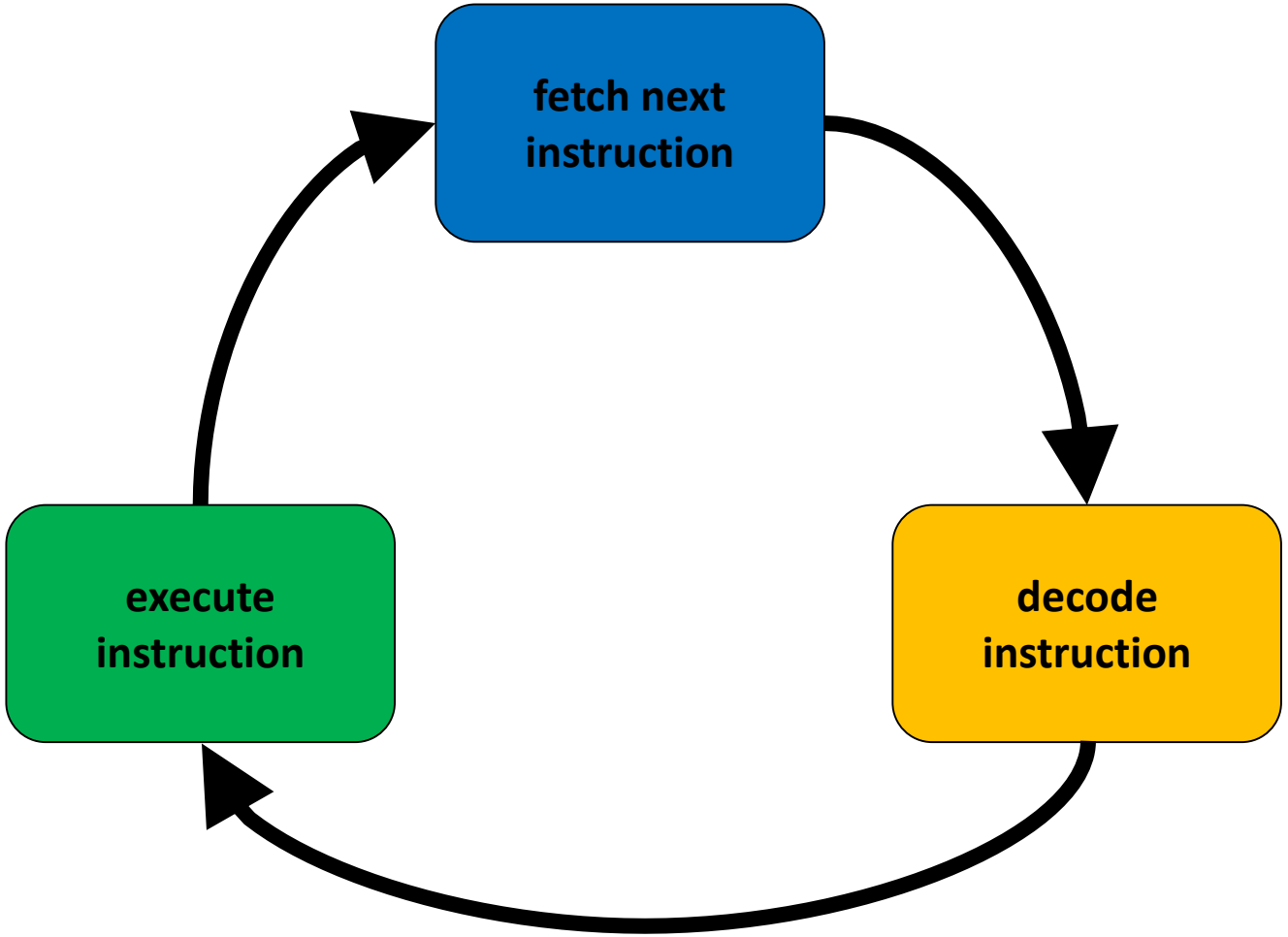
- Overview
- History of the Intel Processor Architecture
- The x86 ISA
  - The Programmer's View
  - Data Types and Alignment
  - First Steps
  - Accessing Information
  - Arithmetic and Logical Operations
  - Control Operations
  - Procedures
- Module Summary


CPU
PC
Registers
Condition Codes
Addresses
Data
Instructions
Memory
Object Code
Program Data
OS Data
Stack

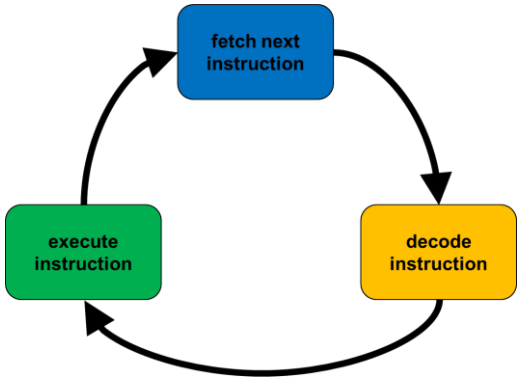
Overview

Life as a Processor

- As long as there is power



Life as a Processor



- **Lots of problems to solve**
  - fetch next instruction
    - “next instruction”
    - fetch from where
  - decode instruction
    - instruction format
    - supported operations
    - supported operands
    - where do operands come from/go to?
  - execute instruction
    - implementation

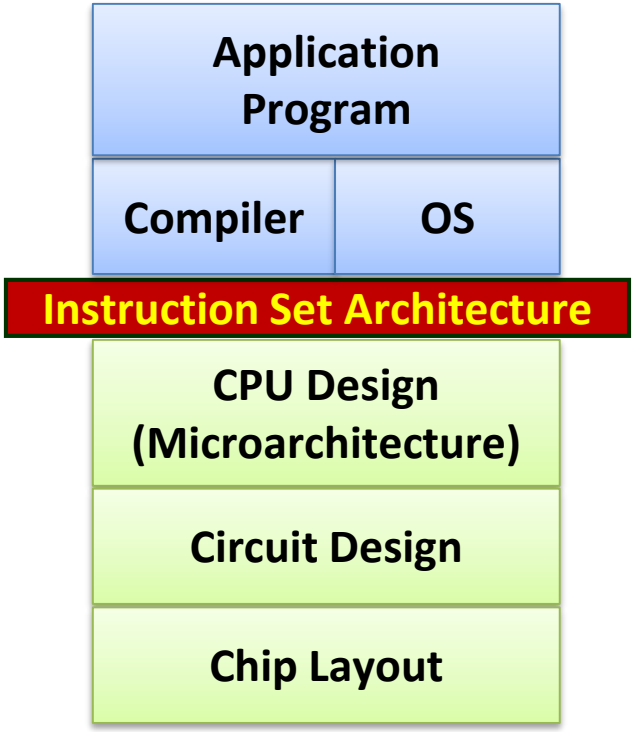
Instruction Set Architecture (ISA)

> **"the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation"**
>
> *-- Amdahl, Blaauw, and Brooks, Architecture of the IBM System/360, IBM Journal of Research and Development, April 1964.*

- The visible interface between software and hardware
- What the user (OS, compiler, programmer, …) needs to know in order to reason about how the machine behaves
- Abstracted from the details of how it may accomplish its task

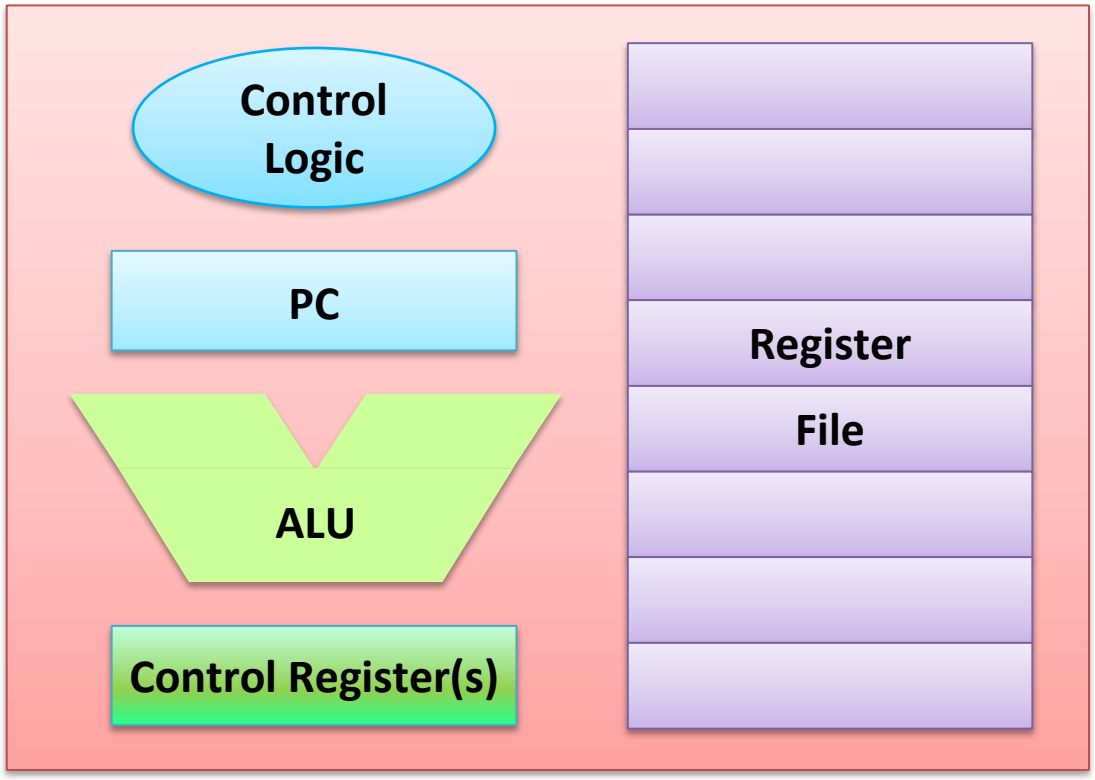
Instruction Set Architecture (ISA)

- **The ISA abstracts the details of the processor implementation**
  - Above: how to program machine
    - Processors execute instructions in sequence
  - ISA: hardware/software interface
  - Below: what needs to be built
    - Use variety of tricks to make it run fast



Instruction Set Architecture (ISA)

- **The ISA defines the programmer-visible state of a processor architecture**



  - the ISA tells us what we can expect the processor to do when we execute instructions (but not *how* the processor does it)

Instruction Set Architecture (ISA)

- **The ISA defines the programmer-visible state of a processor architecture**
  - **Memory addressing**
    - interpretation of addresses
      - bytes, half words, words, double words, …?
    - byte ordering in memory
      - little-endian vs. big-endian
    - addressing modes
      - how to access objects in memory

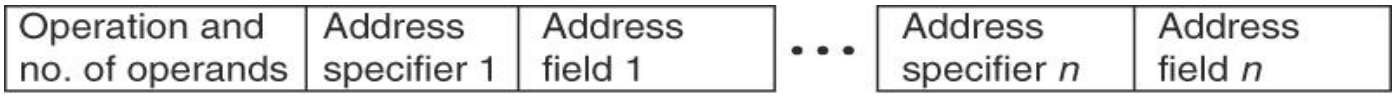
Instruction Set Architecture (ISA)

- **The ISA defines the programmer-visible state of a processor architecture**
  - **Type and size of operands**
    - specifying types
      - byte, half word, word, float, double…
  - **Operations**
    - arithmetic and logical
    - data transfer
    - control
    - system
    - floating point
    - string
    - multimedia

Rank	x86 operation	%
1	load	22
2	conditional branch	20
3	compare	16
4	store	12
5	add	8
6	and	6
7	sub	5
8	move reg-reg	4
9	call	1
10	return	1
	total	**96**

Instruction Set Architecture (ISA)

- **The ISA defines the programmer-visible state of a processor architecture**
  - **Instruction encoding**
    - defines binary representation for each operation and operand
    - fixed size vs. variable size

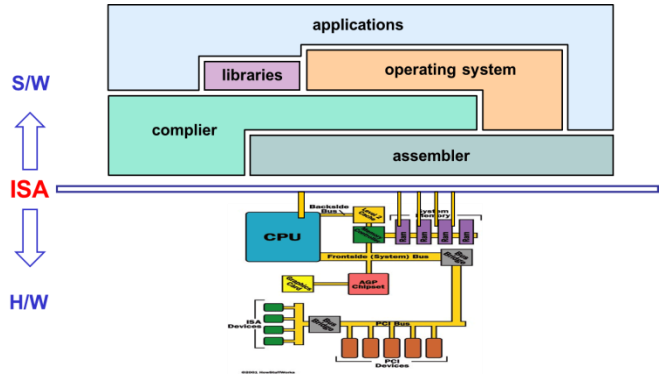


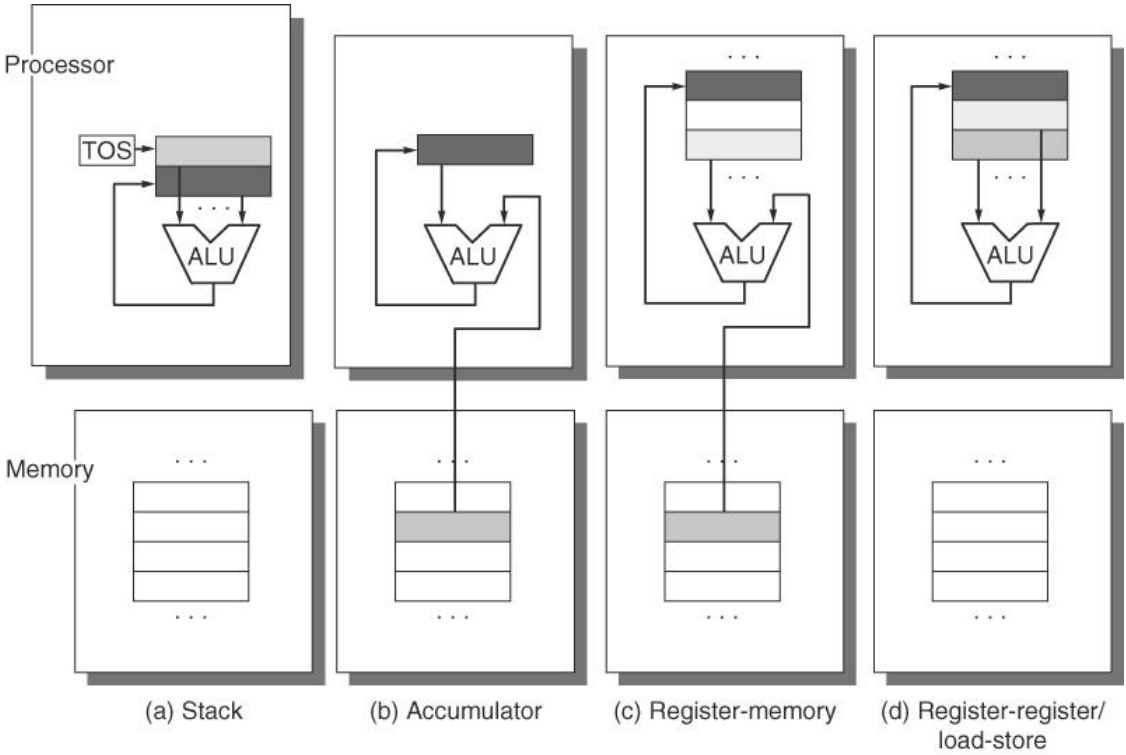
(a) Variable (e.g., Intel 80x86, VAX)

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

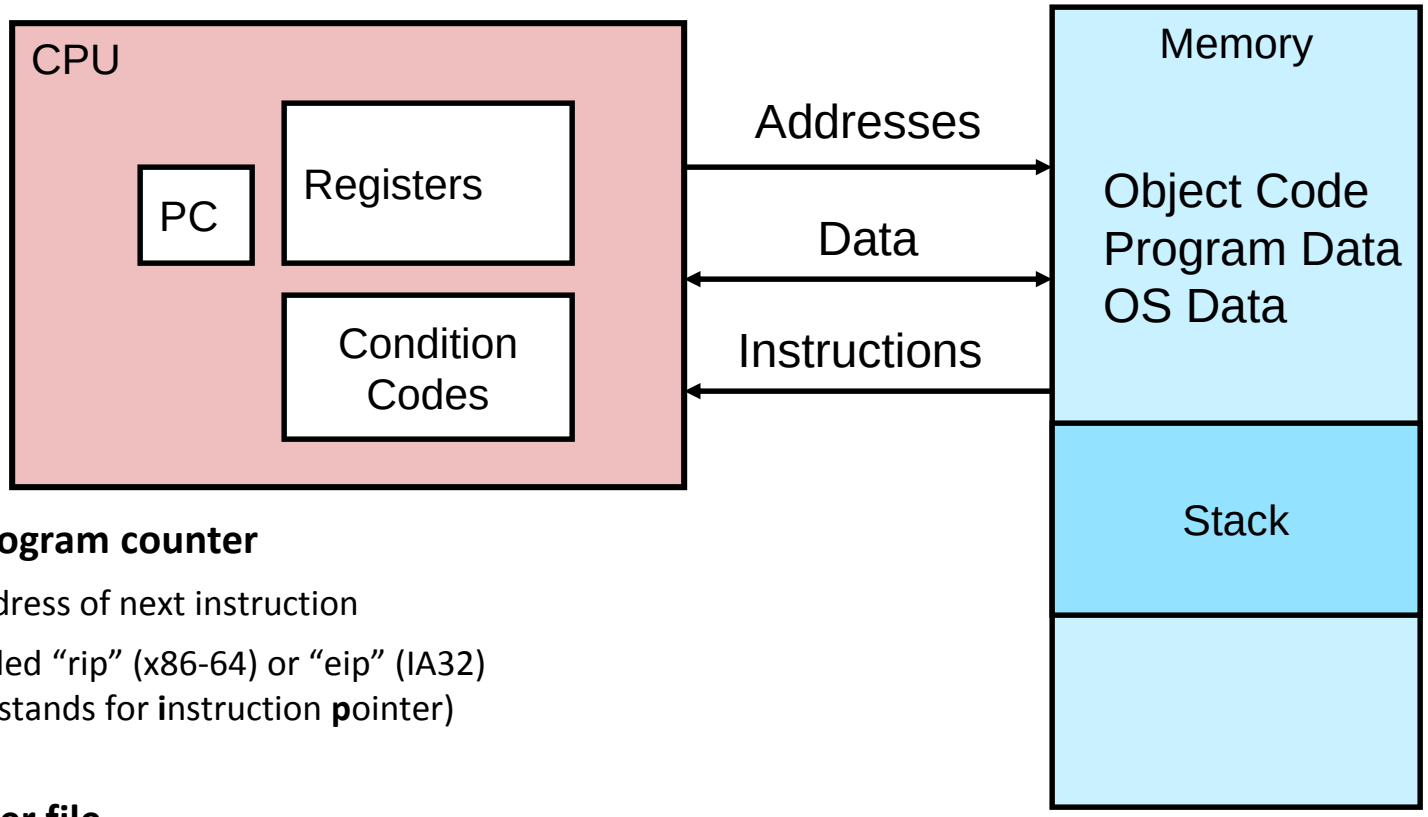
Instruction Set Architecture (ISA)

- **ISA classification**
  - RISC vs CISC
  - general-purpose
    - two or three operands?
    - # of memory operands?
    - classification
      - load-store
      - register-memory
      - memory-memory
  - stack
  - accumulator





Assembly Programmer's View



- **PC: Program counter**
  - Address of next instruction
  - Called "rip" (x86-64) or "eip" (IA32)
    (ip stands for **i**nstruction **p**ointer)
- **Register file**
  - Fast & temporary storage of values
- **Condition codes**
  - Status information about most recent arithmetic operation
  - Used for conditional branching
- **Memory**
  - Byte addressable array
  - Code, user data, (some) OS data
  - Includes stack to support procedures

History of the Intel Processor Architecture

Intel x86 Processors

- (Still) dominate laptop/desktop/server market
- Evolutionary design
  - Backwards compatible up until 8086, introduced in 1978
  - Added more features as time goes on
- Complex instruction set computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!
    - In terms of speed. Less so for low power.

Intel x86 Evolution: Milestones

Name	*Date*	*Transistors*	*MHz*
8086	**1978**	**29K**	**5-10**

- First 16-bit processor. Basis for IBM PC & DOS
- 1MB address space

386	**1985**	**275K**	**16-33**

- First 32 bit processor , referred to as IA32
- Added “flat addressing”
- Capable of running Unix
- 32-bit Linux/gcc uses no instructions introduced in later models

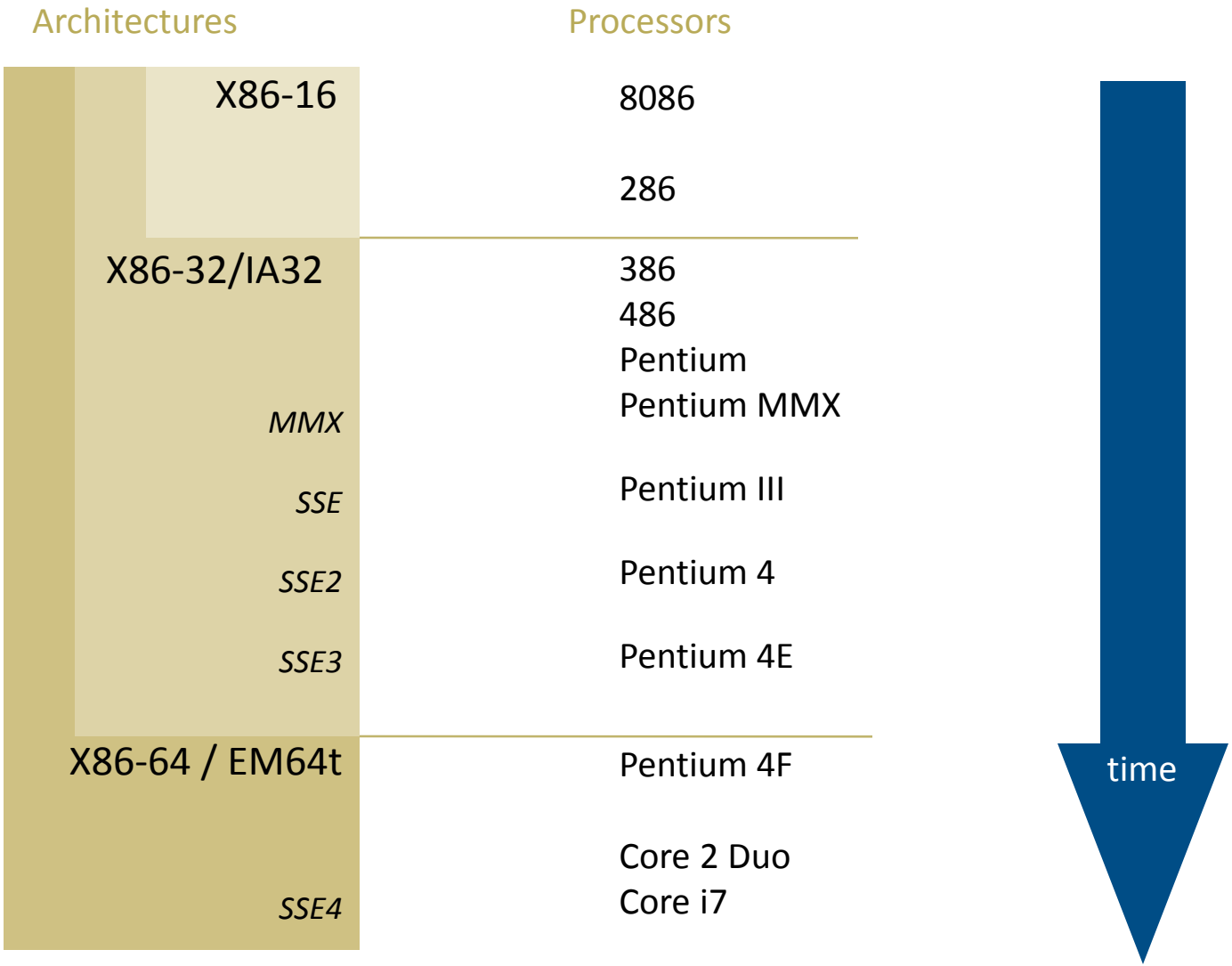
Pentium 4F	**2004**	**125M**	**2800-3800**

- First 64-bit processor, referred to as x86-64

Core i7	**2008**	**731M**	**2667-3333**

- multiple cores

Intel x86 Processors: Overview

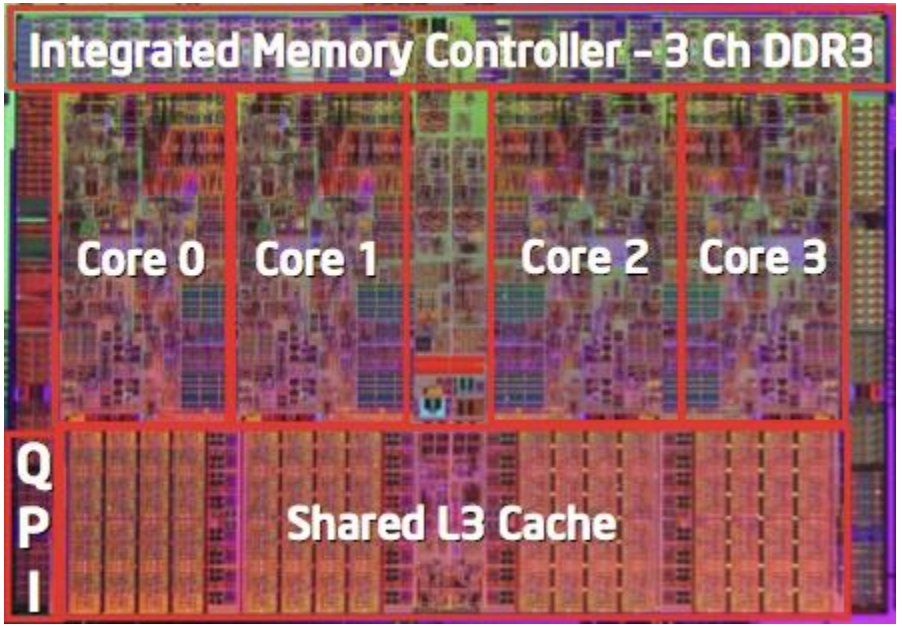


IA: often redefined as latest Intel architecture

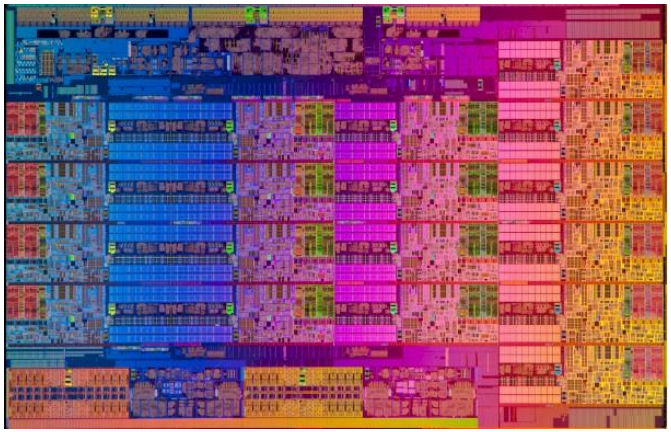
Intel x86 Processors

- **Machine Evolution**

Name	*Date*	*Transistors*
386	1985	0.3M
486	1989	1.9M
Pentium	1993	3.1M
Pentium/MMX	1997	4.5M
PentiumPro	1995	6.5M
Pentium III	1999	8.2M
Pentium 4	2001	42M
Core 2 Duo	2006	291M
Core i7 (4 cores)	2008	731M
Core i7 (6 cores)	2011	2'270M
Xeon E5 v4 (22 cores)	2016	7'200M (est.)
Xeon Plat. 8284 (28 c)	2019	8'000M (est.)



