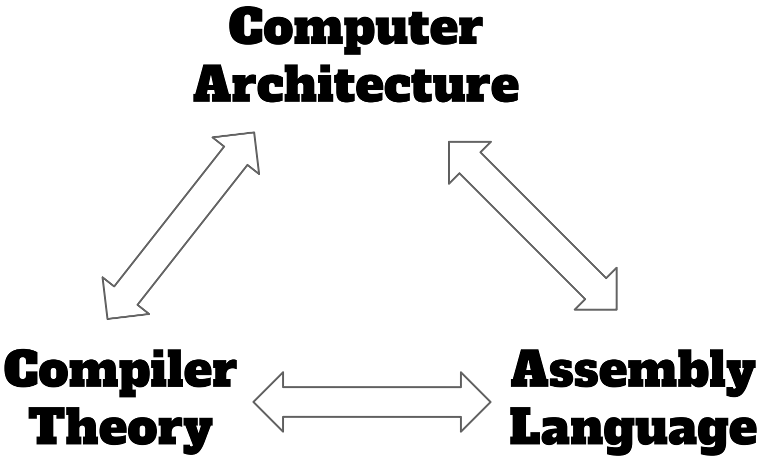
Link with High-level language

CSC 236

Linking

- Combine code from:
  - Assembly language
  - High-level languages
    (as long as they compile down
    to object files)
- High-Level Languages?
  - This works for native-compiled languages
    - C, pascal, rust, swift, …
  - … not for interpreted languages
    - Java, python, javascript …



Computer Architecture

Compiler Theory

Assembly Language

Applications

- Written in a high-level language
- Access to devices
  - Disk
  - Display
  - Network card
- How
  - Via the Operating System
  - Ultimately through Device Drivers in the OS
  - Some part of each device driver is typically in assembly
  - Access to “hidden” architecture

Link ASM & HLL

- Why put together assembly and some other language
    - Access special instructions and hardware
    - Optimize critical sections of code for speed / size
- Rules are set by HLL
    - A HLL depends on particular calling conventions
    - The rules are determined by:
        - The compiler and processor
        - The host operating system designers
        - The particular language.
- Assembly code must
    - Behave like a subroutine / caller in that particular language

Stack

- C / 8086 compiler passes parameters on the stack

In 8086

- Stack holds words only (no bytes)
  - Push / pop words
- Push
  - Decrement SP by 2
  - Store word at SP
- Pop
  - Remove word at SP
  - Increment SP by 2

- **Words only**
- **Stack grows down**
- **Push pre-decrement**
- **Pop post-increment**
- **SP points to “top of stack”**

Parameters

- Different types of parameters
- Numeric vs character
  - The subroutine must know what type
- Signed vs unsigned
  - The subroutine must know what type
- What about size?
  - Stack only holds words
  - What if parameter is not a word?

Byte parameter

- Stack holds words
- Caller
  - Convert to word
  - Push
- Callee
  - Pop
  - Convert to byte

String

- Pass address
  - `string    db ‘abcde’,’$’`
  - Pass address of string

Array

- Note
  - String is a 1-D character array
- Pass address
- What if 2-D array?
  - Pass address
  - Subroutine must know that this is 2-D array
  - Semantics for accessing elements is part of the subroutine code.
  - That’s the case for any subroutine parameters (structs, linked lists …)

Subroutine protocol

- Do not declare stack (no .stack directive)
- Do not set DS register
- Declare routine public
    - C linking convention prepends _ (underscore before name)
    - For example: **sub** -> **_sub**
- If a component doesn’t contain main(), no label on **end** directive
- Live registers
    - BP, SI, DI, SS, DS
    - Must save/restore any live register that’s modified

Example

```
// C main

rc = asmprint(‘x’,5);
```

- asmprint(char, count)
  - Print char to display count times

asmprint(‘x’, 5) ⇒ xxxxx

Example

```
extern int asmprint(char c, int n);

int main( )
{
   // declare the return code
   int rc;

   // call routine c = 'x' &  n = 5
   rc = asmprint('x', 5);

   // exit the program
   return 0;
}
```

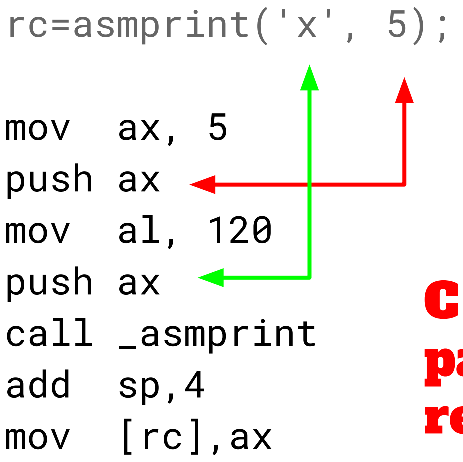
- Code generated by compiler

```
rc=asmprint('x', 5);

mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

Example

```
extern int asmprint(char c, int n);

int main( )
{
   // declare the return code
   int rc;

   // call routine c = 'x' &  n = 5
   rc = asmprint('x', 5);

   // exit the program
   return 0;
}
```

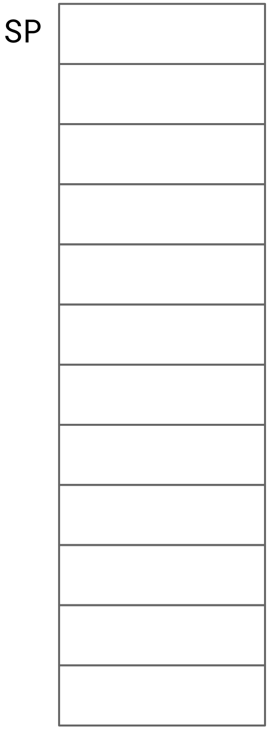
- Code generated by compile



