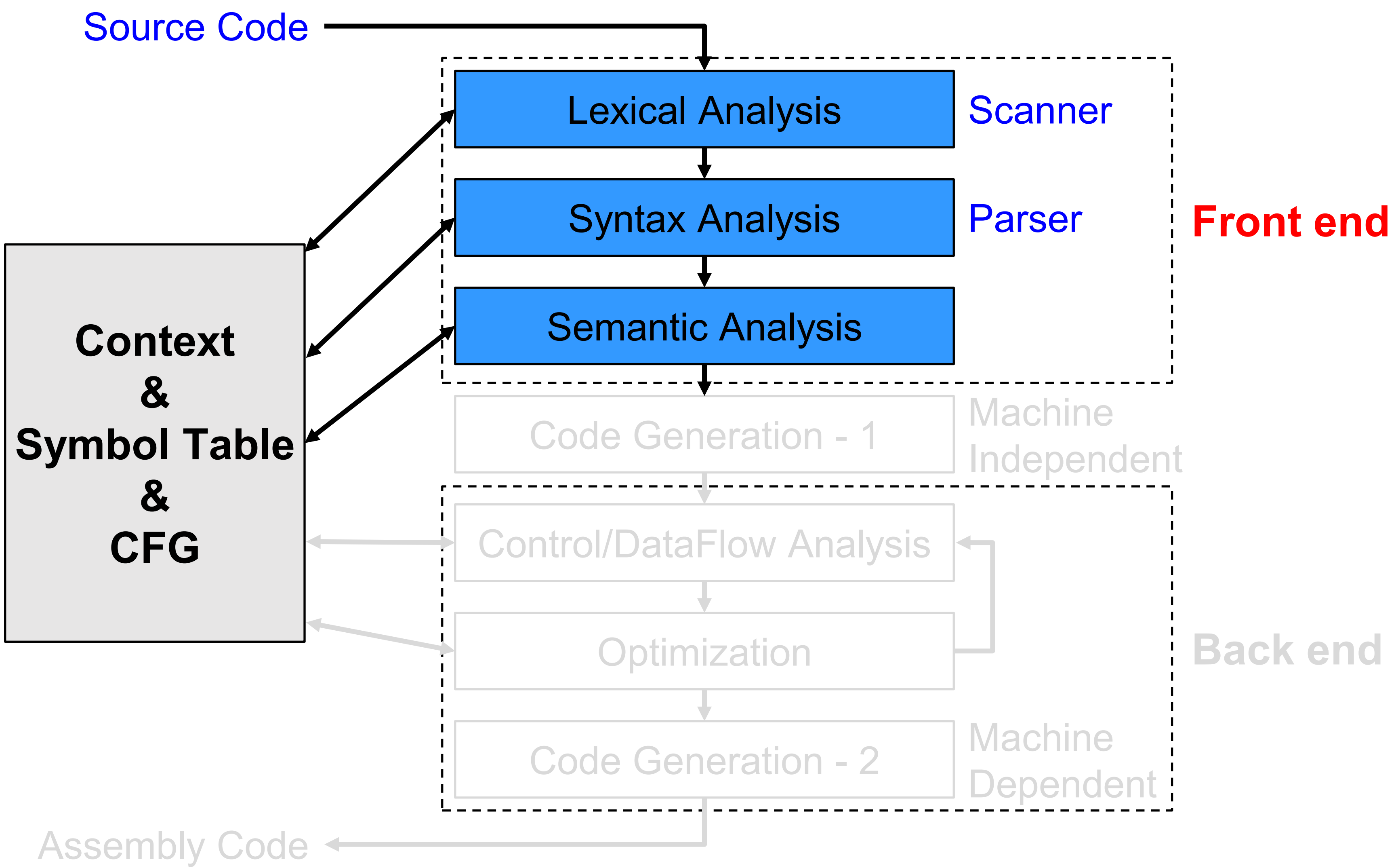
6. Code Generation

2025 Fall

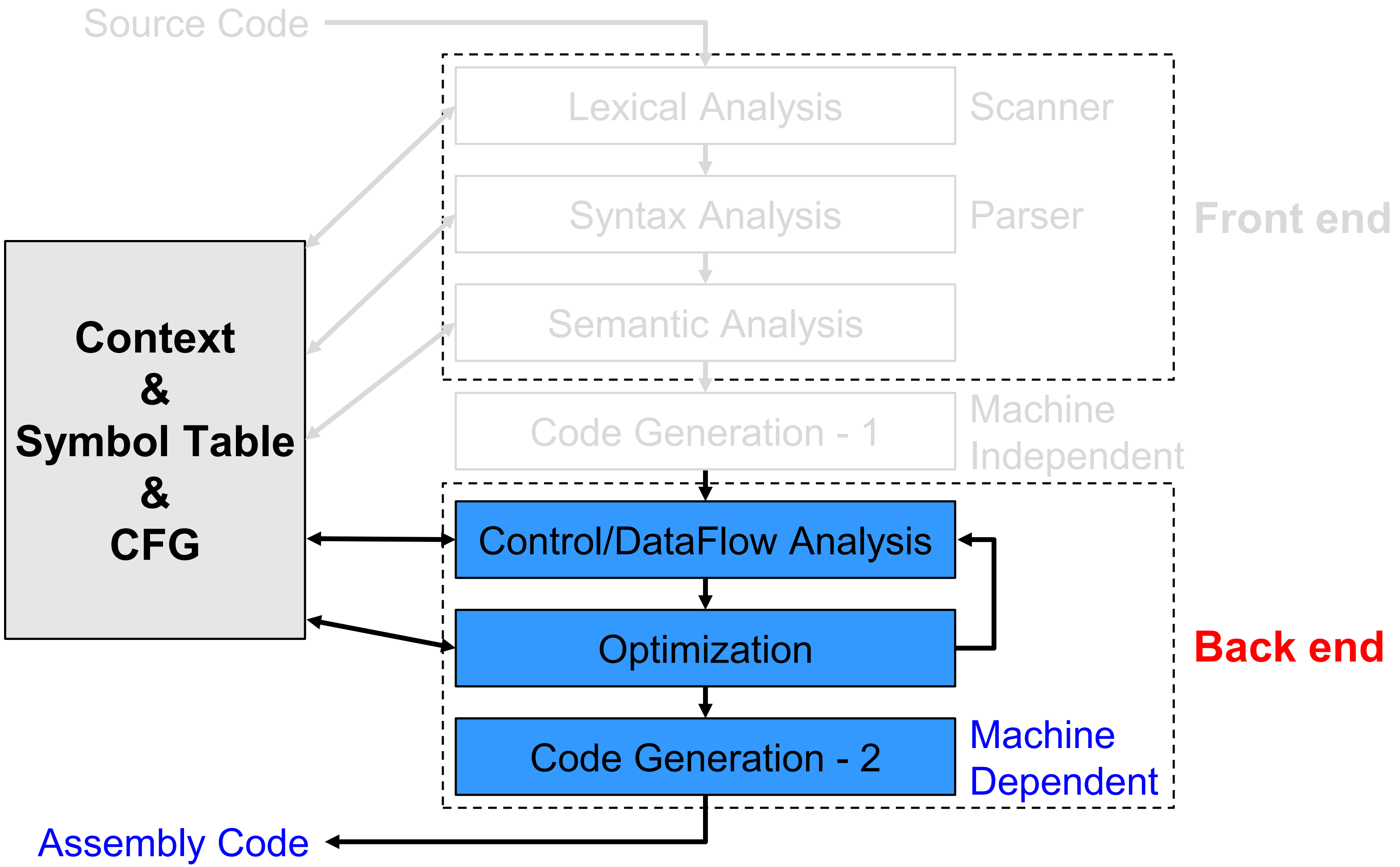
Hunjun Lee

Hanyang University

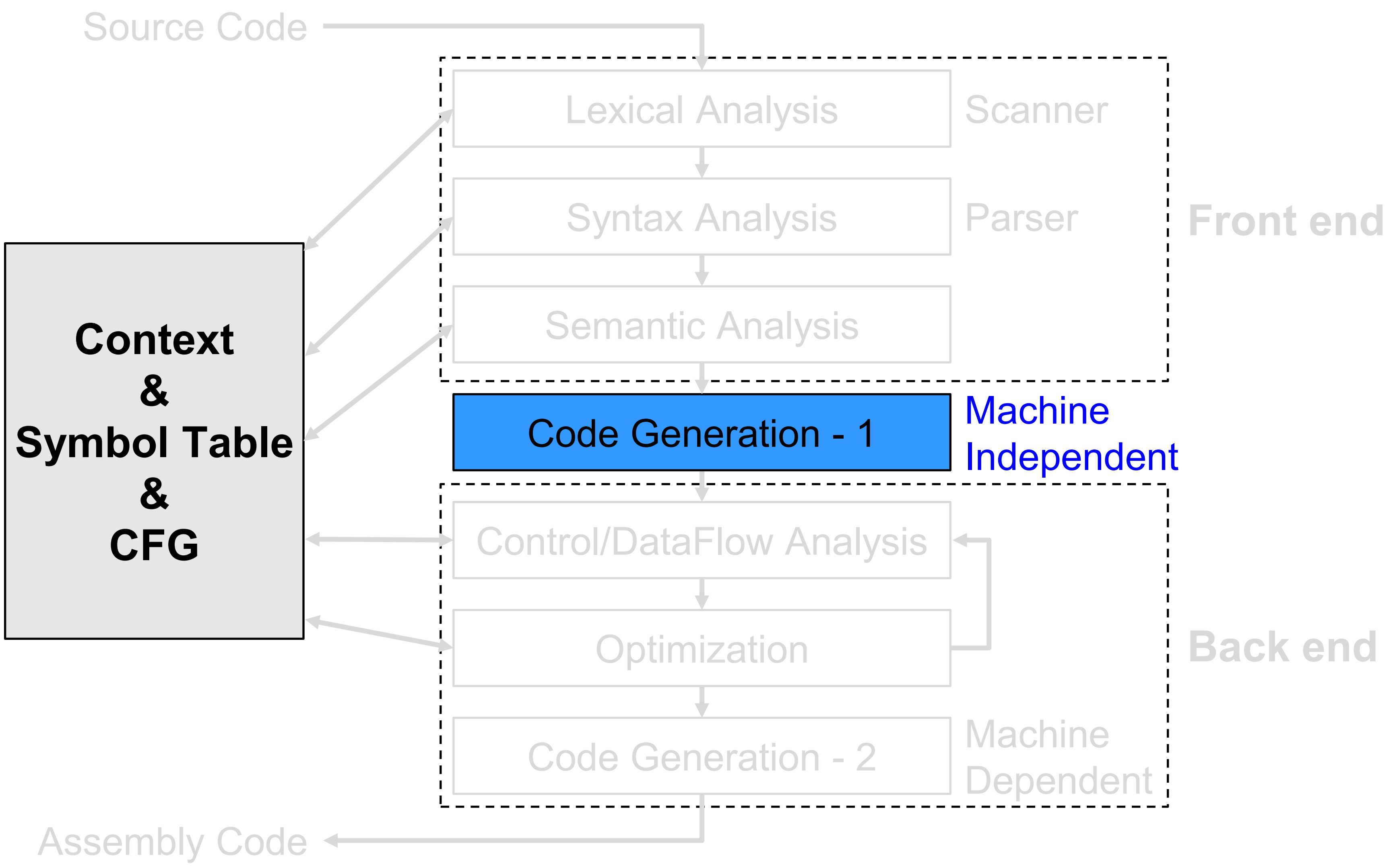
What We've Studied So Far …



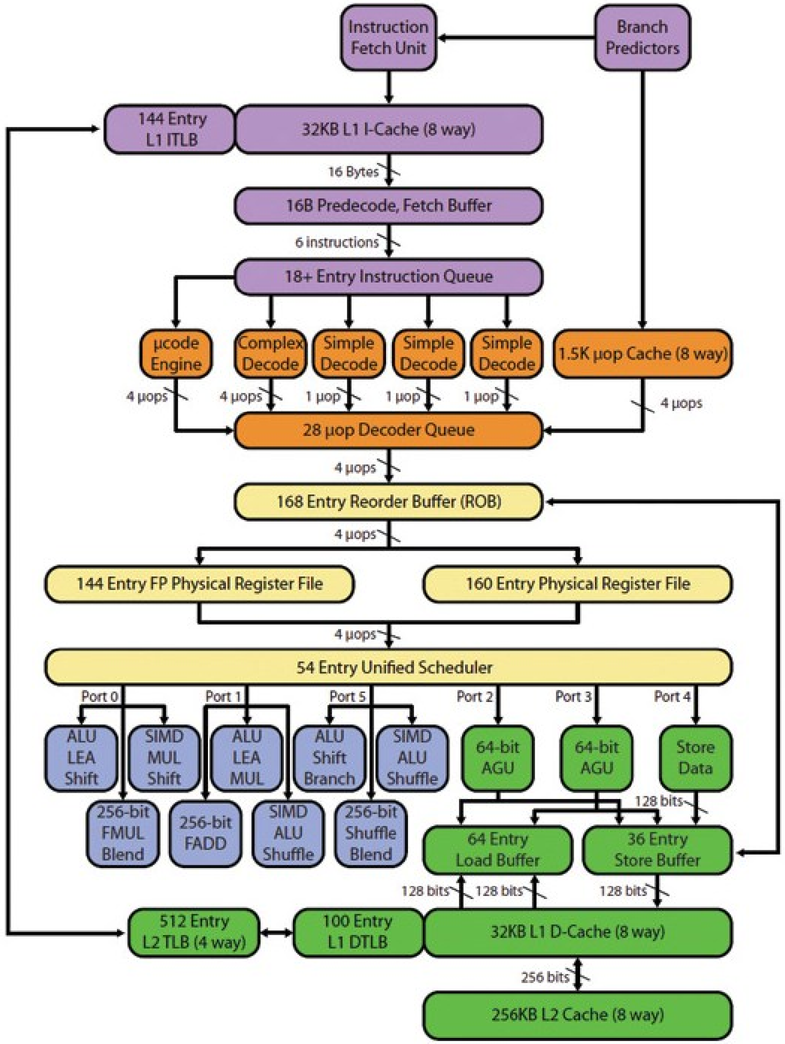
What You Will Learn



What You Will Learn



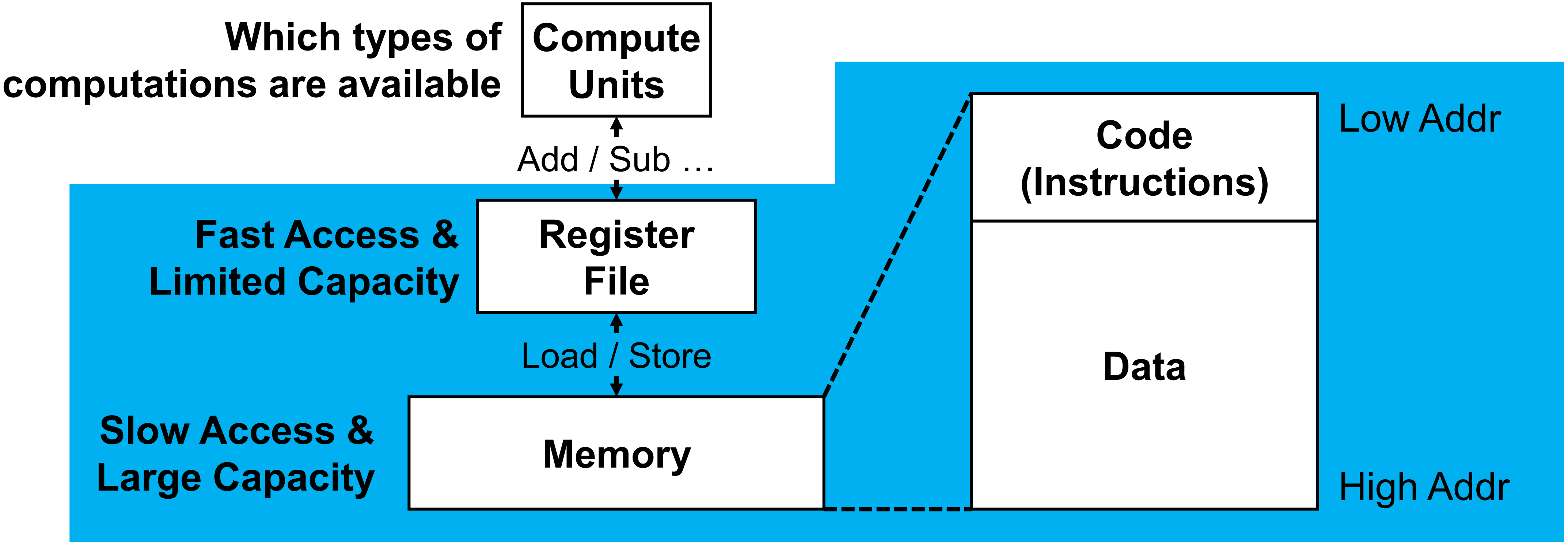
Microarchitecture: What's Behind



- **Core pipeline**
  - CISC→RISC translation
  - Hyper-threading
  - Branch prediction
  - Out-of-order execution
    - 168 Entry ROB & 54 Entry Issue Queue
    - 144~160 Registers
- **Multi-core/multi-thread**
  - 2 thread per core
  - 4-6 cores per chip
- **Cache**
  - 32KB L1 i-cache
  - 32KB L1 d-cache
  - 256KB L2 cache
  - 8-12MB L3 cache

ISA: What the Compiler Knows

