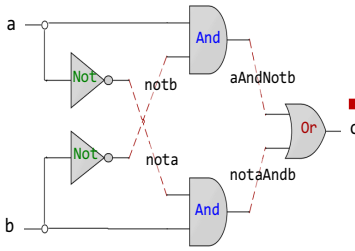
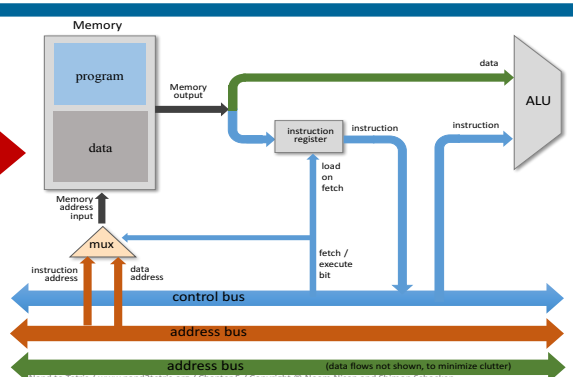
Computer Organization & Assembly Language Programming




```
CHIP Xor {
  IN a, b;
  OUT out;
  PARTS:
  Not(in=a, out=nota);
  Not(in=b, out=notb);
  And(a=nota, b=b, out=w1);
  And(a=a, b=notb, out=w2);
  Or(a=w1, b=w2, out=out);
}
```




```
@R1
D=M
@temp
M=D
```

```
0000000000000001
1111110000010000
0000000000010000
1110001100001000
```

Lecture # 44

C-Function Calling Convention & FSF

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  printf("Learning is fun with Arif\n");
  exit(0);
}
```

```
global main
SECTION .data
   msg: db "Learning is fun with Arif", 0Ah, 0h
   len_msg: equ $ - msg
SECTION .text
   main:
      mov rax,1
      mov rdi,1
      mov rsi,msg
      mov rdx,len_msg
      syscall
      mov rax,60
      mov rdi,0
      syscall
```

```
0:  b8 01 00 00 00
5:  bf 01 00 00 00
a:  48 be 00 00 00 00 00
11: 00 00 00
14: ba 1b 00 00 00
19: 0f 05
1b: b8 3c 00 00 00
20: bf 00 00 00 00
25: 0f 05
```

For resources visit my personal website:
https://www.arifbutt.me
and course bitbucket repository:
https://bitbucket.org/arifpucit/coal-repo

Instructor: Muhammad Arif Butt, Ph.D.

Today's Agenda

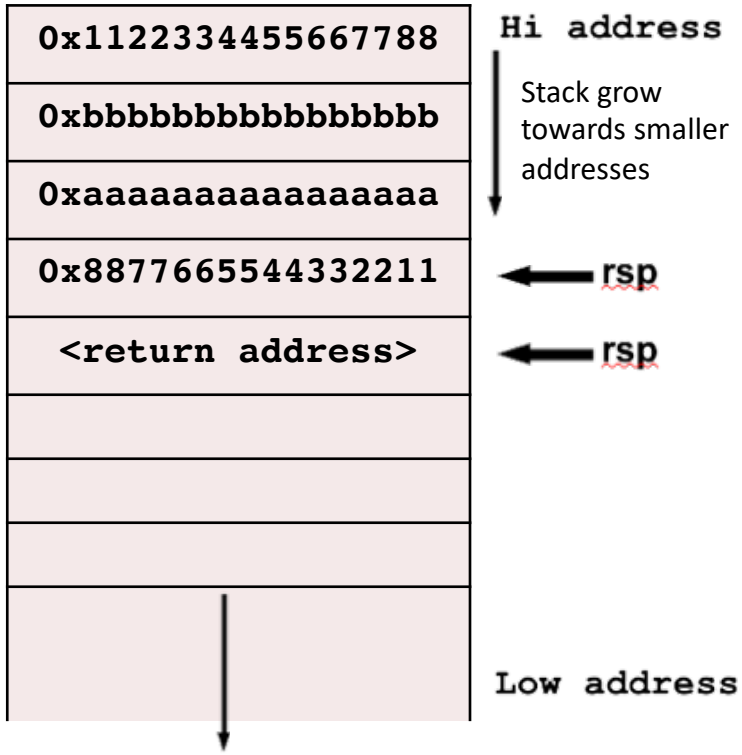
- Recap:
  - The x86 **`call`** and **`ret`** instructions
  - Passing Arguments in x86-64 Assembly Language
- C-Function Calling and the Run-Time Stack
  - Function Stack Frame (FSF)
  - Nested Function Calls
  - Growing and Shrinking of Stack
  - Content of FSF
  - Creation and Removal of FSF
- FSF of a C-Function on x86-64 running Linux OS
  - Demo (***`func_calling.c`***)
- Stack Based Buffer Overflow
  - Demo (***`bufferoverflow.c`***)

Recap

Calling and Returning from Assembly Functions

