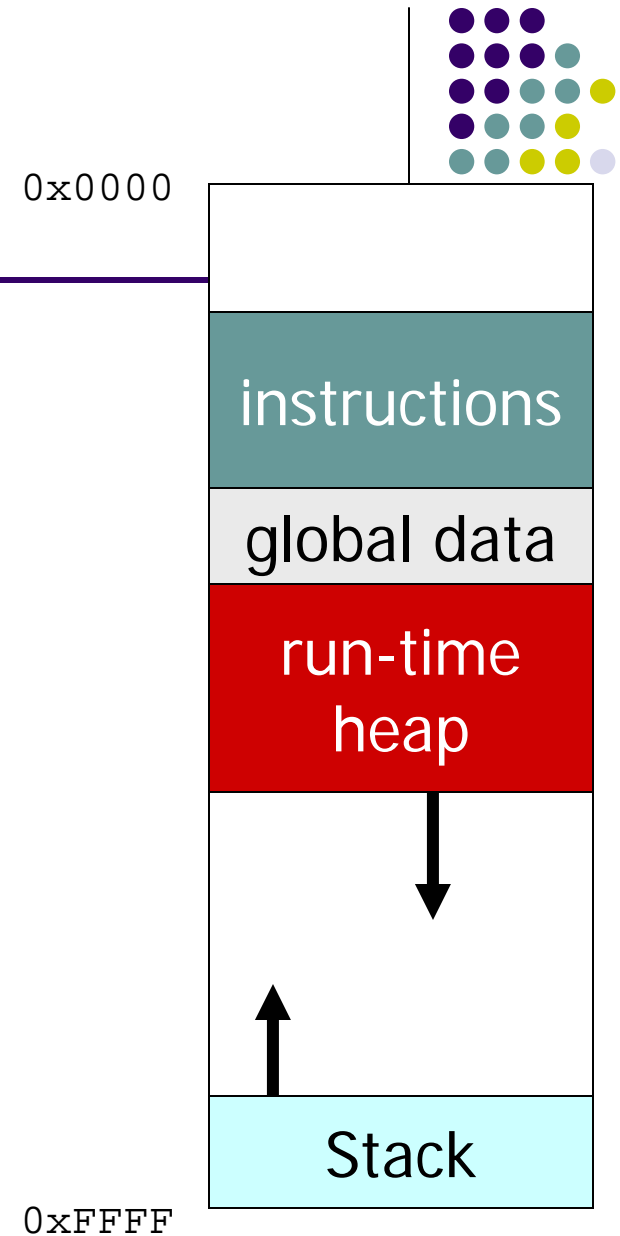


Machine-Level Programming III: Control Flow, Stack frame

- Topic
 - Stack frames

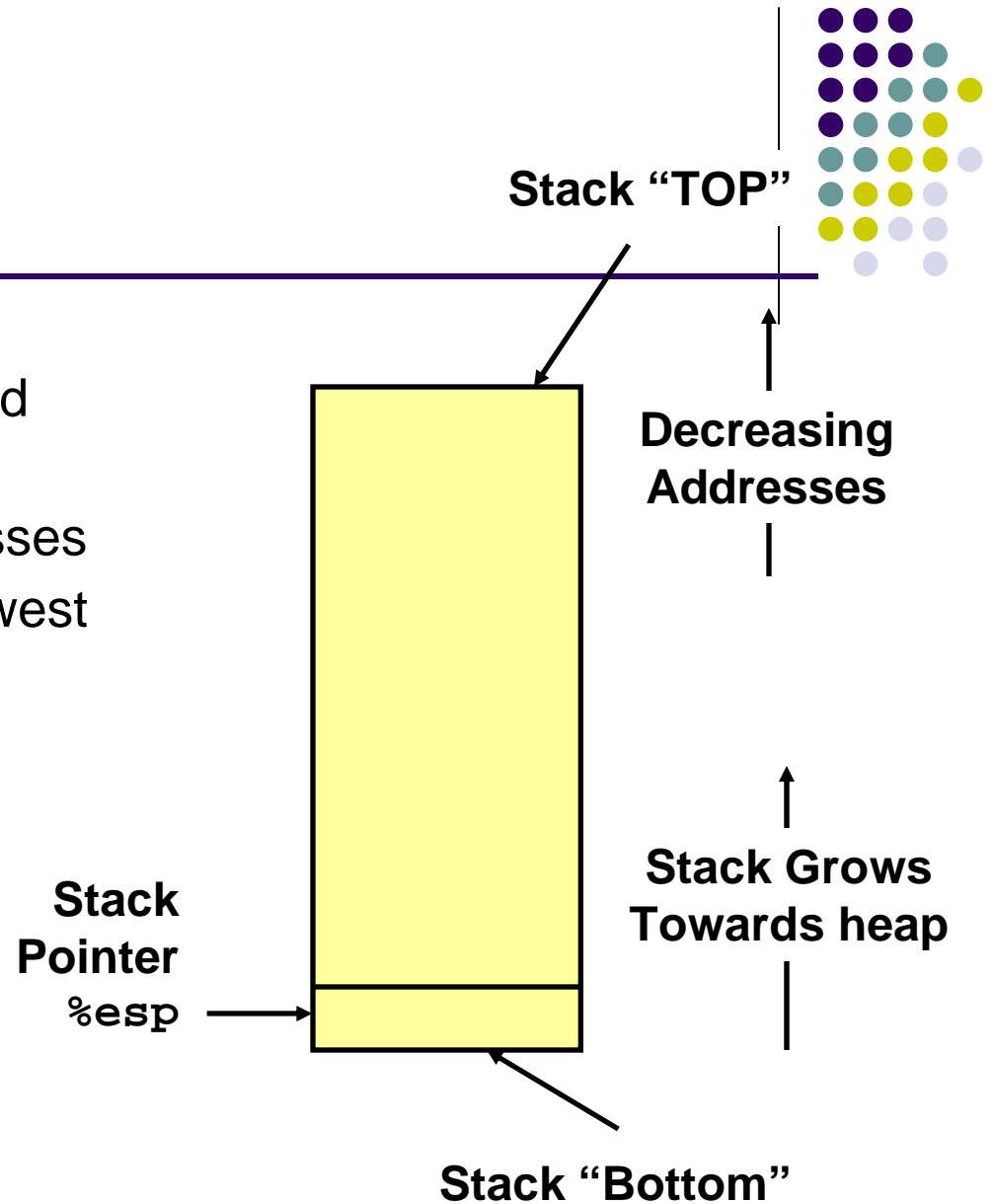
Programs & Memory

- The Von Neumann:
 - programs & data are stored in memory
- Recall assembly code converted to object code (**Assembler**)
- Labels are nothing more than address locations of instructions
- Here: `mov %eax, %edx`
- ...
- `jmp Here`



IA32 Stack

- Region of memory managed with stack discipline
- Grows toward lower addresses
- Register `%esp` indicates lowest stack address
 - address of top element
- Push element on to stack
- Pop the top most element



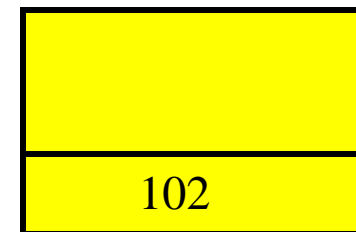
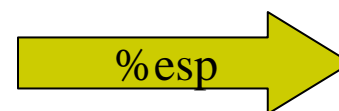


