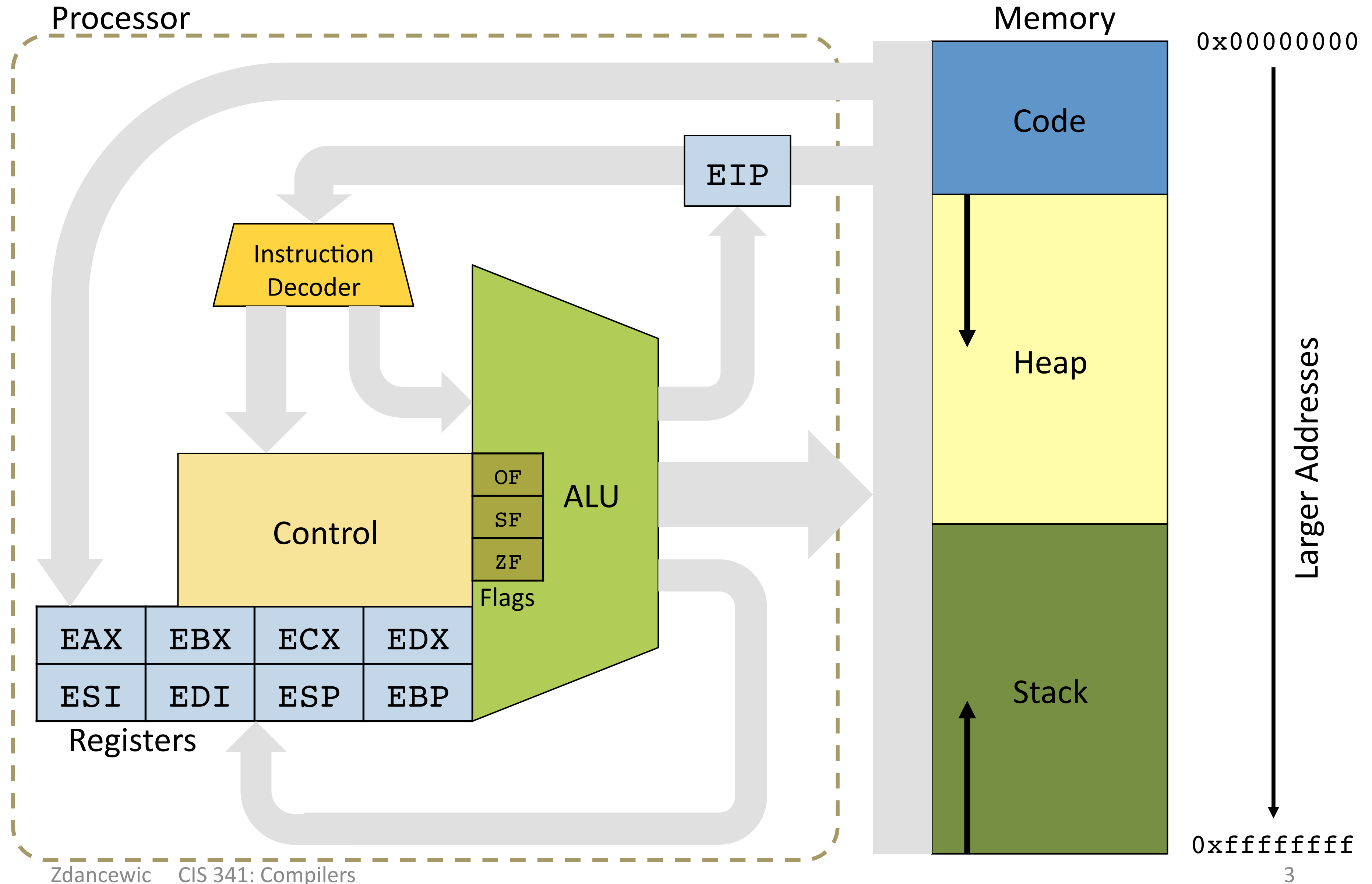
Lecture 4

# CIS 341: COMPILERS

# Announcements

- Project 1: X86lite
  - Available on the course web pages.
  - Due: Thurs. January 31st
  - Pair-programming project: sign up on Piazza

