Introduction to PCs and the x86 instruction set.

Fall 2002

1 Background

```
o Intro...

o Operating system interfaces with hw.
  - list of hw bits

o Computer abstractions
  - Memory
  - Interpreter
  - Communication links

o Memory
  - interface: read/write: what name to read/write? size of access?
  - e.g. ram. disks.
o Interpreter
  - interface: pc, context, instruction set
  - example of something in context? vm, kernel mode
  - instruction set: addressing modes, basic operations,
                     control flow, advanced operations.
o Communication links
  - interface: send/receive
  - like reading from one memory and writing to a remote one.
  - can have complex issues, e.g. asynchrony, reliability, etc.

o Combining these...
  - Computer, with a bus.
  - Build up memory hierarchy.
  - Consider the Beta processor (6.004)...

o Calling conventions
  - What are they?
  - Why are they needed?
  - Things to consider:
    - Callee/Caller saving.
    - How parameters are passed.
    - How to get return address?
    - How to return values?
    - Locals?
    - Standard prologue/epilogue

o Beta:
```

```
- Context: PC. High-bit of address is supervisor bit.
- Instruction sets: reg/reg and reg/immed. standard control flow
  and ops only.
- Calling conventions:
  - Callee save.
  - Reserved registers (e.g. r31 = 0, r29 = sp, ...)
  - Parameters on stack.
  - Return address in r28
  - Return value in r0
  - Locals on stack
  - Prologue saves callers state (LP/BP/etc), sets up locals.
    Epilogue restores.
```

2 x86 register set

general purpose `EAX, EBX, ECX, EDX, ESI, EDI`

stack `ESP, EBP`

segment `CS, SS, DS, ES, FD, GS`

instruction pointer `EIP`

flags and status `EFLAGS` a bit vector of status bits. E.g., overflow flag, sign flag, zero flag, parity flag, carry flag.

system `CR0, CR1, ..., CR4`, and others. Configure mode of the processor.

3 x86 instruction set

data movement `mov, push, pop`

arithmetic/logic `test, shift, add, sub`

port I/O `in, out`

control `jmps, jz, call, ret`

string `rep movsl`

system `iret, int`

4 GCC assembly syntax

```
refer to:
  http://www.cs.princeton.edu/courses/archive/fall99/cs318/Files/djgpp.html
```

4.1 The stack on the x86

On the x86, the convention is that stacks grow down. This convention is a direct consequence of the push and pop instructions. The pseudocode for these is show in Figure 1.

Another convention that can be gleaned from this figure is that the stack pointer (%esp) always points at a valid value (i.e., the last value that was pushed).

```
pushl %REG:
   %esp -= 4
   movl %REG, (%esp)

popl %REG:
   movl (%esp), %REG
   %esp += 4
```

Figure 1: Pseudo code for push and pop.

5 GCC's calling convention on the x86

Calling a procedure is a contract between the caller and the callee. The caller guarrantees a certain state of the machine before the callee takes over. And the callee guarrantees a certain state after it returns control back to the caller. The control transfer is carried out with the call and ret instructions, whose operation is shown in pseudo code below.

```
call ADDRESS:                       ret:
   pushl %eip                           popl %eip
   %eip = ADDRESS

        pseudo-code of call and ret x86 instructions.
```

The contract between the caller and callee is a contract over the state of the registers before and after the callee runs.

```
Immediately after caller executes the 'call' instruction:
   - %eip points to the first instruction of callee
   - %esp + 4  points to the arguments
   - %esp points the return address

Immediately after callee executes the 'ret' instruction:
   - %eip contains the return address
     (i.e., causing execution to resume in caller)
   - %esp points to the arguments
   - Any or all arguments may be written over by the callee
   - %eax contains the result computed by the callee
   - %ecx,%edx  may contain arbitrary values
   - %ebp,$ebx,$esi,%edi must contain orginal contents at point of 'call'

        GCC calling convention for the x86
   (in terms of a contract between caller and callee)
```

Since, arguments are stored on the stack and stacks grown down this can be visualized as follows:

```
        ARG N                         ARG N (or trash)
        ARG N-1                       ARG N-1 (or trash)
         .                             .
         .                             .
         .                             .
        ARG 2                         ARG 2 (or trash)
        ARG 1                %ESP ->  ARG 1 (or trash)
%ESP -> ret address
```

```
Following 'call'           Following 'ret'

                Stack Layout
```

This section has described the **minimal** caller/callee contract–what actions are absolutely **required** of the caller/callee. The next sections are going to describe how GCC typically implements nested procedure calls. The implementation performs more actions than specified by the minimal contract. In fact, the implementation is free to do any thing it chooses just so long as it does not violate the caller/callee contract.

