­­CS 61C Notes 1

**LEC 1: Intro, Number Representation**

Valgrind for C coding

Reduced Instruction Set Computer (RISC-V)

Hardware design is in Logisim (schematics)

5 Great Ideas in Computer Architecture

1. Abstraction
2. Moore’s Law
3. Principles of Locality
4. Parallelism and Amdahl’s law
5. Dependability via Redundancy

0 or 1 is a bit

Byte is 8 bits

2^10 = 1024

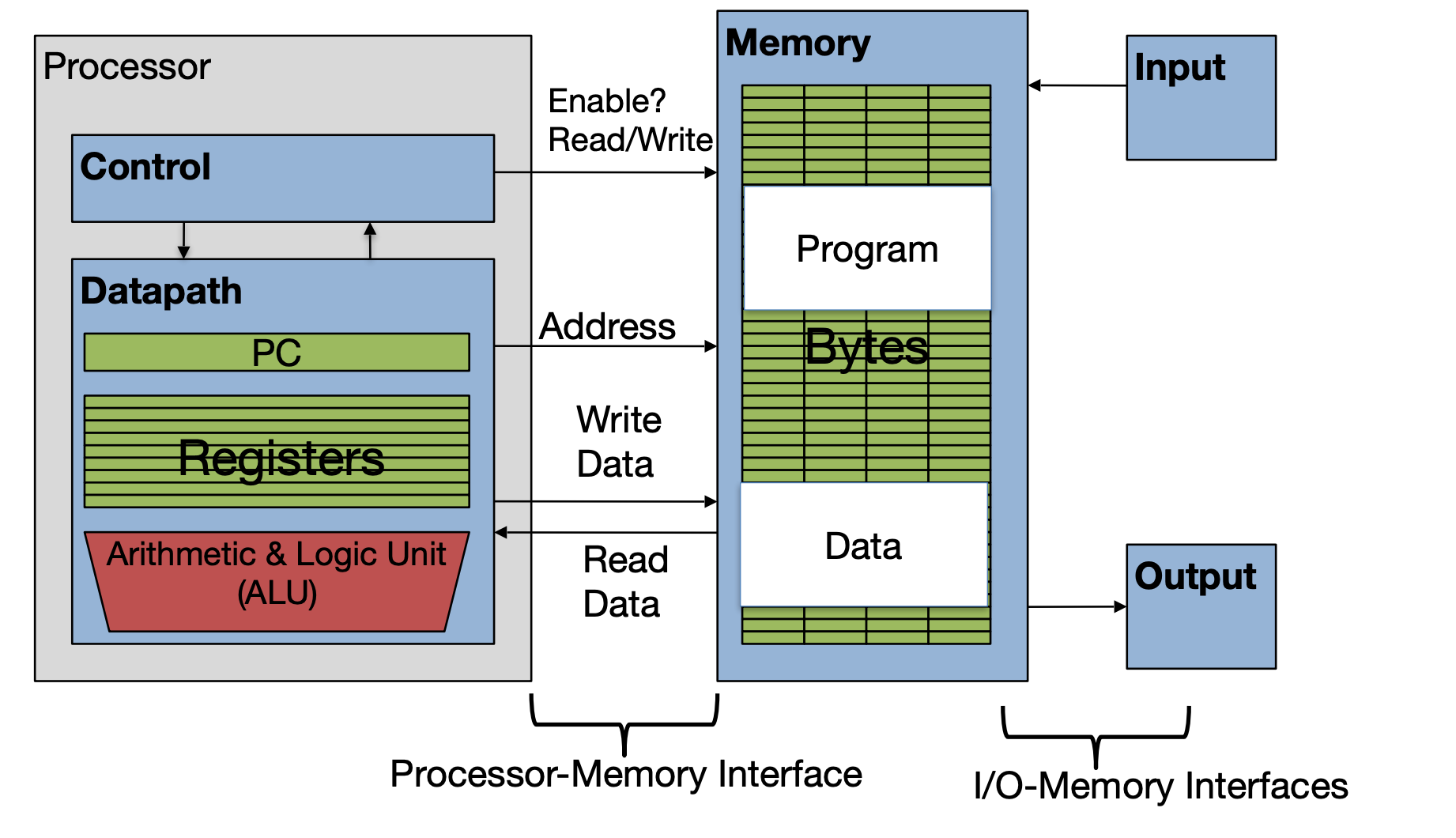
Two’s Complement Representation

* Signed integers in C: want ½ numbers negative and ½ numbers positive
* To **convert** positive from negative or negative to positive **invert the bits and add 1**
* Overflow when magnitude of result too big to fit into result representation
* Can represent **1 more negative than positive number**

**LEC 2: C intro and Pointers**

Know si prefixes

Count in binary and hex on hand

Components of a Computer

Intro to C

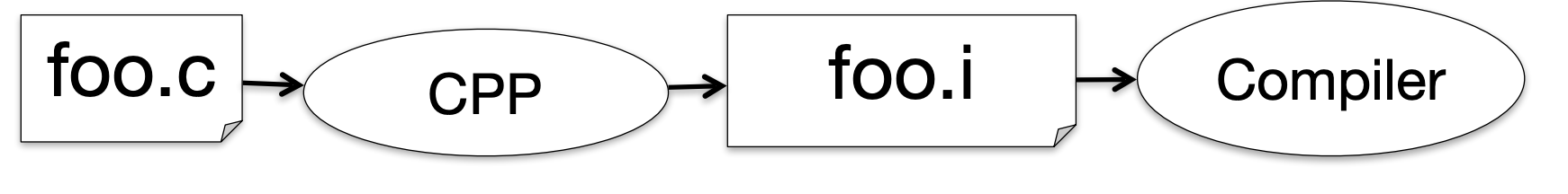
* Allows us to exploit underlying features of the architecture – memory management, special instructions, parallelism
* If starting a new project where performance matters, use either Go (concurrency) or Rust (“C-but-safe”)
* Key C concepts: Pointers, Arrays, Implications for Memory management
  + Key security concept: All of the above are **unsafe**: If your program contains an error in these areas it might not crash immediately but instead leave the program in an inconsistent (and often exploitable) state

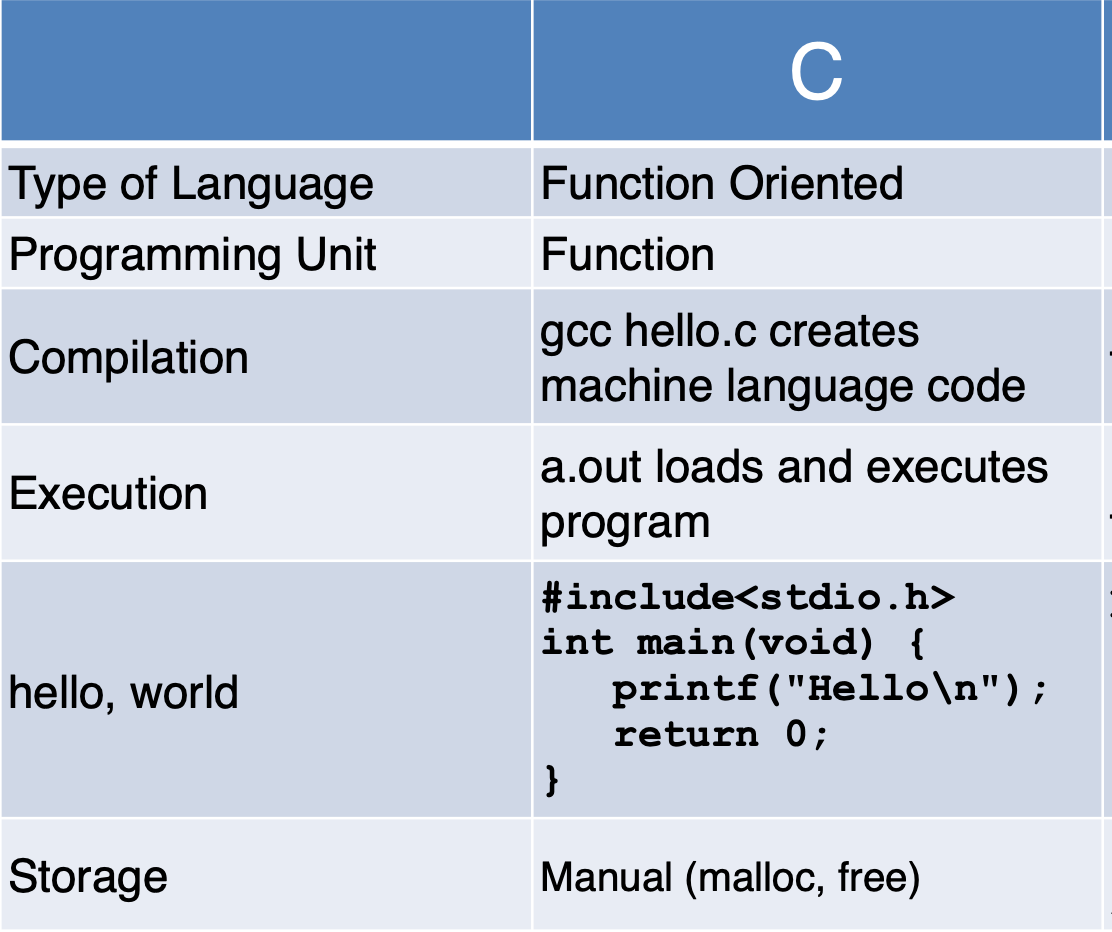
Compilation Overview:

