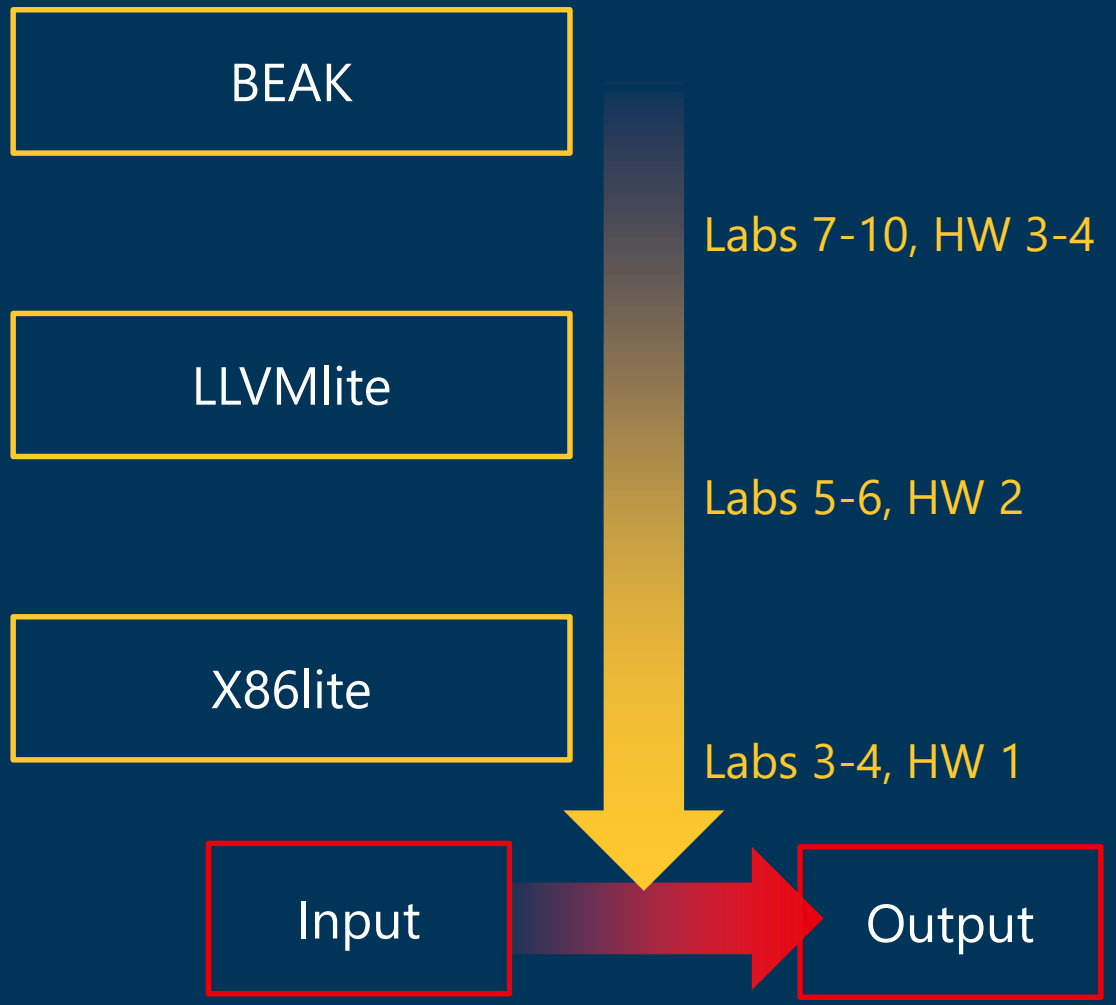
Compiler Construction: X86lite

Course structure



Borrowed liberally from UPenn CIS 341

THE X86 ARCHITECTURE

History

1978: Intel 8086

- Introduced the x86 architecture

1985: Intel 80386

- First 32-bit x86 processor

1995: Intel Pentium Pro

- µ-op translation, speculative execution, &c.

