Procedures Functions and Structures

We will see:

- What procedures and functions are
- How they can be implemented in machine code
- How parameters can be passed to a procedure (or function)
- How results can be returned from a procedure (or function)
- Why we need to save the state of the machine during a procedure call

Program Structuring

"Spaghetti-Code"

- Unrestricted use conditional and unconditional jumps
- Very hard to understand
- Akin to using **`GOTO`** in **`BASIC`**

Structured Programming

- No **`GOTO`**'s
- Only **`IF`**, **`WHILE`**, **`FOR`**, **`DO`**
- Plus *procedure* and *function* calls

Defining a Procedure

A procedure is a named block of statements that end in a return instruction (RET).

It is defined by using two directives PROC and ENDP.

It must be assigned a name.

So far we have written our programs with the PROC/ENDP structure and we have called it *(main)*

Defining a Procedure con’t

When you create a procedure other than your programs startup (main), you need to end the procedure with a RET instruction.

```
myProc PROC
 .
 .
 .
 ret
myProc ENDP
```

Note: our startup (main) is a special case because it ends with (exit)

PROC example

Sum of Three Integers:

```
sumOf3    PROC
     add  eax,ebx
     add  eax,ecx
     ret
sumOf3    ENDP
```

Calling Procedures

A procedure is **CALL**ed from one or many different locations within a program.

At the end of a procedure the **RET** instruction tells the processor how to **RET**urn to where it was called from.

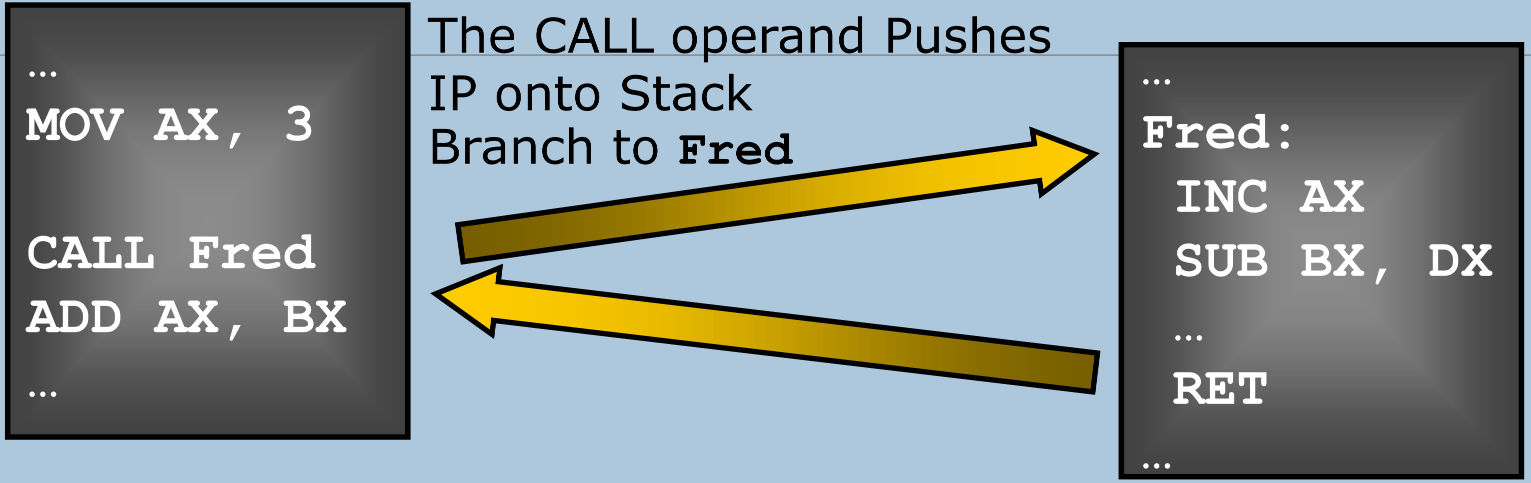
Calling Procedures con’t

Before a procedure is called the processor **PUSH’**es the address of the instruction immediately following the call onto the **STACK**.

When the **RET** instruction is reached the processor **POP’**s the address of the next instruction off of the stack.

Procedure is another way of referring to a subroutine.

