Intro to Computers

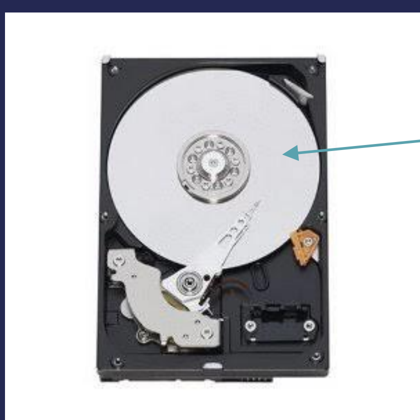
UT LAW 379M

Fall 2023

Lecture Notes

Three Major Computer Parts

Central Processing Unit (CPU)



Hard Drive

Random Access Memory (RAM)

First, a Word About Data

- Computers store and process all data as ***BINARY NUMBERS***
- Everything is a binary number. Games, programs, music, etc
- The binary numbers are ***interpreted*** as letters, music, etc

Quick Binary Lesson

Base 10 Numbers

$10^3 = 1000$	$10^2 = 100$	$10^1 = 10$	$10^0 = 1$
3	0	9	5
(3X1000) +	(0X100) +	(9X10) +	(5X1)
3000 +	0 +	90 +	5
			3095

Base 2 Numbers

$2^3 = 8$	$2^2 = 4$	$2^1 = 2$	$2^0 = 1$
1	0	1	0
(1X8) +	(0X4) +	(1X2) +	(0X1)
8 +	0 +	2 +	0
			10

What are These Binary Numbers?

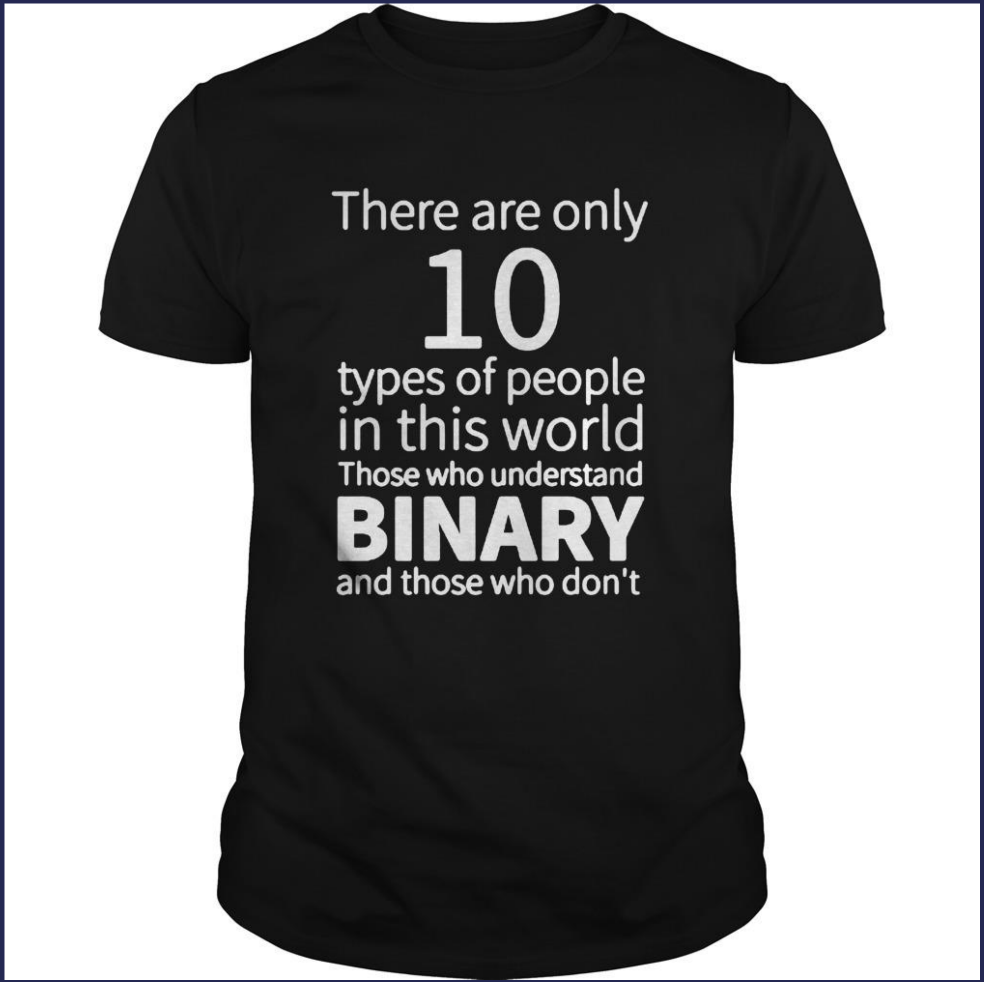
- Binary 1000?
- Binary 0100?
- Binary 0010?
- Binary 0001?
- Binary 0101?
- Binary 1010?
- Binary 1111?

Remember 8, 4, 2,1 columns.
If there's a one, add the column:

0110 = 4+2 = 6

1001 = 8+1 = 9

Decode This Shirt



Binary Sizes

```
09/09/2020  10:50 PM    <DIR>          Links
09/09/2020  10:50 PM    <DIR>          Music
10/04/2020  09:41 AM           397,307 npm-debug.log
09/11/2020  11:28 AM    <DIR>          OneDrive
06/29/2020  01:51 PM                26 osync.log
10/05/2020  11:51 AM    <DIR>          Pictures
08/26/2019  11:10 PM                13 query
09/09/2020  10:50 PM    <DIR>          Saved Games
09/09/2020  10:50 PM    <DIR>          Searches
03/11/2020  08:05 AM         3,298,911 snap_beans1.tgz
08/26/2019  11:10 PM                13 start
05/19/2019  09:57 PM                 0 Sti_Trace.log
08/26/2019  11:10 PM                13 stop
10/06/2020  08:53 AM    <DIR>          Videos
02/10/2020  07:48 PM    <DIR>          VirtualBox VMs
               8 File(s)     61,789,284 bytes
              22 Dir(s)  12,737,953,792 bytes free

C:\Users\seth_>
```

- A one or a zero is a "bit"
- 8 bits is a byte
- Let's use our command line to see how big files are:
  - Open a terminal on Mac or the cmd shell on Windows
  - Type "ls –l" on Mac or "dir" on Windows

```
sethjn@Paladin:~$ ls -l
total 69160
-rwxrwxrwx 1 sethjn sethjn   737282 Jul 27 13:01 chaum_001.pdf
lrwxrwxrwx 1 sethjn sethjn       30 Aug 26  2019 dev -> /mnt/d/seth_/P
v/
-rw-rw-rw- 1 sethjn sethjn  2488927 Mar 30  2020 files.txt
-rwxrwxrwx 1 sethjn sethjn 32655988 Jul 27 09:58 iso11770_1.pdf
-rwxrwxrwx 1 sethjn sethjn   779655 Jul 27 09:59 iso11770_3.pdf
-rwxrwxrwx 1 sethjn sethjn   186903 Jul 27 22:20 places_caswell.pdf
```

Binary is Often Represented as Hex

- Binary is Base 2
- For "short hand", binary is often written in Base 16
- This is because there is a 1:1 mapping between 4 bits and 1 Hex number:

0000 – 0	0100 – 4	1000 – 8	1100 – C (hex 12)
0001 – 1	0101 – 5	1001 – 9	1101 – D (hex 13)
0010 – 2	0110 – 6	1010 – A (hex 10)	1110 – E (hex 14)
0011 – 3	0111 - 7	1011 – B (hex 11)	1111 – F (hex 15)

Interpretation - Letters

- ASCII – The original English language character mapping
- Each letter, symbol, and control character had a code
- ‘a’ – 97 (decimal), 0x61 (hex), 0110 0001 (bin)
- ‘A’ – 65 (decimal), 0x41 (hex), 0100 0001 (bin)
- Newline – 10 (decimal), 0x0a (hex), 0000 1010 (bin)

Interpretation - Images

- Uncompressed pictures can be 1 number per pixel
- Each number is an index into a table of colors
- 1 bit-per-pixel can do two colors (black and white)
- 8 bpp can do 256 colors
- 32 bpp can do 4,294,967,296 colors

(Side Note, Compression)

- Images can be compressed with special techniques
- Groups of same-color pixels stored together
- Videos go even further
  - I frame, full picture
  - B frame/p frame, changes from previous picture

Interpretation - Documents

- Multi-level encoding
- Codes indicate whether text is regular, bold, etc
- All the data is ***rendered*** by the display program

HTML Example

- The entire document, except for media, is text
  - So, there is “interpretation” from binary to text
  - And more “interpretation” from text to codes
- Example:
  - <B>This is bold</B>
  - The <B> says “all text until next </B>” is bold

Processors and Binary

Computer "Instruction" is just a binary number. In older computers, like the 486 shown here, it was 32-bit. In more modern processors it is 64-bit.

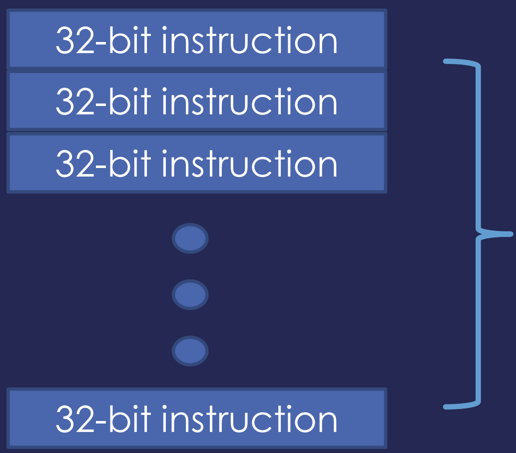
1	0	0	1	1	0	0	0	0	1	1	0	0	1	0	0	0	0	1	1	1	1	1	0	1	0	1	0	0	1	1	0

CPU Instructions

The instruction is broken down into pieces. One part is called the "op code" or operation code, it tells the computer what to do. The other parts are parameters. Each instruction "add", "subtract," etc. has its own op code.

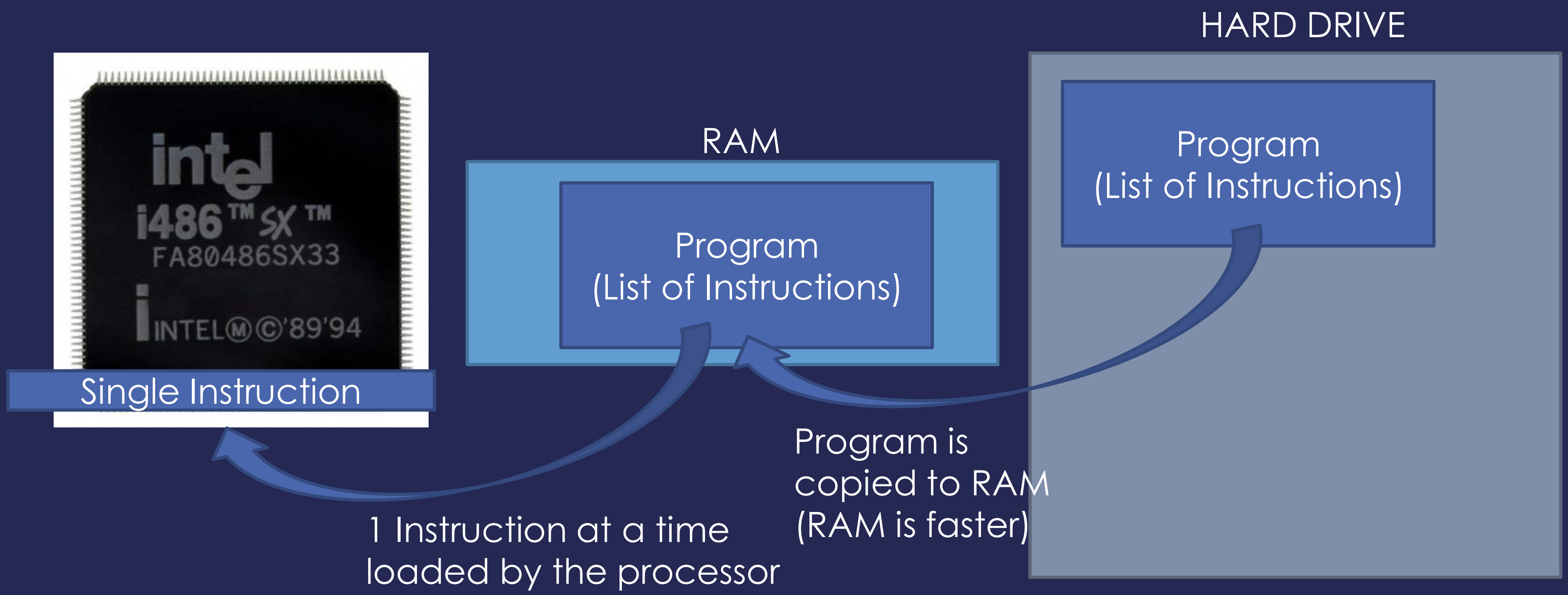
$$\underbrace{1\,0\,0\,1\,1\,0}_{\text{OP CODE}}\;\underbrace{0\,0\,0\,1\,1\,0\,0\,1\,0\,0\,0\,0}_{\text{PARAMETER 1}}\;\underbrace{1\,1\,1\,1\,1\,0\,1\,0\,1\,0\,0\,1\,1\,0}_{\text{PARAMETER 2}}$$

