Lecture 3

CIS 341: COMPILERS

Announcements

- Project 0: Hellocaml!
  - is due *tonight* at 11:59:59pm.

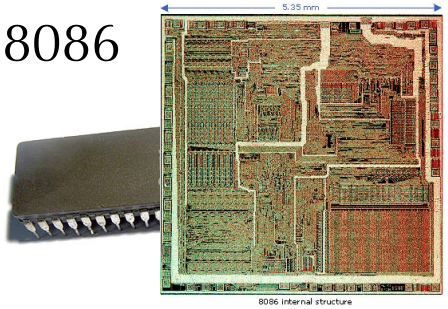
- Project 1: X86lite
  - Will be available soon… look for an announcement
  - Due: Thurs. January 31st
  - Pair-programming project: sign up on Piazza

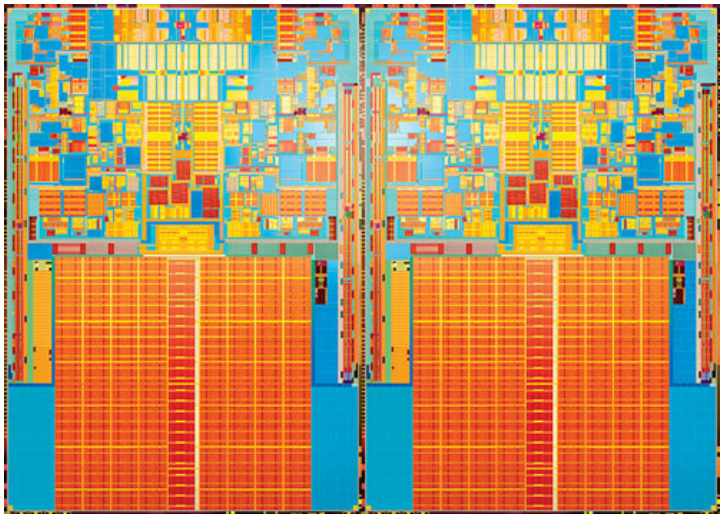
The target architecture for CIS341

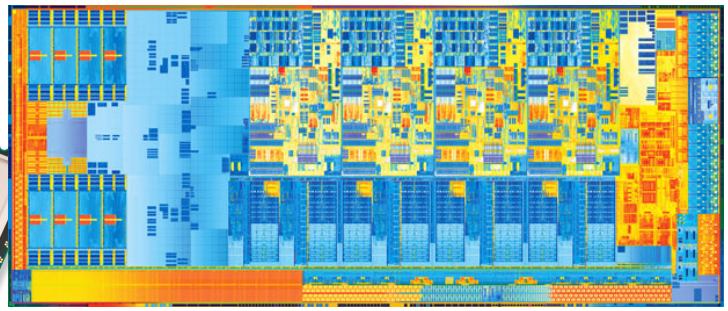
X86LITE

Intel's X86 Architecture

- 1978: Intel introduces 8086
- 1982: 80186, 80286
- 1985: 80386
- 1989: 80486
- 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