Procedures



```
…
MOV AX, 3

CALL Fred
ADD AX, BX
…
```

```
…
Fred:
  INC AX
  SUB BX, DX
  …
  RET
…
```

Call of procedure "Fred"

Pop address from stack and branch to it

Definition of Procedure "Fred"

Like a branch instruction

- But control is returned to the point of call on completion

You can think of a procedure as a new instruction

- Only need to know *what* it does, not *how* it does it

Very useful method for structuring programs

See simpleProc.asm

Passing Parameters

- We often need to pass parameters to a procedure (in python this would be a (def)
  - E.g. "**`def name(`***<PARAMETER(s)>*`)`"
- Need to decide *how* to pass each parameter:
  - By value
  - By reference
- Need to decide *where* to pass the parameters:
  - In registers
  - In global variables
  - On the stack
  - In a parameter block
  - In the code-stream

Passing Parameters By Value

The procedure receives a *copy* of the parameter stored in AX.

```
ADDING Proc
        MOV    BX,0
        MOV    BL,AH
        MOV    AH,00
        ADD    BX,AX
        RET
ADDING endp
```

This routine adds two bytes stored in AH and AL together and puts the result in BX

See simpleProc.asm

Passing Parameters By Reference

To pass a parameter by reference, you must pass the address of a variable rather than its value. i.e. You must pass a pointer to the data. The procedure must dereference this pointer to access the data.

The procedure receives an address of a variable

The called procedure is given the opportunity to modify the variables contents.

A good rule of thumb is that you only pass by reference if you expect the procedure to modify the variable.

Passing parameters by reference can produce some peculiar results.

Pass by reference is usually less efficient than pass by value

IGNORE code on your slides will provide a better example.

Good for passing large data-structures

- E.g. Objects, arrays,…

Passing Parameters using Registers

Registers are good places to pass a small number of bytes to a procedure

Single parameter:

- Byte: **`AL`**
- Word: **`AX`**
- Double Word: **`DX:AX`**

If passing several bytes, use:

- **`AX -> DX -> SI -> DI -> BX -> CX`**

Pass by value = move *parameter* into the required register

Pass by reference = move *address* of parameter into required register

Example: Using Registers

Simple procedure to output a given character a given number of times

```
def writeChar(c, reps):
    for counter in range (reps):
        print(c),

writeChar('C',10)
```

We will pass the parameters in the AX register

- **`AL`** = *c*, the character to be output
- **`AH`** = *reps*, the number of times to output it

We will *reverse* the loop so that we can use the **`LOOP`** instruction

- This is more efficient (see procReg.asm)

Passing Parameters using Global Variables

If there are insufficient registers, then we can use global variables

- Pass by value = move *data* into the correct memory location
- Pass by reference = move *address* of data into the correct memory location

This method is very easy to use, but has a number of disadvantages:

- Inefficient
  - Need to store and retrieve data from main memory
  - Can't do recursion

Example: Using Global Variables

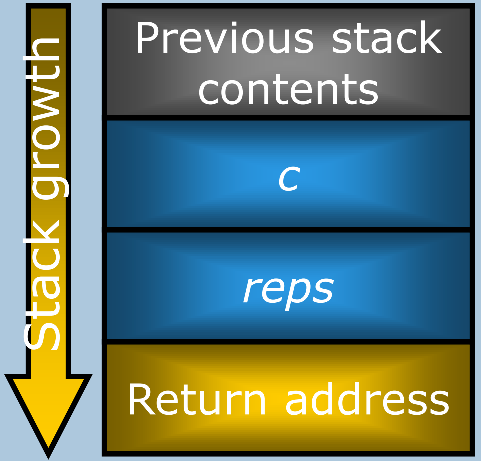
We first define the variables we want to pass to the procedure in **(.DATA)**

Before calling the procedure, we load the parameters into their corresponding global variables

Inside the procedure, we can access the parameters directly through the global variables.

See procGlob.asm

Passing Parameters using the Stack



```
                def writeChar(c, reps):
                    for counter in range (reps):
                        print(c),

…               writeChar('C',10)
PUSH c
PUSH reps
CALL writeChar
…
```