- **In a "software view", the CPU consists of a compute units, register file, and memory**



Storage Class Selection Problem

- **Determines where to place the data (register file vs. memory)**

- **Standard approach:**
  - Globals / Statics ➔ memory
  - Locals:
    - Composite types (structs, arrays, etc) ➔ memory
    - Rest ➔ Virtual register (this will be mapped in later lectures)

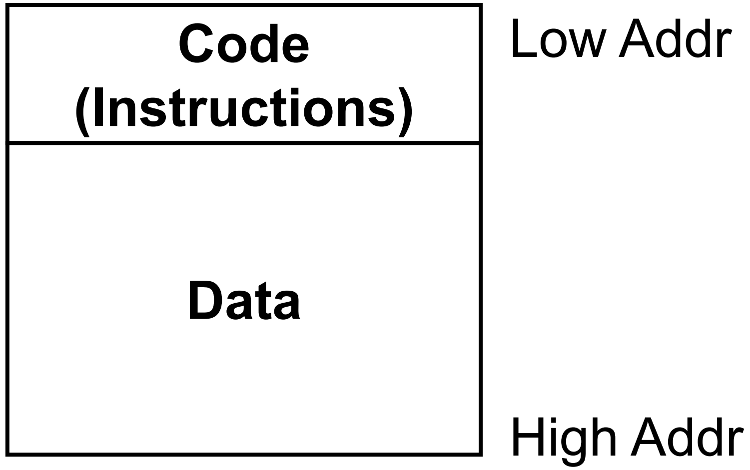
- **All memory approach:**
  - Put all variables into memory

Compiler Backend

- **The compiler is responsible for generating the code**

- **Also, it is responsible for orchestrating (managing) how the data are allocated to the memory space**
  - ➔ Then, generate the code that properly orchestrates the data

The code and data layout are codesigned



Runtime Organization - 1

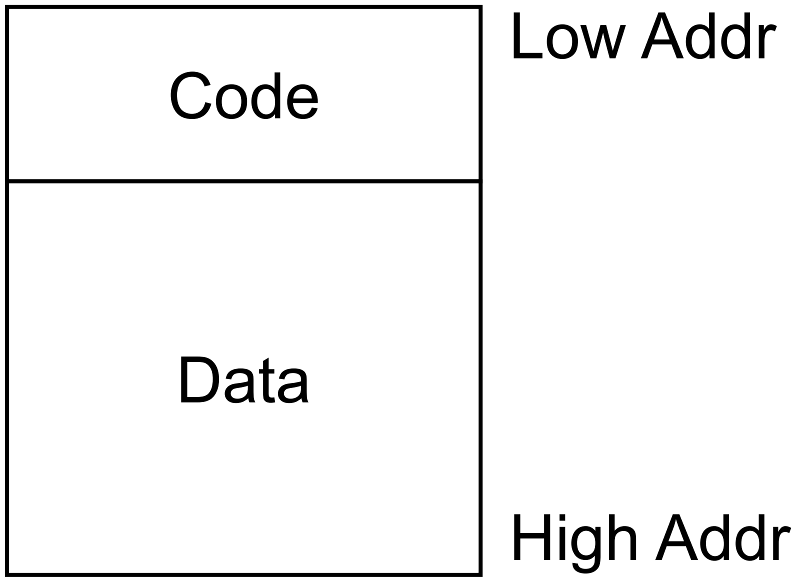