Core i7 (45nm)



Xeon E5 v3 (22nm)

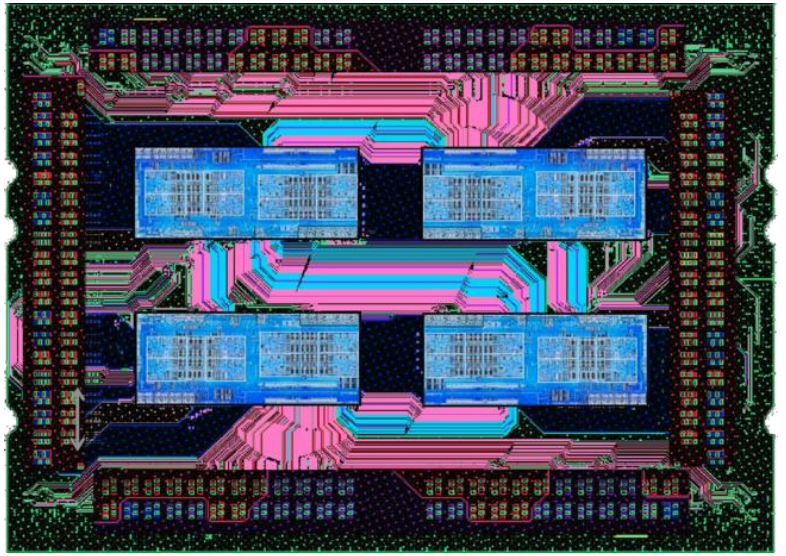
- Added Features
  - Instructions to support multimedia operations
    - Parallel operations on 1, 2, and 4-byte data, both integer & FP
  - Instructions to enable more efficient conditional operations

AMD Zen Architecture

- **Clever Design**
  - scalable modular design
  - AMD EPYC 2 (32 cores / 64 threads), released 2019 with only 4’800M transistors

Zen Architecture (2017)

source: Wikichip



Zen Architecture (2017)

source: Wikichip

Transistor Technology

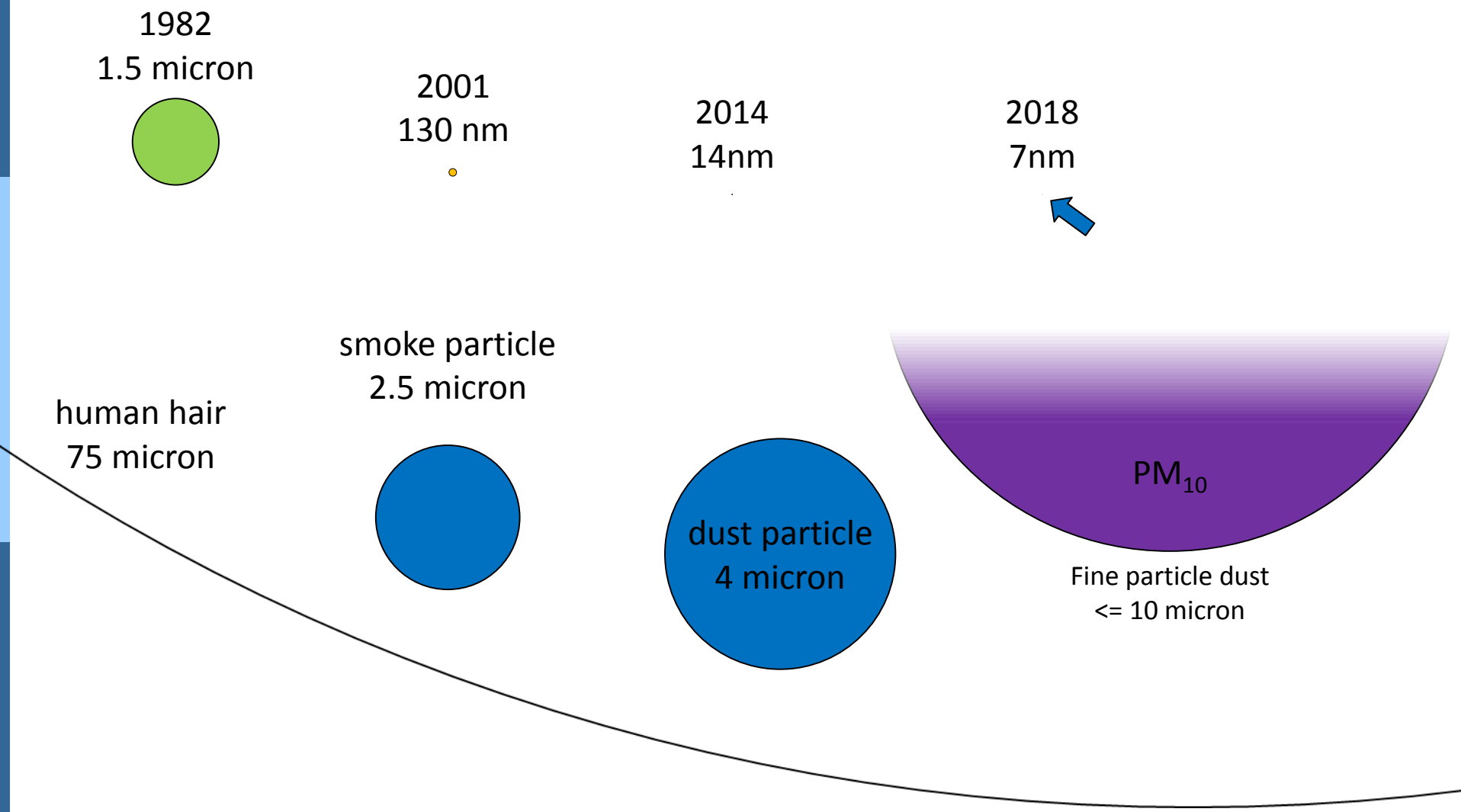
1982
1.5 micron

2001
130 nm

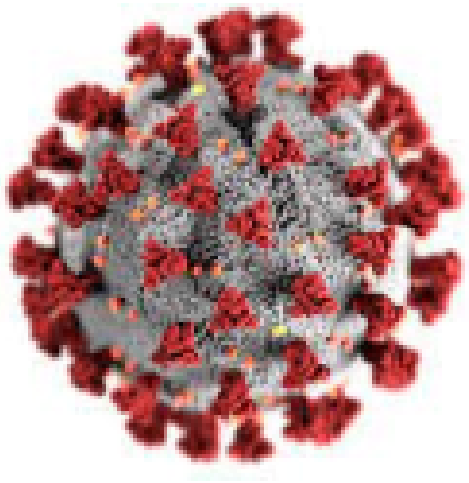
2014
14nm

2018
7nm

Transistor Technology – A Comparison



Transistor Tech – A Contemporary Comparison



2020
90nm
(50-140 nm)

2014
14nm

2018
7nm

New Species: ia64, then IPF, then Itanium,…

Name	*Date*	*Transistors*
Itanium	**2001**	**10M**

- First shot at 64-bit architecture: first called IA64
- Radically new instruction set designed for high performance
- Can run existing IA32 programs
  - On-board “x86 engine”
- Joint project with Hewlett-Packard

Itanium 2	**2002**	**221M**

- Big performance boost

Itanium 2 Dual-Core	**2006**	**1.7B**

- Itanium has not taken off in marketplace
  - very fast (esp. FP), very hot, and very expensive
  - Lack of backward compatibility, no good compiler support, Pentium 4 got too good, overtaken by 64-bit x86 designs

x86 Clones: Advanced Micro Devices (AMD)

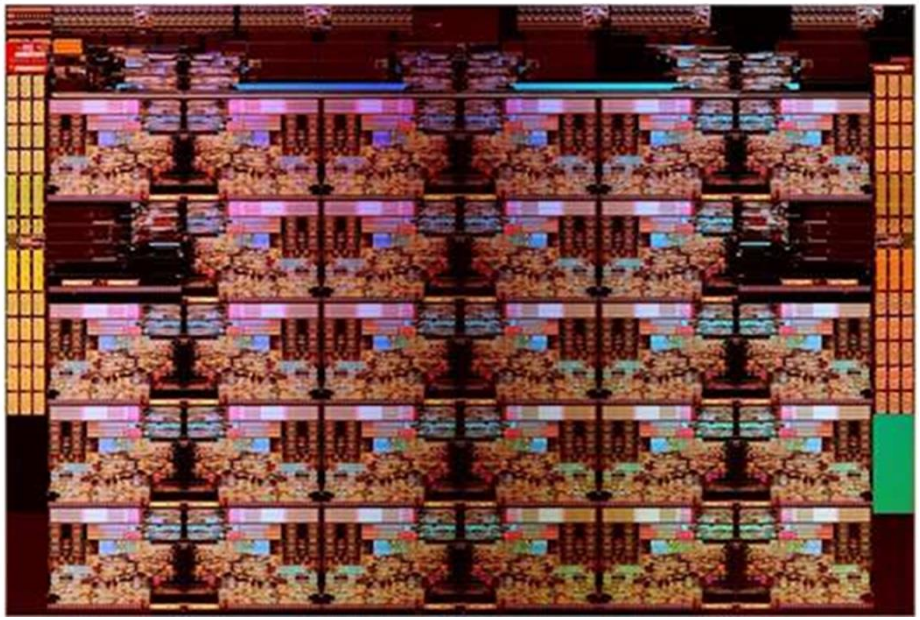
- Historically
  - AMD has followed just behind Intel
  - A little bit slower, a lot cheaper
- Then
  - Recruited top circuit designers from Digital Equipment Corp. and other downward trending companies
  - Built Opteron: tough competitor to Pentium 4
  - Developed x86-64, their own extension to 64 bits
- About 10 years back
  - Intel much quicker with multi-core design
  - Intel far ahead in performance
  - Intel em64t backwards compatible to x86-64
- Today
  - AMD outperforms Intel in terms of cost, #cores, design, and matches single-thread performance!

Intel's 64-Bit

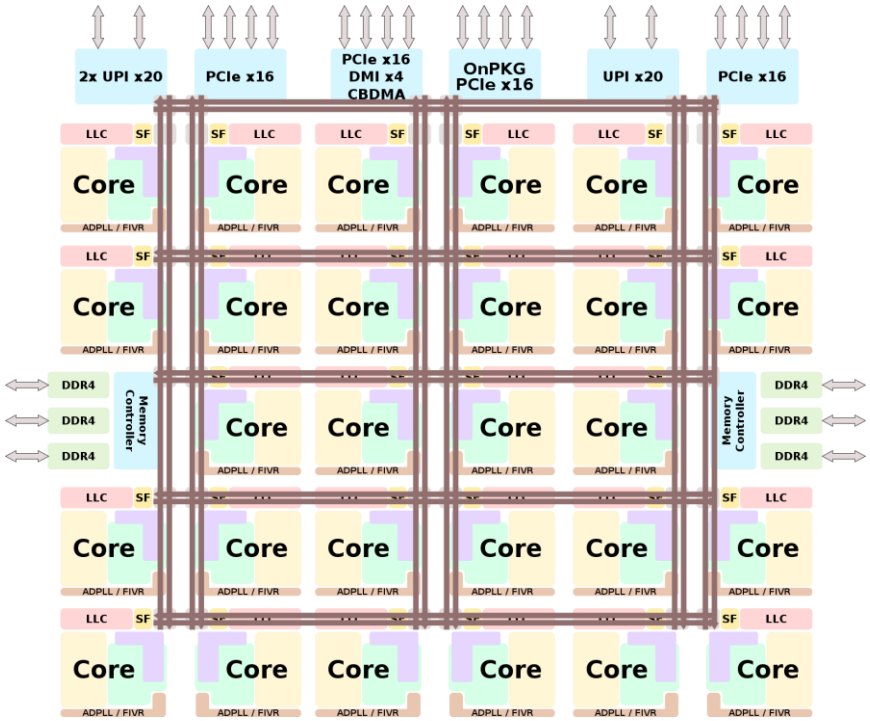
- Intel Attempted Radical Shift from IA32 to IA64
  - Totally different architecture (Itanium)
  - Executes IA32 code only as legacy
  - Application performance disappointing
- AMD Stepped in with Evolutionary Solution
  - x86-64 (now called "AMD64")
- Intel Felt Obligated to Focus on IA64
  - Hard to admit mistake or that AMD is better
- 2004: Intel Announces EM64T extension to IA32
  - Extended Memory 64-bit Technology
  - Almost identical to x86-64

Intel Cascade Lake SP

- Current leader: Xeon Platinum 8284 (Cascade Lake, July 2019)
  - ~8 billion transistors
  - 3/4 GHz
  - virtualization support
  - no GPU on chip



source: WikiChip



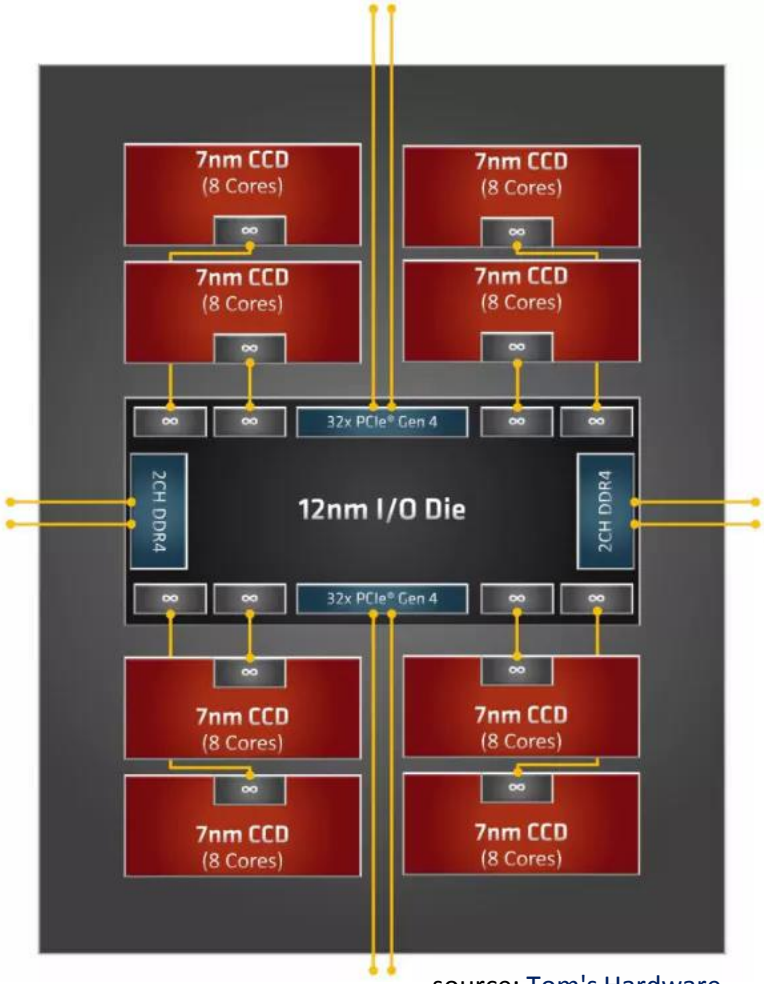
source: WikiChip

- up to 28 cores, 56 threads
- per core: 896KB L1(I/D), 1MB L2 cache
- 38.5MB shared L3 cache
- 240W TPD

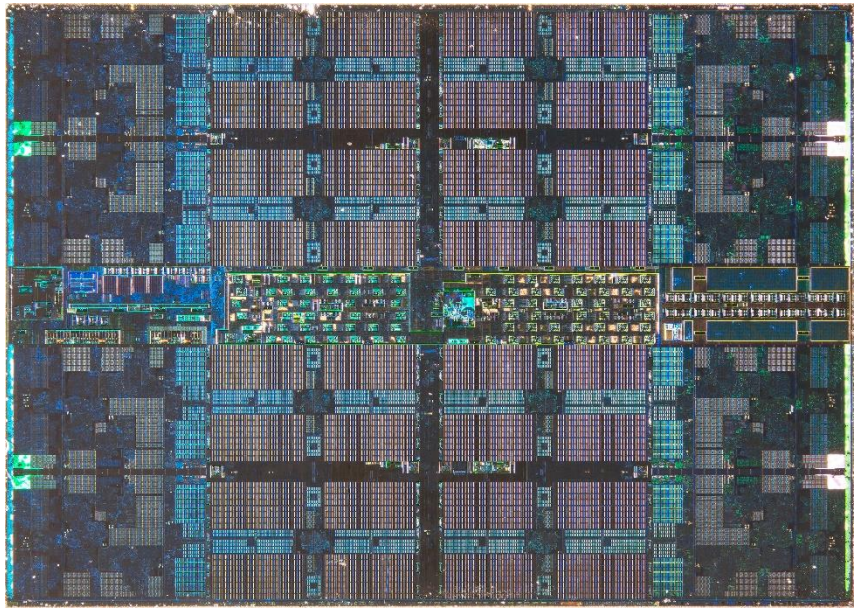
HOCHSCHULE LUZERN

AMD Zen2 Architecture

- AMD Threadripper 3990X
  (Zen2, July 2019)
  - 64 cores, 128 threads
  - 40 billion transistors
    (4 billion per 8-core chiplet)
  - 3/4 GHz



source: Tom's Hardware

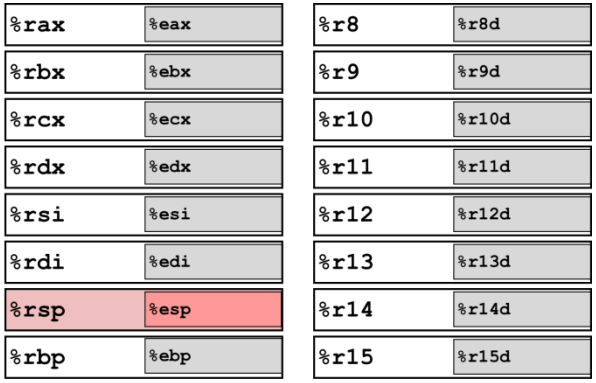


source: flickr

- Built by combining 8-core ‘core-chiplets’
- 32MB L2, 256MB L3 cache
- 280W TPD

Our Coverage

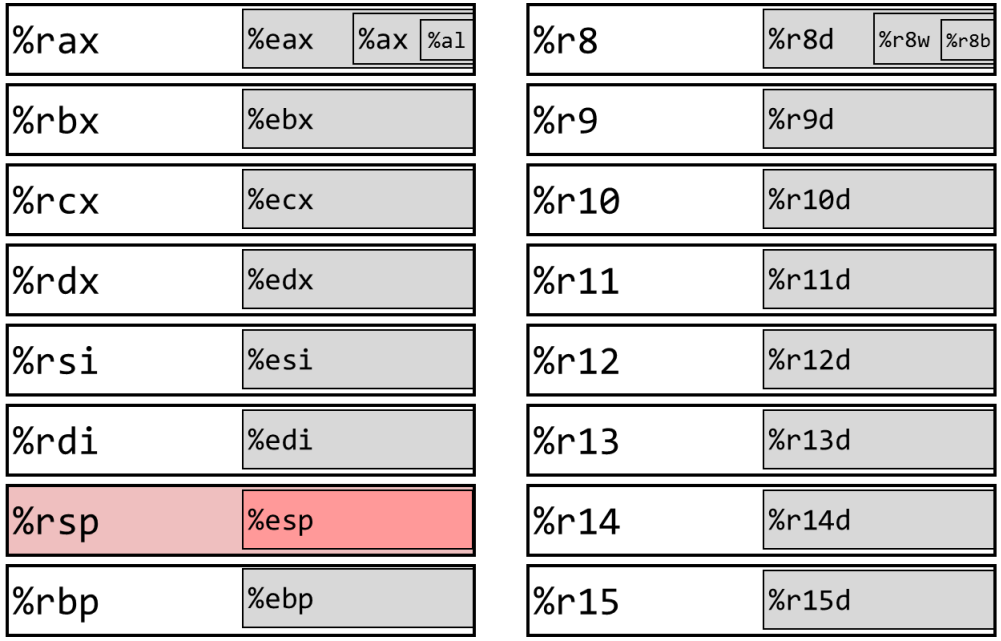
- x86-64/EM64T
  - the 64-bit standard
- IA32
  - the traditional x86



The x86 Instruction Set Architecture:

The Programmer's View

x86_64 ISA

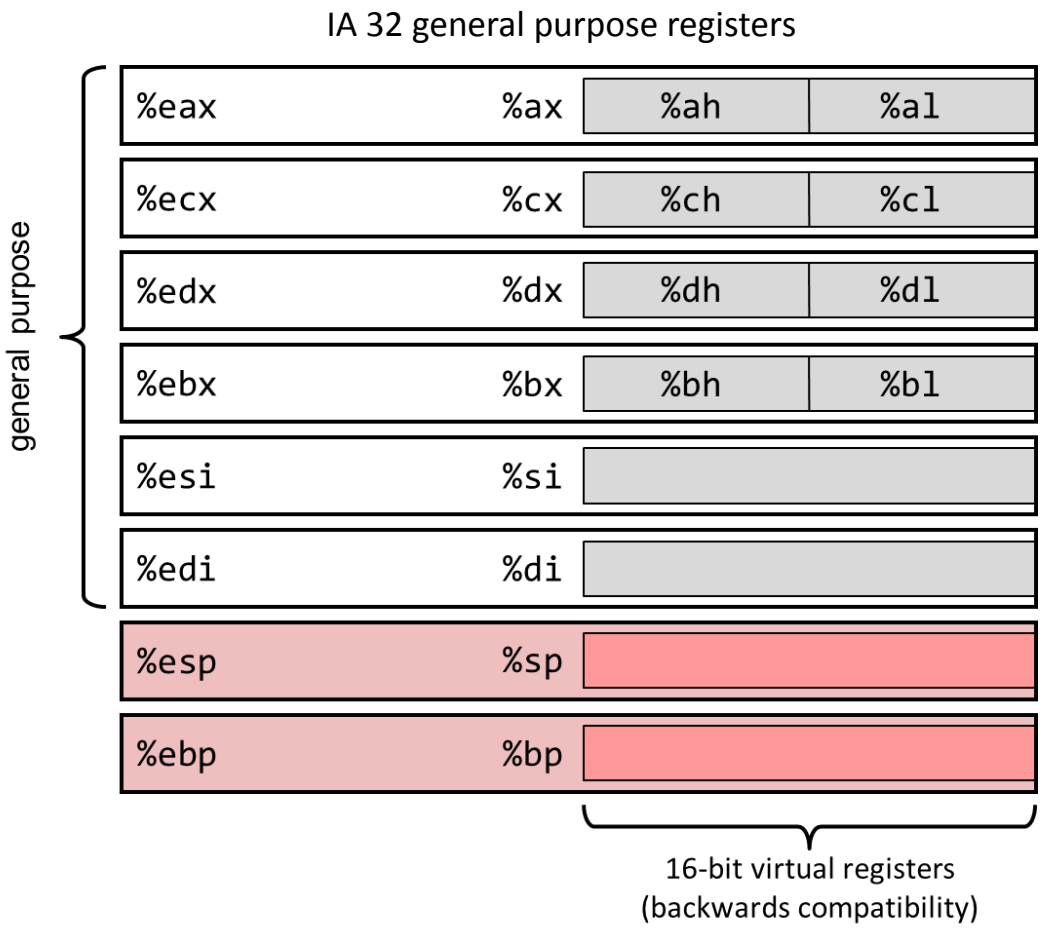


x86_64 general purpose registers

- Register-memory architecture
  - 2 operands, max 1 memory
- Registers
  - 16 general purpose 64-bit registers, PC (rip), condition codes, processor status
- Byte-addressed memory, little-endian
- 1, 2, 4, 8, 16 byte data types

IA32 ISA

- Same as x86_64 except
  - fewer registers
  - PC is called eip
  - no 16byte (128bit) data type



HOCHSCHULE LUZERN

x86 ISA: Instruction Classes

- **Arithmetic, logic, bit operations**
  Perform arithmetic/logic functions on registers or memory data

  `add, sub, mul, inc, and, shl, rot, test`

- **Data transfer (load/store, conversion) operations**
  Transfer data between memory and registers

  `mov, push, pop`

- **Control operations**
  Transfer control (un)conditionally, call/return from functions

  `jmp, jXX, call, ret`

- **Special operations**
  string operations, processor control operations, …

  `rep stosb, clc, cli/sti, pushf/popf`

x86 ISA: Instruction Format

- **General form**

```
operation  [operands]
```

- **Two-operand format**

```
operation operand2, operand1
```

first operand also serves as destination:

```
add b, a        ↔      a = a + b
```

- careful with non-commutative operations!

```
subl   a, b    ≠    subl   b, a
```

x86 ISA: Instruction Format

- **Operand size specifier**
  - postfix to the operation
  - b (1 byte), w (2 bytes), l (4 bytes), q (8 bytes – x86_64 only)

- **Operand types**
  - immediate ($), register (%), or memory
  - at most one memory operand
  - register size specification
    - rX (64 bit), eX (32 bit), X (16 bit), Xh/l (8 bit)
    - example: rax, eax, ax, ah/al
    - not all combinations valid!

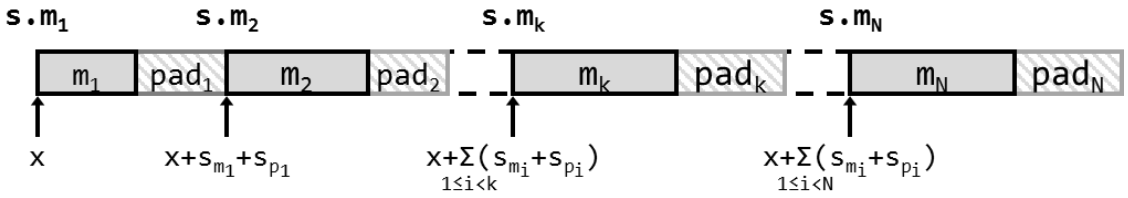
x86 ISA: Operand Specifiers

Type	Form	Operand value	Name
Immediate	`$Imm`	`Imm`	Immediate
Register	`Ea`	`R[Ea]`	Register
Memory	`Imm`	`M[Imm]`	Absolute
	`(Eb)`	`M[R[Eb]]`	Indirect
	`Imm(Eb)`	`M[R[Eb] + Imm]`	Base + displacement
	`(Eb, Ei)`	`M[R[Eb] + R[Ei]]`	Indexed
	`Imm(Eb, Ei)`	`M[R[Eb] + R[Ei] + Imm]`	Indexed
	`(, Ei, s)`	`M[R[Ei]*s]`	Scaled indexed
	`Imm(, Ei, s)`	`M[R[Ei]*s + Imm]`	Scaled indexed
	`(Eb, Ei, s)`	`M[R[Eb] + R[Ei]*s]`	Scaled indexed
	`Imm(Eb, Ei, s)`	`M[R[Eb] + R[Ei]*s + Imm]`	Scaled indexed

x86 ISA: Instruction Listing

- x86 instructions by architecture type
  https://en.wikipedia.org/wiki/X86_instruction_listings

- Intel Architecture Reference Manual
  https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html



The x86 Instruction Set Architecture:

Data Types and Alignment

Data Types

- Integer data of 1, 2, 4, 8, or 16 bytes
  - Data values
  - Addresses (untyped pointers)
- Floating point data of 4, 8, or 16 bytes
- No aggregate types such as arrays or structures
  - Just contiguously allocated bytes in memory

Recap: C Data Types

C Data Type	Size (bytes)		
	Typical 64-bit	x86-64	Intel IA32
char	1	1	1
short	2	2	2
int	4	4	4
long	8	8	4
long long	8	8	8
float	4	4	4
double	8	8	8
long double	8	10/16	10/12
*** (pointer)**	8	8	4
__[u]int128_t	16	16	-

x86 Data Representations

C Data Type	x86-64	Intel IA32
char	b	b
short	w	w
int	l	l
long	q	l
long long	q	(requires two 32-bit registers)
float	s	s
double	d	d
long double	(native Intel format)	
*** (pointer)**	q	l
__[u]int128_t	(requires two 64-bit registers)	(no direct support)

Data Alignment

- **Alignment of data**
  - Rules regarding the location of data in the memory of a computer system
    - required on some machines; advised on IA32
      - treated differently by IA32 Linux, x86-64 Linux, and Windows!
  - Basic rule
    - primitive data type requires *K* bytes → address must be K-byte aligned
    - K-byte aligned = address divisible by K
  - Example:
    - 4-byte integers must be located at addresses divisible by 4
      - valid addresses: 0, 4, 8, 12, 16, …
      - invalid addresses: 1,2,3,5,6,7,9,10,11,…

Data Alignment

- **Motivation for aligning data**
  - Memory accessed by (aligned) chunks of 4 or 8 bytes (system dependent)
    - inefficient to load or store datum that spans quad word boundaries
    - virtual memory management tricky when datum spans 2 pages

x86-64 Data Alignment Conventions

- **Base rule**
  - Primitive data type of *K* bytes should be K-byte aligned
- **Exceptions**
  - 10-byte primitive type (long double, extended): 16-byte aligned

Size (bytes)	Primitive Data Type	Alignment	
		Linux	Windows
1	char, …	1	1
2	short, …	2	2
4	int, float, …	4	4
8	double, long, long long, *, …	8	8
16	long double, __int128_t, …	16	16

IA32 Data Alignment Conventions

- **Base rule**
  - Primitive data type of *K* bytes should be K-byte aligned

- **Exceptions**
  - 8-byte primitive types (double, long long): 4-byte aligned on Linux, 8-byte aligned on Windows
  - 10-byte primitive type (long double, extended): 4-byte aligned

Size (bytes)	Primitive Data Type	Alignment	
		Linux	Windows
1	char, …	1	1
2	short, …	2	2
4	int, long, float, *, …	4	4
8	double, long long, …	**4**	**8**
12	long double, …	4	4

Example: Size & Alignment of Primitive Types

Finding out about sizes and alignments on your machine

```
#include <stdio.h>

#define SIZE(t) sizeof(t)
#define OFS(v) ((unsigned long)&v.data - (unsigned long)&v)
#define INFO(t) { struct { char dummy; t data; } s; \
                  printf("%-15s %2zu        %2lu\n", #t, SIZE(t), OFS(s)); }

void main(void)
{ printf("DATATYPE       SIZE   ALIGNMENT\n");
  INFO(char);
  INFO(short);
  INFO(int);
  INFO(long);
  INFO(long long);
  INFO(float);
  INFO(double);
  INFO(long double);
  INFO(void *);
}
```

size.c

Macro-expansion generates the following code pattern:

```
void main(void)
{ printf("DATATYPE       SIZE   ALIGNMENT\n");
  { struct { char dummy; char data; } s; printf("%-15s %2zu        %2lu\n",
    "char", sizeof(char), ((unsigned long)&s.data - (unsigned long)&s)); };
  { struct { char dummy; short data; } s; printf("%-15s %2zu        %2lu\n",
    "short", sizeof(short), ((unsigned long)&s.data - (unsigned long)&s)); };
  …
```

Example: Size & Alignment of Primitive Types

Finding out about sizes and alignments on your machine

```
$ gcc –m64 –o size.64 size.c
$ ./size.64
DATATYPE        SIZE    ALIGNMENT
char              1          1
short             2          2
int               4          4
long              8          8
long long         8          8
float             4          4
double            8          8
long double      16         16
void *            8          8
$
```

```
$ gcc –m32 –o size.32 size.c
$ ./size.32
DATATYPE        SIZE    ALIGNMENT
char              1          1
short             2          2
int               4          4
long              4          4
long long         8          4
float             4          4
double            8          4
long double      12          4
void *            4          4
$
```

Maintaining Data Alignment

- **Alignment rules upheld by compiler / assembly programmer**
  - Location of each variable according to alignment rules
    - gaps (padding) inserted as needed
    - also within composite data structures (structs, arrays, unions)
  - Compiler knows the target architecture
  - Generates a layout that is compatible with the architectures ABI
  - As a programmer, we don’t need to worry too much about it
  - However, once we go to the assembly level, the layout becomes relevant

Data Alignment of Variables

```
#include <stdio.h>

int         i  = 1;
char        c  = 1;
…

void info(char *v, size_t s, void *adr, void *prev, size_t ps)
{
  long pad = prev ? (long)adr - (long)prev - ps: 0;

  printf("%-15s %2zu %16p  %ld\n", v, s, adr, pad);
}

int main(void)
{
  printf("Alignment of global variables\n\n");
  printf("VARIABLE       SIZE  ADDRESS         PADDING\n");
  info("int i",           sizeof(i),   &i,  NULL, 0);
  info("char c",          sizeof(c),   &c,  &i,   sizeof(i));
  …
  info("long l",          sizeof(l),   &l,  &ld,  sizeof(ld));
  printf("\n\n");

  foo();

  return 0;
}
```

varalign.c

Data Alignment of Variables

```
$ gcc –m64 –o varalign64 varalign.c
$ ./varalign64
Alignment of global variables

VARIABLE       SIZE  ADDRESS          PADDING
int i            4   0x56183b8ac040   0
char c           1   0x56183b8ac044   0
short s          2   0x56183b8ac046   1
long long ll     8   0x56183b8ac048   0
char c2          1   0x56183b8ac050   0
char c3          1   0x56183b8ac051   0
float f          4   0x56183b8ac054   2
char c4          1   0x56183b8ac058   0
double d         8   0x56183b8ac060   7
long double ld  16   0x56183b8ac070   8
long l           8   0x56183b8ac080   0

…
$
```

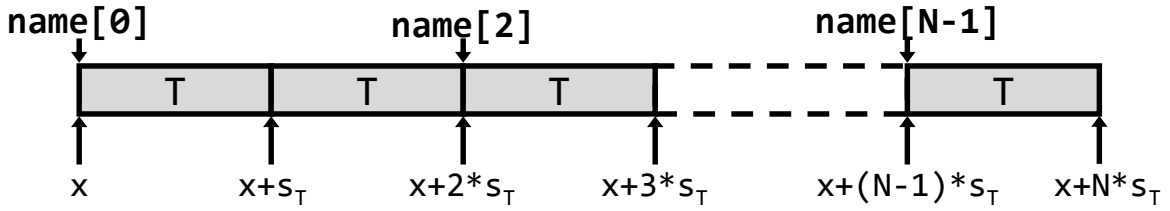
Data Alignment of Arrays

- **Declaration**

  `<T> name[<N>]`

- **Size**
  - one element: s_T = `sizeof(T)`
  - entire array: $s_A = N * s_T$

- **Memory layout**



- **Address of i-th element**

  $adr_i = name + i*s_T$

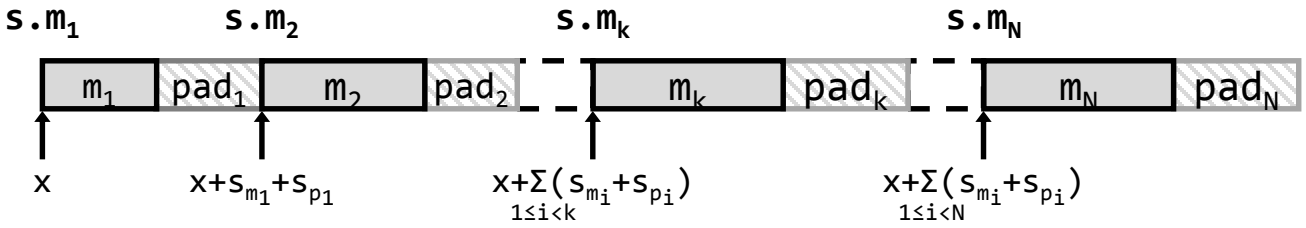
- **Alignment**
  - array alignment = alignment of base type T

Structure Data Alignment

- **Declaration**

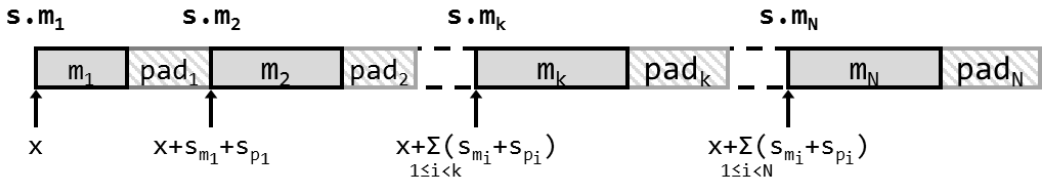
```
struct name {
  <T1> <m1>;
  <T2> <m2>;
  …
  <TN> <mN>;
};
```

- **Memory layout**
  - consecutive memory region containing all members m_i in-order, non-overlapping, and properly aligned



- **Alignment**
  - struct alignment = maximum alignment requirement of any of its members
  - member alignment = alignment requirement of member type
    - padding: “holes” in the memory layout to maintain alignment requirements (denoted pad_i above)

Structures



- **Address of k-th member**
  - start of struct plus sum of sizes of all 1..k-1 members and paddings

$$adr_{m_k} = x + \sum_{1 \le i < k} (s_{m_i} + s_{p_i})$$

- **Size of struct**

$$s_s = \sum_{1 \le i \le N} (s_{m_i} + s_{p_i})$$

  - the last padding (pad_N) is chosen such that the size of the struct is the next multiple of the biggest alignment requirement of any of its members
    - this comes in handy when declaring arrays of structs (all elements of the array will be automatically aligned)