Push the parameters for the procedure onto the stack

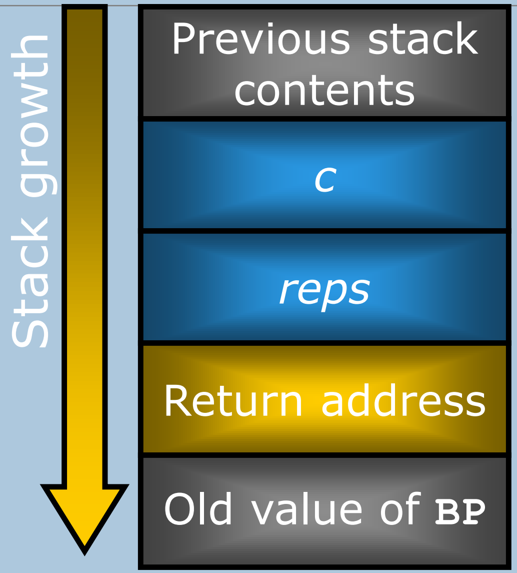
The procedure can then:

- Pop them from the stack, or
- Access them directly through the stack pointer
    - Procedure needs to remove them before exiting

This is the best general-purpose method for passing parameters

See procStak.asm

Example: Using the Stack



Each stack entry is 2 bytes, so
c can be accessed using **`BP[6]`**
reps can be accessed using **`BP[4]`**

We can use **`EQU`** to make the code easier to understand:

```
c EQU BP[6]
reps EQU BP[4]
```

The best way to access parameters on the stack is through an index register (e.g. **`BP`**)

- Save the old value of the register on the stack
- Set it equal to the top of the stack
- We can now access parameters using fixed offsets from the top of the stack

This is a good place to use *indexed addressing*

Saving the State of the Machine

```
    MOV CH, 0          ; Setup CX so we can use LOOP
    MOV CL, 10         ; We will print the first 10 chars
    MOV AL, 'a'        ; The first char to output
LOOPSTART:
   PUSH AX             ; Parameter - the char to output
   PUSH 20             ; Parameter - the number of repeats
   CALL WRITE_CHAR
   INC AL              ; Load next character
LOOP LOOPSTART         ; Repeat until all 10 are done
```

What happens if we call writeChar in a loop?

- Big problem!

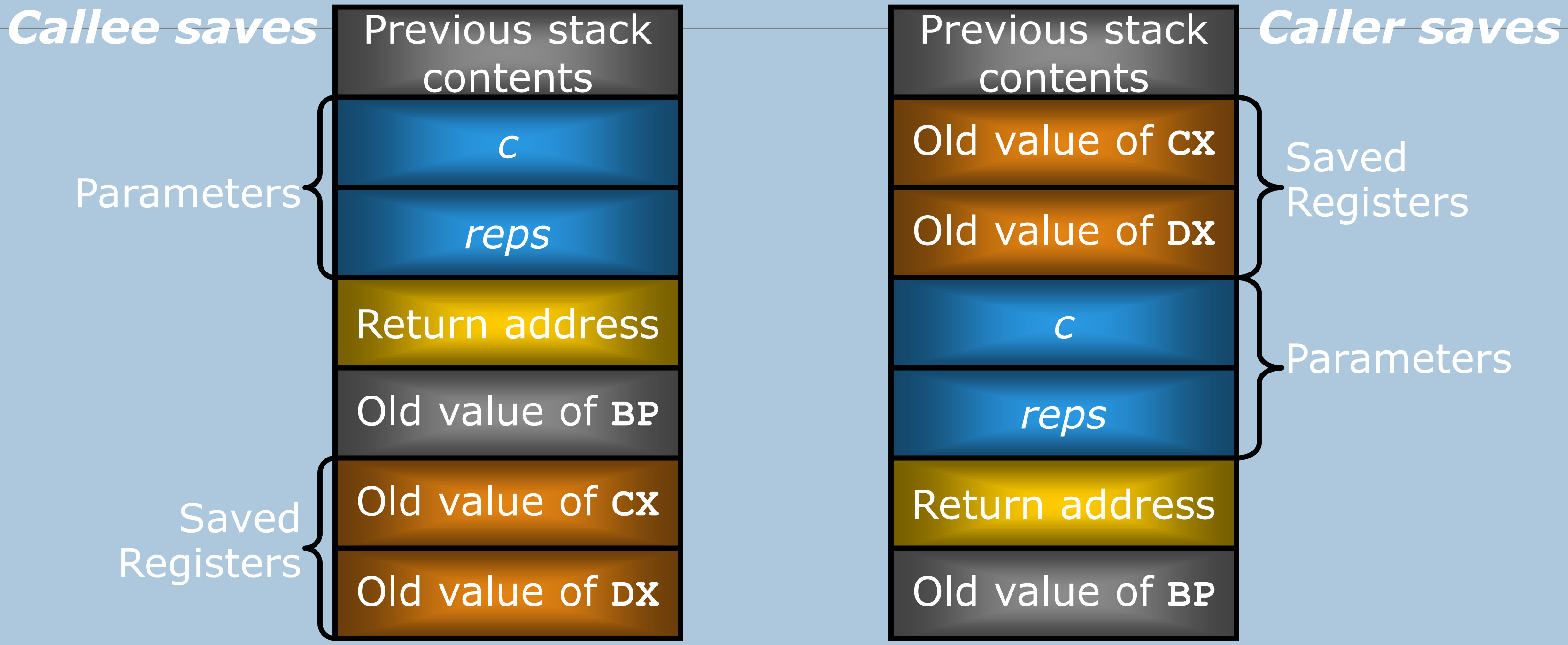
This occurs because the procedure uses registers that are already in use