```
rc=asmprint('x', 5);

mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

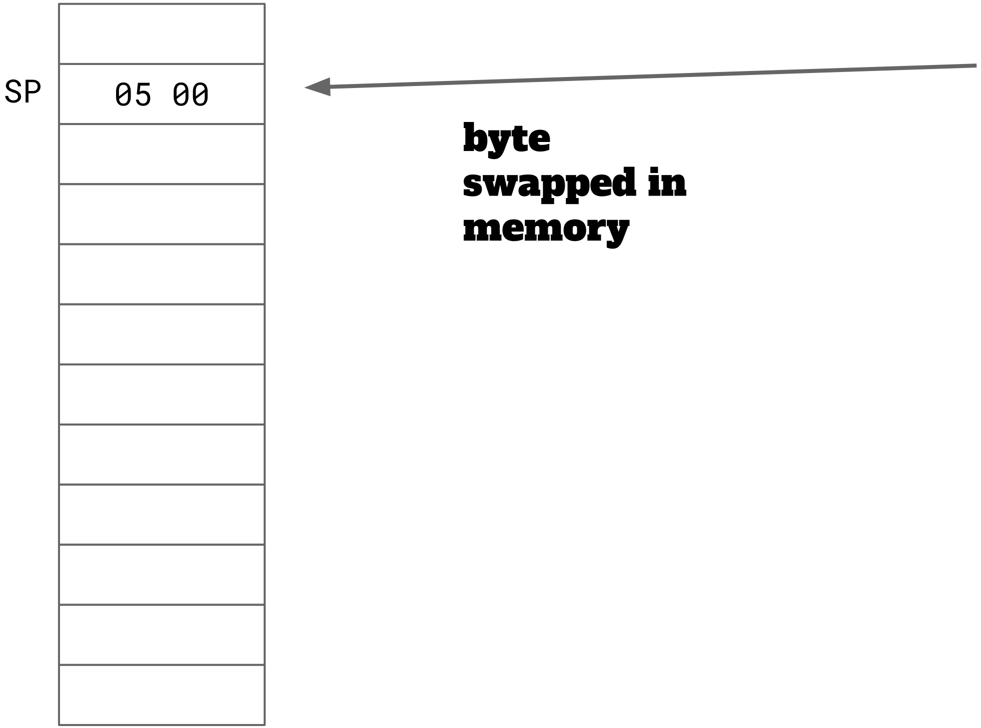
C pushes parameters in reverse order

How to access parameters on the stack



```
mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

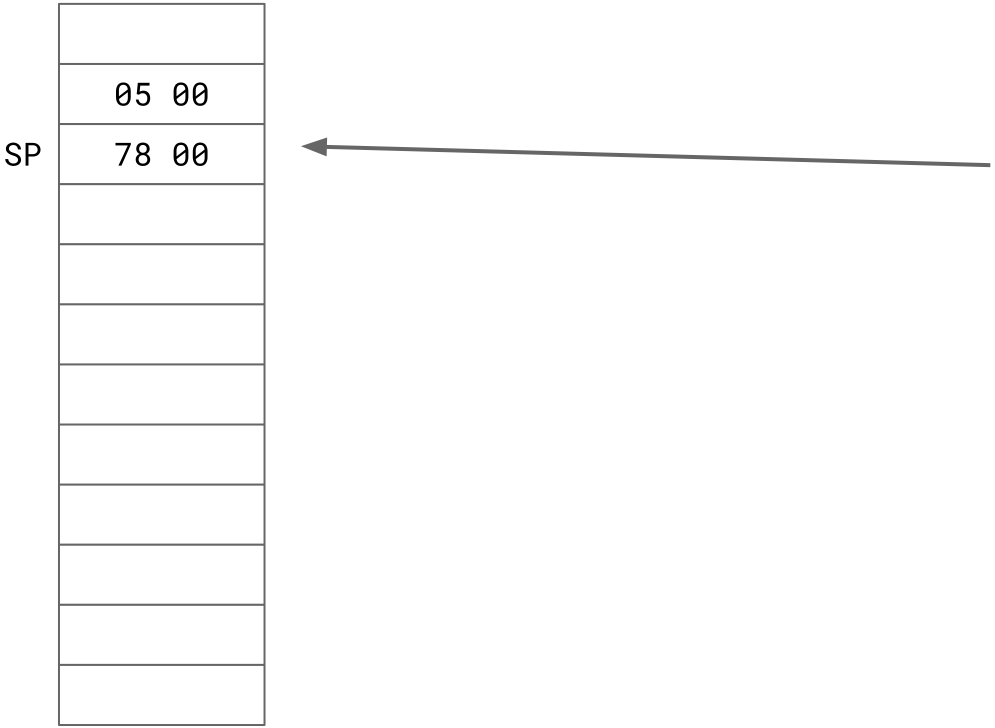
How to access parameters on the stack



byte swapped in memory

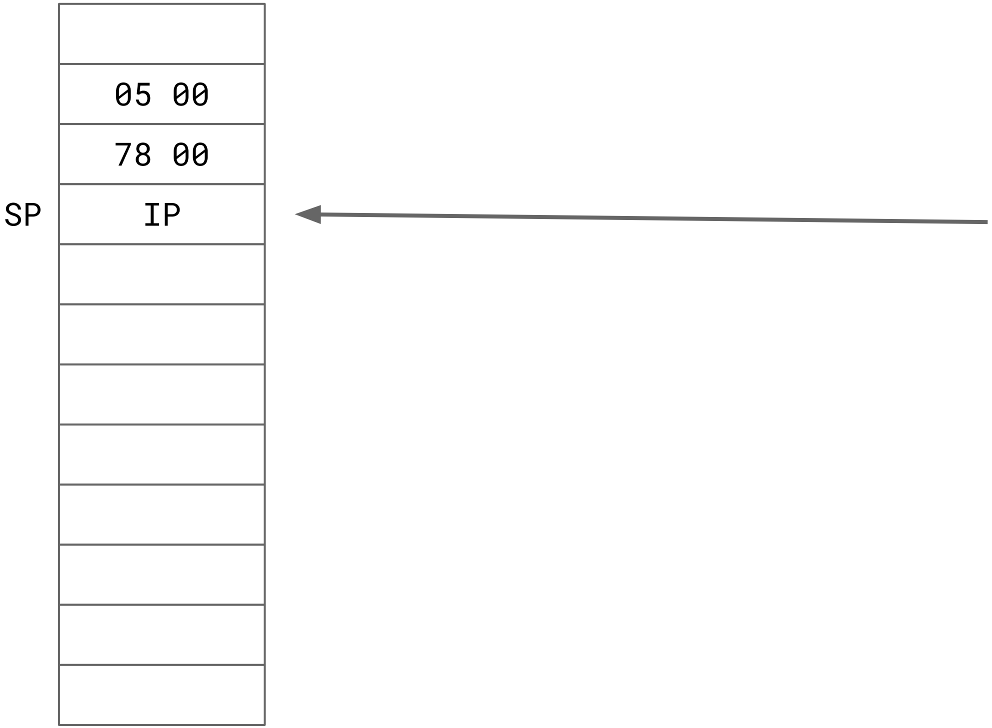
```
mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