- **You need to understand what we are trying to generate to better understand the code generation process**

- **You need to understand three things**
  - How does the code manage the run-time resources
  - Correspondence between the compile-time and run-time structures
  - Storage organization

Runtime Organization - 2

- **The compiler is responsible for:**
  - Generating code
  - Orchestrating use of the data area

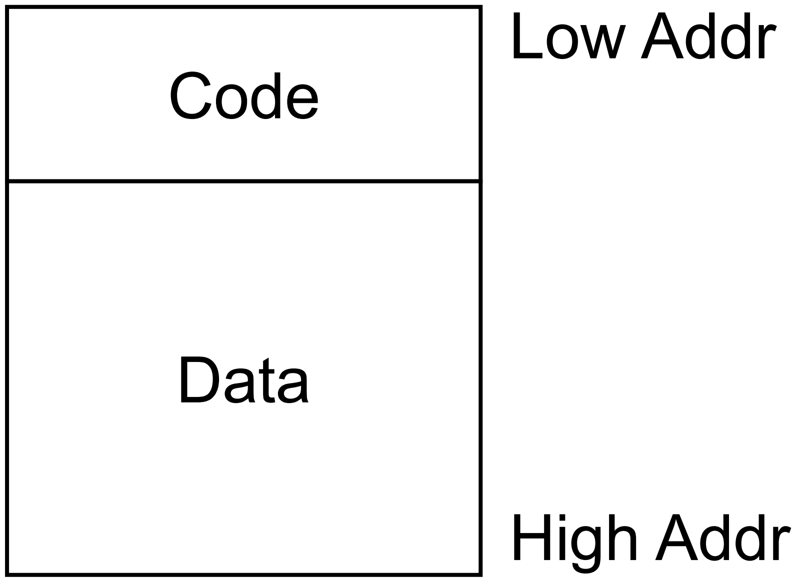
Storage Organization



The code manipulates the data in memory

Executing a Program

- **The operating system controls the program execution**
  - Step #1) Allocates memory space for the program
  - Step #2) The code and data is loaded into the space
  - Step #3) The OS jumps to the entry point (i.e., main)