```
; COAL Video Lecture: 42
;  proc1.nasm
SECTION .data
    msg db "PUCIT", 0xA
    len_msg equ $ - msg
SECTION .text
    global main
    global printmsg
main:
    call printmsg          push rip
                           jmp printmsg
    mov rax,60
    mov rdi, 0
    syscall

printmsg:
    mov rax, 1
    mov rdi,1
    mov rsi,msg
    mov rdx,26
    syscall
    ret                    pop rip
```



Passing Argument and Returning Values

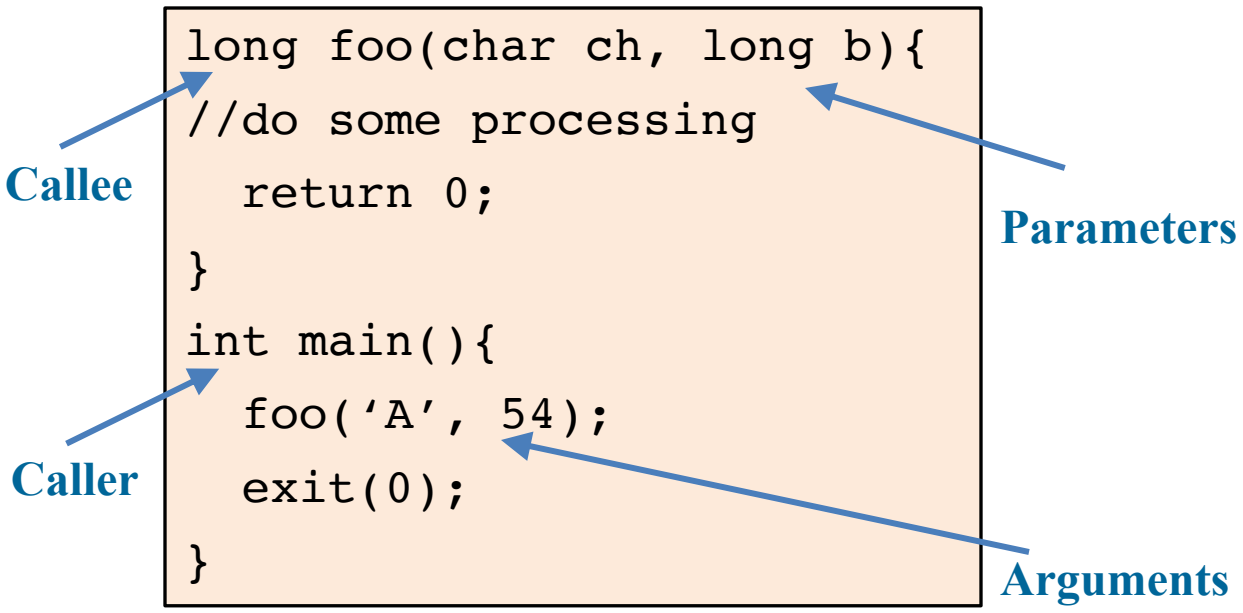
- The agreement on how to pass parameters and return values, and how to share CPU registers between caller and callee is called **calling convention**
- In the 16 and 32 bit days, since there were only eight general purpose registers, therefore, all the parameters were passed via stack in reverse order. This makes the function's first parameter on top of the stack before making the call
