Lecture 3

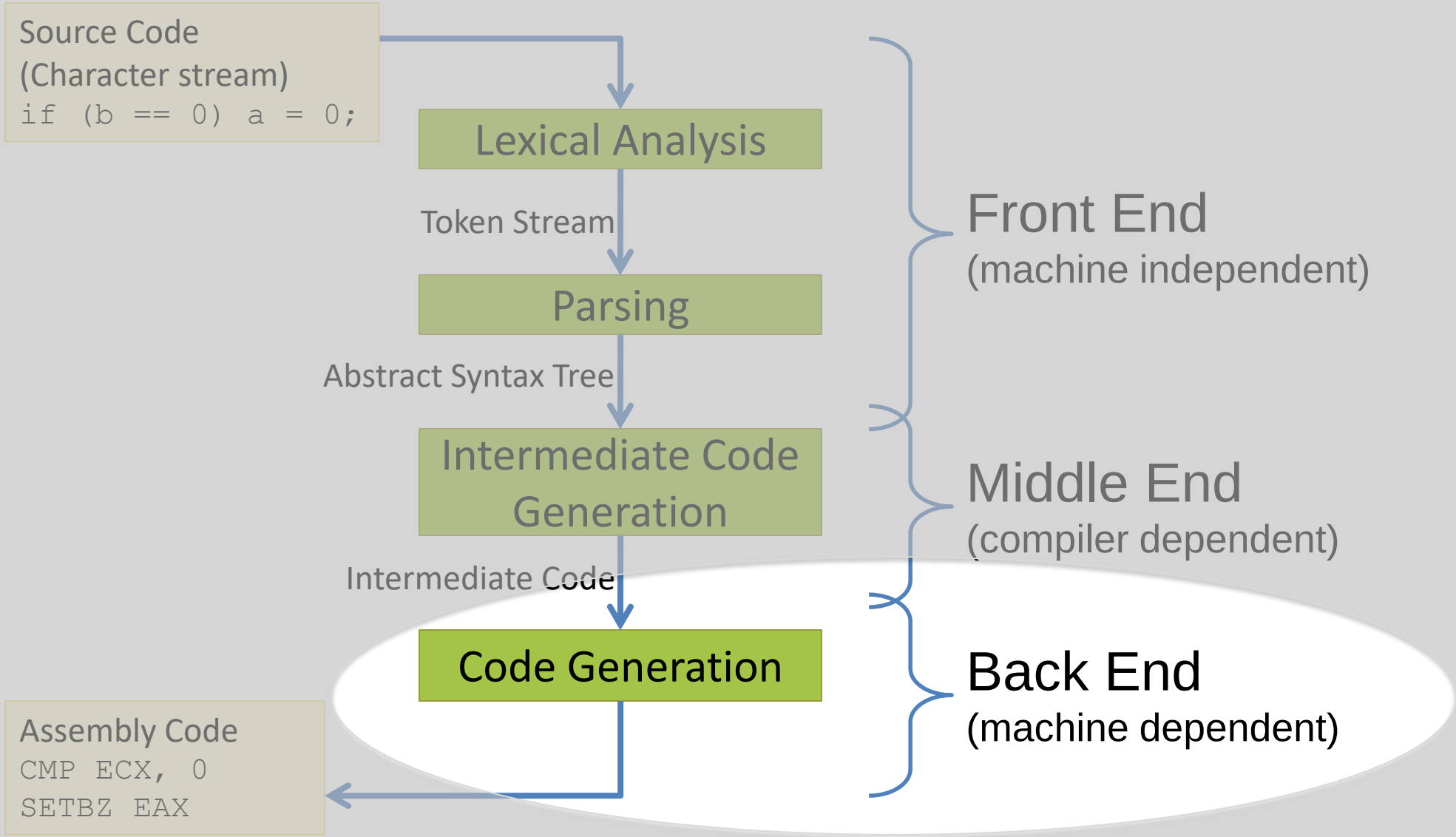
COMPILER DESIGN

Announcements

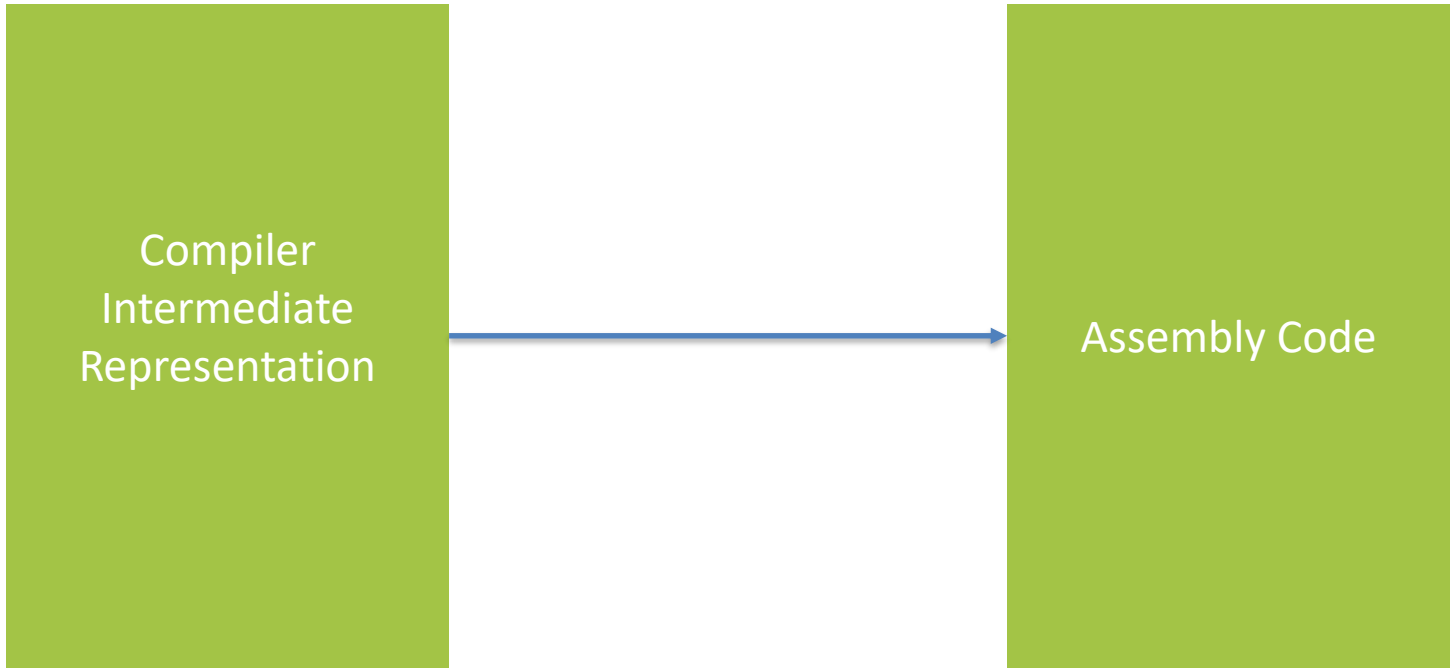
- HW1: Hellocaml
  - Due *Thursday, 1 Oct. at 23:59*

- HW2: X86lite
  - Will be available next week on our course Moodle
  - Simulator / Loader for x86 assembly subset

