HARVARD
John A. Paulson
School of Engineering
and Applied Sciences

CS153: Compilers
Lecture 2: Assembly

Stephen Chong

https://www.seas.harvard.edu/courses/cs153

Contains content from lecture notes by Steve Zdancewic

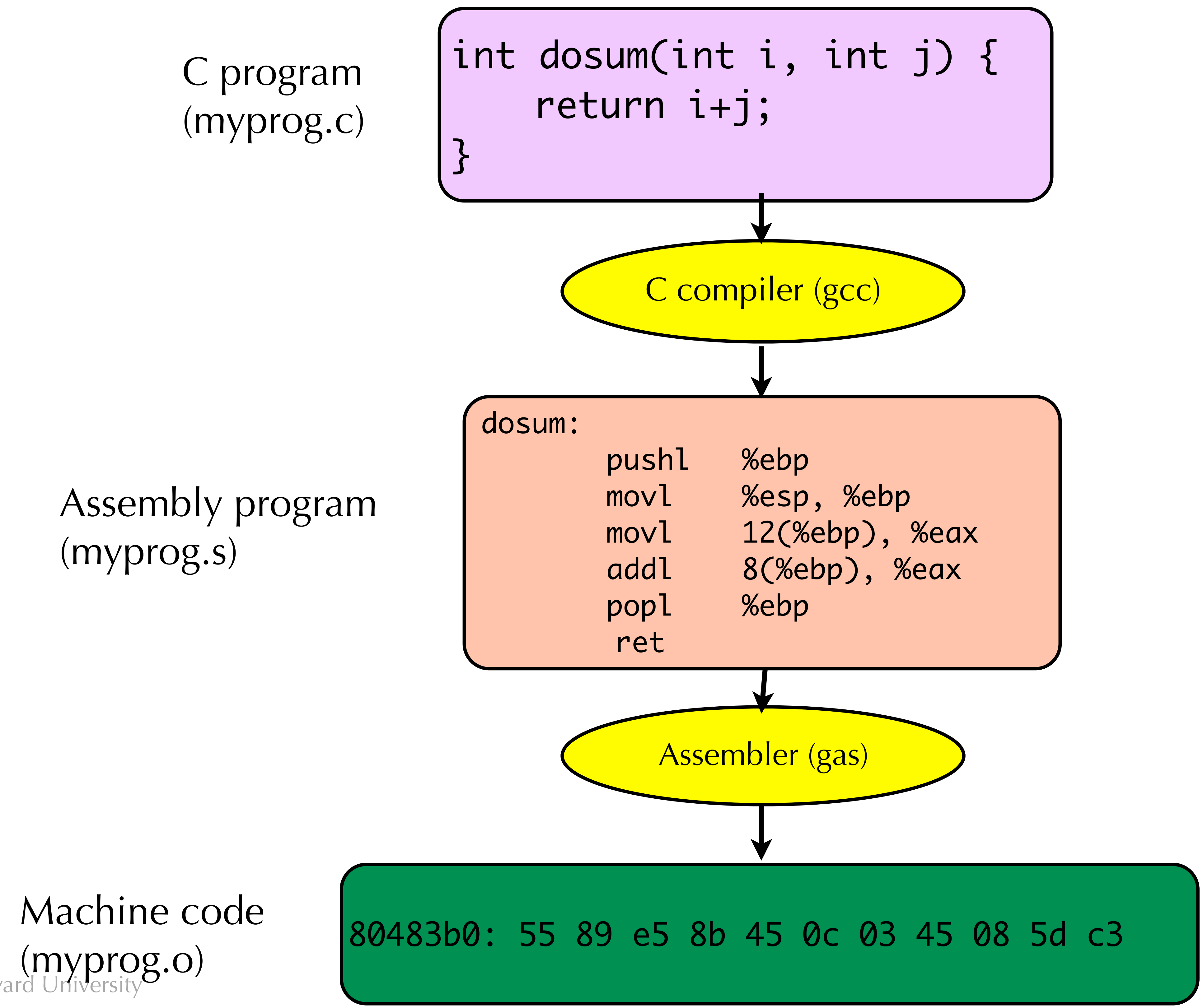
Announcements

- Name tags
- Device free seating
  - Right side of classroom (as facing front): no devices
  - Allow you to commit to being device-free/avoid devices
- College students registering for course: all good?
- Access to Gradescope: all students should have
  - Contact Prof Chong if you don't
- Homework 0 (Google form): please complete this week!
  - https://forms.gle/P65LytJYbKA5MzBj9
- Homework 1 (HellOCaml) out
  - Due Tuesday Sept 10