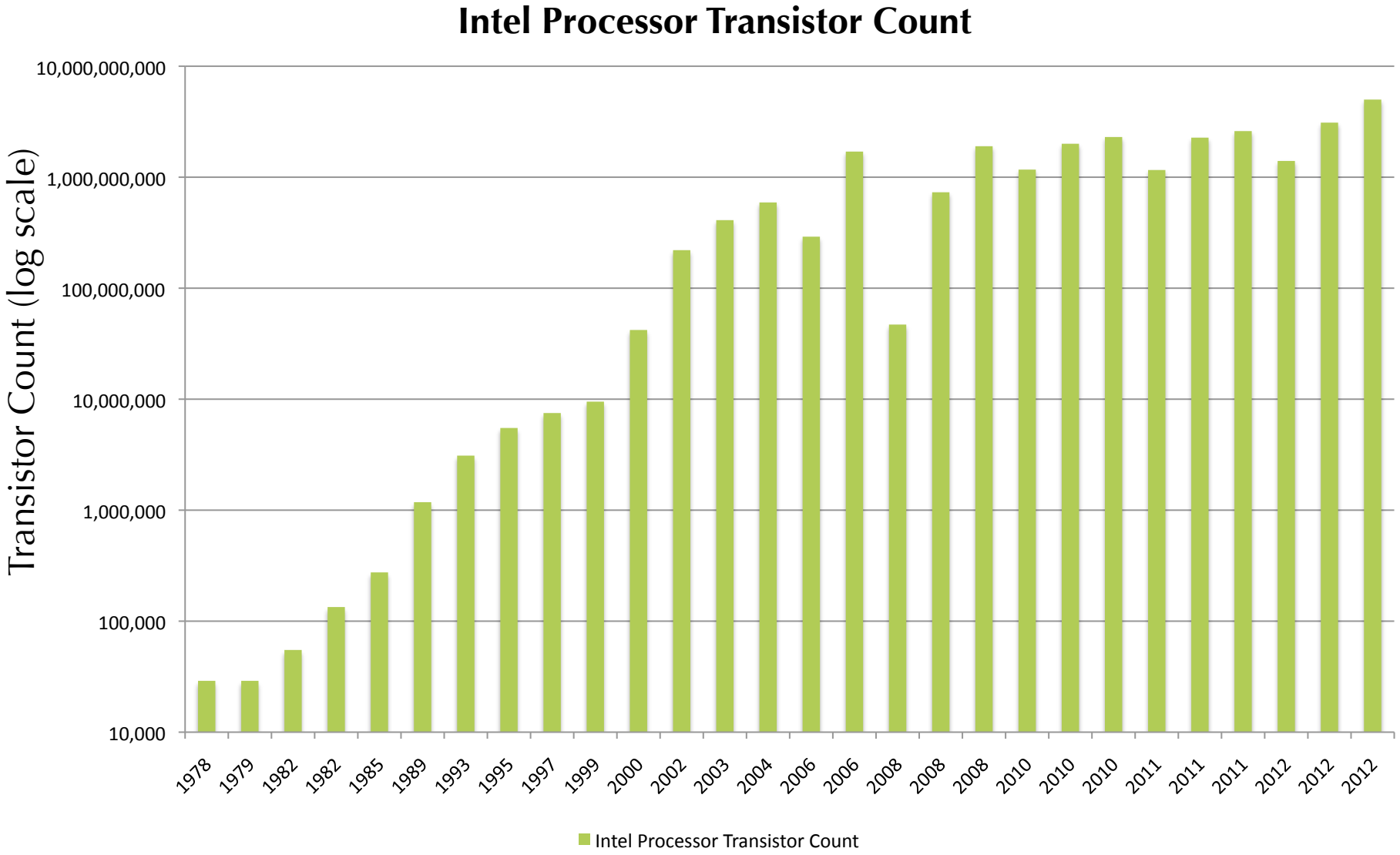






- AMD has a parallel line of processors

X86 Evolution & Moore's Law

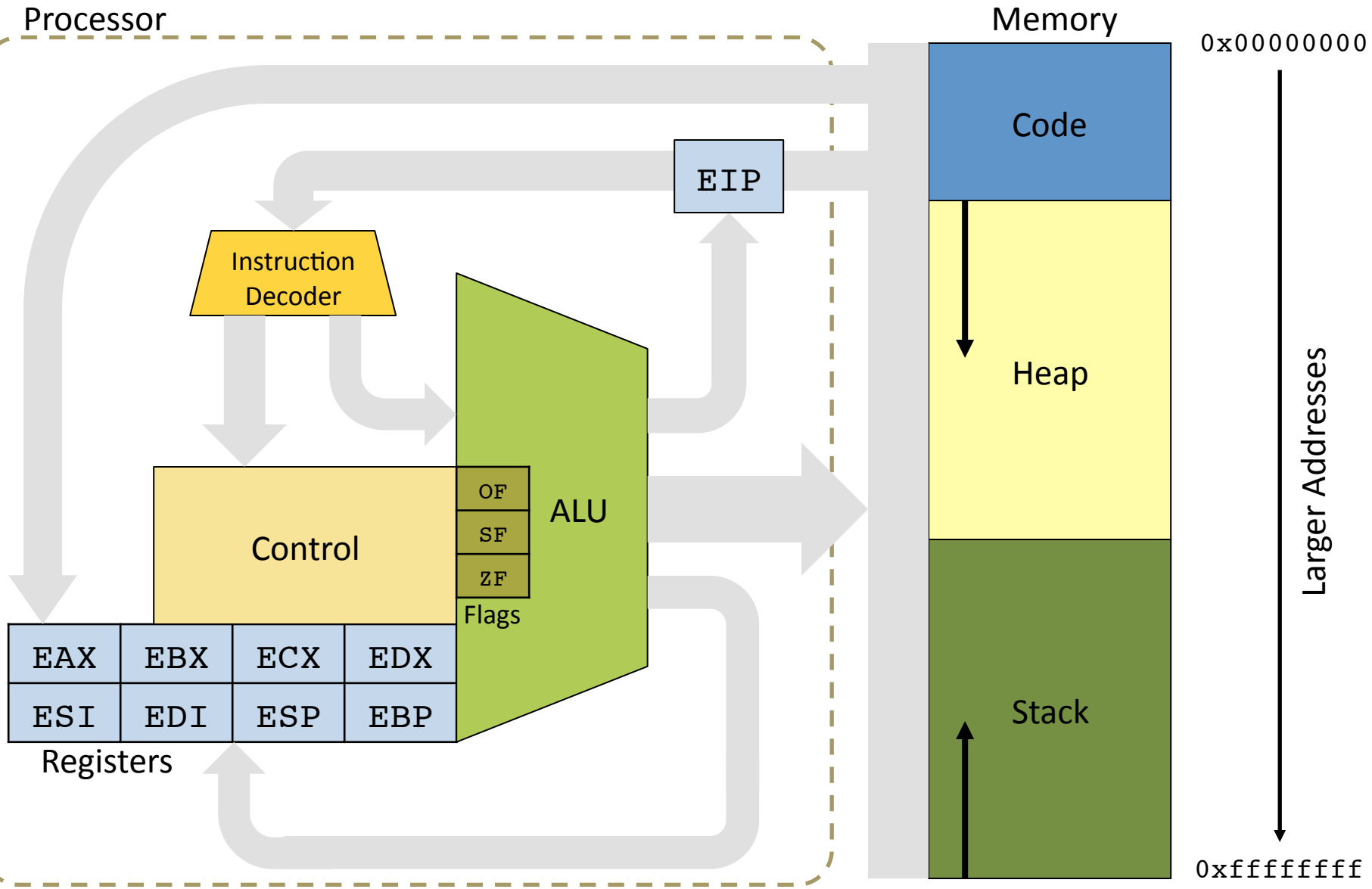


X86 vs. X86lite

- 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 32 bit signed integers (no floating point, no 16bit, no …)
  - Only about 20 instructions
  - Sufficient for implementing the entire OAT language

X86 Schematic



X86lite Machine State: Registers

- Register File: 8 32-bit registers
  - `EAX` general purpose accumulator
  - `EBX` base register, pointer to data
  - `ECX` counter register for strings & loops
  - `EDX` data register for I/O
  - `ESI` pointer register, string source register
  - `EDI` pointer register, string destination register
  - `EBP` base pointer, points to the stack frame
  - `ESP` stack pointer, points to the top of the stack

- `EIP` a "virtual" register, points to the current instruction
  - EIP is manipulated only indirectly via jumps and return.

Simplest instruction: Mov

- `Mov` DEST SRC 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

- `Mov EAX 4` move the immediate value `4` into `EAX`
- `Mov EAX EBX` move the contents of `EBX` into `EAX`

X86lite Arithmetic instructions

- `Neg` DEST — two's complement negation
- `Add` DEST SRC — DEST ← DEST + SRC
- `Sub` DEST SRC — DEST ← DEST - SRC
- `Imul` Reg SRC — Reg ← Reg * SRC (truncated 64-bit mult.)