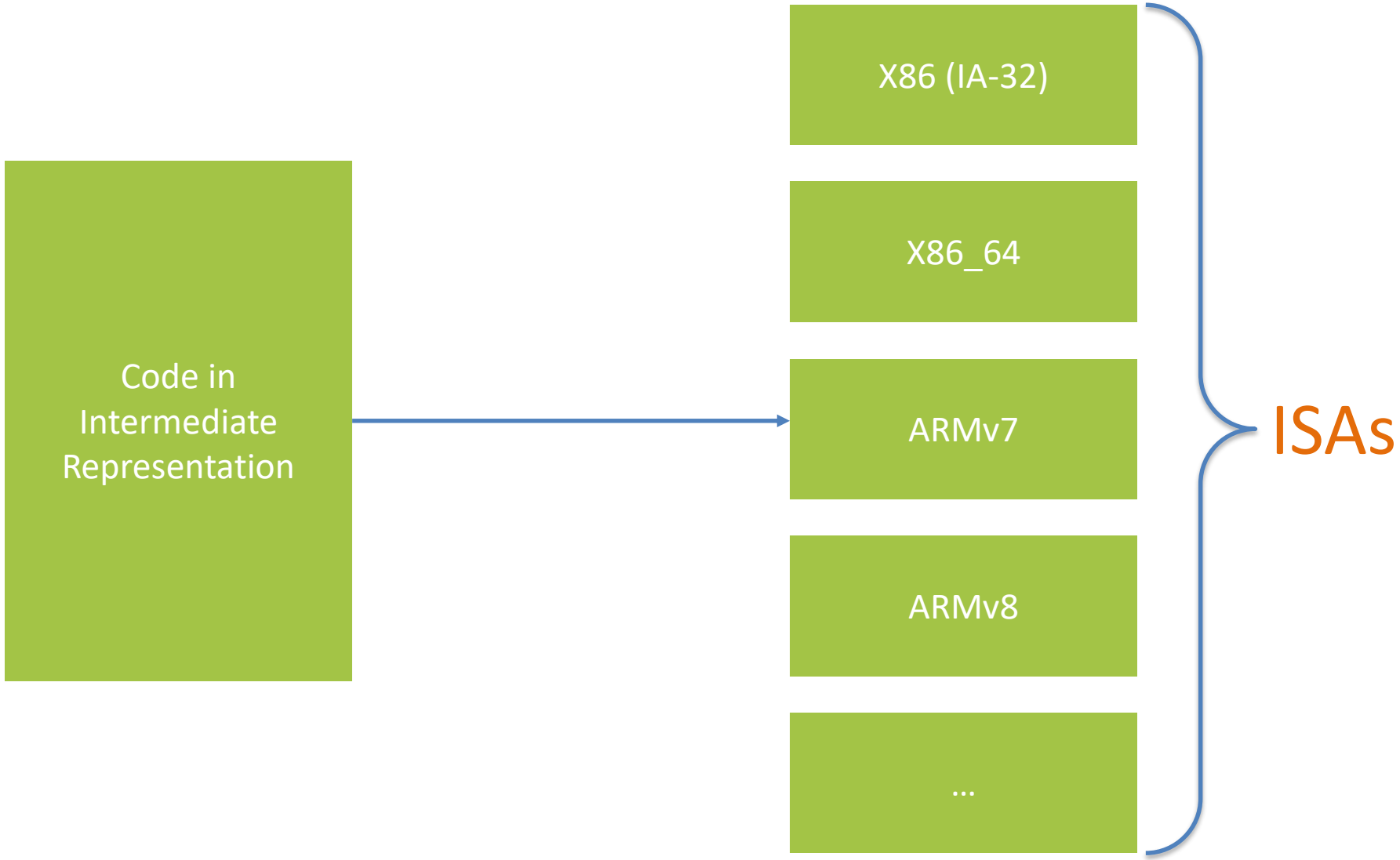
(Simplified) Compiler Structure



Back End



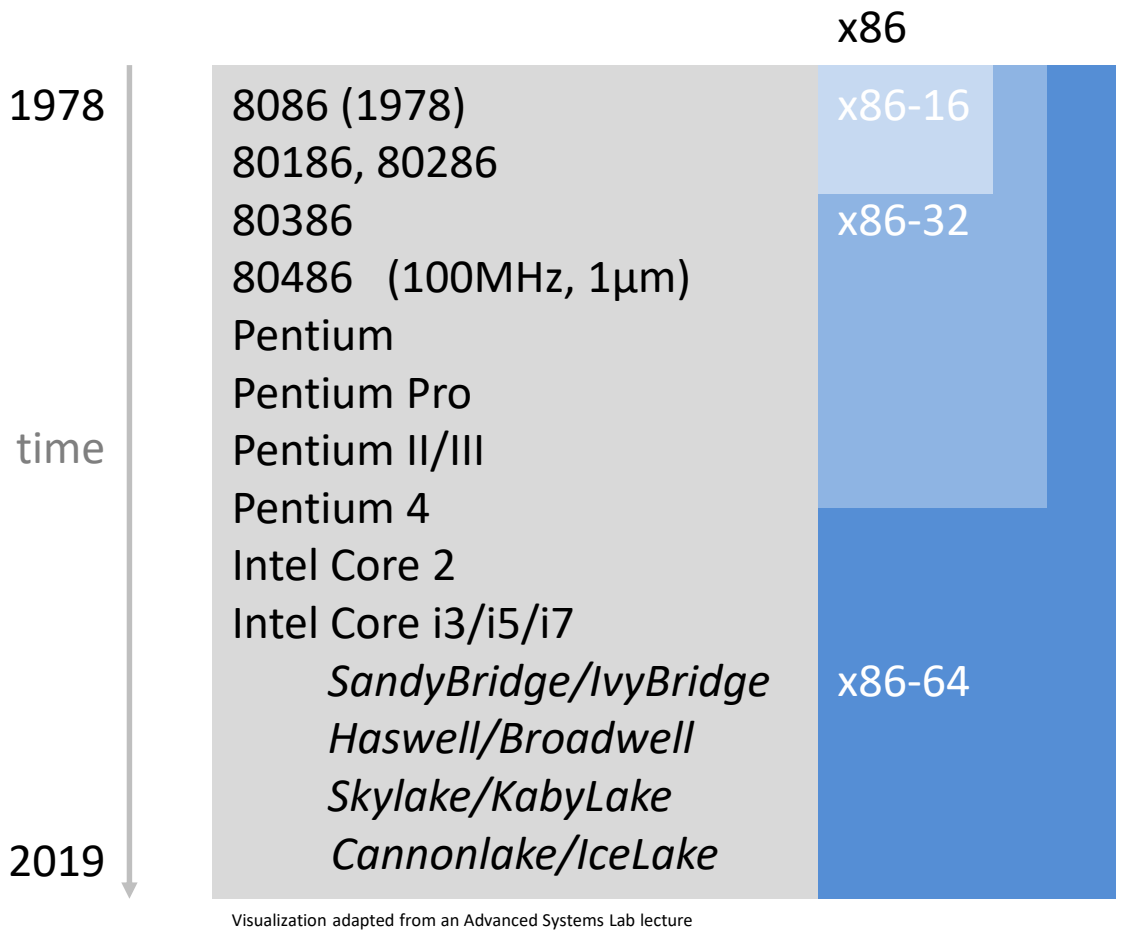
Multiple Back Ends



Instruction Set Architecture

The target architecture

X86LITE



Visualization adapted from an Advanced Systems Lab lecture

- Binary Compatibility (old code runs on new hardware)
- Complex ISA (1500+ instructions)
- Multiple extensions
- Non-Intel implementations (e.g., AMD, Via)

X86 vs. X86lite

X86 assembly is *very* complicated!