Today

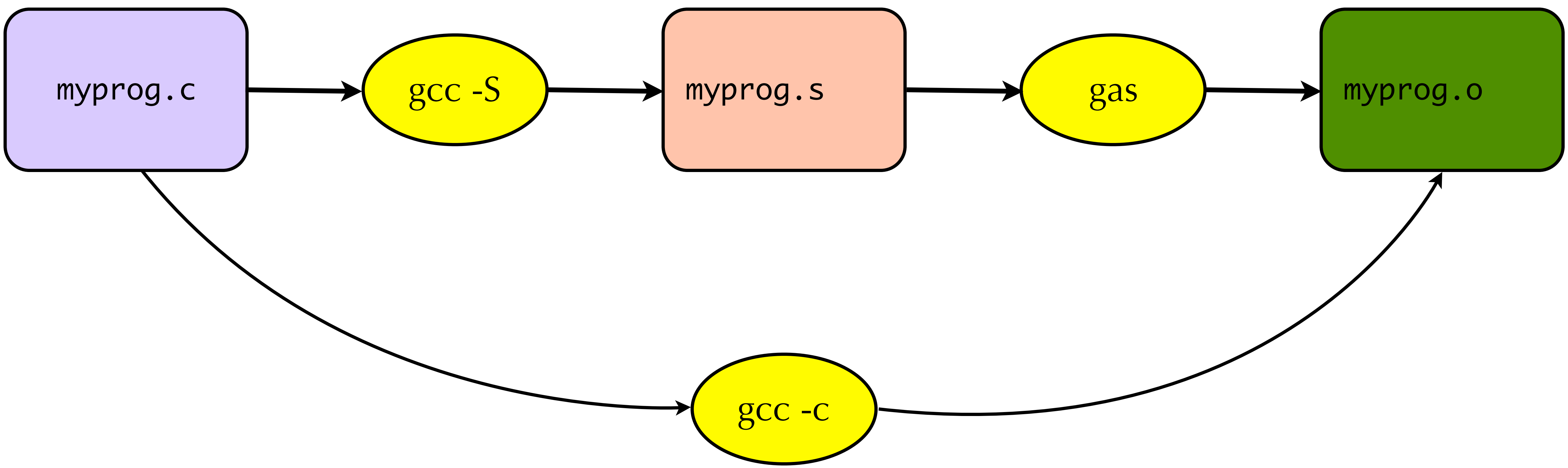
- Turning C into machine code
- Intel x86
- x86lite

Turning C into Machine Code



Skipping assembly language

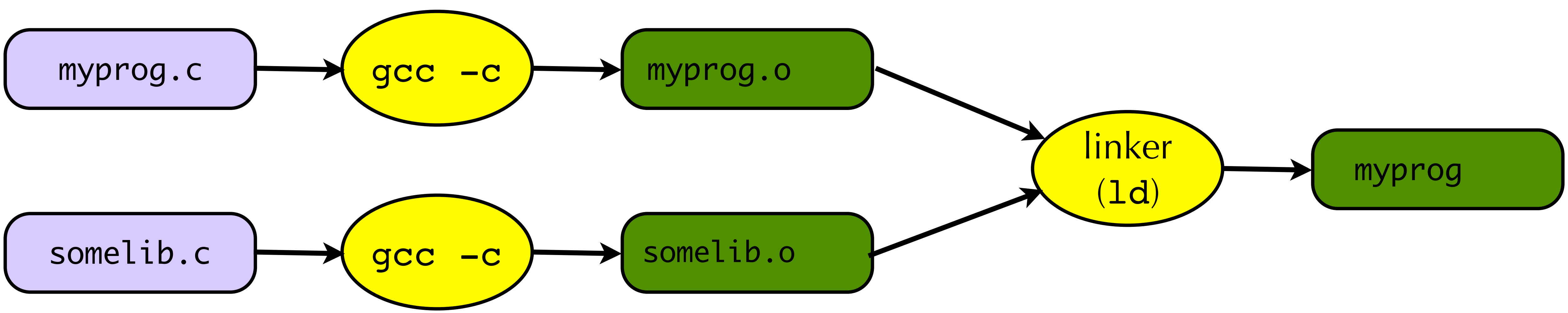
- Most C compilers generate machine code (object files) directly.
  - That is, without actually generating the human-readable assembly file.
  - Assembly language is mostly useful to people, not machines.