# X86 Schematic

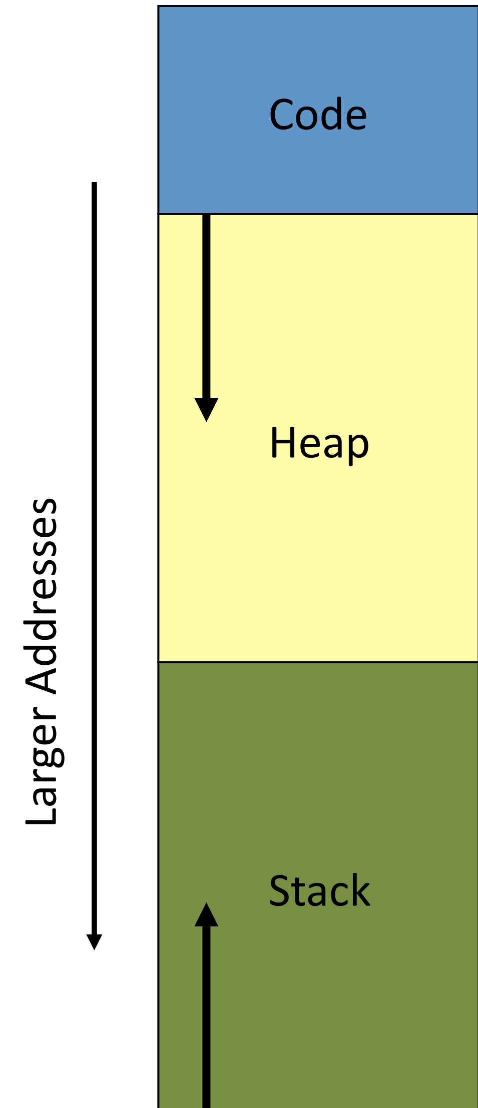


# PROGRAMMING IN X86LITE

# 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.



# 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

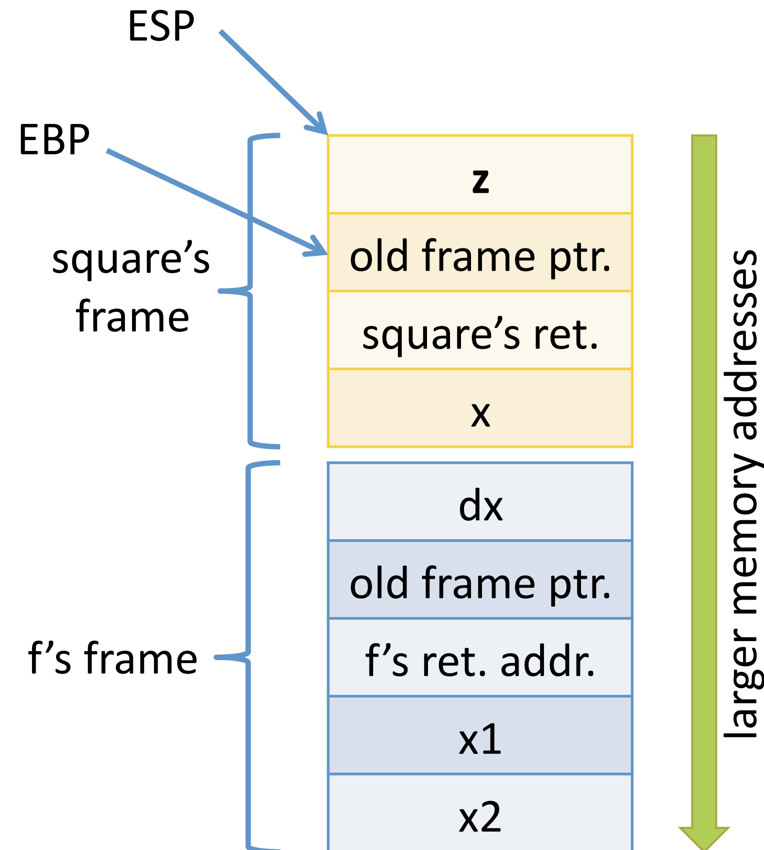
- Specify the locations (e.g. register or stack) of arguments passed to a function
- Designate registers either:
  - Caller Save – e.g. freely usable by the called code
  - Callee Save – e.g. must be restored by the called code
- Protocol for deallocating stack-allocated arguments
  - Caller cleans up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

- Function call:

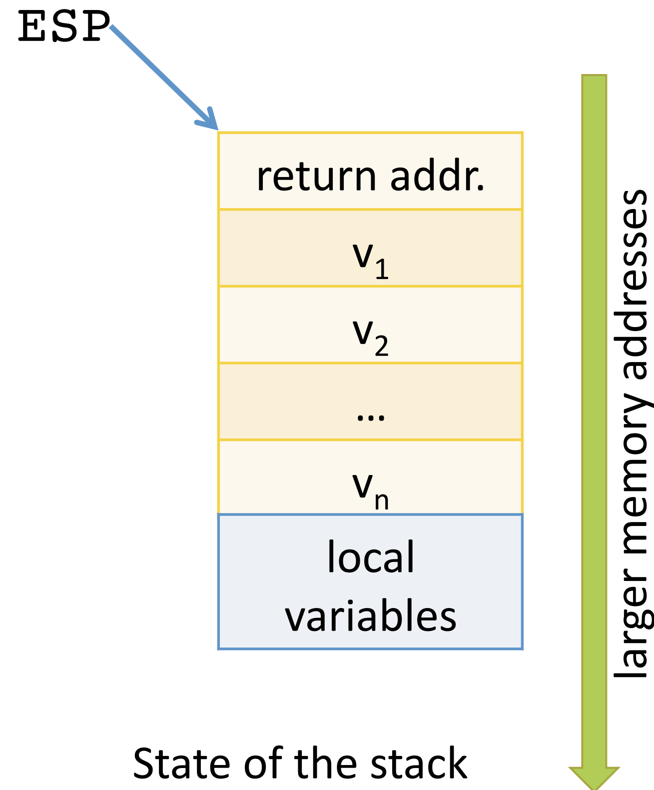
```
f(e1, e2, …, en);
```

1. Save caller-save registers
2. Evaluate $e_1$ to $v_1$, $e_2$ to $v_2$, … , $e_n$ to $v_n$
3. Push $v_n$ to $v_1$ onto the top of the stack.
4. Use `Call` to jump to the code for `f`
   - pushing the return address onto the stack.