Examples:

```
Add EAX EBX          // EAX ← EAX + EBX
Sub ESP 4            // ESP ← ESP - 4
```

- Note: Reg (in `Imul`) must be a register, not a memory address

X86lite Logic/Bit manipulation Operations

- `Not` DEST — logical negation
- `And` DEST SRC — DEST ← DEST && SRC
- `Or` DEST SRC — DEST ← DEST || SRC
- `Xor` DEST SRC — DEST ← DEST xor SRC

- `Sar` DEST Amt — DEST ← DEST >> amt (arithmetic shift right)
- `Shl` DEST Amt — DEST ← DEST << amt (arithmetic shift left)
- `Shr` DEST Amt — DEST ← DEST >>> amt (bitwise shift right)

X86lite Operands

- Operands are the values operated on by the assembly instructions
- Imm — 32-bit literal signed integer "immediate"
- Lbl — a "label" representing a machine address
  the assembler/linker/loader resolve labels
- Reg — One of the 8 registers, the value of a register is its contents
- Ind — [base:Reg][index:Reg,scale:int32][disp]
  machine address (see next slide)

X86 Addressing

- 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(ESP)` denotes address: `ESP – 4`
- Example: `(EAX, ECX, 4)` denotes address: `EAX + 4*ECX`
- Example: `-12(EAX, ECX, 4)` denotes address: `EAX + 4*ECX -12`

- Note: Index cannot be `ESP`
- Note: For our purposes, scale will always be 4

X86lite Memory Model

- The X86lite memory consists of 2^{32} bytes numbered `0x00000000` through `0xffffffff`.
- X86lite treats the memory as consisting of 32-bit (4-byte) words.
- Therefore: legal X86lite memory addresses consist of 32-bit, word-aligned pointers.
  - All memory addresses are evenly divisible by 4

- `Lea` DEST Ind DEST ← addr(Ind) loads a pointer into DEST

- By convention, there is a stack that grows from high addresses to low addresses
- The register ESP points to the top of the stack
  - `Push` SRC `ESP` ← `ESP` - `4`; Mem[`ESP`] ← SRC
  - `Pop` DEST DEST ← Mem[`ESP`]; `ESP` ← `ESP` + `4`

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 32-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
  - `Eq` equality holds when `ZF` = 1
  - `NotEq` inequality holds when `ZF` = 0
  - `Slt` signed less than holds when
    - (`SF`=1 && `OF`=0) || (`SF`=0 && `OF`=1))
    - Equivalently: `SF` <> `OF`
  - `Sle` signed less than or equal holds when (`SF` <> 0 || `ZF`)
  - `Sgt` signed greater than holds when not(`SF` <> 0 || `ZF`)
  - `Sge` signed gtr than or equal holds when not(`SF` = `OF`)

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

- `Cmp` SRC1 SRC2 — Compute SRC1 - SRC2, set condition flags
- `Setb` DEST CC — DEST's lower byte ← if CC then 1 else 0
- `J` CC Lbl — `EIP` ← if CC then Lbl else fallthrough
- Example:

```
Cmp EAX  ECX        Compare EAX to ECX
J Eq  __truelbl     If EAX = ECX then jump to __truelbl
```

Jumps, Call and Return

- `Jmp` SRC `EIP` ← SRC Jump to location in SRC

- `Call` SRC Push `EIP`; `EIP` ← SRC
  - Call a procedure: Push the program counter to the stack (decrementing ESP) and then jump to the machine instruction at the address given by SRC.

- `Ret` Pop `EIP`
  - Return from a procedure: Pop the current top of the stack into `EIP` (incrementing `ESP`). This instruction effectively jumps to the address at the top of the stack

Notation

- Different assemblers use different notation
- This lecture has used one variant: `Mov` DEST SRC
  - It is close to the X86lite implementation in the course projects

- Another common variant: `mov` SRC, DEST
  - Pronouce the comma as "into"
  - Write `EAX` as `%eax`
  - Write immediate `4` as `$4`

- Note: "true x86" uses different instruction suffixes for different word sizes: `mov` vs. `movl`
  - All of our assembly uses the "`l`" suffix
  - `Mov EAX -4(ECX)` is displayed as `movl -4(%ecx), %eax`

- Similarly, "true x86" adds the condition codes as suffixes for the instructions: `Jeq` for "Jump if Eq"