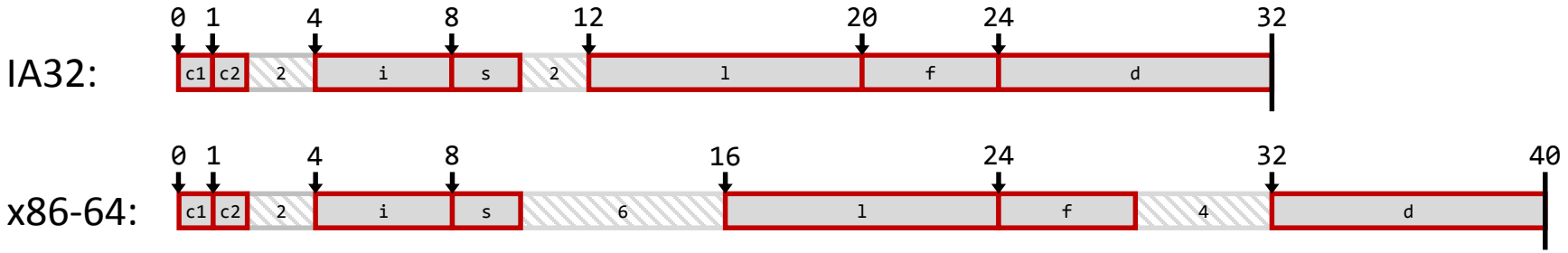
Example: Structure Layout & Alignment

Structure layout & alignment

```
struct s1 {
  char      c1;
  char      c2;
  int       i;
  short     s;
  long long l;
  float     f;
  double    d;
};
```

struct1.c

	IA32		x86-64	
	size	align	size	align
	32	4	40	8
c1	1	1	1	1
c2	1	1	1	1
i	4	4	4	4
s	2	2	2	2
l	8	4	8	8
f	4	4	4	4
d	8	4	8	8



Example: Structure Layout & Alignment

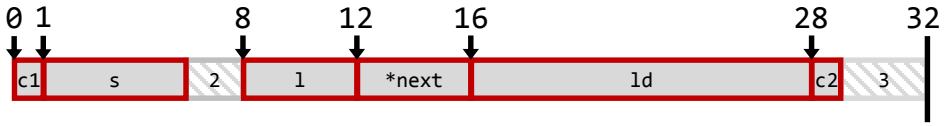
Structure layout & alignment

```
struct s2 {
  char        c1;
  char        s[5];
  long        l;
  struct s2   *next;
  long double ld;
  char        c2;
};
```

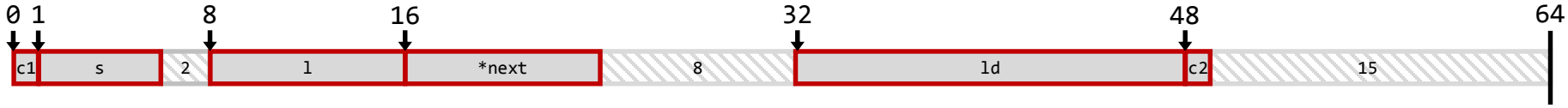
struct2.c

	IA32		x86-64	
	size	align	size	align
struct s2	32	4	64	16
char c1;	1	1	1	1
char s[5];	5	1	5	1
long l;	4	4	8	8
struct s2 *next;	4	4	8	8
long double ld;	12	4	16	16
char c2;	1	1	1	1

IA32:



x86-64:



Example: Size & Alignment of Structures

Finding out about structure offsets and paddings on your machine

```
#include <stdio.h>

struct s2 {
  char        c1;
  char        s[5];
  long        l;
  struct s2   *next;
  long double ld;
  char        c2;
};

#define SIZE(t) sizeof(t)
#define OFS(s,m) ((unsigned long)&s.m - (unsigned long)&s)
#define PAD(s,m,mn) (OFS(s,mn) - OFS(s,m) - sizeof(s.m))

void main(void)
{
  struct s2 s;

  printf("                    OFS  SIZE   PAD\n");
  printf("struct s2 {                     %2zu\n", SIZE(s));
  printf("  char        c1;       %2lu    %2zu    %2lu\n", OFS(s,c1),  SIZE(s.c1),  PAD(s,c1,  s));
  printf("  char        s[5];     %2lu    %2zu    %2lu\n", OFS(s,s),   SIZE(s.s),   PAD(s,s,   l));
  printf("  long        l;        %2lu    %2zu    %2lu\n", OFS(s,l),   SIZE(s.l),   PAD(s,l,   next));
  printf("  struct s2   *next;    %2lu    %2zu    %2lu\n", OFS(s,next),SIZE(s.next),PAD(s,next,ld));
  printf("  long double ld;       %2lu    %2zu    %2lu\n", OFS(s,ld),  SIZE(s.ld),  PAD(s,ld,  c2));
  printf("  char        c2;       %2lu    %2zu    %2lu\n", OFS(s,c2),  SIZE(s.c2),
                                                                       SIZE(s)-OFS(s,c2)-SIZE(s.c2));
  printf("}\n");
}
```

struct2.c

Example: Size & Alignment of Structures

Finding out about structure offsets and paddings on your machine

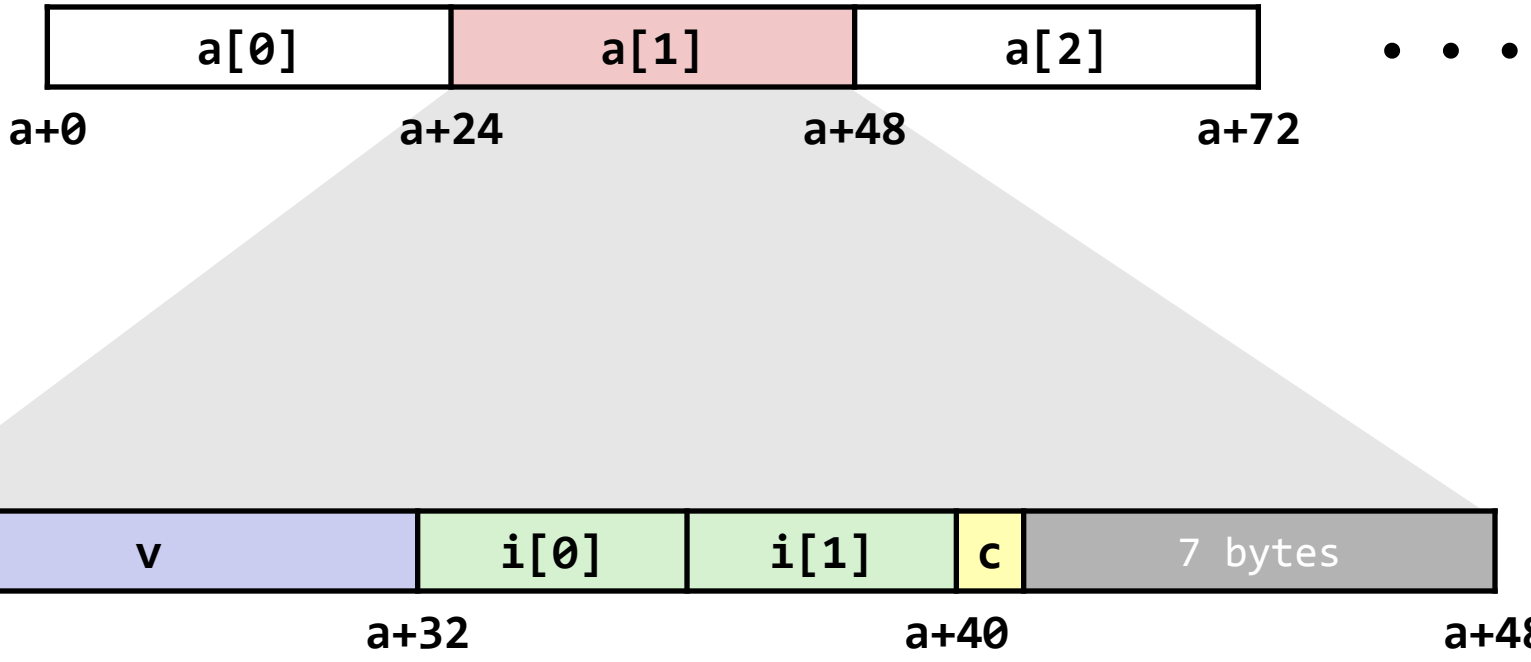
```
$ gcc -m64 -o struct2.64 struct2.c
$ ./struct2.64
                          OFS  SIZE   PAD
struct s2 {                      64
  char        c1;           0     1     0
  char        s[5];         1     5     2
  long        l;            8     8     0
  struct s2   *next;       16     8     8
  long double ld;          32    16     0
  char        c2;          48     1    15
}
$
```

```
$ gcc -m32 -o struct2.32 struct2.c
$ ./struct2.32
                          OFS  SIZE   PAD
struct s2 {                      32
  char        c1;           0     1     0
  char        s[5];         1     5     2
  long        l;            8     4     0
  struct s2   *next;       12     4     0
  long double ld;          16    12     0
  char        c2;          28     1     3
}
$
```

Example: Arrays of Structures

- Overall structure length multiple of K
- Satisfy alignment requirement for every element

```
struct S2 {
  double v;
  int i[2];
  char c;
} a[10];
```



Example: Avoiding Unnecessary Padding

- Reorder members, putting those with the strongest alignment requirements first

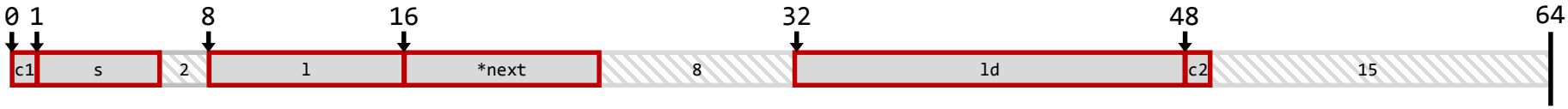
```
struct s2 {
  char        c1;
  char        s[5];
  long        l;
  struct s2   *next;
  long double ld;
  char        c2;
};
```
struct2.c

x86-64	
size	align
64	16
1	1
5	1
8	8
8	8
16	16
1	1

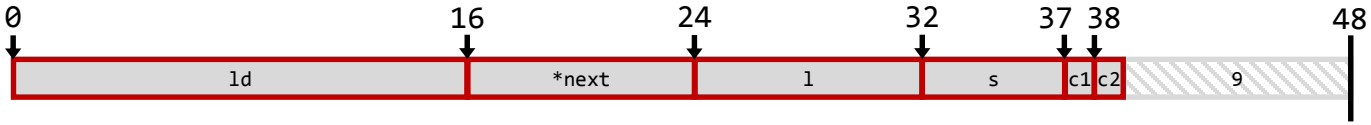
```
struct s2opt {
  long double ld;
  struct s2   *next;
  long        l;
  char        s[5];
  char        c1;
  char        c2;
};
```
struct2opt.c

x86-64	
size	align
48	16
16	16
8	8
8	8
5	1
1	1
1	1

struct s2 on x86-64:



struct s2opt on x86-64:

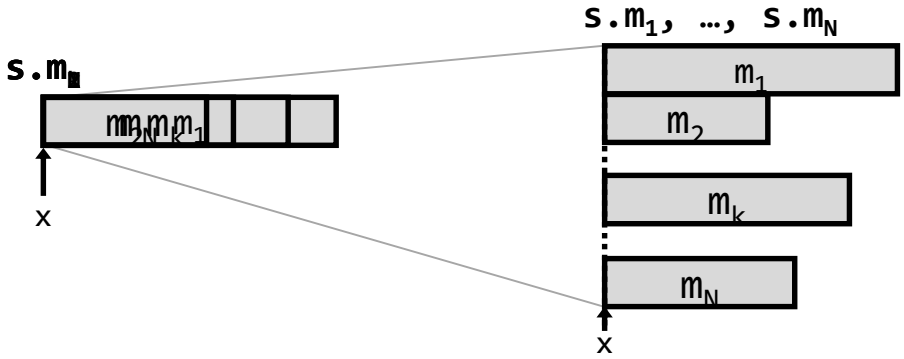


Unions

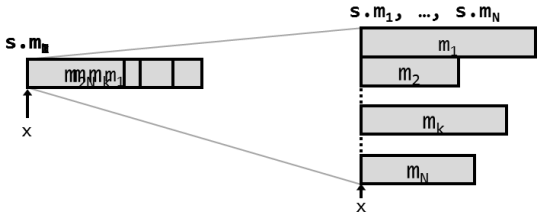
- **Declaration**

```
union name {
  <T1> <m1>;
  <T2> <m2>;
  …
  <TN> <mN>;
};
```

- **Memory layout**
  - consecutive memory region containing all members m_i, and properly aligned
  - all members *are located at offset 0 and overlap in memory*



Unions



- **Alignment**
  - union alignment = maximum alignment requirement of any of its members

- **Address of k-th member**
  - start of union

$$adr_{m_k} = x$$

- **Size of union**

$$s_u = \max_{1 \le i \le N}(s_{m_i})$$

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

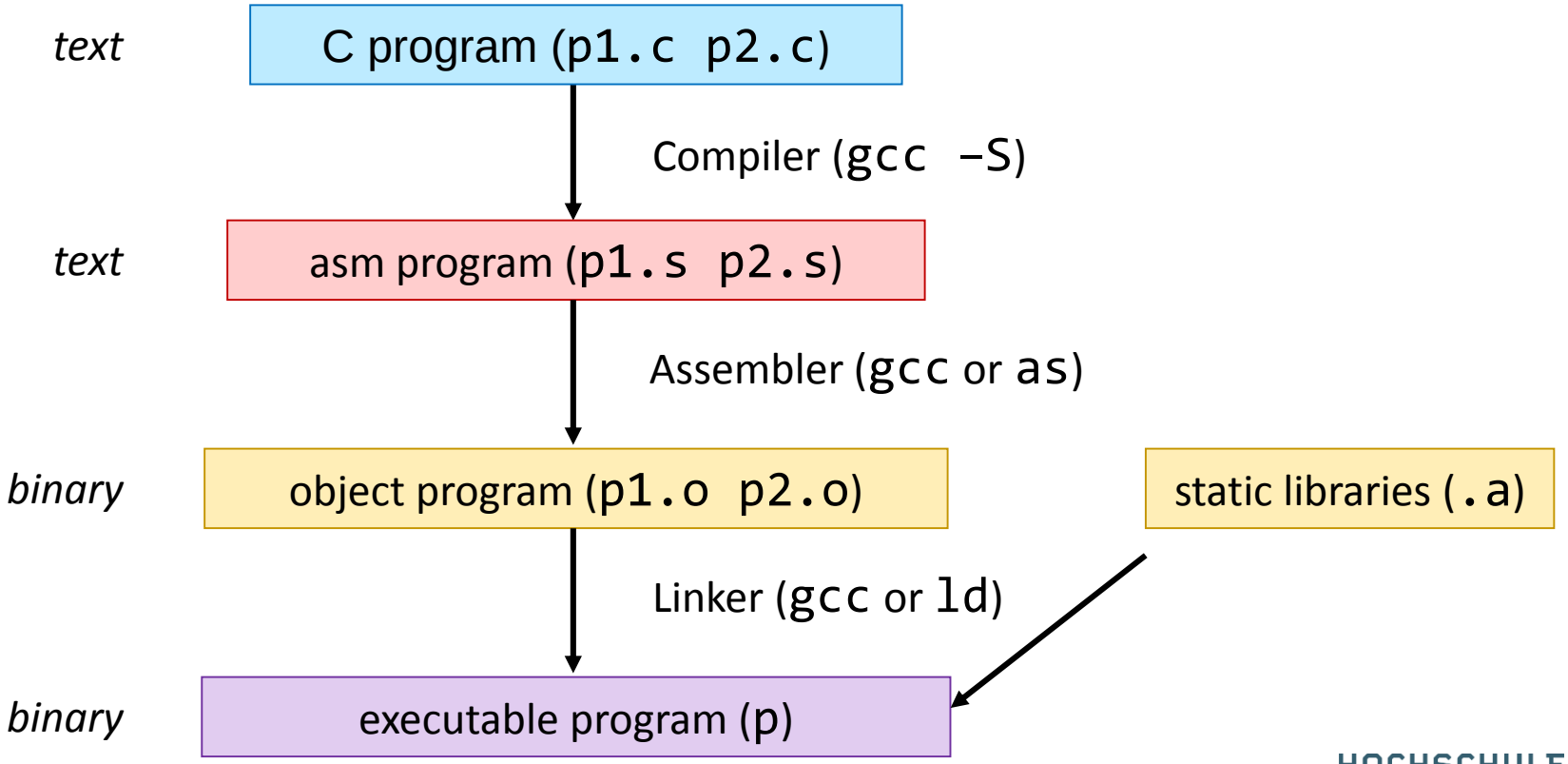
```
sum:
.LFB0:
    .cfi_startproc
    leaq    (%rdi,%rsi), %eax
    ret
    .cfi_endproc
```

The x86 Instruction Set Architecture:

First Steps

From C to Machine Code

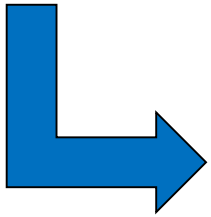
- Code in files **`p1.c p2.c`**
- Compile with command: **`gcc -O p1.c p2.c -o p`**
  - Use optimizations (**`-O`**)
  - Put resulting binary in file **`p`**



Compiling To x86_64 Assembly

C Code

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



```
$ gcc -m64 -O -S sum.c
```

produces the file `sum.s`:

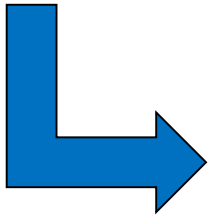
```
sum:
.LFB0:
    .cfi_startproc
    leaq    (%rdi,%rsi), %eax
    ret
    .cfi_endproc
```

→ code generated by `gcc -m64 -O0 -S sum.c`?

Compiling To IA32 Assembly

C Code

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



$ gcc **-m32** -O -S sum.c

produces the file `sum.s`:

compiler: gcc-4.9.4

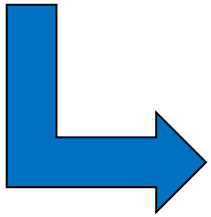
```
sum:
.LFB0:
    .cfi_startproc
    movl    8(%esp), %eax
    addl    4(%esp), %eax
    ret
    .cfi_endproc
```

→ code generated by `gcc -m32 -O0 -S sum.c`?

Compiling To IA32 Assembly – GCC Version

C Code

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



$ gcc **-m32** -O -S sum.c

produces the file `sum.s`:

compiler: gcc-4.4.6

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

Object Code

machine code for sum (32-bit)

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

- Total of 11 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

Machine Instruction Example – x86_64

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

```
leaq      (%rdi,%rsi), %eax
```

Similar to expression:

`t = x+y`

```
0x00400508:  8d 04 37
```

- C Code: add two signed integers
- Assembly
  - lea is an instruction to compute memory addresses, but can be “abused” as a 3-operand add
    - “quad” words in GCC parlance
    - same instruction whether signed or unsigned
  - Operands:

    x: Register **%rdi**

    y: Register **%rsi**

    t: Register **%eax**

    – return function value in **%eax**
- Object Code
  - 3-byte instruction
  - Stored at address `0x00400508`

Machine Instruction Example – IA32

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

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

Similar to expression:

```
x += y
```

More precisely:

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

```
0x0804809a:  03 45 08
```

- C Code: add two signed integers
- Assembly
  - Add two 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 **0x0804809a**

Disassembling Object Code

```
$ gcc –m64 –c –O sum.c
$ objdump -d sum.o
```

```
0000000000000000 <sum>:
   0:  8d 04 37  leq    (%rdi,%rsi,1), %eax
   3:  c3        retq
```

- Disassembler
  - Useful tool for examining object code
  - Analyzes bit pattern of series of instructions
  - Produces *approximate* rendition of assembly code
  - Can be run on executable files and relocatable object files (`.o`)

Disassembling Object Code – IA32

```
$ gcc –m32 –c –O sum.c
$ objdump -d sum.o
```

```
00000000 <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:  5d          pop    %ebp
   a:  c3          ret
```

Alternate Disassembly using gdb

- Within gdb Debugger

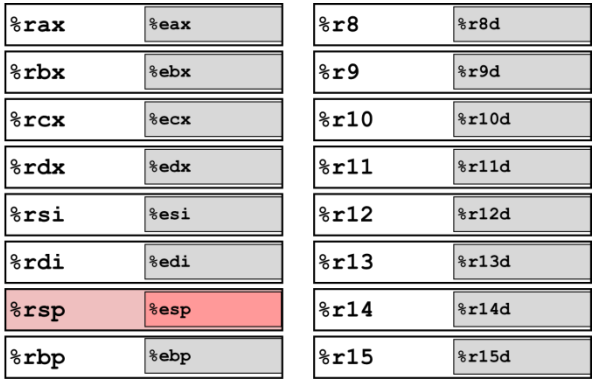
```
$ gcc –m32 –O –o p main.c sum.c
$ gdb p
```

```
(gdb) disassemble sum
Dump of assembler code for function sum:
   0x08048394 <+0>:   push   %ebp
   0x08048395 <+1>:   mov    %esp,%ebp
   0x08048397 <+3>:   mov    0xc(%ebp),%eax
   0x0804839a <+6>:   add    0x8(%ebp),%eax
   0x0804839d <+9>:   pop    %ebp
   0x0804839e <+10>:  ret
End of assembler dump.
(gdb) x/11xb sum
0x8048394 <sum>:      0x55   0x89   0xe5   0x8b   0x45   …
0x804839c <sum+8>:    0x08   0x5d   0xc3
(gdb)
```

What Can be Disassembled?

```
% objdump -d WINWORD.EXE

WINWORD.EXE:   file format pei-i386

No symbols in "WINWORD.EXE".
Disassembly of section .text:

30001000 <.text>:
30001000:  55              push   %ebp
30001001:  8b ec           mov    %esp,%ebp
30001003:  6a ff           push   $0xffffffff
30001005:  68 90 10 00 30  push   $0x30001090
3000100a:  68 91 dc 4c 30  push   $0x304cdc91
```

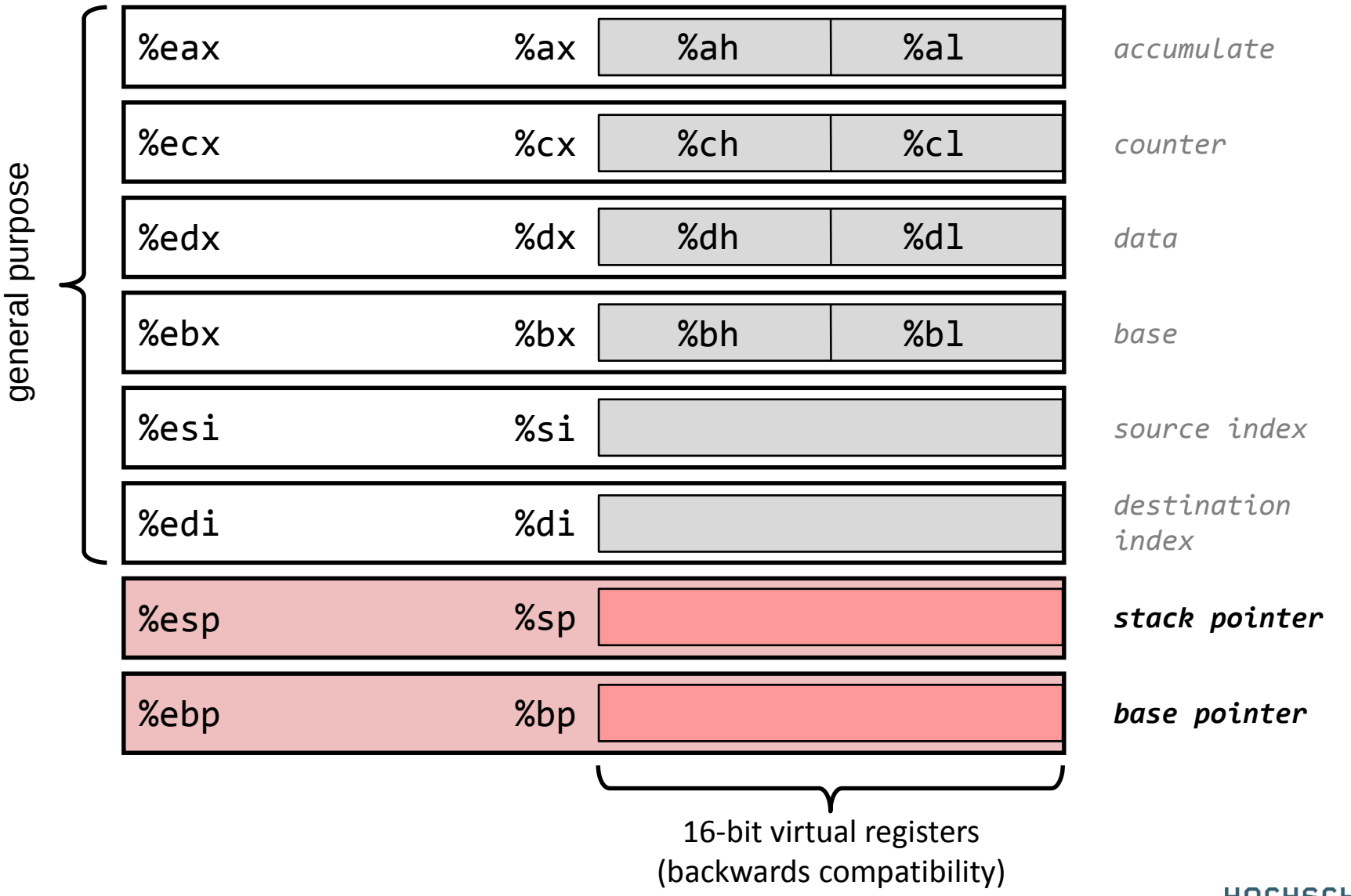
- Anything that can be interpreted as executable code
  (not everything makes sense, of course)
- Disassembler examines bytes and reconstructs assembly source



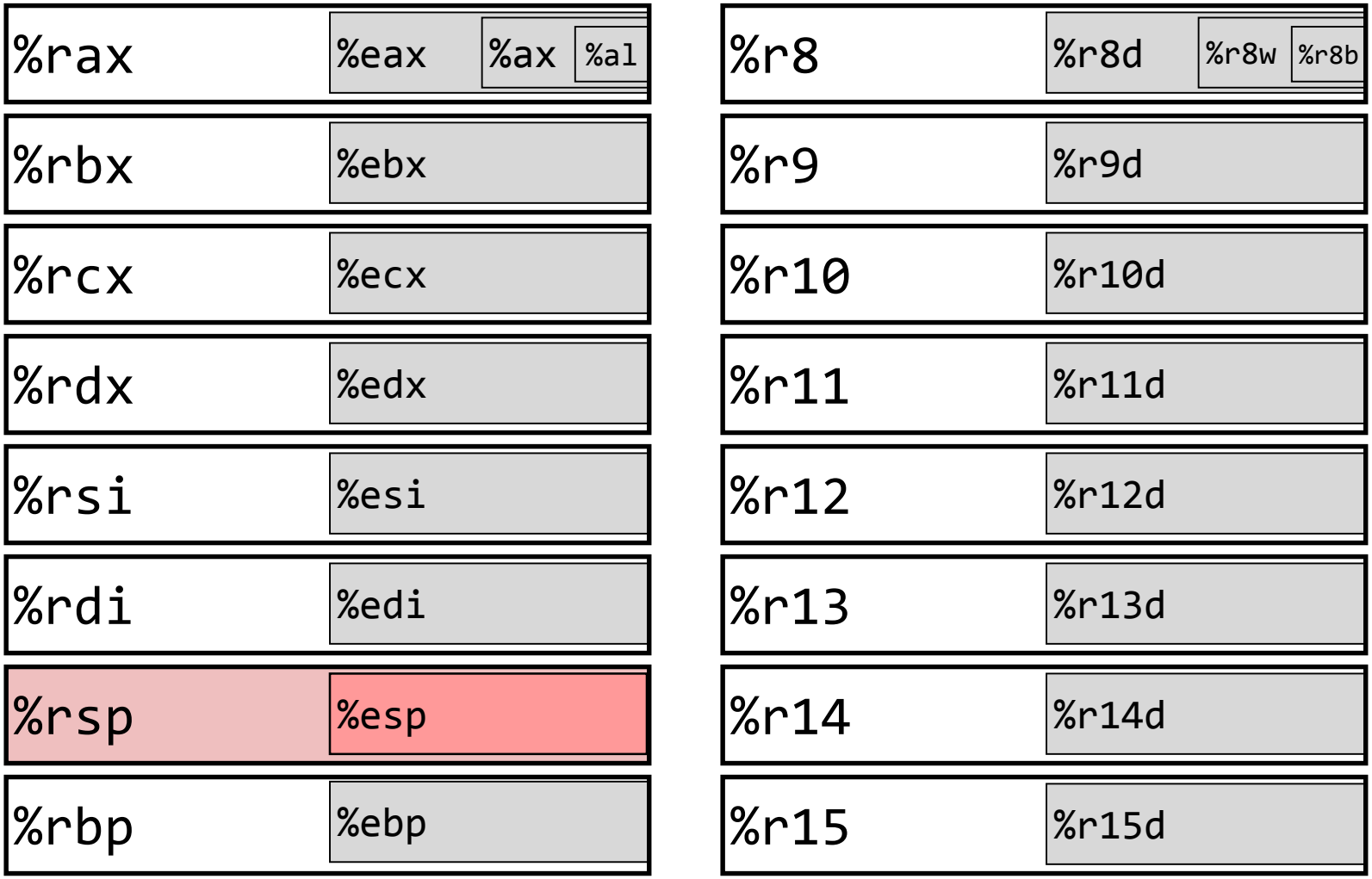
The x86 Instruction Set Architecture:

Accessing Information

IA32 Integer Registers

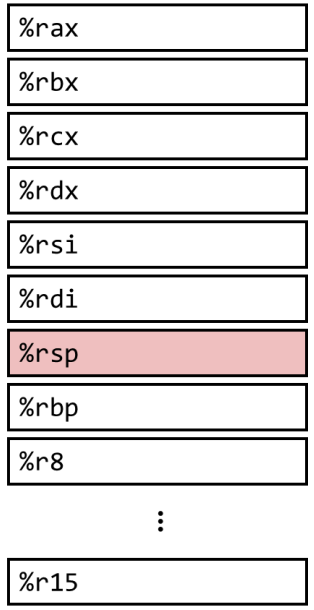


x86-64 Integer Registers



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

Moving Data



- Moving Data

  `movx Source, Dest`

- Operand Types
  - **Immediate:** Constant integer data
    - like C constants with **'$'** prefix
    - encoded with 1, 2, 4, or 8 bytes
    - examples: **`$0x400, $-533`**
  - **Register:** One of the 16 64-bit integer registers (or their 32/16/8-bit parts)
    - examples: **`%rax, %edx, %bx, %cl`**
    - reserved: **`%rsp/esp`** (plus **`%ebp`** on IA32)
    - other registers may have special uses for particular instructions
  - ***Memory:*** 8/4/2/1 consecutive bytes of memory at address given by register
    - number of bytes depends on operand size specifier
    - simplest example: **`(%rax)`**
    - various other "address modes"

movq Operand Combinations

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

Memory-memory transfers with a single instruction are not supported

Simple Memory Addressing Modes

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

```
movl (%rcx),%eax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - register R specifies start of memory region
  - constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Using Simple Addressing Modes – IA32