Program Stored On Disk

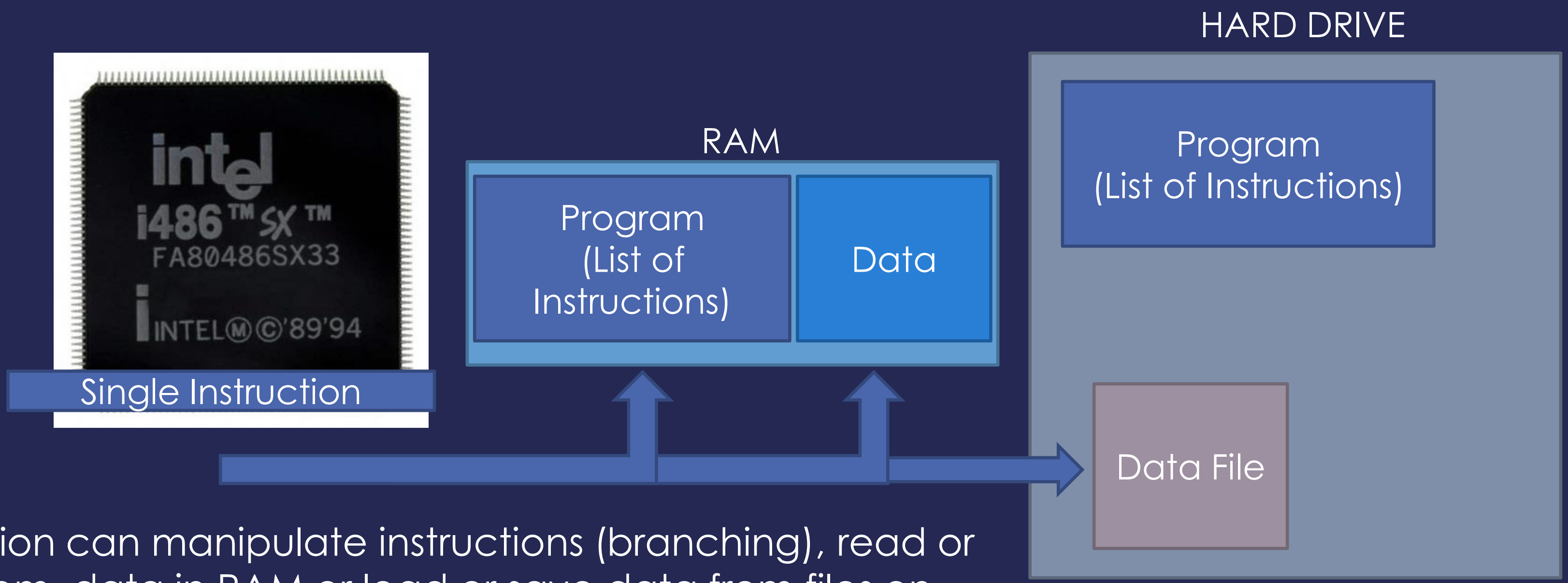


- Program is a list of instructions
- Processor executes one at a time.
- Some instructions are “branches”
- A branch jumps ahead or behind in the list
- Often “conditional” (e.g., if x > 0 then jump)

Running a Program



Data Too



Instruction can manipulate instructions (branching), read or write from data in RAM or load or save data from files on disk. (All are just binary numbers)

Creating a Program

- Programs written in Programming Language
- Programming Language is human readable
- However, programming language ***NOT*** machine readable
- Remember, processor ***ONLY*** speaks in basic instructions

Compiling

- Some languages can be “complied”
- This means the program is converted to machine readable
- Compiler knows how to translate and organize
- Quite complicated and still subject of research
- Example: C/C++

C Example

```
#include <stdio>

int main(void) {
    printf("Hello World\n");
    return 0;
}
```

COMPILER

1100101001111010101
1010010101001110101
1010101111101110000
1010101010001111000

Interpreting

- Some languages can be “interpreted”
- This means the program is “executed” by interpreter
- Usually one instruction at a time
- Interpreter has to be a fully compiled program
- Interpretation is considerably slower
- Example: Python

Python Example

```
if __name__ == "__main__":
    print("Hello World")
```


```
print("Hello World")
```

PYTHON INTERPRETER
(Already Binary Program)

11001010011111010101

Virtual Machine

- An extreme form of "interpreting"
- Virtual processor with virtual instructions
- Programs are compiled to the virtual instructions
- Virtual instructions executed on the virtual processor
- Virtual processor translates instructions to host machine code
- Example: Java

Java Example


```
import System;

class HelloWorld {
  static int main(void) {
    System.Out.println("Hello World!");
    return 0;
  }
}
```

COMPILER

11001010011111010101
10100101010011110101
10101011111011110000
10101010100011111000

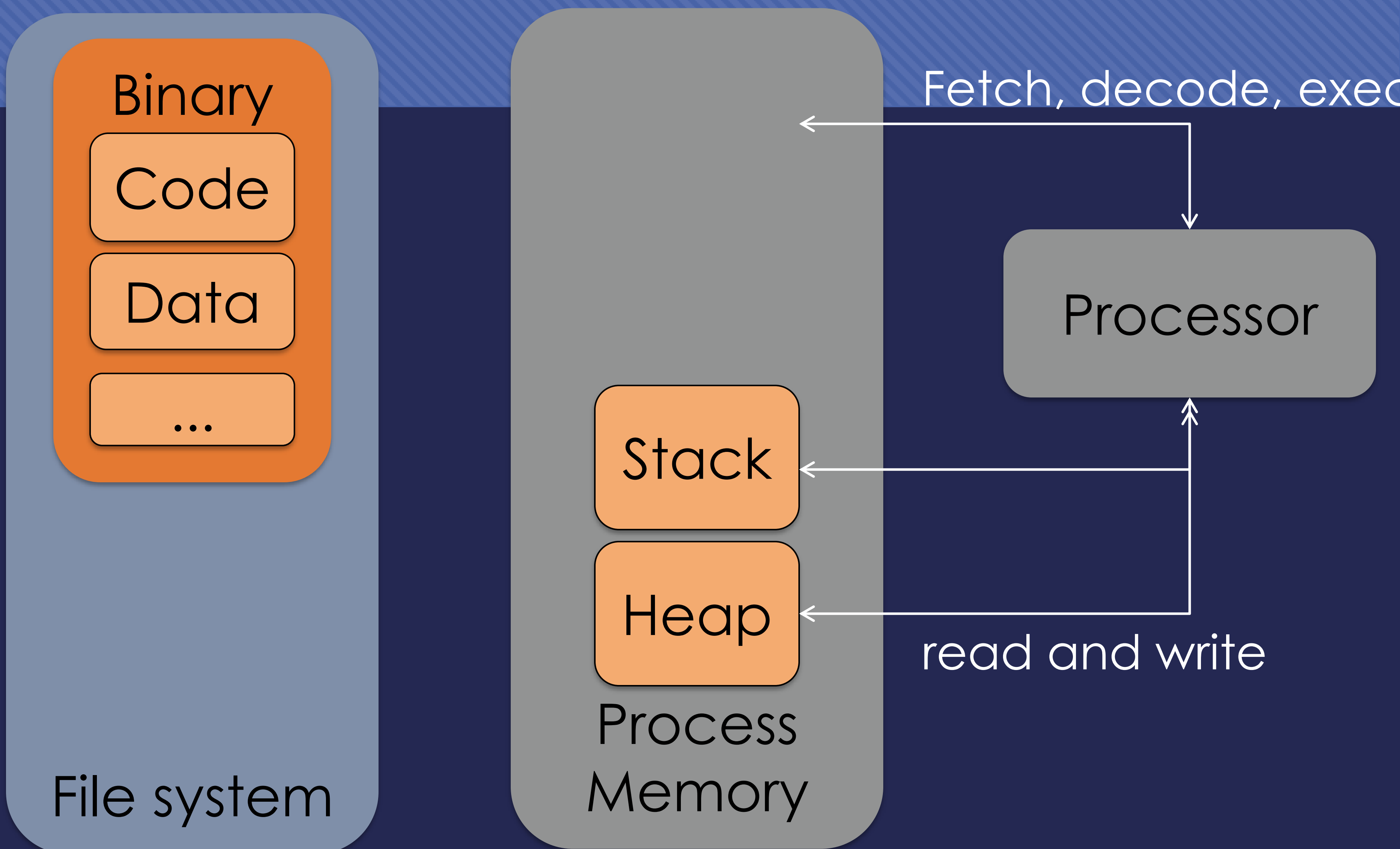
Java Virtual Machine

11001010011111010101

Why This is Useful

- Everything in a computer is a number. Programs, Data, etc
- This is great because programs and data can be stored the same way
- Everything can be copied over a network the same way