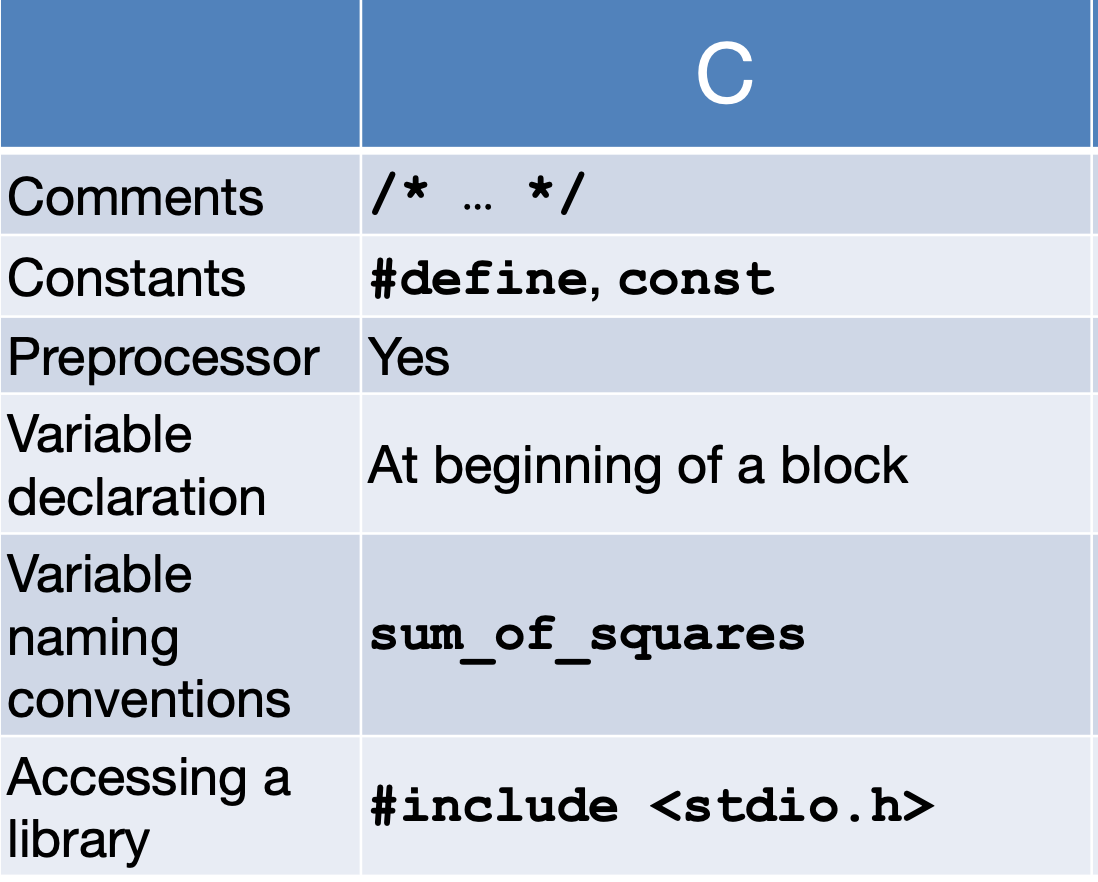
* C compilers map C programs directly into architecture-specific *machine code* (string of 1s and 0s)
* C generally a two part process of compiling .c files to .o files, then linking the .o files into executables
* Advantages of C:
  + Excellent run-time performance (generally much faster than Scheme or Java for comparable code because it optimizes for a given architecture)
    - These days though, a lot of performance is in libraries
  + **Reasonable compilation time**: enhancements in compilation procedure (Makefiles) allow only modified files to be recompiled
* Disadvantages:
  + Compiled files, including the executable, are architecture-specific, depending on processor type (MIPS vs. RISC-V) and the OS (Windows vs Mac)
  + Executable must be rebuilt on each new system (**porting your code to a new architecture**)
  + “Change 🡪 Compile 🡪 Run [repeat]” iteration cycles can be slow during development

How it works:

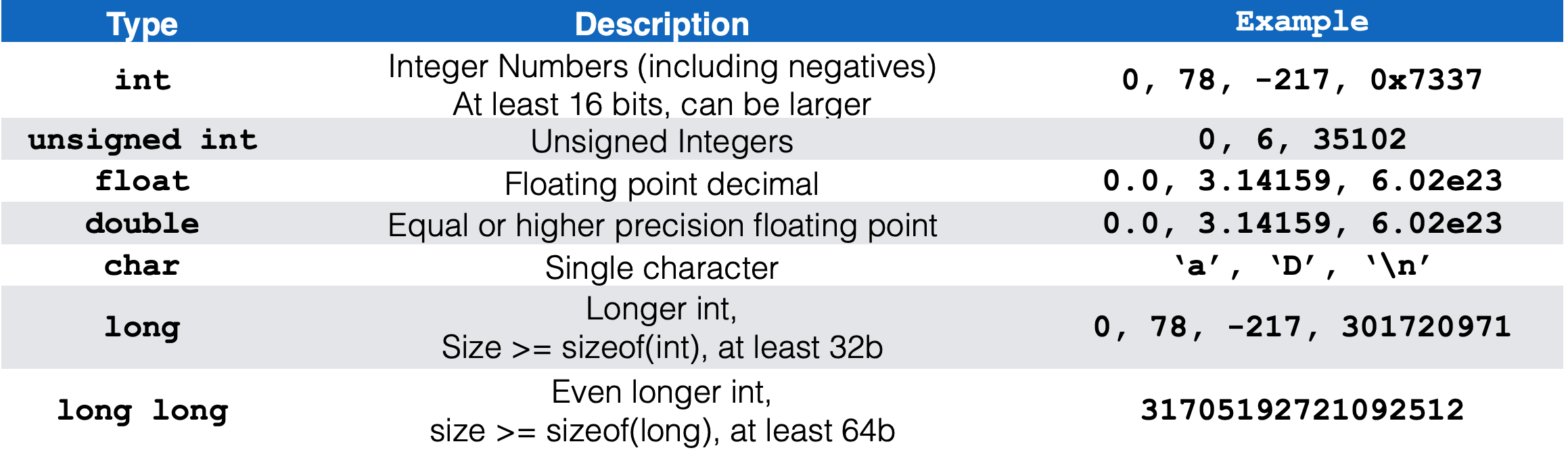
* C source files first pass through macro processor, CPP, before compiler sees code
* CPP replaces comments with a single space
* CPP commands begin with “#”
* #include “file.h” /\* Inserts file.h into output \*/
* #include /\* Looks for file in standard location, but no actual difference! \*/
* #define M\_PI (3.14159) /\* Define constant \*/
* #if/#endif /\* Conditional inclusion of text \*/
* Use –save-temps option to gcc to see result of preprocessing
  + Full documentation at: <http://gcc.gnu.org/onlinedocs/cpp/>







Int representation in C:

* C: int should be integer type that target processor works with most efficiently
* **Only guarantees**:
  + sizeof(long long) ≥ sizeof(long) ≥ sizeof(int) ≥ sizeof(short)
  + short >= 16 bits, long >= 32 bits • All could be 64 bits

Have to declare type of data you plan to return from function

**Structs** are structured groups of variables

All variable declarations must appear before they are used, at beginning of block.

Undefined behavior == un**predictable** behavior

* Often contributes to “heisenbugs” (**unpredictable** bugs)

Syntax:

* If-else, while similar to java
* For (initialize; check; update) statement
* Switch (expression) {

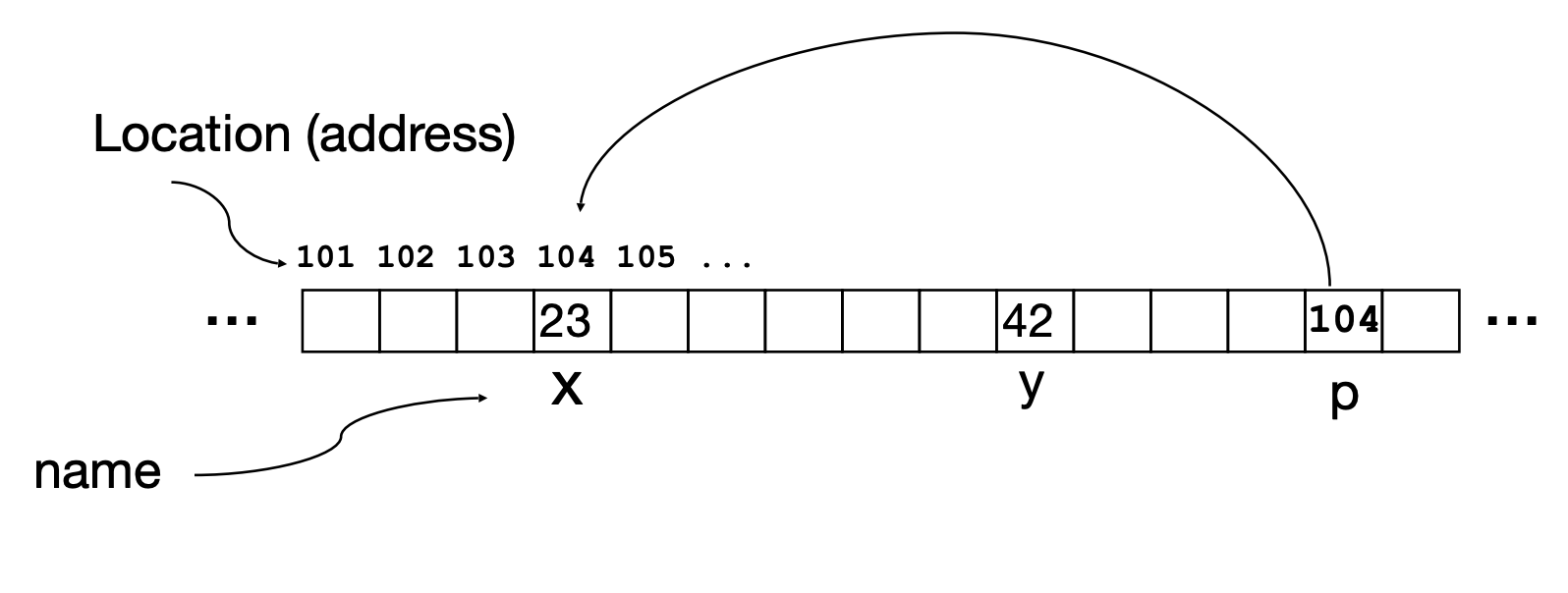
Case const1: statements

Case const2: statements

Default: statements

}

break;



**Pointers and Arrays in C:**

* **Address** is where value is stored
* **Value** is the value
* **Pointer** is a variable that contains the address of a variable

int **\*p**;

* Tells compiler that variable p is address of an int

p = **&y**;

* Tells compiler to assign address of y to p
* **&** called the “address operator” in this context

z = **\*p**;

* Tells compiler to assign value at address in p to z • \* called the “dereference operator” in this context

**LEC 3: Pointers, Array, Memory: Cause of Segfaults**

Consider memory to be a **single** huge array

* Each cell of the array has an address associated with it
* Cell also stores some value
* Address refers to a particular memory location (**pointers** contain address of a variable)

Pointers:

* Used to point to any kind of data (int, char, etc.)
* Normally a pointer only points to one type
* **void \***
  + type that can point to anything (generic pointer)
* You can even have pointers to functions
* Int (\*fn) (void \*, void \*) = &foo
  + Fn is a pointer to a function that accepts two functions void \*, assign that to &foo
* Declaring a pointer just allocates space to hold the pointer – it does not allocate the thing being pointed to
* Local variables in C are not initialized, they may contain anything
* Why use pointers?
  + If we pass large struct or array, its easier/faster to pass a pointer than the whole thing (**pass by reference**)

Modern machines are byte-addressable

* Each has a unique address
* A C pointer is just abstracted memory address
* Type declaration tells compiler how much ytes to fetch on each access through pointer
  + E.g. 32-bit integer stored in 4 consecutive 8-bit bytes

Sizeof(type) operater

* Returns number of bytes in object

Conclusion on Pointers:

* **\*** follows a pointer to its value
* **&** gets the value of the address

**C arrays:**

* **Declaration:**
  + int ar[2]
  + declares a two element integer array (just a block of memory which is uninitialized)
* *Key Concept:*
  + Array variable is simply a pointer to the first (0th) element
    - Ar[0] is the same as \*ar
    - Ar[2] is the same as \*(ar+2)
  + So a[i] = \*(a+i)
* An array is passed to a function as a pointer
  + The array size is lost!
  + Must explicitly pass the size
  + **Its bounds are not checked**
    - We can accidentally access off the end of an array
    - Consequence: we must pass the array **and its size** to any procedure that is going to manipulate it
* Segmentation faults and bus errors:
  + VERY difficult to find
  + Also “fun” to exploit (**security reasons**)
    - Stack overflow exploit (maliciously write off the end of an array on the stack)
    - Heap overflow exploit (maliciously write off the end of an array on the heap)
* **Strings in C are just an array of characters:**
  + char \*c = “hello” == char[] = {‘h’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’}
  + Last character is followed by a 0 byte written as ‘/0’ as a character
  + Consequence: this is how you write a function “length of string:

Int strlen(char s[])

{

int n = 0;

while (s[n] != 0) {

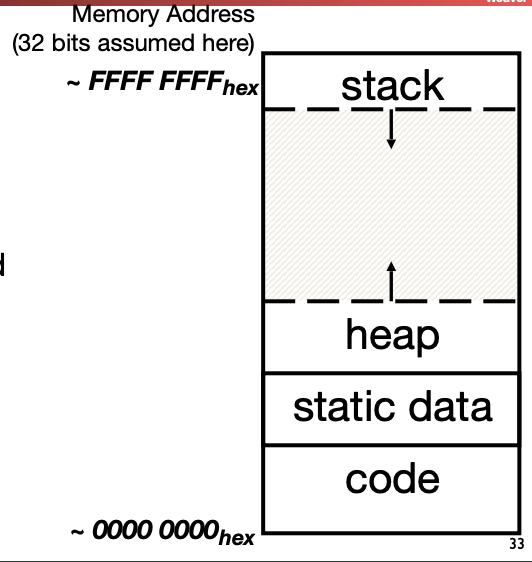
n++;

}

return n;

}

C Memory Management:

Program’s address space contains 4 regions

* Stack (**Local variables inside functions)**
  + main() is considered a function
  + Every time a function is called, a new frame is added to the stack
    - Stack frame includes
      * return address
      * arguments
      * space for local variables
  + LIFO (Last in First Out)
* Heap **(Objects explicitly malloc-ed/free-d)**
  + **void \*malloc(**size\_t n**):** Allocate a block of **uninitialized** memory
    - size\_t is an unsigned integer type enough to count memory bytes
    - n is an integer, indicating size of requested memory
    - returns void \* pointer to block; NULL return indicates no more memory
  + **calloc():** allocate a block of **zeroed** memory
  + **void free(**void \*p**):** **free** previously allocated block of memory
    - **p** is a pointer containing the address originally returned by malloc()
    - When you free memory, you must be sure that you pass the **original** address from malloc() to free()
  + **Realloc():** change size of previously allocated block
    - Careful it might not move
    - **It will not update other pointers pointing to the same block of memory**
  + **Note that if we have this in main():**

int \*baz = malloc(sizeof(int) \* 20);

* + Then \*baz is in the heap and baz is in the stack
* Static Data **(Variables outside functions)**
  + Variables outside functions (global variables)
* Code:
  + functions written in the code

Errors:

1. Memory Leak:
   1. Failure to free allocated memory
   2. Initial symptoms = nothing 🡪 later symptoms = performance drops off a cliff
      1. Memory hierarchy behavior tends to be good just up until the moment it isn’t
2. Returning a pointer into the Stack
   1. Ok to pass a pointer to stack space down
   2. Catastrophically bad to return a pointer to something in the stack
   3. Memory will be overwritten when other functions are called
3. Using a pointer after the free
   1. Writes can corrupt other data
   2. Reads after the free may be corrupted as something takes over that memory
4. Forgetting Realloc Can move Data
   1. When you realloc it can copy data and the result is that it may point
5. Freeing something that malloc never called
6. Freeing something twice

**LEC 4**

Strings:

* Copying strings:
  + strcpy(a, b) or strncpy(a, b, strlen(b) + 1);
  + strcpy doesn't know the length of the destination, so it can be very unsafe
  + strncpy copies only n character for safety, but if it’s too short it will not copy the null terminator!
* Difference between \* and []
  + char \*foo = “this is constant”
    - The compiler interprets it as 'have the constant string "this is constant" somewhere in static memory”
    - Means you cannot write to the string, but can read from it
  + char foo[] = "abc"
    - The compiler interprets this as 'create a 4 character array on the stack, and initialize it to "abc"'
    - Can now read and write from it
    - But now its on the stack, to return it in a function use **malloc**

Pointers to functions:

* You have a function definition
  + char \*foo(char \*a, int b){ … }
* Can create a pointer of that type…
  + char \*(\*f)(char \*, int);
    - Declares f as a function taking a char \* and an int and returning a char \*
* Can assign to it
  + f = &foo
    - Create a reference to function foo
* And can then call it...
  + printf(“%s\n”, (\*f)(“cat”, 3))

Union:

union fubar {

int a;

char \*b;

void c;

} Fubar;

* Accessed just like a struct, but…
  + Fubar \*f = (Fubar \*) malloc(sizeof(union fubar))…

f->a = 1312;

f->b = “baz”

* + F 🡪 b overwrites f.a

Default Alignments rule for the class:

* char: 1 byte, no alignment needed when stored in memory
* short: 2 bytes, 1/2 world aligned • So 0, 2, 4, 6…
* int: 4 bytes, word aligned
* pointers are the same size as ints (4)

Example:

* typedef struct foo\_struct {

int x;

char \*z;

char y;} foo;

* + So how big is a foo?
  + 12... It needs to be padded (4+4+1+padding)
* Say we do f[4].z=”fubar”
  + The address written to in f[4].z = "fubar" is (f + 4 \* 12 + 4)
  + **Note**: This math is the 'under the hood' math: if you actually tried this in C it would not work right! But it is what the compiler produces in the assembly language

Conservative Mark/Sweep Garbage Collectors

* An alternative to malloc & free...
  + malloc works normally, but free just does nothing
* Instead, it starts with the stack & global variables as the "live" memory
  + But it doesn't know if those variables are pointers, integers, or whatevers...
* So assume that every piece of memory in the starting set is a pointer...
  + If it points to something that was allocated by malloc, that entire allocation is now considered live, and "mark it" as live
  + Iterate until there is no more newly discovered live memory
* Now any block of memory that isn't can be deallocated ("sweep

Problems with Mark/Sweep:

* Doesn’t slow down, which is good
* BUT
  + Fragmentation errors:
    - Can’t move memory around, so it gets increasingly fragmented
  + Conservative collector needs to **stop the program!**

Memory Leaks

* Memory leaks are not a problem if your program terminates quickly
* Memory leaks become a much bigger problem when your program keeps running
* Three solutions:
  + Be very diligent about making sure you free all memory
    - Use a tool that helps you find leaked memory
    - Perhaps implement your own reference counter
  + Use a "Conservative Garbage Collector" malloc
  + Just quit and restart your program a lot ("burn down the frat-house")
    - Design your server to crash! But memory leaks will slow down your program long before it actually crashes
* Memory leaks lead to **fragmentation**
  + As a consequence you use more memory, and its more scattered around
* Computers are designed to access **contiguous** memory
* So things that cause your working memory to be spread out more and in smaller pieces slow things down

**LEC 5:**

Assembly Language:

* **Basic job of a CPU (Central Processing Unit):** execute lots of instructions.
* **Instructions:** CPU’s primitive operatons
  + Instructions performed one after another in sequence
  + Each instruction does a small amount of work
  + Each instruction has an operation applied to operands
* CPU’s belong to families, each implementing its own set of instructions
* **Instruction Set Architectures** (ARM, Intel x86, MIPS, RISC-V, etc.)
  + RISC philosophy
    - Keep the instruction set small and simple, makes it easier to build fast hardware
    - Let software do complicated operations by composing simpler ones
* Each assembly language is tied to a particular ISA (its just a human readable version of machine language)
* Why write assembly language?
  + Back then, hand optimized assembly code could beat what the compiler could generate
  + Nowadays ISAs are simple and compilers beat humans

RISC-V

* High-quality, license-free, royalty-free RISC ISA specification
* Appropriate for all levels of computing system, from micro to supercomputers
  + 32 bit, 64 bit, and 128-bit variants

Registers vs Variables:

* Assembly does not have variables only **registers** as operands
* Limited number of special places to hold values, built directly into hardware
* Can only perform arithmetic operations on these in a RISC
  + Only memory actions are loads and stores
* **Benefit:** Since registers are directly in the hardware, **they are very fast to access**
  + **Registers**: 32 words (128 bytes)
  + **Memory** (DRAM) Billions of bytes (2- GB on laptop)
* On a 32-bit machine, there are 32 32-bit registers in RISC-V referred to by number x0–x31
  + Groups of 32 bits called a **word** in RISC-V ISA
* **X0 is special**, always holds **value zero**
  + So really only holds 31 registers able to hold variable numbers
* Registers have **no type**
  + Operation determines how registers are interpreted (str, char, int, etc.)
* Does **not require** that integers be word aligned (but it is very **very bad** if you don’t make sure they are)
  + Consequences of unaligned integers:
    - **Slowdown**: processor is allowed to be a lot slower
    - **Lack of atomicity:** whole thing doesn’t happen at once, can introduce lots of very subtle bugs

Instructions are fixed, 32b long

* Have an **opcode** and **operands**:
  + Add x1, x2, x3 (in RISC-V)
    - Equivalent to: x1 = x2 + x3 (in C)
  + Sub x3, x4, x5 (in RISC-V)
  + a = b + c + d – e;
    - add x1, x2, x3 (temp = b+c)
    - add x1, x1, x4 (temp = temp + d)
    - sub x1, x1, x5 (a = temp-e)
  + f = g
    - add x3, x4, **x0**
* A **No-op** is an instruction that does nothing
  + By convention, **add x0 x0 x0** is the RISC-V no-op instruction
    - Why? 🡪 Writes to x0 are ignored
* Immediates (represented in 12b):
  + immediates are sign extended (two’s complement)
  + Immediates are used to provide numerical constants
  + constants often appear in code, so there are special instructions for them
    - addi x3, x4, –10
      * f = g – 10
* Memory addresses are in bytes
  + Data typically smaller than 32 bits, but rarely smaller than 8 bites (e.g. char type)
  + 8 bits = 1 byte
  + 4 bytes = 1 word
* Transfer from Register to Memory
  + int A[100];

g = h + A[3];