```
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
```

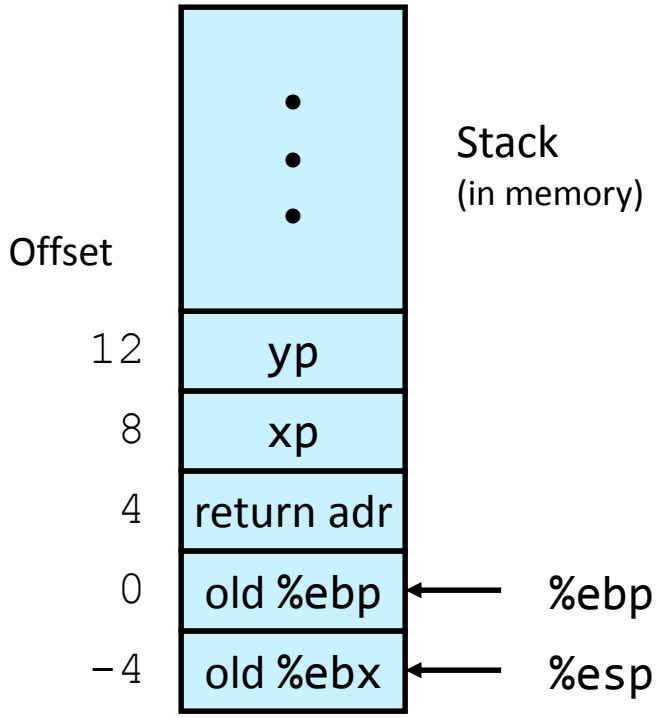
Set Up

Body

Finish

Understanding Swap – IA32

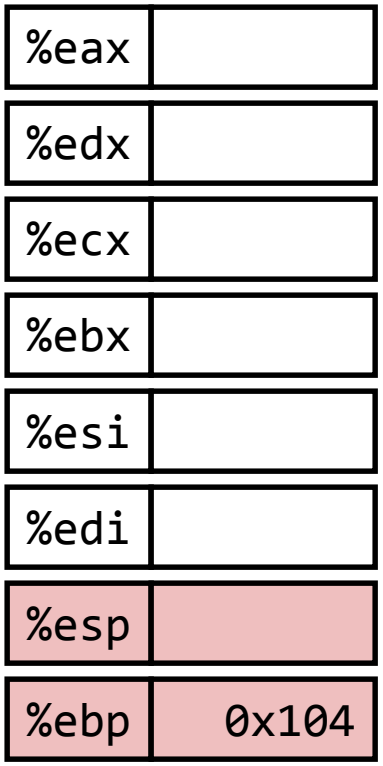
```
void swap(int *xp, int *yp)
{
  int t0 = *xp;
  int t1 = *yp;
  *xp = t1;
  *yp = t0;
}
```

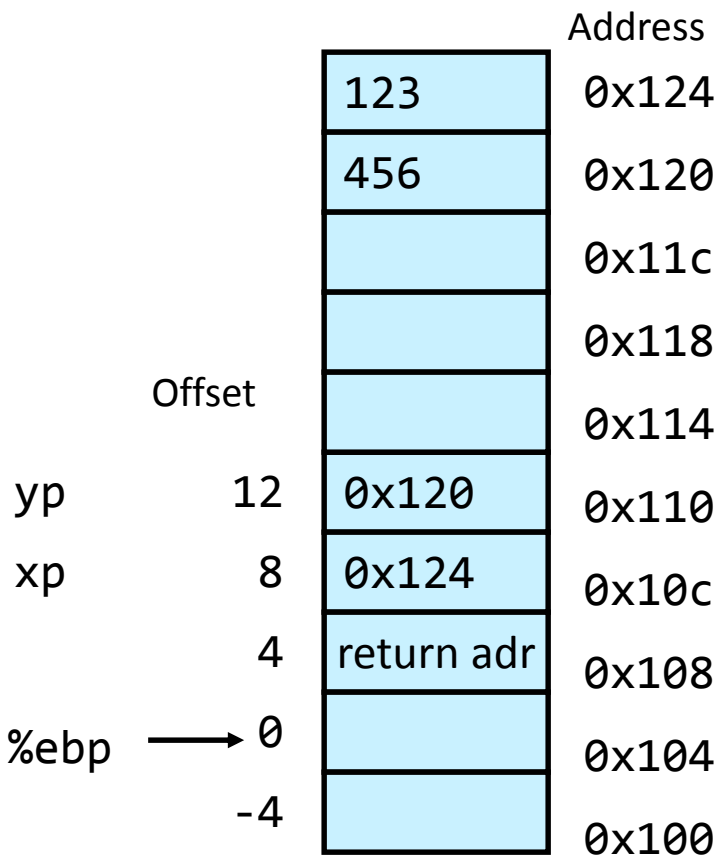


Register	Value
%edx	xp
%ecx	yp
%ebx	t0
%eax	t1

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

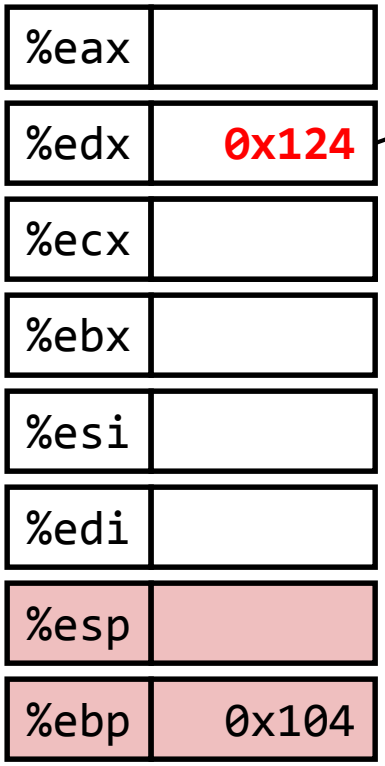
Understanding Swap – IA32

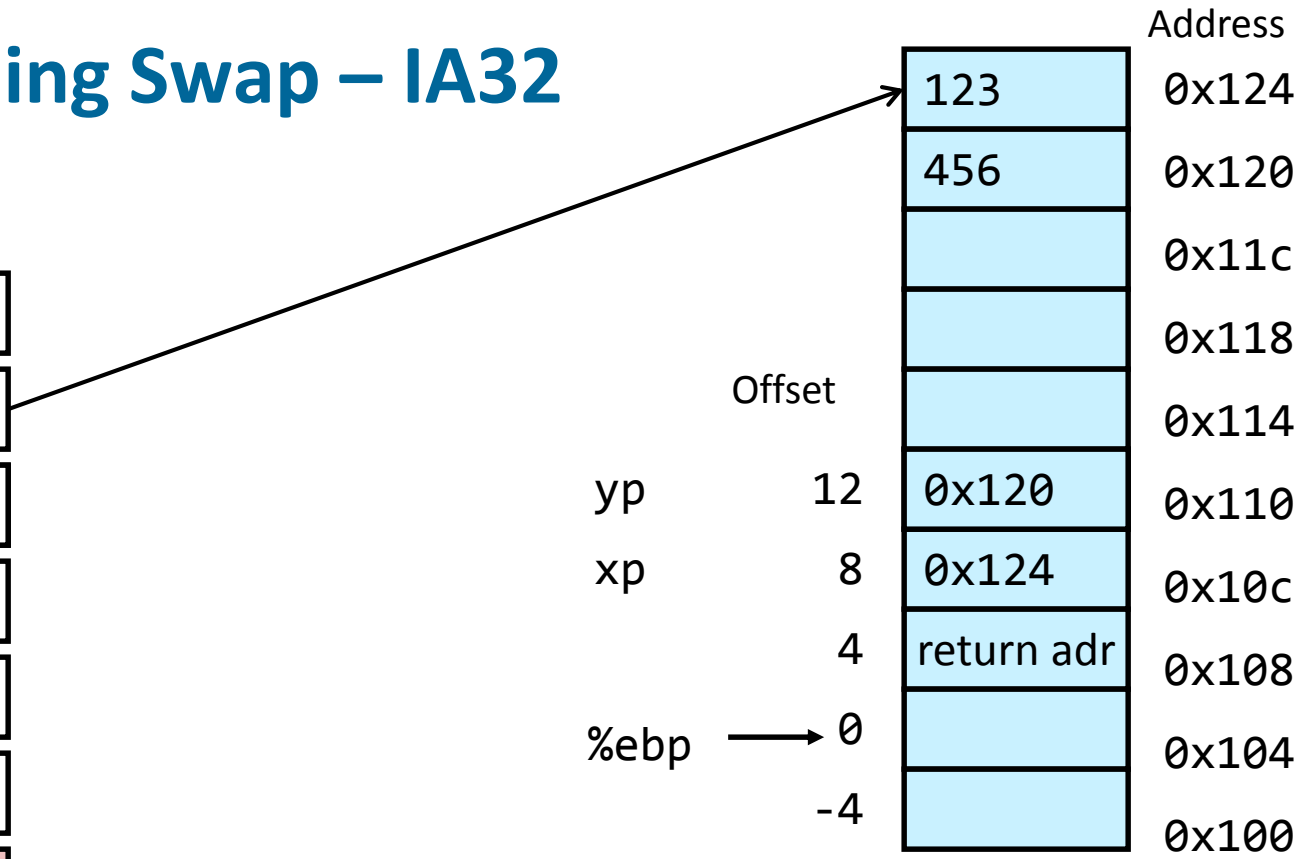





```
movl   8(%ebp), %edx     # edx = xp
movl   12(%ebp), %ecx    # ecx = yp
movl   (%edx), %ebx      # ebx = *xp (t0)
movl   (%ecx), %eax      # eax = *yp (t1)
movl   %eax, (%edx)      # *xp = t1
movl   %ebx, (%ecx)      # *yp = t0
```

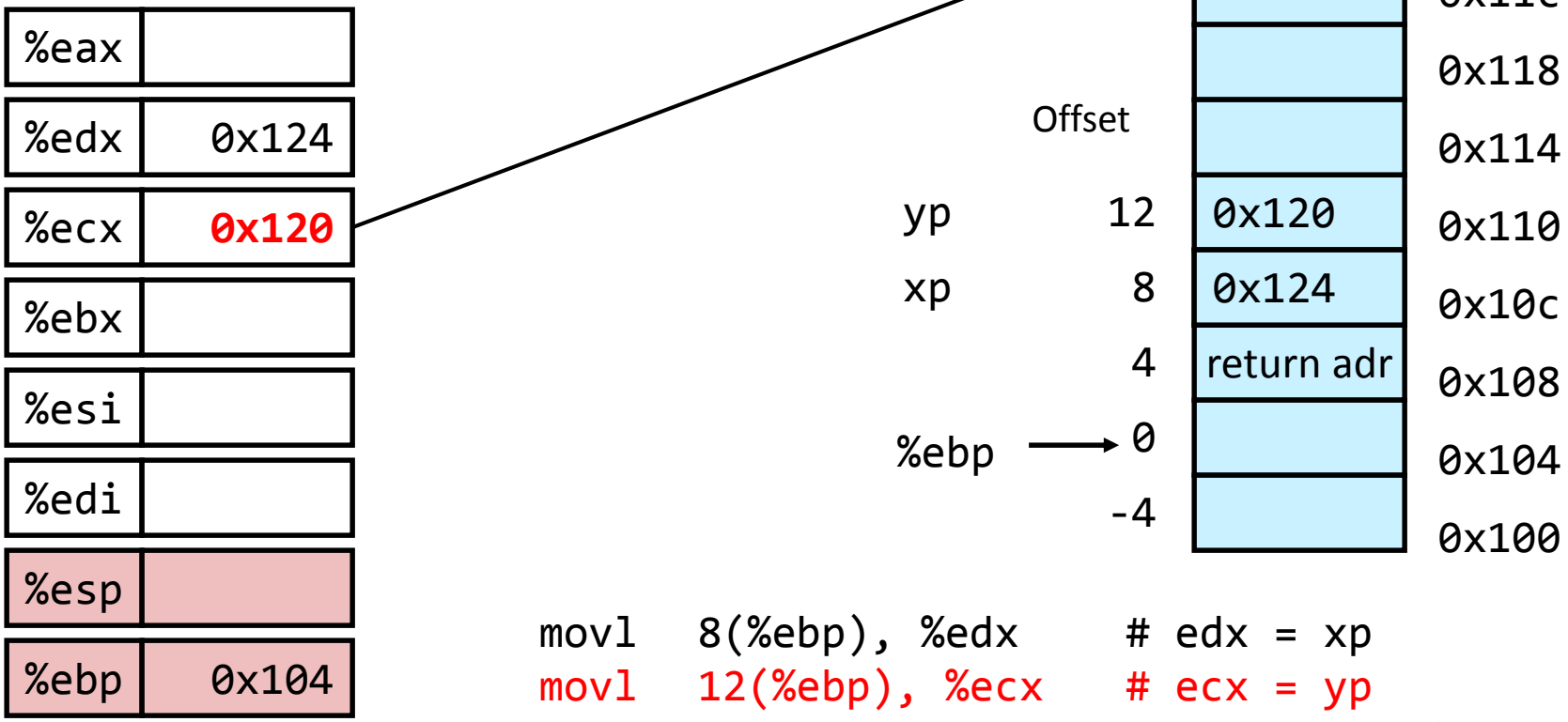
Understanding Swap – IA32






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

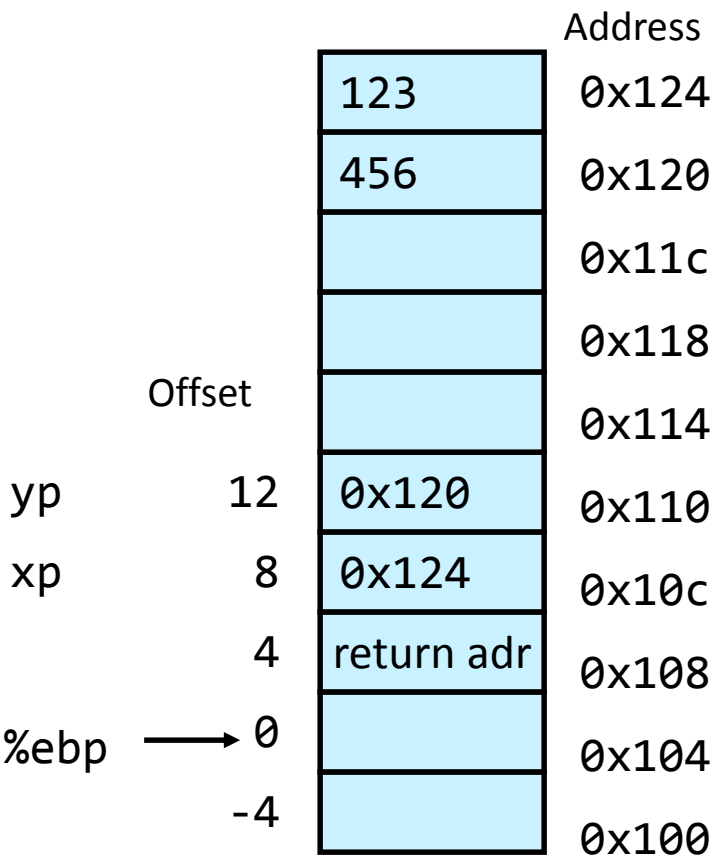
Understanding Swap – IA32




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

Understanding Swap – IA32

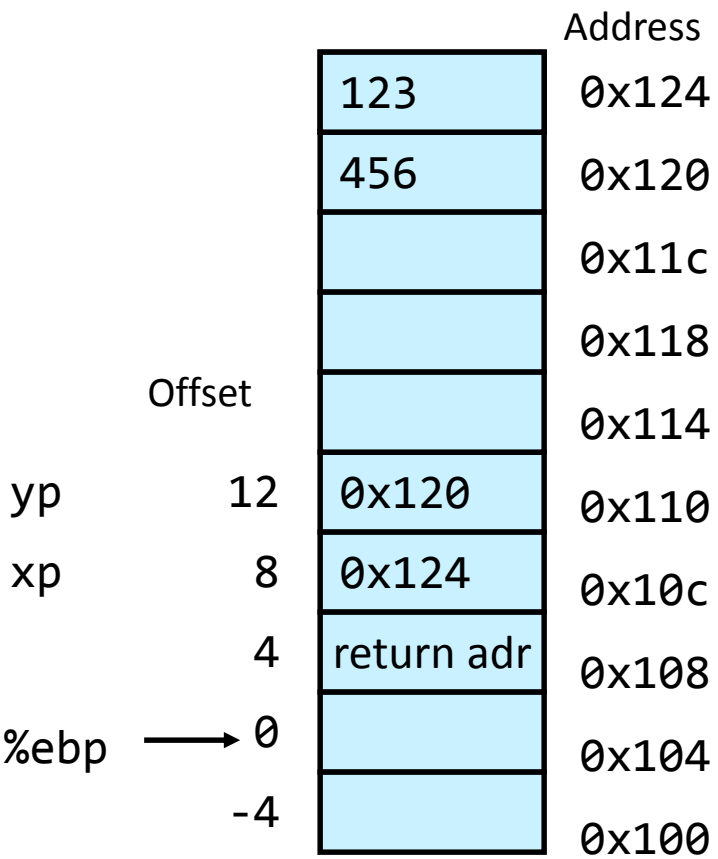
%eax	
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




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

Understanding Swap – IA32

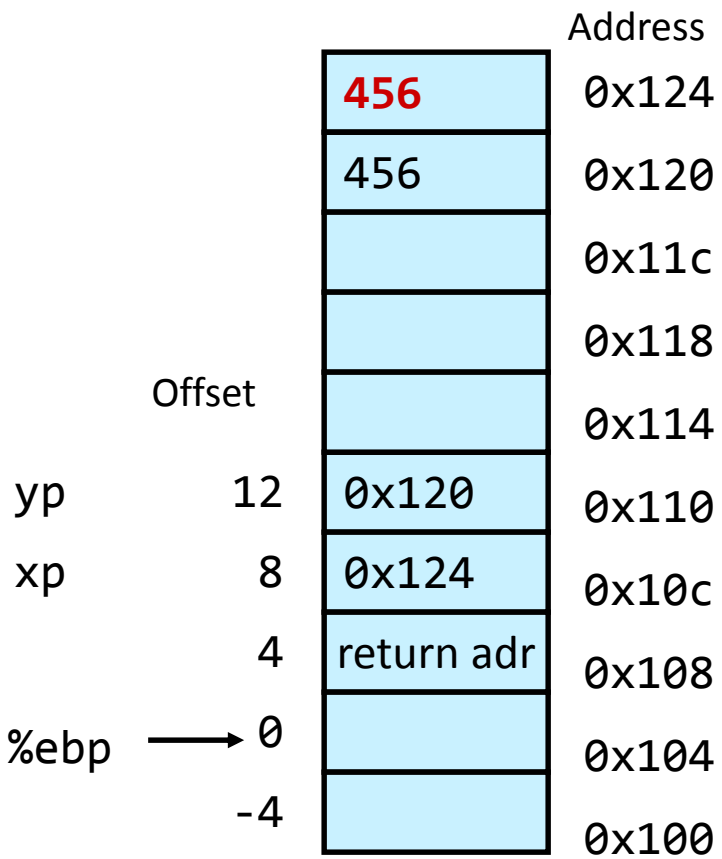
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




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

Understanding Swap – IA32

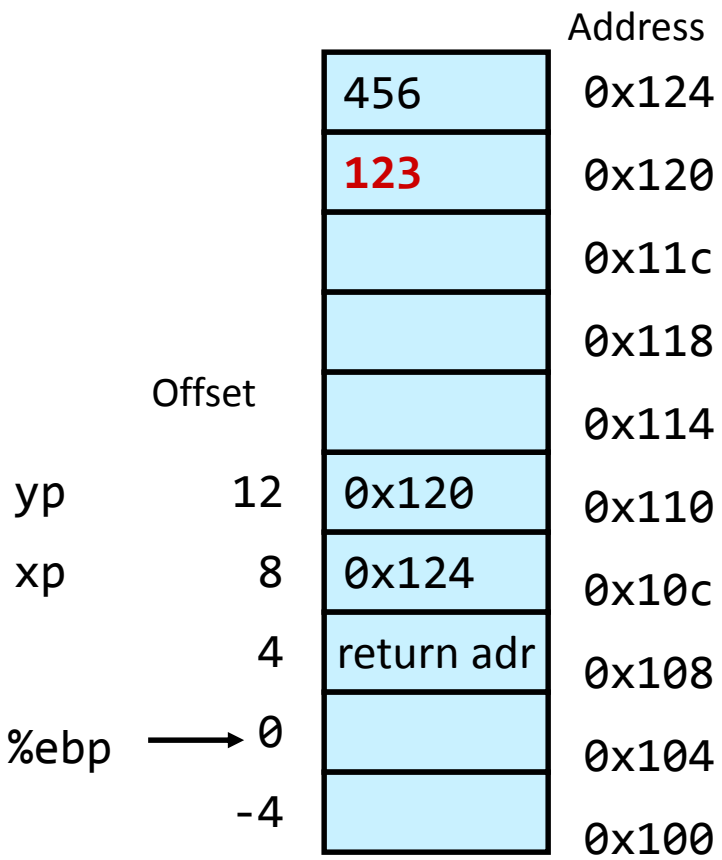
%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




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

Understanding Swap – IA32

%eax	456
%edx	0x124
%ecx	0x120
%ebx	123
%esi	
%edi	
%esp	
%ebp	0x104




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

Complete Memory Addressing Modes

- General Form:

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

  - D: Constant “displacement” (represented with 1, 2, or 4 bytes)
  - Rb: Base register: Any integer register
  - Ri: Index register: Any, except for **`%rsp/%esp`**
  - S: Scale: 1, 2, 4, or 8 (x86_64 and IA32) (btw: ***why these numbers?***)

Complete Memory Addressing Modes

- General Form:

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

- Variations

Form	Operand value	Name
D	M[D]	Absolute
(Rb)	M[R[Rb]]	Indirect
D(Rb)	M[R[Rb] + D]	Base + displacement
(Rb, Ri)	M[R[Rb] + R[Ri]]	Indexed
D(Rb, Ri)	M[R[Rb] + R[Ri] + D]	Indexed
(, Ri, s)	M[R[Ri]*s]	Scaled indexed
D(, Ri, s)	M[R[Ri]*s + D]	Scaled indexed
(Rb, Ri, s)	M[R[Rb] + R[Ri]*s]	Scaled indexed
D(Rb, Ri, s)	M[R[Rb] + R[Ri]*s + D]	Scaled indexed

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

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

Address Computation Instruction `lea`

- **`lea = load effective address`**
  - compute the address of a address mode expression *without* accessing the memory

- **`leaq/l Src, Dest`**
  - **`Src`** is address mode expression
  - Set **`Dest`** to address denoted by expression
    - addresses width: x86_64: 8 byte (leaq); IA32: 4 byte (leal)

Address Computation Instruction `lea`

- Uses
  - Computing addresses *without* a memory reference
    - E.g., translation of **`p = &x[i];`**

```
long* addr(long *a)
{
  return &a[5];
}
```

```
addr:
  leaq 40(%rdi), %rax
  ret
```

  - Computing arithmetic expressions of the form x + y*k + z
    - k = 1, 2, 4, or 8

```
long muladd(long x)
{
  return x*5+3;
}
```

```
muladd:
  leaq 3(%rdi,%rdi,4), %rax
  ret
```

Sign/Zero Expansion

- MOV with operands of different sizes

```
void main(void)
{
  char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```

vs

```
void main(void)
{
  unsigned char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```

  - Expected result?

Sign/Zero Expansion

- MOV with operands of different sizes

```
void main(void)
{
  char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```
expand_signed.c

vs

```
void main(void)
{
  unsigned char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```
expand_unsigned.c

  - Disassembly

```
…
movb   $0xff, -0x5(%rbp)
movsbl -0x5(%rbp), %eax
…
```

```
…
movb   $0xff, -0x5(%rbp)
movzbl -0x5(%rbp), %eax
…
```

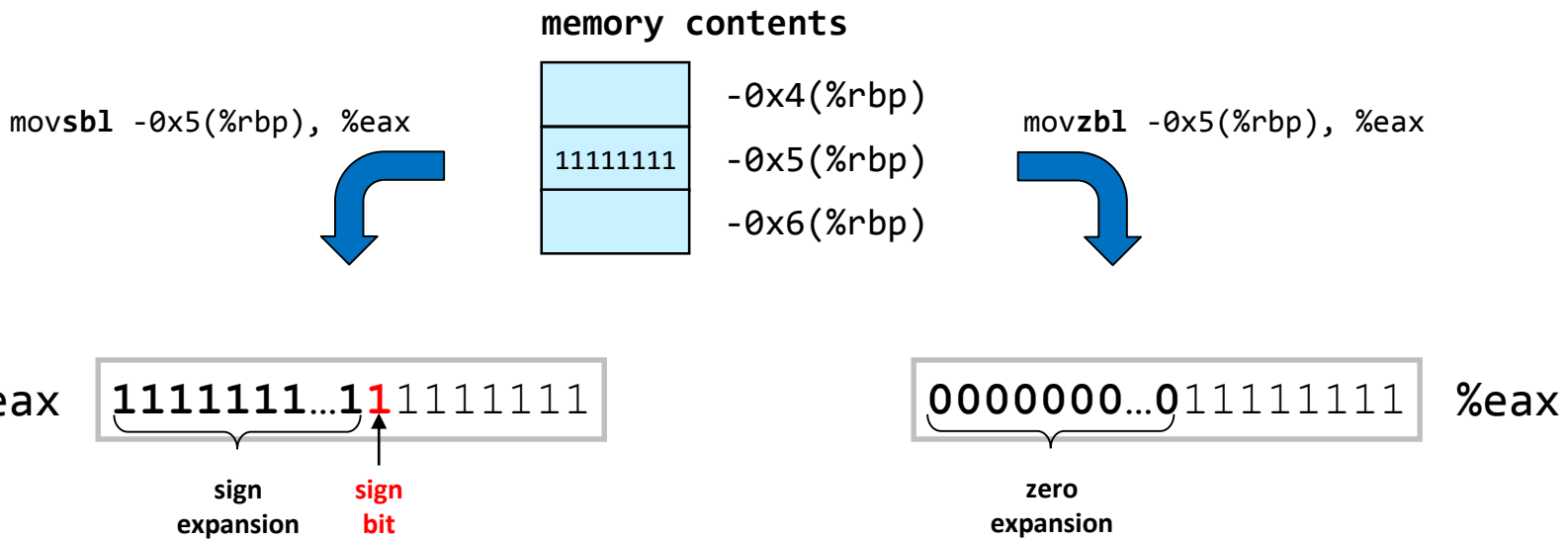
Sign/Zero Expansion

- MOV with operands of different sizes

```
void main(void)
{
  char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```
expand_signed.c

vs

```
void main(void)
{
  unsigned char c = -1;
  int  i = c;

  printf("%d\n", i);
}
```
expand_unsigned.c



Sign/Zero Expansion

- MOV with operands of different sizes
  - `movsbl`: move **s**ign-extand **b**yte to double word(**l**)
  - `movzbl`: move **z**ero-expand **b**yte to double word(**l**)

Instruction		Effect	Description
MOV	**S, D**	**D ← S**	**Move (equal-sized operands)**
`movb`		move byte	
`movw`		move word (16-bit)	
`movl`		move double word (32-bit)	
`movq`		move quad word (64-bit)	
MOVS	**S, D**	**D ← SignExtend(S)**	**Move with sign extension**
`movsb[w,l,q]`		move sign-extended byte to word, double word, quad word	
`movsw[l,q]`		move sign-extended word to double word, quad word	
`movslq`		move sign-extended double word to quad word	
MOVZ	**S, D**	**D ← ZeroExtend(S)**	**Move with zero extension**
`movzb[w,l,q]`		move zero-extended byte to word, double word, quad word	
`movzw[l,q]`		move zero-extended word to double word, quad word	
`movzlq`		move zero-extended double word to quad word	

  - `movs/z`: always size(source operand) < size(destination operand)

Stack Operations: PUSH & POP

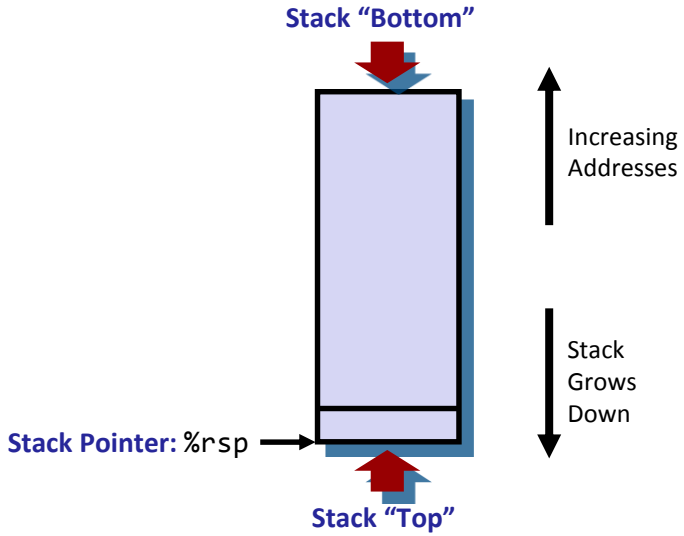
- **`%esp/%rsp`** point to a memory region call the *stack*
- The stack holds temporary data associated with currently active functions
- `push`/`pop` are used to store/retrieve temporary data onto/from the stack

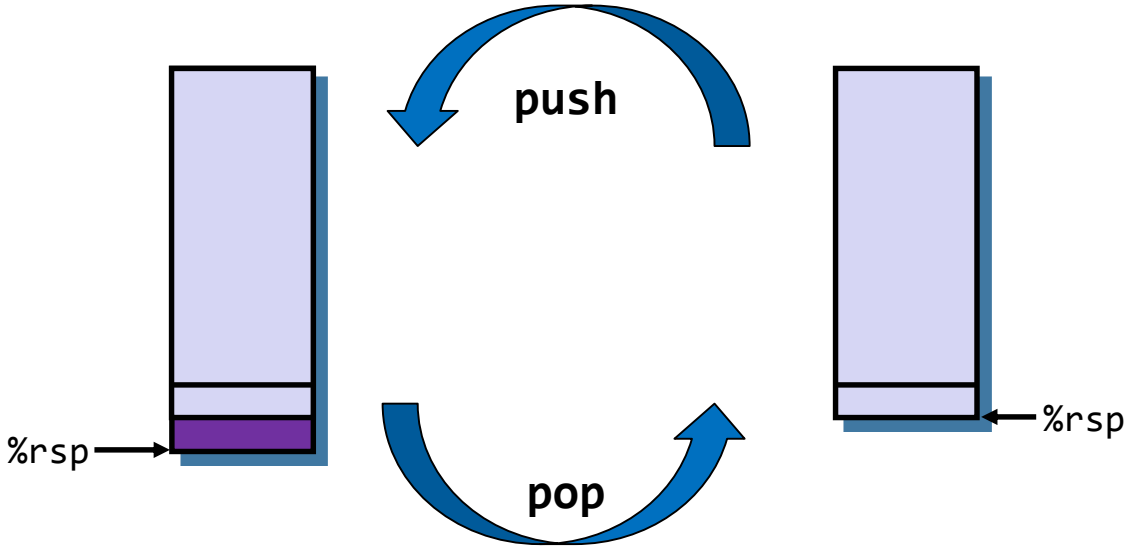
Instruction		Effect	Description
STACK			**Stack operations**
`pushl`	`S`	`R[%esp] ← R[%esp] – 4;` `M[R[%esp]] ← S`	push double word onto stack
`popl`	`D`	`D ← M[R[%esp]];` `R[%esp] ← R[%esp] + 4`	pop double word from stack

- `pushl s` has same effect as "`subl $4, %esp; movl s, (%esp)`"
- `popl d` has same effect as "`movl (%esp), d; addl $4, %esp`"

- size modifiers apply: `push[b,w,l,q]`, `pop[b,w,l,q]`

Stack Operations: PUSH & POP

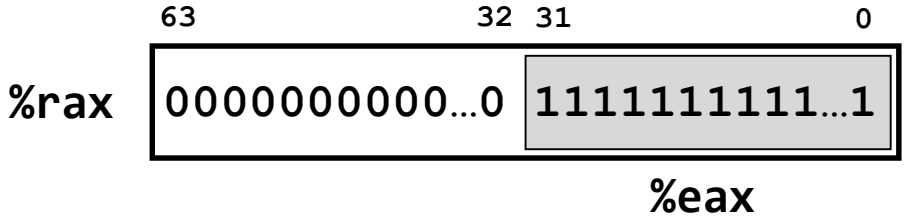




- Maintaining stack discipline: typically
  - push/pop occur in pairs
  - and in reverse order: `push a, push b, push c → pop c, pop b, pop a`

Running IA32 instructions on x86_64

- 32-bit instructions that generate 32-bit results
  - Set higher order bits of destination register to 0
  - Example: **`addl $-1, %eax`**



32-bit code for swap

```
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

64-bit code for swap

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

```
swap:

                                    } Set Up

    movl    (%rdi), %edx
    movl    (%rsi), %eax
    movl    %eax, (%rdi)            } Body
    movl    %edx, (%rsi)

    ret                             } Finish
```

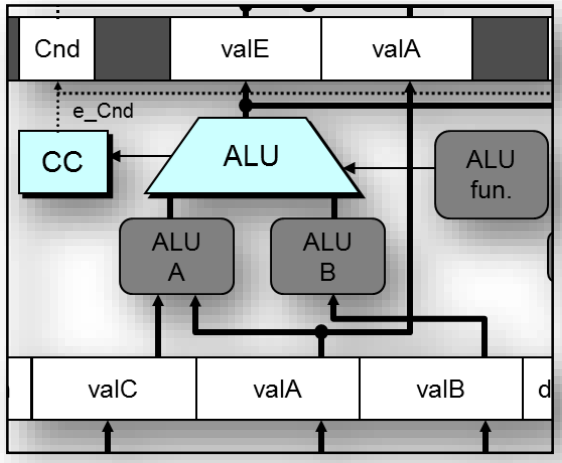
- Operands passed in registers (why is that useful?)
  - First (xp) in `%rdi`, second (yp) in `%rsi`
  - 64-bit pointers
- No stack operations required
- 32-bit data
  - Data held in registers `%eax` and `%edx`
  - `movl` operation

64-bit code for long int swap

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

```
swapl:

                                    } Set Up
movq    (%rdi), %rdx
movq    (%rsi), %rax                } Body
movq    %rax, (%rdi)
movq    %rdx, (%rsi)

ret                                 } Finish
```

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



The x86 Instruction Set Architecture:

Arithmetic and Logical Operations

(Some) Arithmetic and Logical Operations

- Two Operand Instructions

Instruction	Effect	Description
`lea S, D`	D ← &S	load effective address
`add S, D`	D ← D + S	add
`sub S, D`	D ← D - S	subtract
`imul S, D`	D ← D * S	multiply
~~`idiv S, D`~~		does not exist, see `[i]div S` (follows)
`xor S, D`	D ← D ^ S	exclusive-or
`or S, D`	D ← D | S	or
`and S, D`	D ← D & S	and
`sal k, D`	D ← D << k	left shift
`shl k, D`	D ← D << k	left shift (same as sal)
`sar k, D`	D ← D $>>_A$ k	arithmetic right shift
`shr k, D`	D ← D $>>_L$ k	logical right shift

Note: shift amount (k) given as an immediate or in %cl

- No distinction between signed and unsigned int

(Some) Arithmetic and Logical Operations

- Single Operand Instructions

Instruction		Effect	Description
`inc`	D	D ← D + 1	increment
`dec`	D	D ← D - 1	decrement
`neg`	D	D ← -D	negate
`not`	D	D ← ~D	complement

(Some) Arithmetic and Logical Operations

- Special Arithmetic Operations – x86_64

Instruction	Effect	Description
`imulq S`	R[%rdx]:R[%rax] ← S x R[%rax]	signed full multiply
`mulq S`	R[%rdx]:R[%rax] ← S x R[%rax]	unsigned full multiply
`cqto`	R[%rdx]:R[%rax] ← SignExtend(R[%rax])	convert %rax to octa word
`idivq S`	R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S	signed divide
`divq S`	R[%rdx] ← R[%rdx]:R[%rax] mod S; R[%rax] ← R[%rdx]:R[%rax] ÷ S	unsigned divide

  - `R[%rdx]:R[%rax]` viewed as a single 128-bit octa word

(Some) Arithmetic and Logical Operations

- Special Arithmetic Operations – IA32

Instruction	Effect	Description
`imull S`	R[%edx]:R[%eax] ← S x R[%eax]	signed full multiply
`mull S`	R[%edx]:R[%eax] ← S x R[%eax]	unsigned full multiply
`cltd`	R[%edx]:R[%eax] ← SignExtend(R[%eax])	convert %eax to quad word
`idivl S`	R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S	signed divide
`divl S`	R[%edx] ← R[%edx]:R[%eax] mod S; R[%eax] ← R[%edx]:R[%eax] ÷ S	unsigned divide

  - `R[%edx]:R[%eax]` viewed as a single 64-bit quad word

Example: Arithmetic and Logical Operations

- **x86_64**

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0f0f0f0f;
  long t4 = t2 - t3;
  return t4;
}
```

```
arith:

leaq    (%rdx,%rdx,2),%rax
xorq    %rdi,%rsi
andl    $252645135, %esi
salq    $4, %rax
subq    %rsi, %rax

ret
```

body

finish

Understanding arith

- **x86_64: Parameter passing**

x	y	z
%rdi	%rsi	%rdx

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0f0f0f0f;
  long t4 = t2 - t3;
  return t4;
}
```

```
arith:
  leaq    (%rdx,%rdx,2),%rax
  xorq    %rdi,%rsi
  andl    $252645135, %esi
  salq    $4, %rax
  subq    %rsi, %rax
  ret
```

Understanding arith

- **x86_64: Computation**

x	y	z
%rdi	%rsi	%rdx

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0f0f0f0f;
  long t4 = t2 - t3;
  return t4;
}
```

```
arith:
  leaq    (%rdx,%rdx,2),%rax          #
  xorq    %rdi,%rsi                   #
  andl    $252645135, %esi            #
  salq    $4, %rax                    #
  subq    %rsi, %rax                  #
  ret                                 #
```

Understanding arith

- **x86_64: Computation**

x	y	z
%rdi	%rsi	%rdx

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0f0f0f0f;
  long t4 = t2 - t3;
  return t4;
}
```

```
arith:
  leaq    (%rdx,%rdx,2),%rax      # rax = z + 2*z
  xorq    %rdi,%rsi               # rsi = y ^ x
  andl    $252645135, %esi        # esi = esi & 0x0f0f0f0f
  salq    $4, %rax                # rax = rax << 4
  subq    %rsi, %rax              # rax = rax - rsi
  ret                             #
```

Observations about arith

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0f0f0f0f;
  long t4 = t2 - t3;
  return t4;
}
```

Observations

- Parameters passed in registers
- Instruction order differs from C code
- Expressions can require multiple instructions
- Get exact same code when compiling
  z*48 - ((x^y)&0x0f0f0f0f)

```
arith:
  leaq    (%rdx,%rdx,2),%rax
  xorq    %rdi,%rsi
  andl    $252645135, %esi
  salq    $4, %rax
  subq    %rsi, %rax
  ret
```

```
# rax = z + 2*z
# rsi(t1) = y ^ x
# esi(t3) = esi(t1)&0x0f0f0f0f
# rax(t2) = rax(3z) << 4
# rax = rax(t2) - rsi(t3)
# return value = rax
```

Example: Arithmetic and Logical Operations

- **IA32**

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0f0f0f0f;
  long t4 = t2 – t3;
  return t4;
}
```

```
arith:
movl    12(%esp), %eax
movl    8(%esp), %edx
xorl    4(%esp), %edx
leal    (%eax,%eax,2),%eax
andl    $252645135, %edx
sall    $4, %eax
subq    %edx, %eax

ret
```

body

finish

Understanding arith

x	y	z
4(%esp)	8(%esp)	12(%esp)

- **IA32: Parameter passing & computation**

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0f0f0f0f;
  long t4 = t2 - t3;
  return t4;
}
```

```
arith:
  movl    12(%esp), %eax                  #
  movl    8(%esp), %edx                   #
  xorl    4(%esp), %edx                   #
  leal    (%eax,%eax,2),%eax              #
  andl    $252645135, %edx                #
  sall    $4, %eax                        #
  subq    %edx, %eax                      #
  ret                                     #
```

Understanding arith

x	y	z
4(%esp)	8(%esp)	12(%esp)

```
long arith(long x, long y, long z)
{
  long t1 = x ^ y;
  long t2 = z * 48;
  long t3 = t1 & 0x0f0f0f0f;
  long t4 = t2 - t3;
  return t4;
}
```

Observations

- Parameters passed in memory (stack)
- Instruction order differs from C code
- Expressions can require multiple instructions
- Get exact same code when compiling
  z*48 - ((x^y)&0x0f0f0f0f)

```
arith:
  movl   12(%esp), %eax     # eax = z
  movl   8(%esp), %edx      # edx = y
  xorl   4(%esp), %edx      # edx(t1) = edx(y) ^ x
  leal   (%eax,%eax,2),%eax # eax = eax(z) + 2*eax(z)
  andl   $252645135, %edx   # edx(t3) = edx(t1)&0x0f0f0f0f
  sall   $4, %eax           # eax(t2) = eax(3z) << 4
  subq   %edx, %eax         # eax(t4) = eax(t2) - edx(t3)
  ret                       # return value = eax
```

Another Example

- IA32

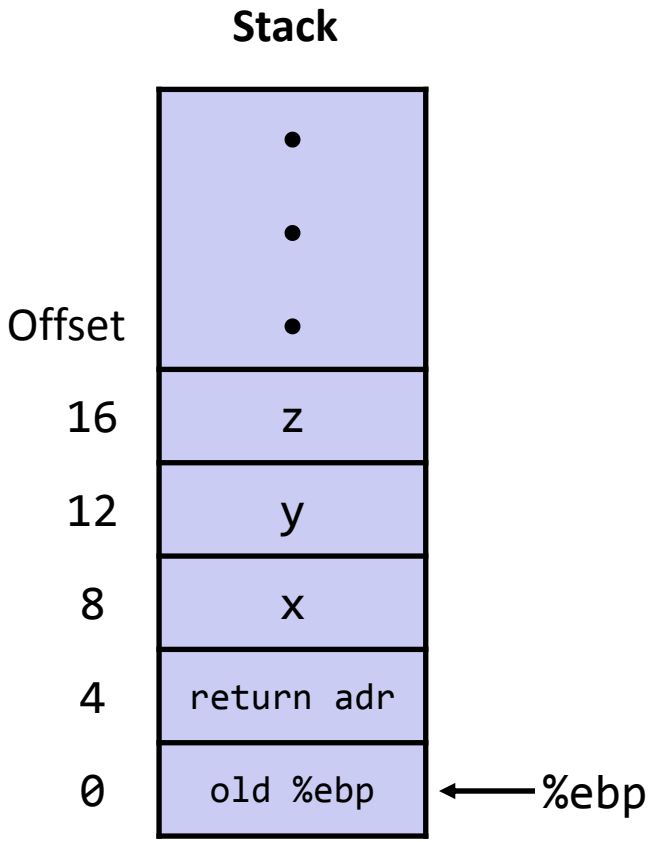
```
long arith2
  (long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
arith2:
   pushl   %ebp                    } setup
   movl    %esp, %ebp

   movl    8(%ebp), %ecx           } body
   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                    } finish
   ret
```

Understanding `arith2`

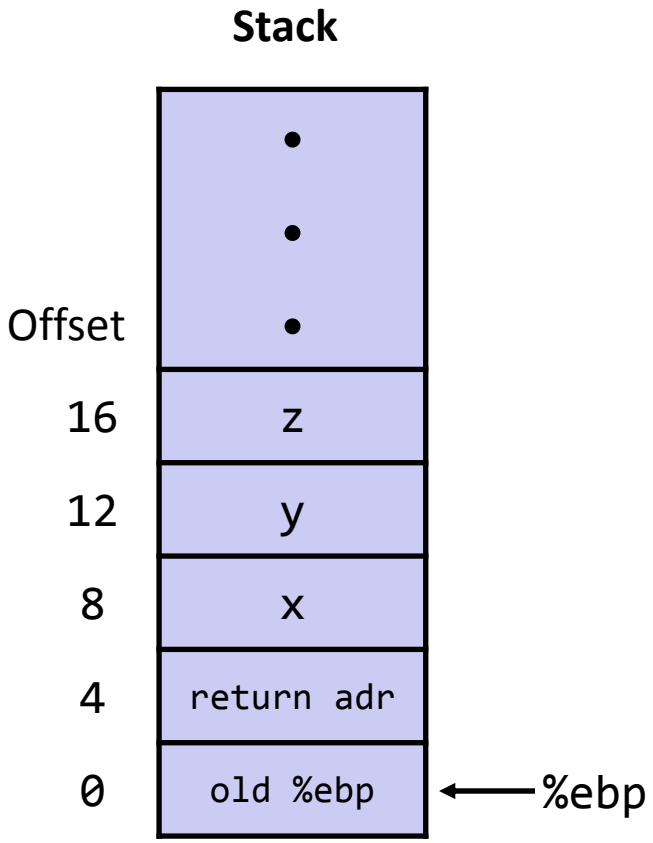
```
long arith2(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```

```
  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
```



Understanding `arith2`

```
long arith2(long x, long y, long z)
{
  long t1 = x+y;
  long t2 = z+t1;
  long t3 = x+4;
  long t4 = y * 48;
  long t5 = t3 + t4;
  long rval = t2 * t5;
  return rval;
}
```




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

Yet 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
    movl %esp,%ebp

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

    popl %ebp
    ret
```

setup

body

finish

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

Yet 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
    movl %esp,%ebp

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

    popl %ebp
    ret
```

setup

body

finish

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

Yet 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
    movl %esp,%ebp

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

    popl %ebp
    ret
```

setup

body

finish

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

Yet 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
    movl %esp,%ebp

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

    popl %ebp
    ret
```

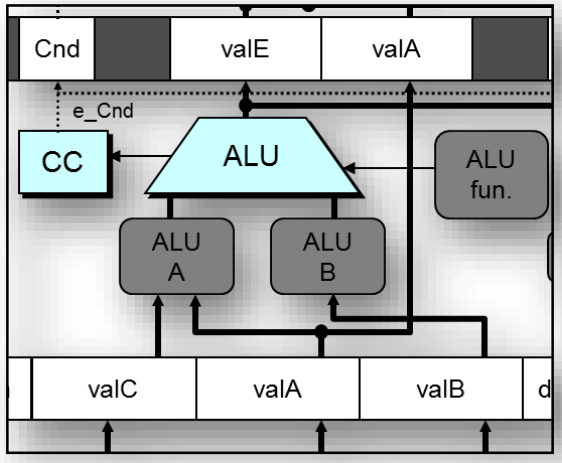
setup

body

finish

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

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

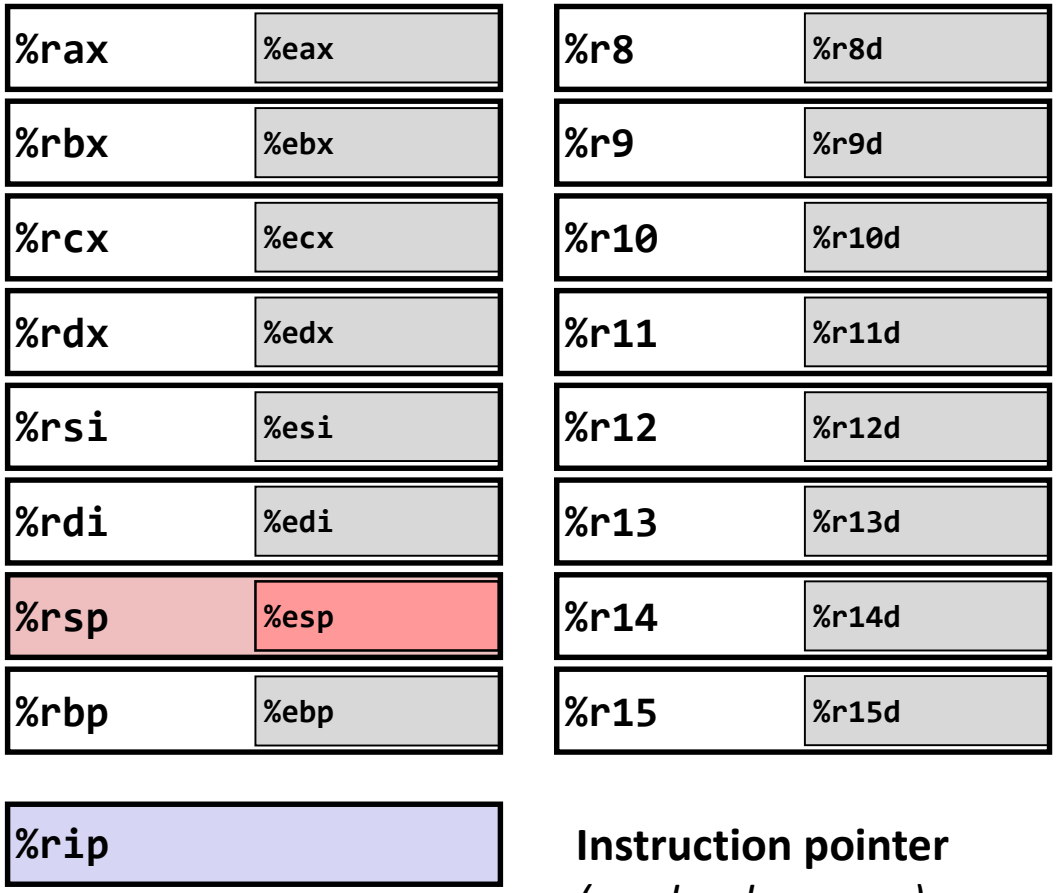


The x86 Instruction Set Architecture:

Control Operations

Processor State (x86_64, Partial)

- Information about currently executing program
  - Temporary data (%rax, …)
  - Location of runtime stack (%rsp)
  - Location of current code control point (%rip, …)
  - **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 by all arithmetic operations (side effect of computation)**

  Example: **`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)
  ```

  - *Except* for **`lea`** - does not set CC

Compare: Explicitly Setting Condition Codes

- Explicit Setting by Compare Instruction

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

  `cmpq b,a` == 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)`

Test: Explicitly Setting Condition Codes

- Explicit Setting by Test instruction

  `testq` *`Src2, Src1`*

  `testq b,a` == computing **`a&b`** without setting destination

  - Sets condition codes based on value of ***`Src1 & Src2`***
  - Useful when one of the operands is a mask
    such as **`if (a & 0xffff) { …`**

  `ZF set` when **`a&b == 0`**
  `SF set` when **`a&b < 0`**

Reading Condition Codes

- SetX Instructions
  - Set single byte to 0/1 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: IA32

- SetX Instructions:
  - Set single byte to 0/1 based on combination of condition codes
- Destination: one of the 8 addressable byte registers
  - Does not alter remaining 3 bytes
  - Typically clear first or 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 : y
setg %al                # al = x > y
movzbl %al,%eax         # zero rest of %eax
```

Reading Condition Codes: x86-64

- SetX Instructions:
  - Set single byte to 0/1 based on combination of condition codes

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

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

Body (same for both)

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

Is %rax zero?

Yes: 32-bit instructions set high order 32 bits to 0!

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 (IA32)

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

- C allows “goto” as means of transferring control
  - Closer to machine-level programming style
- Never use goto in C code!

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

    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6

    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7

.L6:
    subl %edx, %eax

.L7:
    popl %ebp
    ret
```

setup

evaluate condition

if part

else part

finish

Conditional Branch Example (Cont.)

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

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

    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6

    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7

.L6:
    subl %edx, %eax

.L7:
    popl %ebp
    ret
```

setup

evaluate condition

if part

else part

finish

Conditional Branch Example (Cont.)

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

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

    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6

    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7

.L6:
    subl %edx, %eax

.L7:
    popl %ebp
    ret
```

setup

evaluate condition

if part

else part

finish

Conditional Branch Example (Cont.)

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

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

    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6

    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7

.L6:
    subl %edx, %eax

.L7:
    popl %ebp
    ret
```

setup

evaluate condition

if part

else part

finish

Conditional Branch Example (Cont.)

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

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

    movl    8(%ebp), %edx
    movl    12(%ebp), %eax
    cmpl    %eax, %edx
    jle     .L6

    subl    %eax, %edx
    movl    %edx, %eax
    jmp .L7

.L6:
    subl %edx, %eax

.L7:
    popl %ebp
    ret
```

setup

evaluate condition

if part

else part

finish

Using Conditional Moves (cmovX)

- Conditional Move Instructions
  - Instruction supports:

    if (Test) Dest ← Src
  - Supported in post-1995 x86 processors
  - GCC does not always use them
    - Wants to preserve compatibility with ancient processors
    - Enabled for x86-64
    - Use switch `–march=686` for IA32
- Why?
  - Branches are disruptive to instruction flow through pipelines
  - Conditional move do not require control transfer

C Code

```
val = Test
   ? Then_Expr
   : Else_Expr;
```

Goto Version

```
 tval = Then_Expr;
result = Else_Expr;
t = Test;
if (t) result = tval;
return result;
```

Conditional Move Example: 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        # Compare x:y
  cmovle  %edx, %eax        # eax = edx if <=
  ret
```

Bad Cases for Conditional Move

Expensive Computations

```
val = Test(x) ? Hard1(x) : Hard2(x);
```

- Both values get computed

→ cmov only makes sense when computations are very simple

Risky Computations

```
val = p ? *p : 0;
```

- Both values get computed

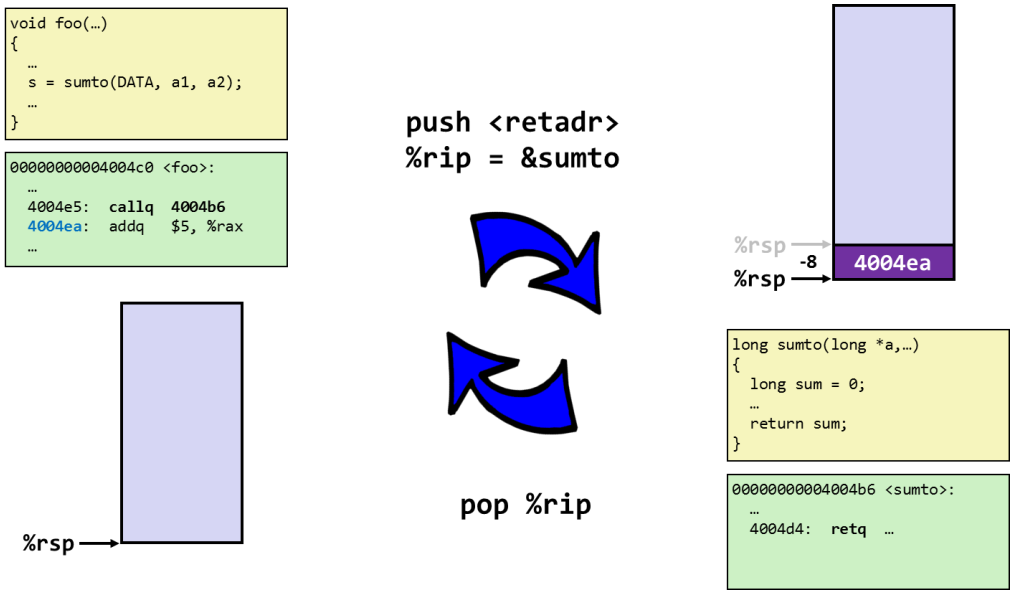
→ cmov may have undesirable effects

Computations with side effects

```
val = x > 0 ? x*=7 : x+=3;
```

- Both values get computed

→ expressions involved in cmov must be side-effect free



The x86 Instruction Set Architecture:

Procedures

The x86 ISA: Procedures

- **Problem Definition**
- The Runtime Stack
- Solving Control Transfer
- Solving Parameter Passing
- Solving Local Storage Allocation
- Calling Conventions and Stack Frames

Calling Procedures/Functions/Methods

```
…
s = sumto(DATA, a1, a2);
…
```

```
long sumto(long *a, int from, int to)
{
  long sum = 0;
  int i;

  for (i=from; i<to; i++) {
    sum += a[i];
  }

  return sum;
}
```

Calling Procedures/Functions/Methods

- **Problems to solve**

1. **Control transfer**
   pass control to `sumto` when the function is invoked, return to the calling code when `sumto` ends

2. **Parameter passing**
   pass arguments in caller to `sumto` such that `sumto` can access them. `sumto` needs to pass a return value back to the caller

3. **Storage for local variables**
   allow `sumto` to store and access local variables for the duration of its execution

```
…
s = sumto(DATA, a1, a2);
…
```

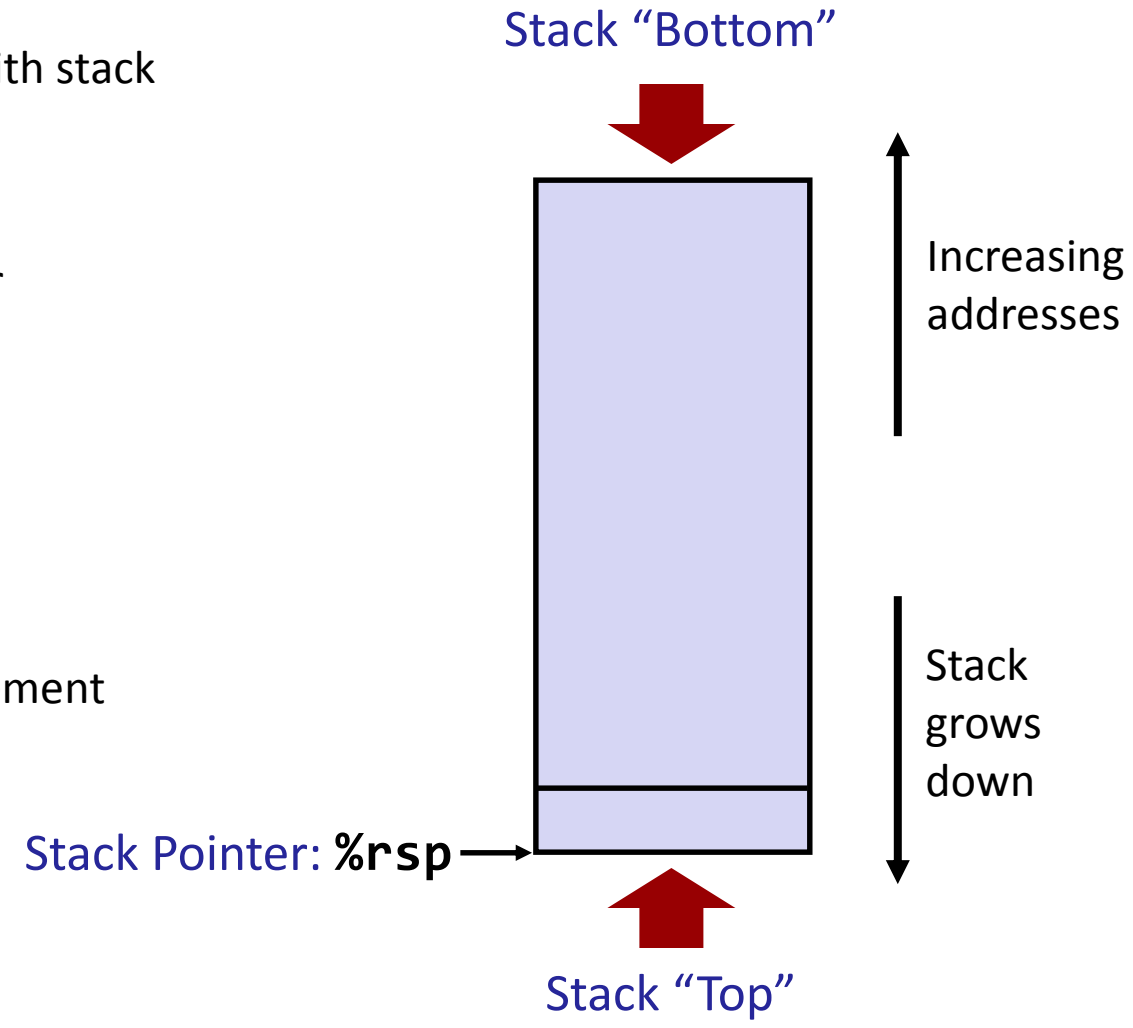
```
long sumto(long *a, int from, int to)
{
  long sum = 0;
  int i;

  for (i=from; i<to; i++) {
    sum += a[i];
  }

  return sum;
}
```

The x86 ISA: Procedures

- Problem Definition
- **The Runtime Stack**
- Solving Control Transfer
- Solving Parameter Passing
- Solving Local Storage Allocation
- Calling Conventions and Stack Frames

The Runtime Stack

- Region of memory managed with stack discipline (first in, last out)
- Provides temporary storage for procedures
- Grows toward lower addresses (for historical reasons)
- Register **`%rsp`** points to top element on stack

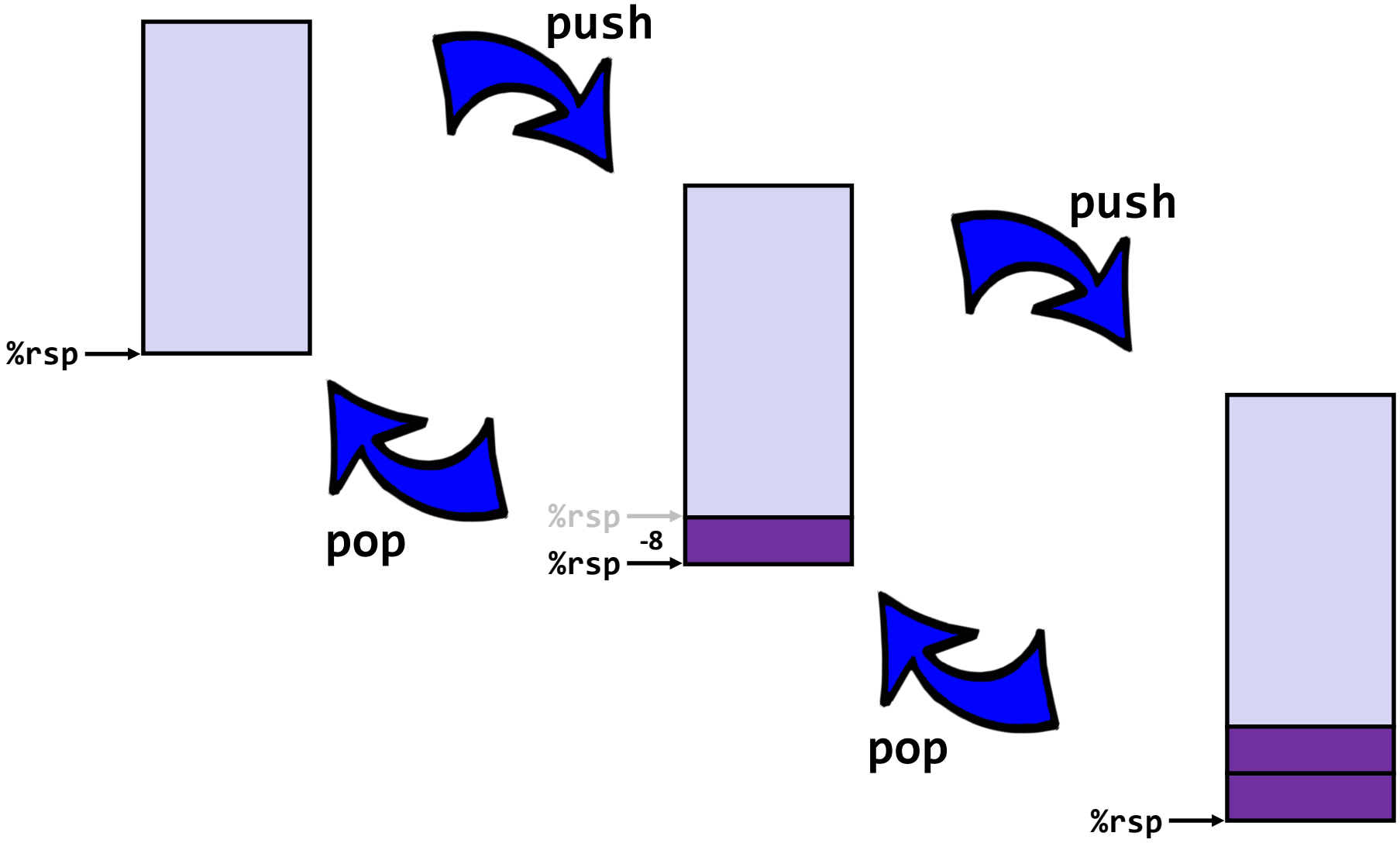


Pushing and Popping Data

- **`pushq` *`Src`***
  - Fetch operand at ***`Src`***
  - Decrement **`%rsp`** by 8
  - Write operand at address given by **`%rsp`**

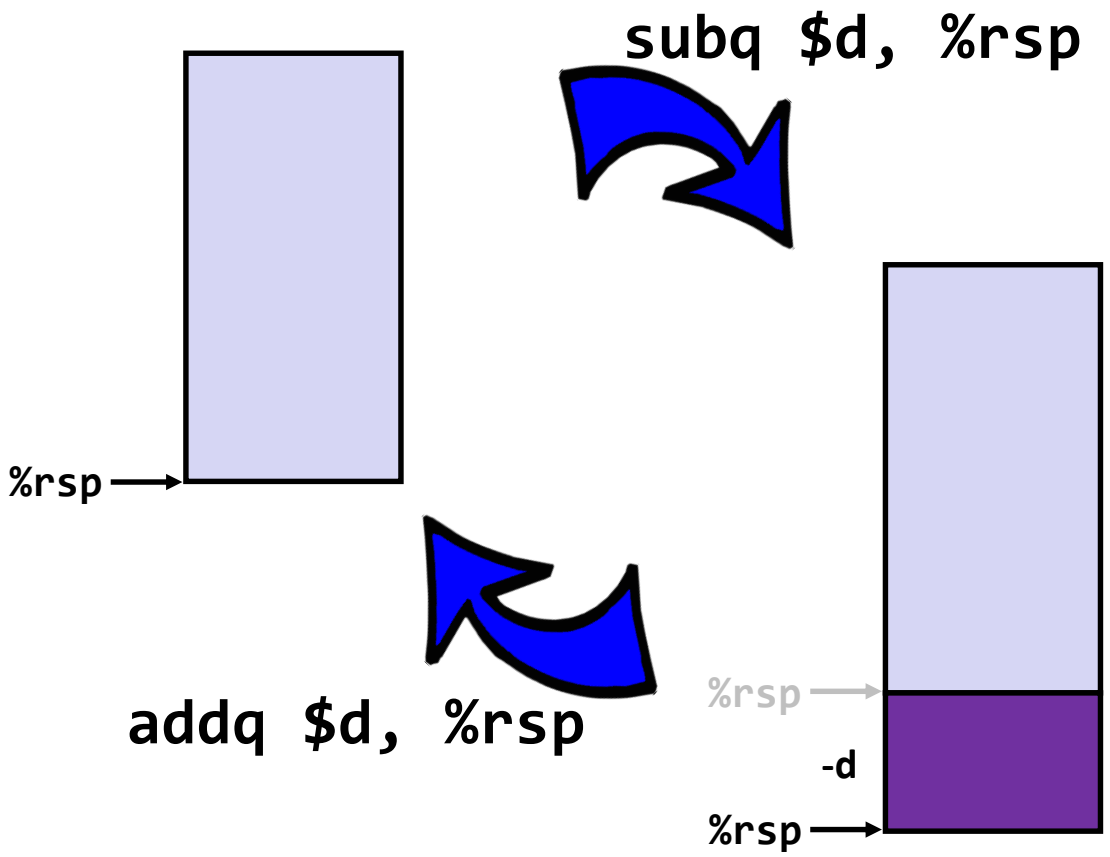
- **`popq` *`Dst`***
  - Fetch operand at **`%rsp`**
  - Increment **`%rsp`** by 8
  - Write operand to ***`Dst`***

Pushing and Popping Data



Allocate / Deallocate Memory on the Stack

- `subq $<amount>, %rsp`
  - Decrement `%rsp` by `amount`
- `addq $<amount>, %rsp`
  - Increment `%rsp` by `amount`



The x86 ISA: Procedures

- Problem Definition
- The Runtime Stack
- **Solving Control Transfer**
- Solving Parameter Passing
- Solving Local Storage Allocation
- Calling Conventions and Stack Frames

Control Transfer: Naïve Approach