Basic Execution

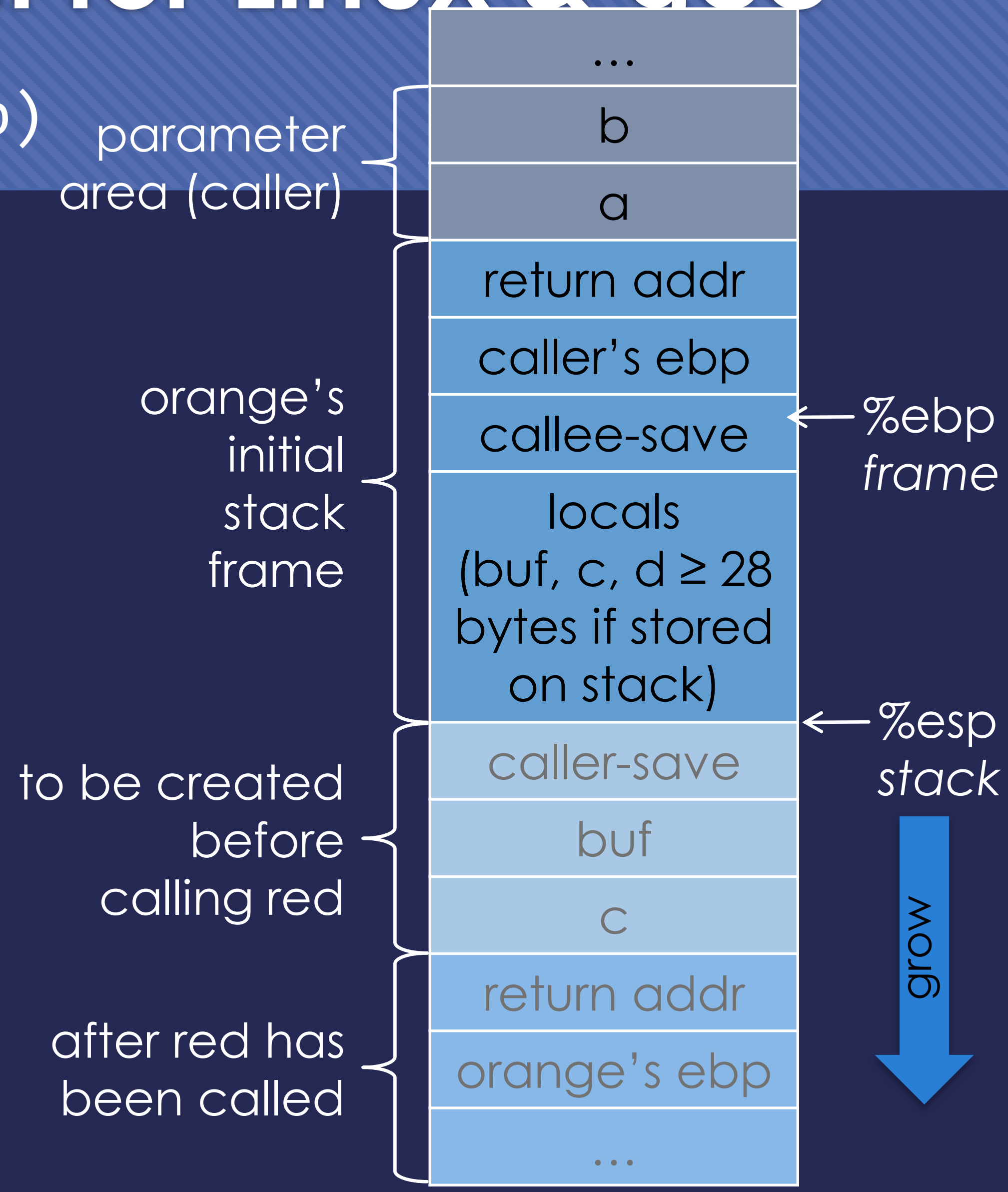


EBP and ESP

- EBP
  - Stack Base Pointer
  - Where the stack was when the routine started
- ESP
  - Stack Pointer
  - Top of the current stack
- EBP is a previous function's saved ESP

cdecl – default for Linux & gcc

```
int orange(int a, int b)
{
  char buf[16];
  int c, d;
  if(a > b)
     c = a;
  else
     c = b;
  d = red(c, buf);
  return d;
}
```