- Can generate assembly from C using "`gcc -S`"
  - And then compile to an object file by hand using "`gas`"

Object files and executables

- C source file (**`myprog.c`**) is compiled into an **object file** (**`myprog.o`**)
  - Object file contains the machine code for that C file.
  - It may contain references to external variables and routines
  - E.g., if **`myprog.c`** calls **`printf()`**, then **`myprog.o`** will contain a reference to **`printf()`**
- Multiple object files are **linked** to produce an executable file.
  - Typically, standard libraries (e.g., "libc") are included in the linking process.
  - Libraries are just collections of pre-compiled object files, nothing more!



Characteristics of assembly language

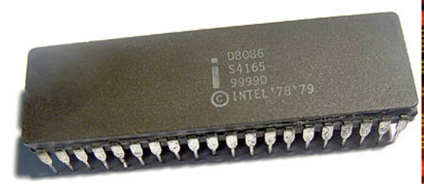
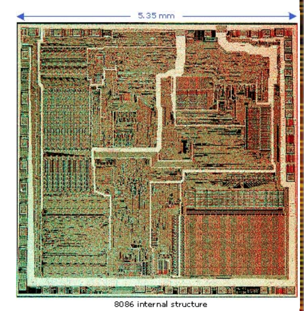
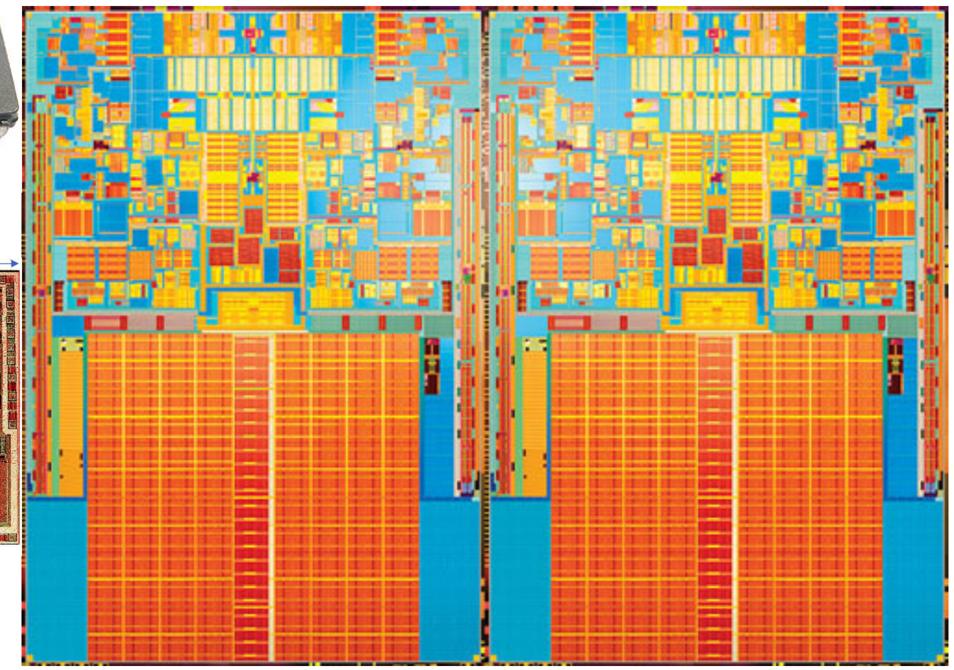
- Assembly language is very, very simple.
- Simple, minimal data types
  - Integer data of 1, 2, 4, or 8 bytes
  - Floating point data of 4, 8, or 10 bytes
  - No aggregate types such as arrays or structures!
- Primitive operations
  - Perform arithmetic operation on registers or memory (add, subtract, etc.)
  - Read data from memory into a register
  - Store data from register into memory
  - Transfer control of program (jump to new address)
  - Test a control flag, conditional jump (e.g., jump only if zero flag set)
- More complex operations must be built up as (possibly long) sequences of instructions.