- On x86-64, Linux, Solaris and Mac OS use a function call protocol called the **System-V AMD64 ABI**. In which first six integer parameters are passed via registers and first eight floating point parameters via `xmm0` to `xmm7` registers (rest on the runtime stack). The **`rax`** register is used to return integer values and **`xmm0`** register to return floating point values

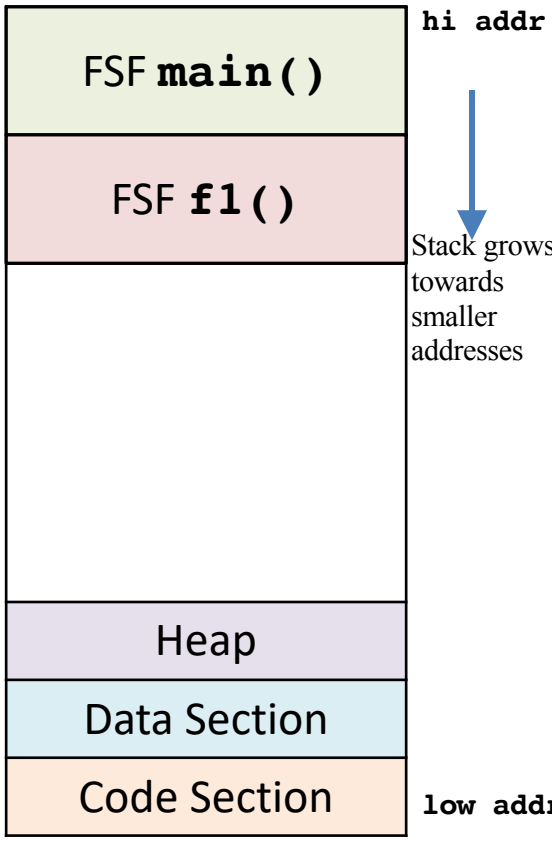
Parameter	Qword	Dword
1	rdi	edi
2	rsi	esi
3	rdx	edx
4	rcx	ecx
5	r8	r8d
6	r9	r9d
>6	Stack	Stack

C Function Calling
&
The Run-Time Stack

The Run-Time Stack

- In high level programming languages like C and C++, the values passed by the caller to the callee are called arguments. When the values are received by the called subroutine, they are called parameters



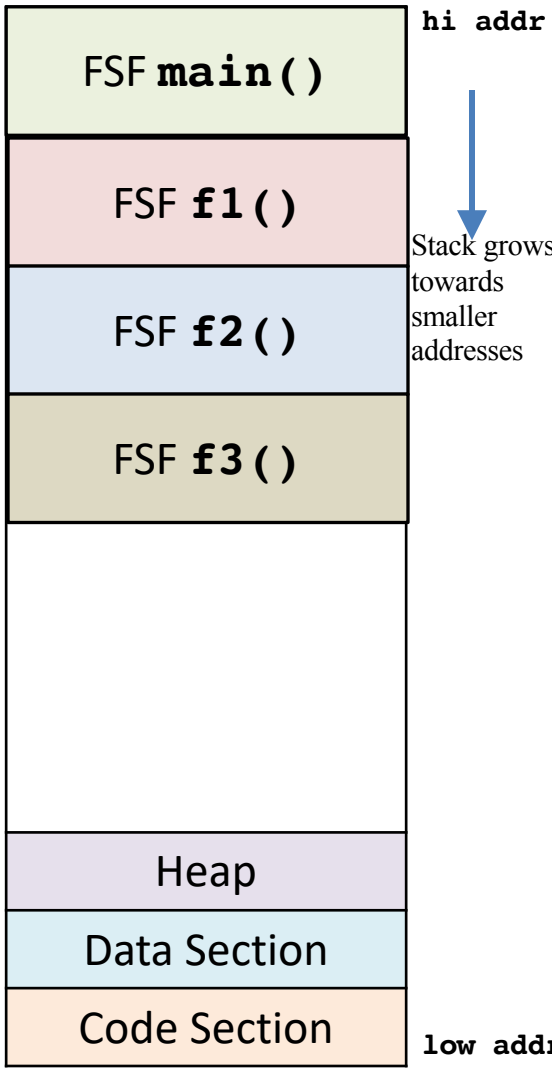


- A Function Stack Frame (FSF) or Activation Record is a stack data structure that is used to store all data on the stack associated with one function. The code for the maintenance of the call stack is generated by high level language compilers

Growing/Shrinking of Run-Time Stack