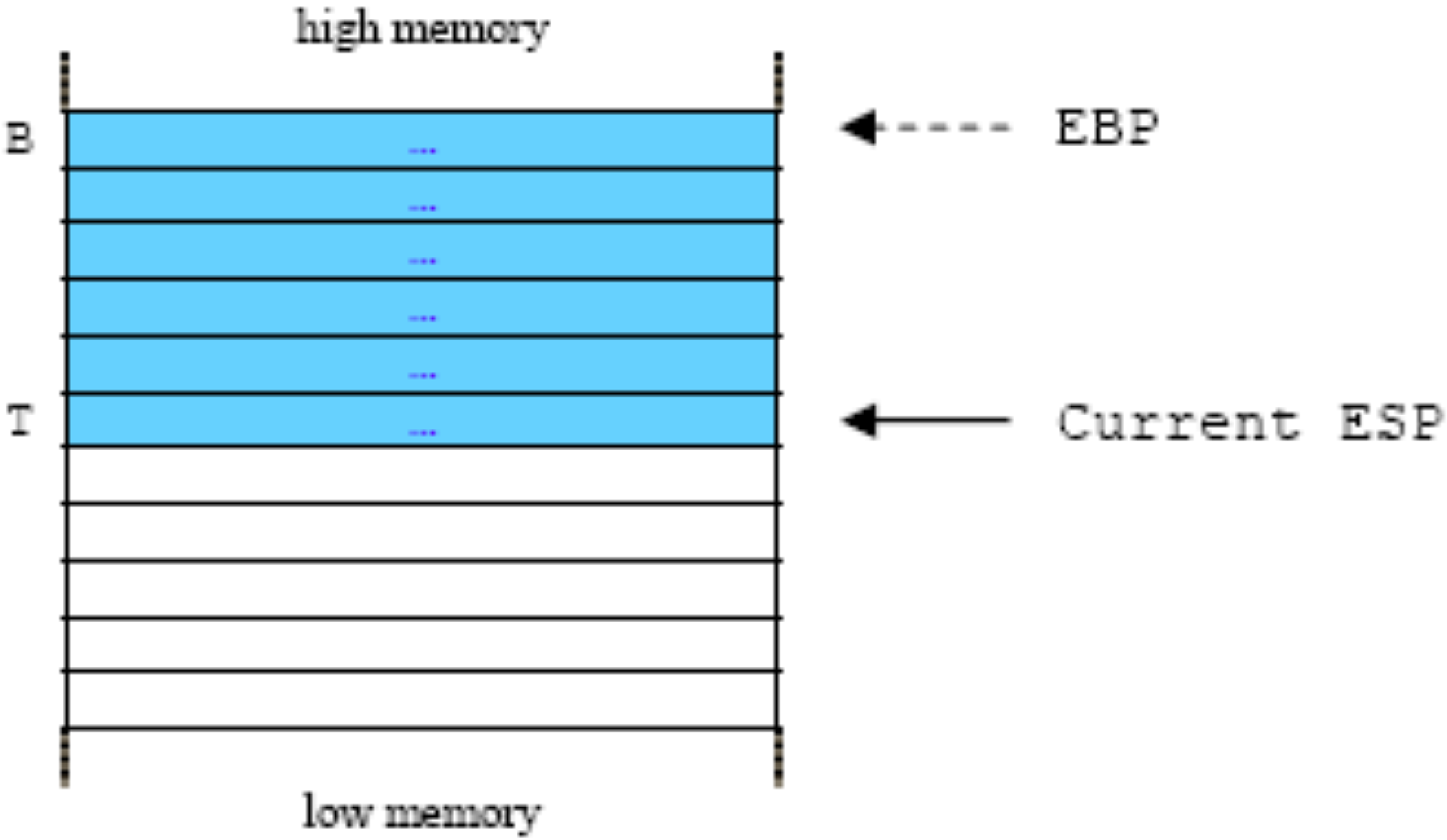
GDB Walkthrough – C Code

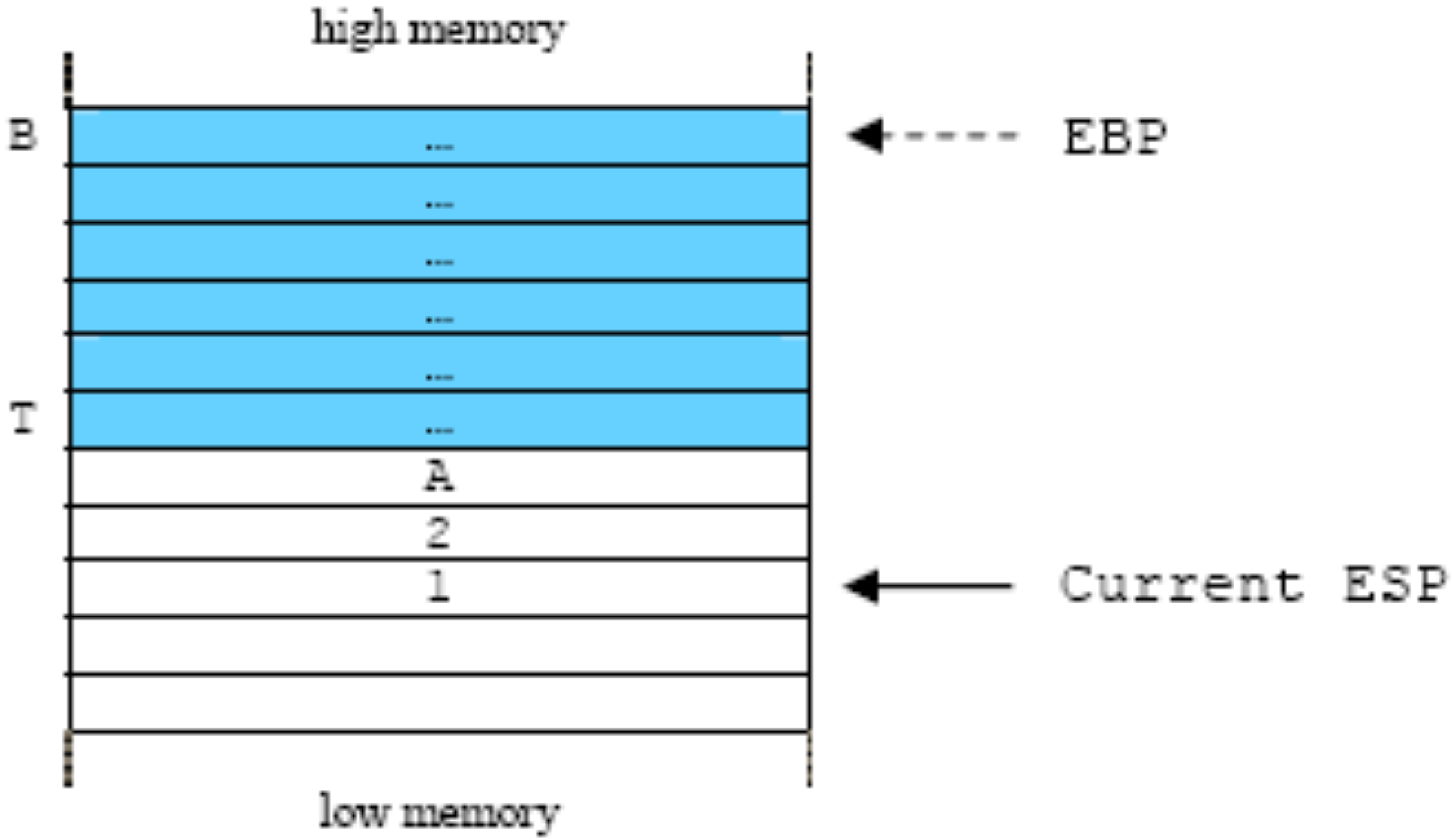
```c
#include <stdio.h>

int TestFunc(int parameter1,int parameter2,char parameter3)
{
int y = 3, z = 4;
char buff[7] = "ABCDEF";

// function's task code here
return 0;
}

int main(int argc, char *argv[ ])
{
TestFunc(1, 2, 'A');
return 0;
}
```

