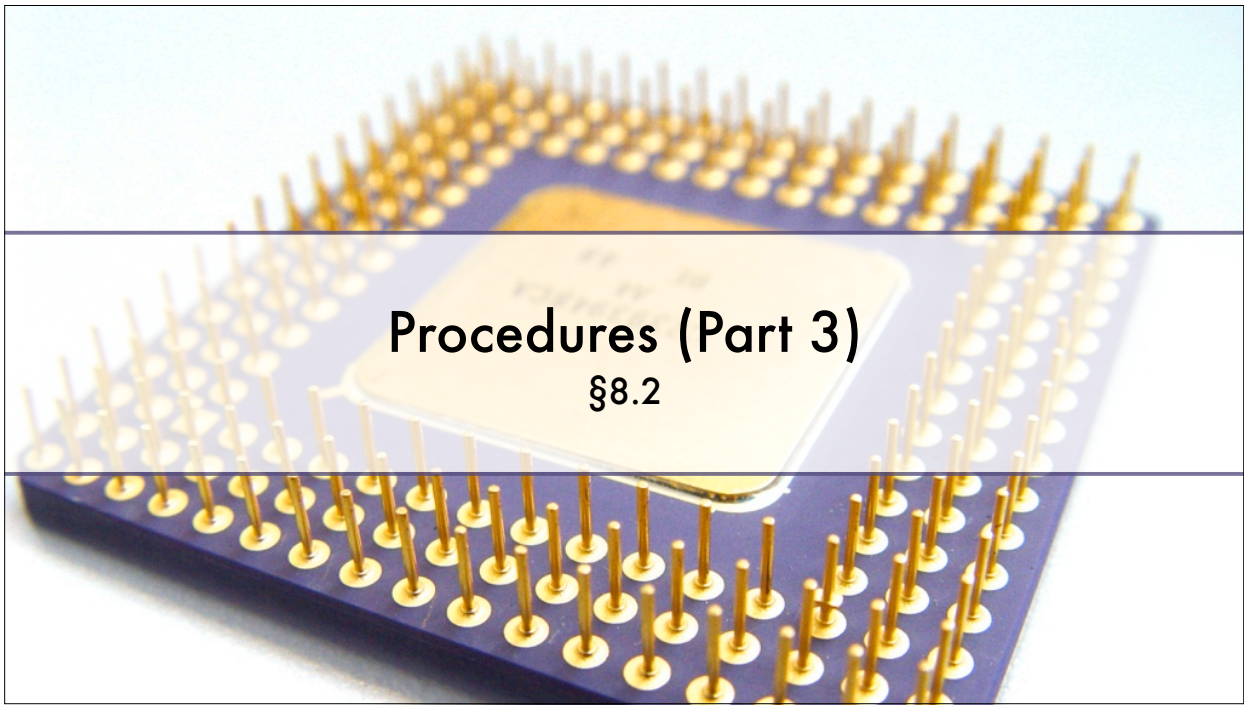
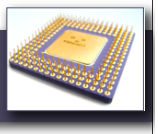


Procedures (Part 3)

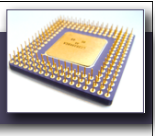
§8.2

Stack Arguments & Locals



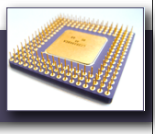
- Until now,
  - We have been passing arguments in registers
  - We have only used global variables (`.data`)
- Now,
  - We will **pass arguments** on the stack (you did this in Lab 4)
  - We will learn how to **store *local variables*** on the stack

Stack Frames



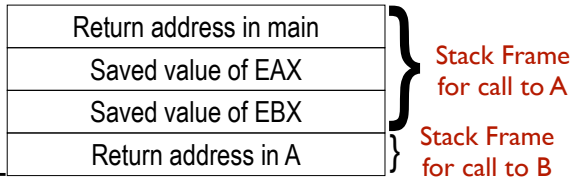
- We have used the runtime stack to
  - save the address a procedure should return to (CALL)
  - save register values inside a procedure (PUSH)
- So, there may be several elements on the stack all relating to the same procedure call
  - E.g., return address, saved register values
- Collectively, the part of the stack containing the return address, saved registers, etc. for a procedure call is called a *stack frame* or *activation record*
  - Stack frames will also contain arguments, local variables

Stack Frames

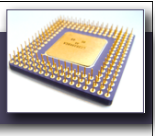


```
main PROC
    call A
    exit
main ENDP

A PROC
    push eax
    push ebx
    call B
    pop ebx
    pop eax
    ret
A ENDP

B PROC
    ret
B ENDP
```

at this point, the stack contains:



Stack Arguments



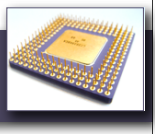
- Procedures written in high-level languages (e.g., C/C++) receive arguments *on the stack*, not in registers
- Arguments are conventionally pushed from *right to left*
- In C:

```
AverageOf3(10, 20, 30);
```

- In assembly:

```
push 30
push 20
push 10
call AverageOf3
```

Local Variables

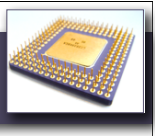


- *Local variables* (AKA "locals" or "temporary variables") are created "fresh" each time a procedure is invoked and disappear when the function returns
- Local variables are especially important when implementing recursive procedures (What would happen below if `sum` were a global variable?)

```
int BadSum(unsigned int n) {
  int sum;
  sum = n;
  if (n > 0) sum += BadSum(n-1);
  return sum;
}
```

  - You do **not** need local variables for the recursive procedure in Homework 4. If your procedure stores values in registers and pushes/pops them, then registers are a lot like local variables.
- Local variables are especially useful when you run out of registers

Creating a Stack Frame



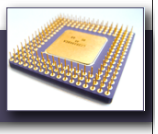
- **At the call site (i.e., inside the calling function)...**

1. Push arguments onto the stack
2. Call the subroutine (CALL pushes the return address)

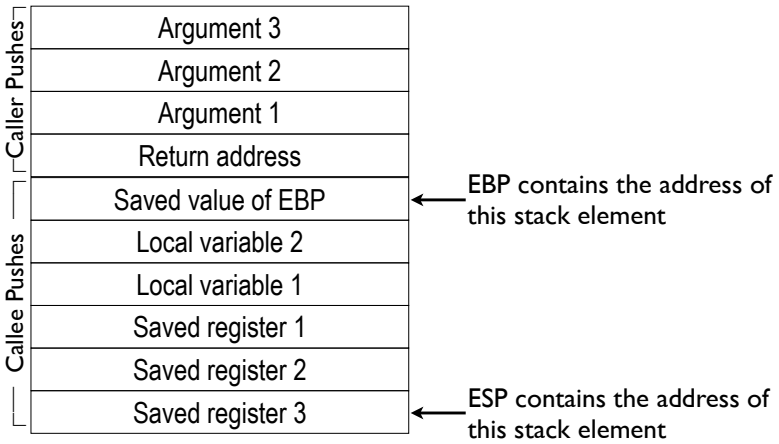
- **Inside the function being called...**

3. Push EBP (it will be used to retrieve the arguments)
4. Set EBP equal to ESP
5. Decrement ESP to allocate stack storage for locals
6. Save register values by pushing them on the stack

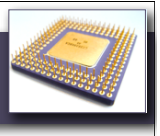
Creating a Stack Frame

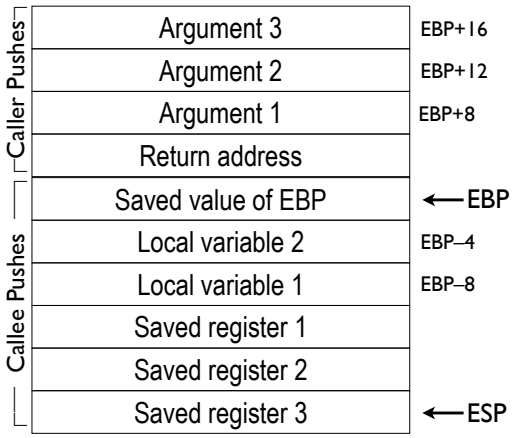


- After the stack frame is created...



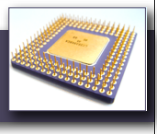
Accessing Args & Locals





- Three 32-bit arguments are at
  - [ebp+8]
  - [ebp+12]
  - [ebp+16]
- Two 32-bit local variables are at
  - [ebp–4]
  - [ebp–8]

Terminating a Stack Frame



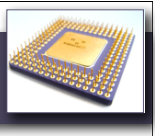
- **Inside the function being called...**

1. If the function returns a value, put it in EAX
2. Pop register values off the stack
3. Set ESP equal to EBP to remove local variables
4. Pop EBP
5. Return (RET pushes the return address); if using the *STDCALL calling convention,* remove arguments by supplying an immediate operand to the RET instruction

- **Back in the calling function...**

6. If using the *C calling convention,* remove arguments

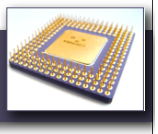
Calling Convention



- A *calling convention* specifies how a procedure receives parameters and returns a result
  - How are arguments passed to the procedure: in registers or on the stack?
  - In what order are arguments pushed on the stack?
  - **Who removes arguments from the stack?**
  - What other steps are taken by the caller vs. callee before and after the function executes?
  - What registers may be overwritten by the callee?