* + using store word (sw)

lw x10, 12 (x13) **#Temp reg x10 gets A[3]**

add x10, x12, x10 **#Temp reg x10 gets h + A[3]**

sw x10, 40 (x13) **#A[10] = h + A[3]**

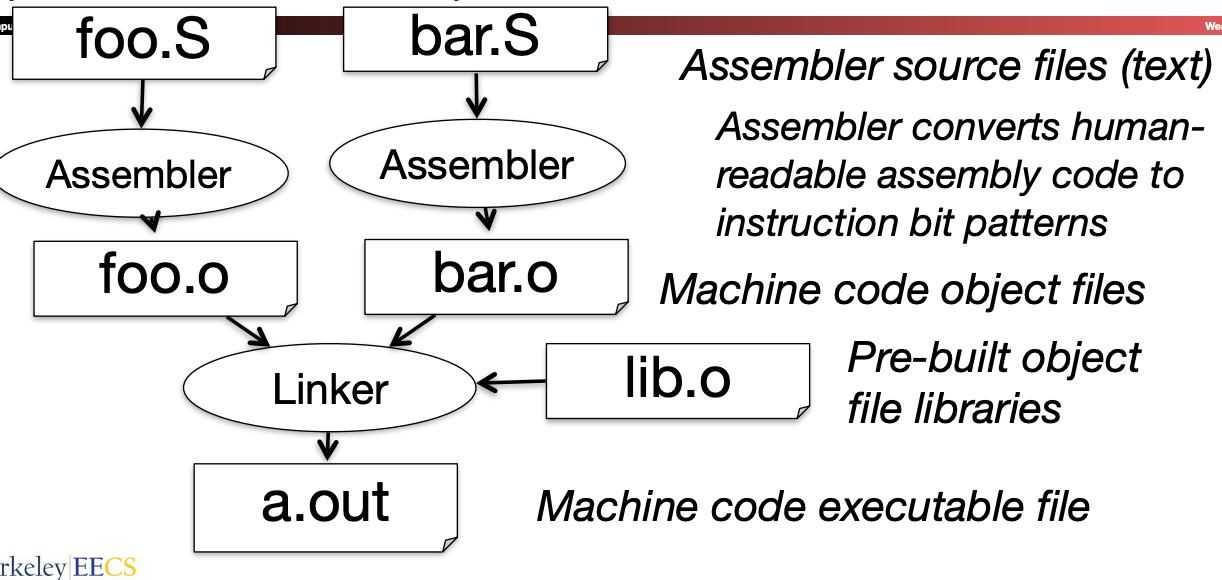
**LEC 6**

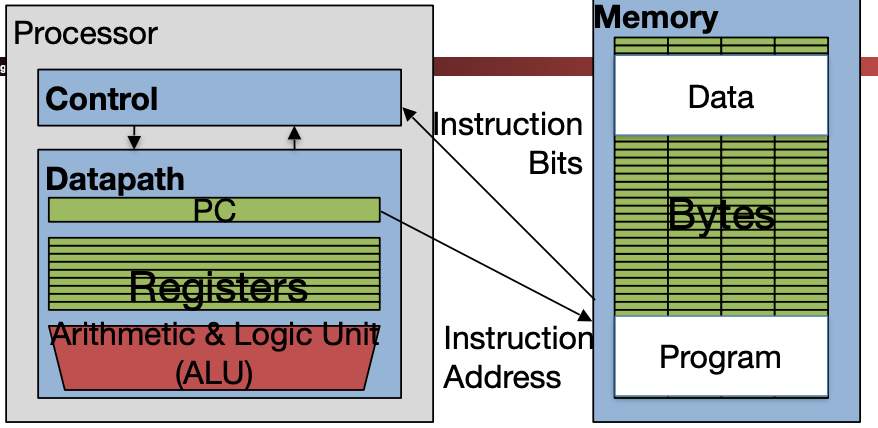
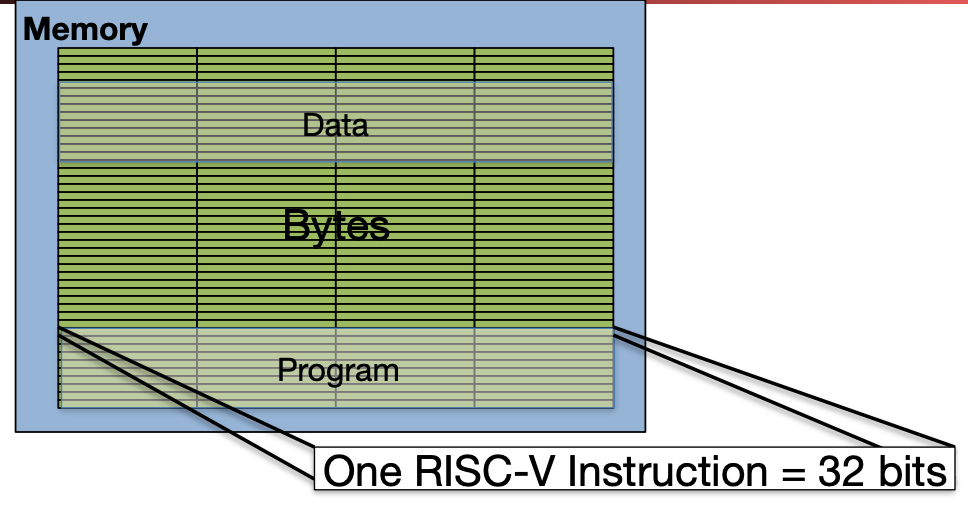
RISC-V Logical Instructions:

* and (&)
* or (|)
* xor (^)
* sll (<<)
* srl (>>)
  + shift right arithmetic (**srai**) moves n bits to the right (**insert** **high-order sign bit** **into empty bits**)

Branching:

* **Branch** – change of control flow
* **Conditional** **Branch** – change control flow depending on outcome of comparison
  + branch if equal (**beq**) or branch if not equal (**bne**)
  + Also branch if less than (**blt**) and branch if greater than or equal (**bge**)
    - syntax: **blt** reg1, reg2, label
    - meaning: if (reg1 < reg2)
* **Unconditional Branch** – always branch
  + a RISC-V instruction for this: **jump (j)**
    - **jal** rd offset
    - **jalr** rd rs (offset)
  + **Jump And Link**
    - Add the immediate value to the current address in the program (the “Program Counter”), go to that location
    - At the same time, store into rd the value of PC+4
      * **j offset == jal x0 offset** (yes, jump is a pseudo-instruction in RISC-V)
    - Two uses:
      * Unconditional jumps in loops and the like
      * Calling other functions

Assembler to Machine Code:

How Programs are stored:

**PC** (Program Counter) is a special register inside processor holding byte address of next instruction to be executed

**Instruction** is fetched from memory, then **control unit** executes instruction using datapath and memory system, and updates program counter (default is add +4 bytes to PC, to move to next sequential instruction)

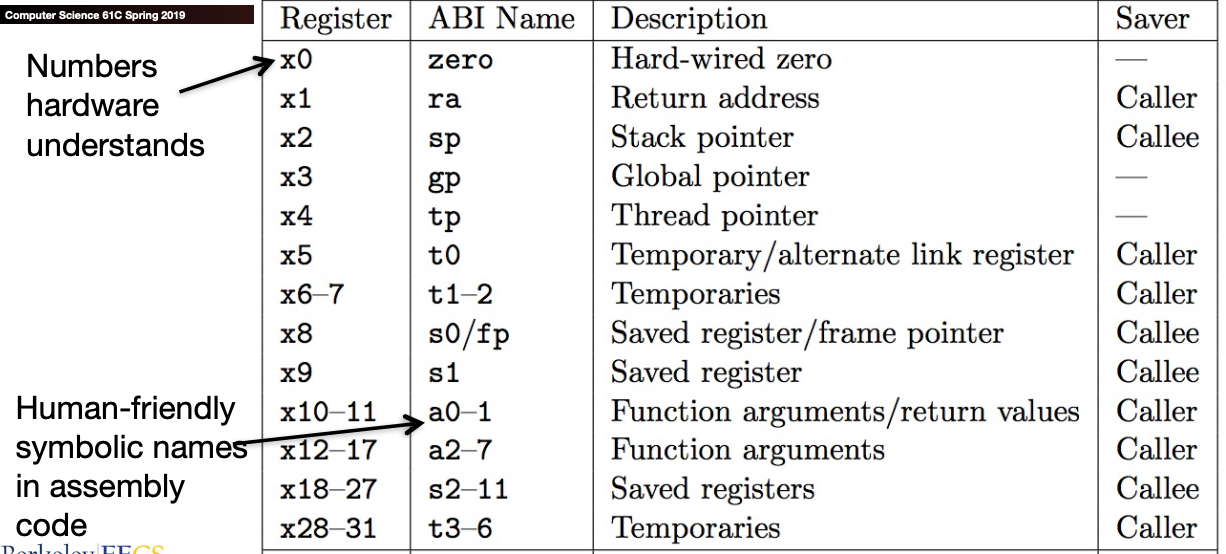
The Six Fundamental Steps in Calling a Function