How to access parameters on the stack



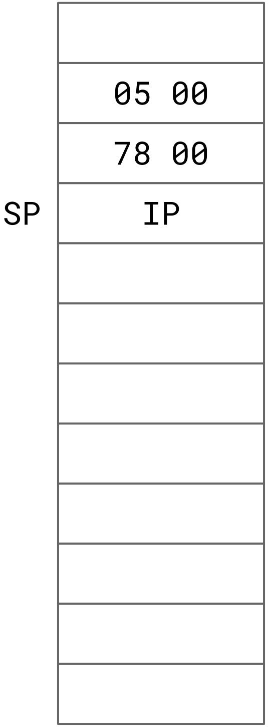
```
mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

How to access parameters on the stack



```
mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

How to access parameters on the stack



05 00

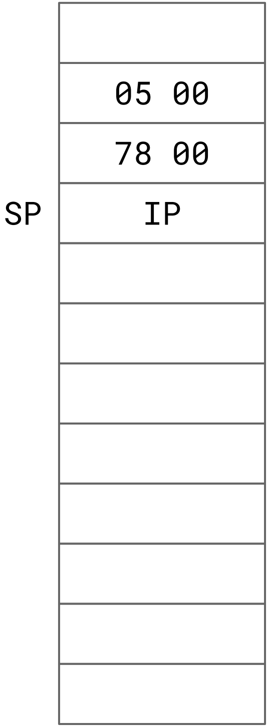
78 00

SP IP

```
mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

What is this for?

How to access parameters on the stack



05 00

78 00

SP IP

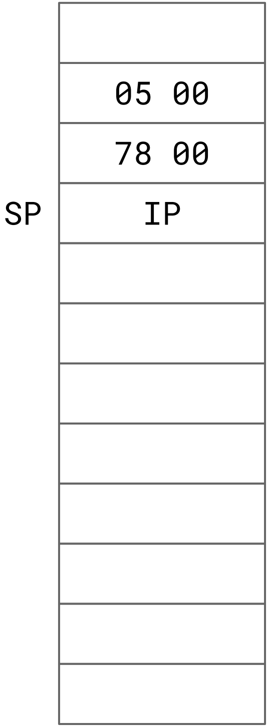
```
mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

What is this for?

It's equivalent to two pops … in just one instruction.

One reason why stack space is so cheap.

How to access parameters on the stack

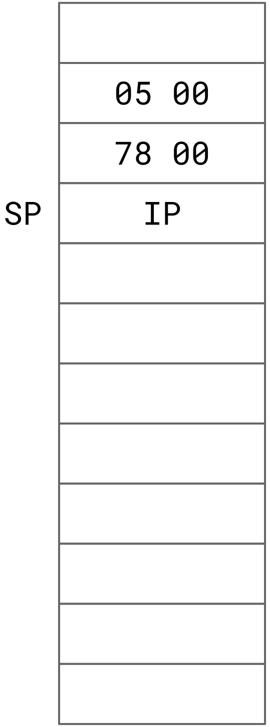


	05 00
	78 00
SP	IP

Now we look at the subroutine

```
_asmprint:
   push  bp        ;save bp
   mov   bp,sp         ;setup bp
   push  si        ;save si
   push  di        ;save di
   mov   dx,[bp+4]     ;0078  "x"
   mov   cx,[bp+6]     ;0005  count
x: mov   ah,2      ;dos code
   int   21h       ;write
   loop  x         ;repeat
   pop   di        ;restore di
   pop   si        ;restore si
   pop   bp        ;restore bp
   mov   ax,0      ;set rc
   ret             ;return
```

How to access parameters on the stack



05 00

78 00

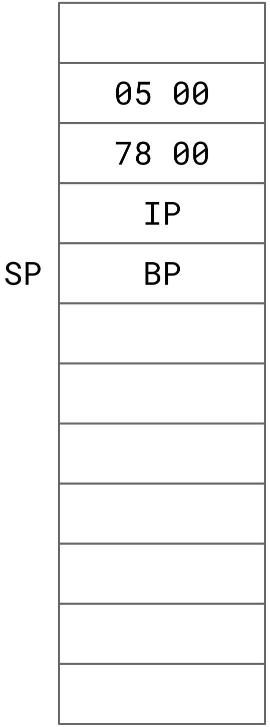
SP IP

Save registers:
BP, SI, DI, SS, DS
(if used)
Only using BP

Pretend we also use SI & DI

```
_asmprint:
   push  bp        ;save bp
   mov   bp,sp         ;setup bp
   push  si        ;save si
   push  di        ;save di
   mov   dx,[bp+4]     ;0078  "x"
   mov   cx,[bp+6]     ;0005  count
x: mov   ah,2      ;dos code
   int   21h       ;write
   loop  x         ;repeat
   pop   di        ;restore di
   pop   si        ;restore si
   pop   bp        ;restore bp
   mov   ax,0      ;set rc
   ret             ;return
```

