Frame pointer

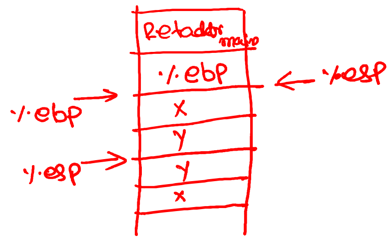
- To simplify code generation and aid debugging, the frame pointer (%ebp) is used on the function prologue and epilogue of the function body

- Function prologue
  - push %ebp
  - mov %esp, %ebp

- Function epilogue
  - mov %ebp, %esp
  - pop %ebp

Example

void ~~int~~ foo() {
  int x = 10;
  int y = 20;
  bar(x, y);
}
int bar(int x, int y) {
  int z = 100;
  int arr[100];
  return x + y + z;
}

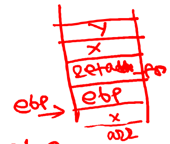


bar:

```
push %ebp
mov %esp, %ebp
sub $404, %esp
mov $100, -4(%ebp)
mov 8(%ebp), %eax
add 12(%ebp), %eax
add -4(%ebp), %eax
mov %ebp, %esp
pop %ebp
ret
```

foo:

```
push %ebp
mov %esp, %ebp
sub $8, %esp
mov $10, 4(%esp)
mov $10, -4(%ebp)
mov $20, -8(%ebp)
sub $8, %esp
mov -4(%ebp), %eax
mov %eax, (%esp)
mov -8(%ebp), %eax
mov %eax, 4(%esp)
call bar
add $8, %esp
mov %ebp %esp
pop %ebp
ret
```



At function entry: caller's %ebp is saved on the stack, and %ebp is set to current %esp. After this step, %ebp points to the stack location that contains the caller's %ebp, (%ebp + 4) points to return address, (%ebp + 8) points to the first argument, and so on. The %ebp value doesn't change throughout the function body, except just before the ret instruction. Due to this relative address of local variables and function parameters corresponding to %ebp remains the same throughout the program. The use of frame pointer is entirely optional. The compilers can still generate code without the frame pointer. The frame pointer simplifies code generation and also aid debugging that we will discuss next.

Example

```
bar:
push %ebp
mov %esp, %ebp
sub $404, %esp
movl $100, -4(%ebp)
mov 8(%ebp), %eax
add 12(%ebp), %eax
add -4(%ebp), %eax
mov %ebp, %esp
pop %ebp
ret
```

```
int bar(int x, int y) {
    int z = 100;
    int arr[100];
    return x + y + z;
}
```

```
foo:
push %ebp
mov %esp, %ebp
sub $8, %esp
movl $10, -4(%ebp)
movl $20, -8(%ebp)
sub $8, %esp
mov -4(%ebp), %eax
mov %eax, (%esp)
mov -8(%ebp), %eax
mov %eax, 4(%esp)
call bar
add $8, %esp
mov %ebp, %esp
pop %ebp
ret
```

```
void foo() {
   int x = 10;
   int y = 20;
   bar(x, y);
}
```

Frame pointer

- The frame pointer is not strictly needed because a compiler can compute the address of variables and its parameters based on its knowledge of the current stack depth

Backtrace

```
foo() {
  bar();
}
bar() {
  baz();
}
baz() {
}
```

~~printf("%p", *(ebp+1))~~; ebp_{foo} →

unsigned * ebp = ebp_reg

printf("%p", ~~*(ebp+1)~~ ebp[1]); ebp_{bar} →

ebp = (unsigned *) ebp[0];

printf("%p", ebp[1]); ebp_{baz} →

The backtrace program prints the call stack. If we save the frame pointers on the stack, we can identify the location of stack locations that store the return addresses. The backtrace program is shown in the next slide.

Backtrace

```
unsigned *ebp = read_register_val(%ebp);
do {
    printf("%p\n", (char*)ebp[1]);
    ebp = ebp[0];
} while (ebp);
```

What is the purpose of registers?