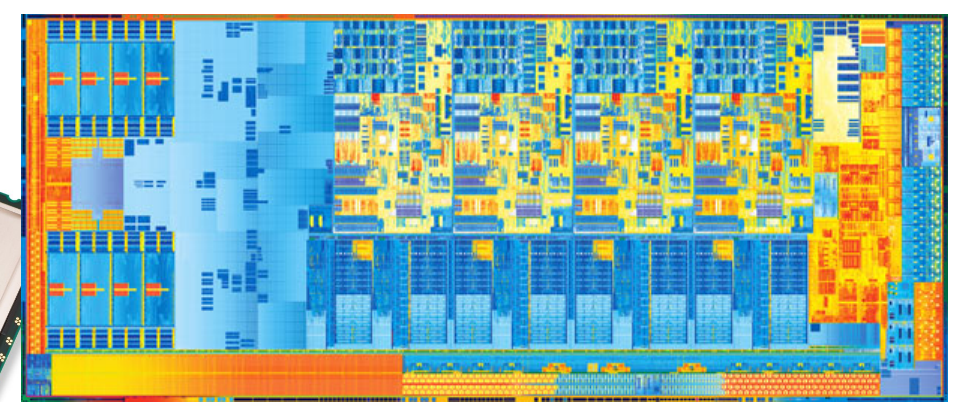
Assembly vs Machine Code

- We write assembly language instructions
  - e.g., "`addq %rbx, %rax`"
- The machine interprets machine code bits
  - e.g., "`101011001100111`..."
- The assembler takes care of compiling assembly language to bits for us.
  - It also provides a few conveniences

Intel’s X86 Architecture

- 1978: Intel introduces 8086
- 1982: 80186, 80286
- 1985: 80386
- 1989: 80486   (100MHz, 1μm)
- 1993: Pentium
- 1995: Pentium Pro
- 1997: Pentium II/III
- 2000: Pentium 4
- 2003: Pentium M, Intel Core
- 2006: Intel Core 2
- 2008: Intel Core i3/i5/i7
- 2011: SandyBridge / IvyBridge
- 2013: Haswell
- 2014: Broadwell
- 2015: Skylake (4.2GHz, 14nm)
- AMD has a parallel line of processors