Use of stack

- For storing return address of calling procedure
- When a procedure is called
- Program needs to jump to the starting address of the procedure
- After execution of the procedure, return to the next statement following the call statement
- CALL .. Push 0102
- RET ... POP %eip

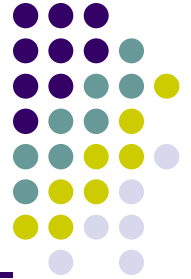
```
... main(){  
....  
100: CALL myfunc()  
102  
....  
}  
  
500: void myfunc()  
{  
...  
}
```

```
...  
....  
100: push 102  
      jmp 500  
102  
....  
}  
  
500:  
{  
... RET  
}
```

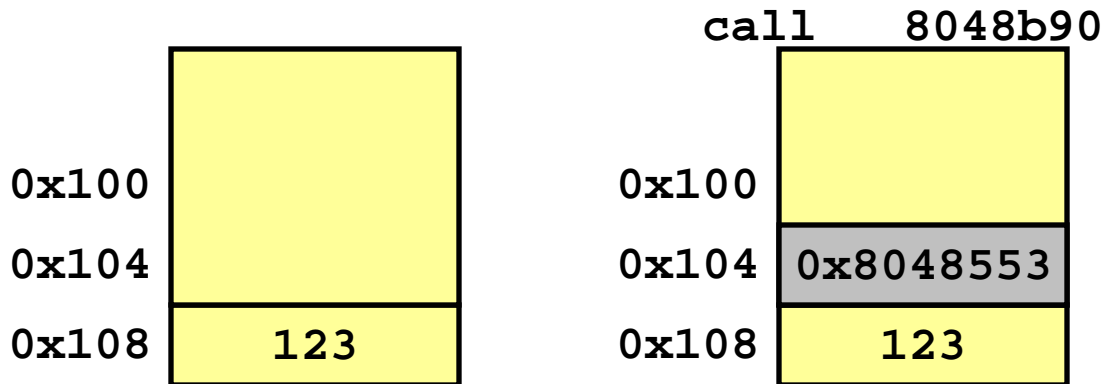


996

Procedure Call Example



```
804854e: e8 3d 06 00 00    call 8048b90 <main>
8048553: 03                add ...
```