- `CL` is used in `writeChar`, *and* in the loop which calls it

We need to *save* any registers which might be affected by the procedure

- We can save them on the stack

Saving Registers



Caller saves: push registers, then call procedure, then restore registers

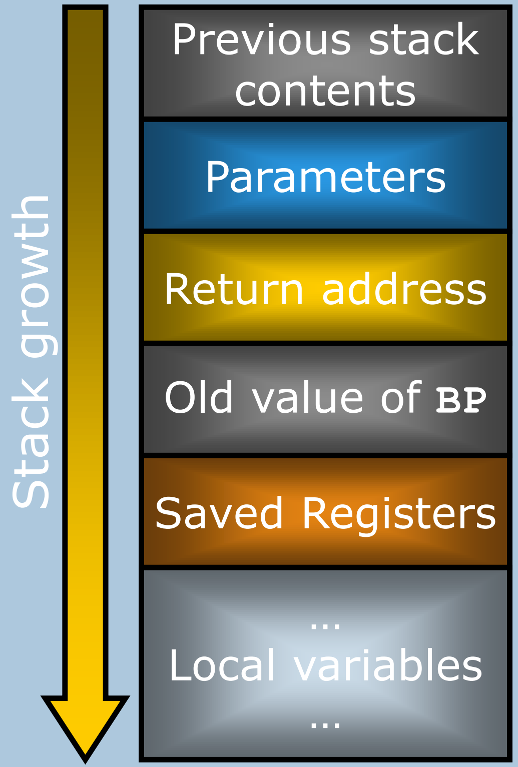
- Can be hard to maintain

Callee saves: procedure saves registers, and restores them before returning

- Can lead to unnecessary stack operations

Local Variables

We can also use the stack to store local variables in a safe manner



We *subtract* from **`SP`** (the stack pointer) to allocate some space on the stack:

```
    SUB SP, 2; allocate 2 bytes
```

Once we're done, we *add* to it to remove them:

```
    ADD SP, 2; deallocate 2 bytes
```

We can use **`EQU`** again to make the code simpler:

```
        var1 EQU BP[-4]
        var2 EQU BP[-6]
```

Returning Values: *Functions*

A function is a procedure which returns a value

We return values from a procedure in much the same way as we can pass parameters to it

Use the **`AX`** (or other) register

- This is the best option
- Don't need to save it beforehand

Use the Stack

- Value at the top of the stack after call is the result

Push Flags to Stack

There are times when it is necessary to save the state of each flag in the processors flag register.

This is usually done whenever the processor is interrupted.

Saving the flags and restoring them at a later time, along with the processor registers is a proven technique for resuming program execution after an interrupt.

Pushing flags con’t

To make life easy the 80x86 family has an instruction to do this automatically.

This is PUSHF with it counter part POPF.

Recursion

This is where one procedure (or function) calls itself:

```
fib(0) = 1
fib(1) = 1
fib(N) = fib(N-1) + fib(N-2)
```

- E.g. `fib(4) = fib(3) + fib(2) = 5`

Very useful for expressing algorithms:

- Sorting (quicksort, mergesort,…)
- Trees and graphs (shortest-path, spanning tree…)
- Mathematical functions (factorial, Fibonacci,…)

Recursion only works if we pass parameters using the stack

Summary

Procedures and functions are the most important concepts for structuring programs

There are many ways to pass parameters to a procedure

- Passing them via the stack is best
- Using the stack also allows *recursive* procedures

To avoid problems, we also need to save the state of the machine

- This is like the interrupt mechanism (**`INT`**)

We can also use the stack to hold local variables

We can now see how a compiler translates procedures into machine code