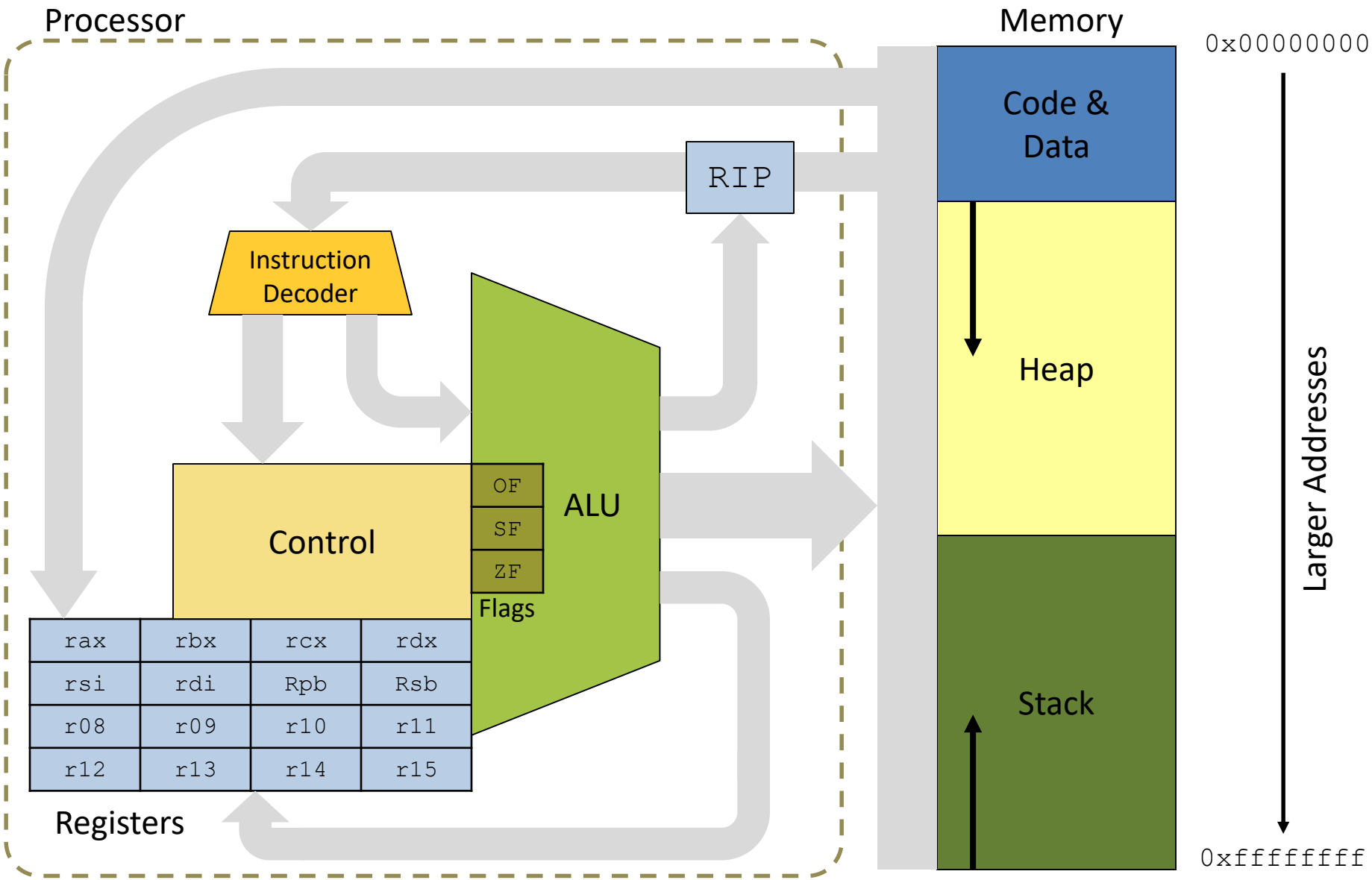
- 8-, 16-, 32-, 64-bit values + floating point, etc.
- Intel 64 and IA 32 architectures have a *huge* number of functions
- "CISC" complex instructions
- Machine code: instructions range in size from 1 byte to 17 bytes
- Lots of hold-over design decisions for backward compatibility
- Hard to understand, there is a large book about optimizations at just the instruction-selection level[1]