Execution Sequence

- **Execution is sequential: control moves from one point in a program to another in a well-defined order**
  - Note) concurrency

- **When a procedure is called, control always returns to the point immediately after the call**
  - Note) exceptions

Activation & Lifetime - 1

- **An invocation of procedure P is an activation of P**

- **The lifetime of an activation of P is**
  - All the steps to execute P, including all the sub-procedures in P

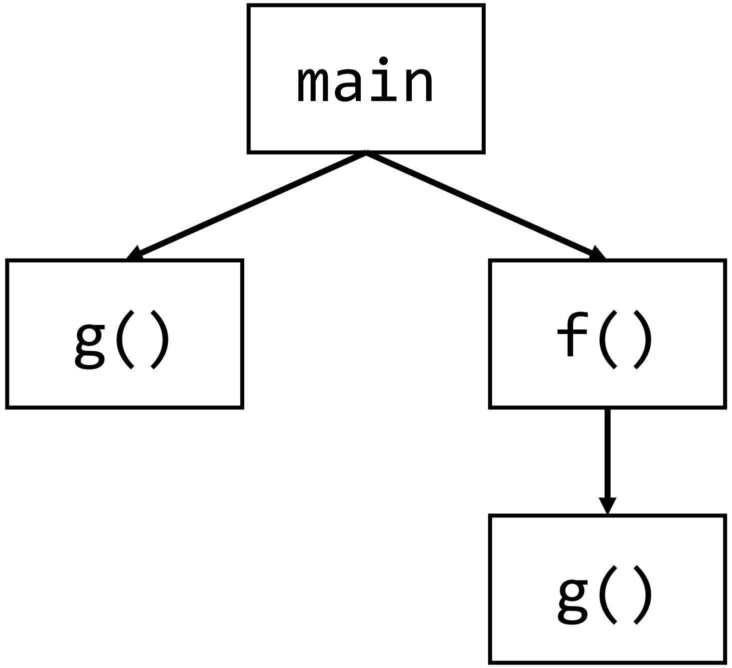
- **The lifetime of a variable x is the portion of execution in which x is defined**

- **Lifetime is dynamic concept**

Activation & Lifetime - 2

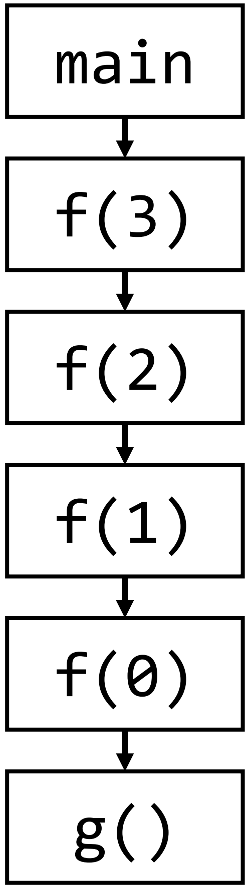
- **When P calls Q, then Q returns before P returns**
  - Lifetimes of procedure activations are properly nested
  - Activation lifetimes can be depicted as a tree (i.e., activation tree)

```
int g() { return 1; }
int f() { return g(); }
void main() {
     g();
     f();
}
```



Complex Example

```
int g() { return 1; }
int f(int x) {
      if (x==0)  return g();
      else       return f(x-1);
}
void main() {
      f(3);
}
```



Activation Tree

- **The activation tree depends on run-time behavior**
  - The tree structure differs for every program input

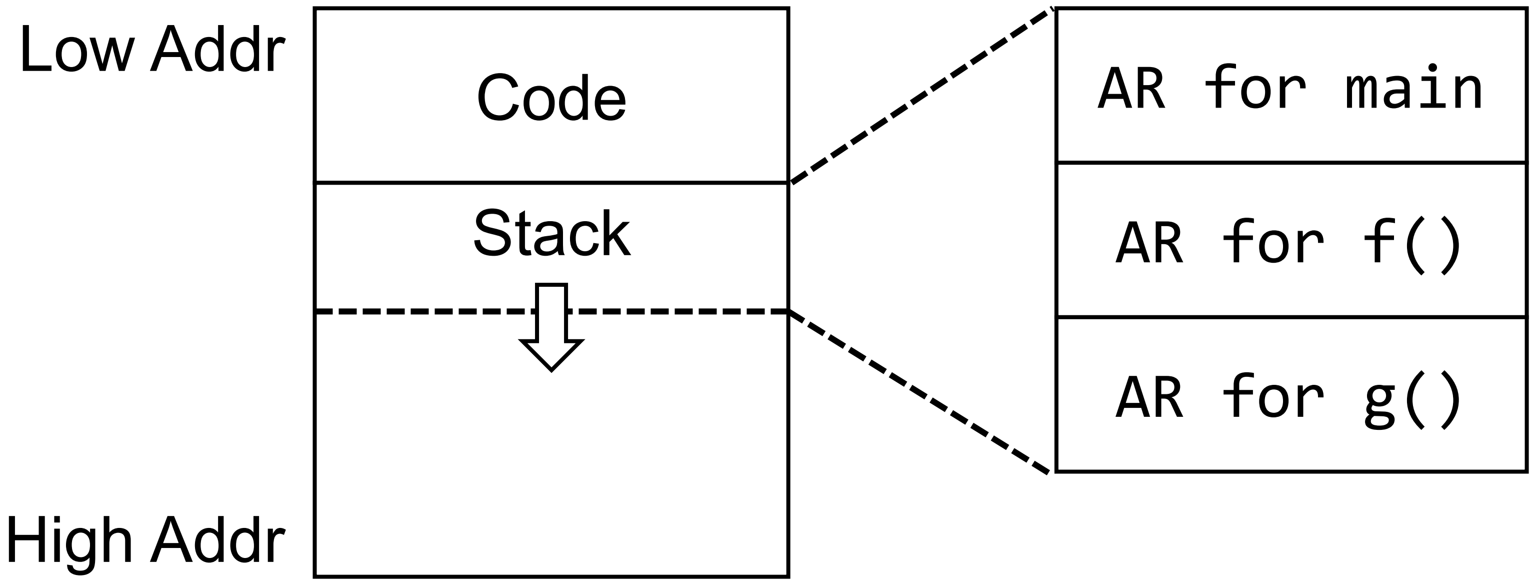
- **We can utilize a "stack" to determine active procedures**

```
int g() { return 1; }
int f() { return g(); }
void main() {
      g();
      f();
}
```

Step1	Step2	Step3	Step4	Step5	Step6	Step7	Step8
main	main	main	main	main	main	main	
	g()		f()	f()	f()		
				g()			

Stack Management

- **Stack data is stored starting from the low address, which grows downwards**
- **The information to manage one activation is called activation record (AR) or frame**



Stack Management Example

```
        int g() {
0x00:       return 1; }
        int f(int x) {
0x04:       int y;
0x08:       if (x==0) {
0x0c:           y = g();
0x10:           return y; }
            else {
0x14:           y = f(x-1);
0x18:           return y; }}
        void main() {
0x1c:       f(3);
0x20:       …}
```

0x100	AR for main	
	Argument x (i.e., 3)	
0x120	Return for f(3)	f(3) AR
	Prev FP (0x100)	
	Local data (int y)	
	Return addr (0x20)	
	Argument x (i.e., 2)	
0x134	Return for f(2)	f(2) AR
	Prev FP (0x120)	
	Local data (int y)	
	Return addr (0x18)	

About Return Data in x86

- **We are assuming an all-memory scheme (therefore, we store the return data in the memory as an example)**

- **However, in modern processors, there is a dedicated register file (`rax`) to store the returned data**
  - It directly stores the data into the `rax` register for primitive types (less that 4 or 8 bytes)
  - It stores the address of the data in the `rax` register