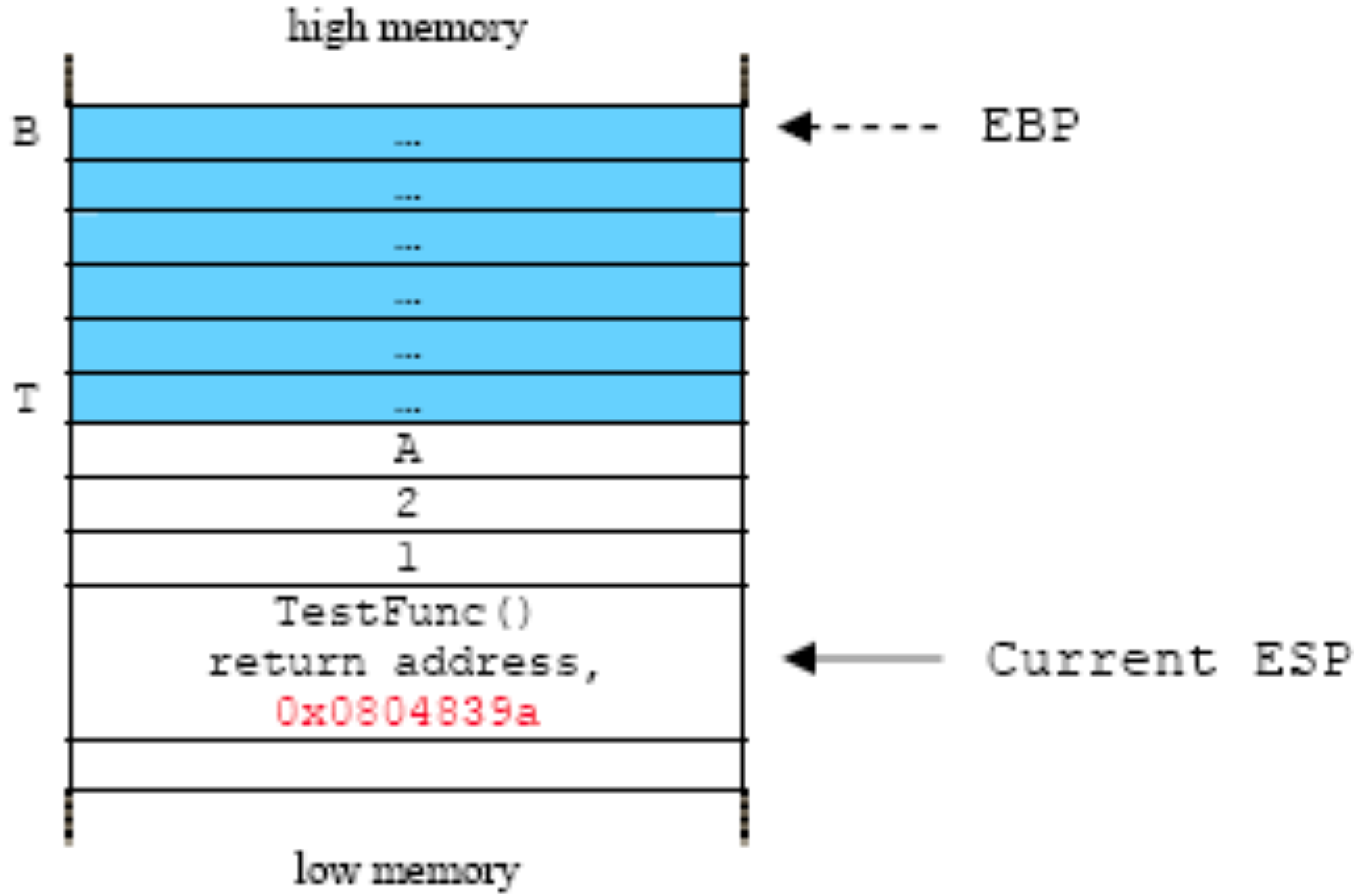
GDB Walkthrough – Call TestFunc

Register eax loaded with character ‘A’ (0x41), not shown

```
0x08048390 <main+36>:   push   %eax                      ;push the third parameter, 'A' prepared in eax onto the stack, [ebp+16]
0x08048391 <main+37>:   push   $0x2                      ;push the second parameter, 2 onto the stack, [ebp+12]
0x08048393 <main+39>:   push   $0x1                      ;push the first parameter, 1 onto the stack, [ebp+8]
--------------------------------------------------------------------------------------------------------------------
0x08048395 <main+41>:   call   0x8048334 <TestFunc>   ;function call. Push the return
                                                      ;address [0x0804839a] onto the stack, [ebp+4]
```


high memory
B
T
EBP
Current ESP
low memory


high memory
B
T
...
...
...
...
...
...
A
2
1
EBP
Current ESP
low memory


high memory
B
...
...
...
...
...
T
...
A
2
1
TestFunc()
return address,
0x0804839a
EBP
Current ESP
low memory

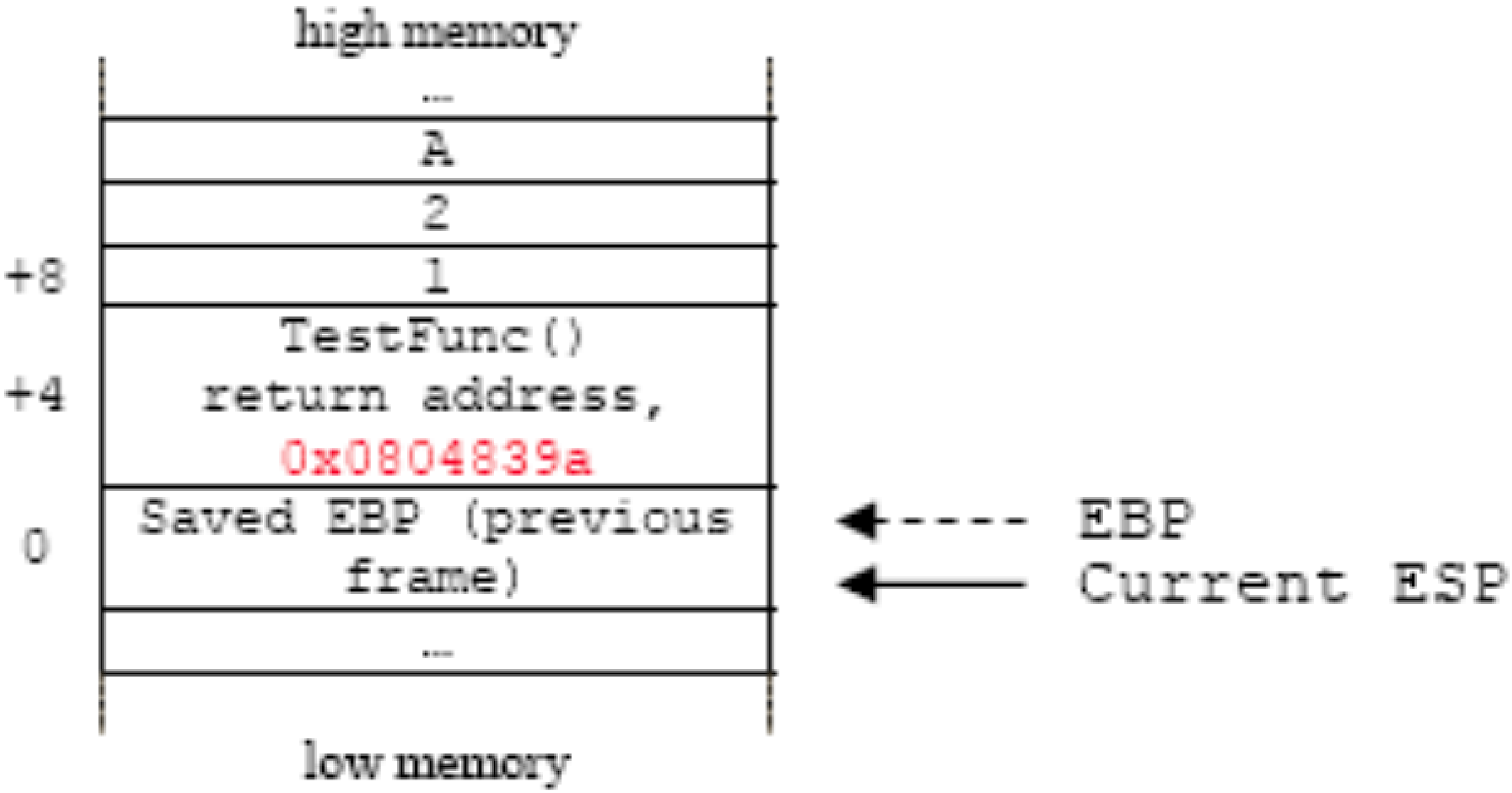
GDB Walkthrough – TestFunc() C Code

```c
int TestFunc(int parameter1,int parameter2,char parameter3)
{
int y = 3, z = 4;
char buff[7] = "ABCDEF";

// function's task code here
return 0;
}
```