X86lite is a *very* simple subset of X86

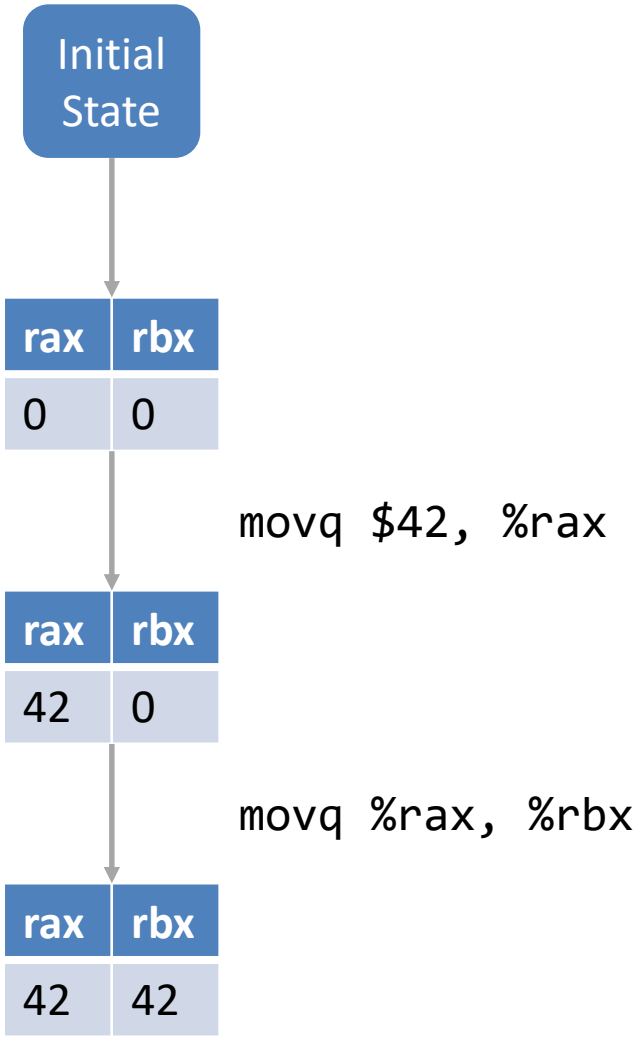
- Only 64-bit signed integers (no floating point, no 16-bit, no …)
- Only about 20 instructions
- Sufficient as a target language for general-purpose computing

[1]Intel® 64 and IA-32 Architectures Software Developer's Manual

X86 Schematic



Simplest instruction: `mov`



- `movq SRC, DEST`
- Here, DEST and SRC are *operands*
- DEST is treated as a *location*
  - A location can be a register or a memory address
- SRC is treated as a *value*
  - A value is the *contents* of a register or memory address
  - A value can also be an *immediate* (constant) or a label

A Note About Instruction Syntax

AT&T notation: source *before* destination

- Prevalent in the Unix ecosystems
- Immediate values prefixed with '`$`'
- Registers prefixed with '`%`'
- Mnemonic suffixes: `movq` vs. `mov`
  - `q` = quadword (4 words)
  - `l` = long (2 words)
  - `w` = word (16-bit)
  - `b` = byte (8-bit)

```
movq $5, %rax

movl $5, %eax
```

Intel notation: destination *before* source

- Used in the Intel specification / manuals
- ...e Windows ecosystem
- Instruction variant determined by register name

```
mov rax, 5

mov eax, 5
```

Note: X86lite uses the AT&T notation and the 64-bit only version of the instructions and registers

X86 Operands

Type	Description	Example
Imm	64-bit literal signed integer "immediate"	`movq $4, %rax`
Lbl	a "label" representing a machine address, the assembler/linker/loader resolves labels	callq **FOO**
Reg	One of the 16 registers, the value of a register is its contents	`movq %rbx, %rax`
Ind	machine address: [base:Reg][index:Reg][disp:int32] (base + index*8 + disp)	`movq 12(%rax, %rcx), %rbx`