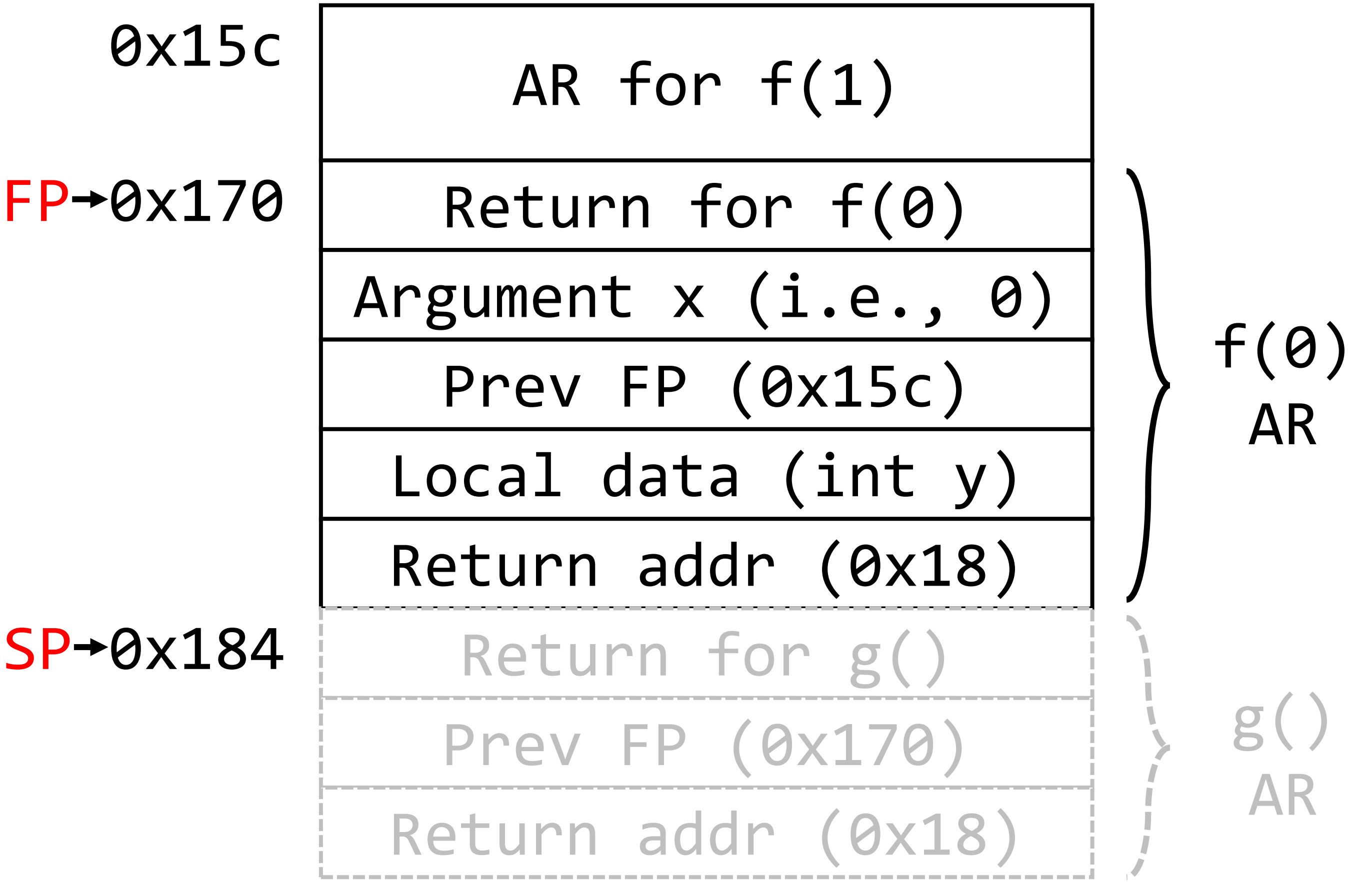
Stack Management Using Register File - 1

- **The compiler utilizes the special register files to index the frame**
  - Stack pointer `(SP)`: points to the top of the frame
  - Frame pointer `(FP)`: points to the frame base
  - Utilize the pointers to index the data in the stack

0x15c	AR for f(1)	
	Argument x (i.e., 0)	
0x170	Return for f(0)	f(0) AR
	Prev FP (0x15c)	
	Local data (int y)	
	Return addr (0x18)	
FP→0x184	Return for g()	g() AR
	Prev FP (0x170)	
	Return addr (0x18)	
SP→0x190		

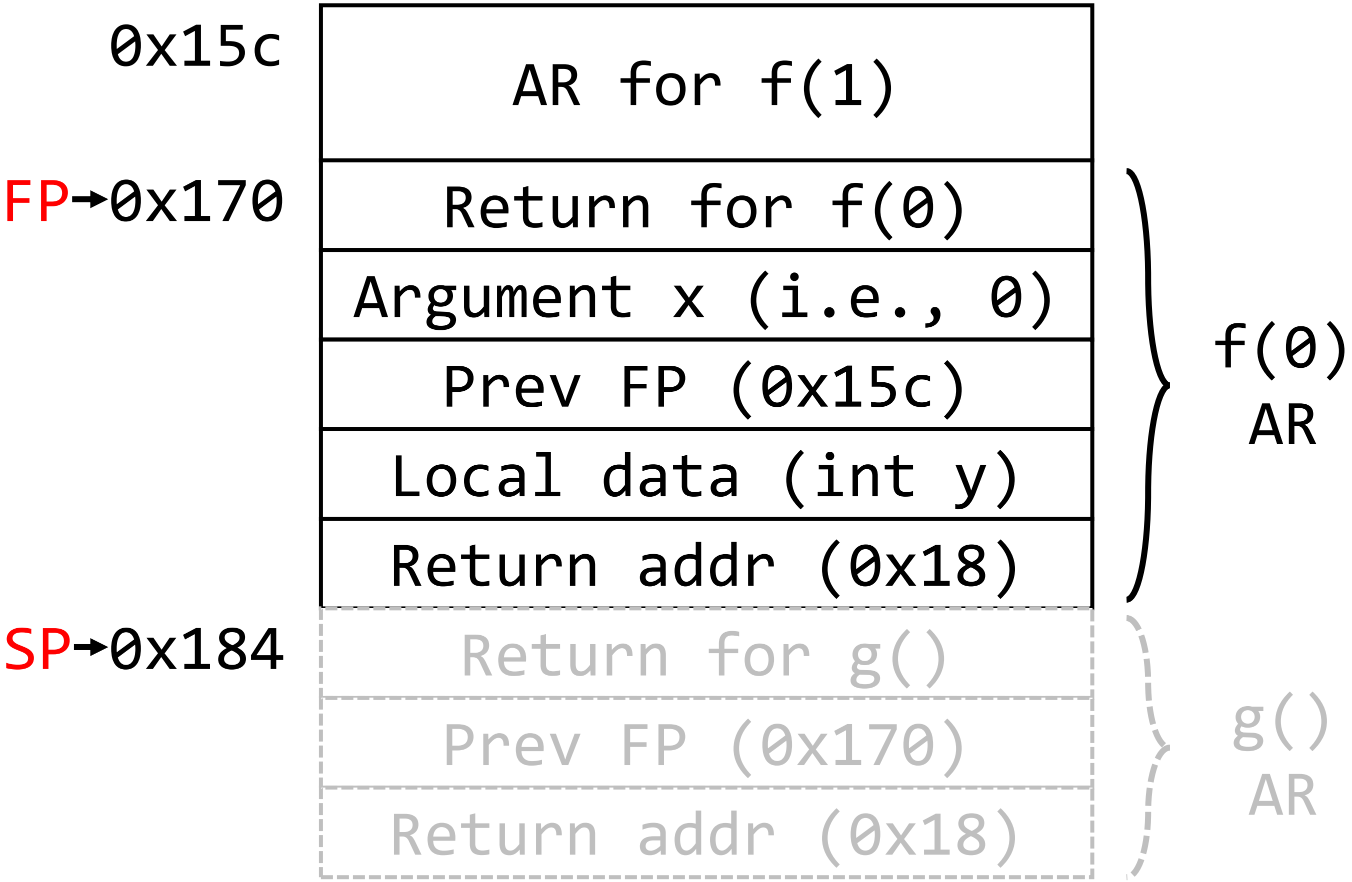
Stack Management Using Register File - 2