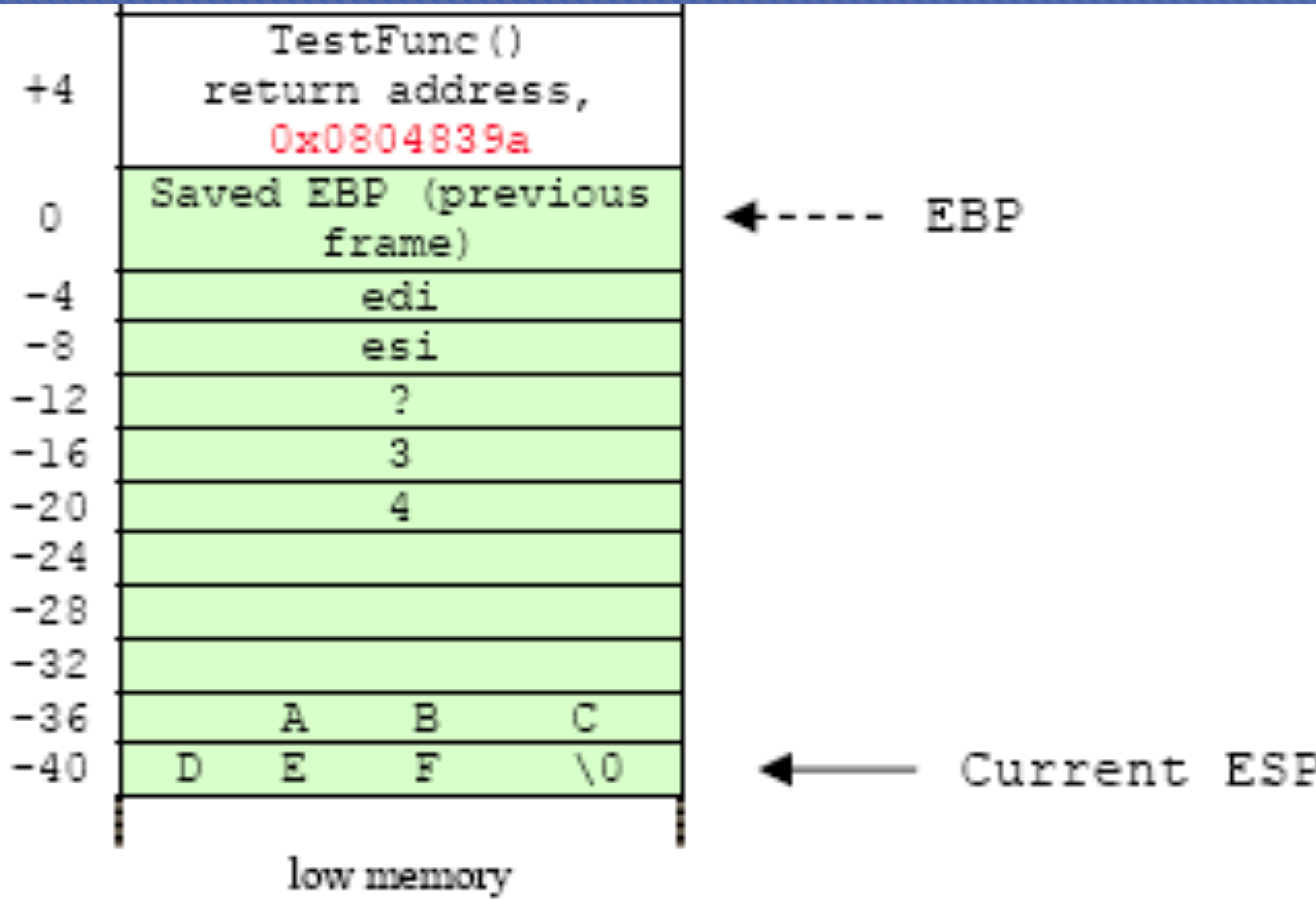
GDB Walkthrough – TestFunc() Assembly

```
0x08048334 <TestFunc+0>:        push    %ebp          ;push the previous stack frame
                                                       ;pointer onto the stack, [ebp+0]
0x08048335 <TestFunc+1>:        mov     %esp, %ebp     ;copy the ebp into esp, now the ebp and esp
                                                       ;are pointing at the same address,
                                                       ;creating new stack frame [ebp+0]
0x08048337 <TestFunc+3>:        push    %edi          ;save/push edi register, [ebp-4]
0x08048338 <TestFunc+4>:        push    %esi          ;save/push esi register, [ebp-8]
0x08048339 <TestFunc+5>:        sub     $0x20, %esp    ;subtract esp by 32 bytes for local
                                                       ;variable and buffer if any, go to [ebp-40]
```

32 bytes allocated on stack (0x20). Variables Loaded into this space (not shown



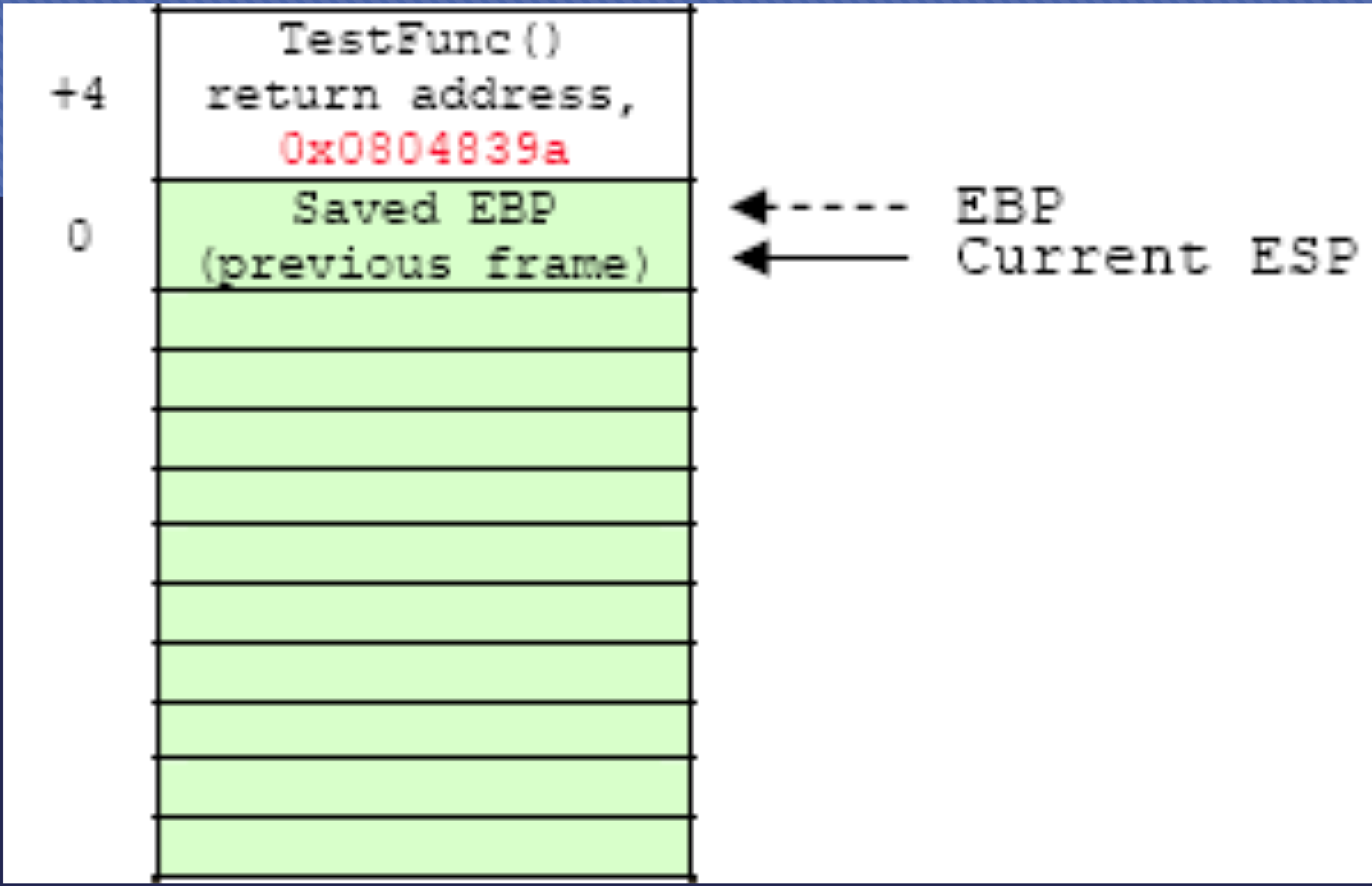
high memory
...
A
2
+8
1
TestFunc()
+4
return address,
0x0804839a
0
Saved EBP (previous frame)
EBP
Current ESP
...
low memory