How to access parameters on the stack



05 00

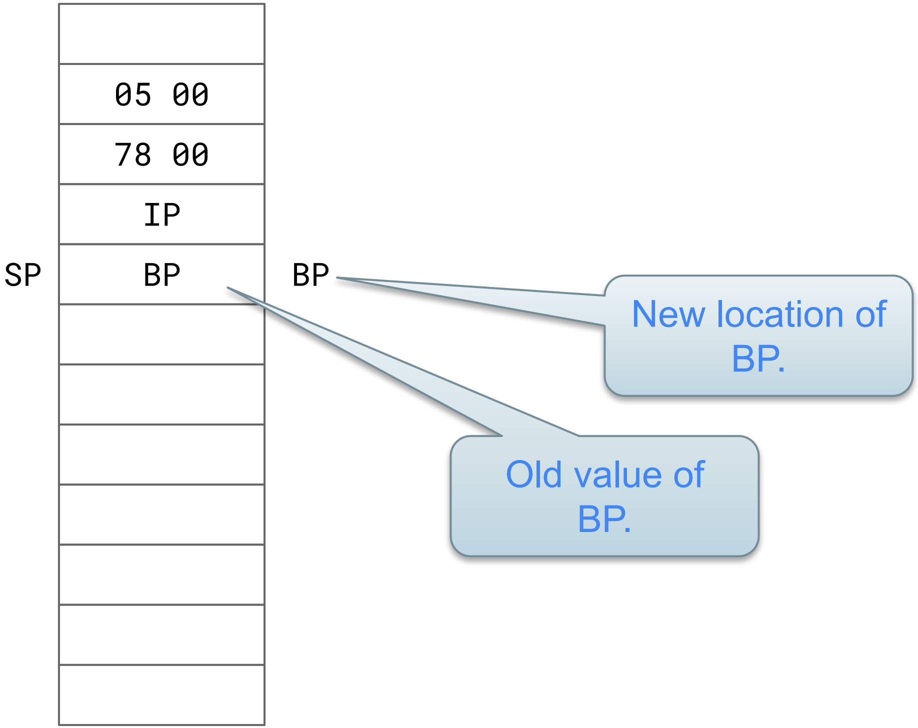
78 00

IP

SP BP

```
_asmprint:
   push  bp        ;save bp
   mov   bp,sp         ;setup bp
   push  si        ;save si
   push  di        ;save di
   mov   dx,[bp+4]     ;0078  "x"
   mov   cx,[bp+6]     ;0005  count
x: mov   ah,2      ;dos code
   int   21h       ;write
   loop  x         ;repeat
   pop   di        ;restore di
   pop   si        ;restore si
   pop   bp        ;restore bp
   mov   ax,0      ;set rc
   ret             ;return
```

How to access parameters on the stack

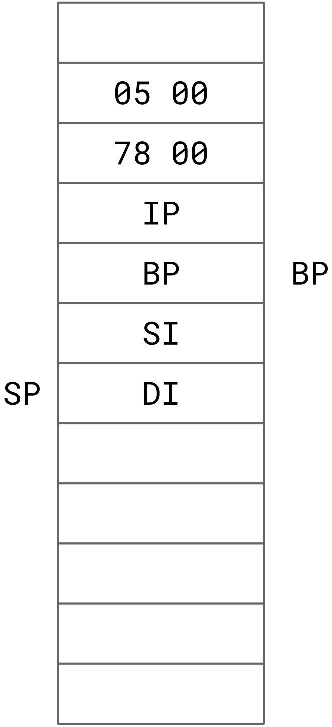


New location of BP.

Old value of BP.

```
_asmprint:
   push  bp        ;save bp
   mov   bp,sp         ;setup bp
   push  si        ;save si
   push  di        ;save di
   mov   dx,[bp+4]     ;0078  "x"
   mov   cx,[bp+6]     ;0005  count
x: mov   ah,2      ;dos code
   int   21h       ;write
   loop  x         ;repeat
   pop   di        ;restore di
   pop   si        ;restore si
   pop   bp        ;restore bp
   mov   ax,0      ;set rc
   ret             ;return
```

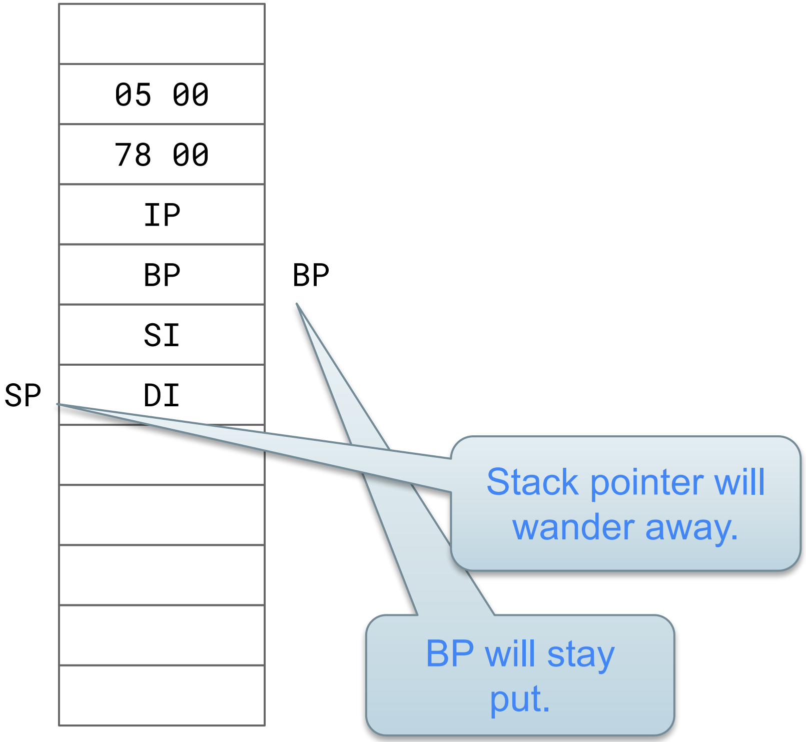
How to access parameters on the stack



	05 00	
	78 00	
	IP	
	BP	BP
	SI	
SP	DI	

```
_asmprint:
   push  bp       ;save bp
   mov   bp,sp        ;setup bp
   push  si       ;save si
   push  di       ;save di
   mov   dx,[bp+4]    ;0078  "x"
   mov   cx,[bp+6]    ;0005  count
x: mov   ah,2     ;dos code
   int   21h      ;write
   loop  x        ;repeat
   pop   di       ;restore di
   pop   si       ;restore si
   pop   bp       ;restore bp
   mov   ax,0     ;set rc
   ret            ;return
```