- **After the procedures ends, modify the FP and SP**
  - Copy FP to SP
  - Copy the data in the ctrl link to the `FP`
- **Access return data using SP**
  - Load the data in SP (`load $sp`)



Stack Management Using Register File - 3

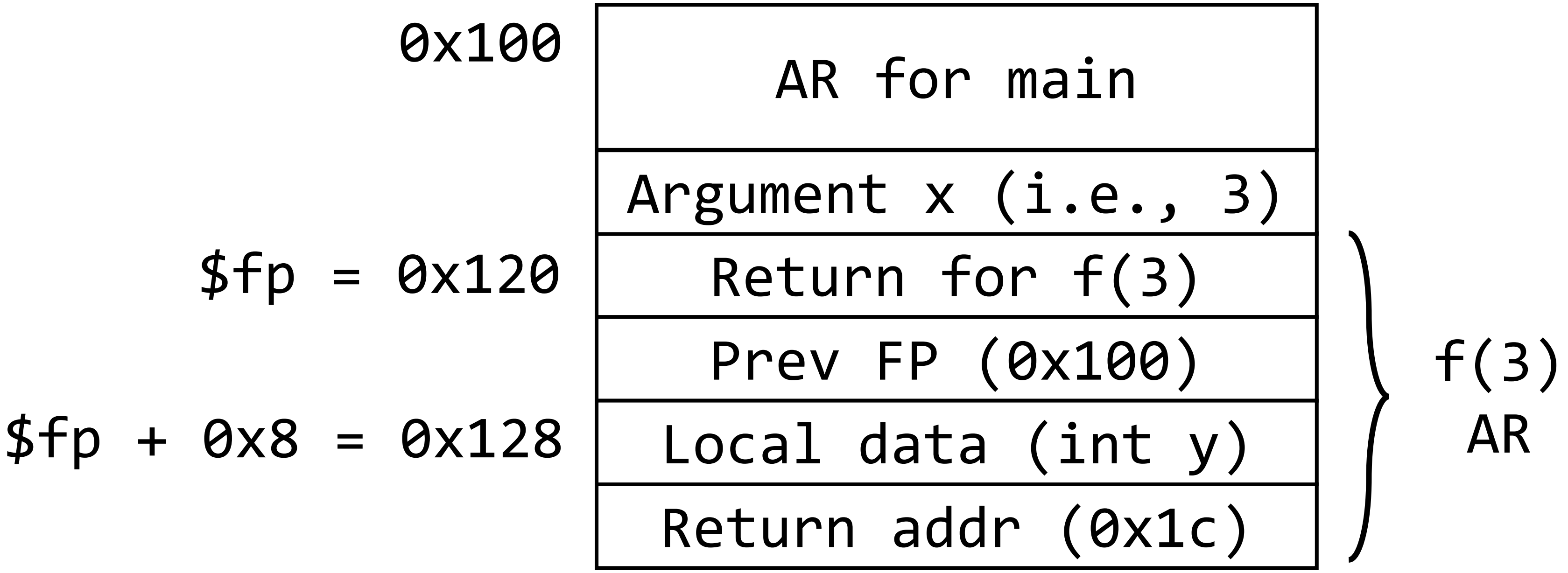
- **Why Two Stack Pointers?**
  - Stack frame size is not always known at compile time (e.g., _alloca)



Stack Management Using Register File - 4

- **The compiler must determine the AR layout, and generate code that correctly addresses locations in the AR**

- **The layout and the code generator must be designed together**



Saving Registers

- **Problem: The callee may overwrite useful values (of the caller) in the registers**

- **The generated code must modify the "stack!"**
  - Save registers when function is invoked (Push)
  - Restore registers when the callee returns (Pop)

- **Possibilities:**
  - Either callee or caller saves and restores the registers
  - **Split the job (both do part of it)**

Pushing Values on the Stack

- **Code before call instruction**
  - Push the parameters
  - Push caller-saved registers
  - Push return address (current PC) and jump to callee code
- **Prologue = code at function entry**
  - Push dynamic link (i.e., FP)
  - Old stack pointer becomes new frame pointer
  - Push callee-saved registers
  - Push local variables

Params
Reg1
Reg2
Return Addr
Prev FP
Reg3
Reg4
Local Variables

Popping Values on the Stack

- **Epilogue = code at return instruction**
  - Pop (restore) callee-saved registers
  - Store return value at appropriate place
  - Restore old stack pointer
  - Pop old frame pointer
  - Pop return address and jump to that address
- **Code after call**
  - Pop (restore) caller-saved registers
  - Use return value

Params
Reg1
Reg2
Return Addr
Prev FP
Reg3
Reg4
Local Variables
Return Val

Example Call - 1

- **Consider call foo(3,5), assume machine has 2 registers r1, r2 that are both callee save**
- **Code before call instruction**
  - push arg1: [sp] = 3
  - push arg2: [sp+4] = 5
  - make room for return address and 2 args: sp = sp+12
  - call foo
- **Prologue**
  - push old frame pointer: [sp] = fp
  - compute new fp: fp = sp
  - push r1, r2: [sp+4] = r1, [sp+8] = r2
  - create frame with 3 local (int) variables, sp = sp+24

Example Call - 2

- **Epilogue**
  - pop r1, r2: r1 = [sp-20], r2 = [sp-16]
  - restore old fp: fp = [sp-24]
  - pop frame: sp = sp-24
  - pop return address and execute return: rts
- **Code after call**
  - use return value
  - pop args: sp = sp-12

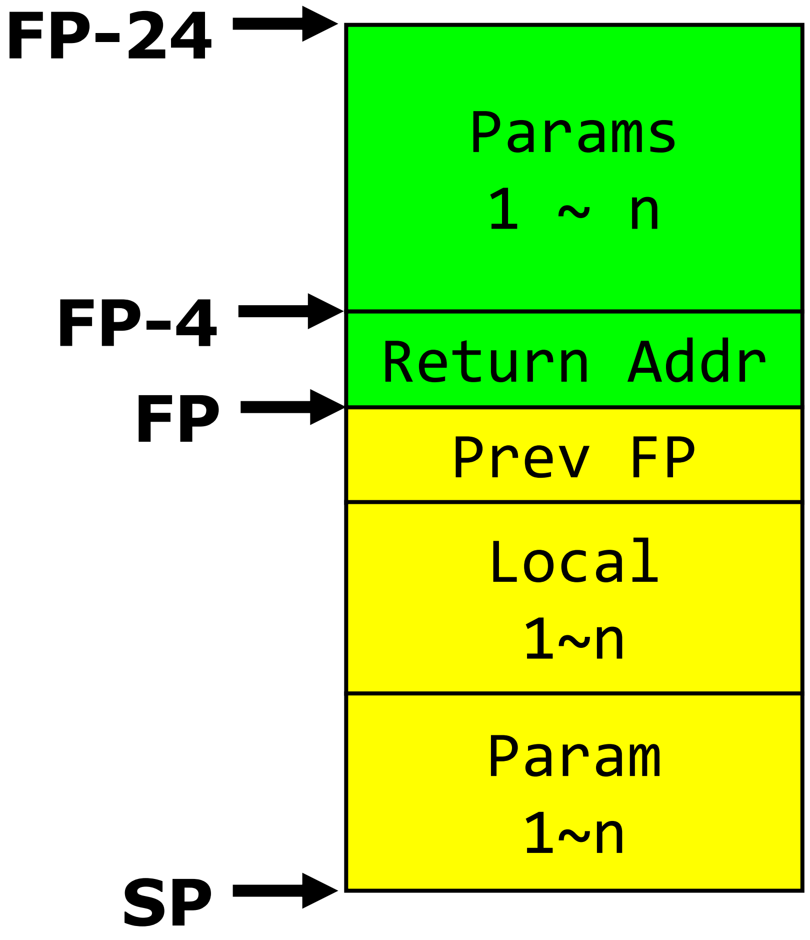
Caller vs. Callee Saved Registers

- **Caller-saved registers (AKA volatile registers, or call-clobbered):** used to hold temporary quantities that need not be preserved across calls
  - It's the caller's responsibility to save / restore the registers (if the caller wants)