See files: x86.ml, x86.mli, cunit.ml, cunit.mli

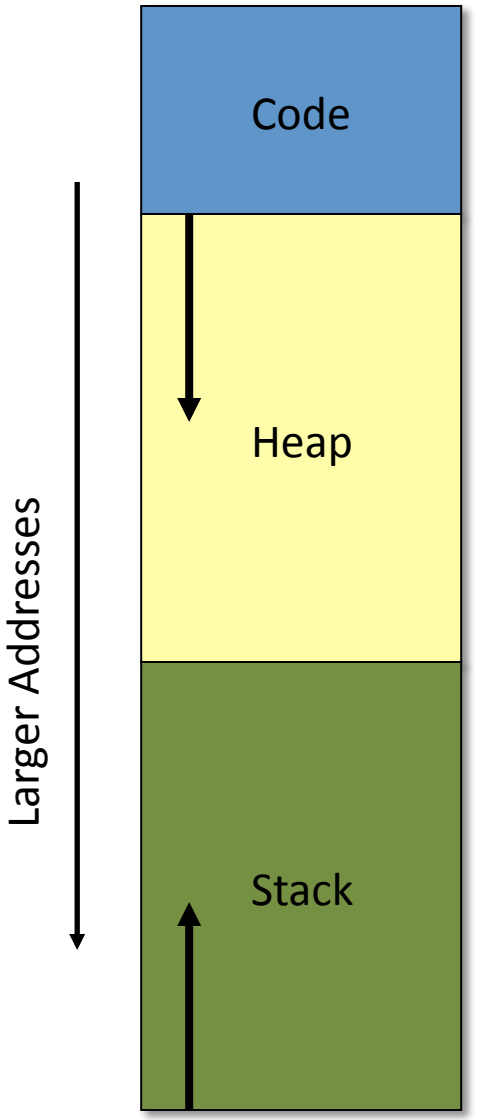
IMPLEMENTING X86LITE

PROGRAMMING IN X86LITE

3 parts of the C memory model

- The code & data (or "text") segment
  - contains compiled code, constant strings, etc.
- The Heap
  - Stores dynamically allocated objects
  - Allocated via "malloc"
  - Deallocated via "free"
  - C runtime system
- The Stack
  - Stores local variables
  - Stores the return address of a function

- In practice, most languages use this model.



Implementing Functions/Procedures

- Consider the following program:

```
int square(int x) {
  int z = x * x;
  return z;
}

int f(int x1, int x2) {
  int dx = x2 - x1;
  return square(dx);
}
```

- What do we need to do to compile this?

- Local variables…
- Function arguments passed in the call…
- Control-flow: jump to "square" return to "f"…

Local/Temporary Variable Storage

- Need space to store:
  - Global variables
  - Values passed as arguments to procedures
  - Local variables (either defined in the source program or introduced by the compiler)

- Processors provide two options
  - Registers: fast, small size (32 or 64 bits), very limited number
  - Memory: slow, very large amount of space (2 GB)

- In practice on X86:
  - Registers are limited (and have restrictions)
  - Divide memory into regions including the *stack* and the *heap*