How to access parameters on the stack

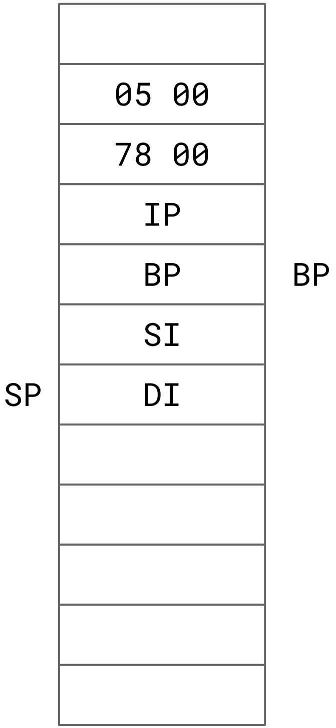


Stack pointer will wander away.

BP will stay put.

```
_asmprint:
   push  bp       ;save bp
   mov   bp,sp        ;setup bp
   push  si       ;save si
   push  di       ;save di
   mov   dx,[bp+4]     ;0078  "x"
   mov   cx,[bp+6]     ;0005  count
x: mov   ah,2     ;dos code
   int   21h      ;write
   loop  x        ;repeat
   pop   di       ;restore di
   pop   si       ;restore si
   pop   bp       ;restore bp
   mov   ax,0     ;set rc
   ret            ;return
```

How to access parameters on the stack

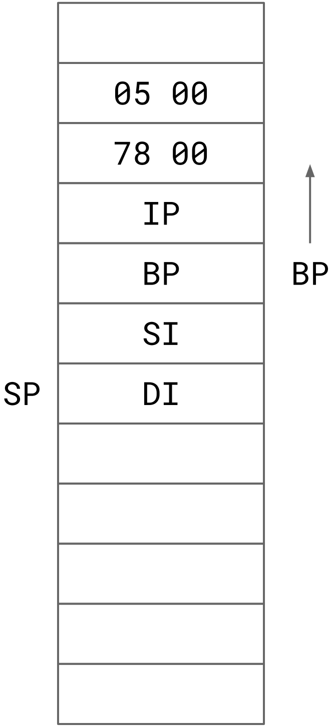


05 00
78 00
IP
BP
SI
DI

Execute body of subroutine

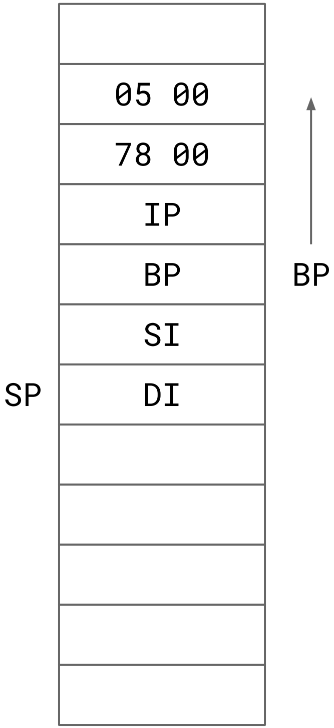
```
_asmprint:
   push  bp       ;save bp
   mov   bp,sp        ;setup bp
   push  si       ;save si
   push  di       ;save di
   mov   dx,[bp+4]     ;0078  "x"
   mov   cx,[bp+6]     ;0005  count
x: mov   ah,2     ;dos code
   int   21h      ;write
   loop  x        ;repeat
   pop   di       ;restore di
   pop   si       ;restore si
   pop   bp       ;restore bp
   mov   ax,0     ;set rc
   ret            ;return
```

How to access parameters on the stack



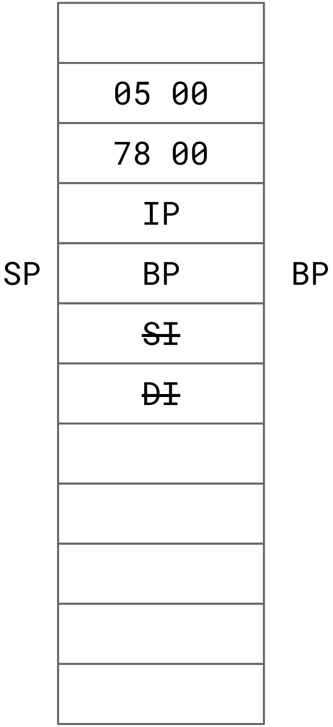
```
_asmprint:
   push  bp       ;save bp
   mov   bp,sp         ;setup bp
   push  si       ;save si
   push  di       ;save di
   mov   dx,[bp+4]     ;0078  "x"
   mov   cx,[bp+6]     ;0005  count
x: mov   ah,2     ;dos code
   int   21h      ;write
   loop  x        ;repeat
   pop   di       ;restore di
   pop   si       ;restore si
   pop   bp       ;restore bp
   mov   ax,0     ;set rc
   ret            ;return
```

How to access parameters on the stack



```
_asmprint:
   push  bp        ;save bp
   mov   bp,sp         ;setup bp
   push  si        ;save si
   push  di        ;save di
   mov   dx,[bp+4]     ;0078  "x"
   mov   cx,[bp+6]     ;0005  count
x: mov   ah,2      ;dos code
   int   21h       ;write
   loop  x         ;repeat
   pop   di        ;restore di
   pop   si        ;restore si
   pop   bp        ;restore bp
   mov   ax,0      ;set rc
   ret             ;return
```