- **Callee-saved registers (AKA non-volatile registers, or call-preserved):** used to hold long-lived values that should be preserved across calls
  - It's the callee's responsibility to save / restore the registers (necessary)

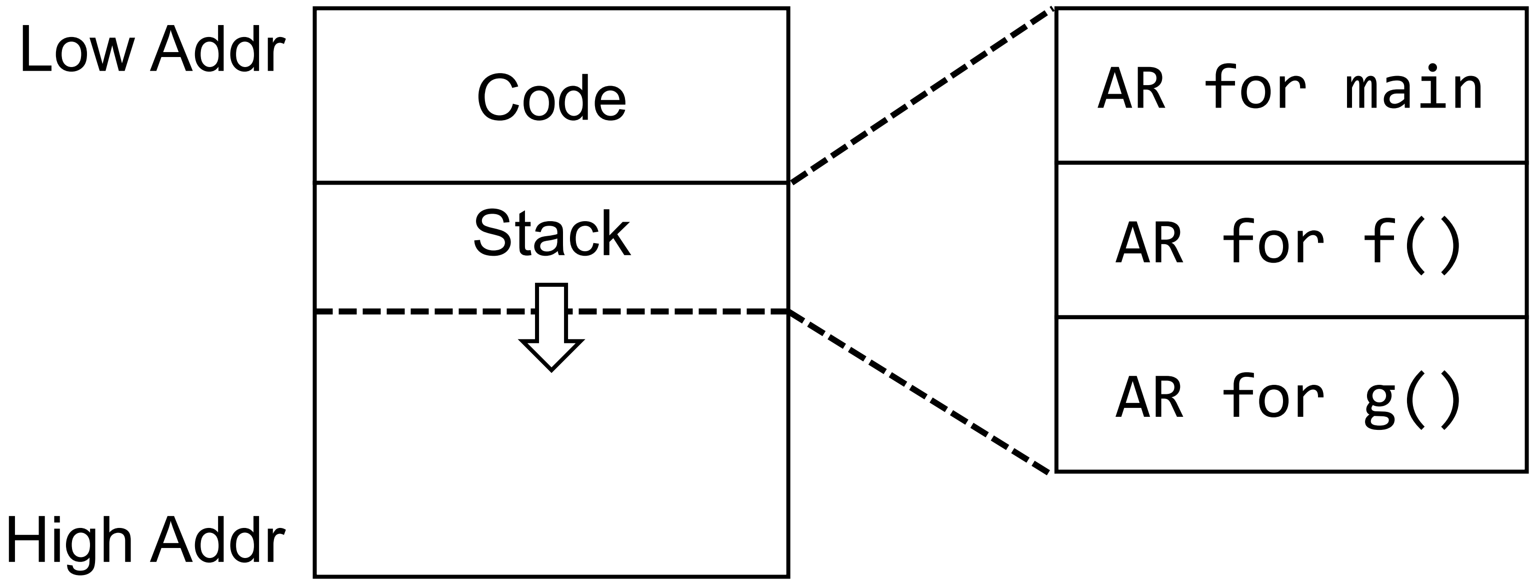
Accessing Stack Variables

- **To access stack variables: use offsets from FP**
- **Example**
  - [fp-8] = param n
  - [fp-24] = param 1
  - [fp+4] = local 1



Review: Stack Management

- **Stack data is stored starting from the low address, which grows downwards**
- **The information to manage one activation is called activation record (AR) or frame**



Global Variables

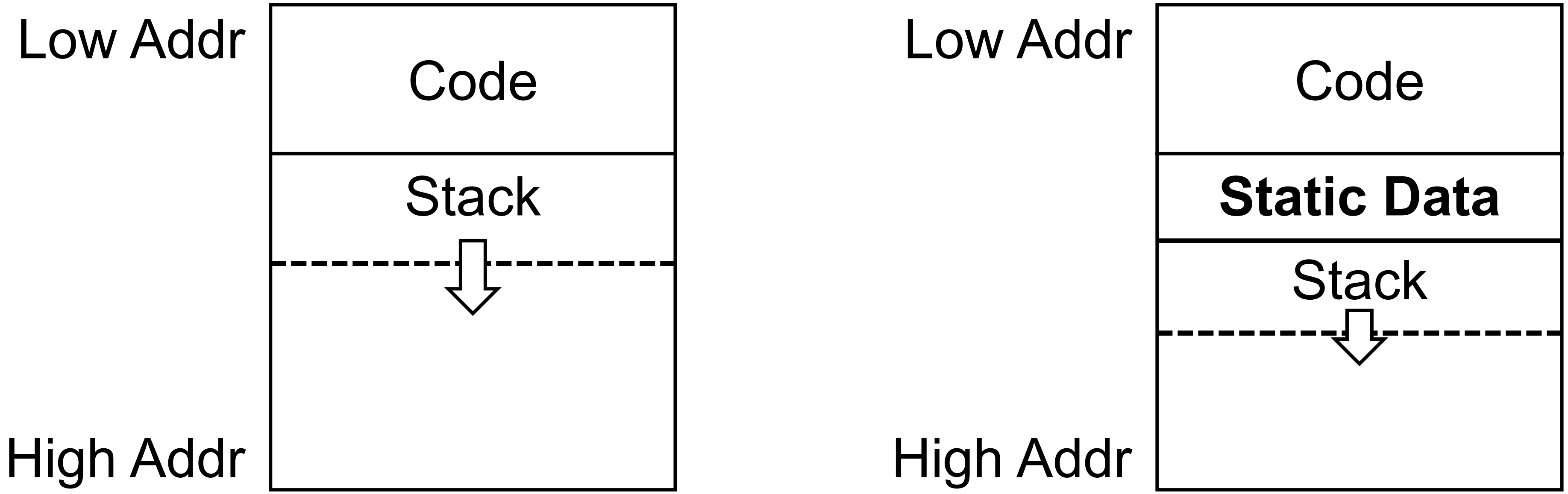
- **All references to a global variable point to the same object**
  - We would be impossible (or inefficient) to store a global activation in an activation record