2003: AMD Athlon 64

- First 64-bit x86 processor

History

1978: Intel 8086

1985: Intel 80386

1995: Intel Pentium Pro

2003: AMD Athlon 64

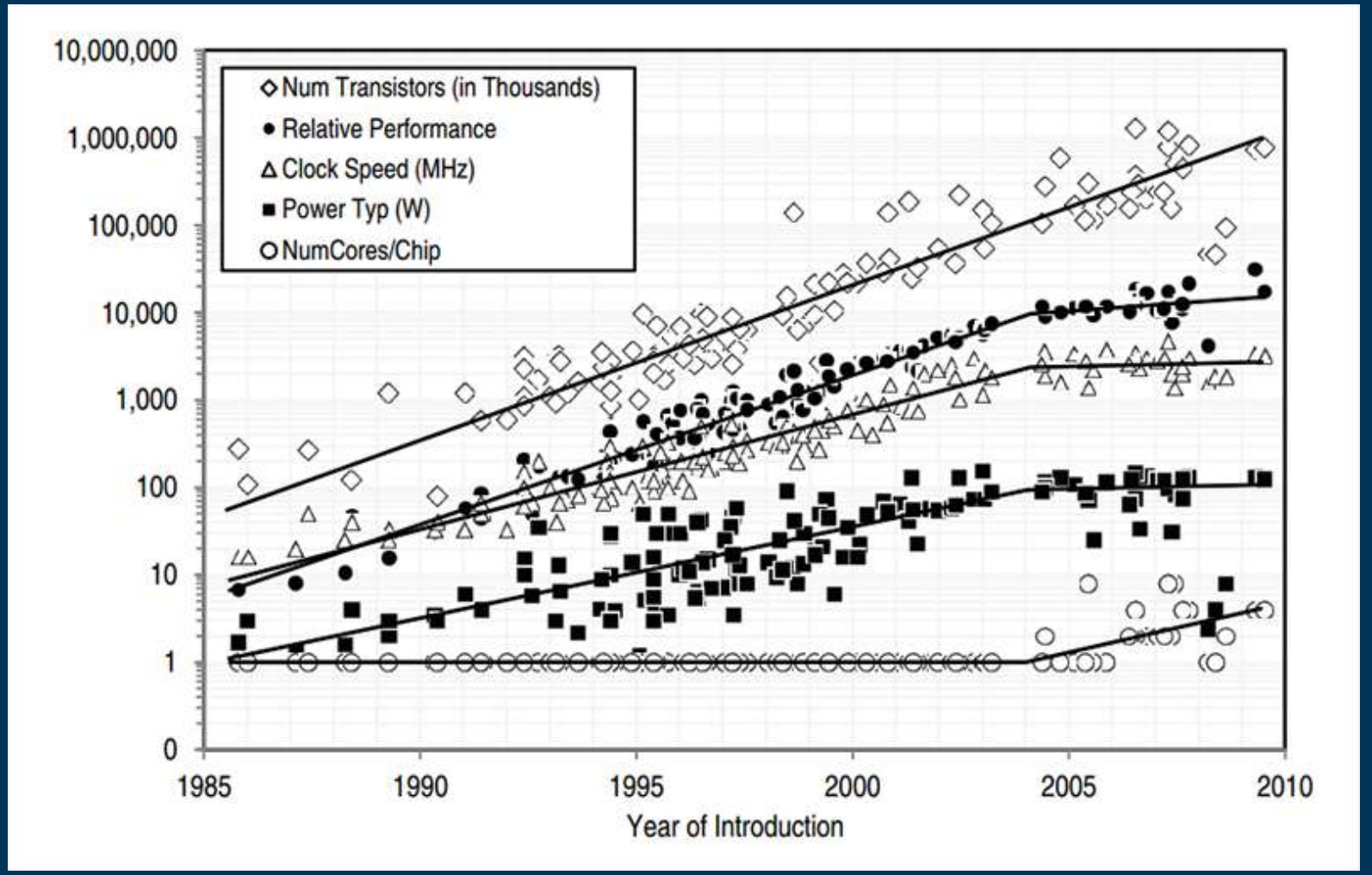
2006: Intel Core 2

- Intel accepts the demise of single core processors

2010: AMD FX

- First consumer 8-core processor

Moore's law


10,000,000
1,000,000
100,000
10,000
1,000
100
10
1
0
Num Transistors (in Thousands)
Relative Performance
Clock Speed (MHz)
Power Typ (W)
NumCores/Chip
1985
1990
1995
2000
2005
2010
Year of Introduction

Features of X86 assembly

- 8-, 16-, 32-, 64-bit values, varying-precision floating point, vectors
- CISC: Intel 64 and IA 32 architectures have a large number of functions
- Binary encoding: instructions range in size from 1 byte to 17 bytes
- Design constrained by backwards compatibility
- Complexity makes simple decisions hard: whole books just about optimizations in instruction selection

Features of X86lite assembly

X86:

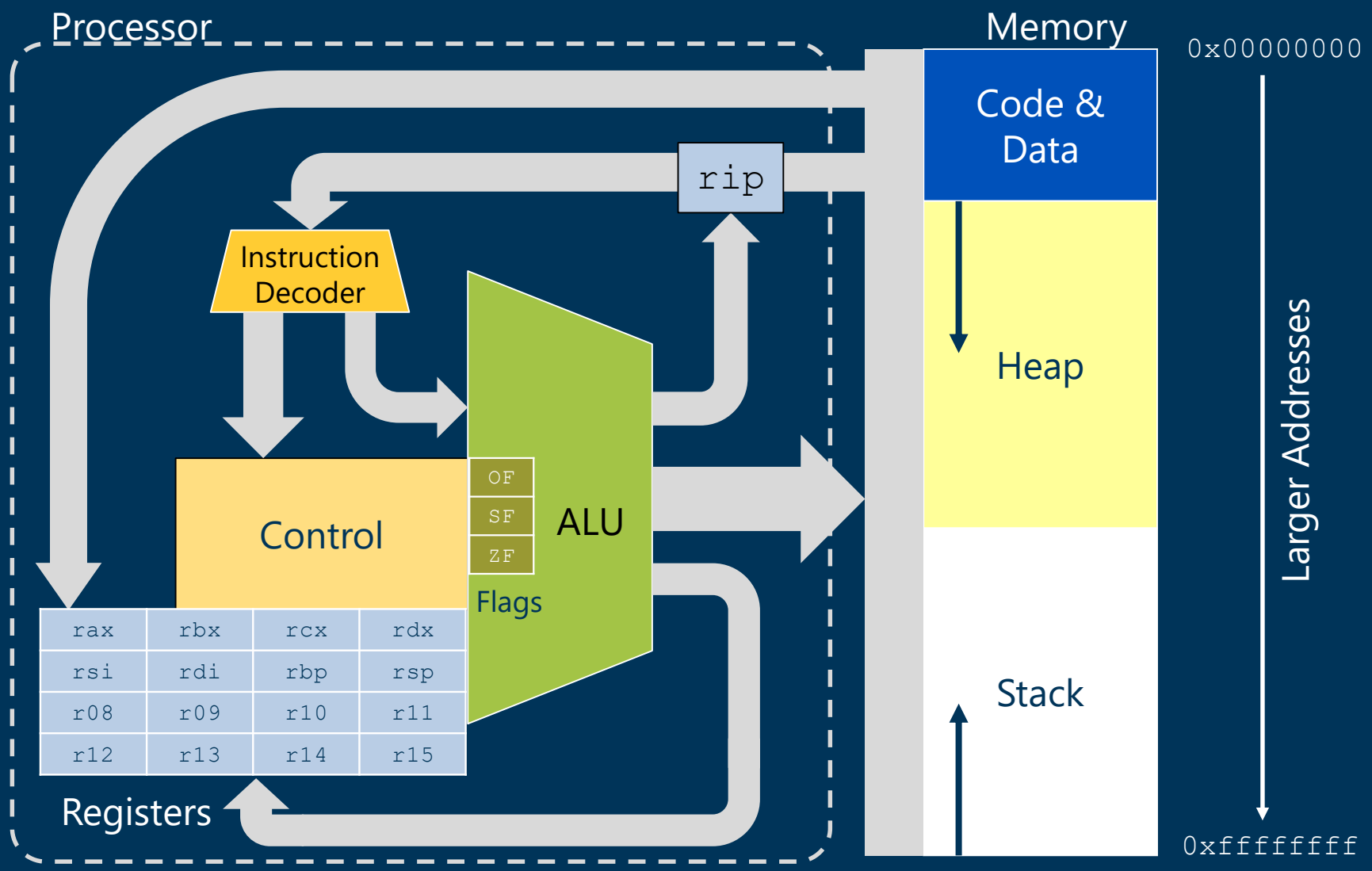
- 8-, 16-, 32-, 64-bit values
- Large number of functions
- Instructions range in size from 1 byte to 17 bytes
- Design constrained by backwards compatibility
- Complexity makes simple decisions hard

X86lite:

- 64-bit signed integers
- 20 operations
- Uniform instruction encoding
- No concerns about compatibility
- Complexity (mostly) removed

But still sufficient for general purpose computing

X86(lite) schematic



X86(lite) machine state: registers

"General purpose" registers

Name	*Purpose (maybe)*
`rax`	Accumulator
`rbx`	Base (pointer)
`rcx`	Counter (for strings and loops)
`rdx`	Data (for I/O)
`rsi`	String source (pointer)
`rdi`	String destination (pointer)
`rbp`	Base pointer (bottom of the stack)
`rsp`	Stack pointer (top of the stack)
`r08-r15`	General purpose

Special registers

Name	*Purpose*
rip	Instruction pointer
rflags	Conditions after last op

Our first instruction

`movq` *`src dst`*

- Copy from *`src`* into *`dst`*

- *`dst`* is treated as a location
  - Either a register or a memory address

- *`src`* is treated as a value
  - *Contents* of a register or memory address
  - Or a constant or label (called an *immediate*)

Our first instruction

`movq` *`src dst`*

- Copy from *`src`* into *`dst`*

- `movq $4, %rax`
  - Moves the 64 bit value `0…0100` into register `rax`
- `movq %rbx, %rax`
  - Moves the *contents* of register `rbx` into register `rax`
- `movq (%rbx), %rax`
  - Moves the contents of the memory location *pointed to* by register `rbx` into register `rax`

It's already gone complicated

AT&T syntax

- Source before destination
- Prefixes for immediates ($), registers (%)
- Mnemonic suffixes for sizes

```
movq $5, %rax
movl $5, %eax
movq $5, (%rax)
```

- Prevalent in Unix (derived) ecosystems

We'll stick to AT&T syntax in EECS665

- ource
- etermined by register name
- Size directives, sometimes

```
mov rax, 5
mov eax 5
mov dword ptr [rax], 5
```

- Used in Intel specifications/manuals
- Prevalent in Windows ecosystem

X86lite arithmetic

Instruction	*Schematic*	*Description*
`negq` *dst*	*dst* ← ~*dst*	Two's complement negation
`addq` *src dst*	*dst* ← *dst* + *src*	Addition
`subq` *src dst*	*dst* ← *dst* – *src*	Subtraction
`imulq` *src dst*	*dst* ← *dst* * *src*	Truncated 128-bit multiplication

  - The destination in `imulq` *must* be a register!

- `addq %rax, (%rbx)`
  - Memory at `rbx` gets `rax` + (memory at `rbx`)
- `imulq $4, %rax`
  - `rax` gets `4 * rax`

X86lite logical operators (the easy ones)

Instruction	*Schematic*	*Description*
`notq` *dst*	*dst* ← ¬*dst*	Bitwise negation
`andq` *src dst*	*dst* ← *src* & *dst*	Bitwise AND
`orq` *src dst*	*dst* ← *src* | *dst*	Bitwise OR
`xorq` *src dst*	*dst* ← *src* ⊕ *dst*	Bitwise XOR

X86lite logical operators (the hard ones)

Instruction	*Schematic*	*Description*
`shlq` *imm dst*	*dst* ← *dst* << *amt*	Logical (or arithmetic) shift left
`shrq` *imm dst*	*dst* ← *dst* >> *amt*	Logical shift right
`sarq` *imm dst*	*dst* ← *dst* >>> *amt*	Arithmetic shift right

- `movb $-8, %al` *%al = 1111 1000*
- `sarb $1, %al` *%al = 1111 1100*
- `shrb $1, %al` *%al = 0111 1110*

X86(lite) operands

So what are `src` and `dst` really?