Arithmetic instructions

Instruction	Description	Example	Notes
`negq` DEST	2's complement negation	`negq %rax`	
`addq` SRC, DEST	DEST ← DEST + SRC	`addq %rbx, %rax`	
`subq` SRC, DEST	DEST ← DEST – SRC	`subq $4, %rsp`	
`imulq` SRC, Reg	Reg ← Reg * SRC (truncated 128-bit mult.)	imulq $2, %rax	Reg must be a register, not a memory address

```
movq $2, %rax
movq $3, %rbx
imulq %rbx, %rax
// rax = 6, rbx = 3
```

Logic/Bit Manipulation instructions

Instruction	Explanation	Example	Notes
`notq` DEST	logical negation	`notq %rax`	bitwise not
`andq` SRC, DEST	DEST ← DEST & SRC	`andq %rbx, %rax`	bitwise and
`orq` SRC, DEST	DEST ← DEST | SRC	`orq $4, %rsp`	bitwise or
`xorq` SRC, DEST	DEST ← DEST xor SRC	xorq $2, %rax	bitwise xor

Instruction	Explanation	Example	Notes
`sarq` Amt, DEST	DEST ← DEST >> Amt	`sarq $4, %rax`	arithmetic shift right
`shlq` Amt, DEST	DEST ← DEST <<< Amt	`shlq %rbx, %rax`	logical shift left
`shrq` Amt, DEST	DEST ← DEST >>> Amt	`shrq $1, %rsp`	logical shift right

X86lite State: Condition Flags & Codes

Some X86 instructions set flags as side effects

- `OF`: "overflow" set when the result is too big/small to fit in 64-bit reg.
- `SF`: "sign" set to the sign of the result (0=positive, 1 = negative)
- `ZF`: "zero" set when the result is 0

From these flags, we can define *Condition Codes*

To compare SRC1 and SRC2, compute SRC1 – SRC2 to set the flags

Code	Condition
e (equality)	ZF is set
ne (inequality)	(not ZF)
g (greater than)	(not ZF) and (SF = OF)
l (less than)	SF <> OF
ge (greater or equal)	(SF = OF)
le (less than or equal)	SF <> OF or ZF

Examples

```
movq $INTMAX, %rax
subq $1,      %rax
//rax=INTMAX-1
//OF=0, SF=0, ZF=0
```

```
movq $INTMIN, %rax
subq $-1,     %rax
//rax=INTMIN+1
//OF=0, SF=1, ZF=0
```

```
movq $INTMAX, %rax
subq $-1,     %rax
//rax=INTMIN (neg)
//OF=1, SF=1, ZF=0
```

```
movq $INTMIN, %rax
subq $1,      %rax
//rax=INTMAX
//OF=1, SF=0, ZF=0
```

Conditional Instructions

```
if (a == b){
    //something
} else {
    //somethingElse
}
//commonCode
```

Instruction	Description
cmpq SRC2, SRC1	Compute SRC1 – SRC2, set condition flags
setbCC DEST	DEST's lower byte ← if CC then 1 else 0
jCC SRC	rip ← if CC then SRC else fallthrough

```
    cmpq %rax, %rbx
    je   something

somethingElse:
    <instruction>
    …
    jmp commonCode

something:
    <instruction>
    …

commonCode:
    <instruction>
    …
```

Code Blocks & Labels

- X86 assembly code is organized into *labeled blocks*
- Labels indicate code locations that can be jump targets (either through conditional branch instructions or function calls).
- Labels are translated away by the linker and loader – instructions live in the heap in the "code segment".
- An X86 program begins executing at a designated code label (usually "main").

```
    cmpq %rax, %rbx
    je    something

somethingElse:
    <instruction>
    …
    jmp commonCode

something:
    <instruction>
    …

commonCode:
    <instruction>
    …
```