5.1 GCC's implementation of Stack Frames

Stack memory is organized into regions called **stack frames**. Each procedure has a stack frame and, while executing, stores all of it's intermediate state there.

The %esp and %ebp registers are the processor registers used to support this abstraction; they demarcate the beginning and end of the current stack frame. For example, while executing within a procedure, the %ebp points to the base of the stack frame, and the %esp points at end of the stack frame.

The code in Figure 2 and the Figure 3 show the individual steps taken to create and destroy stack frames. The figures also show the overall structure of the stack frames and how they are linked together to support nested procedure calls.

```
int plus3 (int x) { int res = x + 3; return res; }
int doit (int x) { return plus3 (x); }
int main (void) {  return doit (8); }
_plus3:
     pushl    %ebp
     movl     %esp, %ebp
     pushl    %esi
     movl     8(%ebp), %esi
     addl     $3, %esi
     movl     %esi, %eax
     popl     %esi
     movl     %ebp, %esp
     popl     %ebp
     ret
_doit:
     pushl    %ebp
     movl     %esp, %ebp
     pushl    8(%ebp)
     call     _plus3
     movl     %ebp, %esp
     popl     %ebp
     ret
_main:
     pushl    %ebp
     movl     %esp, %ebp
     pushl    $8
     call     _doit
     movl     %ebp, %esp
     popl     %ebp
     ret
```

Figure 2: Example code for Figure 3

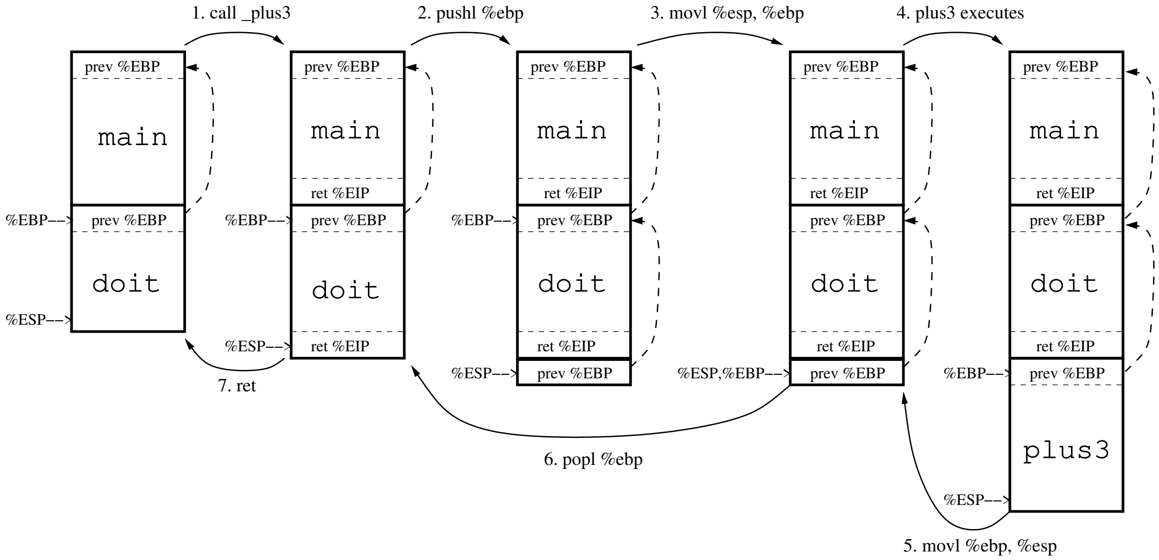

Figure 3: Nested stack frames. Shows, step-by-step, the creation and destruction of a stack frame. The stack frame for each function is outlined with a thick black line. Initially, doit is the active stack frame. **Step 1**: doit calls plus3. The call instruction places the return address on the stack. **Step 2**: plus3's prologue saves doit's stack frame base pointer (%ebp). **Step 3**: plus3's prologue sets %ebp to point at the start of its stack frame. Throughout the execution of plus3 %ebp never changes; it always points at the base of plus3's stack frame. **Step 4**: plus3 executes, during which it alllocates more stack space. This causes %esp decrease (recall on x86 stacks grow down). The addresses between %ebp and %esp constitute plus3's stack frame. **Step 5**: plus3's epilogue deallocates (most) of its stack frame with `movl %ebp, %esp`. Note: This instruction is oblivious of how much stack space was actually allocated, since %ebp never changed. **Step 6**: plus3's epilogue restores doit's base pointer with `popl %ebp`. Executing this instruction also deallocates the final word of plus3's stack frame. **Step 7**: plus3 returns to doit. The `ret` instruction pops the return address off the stack and into the %eip.
Note: the `leave` instruction is identical to executing `movl %ebp, %esp; popl %ebp`. The compiler may opt to emit it. In which case, steps 5 and 6 above are collapsed into one step.