- *Immediate* values: 64-bit literal signed integers
- *Labels*: names for addresses (resolved before execution by assembler/linker/loader)
- *Registers*: of the general-purpose variety
- *Indirect* references: memory

X86(lite) operands: indirect references

What does "memory" mean?

- *Base*: a machine address, stored in a register
- *Index*scale*: a variable offset from the base register
- *Displacement*: a constant offset from the (indexed) base register

AT&T syntax	*Intel syntax*
`(%rax)`	`[rax]`
`-4(%rax)`	`[rax-4]`
`(%rax, %rcx, 4)`	`[rax+rcx*4]`
`12(%rax, %rcx, 4)`	`[rax+rcx*4+12]`

X86lite operands: indirect references

What does "memory" mean?

- *Base*: a machine address, stored in a register
- ~~*Index*scale*: a variable offset from the base register~~
- *Displacement*: a constant offset from the (indexed)

X86lite doesn't have index*scale addressing

AT&T syntax	*Intel syntax*
`(%rax)`	`[rax]`
`-4(%rax)`	`[rax-4]`
~~`(%rax, %rcx, 4)`~~	~~`[rax+rcx*4]`~~
~~`12(%rax, %rcx, 4)`~~	~~`[rax+rcx*4+12]`~~

X86(lite): indirect references

Instruction	*Schematic*	*Description*
`leaq` *src dst*	*dst* ← addr(*src*)	Load effective address

- Gives access to computation of indirect addresses
  - src must be an indirect reference

- `leaq -4(%ebx), %eax` `eax ← ebx-4`
- `leaq 4(%ebx, %ecx, 12), %eax` `eax ← ebx+ecx*12+4`

X86lite condition flags

X86(lite) instructions set flags as a side effect

Name	*Mnemonic*	*Meaning (set if…)*
OF	Overflow	Result doesn't fit in 64 bits
SF	Sign	Result was negative
ZF	Zero	Result was zero

X86 condition flags: comparisons

Flags can be used to define comparison.

Condition	*Description*	*Flags after `src1-src2`*
E	Equality	ZF
NE	Inequality	¬ZF
G	Greater than	(¬ZF&¬SF)⊕OF
L	Less than	SF⊕OF
GE	Greater than or equal	¬SF⊕OF
LE	Less than or equal	(SF⊕OF) | ZF

Conditional instructions (part 1)

Instruction	*Schematic*	*Description*
`cmpq` *src1 src2*		Set flags based on *src2-src1*
`set`*CC dst*	*dst* ← 1 if *CC*, 0 otherwise	*dst* set based on given condition code

```
movq $4, %rbx
movq $5, %rcx
cmpq %rbx, %rcx
setg %rax
```

%rbx = … 0100
%rcx = … 0101
of = 0, zf = 0, sf = 0
%rax = … 0001

Code blocks and labels

- X86 assembly is organized into labeled blocks
- Labels indicates jump targets (either through conditionals or function calls)
- Labels are translated away by the linker and loader
- Designated label to start execution

```
_factorial:
    pushl  %ebp
    movl   %esp, %ebp
    subl   $8, %esp
    movl   8(%ebp), %eax
    movl   %eax, -4(%ebp)
    movl   $1, -8(%ebp)
LBB0_1:
    cmpl   $0, -4(%ebp)
    jle    LBB0_3
    movl   -8(%ebp), %eax
    imull  -4(%ebp), %eax
    movl   %eax, -8(%ebp)
    movl   -4(%ebp), %eax
    subl   $1, %eax
    movl   %eax, -4(%ebp)
    jmp    LBB0_1
LBB0_3:
    movl   -8(%ebp), %eax
    addl   $8, %esp
    popl   %ebp
    retl
```

Conditional instructions (part 2)

Instruction	*Schematic*	*Description*
`cmpq` *src1 src2*		Compare src1 and src2
`set`*CC dst*	*dst* ← 1 if *CC*, 0 otherwise	*dst* set based on given condition code
`jmp` *src*	`rip` ← *src*	Jumps to *src*
`j`*CC dst*	`rip` ← *dst* if *CC*	Jump if condition