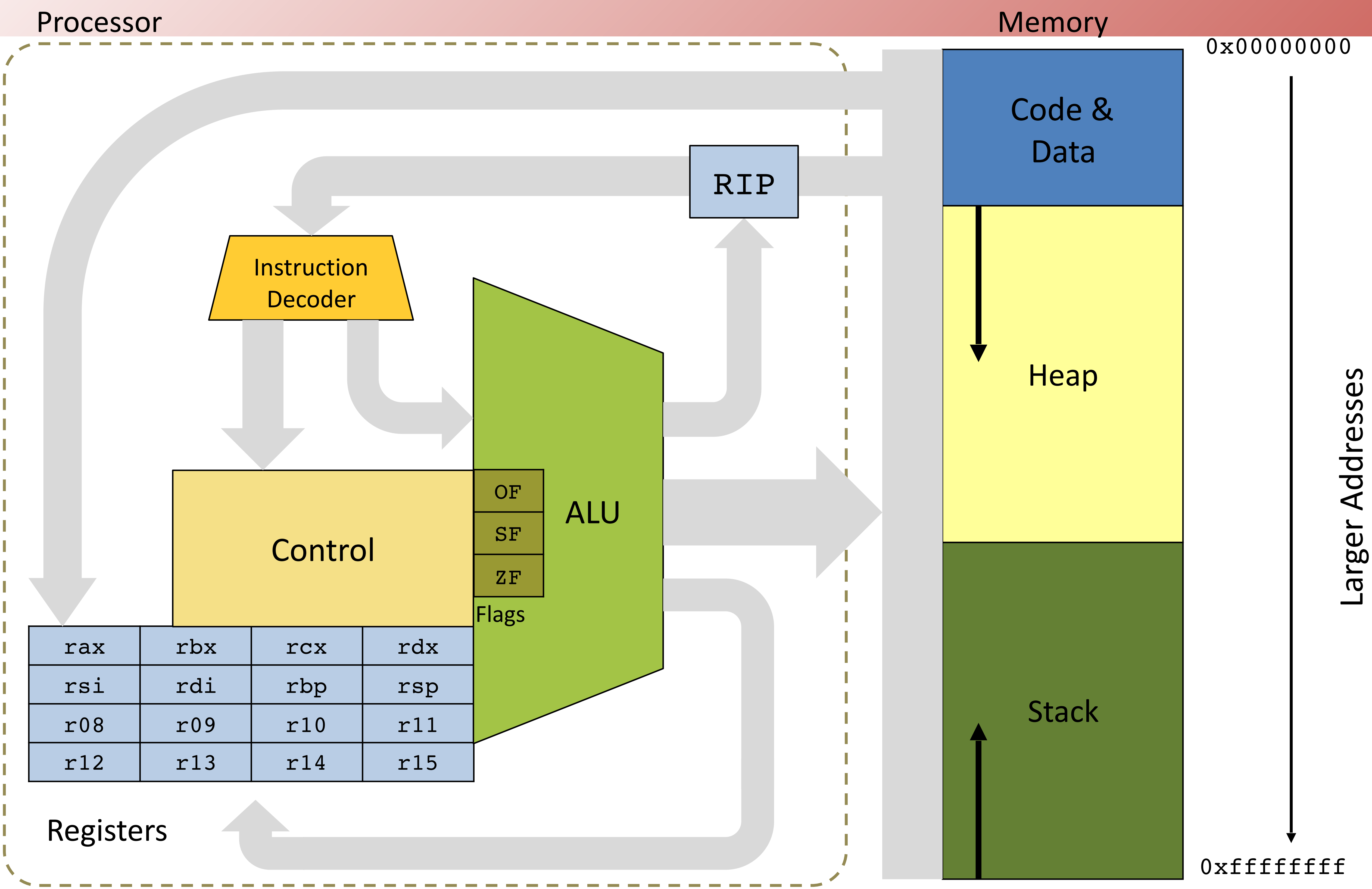




X86 vs. X86lite

- X86 assembly is *very* complicated:
  - 8-, 16-, 32-, 64-bit values + floating points, 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 backwards compatibility
  - Hard to understand, there is a large book about optimizations at just the instruction-selection level
- X86lite is a *very* simple subset of X86:
  - Only 64 bit signed integers (no floating point, no 16bit, no …)
  - Only about 20 instructions
  - Sufficient as a target language for general-purpose computing

X86 Schematic



X86lite Machine State: Registers

- Register File: 16 64-bit registers
  - `rax` general purpose accumulator
  - `rbx` base register, pointer to data
  - `rcx` counter register for strings & loops
  - `rdx` data register for I/O
  - `rsi` pointer register, string source register
  - `rdi` pointer register, string destination register
  - `rbp` base pointer, points to the stack frame
  - `rsp` stack pointer, points to the top of the stack
  - `r08-r15` general purpose registers
- `rip` a “virtual” register, points to the current instruction
  - `rip` is manipulated only indirectly via jumps and return.

Simplest instruction: mov

- `movq SRC, DEST` copy SRC into 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

- `movq $4, %rax` // move the 64-bit immediate value 4 into `rax`
- `movq %rbx, %rax` // move the contents of `rbx` into `rax`

A Note About Instruction Syntax

- X86 presented in **two** common syntax formats
- AT&T notation: source before destination
  - Prevalent in the Unix/Mac ecosystems
  - Immediate values prefixed with '`$`'
  - Registers prefixed with '`%`'
  - Mnemonic suffixes: `movq` vs. `mov`
    - `q` = quadword (4 words)
    - `l` = long (2 words)
    - `w` = word
    - `b` = byte

src dest

```
movq $5, %rax

movl $5, %eax
```

- Intel notation: destination before source
  - Used in the Intel specification / manuals
  - Prevalent in the Windows ecosystem
  - Instruction variant determined by register name

dest src

```
mov rax, 5

mov eax, 5
```

- Note: X86Lite uses AT&T notation and the 64-bit only version of the instructions and registers

Detour: 2's complement

- Representing non-negative integers in bits is straightforward
- How do we represent negative integers in bits?
- Three common encodings:
  - Sign and magnitude
  - Ones' complement
  - Two's complement