The Function Stack Frame (FSF) of a function is created on the stack when a function is called and removed from the stack when a function returns

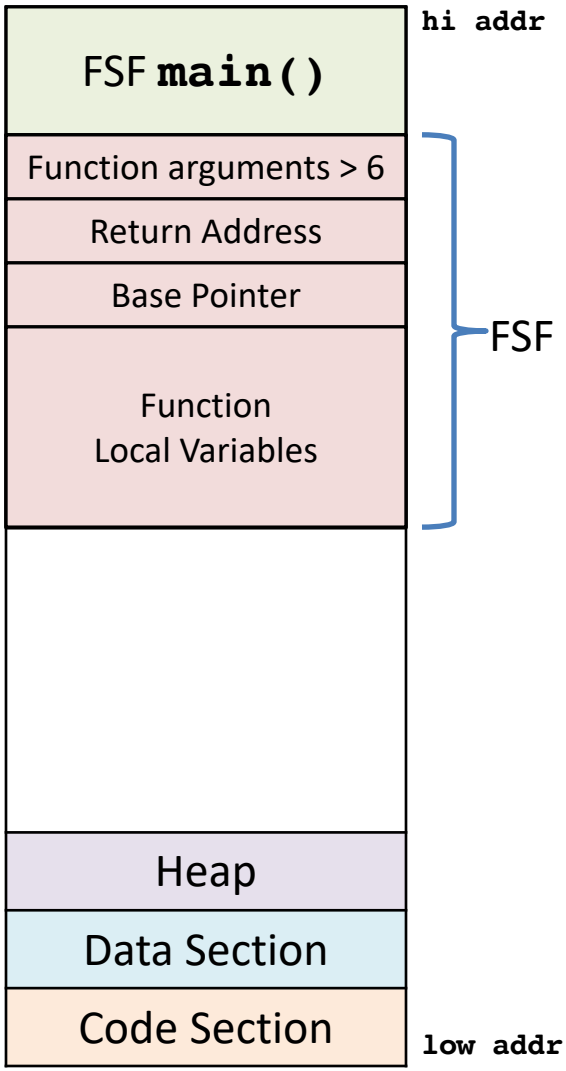
```
int main(){
   f1(54);
   return 0;
}
long f1(long a){
   f2();
   return 0;
}
void f2(){
   f3();
   return;
}
void f3(){
   return;
}
```



Contents of Function Stack Frame

On x86-64 running Linux Operating System, the contents of FSF for a function contains:

1. Function arguments (if greater than 6)
2. Return address of caller
3. The current contents of **`rbp`** register
4. Space for function local variables



FSF: Scuba Diving

On x86-64 running Linux Operating System, the FSF for a function is created by the following sequential steps:

- The function arguments (>6) are pushed on the stack by the caller
- The contents of **`rip`** (return address) is also pushed on the stack

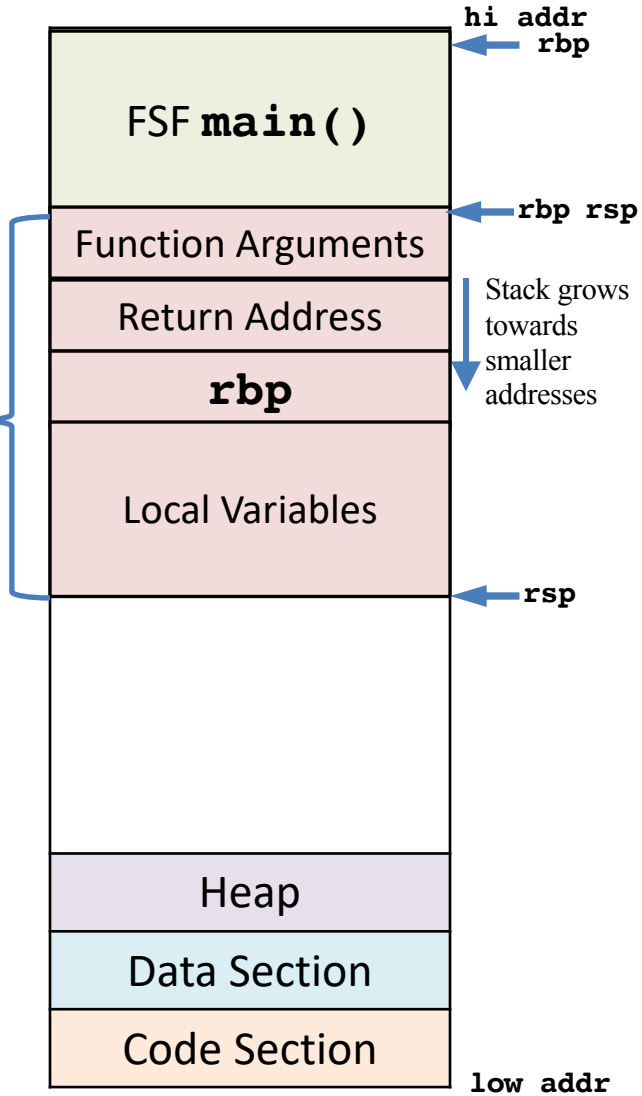
After that, control is shifted to the first instruction of the callee, which performs a **procedure prolog**:

```
PUSH rbp
MOV rbp, rsp
SUB rsp, 0X20
```

When callee is done with its execution, it first cleans up the FSF and then calls the return statement to transfer control to its caller by performing a **procedure epilog**:

```
LEAVE  ──►  MOV rsp, rbp
            POP rbp

RET    ──►  POP rip
```



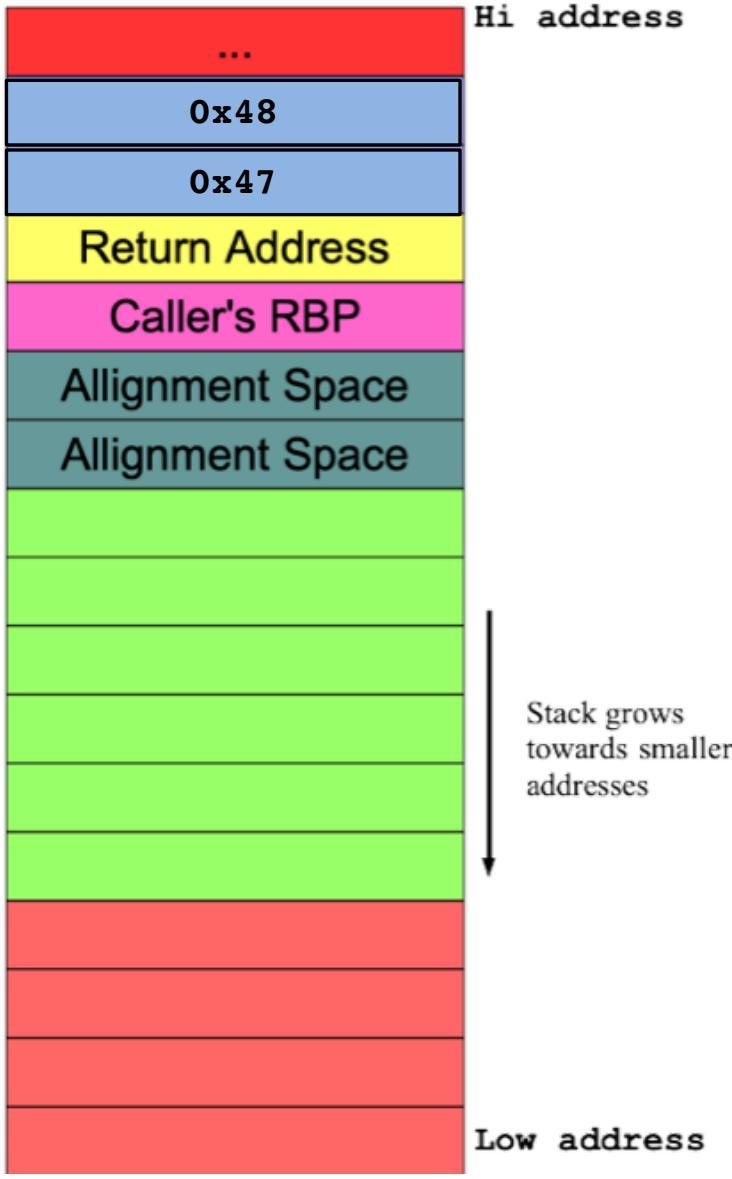
FSF on x86-64 Running Linux OS