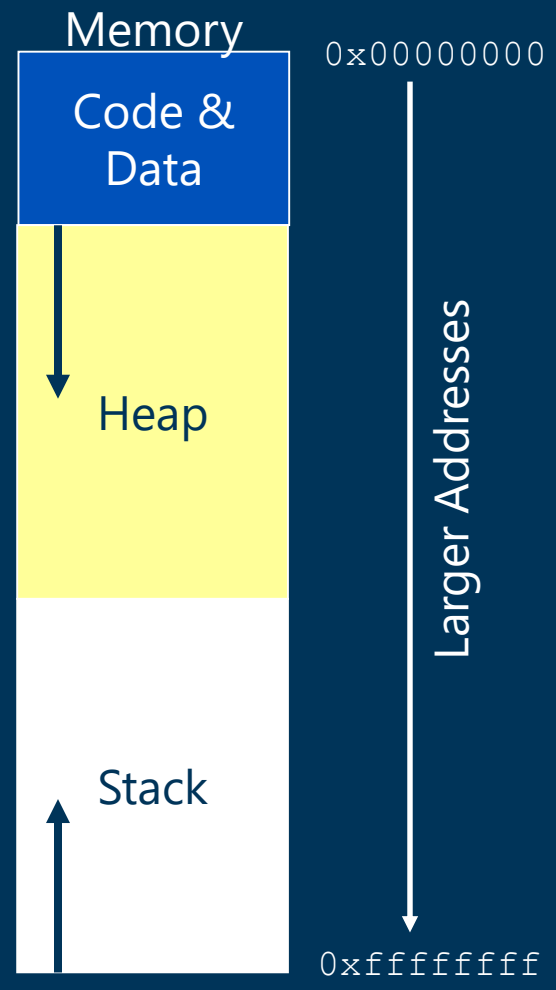
…	*%ebx = 4, %ecx = 5*
`cmpq %rbx, %rcx`	*of = 0, zf = 0, sf = 0*
`jg _dest`	*rip = _dest*

The X86lite/C memory model

X86lite assumes 2^{64} bytes of memory.

Conventionally divided into three parts:

- The code & data (or "text") segment stores compiled code, constant data, &c.



The X86lite/C memory model

The heap:

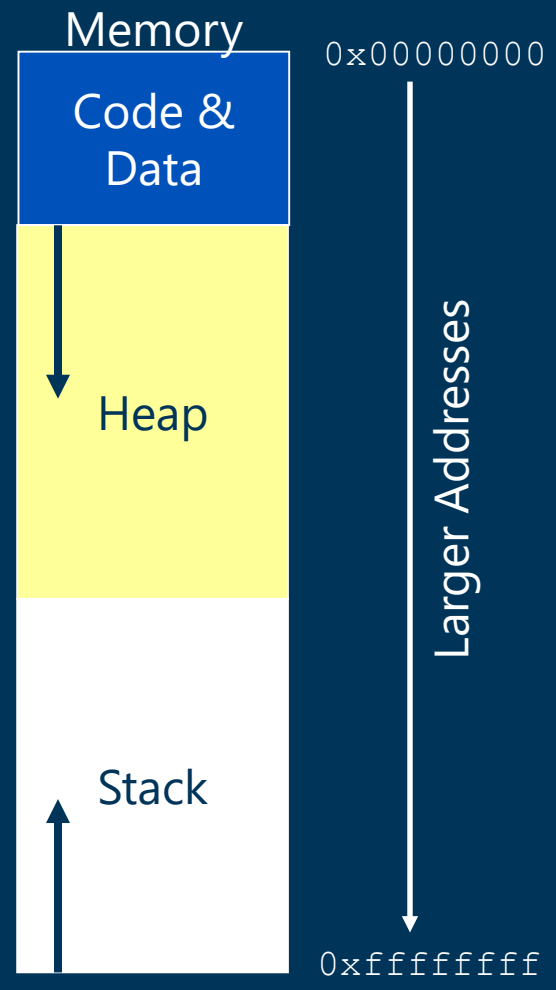
- Starts low in memory and grows upwards.
- Contains dynamically allocated objects

Heap management in C:

- Objects allocated by “malloc”
- Deallocated via “free”

Heap management in Haskell:

- “Bump” allocation
- Deallocation via garbage collection

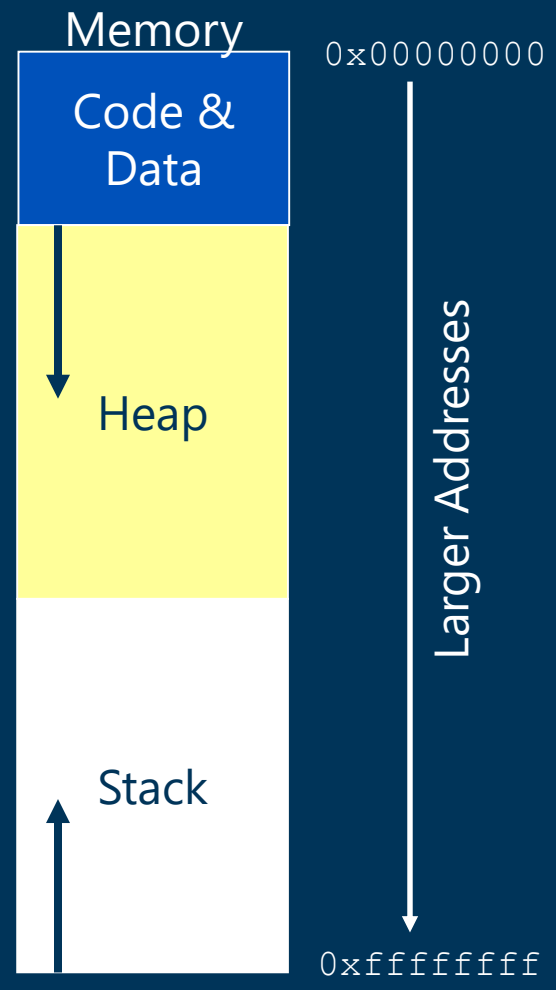


The X86lite/C memory model

The stack:

- Starts high, grows downwards
  - Register `rsp` points to the "top" of the stack, `rbp` points to the bottom of the current stack frame.
- Stores function arguments, return addresses, and local variables

Instruction	*Schematic*
`pushq` *src*	*rsp* ← *rsp* – 8; Mem[*rsp*] ← *src*
`popq` *dst*	*dst* ← Mem[*rsp*]; *rsp* ← *rsp* + 8



Call, and return

Instruction	*Schematic*	*Description*
`callq` *src*	`push rip` `rip ←` *src*	Procedure call
`retq`	`pop rip`	Return from procedure

Procedure calls are implemented using the stack:

- To call a procedure: push the current rip to the stack, then jump
- To return: jump to the address on top of the stack

Calling conventions

Implement function calls in terms of `callq`/`ret`: need to specify

- Locations for function arguments
- Treatment of registers:
  - "Caller-save": freely usable by called code; caller responsible for saving values
  - "Callee-save": called code responsible for restoring values at call
- Protocol for stack-allocated arguments
  - Caller cleans
  - Callee cleans: variable argument functions harder

32-bit calling conventions

- EAX, ECX, EDX are caller-save. All others are callee-save
- Return value in EAX, or in EAX and EDX

`cdecl`:

- Arguments passed right-to-left
- Caller cleans parameters after return
- Allows variable-length argument lists
- Standard in stand-alone C programs and Unix-y operating systems

`pascal`:

- Arguments passed right-to-left
- Callee cleans parameters before return
- "Fractionally faster" (in 1985)
- Used in Win32 API calls

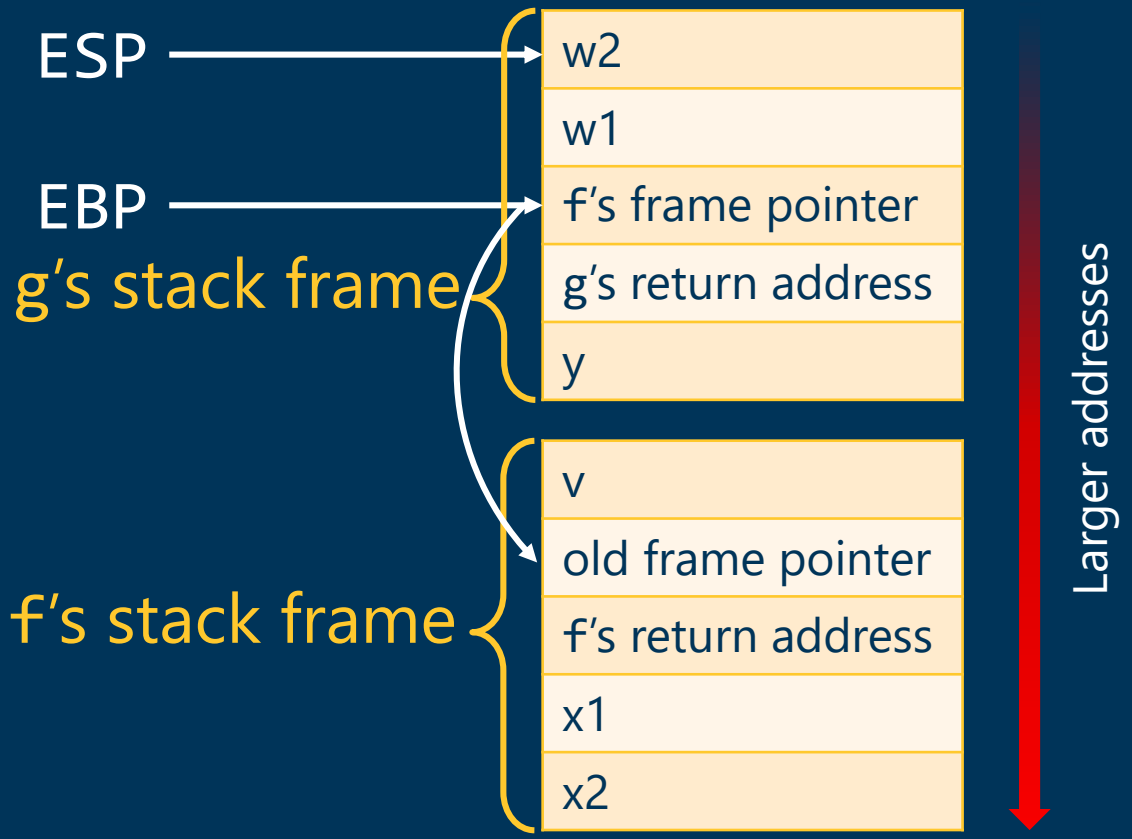
We're only going to use cdecl-like conventions

32-bit call stacks

Scenario: `f(x1, x2)` with local variable v calls `g(y)` with local variables `w1`, `w2`

Stack frame (at EBP) contains:

- Local variables (*above* EBP)
- Callee-save registers (*above* EBP)
- Return address (*below* EBP)
- Parameters (*below* EBP)