- Invariant: returned value passed in `EAX`

- After call:

1. clean up the pushed arguments by popping the stack.
2. Restore caller-saved registers




State of the stack just after the Call instruction:

# Call Stacks: Callee's protocol

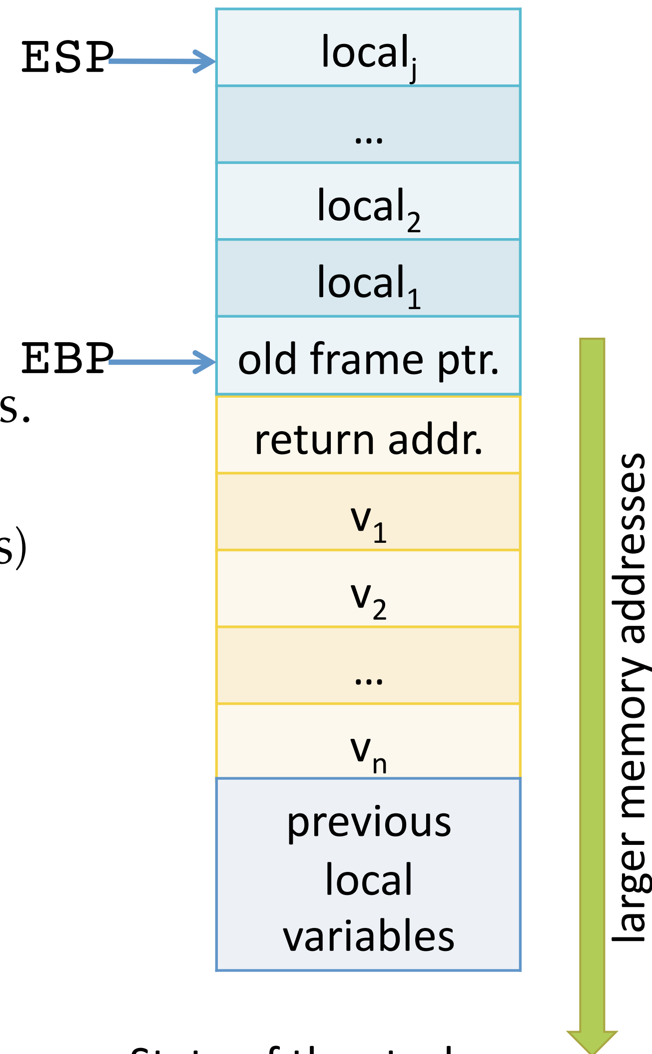
- On entry:

1. Save old frame pointer
   – EBP is callee save
2. Create new frame pointer
   – Mov(Esp, Ebp)
3. Allocate stack space for local variables.

- Invariants: (assuming word-size values)
  – Function argument n is located at:
    EBP + (1 + n) * 4
  – Local variable j is located at:
    EBP – j * 4

- On exit:

1. Pop local storage
2. Restore EBP




State of the stack after Step 3 of entry.

See: handcoding.ml, runtime.c

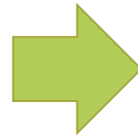
# DEMO: HANDCODING X86LITE

# Hand-generated Implementation

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

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

int program() {
  return f(12,3);
}
```



```
    .align 4
    .text
.globl _program
_program:
    pushl %ebp
    movl %esp, %ebp
    pushl $12
    pushl $3
    call f
    addl $8, %esp
    popl %ebp
    ret
f:
    pushl %ebp
    movl %esp, %ebp
    movl 12(%ebp), %eax
    movl 8(%ebp), %ebx
    subl %ebx, %eax
    pushl %eax
    call square
    addl $4, %esp
    popl %ebp
    ret
square:
    movl 4(%esp), %eax
    imull %eax, %eax
    ret
```

# 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

# Using GDB (cheat sheet)

- Useful GDB commands:
  - `break` <lbl> — set a breakpoint at the given label
  - `disas` <lbl> — disassemble the code at a given label
  - `info registers` — display the current register state (abbrev: i r)
  - `info stack` — display the current stack trace (abbrev: i s)
  - `disaplay/i` $pc — tell gdb to display the current EIP and instruction
  - `display/x` <expr> — tell gdb to display the expression in hex

  - `si` — step a single machine instruction, into function calls
  - `ni` — step a single machine instruction, past function calls

  - `p/d $`<reg> — print the contents of <reg> in decimal
  - `p/x $`<reg> — print the contents of <reg> in hex
  - `x/i` <addr> — print the contents of address <addr> as instruction
  - `x/d` <addr> — print the contents of address <addr> as decimal
  - `x/x` <addr> — print the contents of address <addr> as hex

# DEMO: GDB