Two's complement

- If integer k is represented by bits $b_1 \ldots b_n$, then $-k$ is represented by `100...00` - $b_1 \ldots b_n$ (where |`100...00`|=n+1)
  - Equivalent to taking ones' complement and adding 1
  - E.g., using 4 bits:
    - 6 = `0110`
    - -6 = `10000-0110 = 1010 = (1111-0110)`+1
- Using n bits, can represent numbers 2^n values
  - E.g., using 4 bits, can represent integers
    -8, -7, …, -1, 0, 1, …, 6, 7
  - Like sign and magnitude and ones' complement, first bit indicates whether number is negative

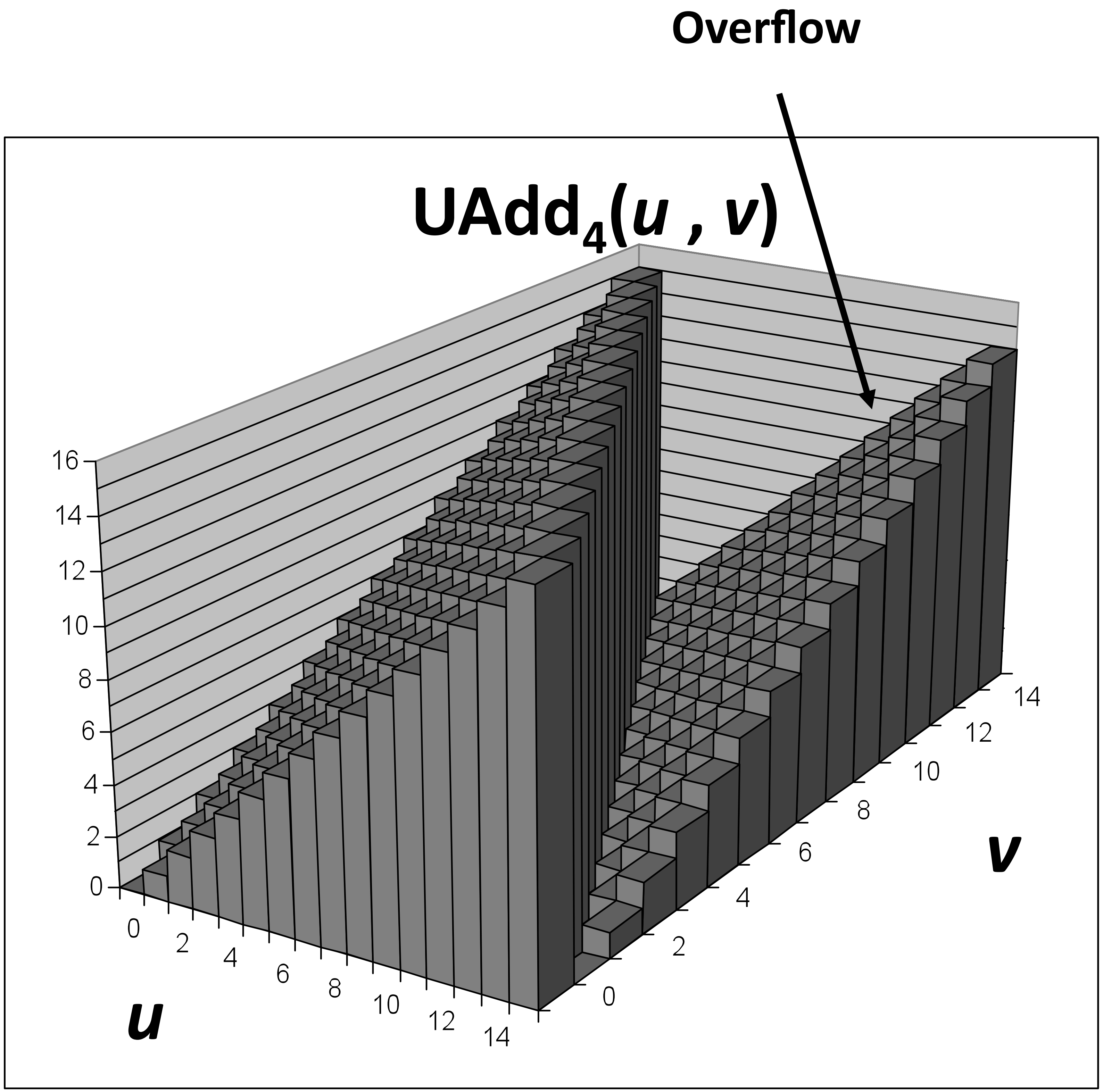
Properties of two's complement

- Same implementation of arithmetic operations as for unsigned
  - E.g., addition, using 4 bits
    - unsigned: `0001` + `1001` = 1 + 9 = 10 = `1010`
    - two's complement: `0001` + `1001` = 1 + -7 = -6 = `1010`
- Only one representation of zero!
  - Simpler to implement operations
- Not symmetric around zero
  - Can represent more negative numbers than positive numbers
- Most common representation of negative integers