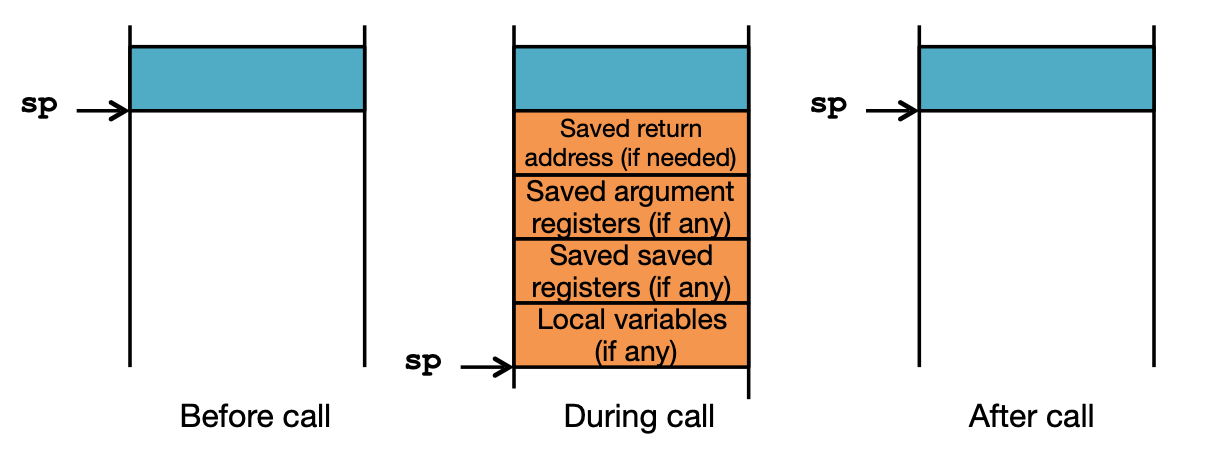
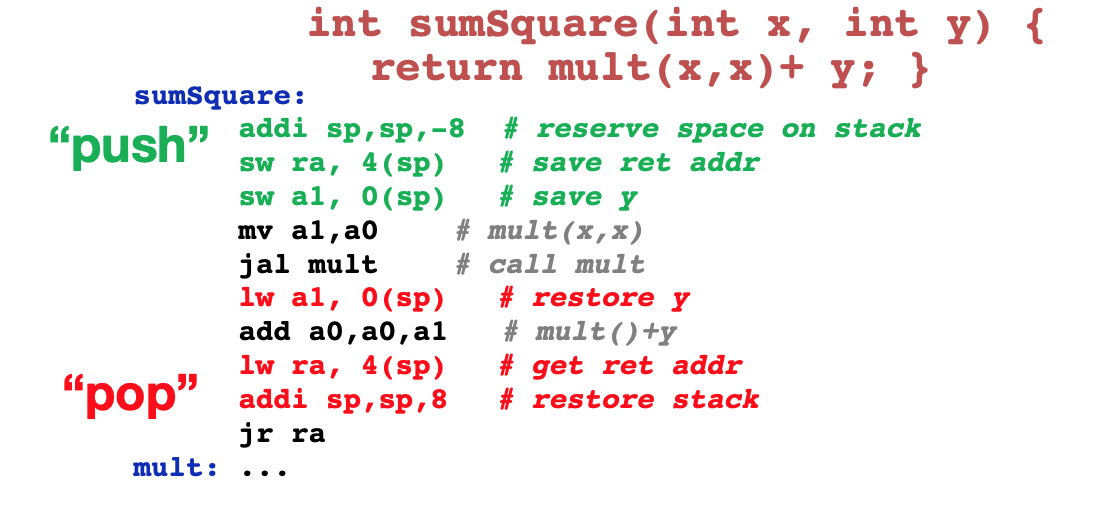
1. Put parameters in a place where function can access them
2. Transfer control to function
3. Acquire (local) storage resources needed for function
4. Perform desired task of the function
5. Put result value in a place where calling code can access it and maybe restore any registers you used
6. Return control to point of origin.
   1. (Note: a function can be called from several points in a program, including from itself.)

Calling Convention:

* Format/usage of registers in a way between the function **caller** and the function **callee**
* Registers are **two** types •
  + **caller-saved** 
    - The function invoked (the callee) can do whatever it wants to them!
  + **callee-saved** 
    - The function invoked must restore them before returning (if used)

RISC-V Symbolic Register Names:



* **a0–a7 (x10-x17**): eight **argument** registers to pass parameters and two return values (a0-a1) (caller saved)
* **ra**: one **return** **address** register for return to the point of origin (x1) **(caller saved)**
* **sp**: pointer to the bottom of the stack **(callee saved)**
  + **sp** is the stack pointer in RISC-V (x2)
  + **sp** always points to the **last used place** on the stack
  + ****Convention is grow stack down from high to low addresses
* **s0-s11** Saved registers: Preserved across function calls **(callee saved)**
* **fp** Frame Pointer: Pointer to the top of the call frame
  + Also is s0, the first saved register, callee saved
  + Frame pointer can often be omitted by the compiler, but we will use it because it makes things clearer how functions are translated.
* **t0-t6** Temporaries: Caller saved

Where is the Stack in Memory:

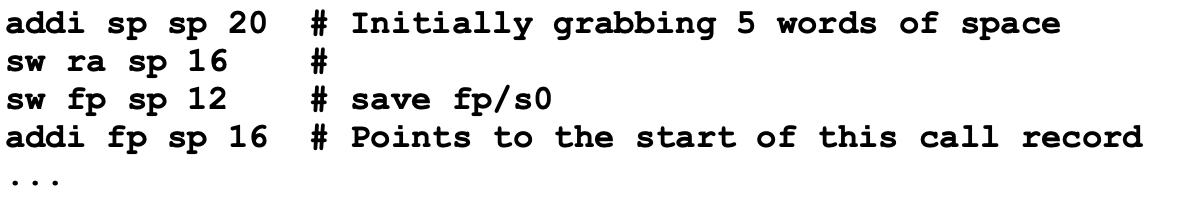
* Stack starts in high memory and grows down
  + Hexadecimal (base 16) : bfff\_fff0hex
* RV32 programs (**text segment**) in low end (where **programs** in memory)
  + 0001\_0000hex
* **Static data segment** (constants and other static variables) above text for static variables
  + RISC-V convention **global pointer (gp**) points to static
  + RV32 gp = 1000\_0000hex
* **Heap** above static for data structures that grow and shrink ; grows up to high addresses

Summary of LEC:

* Functions called with **jal**, return with **jr ra**.
* The stack is your friend: Use it to save anything you need. Just leave it the way you found it!
* Instructions we know so far…
  + Arithmetic: **add, addi, sub**
  + Memory: **lw, sw, lb, lbu, sb**
  + Decision: **beq, bne, blt, bge**
  + Unconditional Branches (Jumps): **j, jal, jr**

**LEC 7: RISC-V Instruction Formats**

Use **S0** as a **Frame Pointer (fp):**

****

**General Stored Program Computer (EDSAC 1949) Consequences:**

1. **Everything has a memory address:**

* Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  + Both branches and jumps use these
* C pointers are just memory addresses: they can point to anything in memory
* One register keeps address of instruction being executed: “Program Counter” (PC)
  + Basically a pointer to memory
  + Intel calls it Instruction Pointer (a better name)

1. **Binary Compatibility:**

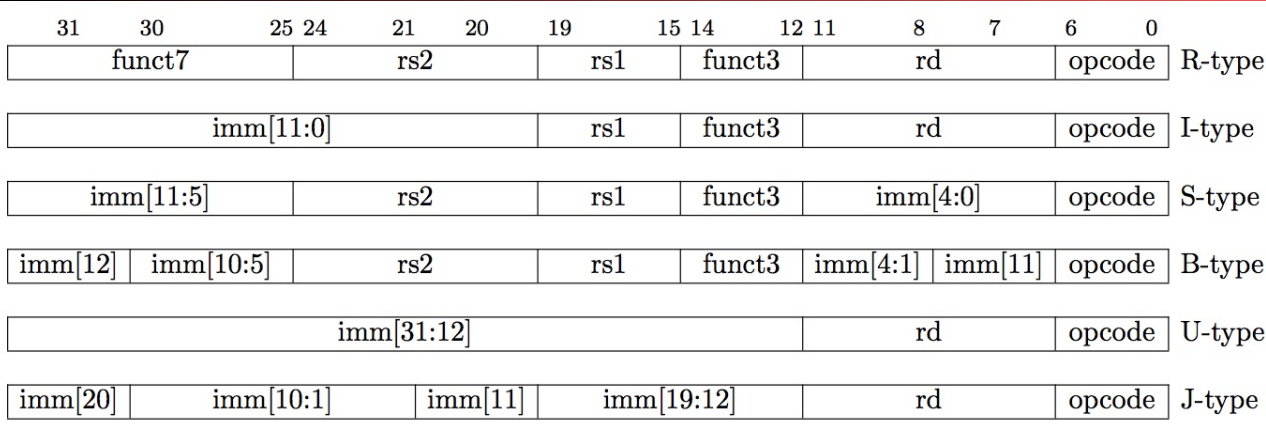
* Programs are distributed in binary form
* New machines in the same family want to run old programs (“binaries”) as well as programs compiled to new instructions
* ­Leads to “backward-compatible” instruction set evolving over time

Instructions as numbers:

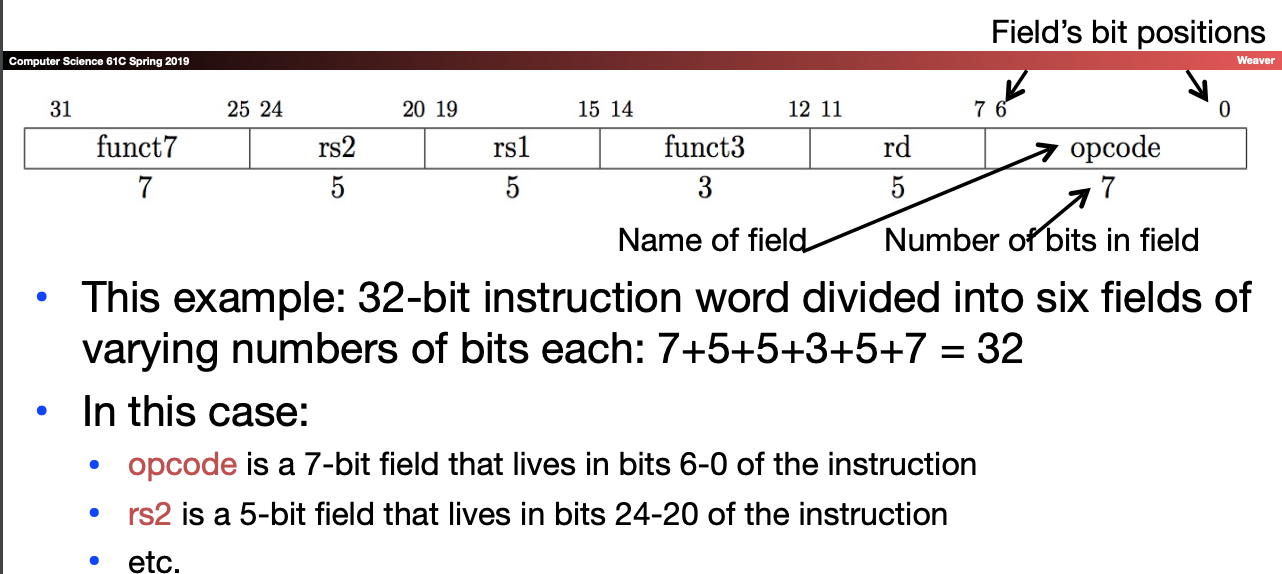
(Remember each register is a word (32 bits), we can represent instructions as:

* Divide 32-bit instruction word into “fields”
* Each field tells processor something about instruction
* We could define different set of fields for each instruction, but RISC-V seeks simplicity, so group possible instructions into six basic types of instruction formats:
  + **R**-**format** for register-register arithmetic/logical operations
  + **I**-**format** for register-immediate ALU operations and loads
  + **S**-**format** for stores
  + **B**-**format** for branches
  + **U**-**format** for 20-bit upper immediate instructions
  + **J**-**format** for jumps

RISC-V Instruction Formats:



Layout:



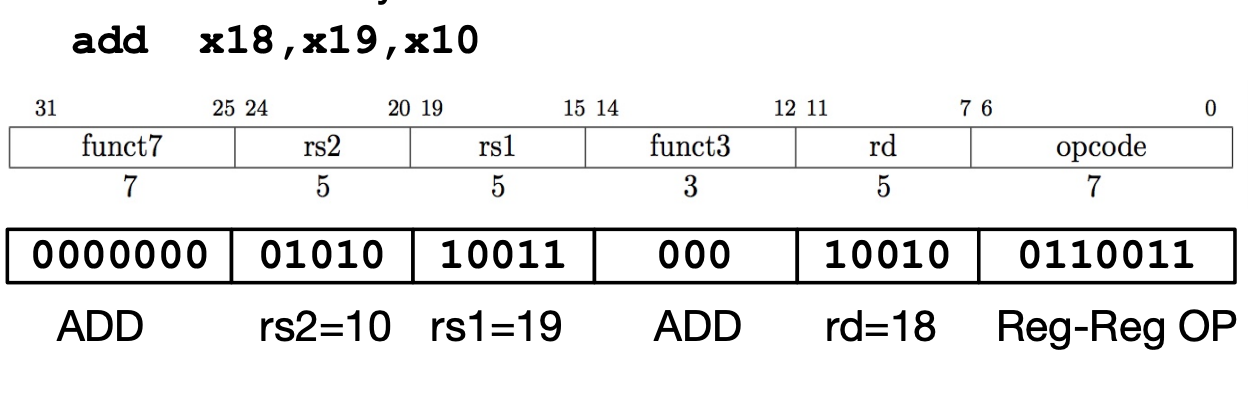
Opcode:

* **opcode**: partially specifies which instruction it is
  + Note: This field is equal to 0110011 for all R-Format register-register arithmetic/logical instructions

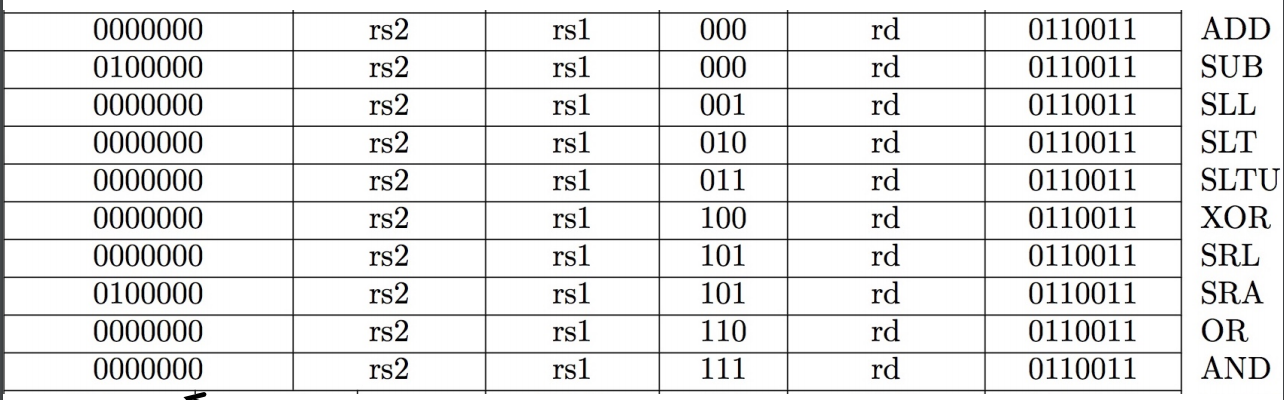
Register:

* Each register field (rs1, rs2, rd) holds a 5-bit unsigned integer (0-31) corresponding to a register number (x0-x31)
  + **rs1 (Source Register #1):** specifies register containing first operand
  + **rs2** : specifies second register operand
  + **rd (Destination Register):** specifies register which will receive result of computation

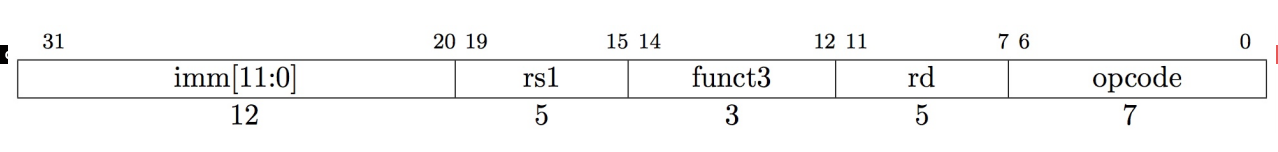
Example:



All RV32 R-format instructions:

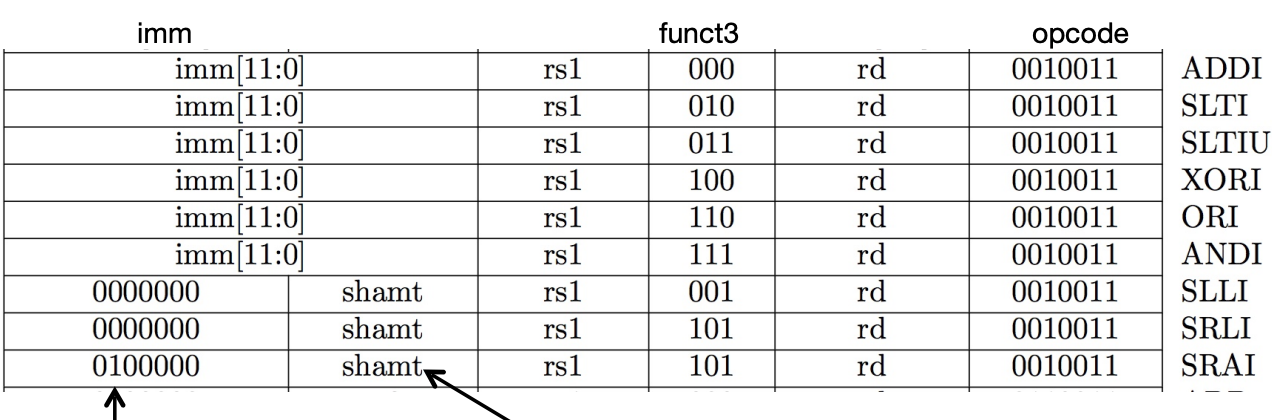


What about immediates (I-Format):



* Immediate is **always** sign-extended to 320bits before use in an arithmetic operation
* imm[11:0] can hold values in range [-2048ten , +2047ten ]
* Everything else same as R-Format

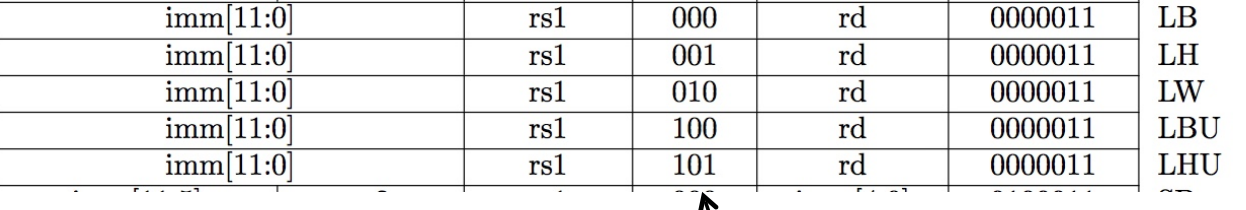
All RV32 I-Format Arithmetic/Logical Instructions:



shift-by-immediate instructions only use 5 bits of the

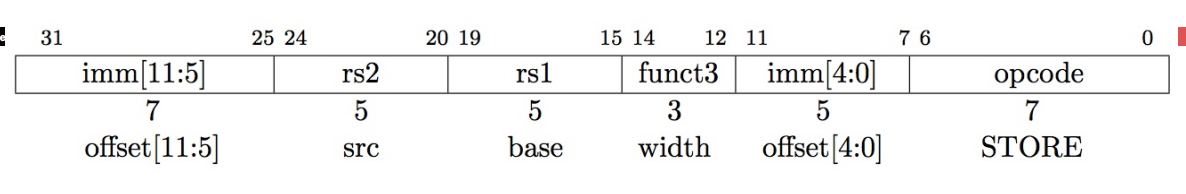
immediate value for shift amt (can only shift by 0-31 pos)

All RV 32 Load Instructions (I-Format):



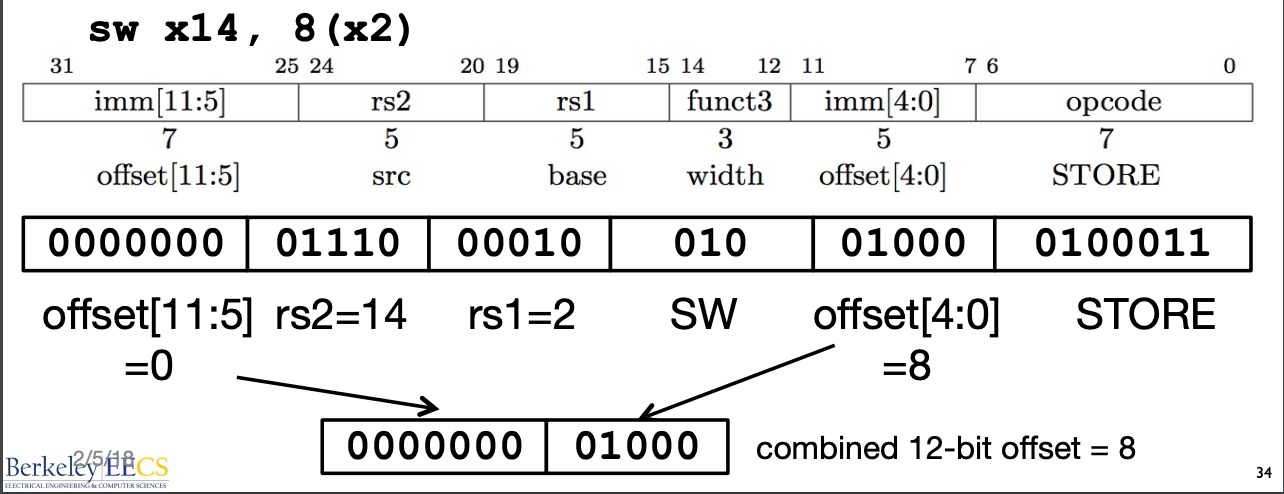
* **LBU** is “load unsigned byte”
* **LH** is “load halfword”, which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
* **LHU** is “load unsigned halfword”, which zero-extends 16 bits to fill destination 32-bit register
* There is no **LWU** in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32-bit register

S-Format Used for Stores:

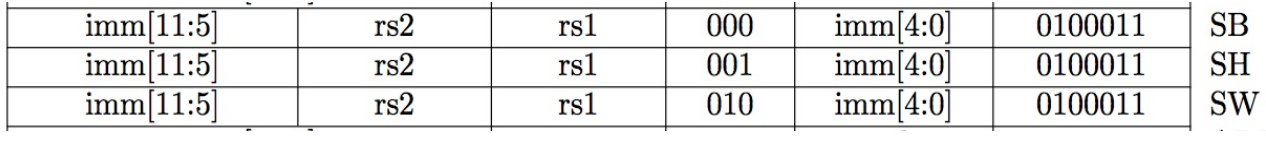


* Store needs to read two registers, rs1 for base memory address, and rs2 for data to be stored, as well as need immediate offset!
* Can’t have both rs2 and immediate in same place as other instructions!
* Note that stores don’t write a value to the register file, no rd!
* RISC-V design decision is move low 5 bits of immediate to where rd field was in other instructions – keep rs1/rs2 fields in same place.