32-bit function calls: caller protocol

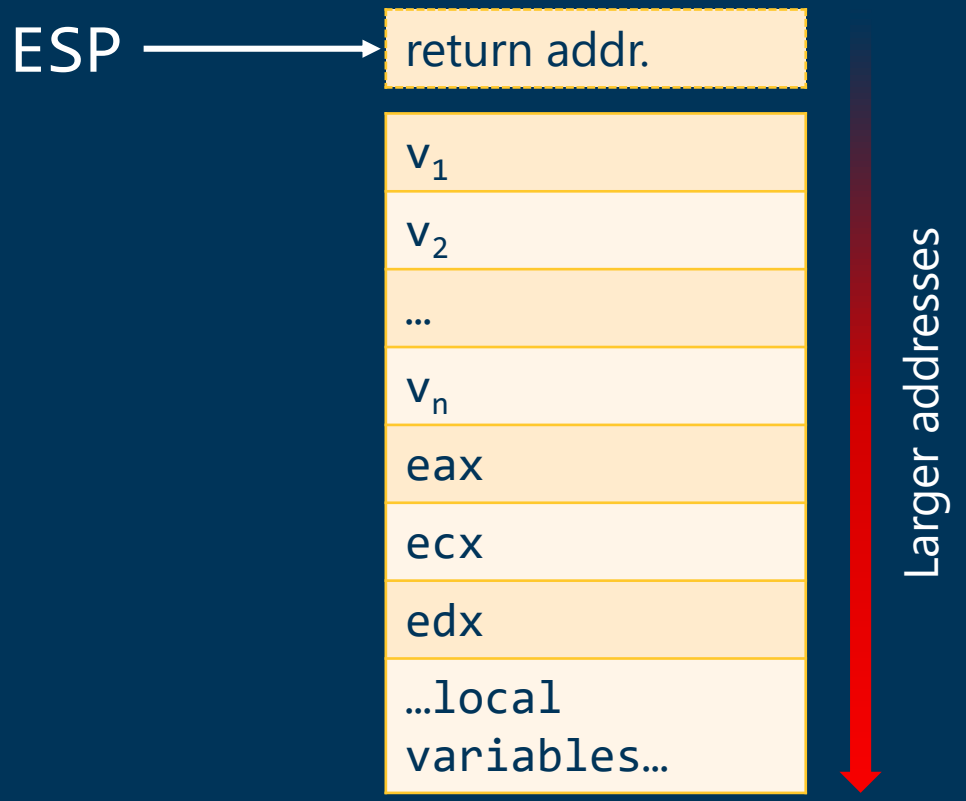
To call function `f(`e_1`,`e_2`,…,`e_n`)`

1. Save caller-save registers
2. Push values of e_n…e_1 onto the stack
3. `callq f`

After `f` returns:

1. Clean values of e_n…e_1 from the stack
2. Restore caller-saved registers

Note: return value in `eax`, `edx`.



32-bit function calls: caller protocol

To call function $f(e_1, e_2, e_3)$:

```
push %edx
push %ecx
push -4(%ebp)
push $42
push %ebx
call _f
addl $12, %esp
pop %ecx
pop %edx
```