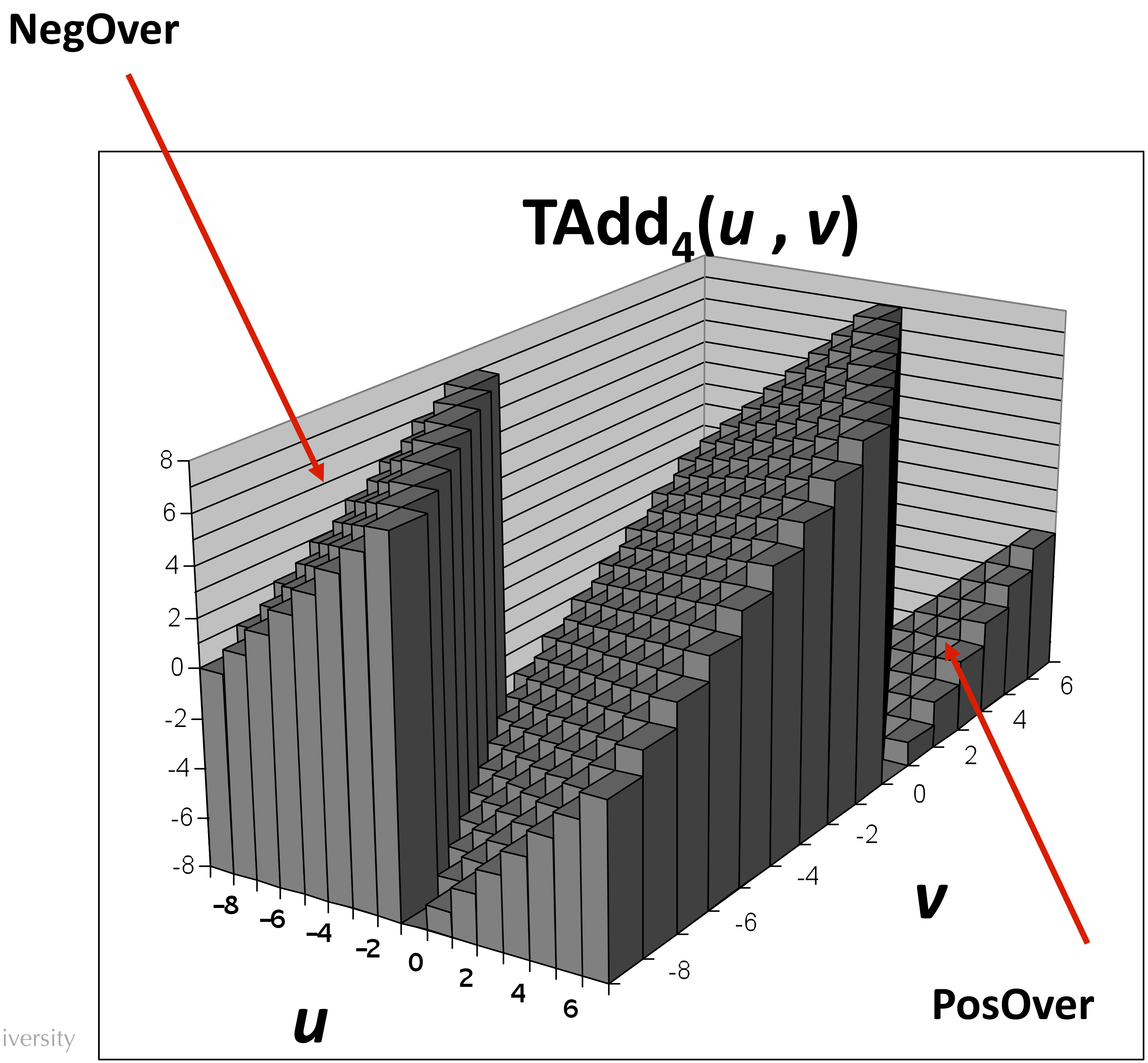
Integer overflow

- Overflow can also occur with negative integers
- With 32 bits, maximum integer expressible in 2's complement is 2^{31}-1 = `0x7fffffff`
- `0x7fffffff` + `0x1` = `0x80000000` = -2^{31}
  - Minimum integer expressible in 32-bit 2's complement
- `0x80000000` + `0x80000000` = `0x0`