```
void foo(…)
{
  …
  s = sumto(DATA, a1, a2);
  …
}
```
foo.c

```
00000000004004c0 <foo>:
  …
  4004d5:  mov    $0x0,%edx
  4004da:  mov    0x202ba0(%rip),%esi
  4004e0:  mov    $0x601080,%edi
  4004e5:  jmp    4004b6 <sumto>     # goto sumto
  4004ea:  addq   $5, %rax
  …
```

```
long sumto(long *a,
       int from, int to)
{
  long sum = 0;
  int i;

  for (i=from; i<to; i++) {
    sum += a[i];
  }

  return sum;
}
```
procedure.c

```
00000000004004b6 <sumto>:
  4004b6:  cmp    %edx,%esi
  4004b8:  jge    4004cf <sumto+0x19>
  4004ba:  mov    $0x0,%eax
  4004bf:  movslq %esi,%rcx
  …
  4004cf:  jmp    4004ae          # return to foo
```

Control Transfer: Why it doesn't work

```
void foo(…)
{
  …
  s = sumto(DATA, a1, a2);
  …
}
```

```
00000000004004c0 <foo>:
  …
  4004e5:  jmp    4004b6 <sumto>
  4004ea:  addq   $5, %rax
  …
```

```
void bar(…)
{
  …
  res = sumto(arr, 0, 5);
  …
}
```

```
00000000004003c8 <bar>:
  …
  4004cf:  jmp    4004b6 <sumto>
  4004d4:  addq   $5, %rax
  …
```

```
long sumto(long *a,
       int from, int to)
{
  long sum = 0;
  int i;

  for (i=from; i<to; i++) {
    sum += a[i];
  }

  return sum;
}
```

```
00000000004004b6 <sumto>:
  4004b6:  cmp    %edx,%esi
  4004b8:  jge    4004cf <sumto+0x19>
  4004ba:  mov    $0x0,%eax
  4004bf:  movslq %esi,%rcx
  …
  4004cf:  jmp    ...?          # return to where?
```

Solving Procedure Control Flow

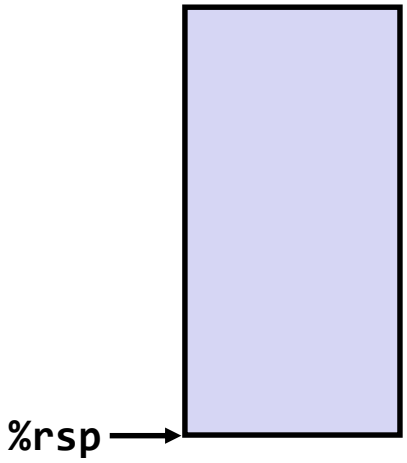
- **Invoking a procedure: `call label`**
  - push address of next instruction onto stack
  - jump to **`label`**

- **Returning from a procedure: `ret`**
  - pop return address from stack
  - jump to address

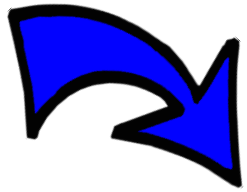
Solving Procedure Control Flow

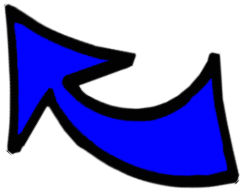
```
void foo(…)
{
  …
  s = sumto(DATA, a1, a2);
  …
}
```

```
00000000004004c0 <foo>:
  …
  4004e5:  callq  4004b6
  4004ea:  addq   $5, %rax
  …
```

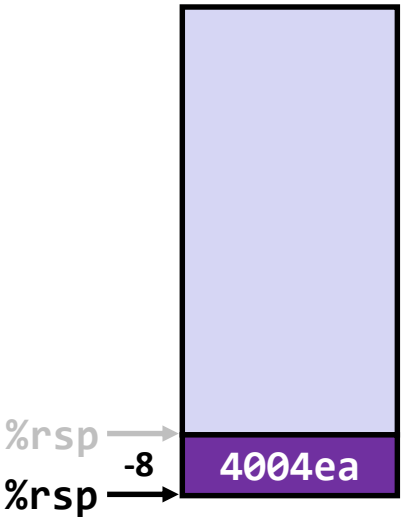


push <retadr>
%rip = &sumto





pop %rip




```
long sumto(long *a,…)
{
  long sum = 0;
  …
  return sum;
}
```

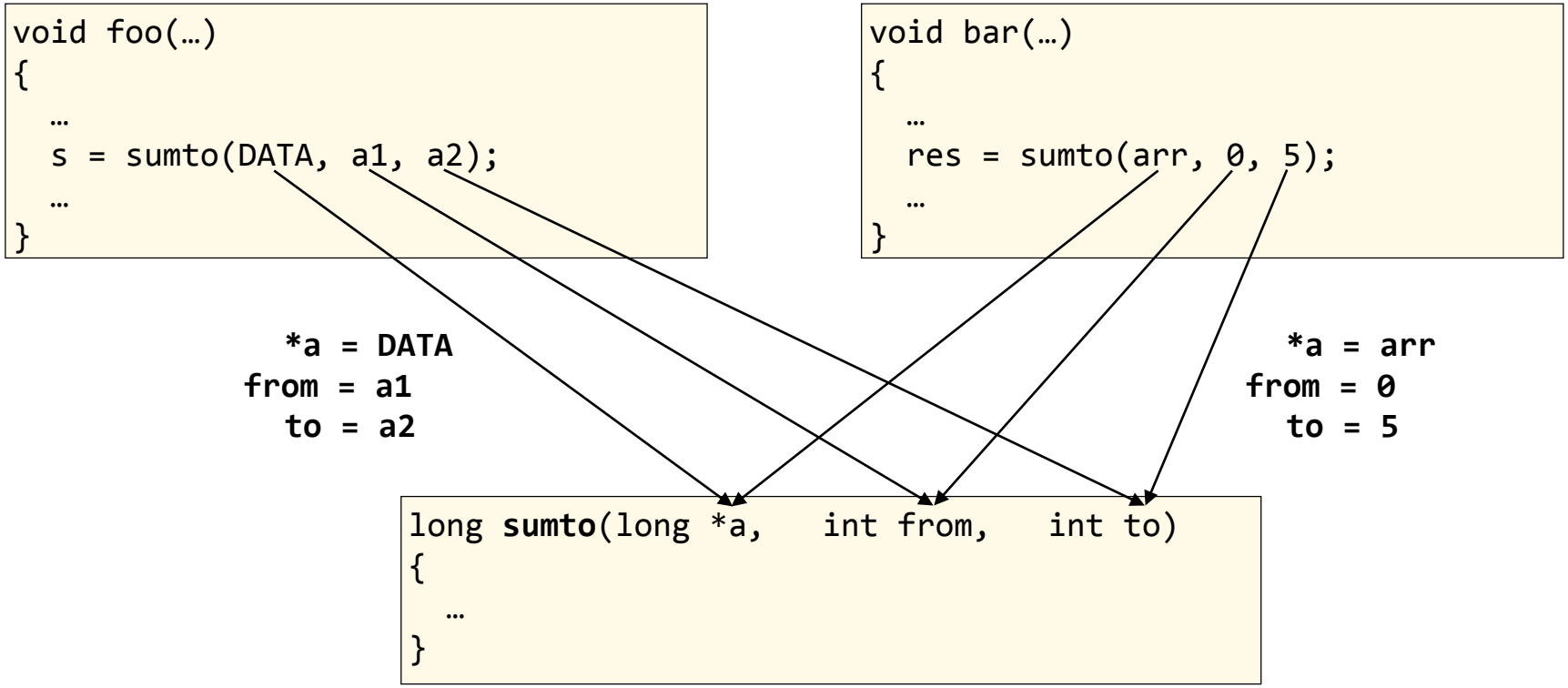
```
00000000004004b6 <sumto>:
  …
  4004d4:  retq  …
```

The x86 ISA: Procedures

- Problem Definition
- The Runtime Stack
- Solving Control Transfer
- **Solving Parameter Passing**
- Solving Local Storage Allocation
- Calling Conventions and Stack Frames

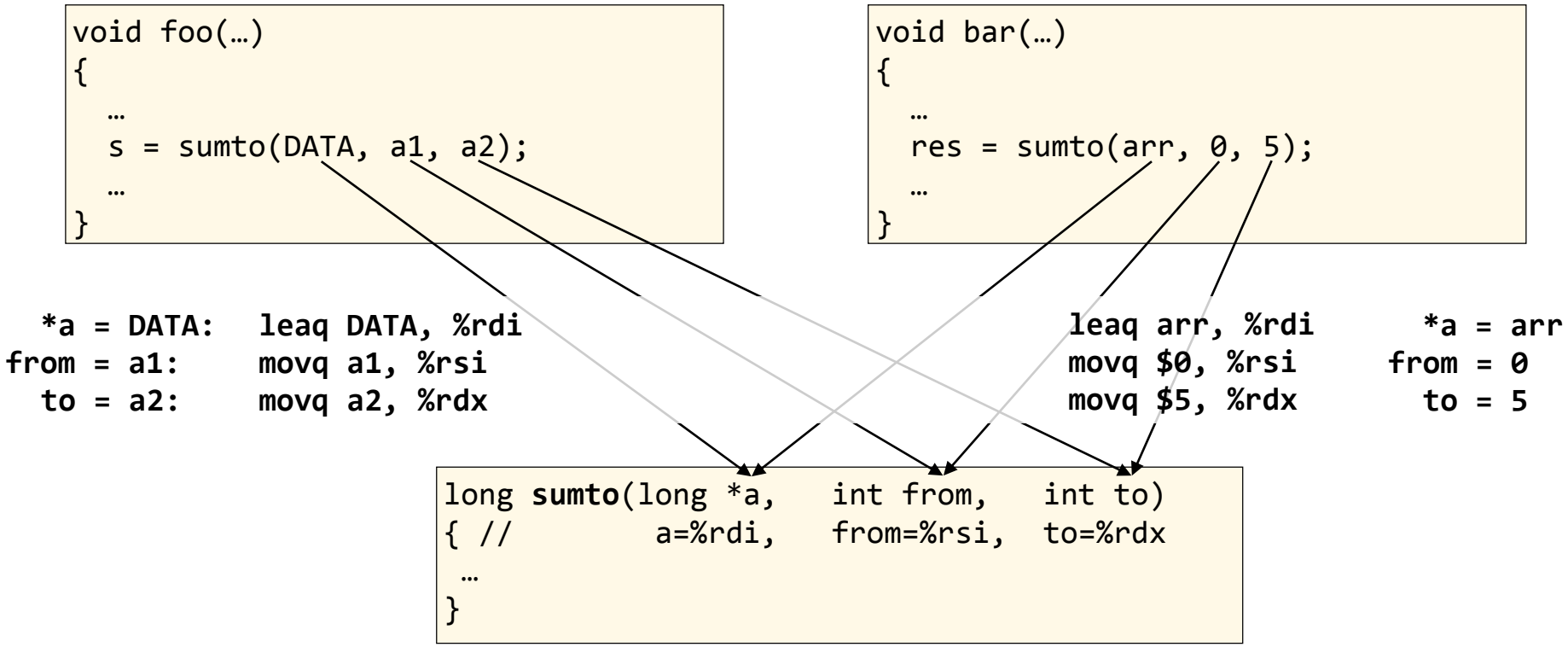
Parameter Passing

- **Need a mapping between arguments and parameters**



Solving Parameter Passing

- **Pass parameters in registers and on the runtime stack**
  - need a convention that defines which parameter maps to which register
  - IA32: pass all parameters on the stack
  - x86_64: pass first 6 parameters in registers, remaining on stack



The x86 ISA: Procedures

- Problem Definition
- The Runtime Stack
- Solving Control Transfer
- Solving Parameter Passing
- **Solving Local Storage Allocation**
- Calling Conventions and Stack Frames

Local Variable Mapping

- **Could try to allocate local variables to a (fixed) memory address**

```
int foo(int n)
{
  int i, j=0;

  for (i=0;i<n;i++)
    j = j+i;

  return j
}
```

```
movl $0, 0x40078c       # j = 0

…
movl 0x400788, %rax     #   =     i
addl %rax, 0x40078c     # j = j +
…
```

```
0x0000000000400788: i
0x000000000040078c: j
```

Local Variable Mapping

- **Fails for recursively called procedures**

```
int foo(int n)
{
  int i, j=0;

  for (i=0;i<n;i++)
    j = j+foo(n-1);

  return j
}
```

```
0x0000000000400788: i
0x000000000040078c: j
```

Solving Local Variable Mapping

- **Allocate on runtime stack**

```
int foo(int n)
{
  int i, j=0;

  for (i=0;i<n;i++)
    j = j+foo(n-1);

  return j
}
```

```
foo:
            pushq       %r13
            pushq       %r12
            pushq       %rbp                # save %rbp
            pushq       %rbx
            subq        $8, %rsp
            testl       %edi, %edi
            jle         .L4
            movl        %edi, %r12d
            movl        $0, %ebp            # j in %rbp
            movl        $0, %ebx
            leal        -1(%rdi), %r13d
.L3:
            movl        %r13d, %edi
            call        foo
            addl        %eax, %ebp          # j = j+foo()

            addl        $1, %ebx
            cmpl        %r12d, %ebx
            jne         .L3
            jmp         .L2
.L4:
            movl        $0, %ebp
.L2:
            movl        %ebp, %eax          # move j to %rax
            addq        $8, %rsp
            popq        %rbx
            popq        %rbp                # restore %rbp
            popq        %r12
            popq        %r13
            ret
```

The x86 ISA: Procedures

- Problem Definition
- The Runtime Stack
- Solving Control Transfer
- Solving Parameter Passing
- Solving Local Storage Allocation
- **Calling Conventions and Stack Frames**

Register Saving Conventions

- When procedure **yoo** calls **who**:
  - **yoo** is the ***caller***
  - **who** is the ***callee***

- Can registers be used for temporary storage?

```
yoo:
    …
    movl $15213, %edx
    call who
    addl %edx, %eax
   …
   ret
```

```
who:
    …
    movl 8(%ebp), %edx
    addl $18243, %edx
    …
    ret
```

  - Contents of register **%edx** overwritten by **who**
  - This could be trouble → something should be done!
    - Need some coordination

Register Saving Conventions

- When procedure **yoo** calls **who**:
  - **yoo** is the ***caller***
  - **who** is the ***callee***

- **Calling Convention**
  - ***"Caller Save"***
    - registers that the callee can overwrite
      (the caller assumes their value is not preserved across procedure calls)
    - Caller saves temporary values in its frame before the call
  - ***"Callee Save"***
    - registers that the callee must preserve before overwriting with a new value
      (the caller can reuse the value across procedure calles)
    - Callee saves temporary values in its frame before using

Calling Convention on x86_64

- **Arguments passed to functions via registers**
  - If more than 6 integral parameters, then pass rest on stack
  - These registers can be used as caller-saved as well
- **All references to stack frame via stack pointer `%rsp`**
- **Register saving convention**
  - 6 “callee saved”
  - 2 “caller saved”
  - 1 return value (also usable as caller saved)
  - 1 special (stack pointer)

Calling Convention on x86_64

%rax	Caller saved / Return value
%rbx	Callee saved
%rcx	Caller saved / Argument #4
%rdx	Caller saved / Argument #3
%rsi	Caller saved / Argument #2
%rdi	Caller saved / Argument #1
%rsp	Stack pointer
%rbp	Callee saved

%r8	Caller saved / Argument #5
%r9	Caller saved / Argument #6
%r10	Caller saved / Caller saved
%r11	Caller Saved
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

Calling Convention on IA32

- **Arguments passed to functions via stack**
- **References to stack frame via base pointer `%ebp`**
- **Register saving convention**
  - 3 “callee saved”
  - 3 “caller saved”
  - 1 return value (also usable as caller saved)
  - 2 special (stack pointer)

IA32/Linux+Windows Calling Convention

- **`%eax, %ecx, %edx`**
  - caller saved prior to call (if values are used later)
  - %eax used to return integer value
- **`%ebx, %esi, %edi`**
  - callee saved (if used)
- **`%esp, %ebp`**
  - used to manage the stack frames
  - must restore original values upon exit from procedure (= special form of callee saved)

`%eax`	Caller saved / Return value
`%ecx`	Caller saved
`%edx`	Caller saved
`%ebx`	Callee saved
`%esi`	Callee saved
`%edi`	Callee saved
`%esp`	Stack pointer
`%ebp`	Frame pointer

Runtime Stack = Good Match for Stack-Based Languages

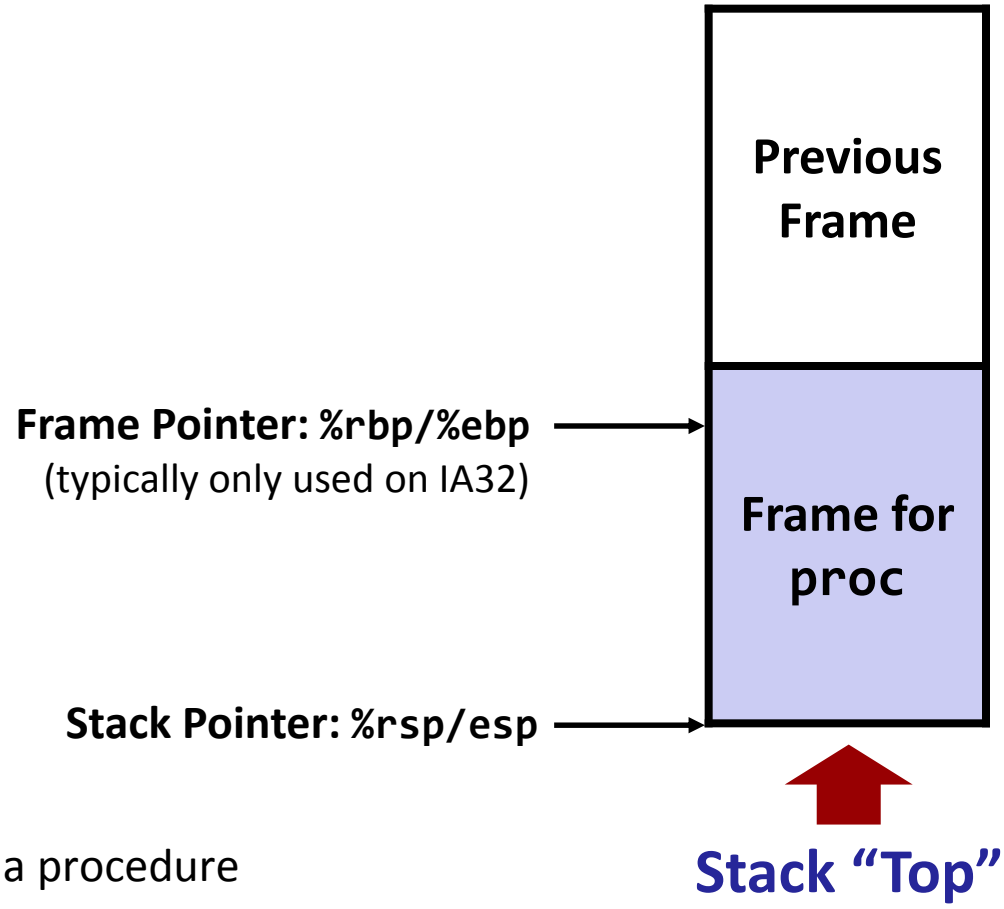
- Languages that support recursion
  - e.g., C, Pascal, Java
  - Code must be *"reentrant"*
    - Multiple simultaneous instantiations of single procedure
  - Need some place to store state of each instantiation
    - Arguments
    - Local variables
    - Return pointer

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

- Stack allocated in ***frames***
  - state for single procedure instantiation

Stack Frames

- Contents
  - Local variables
  - Return information
  - Temporary space
- Management
  - Space allocated when entering a procedure
    - “Set-up” code
  - Deallocated when returning to the caller
    - “Cleanup” code



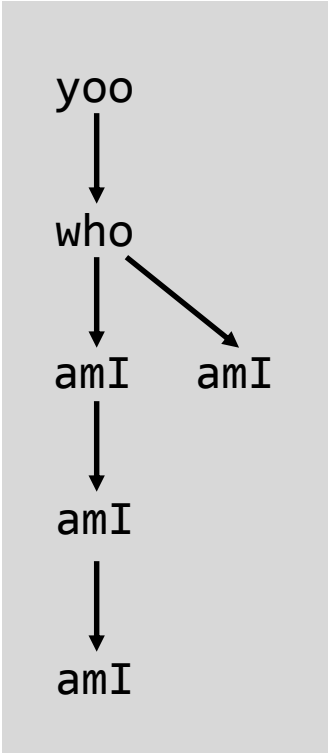
Call Chain Example