5.2 Stack Frame Contents for GCC

This section looks in more detail at the content of the stack frames. It does this by examining the exact steps taken to perform a procedure call. This is illustrated in Figure 4.

The figure shows the calling convention used by GCC. Bear in mind that the x86 architecture does not impose this particular convention. For example, x86 does not specify which registers are callee and which are caller. This was specified by GCC. The only aspect of the calling convention prescribed by the x86, is the fact that stacks grow down.

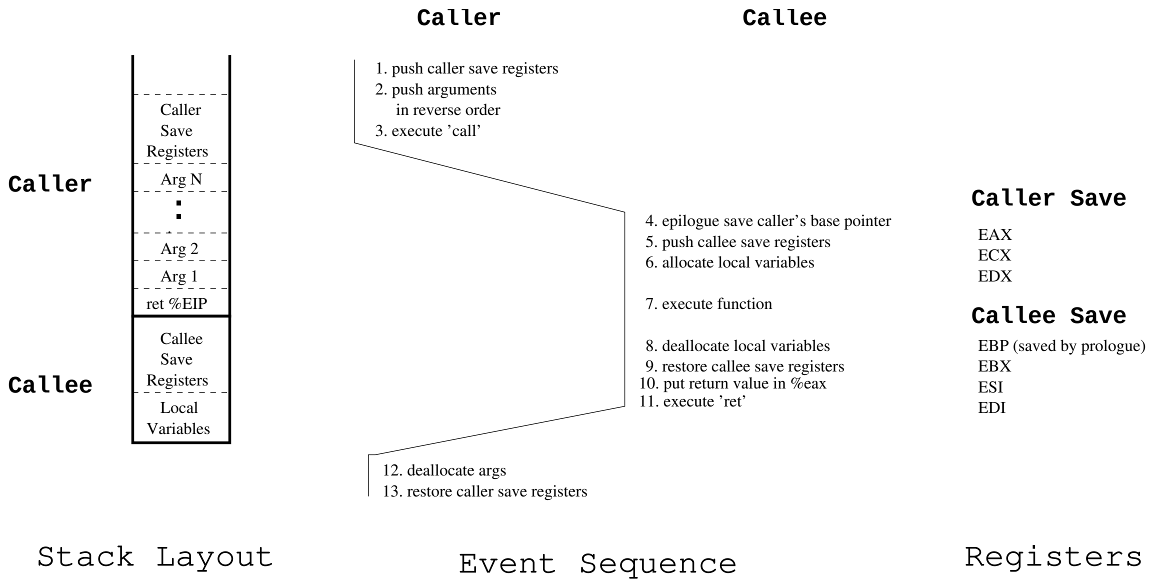

Figure 4: GCC calling convention for the x86.

6 PC architecture – what about IO?

It's fine and dandy to know all about the instruction set. But what can you really do with it? You can compute a lot of interesting results and store them in registers or in memory. What you really want to do is save them to disk, email them, or print them. Therefore, what we really need to know is the IO interface.

What ever appears on the CPU's address lines is termed the physical address. The system controller interposes between the physical address and memory (and the devices) to perform memory mapped IO. For example, a physical address of 0xb8000 is re-directed by the system controller to the video card.

In addition, devices can be access with the `in/out` instruction. (see `read_sector` from lab 1).

```
+---------------+ ctl     +------------+ ctl     +------------+
|               |---------|            |---------|            |
|               |         |  SYSTEM    |         |            |
|               |address  |            |address  |            |
|    CPU        |---------|            |---------|   DRAM     |
|               |         | CONTROLER  |         |            |
|               |data     |            |data     |            |
|               |---------|            |---------|            |
+---------------+         +------------+         +------------+
                              |    |
                           +--+----+---------------+
                PCI        |  |    | video card    |
                           +--+----+---------------+
                BUS           |    |
                           +--+----+---------------+
                           |  |    | network card  |
                           +--+----+---------------+
                              |    |
                         address   data
```

Figure 5: PC block diagram.