- If all the variables live in memory, what is the role of registers?
  - register access is faster than memory access
  - local variables can live in registers
    - we can’t put all variables because registers are limited
  - registers are used to store temporary computation
  - registers are also used to temporarily store the values of variables (who live in memory)

Registers

C program:	assembly:
foo () {	a : eax
int a, b, c, d, e;	b : ebx
a = 1, b = 2, c = 3, d = 4;	c : ecx
e = a + b + c + d;	d : stack ebp−4
d = e – 1;	e : stack ebp-8
…	
}	

```
foo:
push %ebp
mov %esp, %ebp
sub $8, %esp
mov $1, %eax
mov $2, %ebx
mov $3, %ecx
mov $4, -4(%ebp)
mov %eax, %edx
add %ebx, %edx
add %ecx, %edx
add -4(%ebp), %edx
mov %edx, -8(%ebp)
mov -8(%ebp), %edx
sub $1, %edx
mov %edx, -4(%ebp)
...
```

Registers

```
foo() {
  int a = 1, b = 2, c = 3, d = 4;
  bar(a, b);
  return a + b + c + d;
}
int bar(int x, int y) {
  int a = x + 1;
  int b = y + 1;
  return a + b;
}
```

%eax %ebx %ecx %edx

assembly:

pushf

popf

%esp

```
bar:
  push %ebp
  mov %esp, %ebp
  mov $1, %eax
  add 8(%ebp), %eax
  mov $1, %ebx
  add 12(%ebp), %ebx
  add %ebx, %eax
  mov %ebp, %esp
  pop %ebp
  ret
```

```
foo:
  push %ebp
  mov %esp, %ebp
  mov $1, %eax
  mov $2, %ebx
  mov $3, %ecx
  mov $4, %edx
  sub $8, %esp
  mov %eax, (%esp)
  mov %ebx, 4(%esp)
  call bar
  add $8, %esp
  add %ebx, %eax
  add %ecx, %eax
  add %edx, %eax
  mov %ebp, %esp
  pop %ebp
  ret
```

A function can use registers for its variables. However, a function call may trash the registers values (because the target function may use the same registers). If a caller wishes to use a register across a function call, then it must save/restore them before/after the function call.

Caller saves/restore all live registers

```
foo() {
  int a = 1, b = 2, c = 3, d = 4;
  bar(a, b);
  return a + b + c + d;
}
int bar(int x, int y) {
          %eax
  int a = x + 1;
          %ebx
  int b = y + 1;
  return a + b;
}
```

assembly:

```
foo:
mov $1, %eax
mov $2, %ebx
mov $3, %ecx
mov $4, %edx
push %eax
push %ebx
push %edx
push %ecx
sub $8, %esp
  mov %eax, (%esp)
  mov %ebx, 4(%esp)
  call bar
  add $8, %esp
```

```
foo:
  pop %ecx
  pop %edx
  pop %ebx
  pop %eax
  add %ebx, %eax
  add %ecx, %eax
  add %edx, %eax
  ret
```

In this example, foo is using variables a(%eax), b(%ebx), c(%ecx), d(%edx) that are allocated on registers after calling the bar routine. Because foo doesn't know which registers bar is using, it saves all of them before the function call and restores them after the function call. However, this is not the best strategy. For example, in this case, the bar is not using %ecx and %edx at all. If foo would have known about this fact, then saving/restoring of these registers could have avoided. Because the compiler doesn't see the register allocation of the target routine, the saving/restoring of the values of the registers across the function call is divided among caller and callee. Registers are divided into callee and caller saved registers. If a caller uses a caller-saved register across a function call, then it must save/restore them before/after the function call. If a callee uses a callee saved register, then it must save/restore them at function entry/exit. The callee is free to trash the values of caller saved registers, but not the callee saved registers.

Caller saves/restore caller-saved live registers

caller callee

```
foo() { eax ebx esi edi
  int a = 1, b = 2, c = 3, d = 4;
  bar(a, b);   bar(a,b);
  return a + b + c + d;
}
int bar(int x, int y) {
  int a = x + 1;
  int b = y + 1;
  return a + b;
}
```

assembly:

```
foo:
    push %esi
    push %edi
    mov $1, %eax
    mov $2, %ebx
    mov $3, %esi
    mov $4, %edi
push %eax
push %ebx
    sub $8, %esp
    mov %eax, (%esp)
    mov %ebx, 4(%esp)
    call
    add $8, %esp
    pop %ebx
    pop %eax
```

```
foo:
    add %ebx, %eax
    add %esi, %eax
    add %edi, %eax
    pop %edi
    pop %esi
    ret
```

In this example, we assume that %eax and %ebx are caller saved registers, and %esi and %edi are callee saved registers. The bar implementation is the same as in the previous slide because it is not using any callee saved register. Notice, in this case, when foo calls the bar, then it need not save/restore %esi and %edi before/after calling bar.