```
yoo(…)
{
   •
   •
   who();
   •
   •
}
```

```
who(…)
{
   • • •
   amI();
   • • •
   amI();
   • • •
}
```

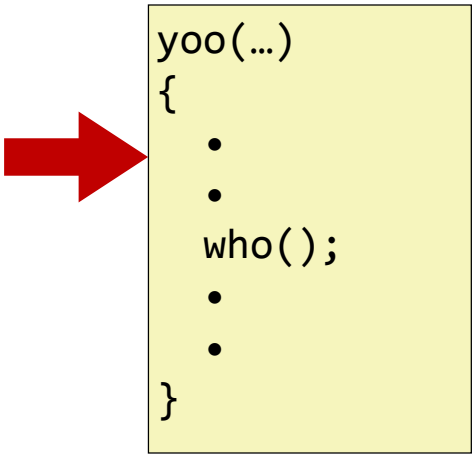
```
amI(…)
{
   •
   •
   amI();
   •
   •
}
```

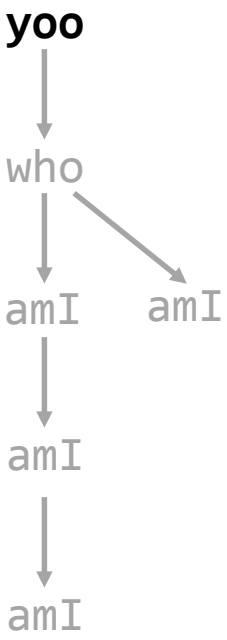
Example Call Chain

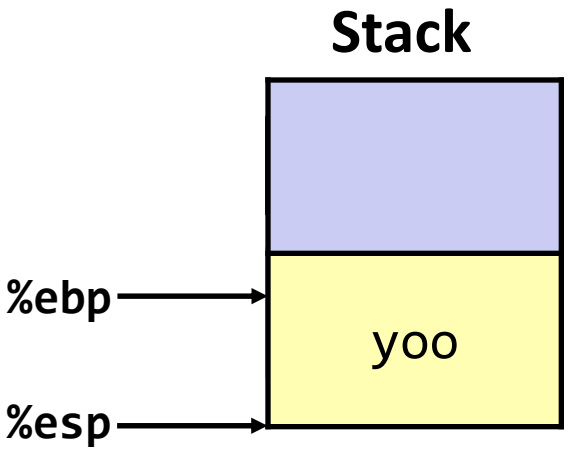


Procedure `amI()` is recursive

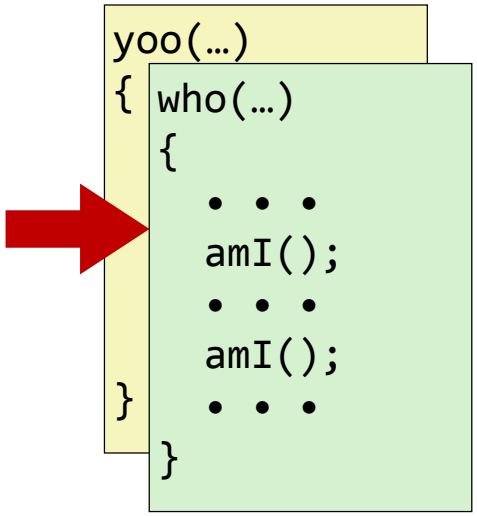
Example

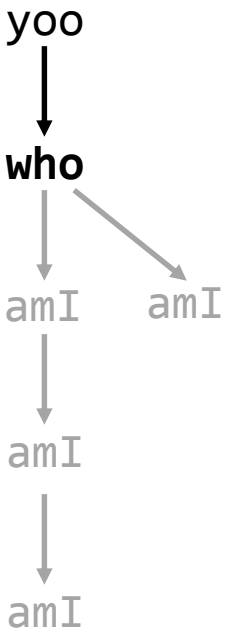


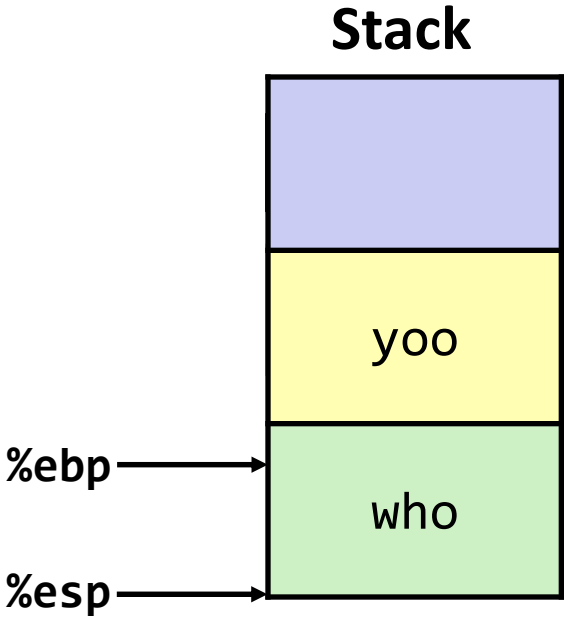




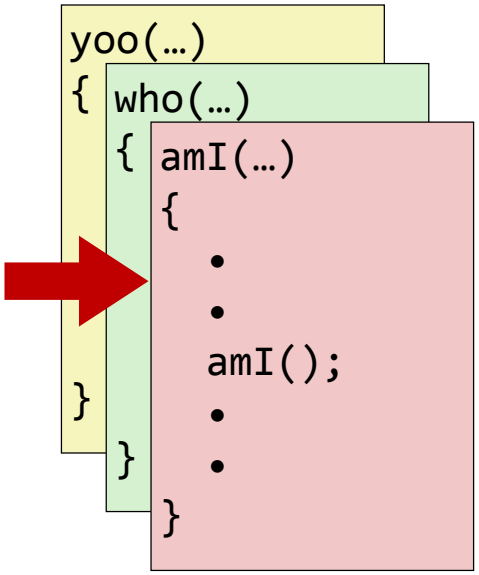
Example

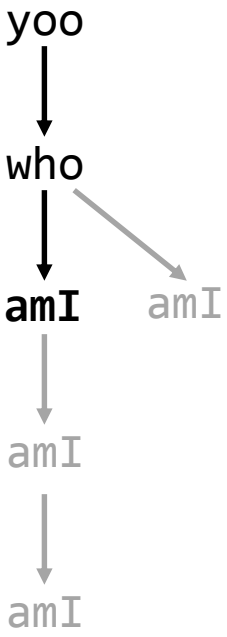


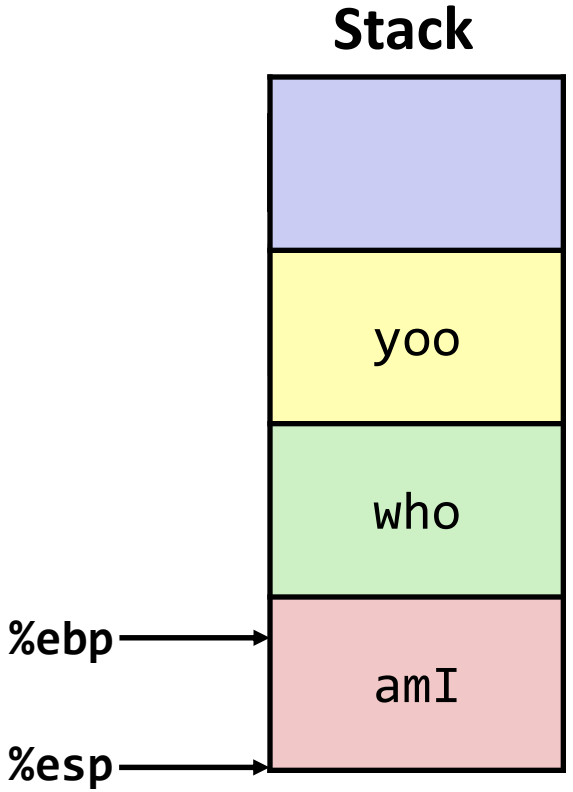




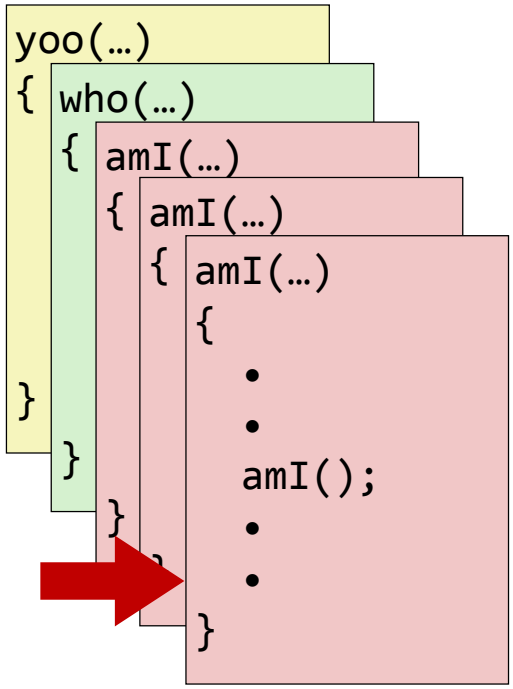
Example

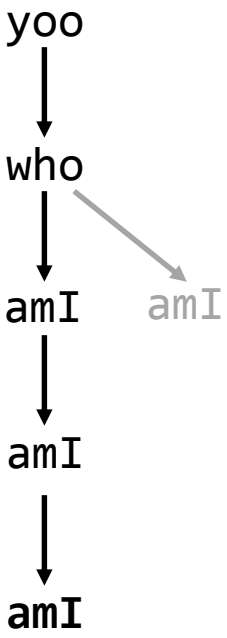

yoo(…)
{ who(…)
{ amI(…)
{
•
•
amI();
•
•
}
}
}

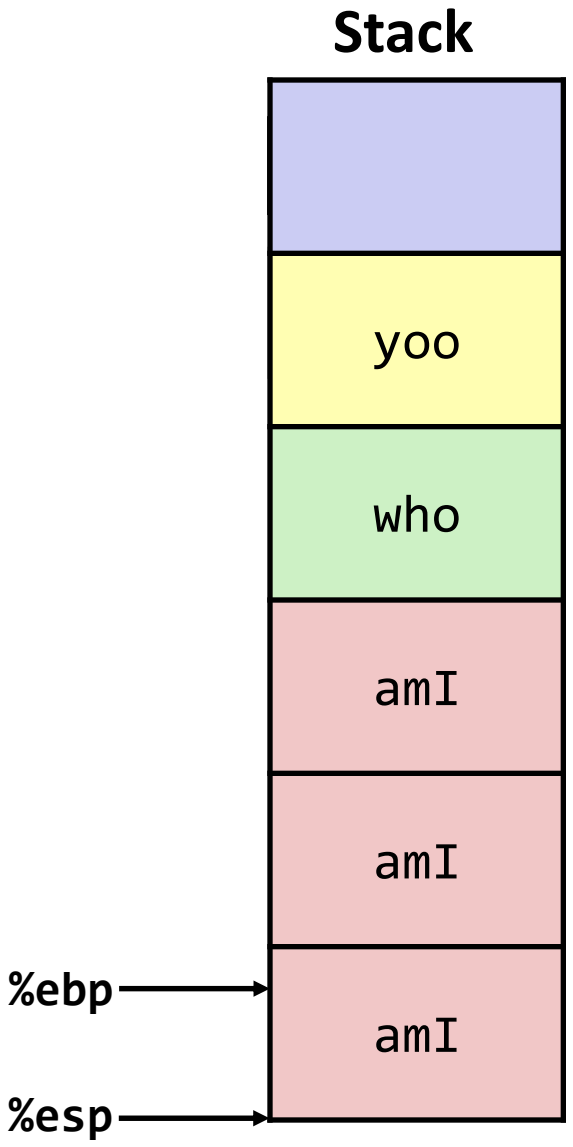

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%ebp
%esp

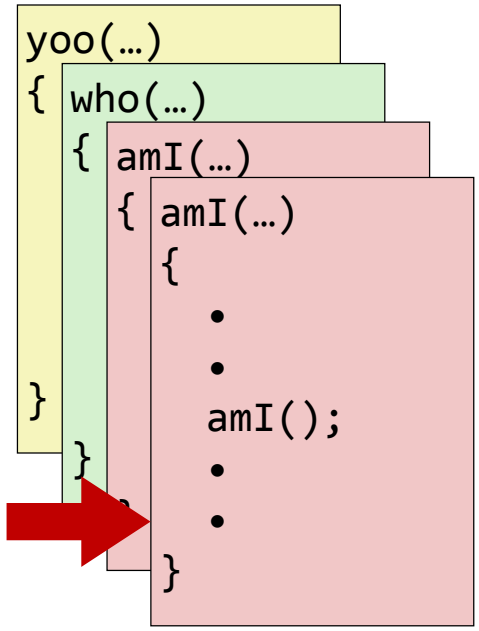
Example

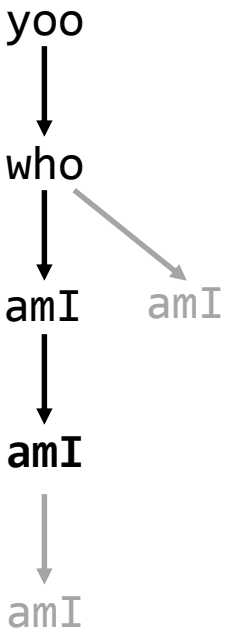

yoo(…)
{ who(…)
{ amI(…)
{ amI(…)
{ amI(…)
{
•
•
amI();
•
•
}
}
}
}

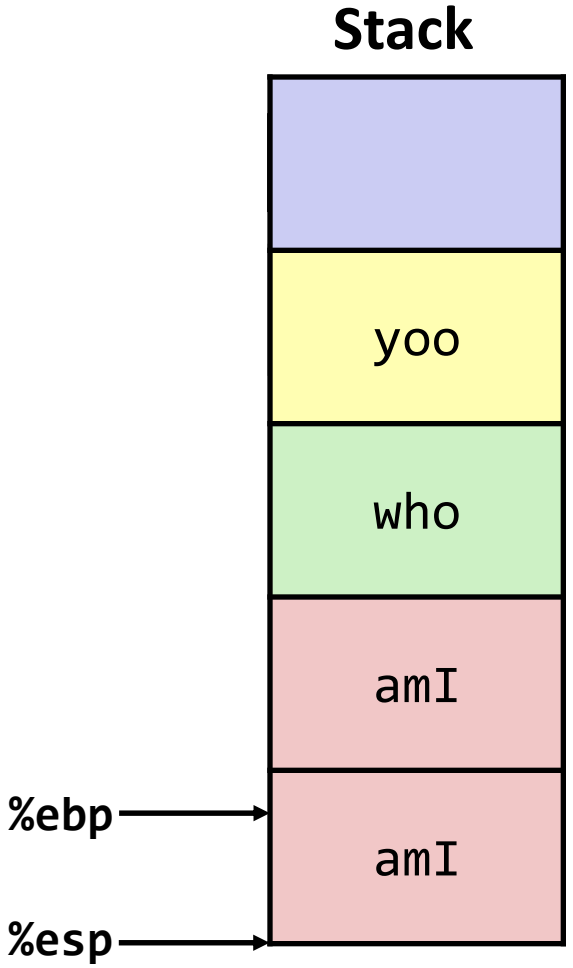

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
amI
%ebp
%esp

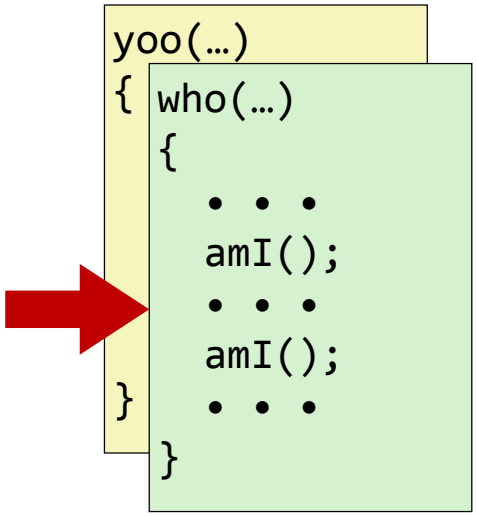
Example

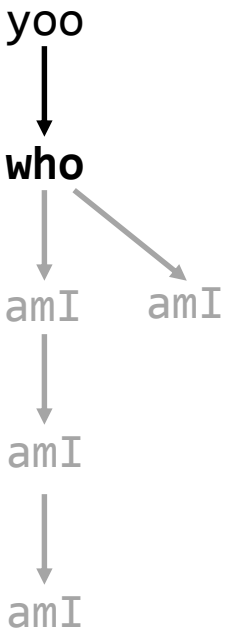

yoo(…)
{ who(…)
{ amI(…)
{ amI(…)
{
•
•
amI();
•
•
}
}
}

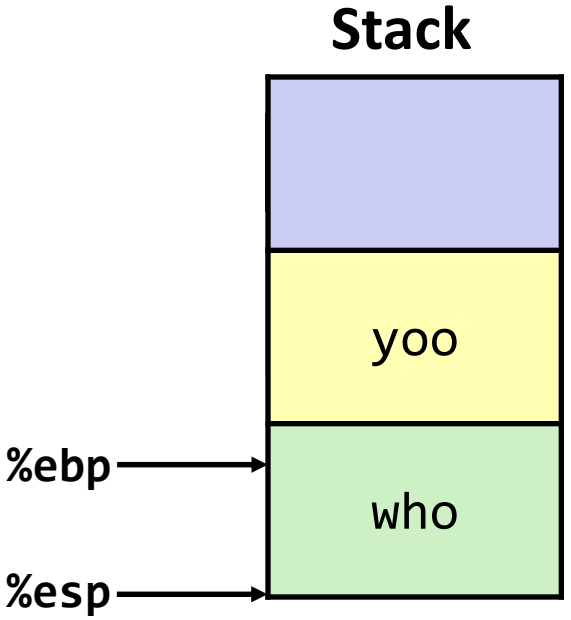

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
amI
%ebp
%esp

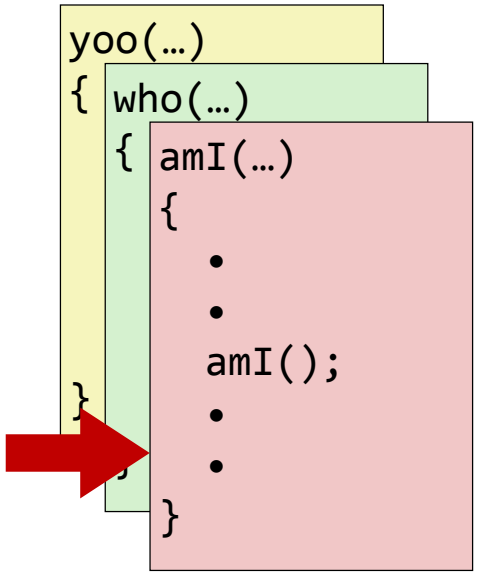
Example

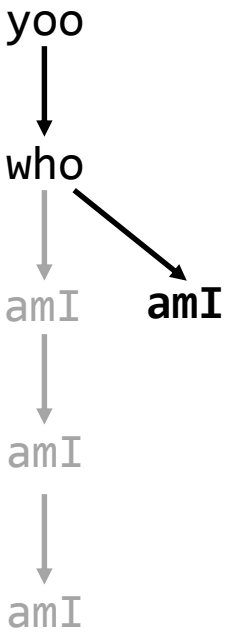


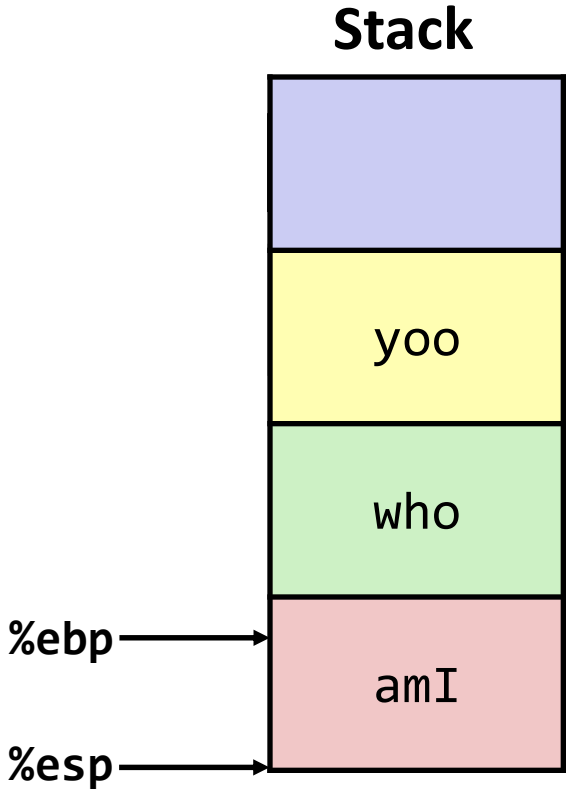




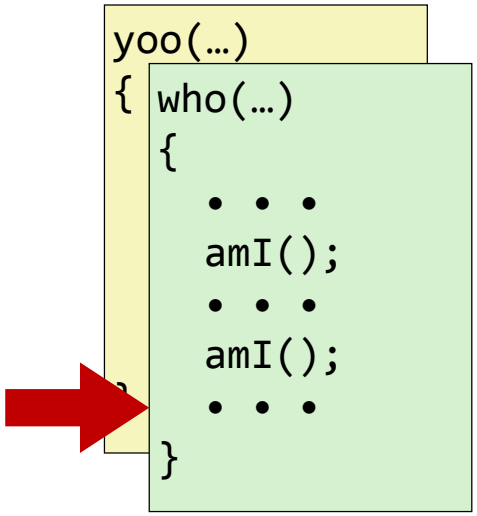
Example

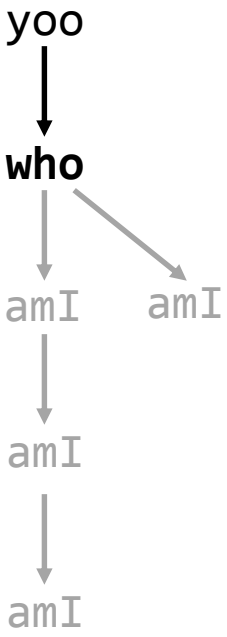

yoo(…)
{ who(…)
{ amI(…)
{
•
•
amI();
•
•
}

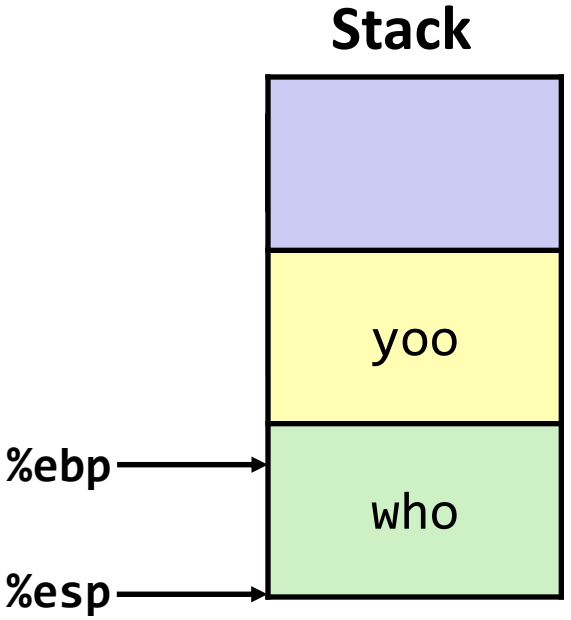

yoo
who
amI
amI
amI
amI


Stack
yoo
who
amI
%ebp
%esp

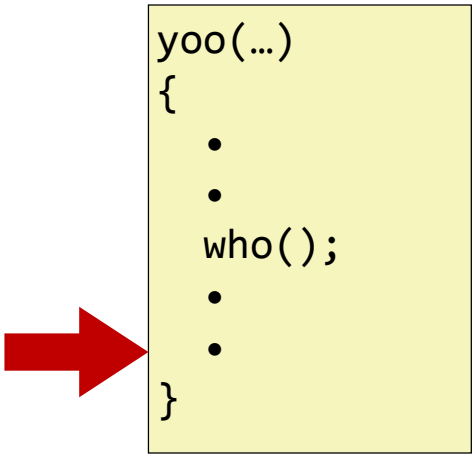
Example

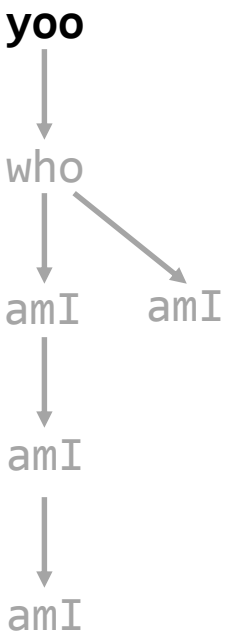


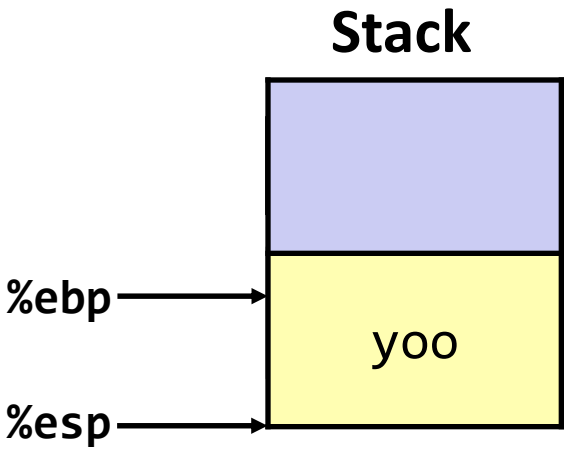




Example



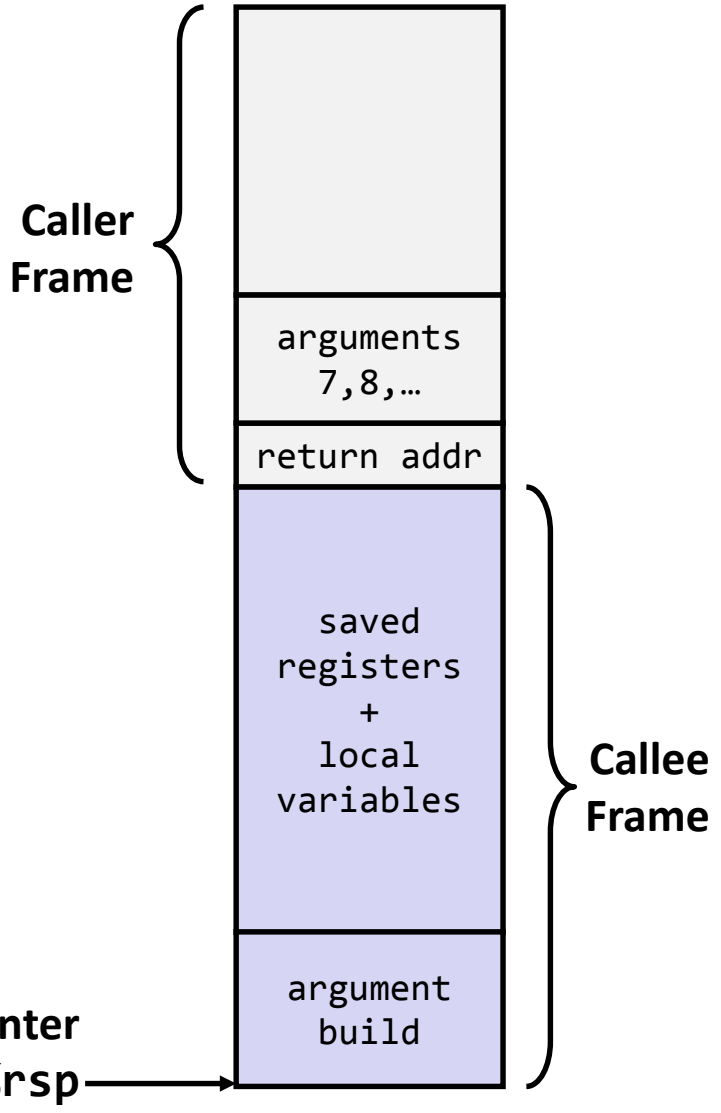




x86_64/Linux Stack Frame

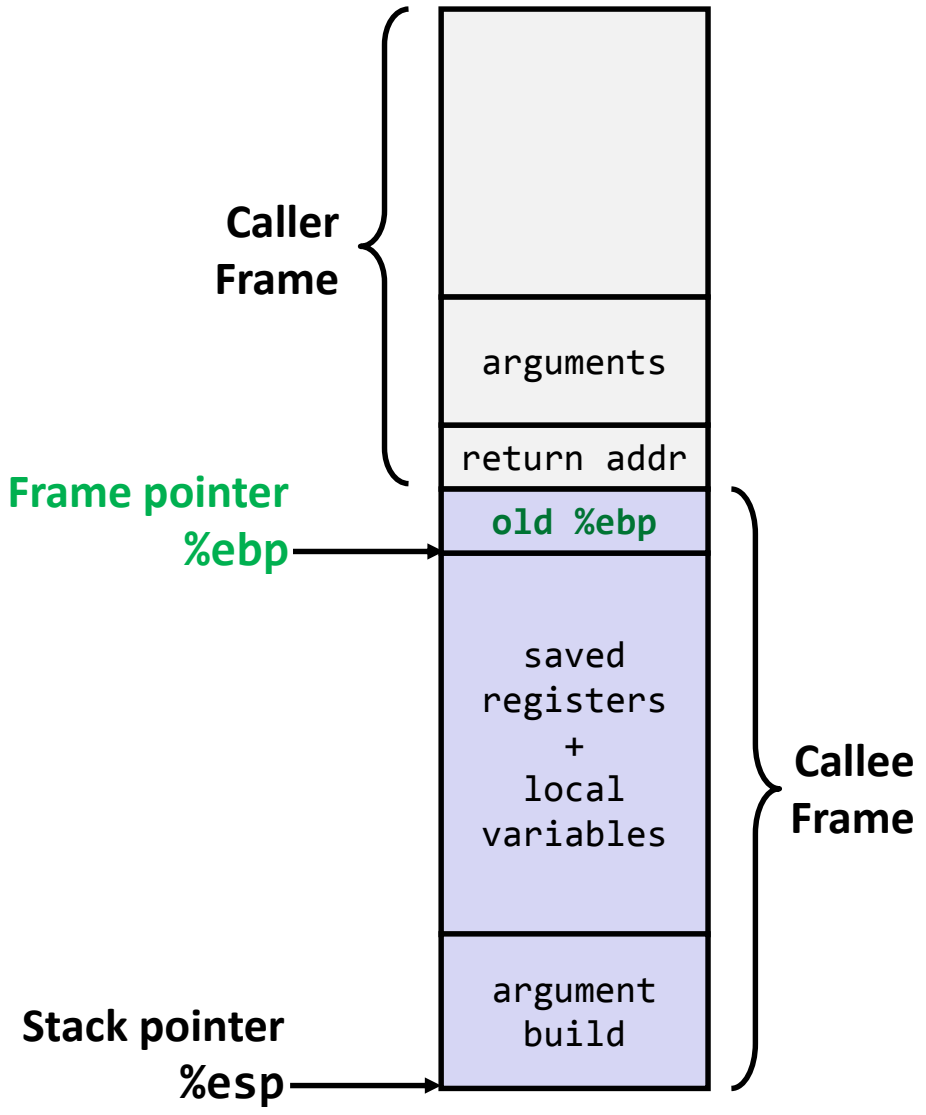
- Current Stack Frame (Top to Bottom)
  - "Argument build"
    arguments ≥ 7 to function about to call
  - Local variables
    those that cannot be kept in local registers
  - Saved register context
    those that need to be saved to adhere to the calling convention

- Caller Stack Frame
  - Return address
    - Pushed automatically by `call` instruction
  - Arguments ≥ 7 for this call
    first 6 are passed in registers



IA32/Linux Stack Frame

- Current Stack Frame (Top to Bottom)
  - "Argument build:"
    Parameters for function about to call
  - 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



x86-64 Stack Frame Example

```
long sum = 0;

// swap a[i] & a[i+1]
void swap_ele_su(long a[], int i)
{
    swap(&a[i], &a[i+1]);
    sum += (a[i]*a[i+1]);
}
```

- Keeps values of &a[i] and &a[i+1] in callee save registers
- Must set up stack frame to save these registers

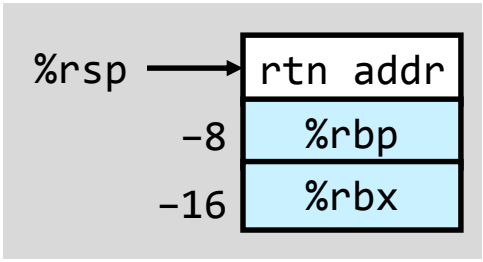
```
swap_ele_su:
    movq    %rbx, -16(%rsp)
    movq    %rbp, -8(%rsp)
    subq    $16, %rsp
    movslq  %esi,%rax
    leaq    8(%rdi,%rax,8), %rbx
    leaq    (%rdi,%rax,8), %rbp
    movq    %rbx, %rsi
    movq    %rbp, %rdi
    call    swap
    movq    (%rbx), %rax
    imulq   (%rbp), %rax
    addq    %rax, sum(%rip)
    movq    (%rsp), %rbx
    movq    8(%rsp), %rbp
    addq    $16, %rsp
    ret
```

Understanding x86-64 Stack Frames

```
swap_ele_su:
    movq    %rbx, -16(%rsp)          # Save %rbx
    movq    %rbp, -8(%rsp)           # Save %rbp
    subq    $16, %rsp                # Allocate stack frame

    movslq  %esi,%rax                # Extend i
    leaq    8(%rdi,%rax,8), %rbx     # &a[i+1] (callee save)
    leaq    (%rdi,%rax,8), %rbp      # &a[i]   (callee save)
    movq    %rbx, %rsi               # 2nd argument
    movq    %rbp, %rdi               # 1st argument
    call    swap

    movq    (%rbx), %rax             # Get a[i+1]
    imulq   (%rbp), %rax             # Multiply by a[i]
    addq    %rax, sum(%rip)          # Add to sum

    movq    (%rsp), %rbx             # Restore %rbx
    movq    8(%rsp), %rbp            # Restore %rbp
    addq    $16, %rsp                # Deallocate frame

    ret
```

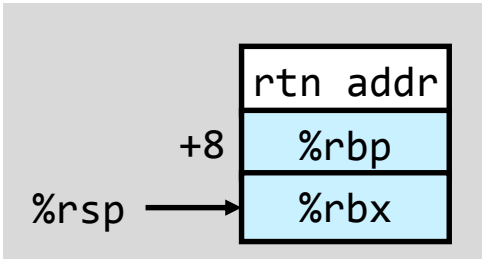
Understanding x86-64 Stack Frames

```
movq    %rbx, -16(%rsp)        # Save %rbx
movq    %rbp, -8(%rsp)         # Save %rbp
```




```
subq    $16, %rsp              # Allocate stack frame
```

…



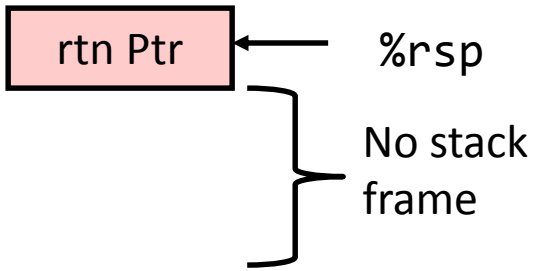

```
movq    (%rsp), %rbx           # Restore %rbx
movq    8(%rsp), %rbp          # Restore %rbp

addq    $16, %rsp              # Deallocate frame
```

x86-64 Long Swap

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

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

- Operands passed in registers
  - First (xp) in `%rdi`, second (yp) in `%rsi`
  - 64-bit pointers
- No stack operations required (except ret)
- Avoiding stack
  - Can hold all local information in registers

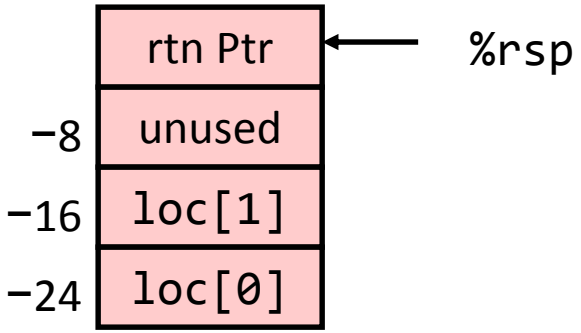


x86-64 Locals in the Red Zone

```
// Swap, using local array
void swap_a(long *xp, long *yp) {
    volatile long loc[2];
    loc[0] = *xp;
    loc[1] = *yp;
    *xp = loc[1];
    *yp = loc[0];
}
```

```
swap_a:
   movq   (%rdi), %rax
   movq   %rax, -24(%rsp)
   movq   (%rsi), %rax
   movq   %rax, -16(%rsp)
   movq   -16(%rsp), %rax
   movq   %rax, (%rdi)
   movq   -24(%rsp), %rax
   movq   %rax, (%rsi)
   ret
```

- Avoiding Stack Pointer Change
  - Can hold all information within small window beyond stack pointer



x86-64 NonLeaf without a Stack Frame

```
// Swap a[i] & a[i+1]
void swap_ele(long a[], int i)
{
    swap(&a[i], &a[i+1]);
}
```

- No values held while swap being invoked
- No callee save registers needed
- rep instruction inserted as no-op (recommendation from AMD)

```
swap_ele:
    movslq %esi,%rsi                # Sign extend i
    leaq   8(%rdi,%rsi,8), %rax     # &a[i+1]
    leaq   (%rdi,%rsi,8), %rdi      # &a[i] (1st arg)
    movq   %rax, %rsi               # (2nd arg)
    call   swap
    rep                             # No-op
    ret
```

Interesting Features of Stack Frames

- Allocate entire frame at once
  - All stack accesses can be relative to %rsp
  - Do by decrementing stack pointer
  - Can delay allocation, since safe to temporarily use red zone
- Simple deallocation
  - Increment stack pointer
  - No base/frame pointer needed

x86-64 Procedure Summary

- Heavy use of registers
  - Parameter passing
  - More temporaries since more registers
- Minimal use of stack
  - Sometimes none
  - Allocate/deallocate entire block
- Many tricky optimizations
  - What kind of stack frame to use
  - Various allocation techniques

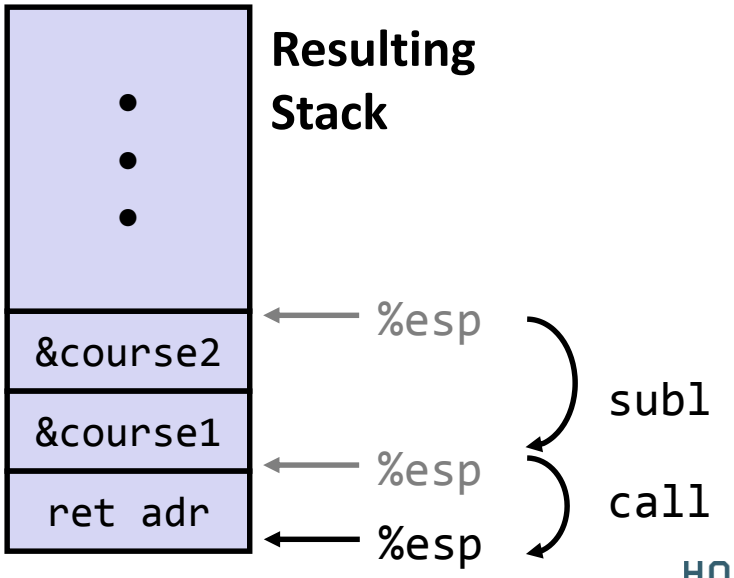
IA32 – Revisiting swap

```
int course1 = 15213;
int course2 = 18243;

void call_swap() {
  swap(&course1, &course2);
}
```

Calling `swap` from `call_swap`

```
call_swap:
    …
    subl    $8, %esp
    movl    $course2, 4(%esp)
    movl    $course1, (%esp)
    call    swap
    …
```

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



IA32 – Revisiting swap