%esp 0x108

%esp 0x104

%eip 0x804854e

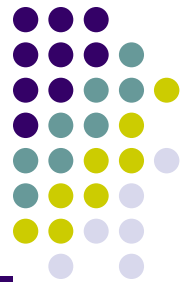
%eip 0x8048b90

%eip is program counter

Procedure Control Flow

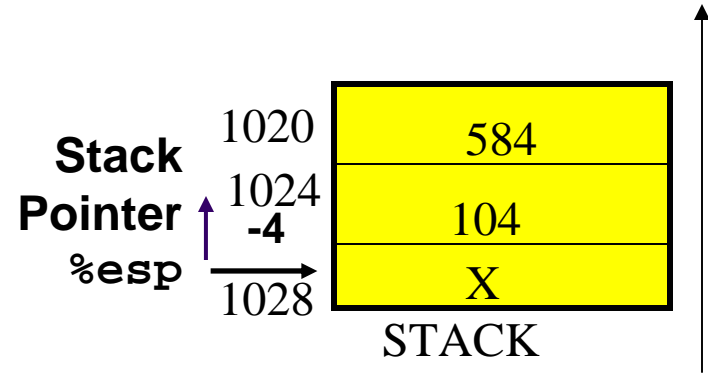


- Use stack to support procedure call and return
- Procedure call:
`call label` Push return address on stack; Jump to `label`
- Return address value
 - Address of instruction beyond `call`
 - Example from disassembly
804854e: e8 3d 06 00 00 call 8048b90 <main>
8048553: 08 add ...
 - Return address = 0x8048553
- Procedure return:
 - `ret` Pop address from stack; Jump to address

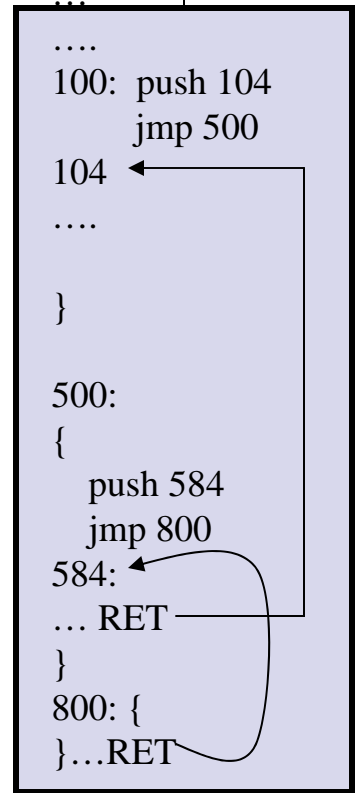


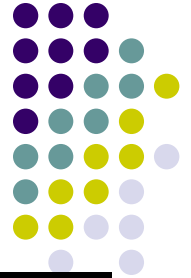
Nested procedures

- On each call, push the return address on stack
- On RET, pop the address to return from the innermost procedure call
- RET → pop %eip



```
... main(){  
...  
100: myfunc()  
102  
...  
}  
500: void myfunc()  
{  
580: otherfunc()  
}  
800: void otherfunc()
```



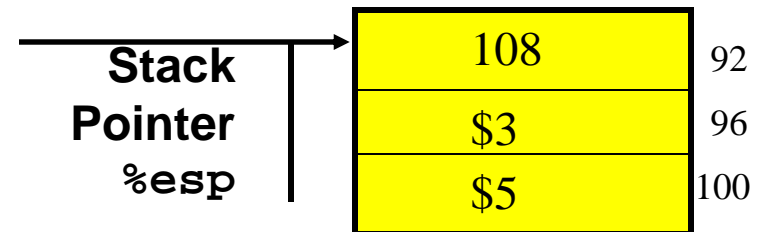


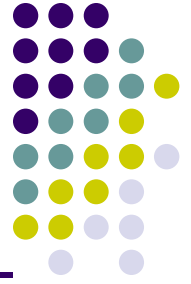
Passing parameters

- What if the procedure has parameters?
- Use stack to store the values
- First,
 - Push parameters onto stack
- Where are the parameters?
- $\%esp + 4$, $\%esp + 8$
- $\%esp$ points to top of stack
- Second, Push return address
- Top of stack still contains the return address

```
... main(){  
....  
100: myfunc(3,5)  
102  
....  
}  
  
500: void myfunc(int x, int y)  
{  
...  
}
```