+4
TestFunc()
return address,
0x0804839a
0
Saved EBP (previous frame)
EBP
-4
edi
-8
esi
-12
?
-16
3
-20
4
-24
-28
-32
-36
A
B
C
-40
D
E
F
\0
Current ESP
low memory

GDB Walkthrough – TestFunc() Exit

```
0x08048365 <TestFunc+49>:   add    $0x20, %esp     ;add 32 bytes to esp, back to [ebp-8]
0x08048368 <TestFunc+52>:   pop    %esi           ;restore the esi, [ebp-4]
0x08048369 <TestFunc+53>:   pop    %edi           ;restore the edi, [ebp+0]
```

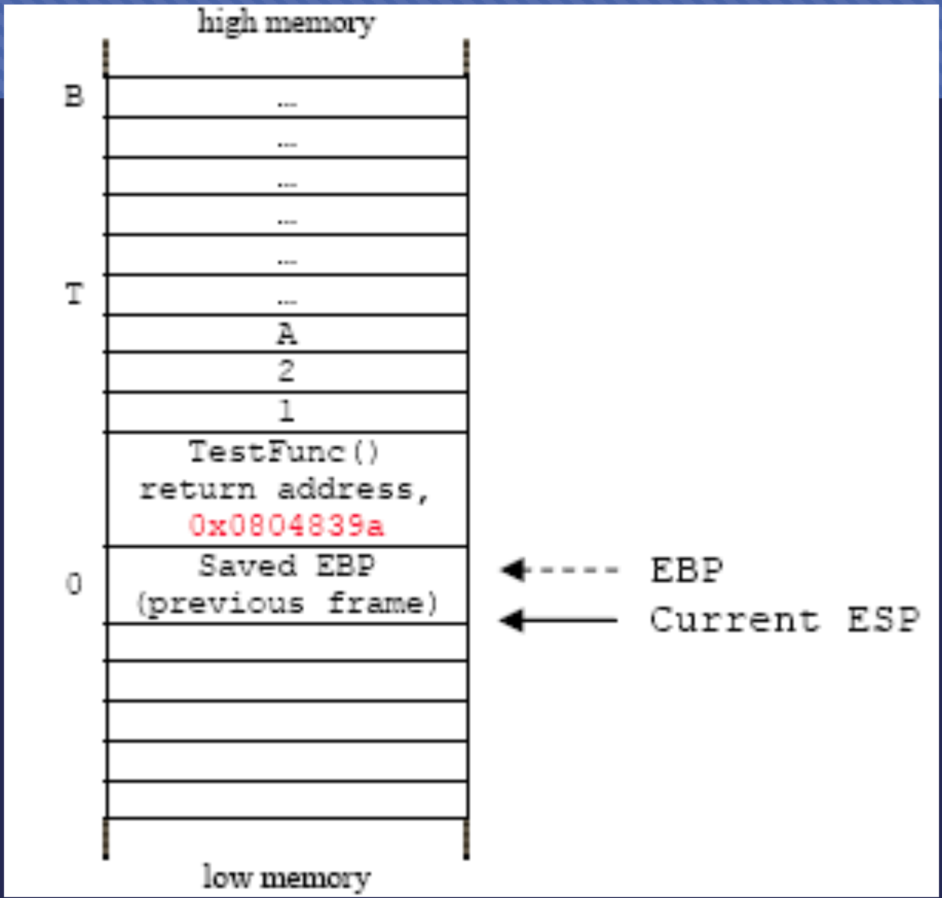

+4
TestFunc()
return address,
0x0804839a
0
Saved EBP
(previous frame)
EBP
Current ESP

GDB Walkthrough – TestFunc() Exit, part 2

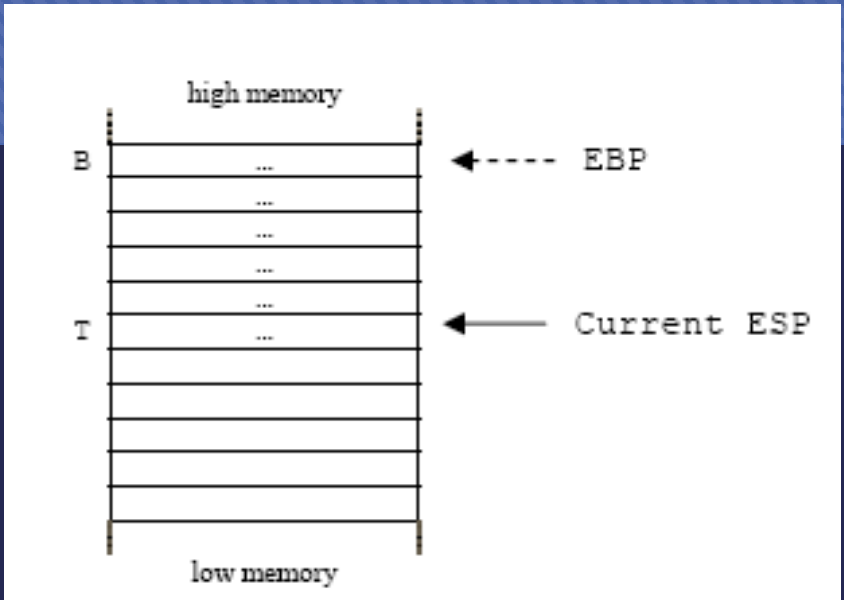
```
0x0804836a <TestFunc+54>:    leave   ;restoring the ebp to the previous stack frame, [ebp+4]
0x0804836b <TestFunc+55>:    ret     ;transfer control back to calling function using
                                      ;the saved return address at [ebp+8]
```

GDB Walthrough – Main() after TestFunc() return

```
0x0804839a <main+46>:   add    $0xc, %esp   ;cleanup the 3 parameters pushed on the stack
                                             ;at [ebp+8], [ebp+12] and [ebp+16]
                                            ;total up is 12 bytes = 0xc
```



high memory
B
...
...
...
...
...
T
...
A
2
1
TestFunc()
return address,
0x0804839a
0
Saved EBP
(previous frame)
EBP
Current ESP
low memory



high memory
B
...
...
...
...
...
T
...
EBP
Current ESP
low memory