Calling Conventions

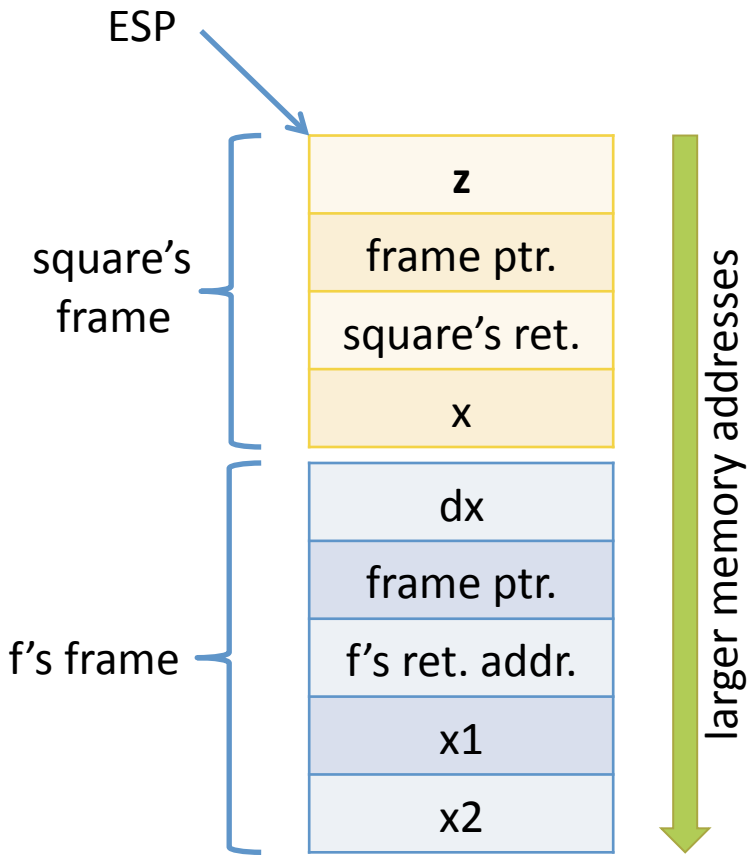
- Specify the locations (e.g. register or stack) of arguments passed to a function
- Designate registers either:
  - Caller Save – e.g. freely usable by the called code
  - Callee Save – e.g. must be restored by the called code
- Protocol for deallocating stack-allocated arguments
  - Caller cleans up
  - Callee cleans up (makes variable arguments harder)

cdecl calling conventions

- "Standard" on X86 for many C-based operating systems (i.e. almost all)
  - Still some wrinkles about return values (e.g. some compilers use `EAX` and `EDX` to return small values)
  - This is evolving due to 64 bit (which allows for packing multiple values in one register)
- Arguments are passed on the stack in right-to-left order
- Return value is passed in `EAX`
- Registers `EAX`, `ECX`, `EDX` are caller save
- Other registers are callee save
  - Ignoring these conventions will cause havoc (bus errors or seg faults)

Call Stacks: Example

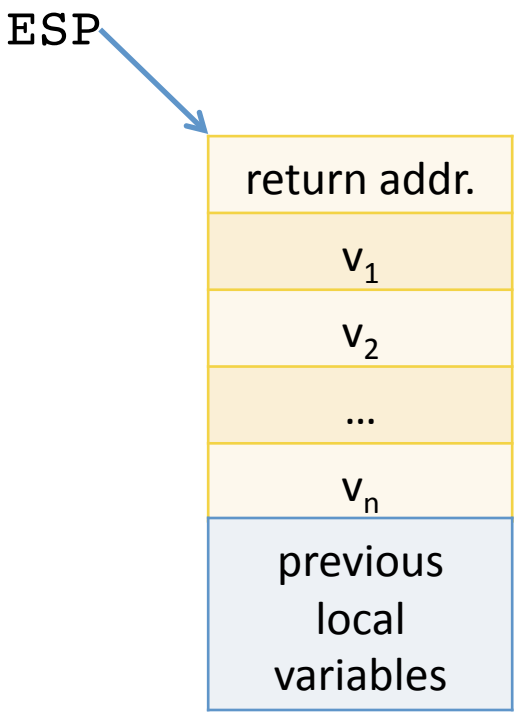
- Use a stack to keep track of the return addresses:
  - `f` calls `g`, `g` calls `h`
  - `h` returns to `g`, `g` returns to `f`
- Stack frame:
  - Functions arguments
  - Local variable storage
  - Return address
  - Link (or "frame") pointer



Call Stacks: In general

- Function call:
  `f(`e_1`, `e_2`, …, `e_n`)`:
- Evaluate e_1 to v_1, e_2 to v_2, … , e_n to v_n
- Push v_n to v_1 onto the top of the stack.
- Use `call` to jump to the code for '`f`', pushing the return address onto the stack.
- Invariant: returned value passed in `EAX`
- After call, the calling code cleans up the pushed arguments

State of the stack just after the Call instruction:



See: handcoding.ml, runtime.c

DEMO: HANDCODING X86LITE

Compiling, Linking, Running

- To use hand-coded X86 in handcoded.ml:
  1. Compile handcoded.ml to either native or bytecode
  2. Run it, redirecting the output to some .s file, e.g.:
     ./handcoded.native >> test.s
  3. Use gcc with the -m32 flag to compile & link with runtime.c:
     gcc -m32 -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