Lecture 3

# CIS 341: COMPILERS

## Announcements

- HW01: Hellocaml!
  - is due *tomorrow tonight* at 11:59:59pm.

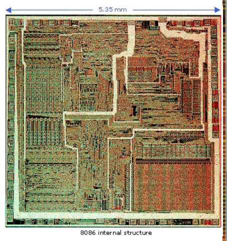
- HW02: X86lite
  - Will be available soon… look for an announcement on Piazza
  - Pair-programming project
  - Simulator / Loader for x86 Assembly subset

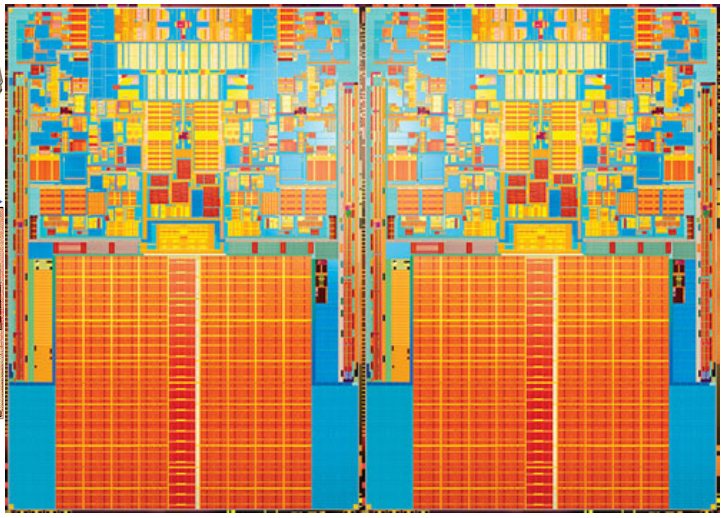
The target architecture for CIS341

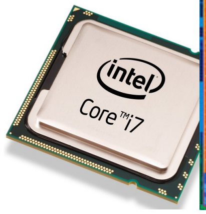
## X86LITE

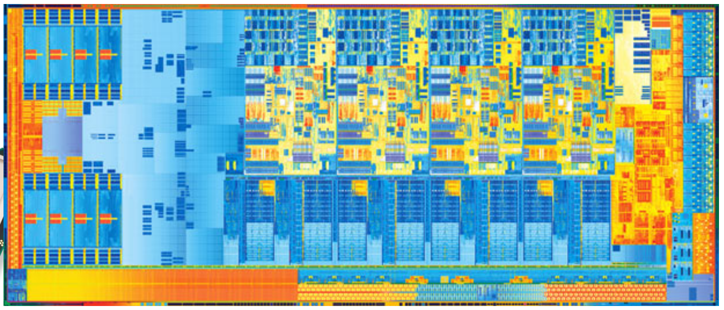
## Intel's X86 Architecture

- 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

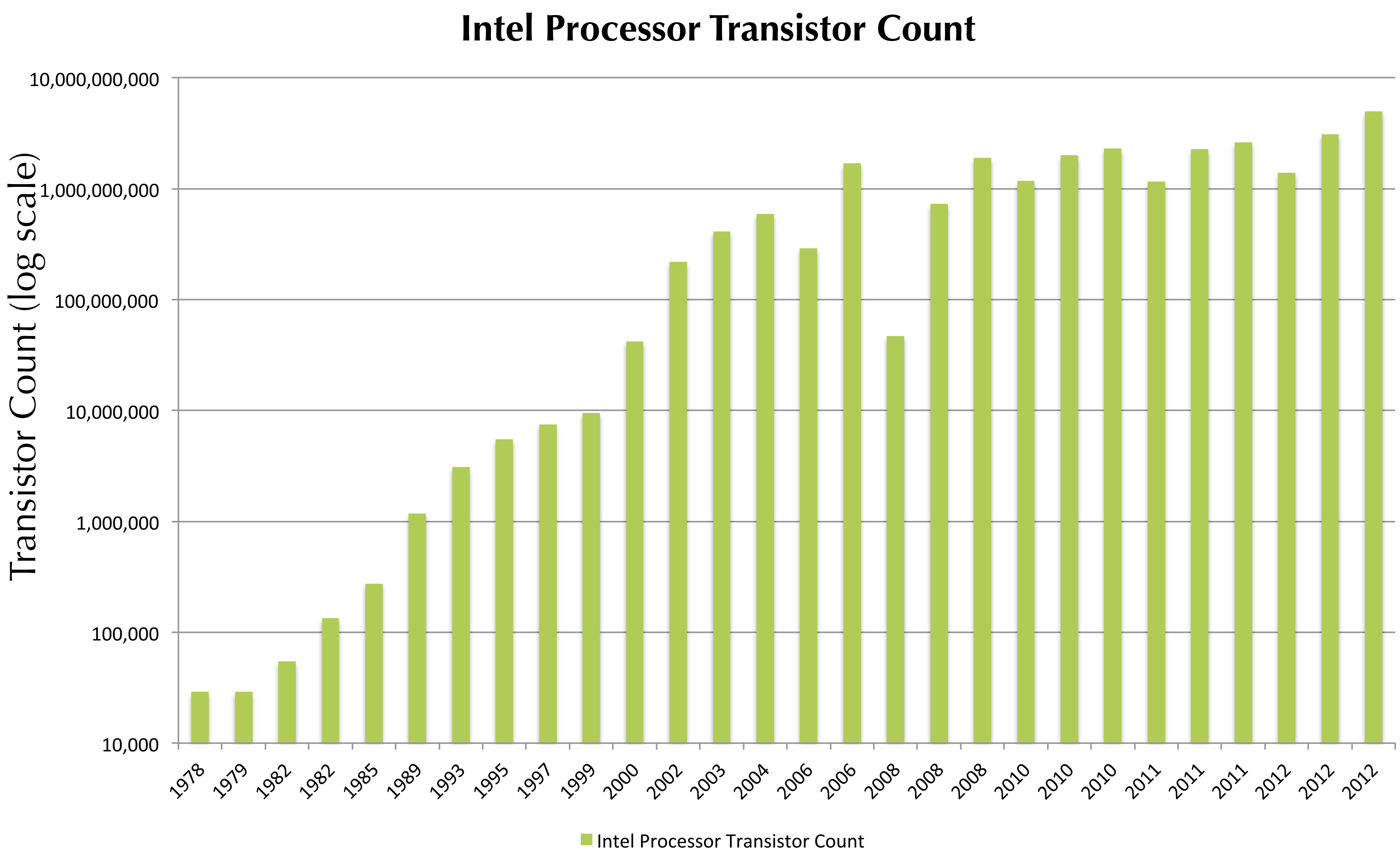








# X86 Evolution & Moore's Law

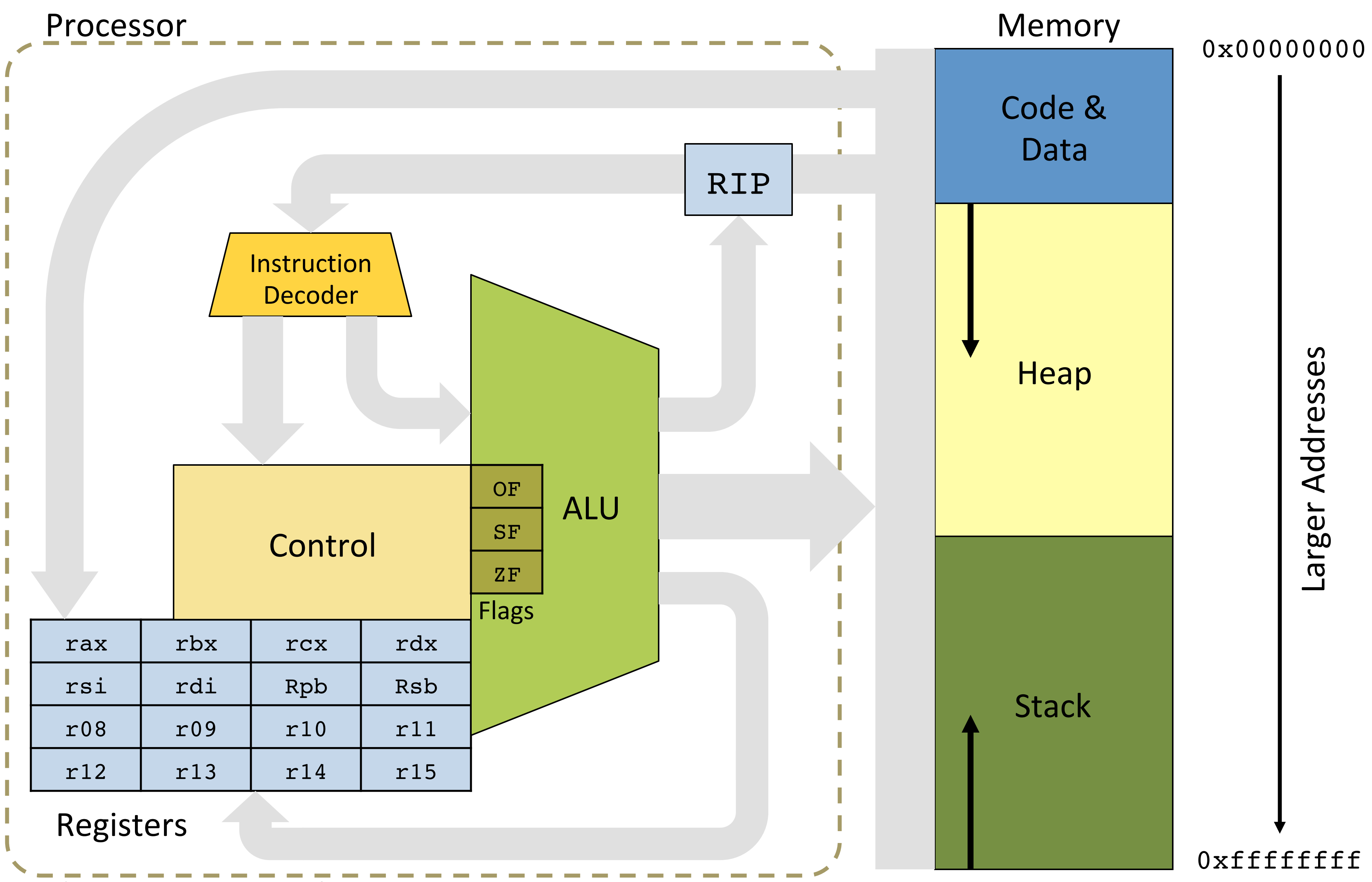


## X86 vs. X86lite

- 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

```
movq $5, %rax

movl $5, %eax
```

src dest

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

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

```
mov rax, 5

mov eax, 5
```

dest src

# 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 as written in:

```
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<br>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]<br>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 need 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` ← `rps - 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 |
| jCC 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

See file: x86.ml

## IMPLEMENTING X86LITE

See: runtime.c

## DEMO: HANDCODING X86LITE

## 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.:
     ./handcoded.native >> 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 exectuable:
     ./test

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