Jumps, Call and Return

Instruction	Description	Notes
jmp SRC	rip ← SRC	Jump to location in SRC
call SRC	Push rip; rip ← SRC (Call a procedure)	Push the program counter to the stack (decrementing rsp), and then jump to the machine instruction at the address given by SRC
ret	Pop into rip (Return from a procedure)	Pop the current top of the stack into rip (incrementing rsp). This instruction effectively jumps to the address at the top of the stack

```
bar:
    <instruction>
    …
    <instruction>
    ret

foo:
    <instruction>
    …
    <instruction>
    call bar
    …
```

X86Lite Addressing

- In general, there are three components of an indirect address
  - Base: a machine address stored in a register
  - Index : a variable offset from the base
  - Disp: a constant offset (displacement) from the base
- addr(ind) = Base + [Index * 8] + Disp
  - When used as a *location*, ind denotes the address addr(ind)
  - When used as a *value*, ind denotes Mem[addr(ind)], the contentsof the memory address
- Examples:

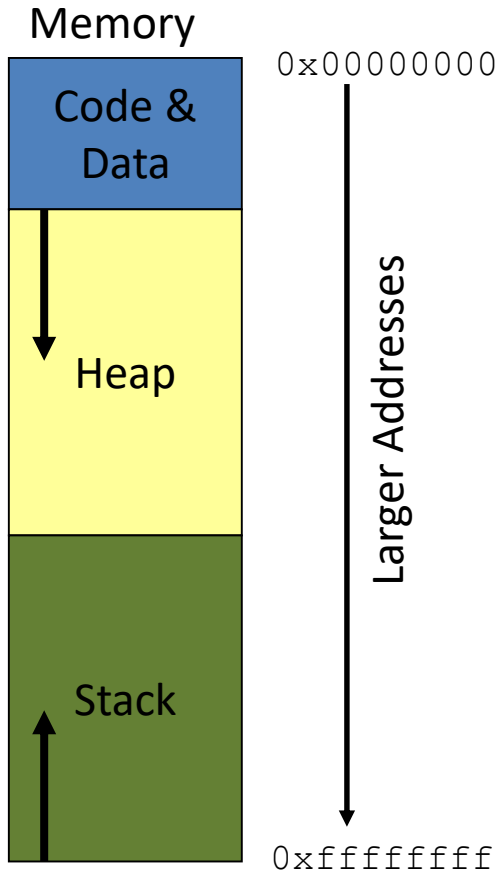
Expression	Address
-8(%rsp)	rsp - 8
(%rax, %rcx)	rax + 8*rcx
8(%rax, %rcx)	rax + 8*rcx + 8

- Note: Index cannot be `rsp`

```
// Array [0,42,2020]
// Array address 0xBEEF

movq %0xBEEF, %rax

movq %rax,   %rbx   //rbx=0xBEEF
movq (%rax), %rbx   //rbx=0
movq 16(%rax), %rbx //rbx=2020

movq $1, %rcx
movq (%rax, %rcx), %rbx   //rbx=42
movq 8(%rax, %rcx), %rbx  //rbx=2020
```

X86lite Memory Model

- The X86lite memory consists of 2^{64} bytes numbered `0x00000000` through `0xffffffff`.
- X86lite treats the memory as consisting of 64-bit (8-byte) quadwords.
- Therefore: legal X86lite memory addresses consist of 64-bit, quadword-aligned pointers.
  - All memory addresses are evenly divisible by 8
- `leaq` Ind, DEST DEST ← addr(Ind) loads a pointer into DEST
- By convention, the stack grows from high addresses to low addresses
- The register `rsp` points to the top of the stack
  - `pushq` SRC `rsp` ← `rsp` - `8`; Mem[`rsp`] ← SRC
  - `popq` DEST DEST ← Mem[`rsp`]; `rsp` ← `rsp` + `8`



DEMO: HANDCODING X86LITE

See file: x86.ml

IMPLEMENTING X86LITE