```
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
```

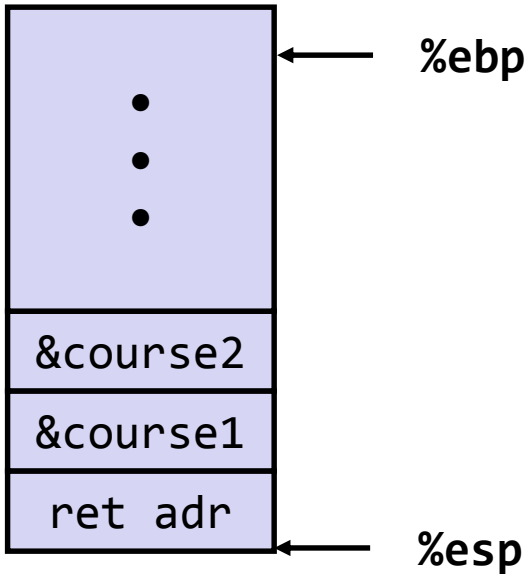
Set Up

Body

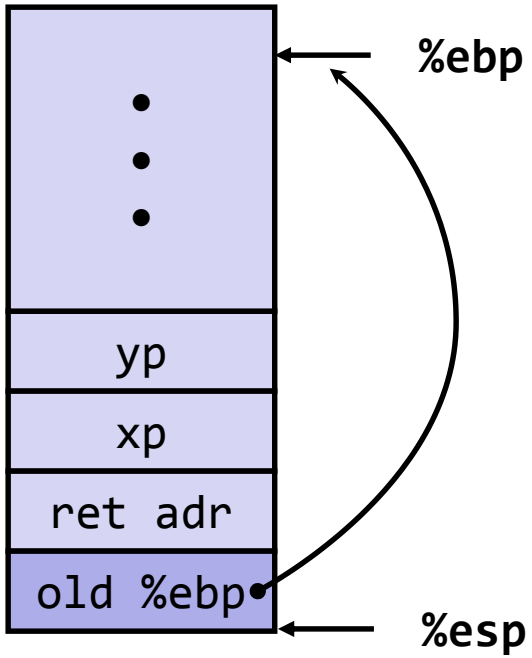
Finish

IA32 – swap Setup #1

Entering Stack



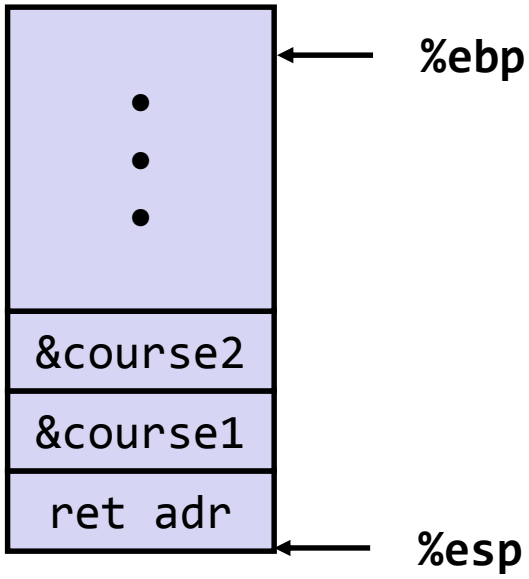
Resulting Stack



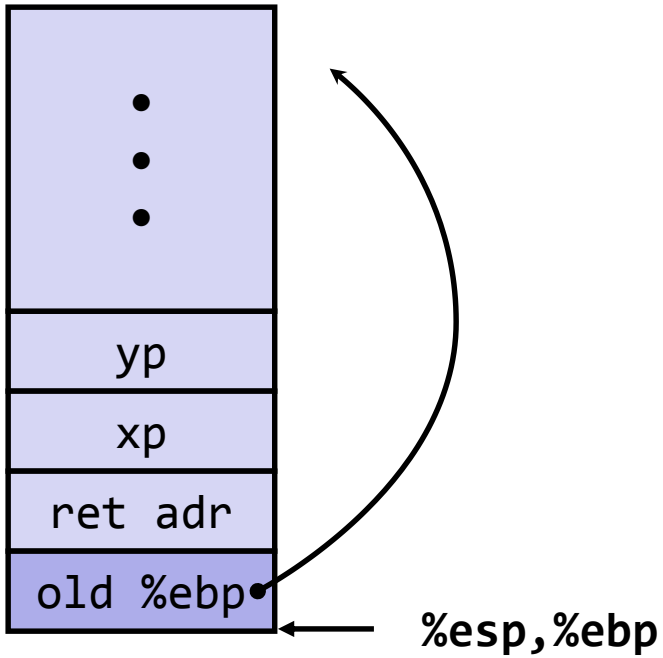

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

IA32 – swap Setup #2

Entering Stack



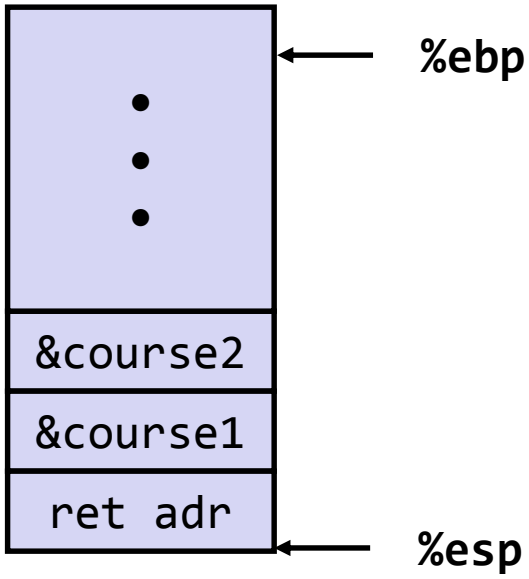
Resulting Stack



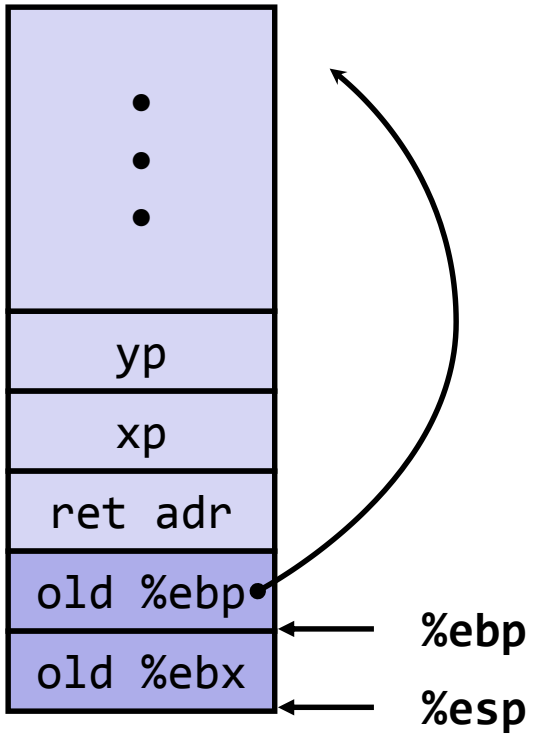

```
swap:
    pushl %ebp
    movl %esp,%ebp
    pushl %ebx
```

IA32 – swap Setup #3

Entering Stack

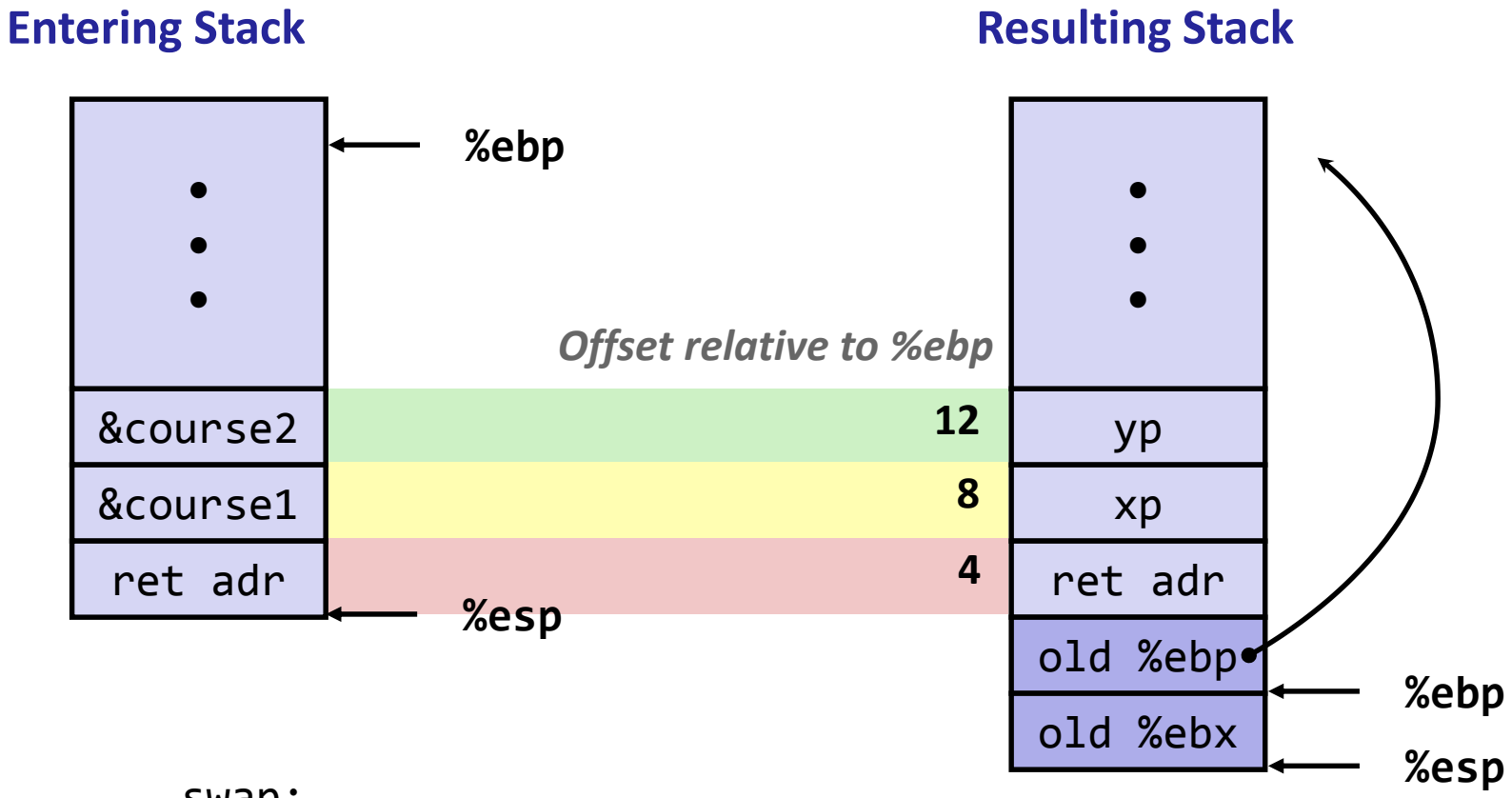


Resulting Stack




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

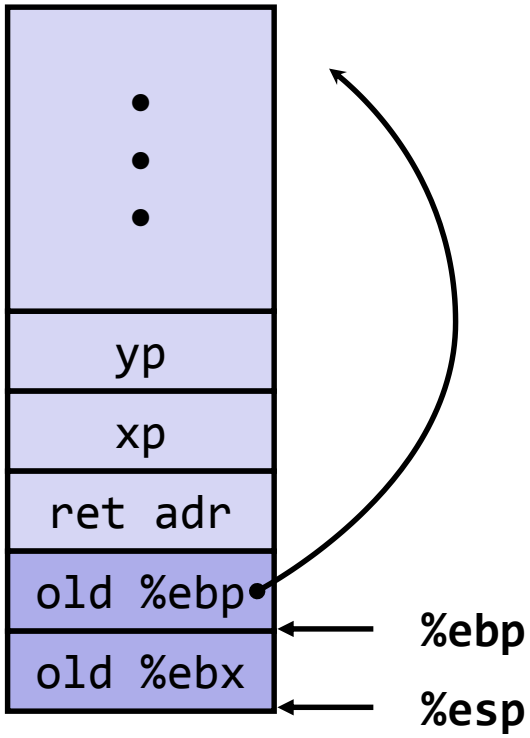
IA32 – swap Setup #3




```
swap:
    ...
    movl 8(%ebp),%edx    # get xp
    movl 12(%ebp),%ecx   # get yp
    ...
```

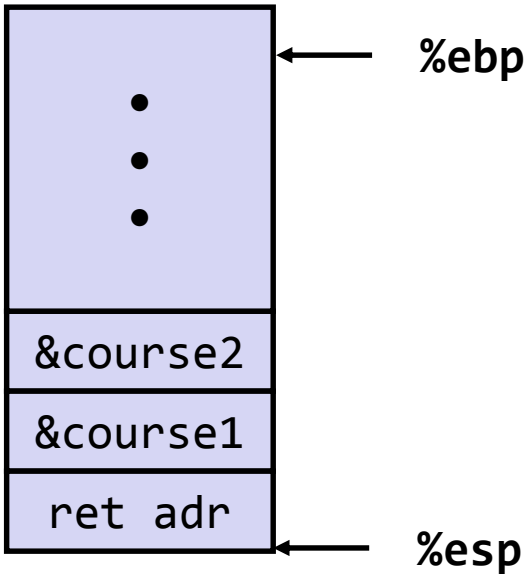
IA32 – swap Cleanup

Stack before Cleanup




```
popl    %ebx
popl    %ebp
```

Resulting Stack



- Observations
  - Saved and restored register **%ebx, %ebp**
  - Not so for **%eax, %ecx, %edx**
  - Modified **%esp**, but value after the call is the same as before the call

IA32 Procedure Summary

- Limited use of registers
  - Only return value passed in a register
- Heavy use of stack
  - Parameter passing
  - Allocate/deallocate dynamically or statically (entire block)
- Strict conventions
  - No values below stack pointer allowed

Intel ISA: Procedures

- Problem Definition
- The Runtime Stack
- Solving Control Transfer
- Solving Parameter Passing
- Solving Local Storage Allocation
- Calling Conventions and Stack Frames
- **Illustrations of Recursion & Pointers**

Recursive Function

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

- Registers
  - %eax, %edx used without first saving
  - %ebx used, but saved at the beginning and restored at the end

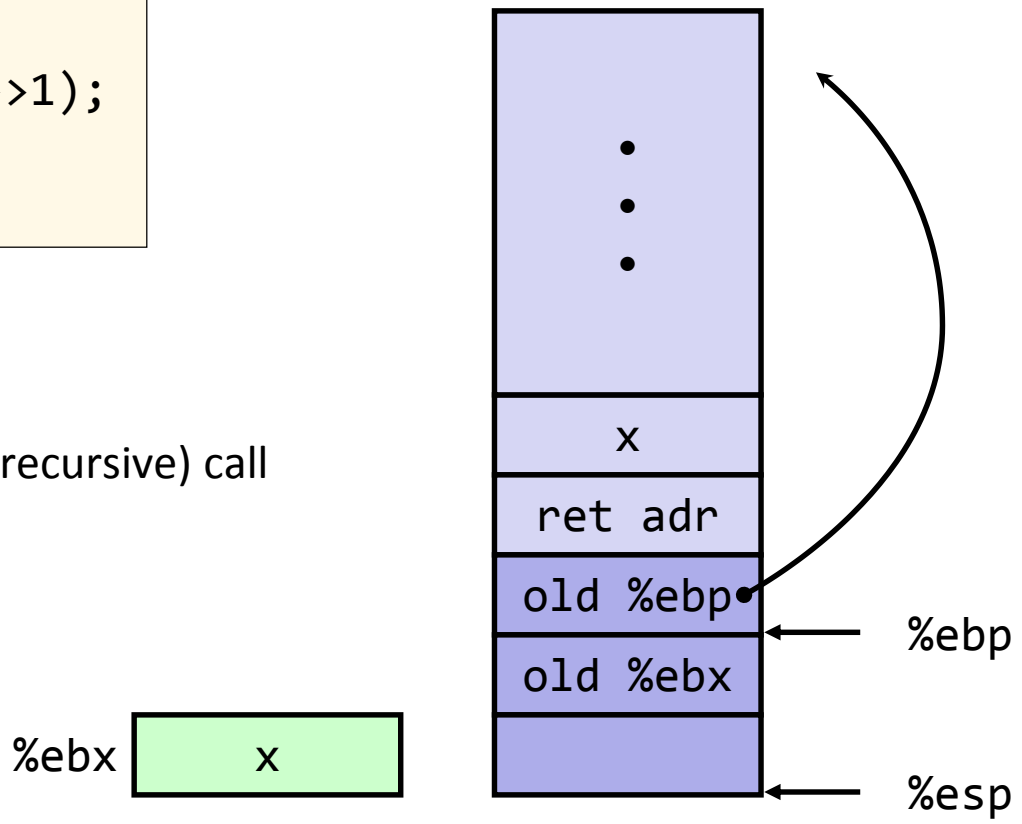
```
pcount_r:
    pushl %ebp
    movl  %esp, %ebp
    pushl %ebx
    subl  $4, %esp
    movl  8(%ebp), %ebx
    movl  $0, %eax
    testl %ebx, %ebx
    je .L3
    movl  %ebx, %eax
    shrl  %eax
    movl  %eax, (%esp)
    call  pcount_r
    movl  %ebx, %edx
    andl  $1, %edx
    leal  (%edx,%eax), %eax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

Recursive Call #1

```
// 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

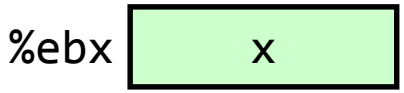


Recursive Call #2

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

```
    …
    movl  $0, %eax
    testl %ebx, %ebx
    je .L3
    …
.L3:
    …
    ret
```

- Actions
  - If x == 0, return
    - with %eax set to 0

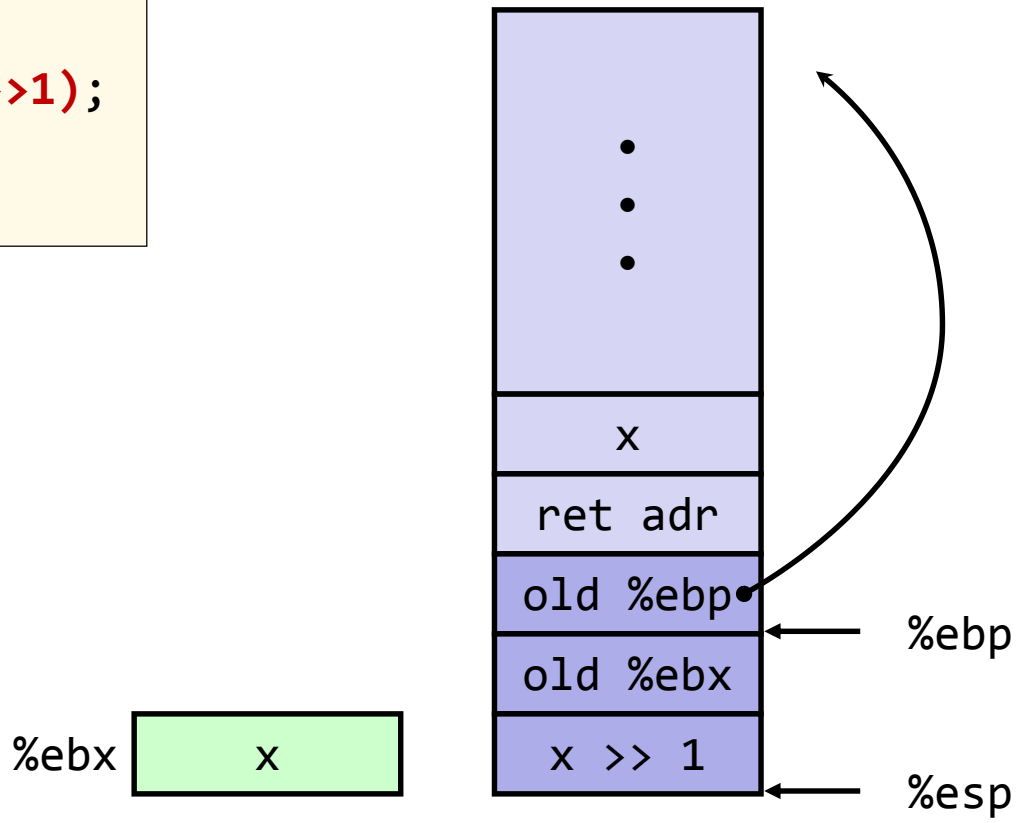


Recursive Call #3

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

- Actions
  - Store x >> 1 on stack
  - Make recursive call
- Effect
  - %eax set to function result
  - %ebx still has value of x

```
…
movl  %ebx, %eax
shrl  %eax
movl  %eax, (%esp)
call  pcount_r
…
```

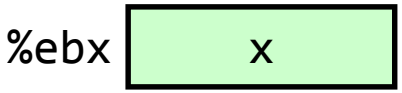


Recursive Call #4

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

```
…
movl  %ebx, %edx
andl  $1, %edx
leal  (%edx,%eax), %eax
…
```

- Assume
  - %eax holds value from recursive call
  - %ebx holds x
- Actions
  - Compute (x & 1) + computed value
- Effect
  - %eax set to function result

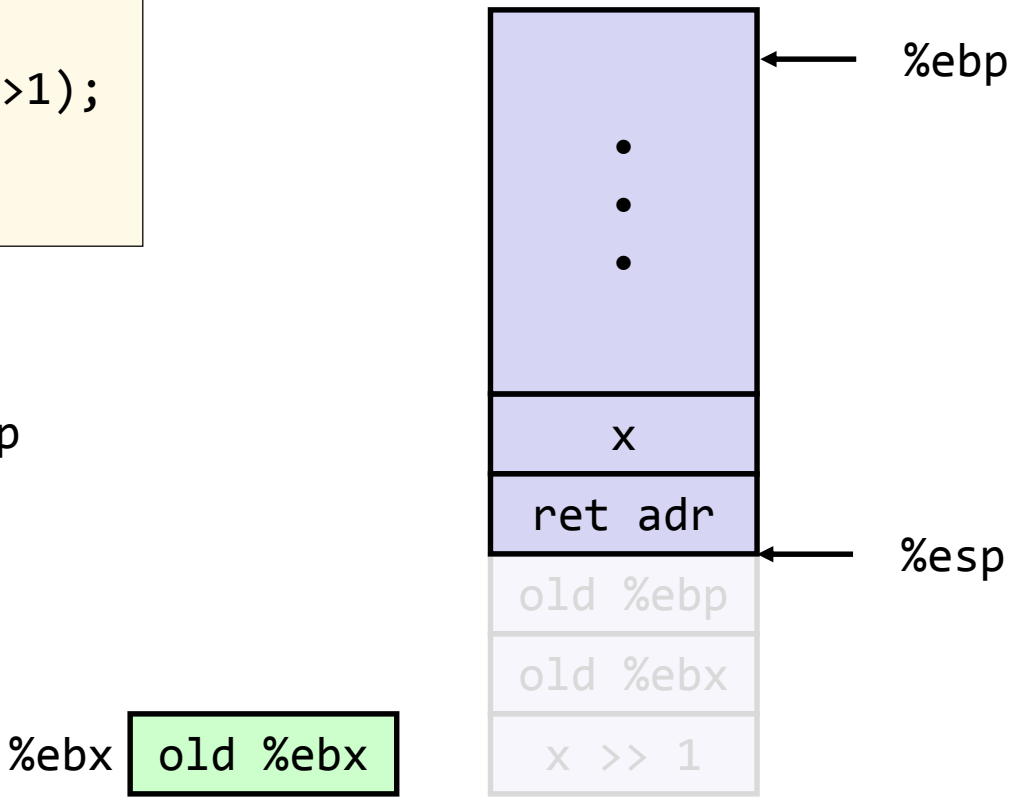


Recursive Call #5

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

- Actions
  - Deallocate space for argument
  - Restore values of %ebx and %ebp
  - **ret** will pop the return address into %eip

```
      …
L3:
      addl  $4, %esp
      popl  %ebx
      popl  %ebp
      ret
```



Observations About Recursion

- Handled Without Special Consideration
  - Stack frames mean that each function call has private storage
    - Saved registers & local variables
    - Saved return pointer
  - Register saving conventions prevent one function call from corrupting another's data
  - Stack discipline follows call / return pattern
    - If P calls Q, then Q returns before P
    - Last-In, First-Out
- Also works for mutual recursion
  - P calls Q; Q calls P

Pointer Code

- `add3` creates pointer and passes it to `incrk`

Generating a Pointer

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

Referencing a Pointer

```
// Increment value by k
void incrk(int *ip, int k) {
  *ip += k;
}
```

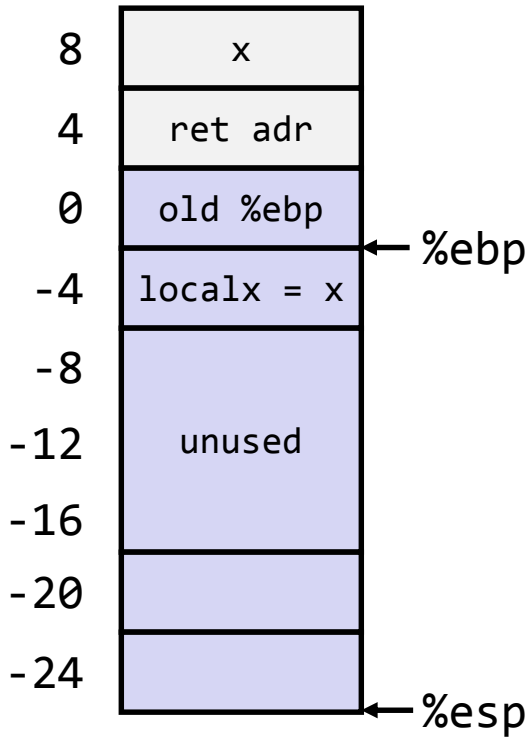
Creating and Initializing Local Variables

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

- variable `localx` must be stored on the stack
  - the compiler needs to create a pointer to it
- compute pointer as `-4(%ebp)`

First part of `add3`

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



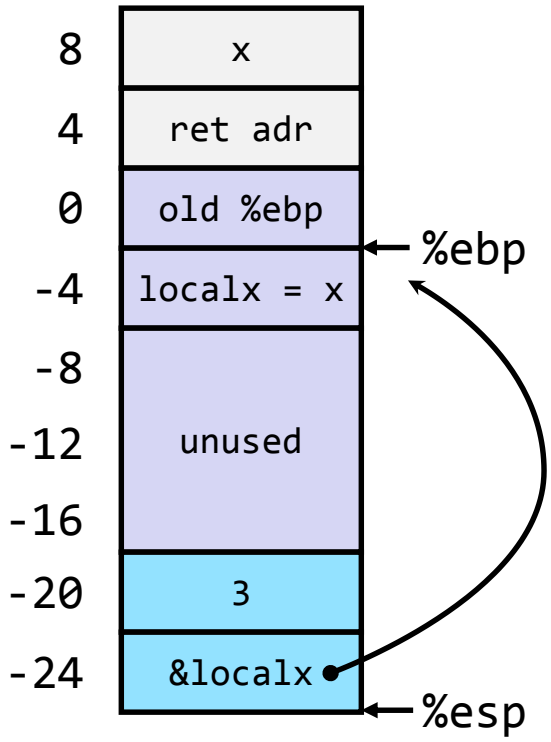
Creating Pointer as Argument

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

- Use `leal` to compute the 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
```



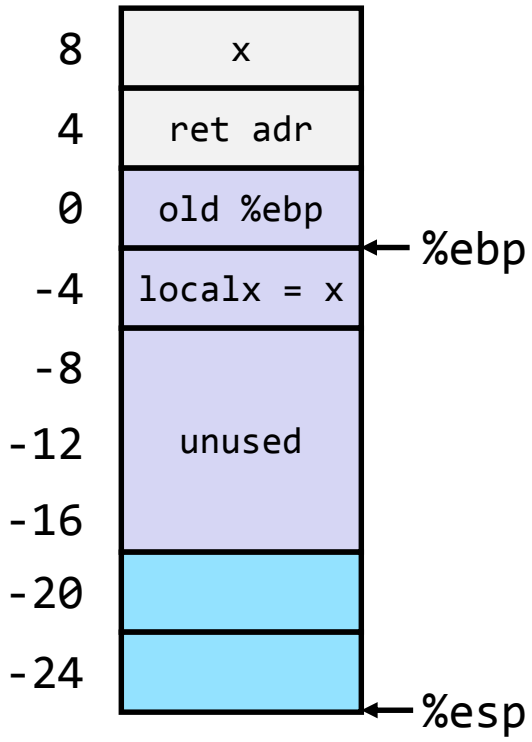
Retrieving local variable

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

- Retrieve `localx` from stack as return value

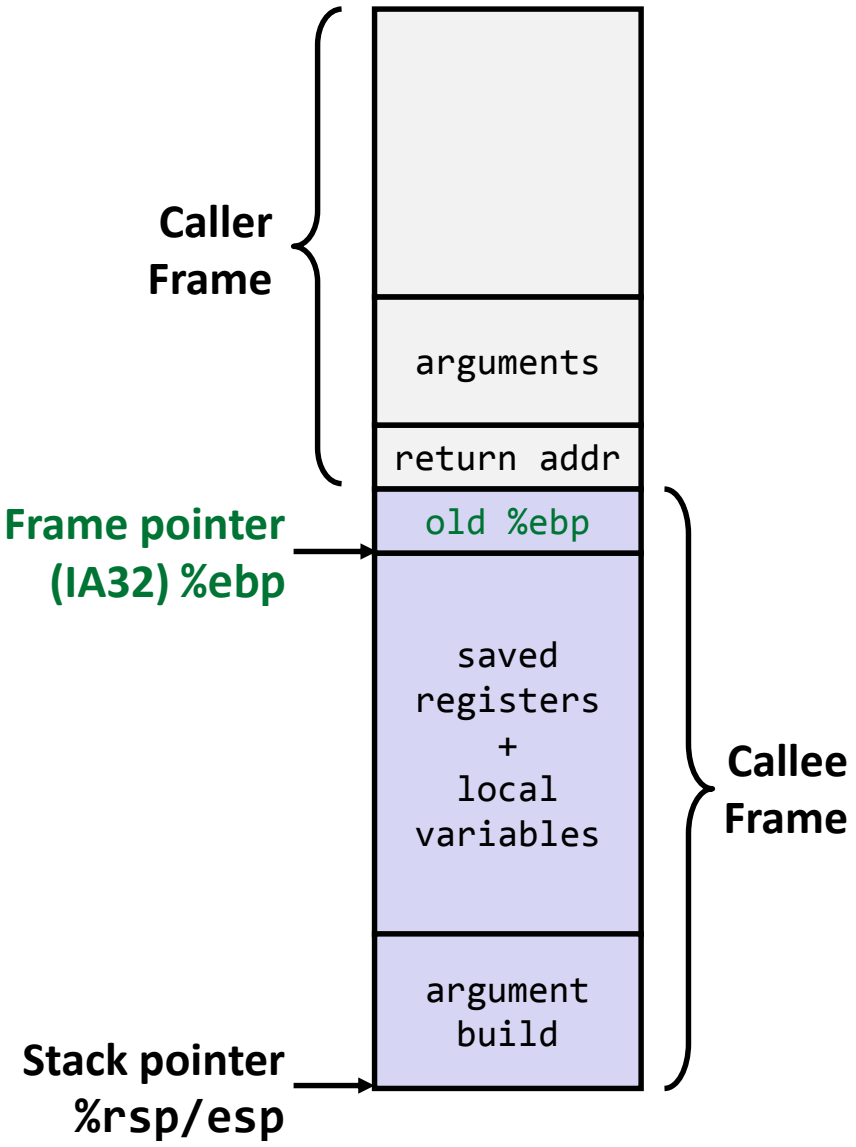
Final part of `add3`

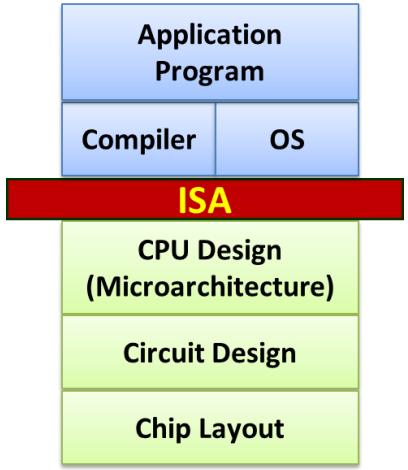
```
  movl  -4(%ebp), %eax  # ret. val=localx
  leave
  ret
```



Procedure Summary

- Important Points
  - Stack is the right data structure for procedure call / return
    - If P calls Q, then Q returns before P
- Recursion (& mutual recursion) handled by normal calling conventions
  - Can safely store values in local stack frame and in callee-saved registers
  - Put function arguments at top of stack
  - Result return in `%rax/eax`
- Pointers are addresses of values
  - On stack or global





Module Summary

Processor Architecture and the ISA

- **The Instruction Set Architecture (ISA) defines the public interface of a processor**
  - Operations, operands, instruction encoding, memory addressing, etc.

- **The x86 Instruction Set Architecture**
  - 16 general-purpose registers in x86_64, 8 in x86
  - Arithmetic/logic/bitwise, memory, and control operations
    - Two-operand format: add b, a → a += b
  - Operand types (register, memory, immediate)
  - By default, data types are aligned at addresses divisible by the size of the data type
  - Composite data structures are simply a collection of natively supported data types
  - Calling conventions (x86: via the stack, x86_64: first 6 via registers, then stack)