S-Format Example:



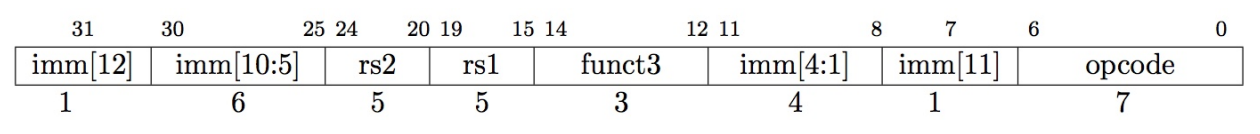
All RV32 Store Instructions:



Branching Instruction Usage (B-type):

* Typically used for loops
  + If we don’t take the branch: PC = PC + 4 (i.e., next instruction)
  + If we do take the branch: PC = PC + immediate\*2
  + **Immediate = number of instructions to jump either forward (+) or backward (-)**

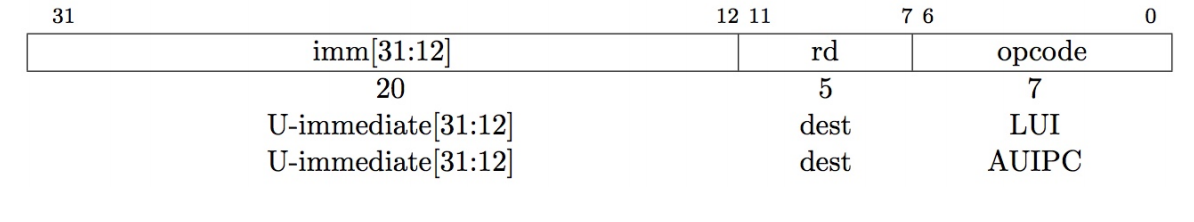
Example:



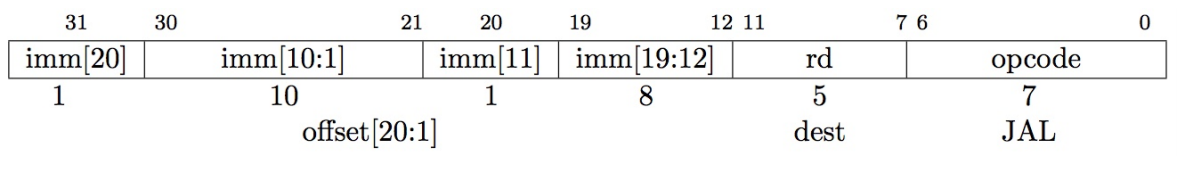
* B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
* But now immediate represents values -4096 to +4094 in 2- byte increments
* The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is always zero, so no need to store it)



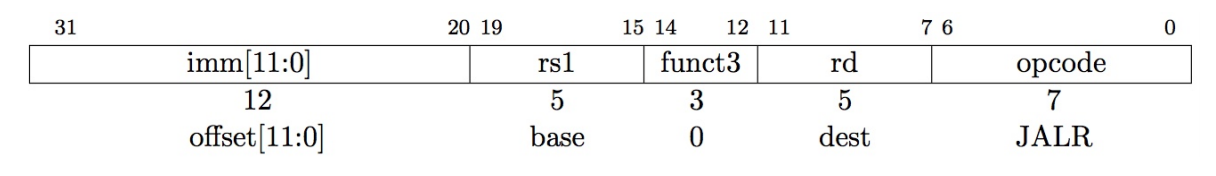
U-Format for “Upper Immediate” instructions



Jump Instructions (J-Format)



JALR Instruction (I-Format)



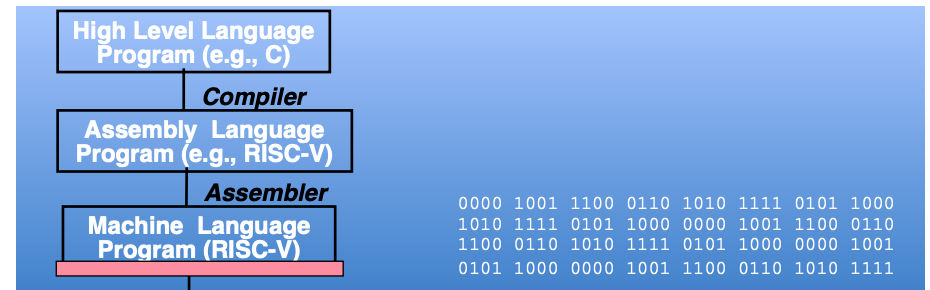
**LEC 8: Compiler, Assembly, Linker, Loader (CALL)**

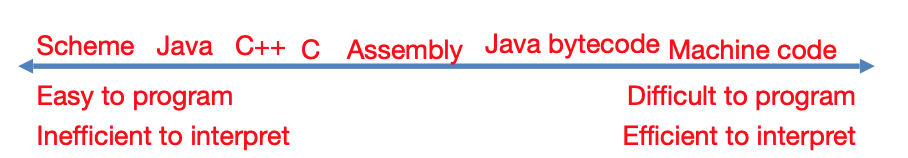
Syntax of Multiplication (signed):

* In RISC-V
  + 32-bit value x 32-bit value = 64 bit value
* **mul** rd, rs1, rs2
  + lower 32 bits of result
* **mulh** rd, rs1, rs2
  + higher 32 bits of result
* Example:
  + in C: **a** = **b** \* **c**;
    - int64\_t **a**;
    - int32\_t **b**, **c**;
    - let **b** be **s2**;
    - let **c** be **s3**;
    - let **a** be **s0** and **s1** (since it **may be up to 64 bits**)
  + mulh s1, s2, s3
  + mul s0, s2, s3

Syntax of Division (signed):

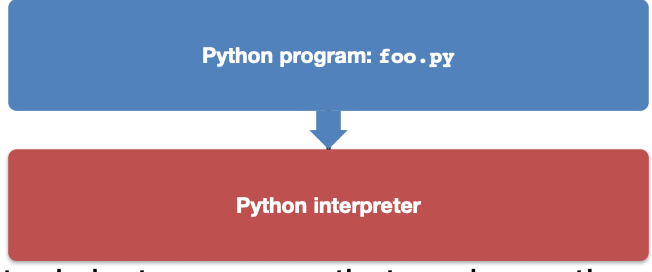
* **div** rd, rs1, rs2
* **rem** rd, rs1, rs2
* Divides 32-bit rs1 by 32-bit rs2, returns the quotient (/) for div, remainder (%) for rem

Interpretation vs. Compilcation:

* An **Interpreter** is a program that executes other programs.
* In general, we **interpret** a high-level language when efficiency is not critical and **translate** to a lower-level language to increase performance

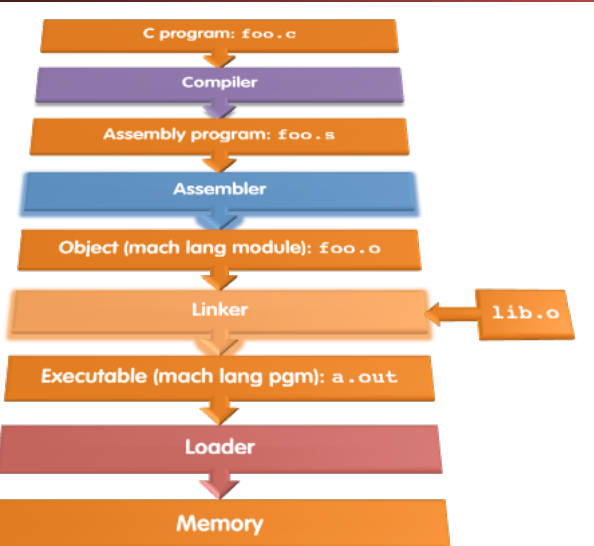
Interpretation vs Translation:

* **Interpreter**: Directly executes a program in the source language
  + Generally easier to write interpreter
  + Closer to high-level, so can give better error messages
  + Provides instruction set independence (run on any machine)
* **Translator**: Converts a program from the source language to an equivalent program in another language
  + almost always more efficient (higher performance)
  + does the hard work once (during compilation)
  + Important for many applications, particularly OS
  + This is why interpreters these days are really “just in time compilers”, don’t throw away the work processing the program

Interpretation Example:

* Python interpreter is just a program that reads a python program and performs the functions of that python program
  + Interpreter converts to a simple bytecode that the interpreter runs…
  + Saved copies end up in .pyc files

The CALL chain:



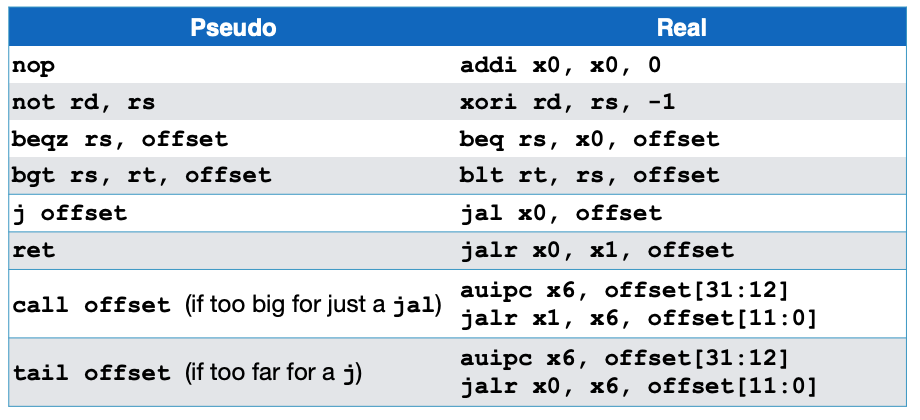
* **Input**:
  + High-Level Language Code (e.g., foo.c)
* **Output**:
  + Assembly Language Code (e.g. MAL) (e.g., foo.s for RISC-V)
* Reads and uses **Directives**
* Replace Pseudo-Instructions
* Produce Machine Language
* Creates **Object File**

Steps in the Compiler:

* **Lexer**:
  + Turns the input into "tokens", recognizes problems with the tokens
* **Parser**:
  + Turns the tokens into an "Abstract Syntax Tree", recognizes problems in the program structure
* **Semantic Analysis and Optimization:** 
  + Checks for semantic errors, may reorganize the code to make it better
* **Code generation:** 
  + Output the assembly code

Assembler Directives:

* Give directions to assembler, but do not produce machine instructions
  + **.text:** Subsequent items put in user text segment (machine code)
  + **.data**: Subsequent items put in user data segment (binary rep of data in source file)
  + **.globl sym:** declares sym global and can be referenced from other files
  + **.string str:** Store the string str in memory and null-terminate it
  + **.word w1…wn:** Store the n 32-bit quantities in successive memory words

Pseudo-Instruction Replacement

* Assembly treats convenient variations of machine language instructions as if they were real instructions

So what is tail?