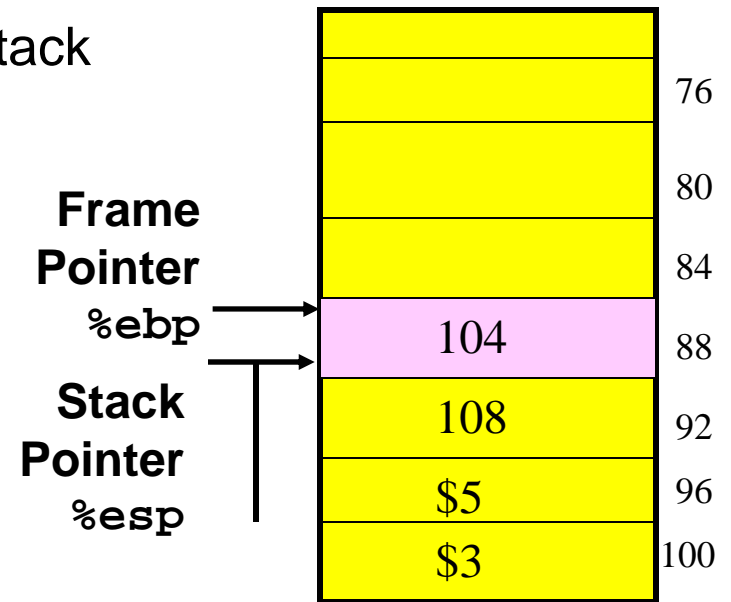
```
...  
....  
100: push $3  
      push $5  
104  jmp 500  
108  
....  
}  
  
500:  
{  
... RET  
}
```





Stack frames

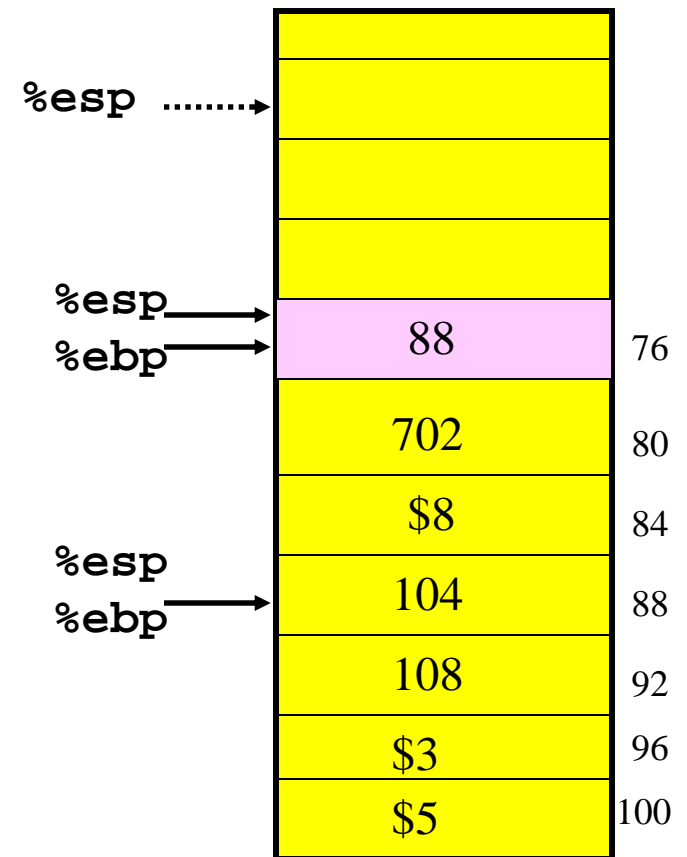
- To preserve the offset for parameters for every called procedure
 - maintain another register called frame pointer
- `%ebp` is the frame pointer
- Every time a procedure is entered
 - push `%ebp` old frame pointer on to stack
 - `mov %esp, %ebp`
 - `%ebp` points to current SP
- When executing within `myfunc`
- `%ebp?` `%esp?`
- `%ebp+8?`, `%ebp+12?`