Calling convention

- Callee may trash the caller's registers values
- If a caller wishes to use the values of registers prior to a function call after the function call, it must save/restore them before/after the function call
- The job of saving/restoring of register values is divided among the caller and callee

caller and callee saved registers

- Registers are divided into two sets
  - caller saved registers
  - callee saved registers
- callee is allowed to trash a caller saved register
  - If a caller wants to use a caller saved register after a function call then it must save/restore the value of the caller saved register before/after the function call
- callee is not supposed to trash a callee saved register
  - If a callee wants to use callee saved registers, then it must save/restore them in the function prologue/epilogue

caller and callee saved register

- In gcc 32-bit compiler
  - %eax, %ecx, %edx are caller saved registers
  - %ebp, %ebx, %esi, and %edi are callee saved registers

Where does a.out live?

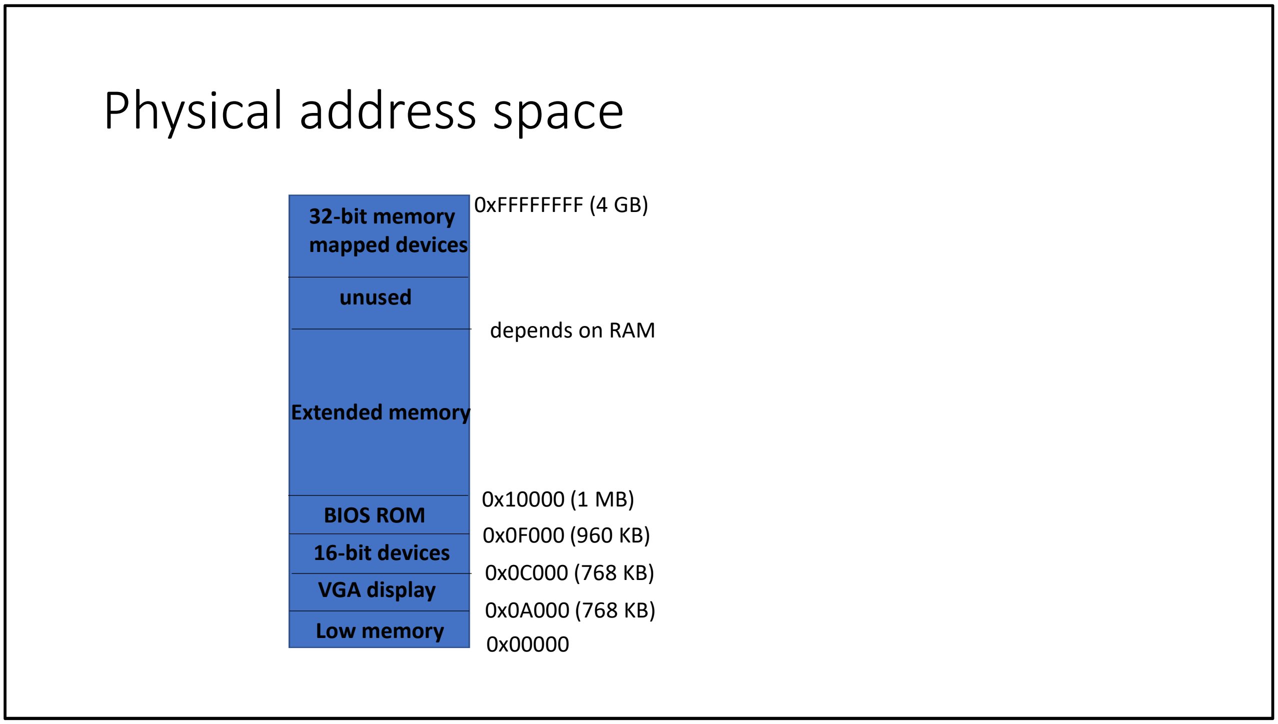
Where does a.out live?

- a.out is a file and lives on disk
- Who loads a.out into the RAM for execution?
  - OS
- Where does OS live?
  - OS is also a software and lives on disk.
- Who loads OS into the RAM for execution?

Where does a.out live?

- a.out is a file and lives on disk
- Who loads a.out into the RAM for execution?
  - OS
- Where does OS live?
  - OS is also a software and lives on disk.
- Who loads OS into the RAM for execution?
  - Part of OS lives in ROM that is persistent memory
  - CPU can execute instructions from ROM
  - OS code in ROM loads rest of the OS into the RAM and transfer control to it

On power-on, the CPU directly jumps to a predefined location (say entry) in ROM. The instructions start at entry load the OS from disk to RAM and transfer control to the main routine of OS.



BIOS ROM is the area where the CPU jumps on startup.