Integer overflow



Integer overflow



X86lite Arithmetic instructions

- `negq` DEST — two's complement negation
- `addq` SRC, DEST — DEST ← DEST + SRC
- `subq` SRC, DEST — DEST ← DEST – SRC
- `imulq` SRC, Reg — Reg ← Reg * SRC (truncated 128-bit mult.)
- Examples:
  - `addq %rbx, %rax` // rax ← rax + rbx
  - `subq $4, rsp` // rsp ← rsp - 4
- Note: Reg (in imulq) must be a register, not a memory address

X86lite Logic/Bit manipulation Operations

• `notq` DEST	logical negation
• `andq` SRC, DEST	DEST ← DEST && SRC
• `orq` SRC, DEST	DEST ← DEST || SRC
• `xorq` SRC, DEST	DEST ← DEST xor SRC
• `sarq` Amt, DEST	DEST ← DEST >> amt (arithmetic shift right)
• `shlq` Amt, DEST	DEST ← DEST << amt (arithmetic shift left)
• `shrq` Amt, DEST	DEST ← DEST >>> amt (bitwise shift right)

X86 Operands

- Operands are the values operated on by the assembly instructions
- Imm — 64-bit literal signed integer "immediate"
- Lbl — a "label" representing a machine address
  the assembler/linker/loader resolve labels
- Reg — One of the 16 registers, the value of a register is its contents
- Ind — [base:Reg][index:Reg,scale:int32][disp]
  machine address (see next slide)

X86 Addressing

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

- Example: `-4(%rsp)` denotes address: `rsp – 4`
- Example: `(%rax, %rcx, 4)` denotes address: `rax + 4*rcx`
- Example: `12(%rax, %rcx, 4)` denotes address: `rax + 4*rcx +12`

- Note: Index cannot be `rsp`
- Note: X86Lite does not needs this full generality. It does not use index * scale

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, there is a stack that 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

X86lite State: Condition Flags & Codes

- X86 instructions set flags as a side effect
- X86lite has only 3 flags:
  - `OF`: "**overflow**" set when the result is too big/small to fit in 64-bit reg.
  - `SF`: "**sign**" set to the sign or 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
  - `e` equality holds when `ZF` is set
  - `ne` inequality holds when (not `ZF`)
  - `g` greater than holds when (not `ZF`) and (not `SF`)
  - `l` less than holds when SF <> OF
    - Equivalently: ((`SF` && not `OF`) || (not `SF` && `OF`))
  - `ge` greater or equal holds when (not `SF`)
  - `le` than or equal holds when `SF` <> `OF` or `ZF`

Code Blocks & Labels

- X86 assembly code is organized into **labeled blocks**:

```
label1:
        <instruction>
        <instruction>
        …
        <instruction>

label2:
        <instruction>
        <instruction>
        …
        <instruction>
```

- 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`")

Conditional Instructions

- `cmpq` SRC1, SRC2 — Compute SRC2 – SRC1, set condition flags
- `setb`CC DEST — DEST's lower byte ← if CC then 1 else 0
- `j`CC SRC — `rip` ← if CC then SRC else fallthrough
- Example:

```
cmpq %rcx, %rax      // Compare rax to ecx
je __truelbl         // If rax = rcx then jump to __truelbl
```

Jumps, Call and Return

- `jmp` SRC   `rip` ← SRC   Jump to location in SRC
- `callq` 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.
- `retq`   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

Implementing X86Lite

- See file `x86.ml`

Compiling, Linking, Running

- To use hand-coded X86:
  - 1.Compile `main.ml` (or something like it) to either native or bytecode
  - 2.Run it, redirecting the output to some `.s` file, e.g.:
    - `./main >> test.s`
  - 3.Use `gcc` to compile & link with `runtime.c`:
    - `gcc -o test runtime.c test.s`
  - 4.You should be able to run the resulting executable:
    - `./test`

- If you want to debug in `gdb`:
  - Call `gcc` with the `–g` flag too