```
long foo(long a, long b, long c, long d, long e,
         long f, long g, long h){
//some computation is done
   return 1;
}

int main(){
long rv =foo(0x41,0x42,0x43,0x44,0x45,0x46,0x47,0x48);
  return rv;
}
```

The 8th and the 7th arguments are pushed on the stack from right to left, and the remaining six arguments are moved inside following registers:

r9:	0x46
r8:	0x45
rcx:	0x44
rdx:	0x43
rsi:	0x42
rdi:	0x41

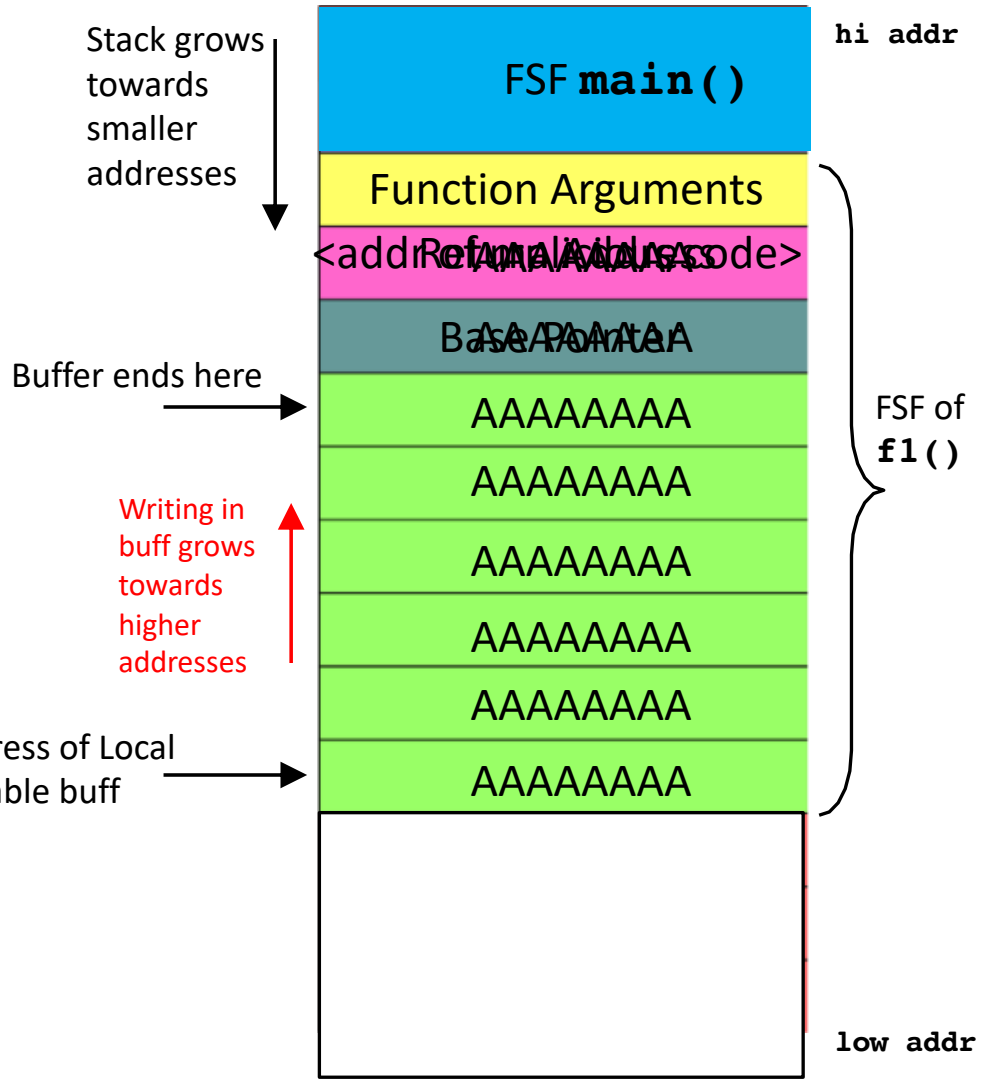


FSF on x86-64 Running Linux OS

Stack Based Buffer Overflow

Stack Based Buffer Overflow

```
void f1(char* data)
{
  char buff[48];
  strcpy(buff, data);
  printf("%s\n", buff);
  return;
}
int main(int argc, char*argv[])
{
  f1(argv[1]);
  exit(0);
}
```



Understanding Stack Based Buffer Overflow

Things To Do

Coming to office hours does NOT mean you are academically week!