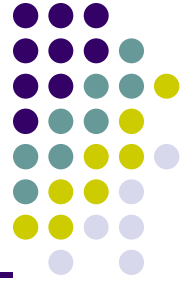


Stack frames

- When myfunc calls otherfunc()
- Push parameters and return address on stack
- Before entering otherfunc()
 - push %ebp old frame pointer
 - mov %esp, %ebp
 - %ebp points to current SP
- When executing within otherfunc()
 - %ebp? %esp?
 - %ebp+8?
- On RET (clean up)
 - mov %ebp, %esp
 - pop %ebp



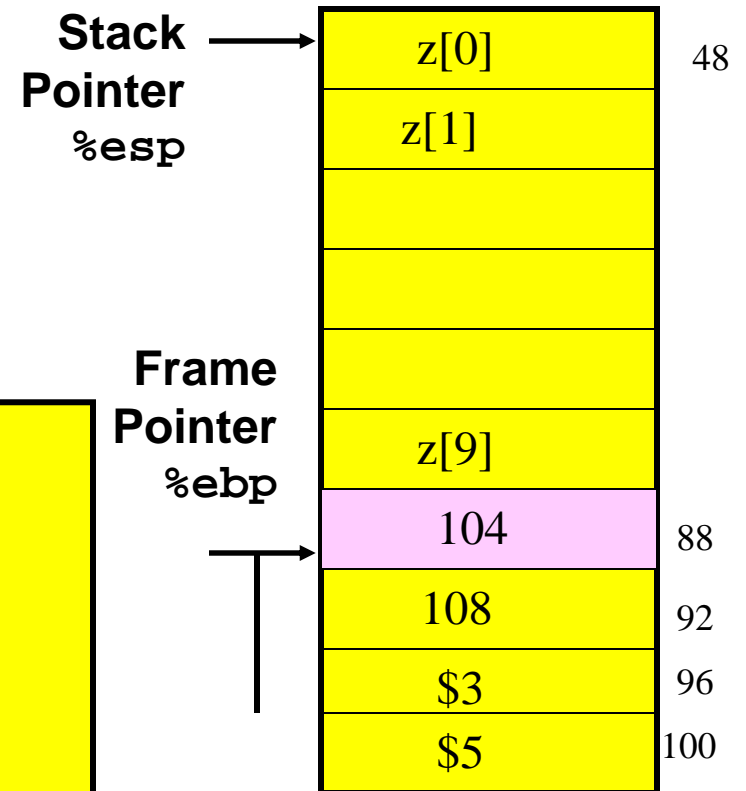
What about local variables within procedures?

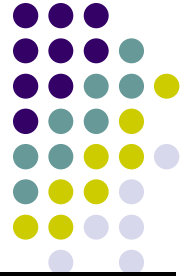


- Also goes on stack!
- The called procedure pushes local variables onto stack
- Where are the parameters?
- $\%ebp + 8$, $\%ebp + 12$
- Where are the local variables?
- $\%ebp - 4$, $\%ebp - 8$, ...
- z is a local variable in `myfunc()`

```
... main(){
....
100: myfunc(3,5)
102
....
}

500: void myfunc(int x, int y)
{int z[9];
z[0]=x+y;
...
}
```





Stack is used for local variables

- z[0] is stored in -40(%ebp)
- On RET from myfunc
- Clean up or LEAVE
- On RET (clean up)
 - mov %ebp, %esp
 - pop %ebp

```
...   main(){
....
100:  myfunc(3,5)
102
....
}
```

```
500: void myfunc(int x, int y)
{int z[9];
z[0]=x+y;
...
}
```

```
...
....
100:  push $3
      push $5
104:  jmp 500
108
....
}

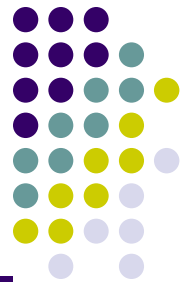
500:
{ mov 8(%ebp), %eax
  add 12(%ebp), %eax
  mov %eax, -40(%ebp)
... RET
}
```




Cleaning the stack of arguments

- Cleaning done by the called procedure
 - Use RET n instructions
 - RET n → pop %eip, and add %esp, \$n
 - Called procedure needs to know the size of the argument
- Cleaning done by the calling procedure
- After returning from the procedure
- addl sizeofargument, %esp

x86 Stack Frame



- Current Stack Frame (“Top” to Bottom)
 - Parameters for function about to call
 - “Argument build”
 - Local variables
 - If can’t keep in registers
 - Saved register context
 - Old frame pointer
- Caller Stack Frame
 - Return address
 - Pushed by `call` instruction
 - Arguments for this call