* Often times your code has a convention like this:
* { ...

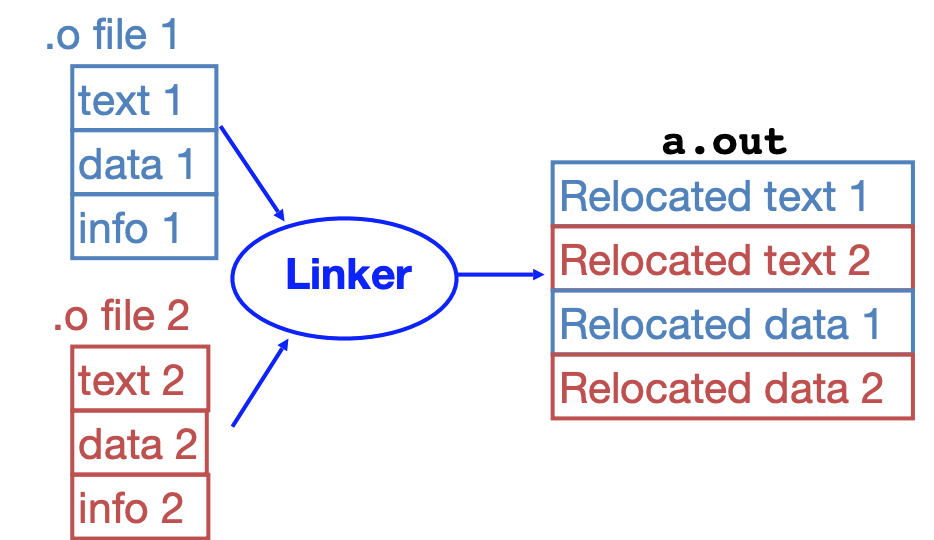
lots of code return foo(y);

}

* It can be a recursive call to foo() if this is within foo(), or call to a different function...
* a
* So for efficiency...
  + Evaluate the arguments for foo() and place them in a0-a7...
  + Restore ra
  + Restore the stack and all callee saved registers
  + Then call foo() with **j or tail**
* Then when foo() returns, it can return directly to where it needs to return to • Rather than returning to wherever foo() was called and returning from there

Producing Machine Language:

* Branching:
  + **Forward Reference”** problem
    - Branch instructions can refer to labels that are “forward” in the program:
  + Solved by taking 2 passes over the program
    - **First pass** remembers position of labels
    - **Second pass** uses label positions to generate code
* Jumps:
  + Jumps within a file are PC relative (and we can easily compute)
  + Jumps to **other** files we can’t
* References to static data
  + la gets broken up into lui and addi
  + These will require the full 32-bit address of the data
  + **Symbol Table:**
    - List of “items” in this file that may be used by other files
    - **Labels**: function calling
    - **Data**: anything in the **.data** section; variables which may be accessed across files
  + **Relocation Table**
    - List of “items” this file needs the address of late
    - Any external label jumped to: **jal**
      * external (including lib files)
    - Any piece of data in static section
      * such as the **la** instruction
* Object File Format:
  + **object file header**:
    - size and position of the other pieces of the object file
  + **text segment**:
    - the machine code
  + **data segment**:
    - binary representation of the static data in the source file
  + **relocation information**:
    - identifies lines of code that need to be fixed up later
  + **symbol table**:
    - list of this file’s labels and static data that can be referenced
  + **debugging information**
  + A standard format is **ELF** (except Microsoft)

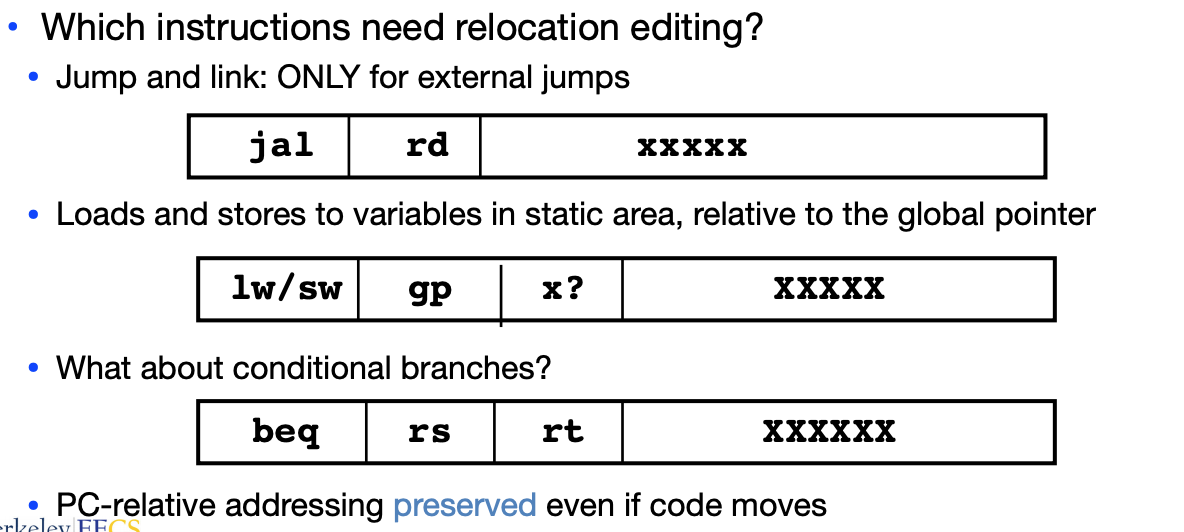
Linker:

* **Input**: Object code files, information tables (e.g., foo.o,libc.o)
* **Output**: Executable code (e.g., a.out)
* Combines several object (.o) files into a single executable (“linking”)
* Enables separate compilation of files
  + Allows parallelization of compilers, but linker is sequential
  + Changes to one file do not require recompilation of whole program

Linker Steps

* **Step 1:** Take text segment from each .o file and put them together
* **Step 2:** Take data segment from each .o file, put them together, and concatenate this onto end of text segments
* **Step 3:** Resolve references
  + Go through Relocation Table; handle each entry
  + That is, fill in all absolute addresses

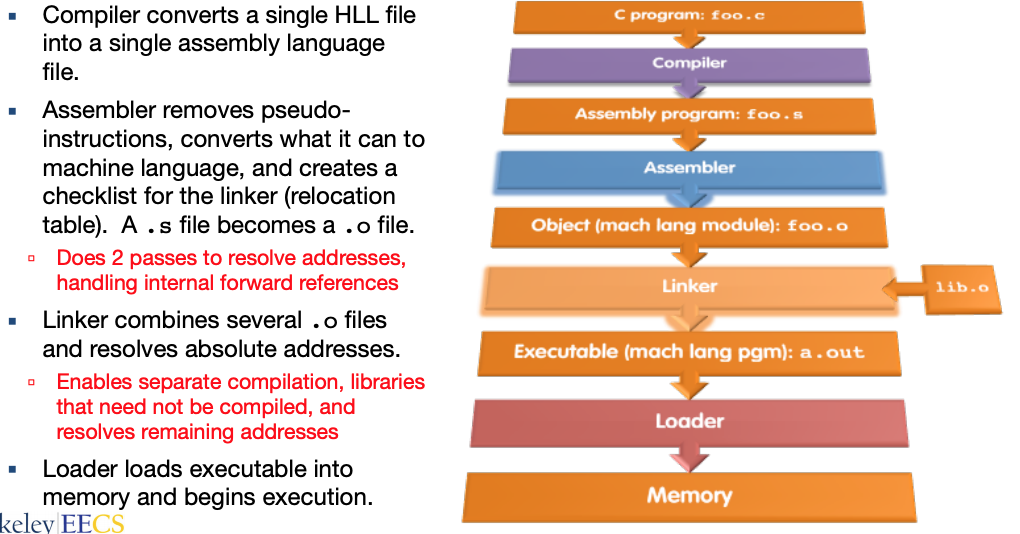
Three Types of Addressing:

* PC-Relative Addressing (beq, bne, jal)
  + never relocate
* External Function Reference (usually jal)
  + always relocate
* Static Data Reference (often auipc and addi)
  + always relocate
  + RISC-V often uses auipc rather than lui so that a big block of stuff can be further relocated as long as it is fixed relative to the pc

Resolving References:

* Linker assumes first word of first text segment is at address 0x04000000.
  + (More later when we study “virtual memory”)
* Linker knows:
  + length of each text and data segment
  + ordering of text and data segments
* Linker calculates:
  + absolute address of each label to be jumped to and each piece of data being referenced
* To resolve references:
  + search for reference (data or label) in all “user” symbol tables
  + if not found, search library files (for example, for printf)
  + once absolute address is determined, fill in the machine code appropriately
* **Output of linker**: executable file containing text and data (plus header)

Summary:



Loader Basics:

* Input: Executable Code (e.g., a.out)
* Output: (program is run) • Executable files are stored on disk
* When one is run, loader’s job is to load it into memory and start it running
* **In reality, loader is the operating system (OS)**
  + loading is one of the OS tasks
  + And these days, the loader actually does a lot of the linking
* Reads executable file’s header to determine size of text and data segments
* Creates new address space for program large enough to hold text and data segments, along with a stack segment
* Copies instructions and data from executable file into the new address space
* Copies arguments passed to the program onto the stack
* Initializes machine registers
  + Most registers cleared, but stack pointer assigned address of 1st free stack location
* Jumps to start-up routine that copies program’s arguments from stack to registers & sets the PC

Static vs. Dynamically Linked Libraries:

* **Statically-Linked Approach**
  + Library is now part of the executable, so if the library updates, we don’t get the fix (have to recompile if we have source)
  + Includes the entire library even if not all of it will be used
  + Executable is self-contained
* **Dynamically-Linked Libraries** 
  + **Space/time issues** 
    - + Storing a program requires less disk space
    - + Sending a program requires less time
    - + Executing two programs requires less memory (if they share a library)
    - – At runtime, there’s time overhead to do link
  + **Upgrades** 
    - + Replacing one file (libXYZ.so) upgrades every program that uses library “XYZ”
    - – Having the executable isn’t enough anymore