How to access parameters on the stack



05 00

78 00

IP

SP BP BP

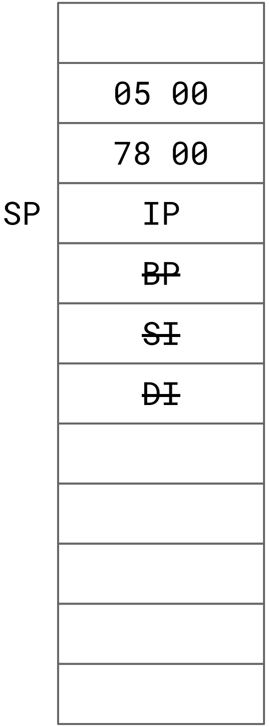
~~SI~~

~~DI~~

Restore caller environment

```
_asmprint:
    push  bp          ;save bp
    mov   bp,sp           ;setup bp
    push  si          ;save si
    push  di          ;save di
    mov   dx,[bp+4]       ;0078  "x"
    mov   cx,[bp+6]       ;0005  count
x:  mov   ah,2        ;dos code
    int   21h         ;write
    loop  x           ;repeat
    pop   di          ;restore di
    pop   si          ;restore si
    pop   bp          ;restore bp
    mov   ax,0        ;set rc
    ret               ;return
```

How to access parameters on the stack



05 00

78 00

SP IP

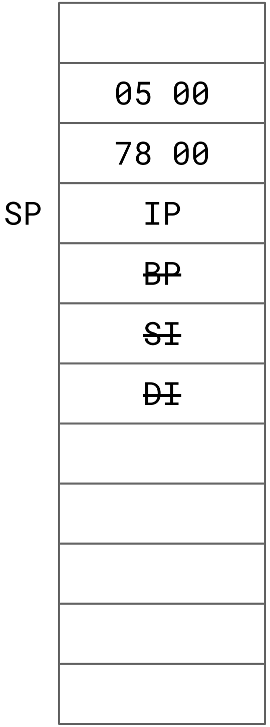
~~BP~~

~~SI~~

~~DI~~

```
_asmprint:
   push  bp        ;save bp
   mov   bp,sp         ;setup bp
   push  si        ;save si
   push  di        ;save di
   mov   dx,[bp+4]      ;0078  "x"
   mov   cx,[bp+6]      ;0005  count
x: mov   ah,2      ;dos code
   int   21h       ;write
   loop  x         ;repeat
   pop   di        ;restore di
   pop   si        ;restore si
   pop   bp        ;restore bp
   mov   ax,0      ;set rc
   ret             ;return
```

How to access parameters on the stack



05 00

78 00

SP IP

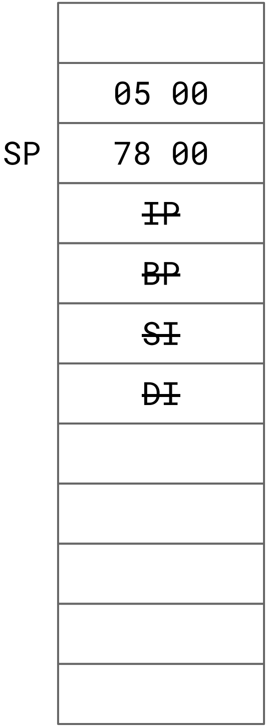
~~BP~~

~~SI~~

~~DI~~

```
_asmprint:
   push  bp         ;save bp
   mov   bp,sp          ;setup bp
   push  si         ;save si
   push  di         ;save di
   mov   dx,[bp+4]      ;0078  "x"
   mov   cx,[bp+6]      ;0005  count
x: mov   ah,2       ;dos code
   int   21h        ;write
   loop  x          ;repeat
   pop   di         ;restore di
   pop   si         ;restore si
   pop   bp         ;restore bp
   mov   ax,0       ;set rc
   ret              ;return
```

How to access parameters on the stack



05 00

SP 78 00

~~IP~~

~~BP~~

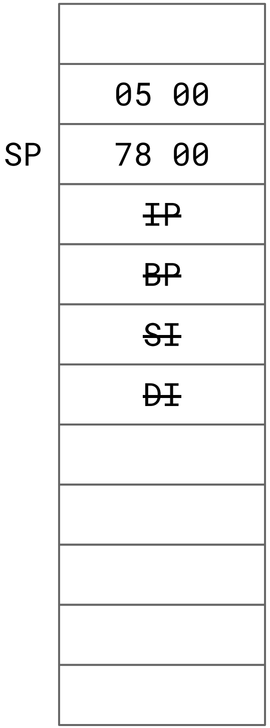
~~SI~~

~~DI~~

```
_asmprint:
   push  bp          ;save bp
   mov   bp,sp           ;setup bp
   push  si          ;save si
   push  di          ;save di
   mov   dx,[bp+4]       ;0078  "x"
   mov   cx,[bp+6]       ;0005  count
x: mov   ah,2        ;dos code
   int   21h         ;write
   loop  x           ;repeat
   pop   di          ;restore di
   pop   si          ;restore si
   pop   bp          ;restore bp
   mov   ax,0        ;set rc
   ret               ;return
```

- **Pops IP off stack**
- **That's a jump**

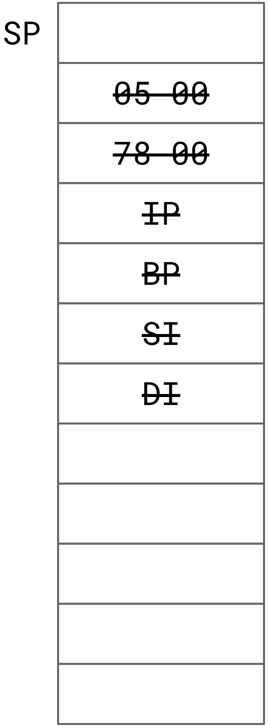
“Cleaning Up” the stack



Back in caller

```
mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

“Cleaning Up” the stack



SP

~~05 00~~

~~78 00~~

~~IP~~

~~BP~~

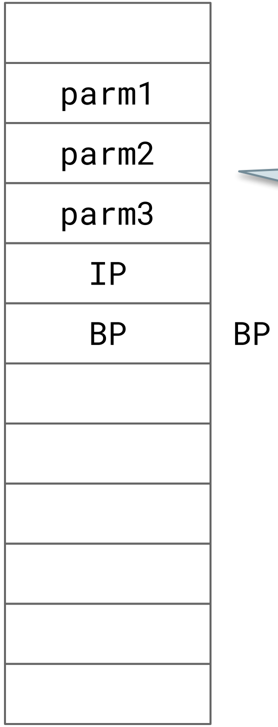
~~SI~~

~~DI~~

This is why uninitialized local variables contain garbage in C!