- **Global variables are assigned a fixed address once (statically allocated)**

- **Depending on the language, there may be other statically allocated values**

Global Variables and Storage Organization

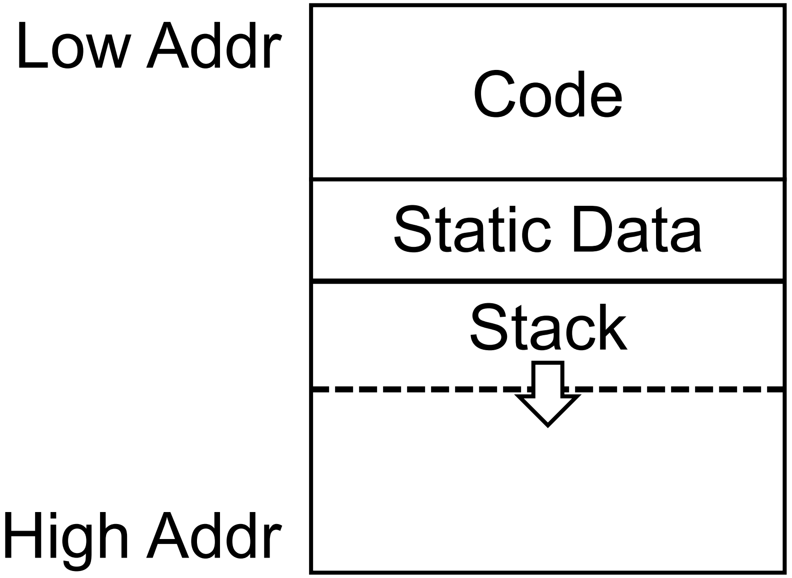
- **We allocate the static variables after the code memory**

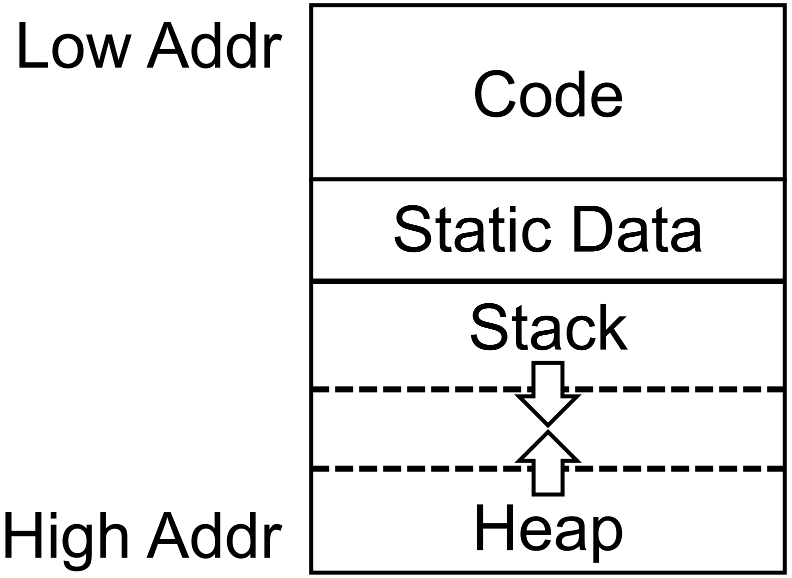


Dynamic Allocation

- **The dynamically allocated value outlives the procedure that creates it (unless deleted beforehand)**

- **We rely on heap to store the dynamically allocated data**





Recap: Organization of Storage

- The code area contains object code which are mostly fixed size and read-only
- The static area contains data with fixed addresses (e.g., global variables)
- The stack contains an AR for each currently active procedure
- The heap contains all other (dynamically allocated) data
  - Ex) C relies on malloc and free

Memory Addressing - 1

- **Most modern machines are 32 or 64 bit**
  - 8 bits in a byte
  - 4 or 8 bytes in a word

- **Machines are either byte or word addressable**
  - The addressing format determines how the data is aligned in the memory
  - A word addressable memory keeps the data word-aligned (it begins at a word boundary)

Memory Addressing - 2

- **Machines are either byte or word addressable**
  - The addressing format determines how the data is aligned
  - A word addressable memory keeps the data word-aligned

Adopted in memory

0x00	'H'
0x01	'e'
0x02	'l'
0x03	'l'
0x04	'o'
0x05	'\0'

Byte Addressing

Adopted in register file

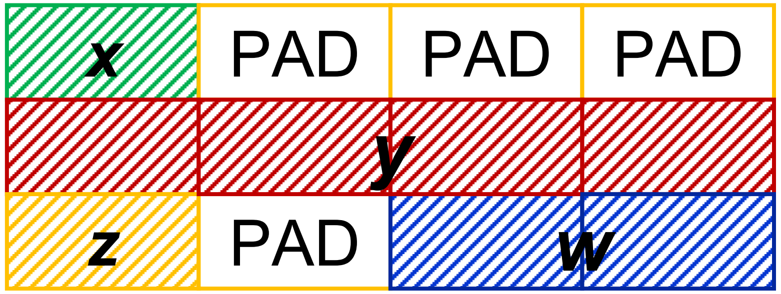
0x00	'H'	'e'	'l'	'l'
0x01	'o'	'\0'	PAD	PAD

Word Addressing

Memory Alignment

- **An address of a variable is aligned based on the size of the variable**
  - Char is byte aligned (any addr is fine)
  - Short is halfword aligned (LSB of byte addr must be 0)
  - Int is word aligned (2 LSBs of byte addr must be 0)

```
char x;      // size 1 byte
int y;       // size 4 byte
char z;      // size 1 byte
short w;     // size 2 byte
```



How are the data stored?
Big Endian vs. Little Endian

- **32-bit signed or unsigned integer comprises 4 bytes**

MSB (most significant)				LSB (least significant)
	8-bit	8-bit	8-bit	8-bit

- **Consider storing 0x12345678**

Big Endian

MSB Addr → LSB

12	34	56	78
byte 7	byte 6	byte 5	byte 4
byte 11	byte 10	byte 9	byte 8
byte 15	byte 14	byte 13	byte 12
byte 19	byte 18	byte 17	byte 16

pointer points to the big end

Little Endian

LSB Addr → MSB

78	56	34	12
byte 4	byte 5	byte 6	byte 7
byte 8	byte 9	byte 10	byte 11
byte 12	byte 13	byte 14	byte 15
byte 16	byte 17	byte 18	byte 19

pointer points to the little end

Data Layout

- **Naive layout strategies generally employed**
  - Place the data in the order the programmer declared it!
- **2 issues: size, alignment**
- **Size – How many bytes is the data item?**
  - Base types have some fixed size
    - E.g., char, int, float, double
  - Composite types (structs, unions, arrays)
    - Overall size is sum of the components (not quite!)
    - Calculate an offset for each field

Memory Alignment

- **Cannot arbitrarily pack variables into memory → Need to worry about <u>alignment</u>**
- **Address of a variable is aligned based on the size of the variable**
  - Char is byte aligned (any addr is fine)
  - Short is halfword aligned (LSB of byte addr must be 0)
  - Int is word aligned (2 LSBs of byte addr must be 0)
  - This rule is for C/C++, other languages may have a slightly different rules

Structure Alignment (for C)

- **Each field is laid out in the order it is declared using Golden Rule for aligning**
- **Identify largest field**
  - Starting address of overall struct is aligned based on the largest field
  - Size of overall struct is a multiple of the largest field
  - Reason for this is so can have an array of structs

Structure Example

- **Struct must start at word-aligned address**

```
struct {                         // Largest field is int (4 bytes)
    char w;                      // Struct size is multiple fo 4
    int x[3]                     // Struct's starting address
    char y;                      // is word aligned
    short z;
}
```

<table>
<tr><td>w</td><td>PAD</td><td>PAD</td><td>PAD</td></tr>
<tr><td colspan="4">x[0]</td></tr>
<tr><td colspan="4">x[1]</td></tr>
<tr><td colspan="4">x[2]</td></tr>
<tr><td>y</td><td>PAD</td><td colspan="2">z</td></tr>
</table>

```
w: 0x7ffd3a916180
x[0]: 0x7ffd3a916184
x[1]: 0x7ffd3a916188
x[2]: 0x7ffd3a91618c
y 0x7ffd3a916190
z: 0x7ffd3a916192
```