Source: http://en.wikipedia.org/wiki/Calling_convention

STDCALL Calling Convention



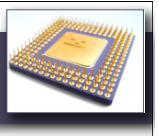
- Arguments are pushed from right to left
- The procedure issues a RET instruction *with an immediate operand* to remove arguments
- The instruction `ret 8` means "pop the return address and set EIP, then add 8 to ESP to remove arguments"

```
push 6
push 5
call AddTwo
```

```
AddTwo PROC
   push ebp
   mov ebp, esp
   mov eax, [ebp+12]
   add eax, [ebp+8]
   pop ebp
   ret 8
AddTwo ENDP
```

STDCALL: Callee removes arguments from the stack

C Calling Convention



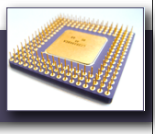
- Arguments are pushed from right to left
- After the CALL instruction, the caller adds a value to ESP to remove arguments from the stack

```
push 6
push 5
call AddTwo
add esp, 8
```

C Calling Convention: Caller removes arguments from the stack

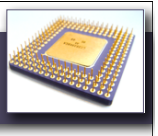
```
AddTwo PROC
   push ebp
   mov ebp, esp
   mov eax, [ebp+12]
   add eax, [ebp+8]
   pop ebp
   ret
AddTwo ENDP
```

Which Calling Convention?

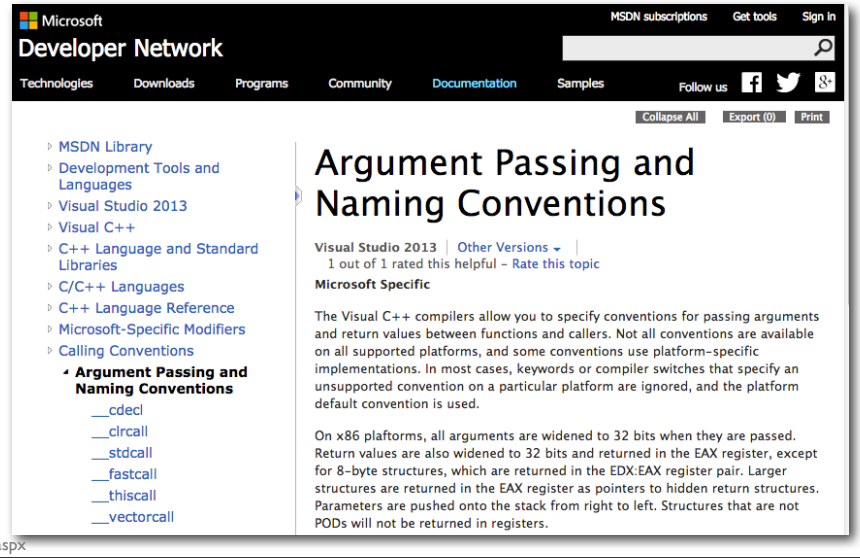


- The C calling convention allows functions with a variable number of arguments, like *printf*
  - `printf("OK"); printf("%d%s\n", 3, OK);`
  - With STDCALL, every procedure must have a fixed number of arguments, since the function must supply an immediate value to the RET instruction
- But with the C calling convention, you *must* remember to clean up the stack after every CALL!
- Windows API functions use STDCALL
- **We will use STDCALL in the future**

Calling Conventions

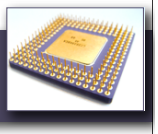


- There are more calling conventions that we won't use
- E.g., FASTCALL: like STDCALL but first two args in ECX and EDX, not on the stack



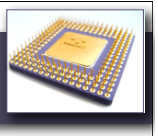
http://msdn.microsoft.com/en-us/library/984x0h58.aspx

Prologue & Epilogue



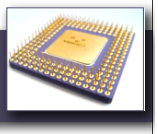
- Inside a procedure...
  - The procedure's *prologue* consists of the instructions to push EBP, set it equal to ESP, reserve space for locals, and save registers
  - The procedure's *epilogue* consists of the instructions to pop registers, remove local variables, restore EBP, and return to the caller

Symbols for Args & Locals



- For better readability, define symbolic constants for parameters and local variables inside the procedure using the EQU directive

```
AddTwo PROC
   arg1 EQU DWORD PTR [ebp+8]
   arg2 EQU DWORD PTR [ebp+12]
   push ebp
   mov ebp, esp

   mov eax, arg1   ; Same as mov eax, DWORD PTR [ebp+8]
   add eax, arg2   ; Same as add eax, DWORD PTR [ebp+12]

   pop ebp
   ret 8
AddTwo ENDP
```



Activity 11