```
mov  ax, 5
push ax
mov  al, 120
push ax
call _asmprint
add  sp,4
mov  [rc],ax
```

Why push last-to-first



parm1

parm2

parm3

IP

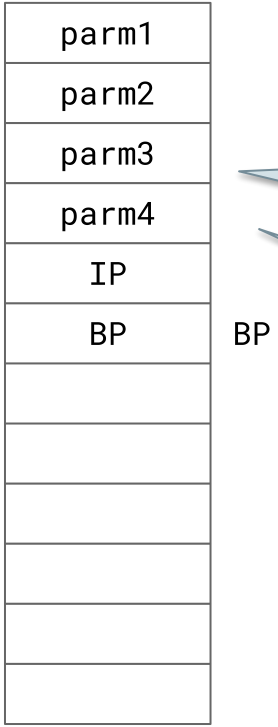
BP

BP

Let’s try first-to-last instead.

```
sub(parm1, parm2, parm3)

     bp+8   bp+6   bp+4
```

Why push last to first



Oops. Adding a parameter moves all the previous parameters (relative to BP)

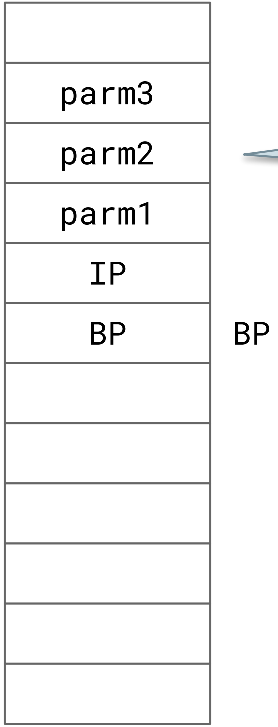
This could require a lot of changes to an assembly-language subroutine.

```
sub(parm1, parm2, parm3)

      bp+8   bp+6   bp+4
```

Add a parameter:

```
sub(parm1, parm2, parm3, parm4)

      bp+10  bp+8   bp+6   bp+4
```

Why push last to first

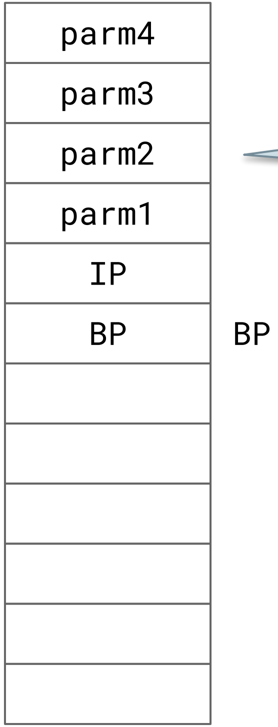


OK. Let’s try last-to-first.

```
sub(parm1, parm2, parm3)
```

bp+4   bp+6   bp+8

Why push last to first



parm4

parm3

parm2

parm1

IP

BP

BP

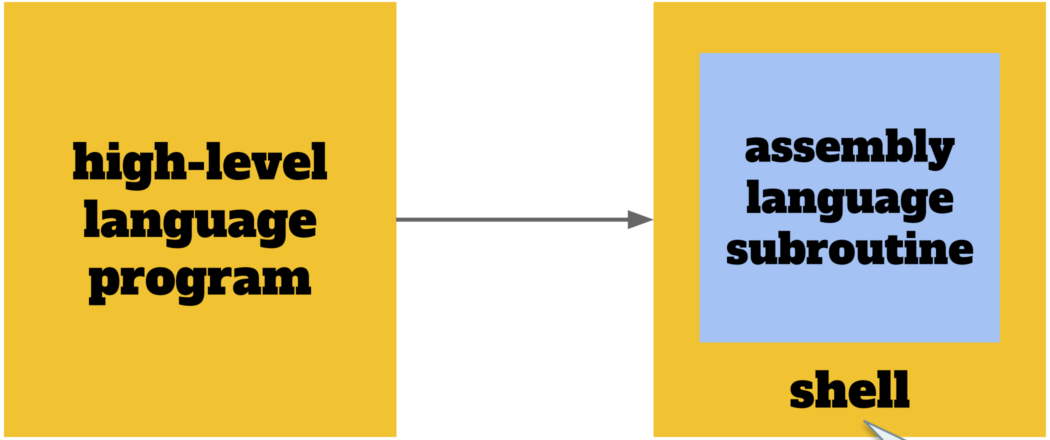
Good. The Offset from BP doesn't change.

```
sub(parm1, parm2, parm3)
      bp+4   bp+6   bp+8
```

Add a parameter:

```
sub(parm1, parm2, parm3, parm4)
      bp+4   bp+6   bp+8   bp+10
```

Simplifying the Assembly Language Subroutine



high-level language program

assembly language subroutine

shell

We **don’t** mean the command-line interpreter here.

Shell *implements* the HLL protocol

Code Characteristics

- Non-reusable
  - may only be used once
  - must be loaded every time it is run
- Serially reusable
  - may be used again after it completes
- Reentrant
  - may be used simultaneously by multiple threads / processor cores.
  - may be called again, before it finishes

How do compilers create reentrant code?

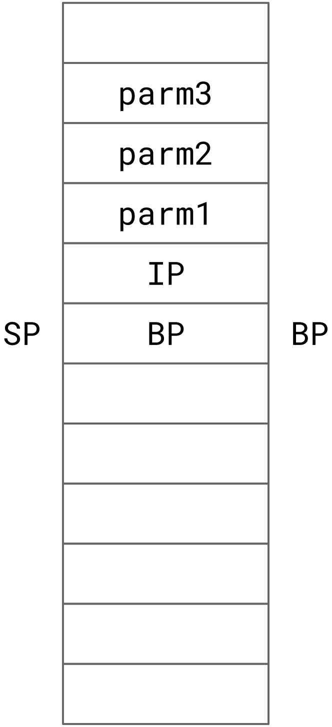
Compilers

- Previously
  - We created data in data segment
  - Only support non-reentrant code
- Compiled HLL code generally uses the stack
  - Can still use the data segment … if you ask for it.
  - `static int count = 7;`
- Data segment
  - One data segment per program
  - (Can have multiple “.data” directives)
  - Data segment variables initialized (and create) once — not re-usable (without some extra work)

Local variables

- Local variable
    - Local to the subroutine
- Created
    - On stack
    - Dynamically
    - Lifetime is same as subroutine
- Access
    - Indirect — by location on the stack, not name
        - Remember the –g flag for gcc.
    - Via the BP (base pointer)
    - BP pointer the *activation record* (aka the *stack frame*)

Local variables



parm3

parm2

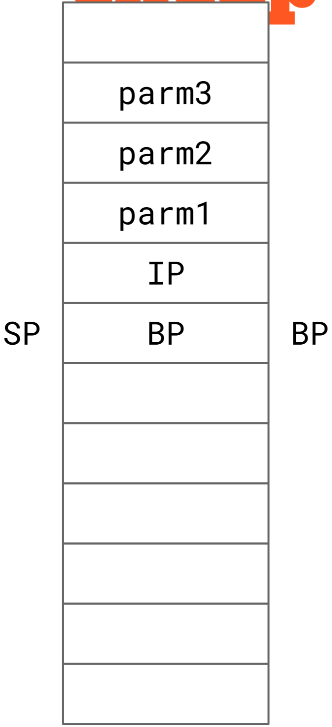
parm1

IP

SP BP BP

```
sub(parm1, parm2, parm3)
{
  int x, y, z;

  …

}
```

Local variables are computationally cheap



parm3

parm2

parm1

IP

SP BP BP

```
sub(parm1, parm2, parm3)
{
  int x, y, z;

  …

}
```

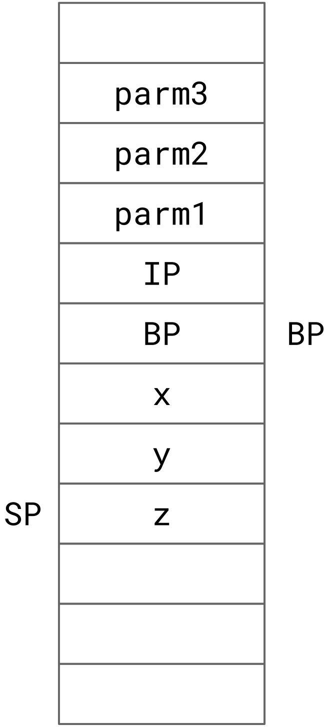
Compiler allocates space in stack for locals

```
SP = SP - 6
```

Cheap to allocate as many local variables as you need.

But, this doesn't initialize them.

Local variables

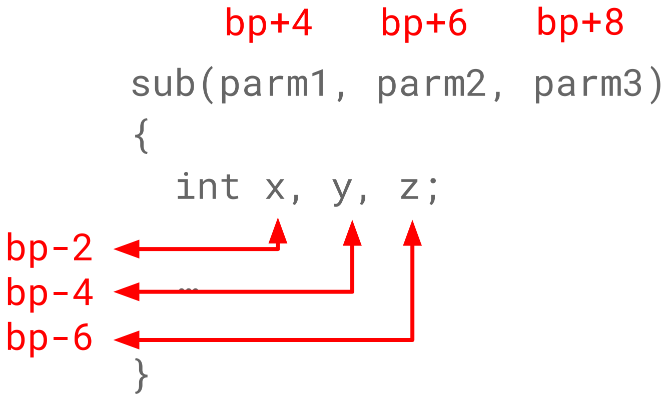


parm3
parm2
parm1
IP
BP
x
y
z

BP

SP

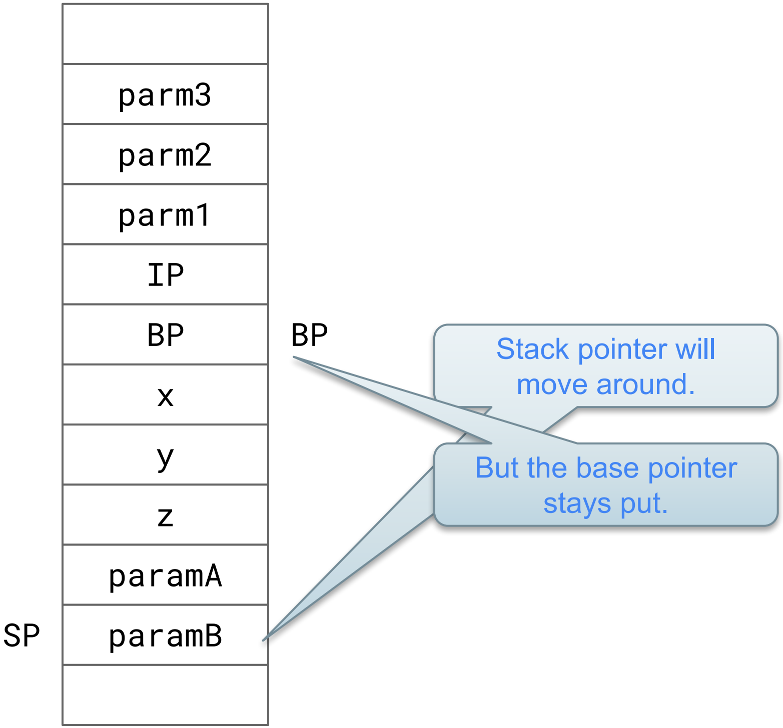
Glad I have this rock-solid base pointer here, staying in the same place.



```
      bp+4    bp+6    bp+8
sub(parm1, parm2, parm3)
{
  int x, y, z;
}
```

bp-2
bp-4
bp-6

Local variables



Stack pointer will move around.

But the base pointer stays put.

```
sub(parm1, parm2, parm3)
{
  int x, y, z;

  anotherSub( paramX, paramY );

}
```