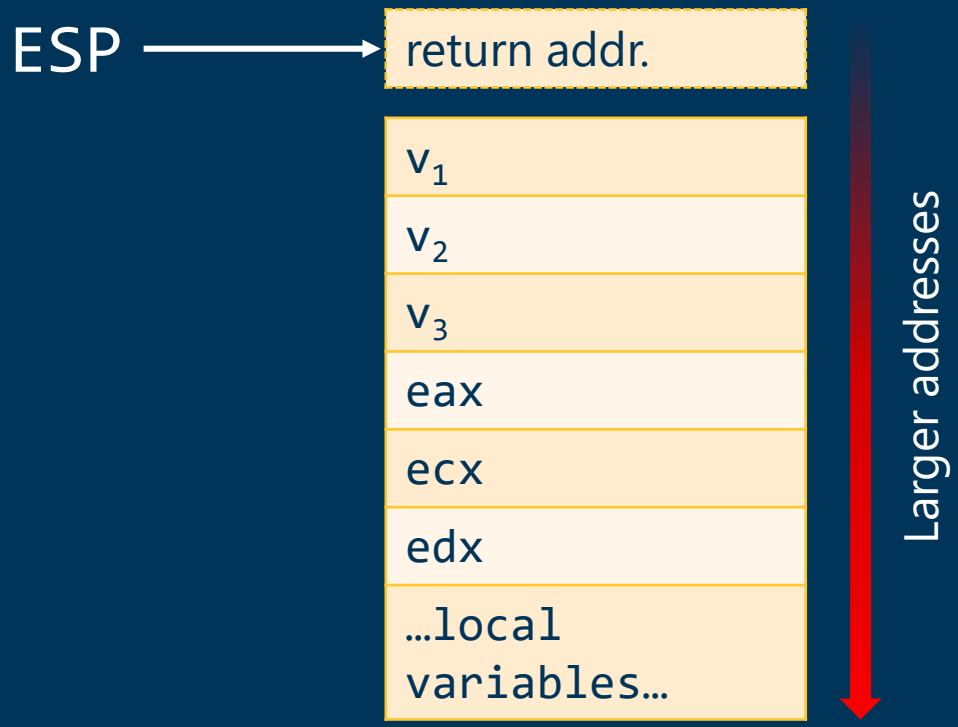
Save registers

Arguments

Function call

Clean arguments

Restore registers



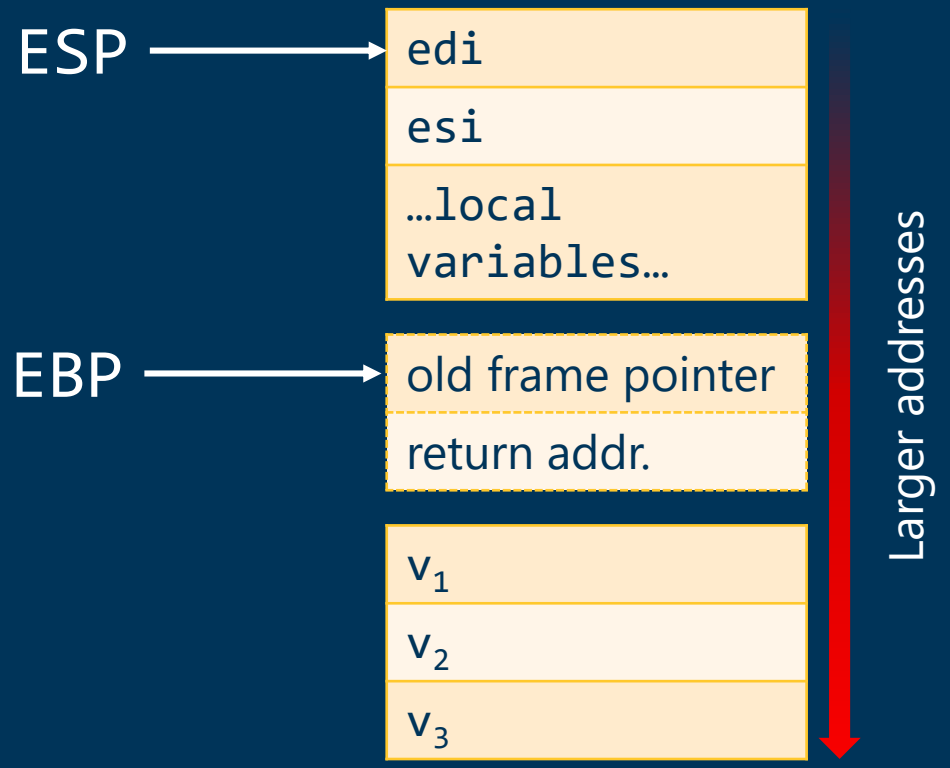
32-bit function calls: callee protocol

To implement function f(e1,…,en)

1. Save old frame pointer
2. Set up new frame pointer
3. Allocate space for local variables
4. Save callee-save registers

To return from f:

1. Restore callee-save registers
2. Deallocate local variables
3. Restore frame pointer
4. Return

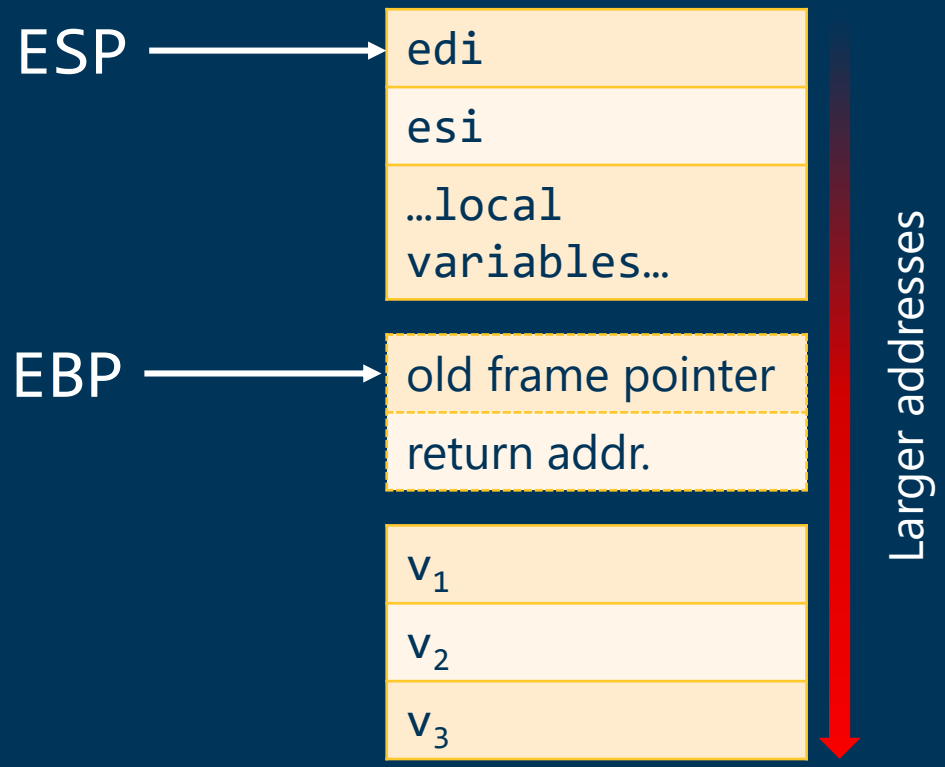


32-bit function calls: callee protocol

To implement function f(x1,…,xn):

```
_f:
    push %ebp
    mov %esp, %ebp
    sub $12, %esp
    push %esi
    push %edi
    …
    pop %edi
    pop %esi
    mov %ebp, %esp
    pop %ebp
    ret
```

- Frame setup
- Local variables
- Save registers
- Function body
- Restore registers
- Local variables
- Old frame pointer
- Return



64-bit function calls

- Callee-save: rbp, rbx, r12-r15
- `Return value in rax

- Parameters
  - 1-6: rdi, rsi, rdx, rcx, r8, r9
  - 7+: on the stack

- 128 byte “red zone”