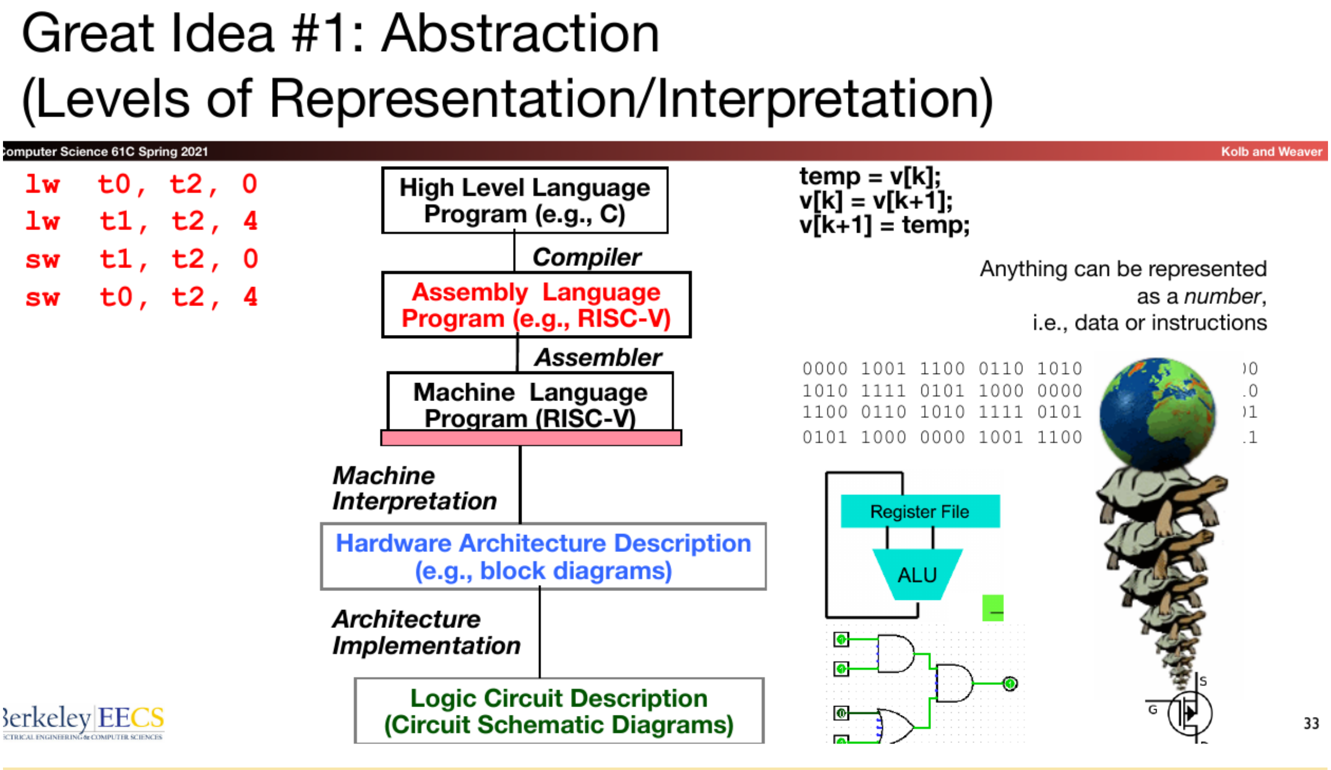
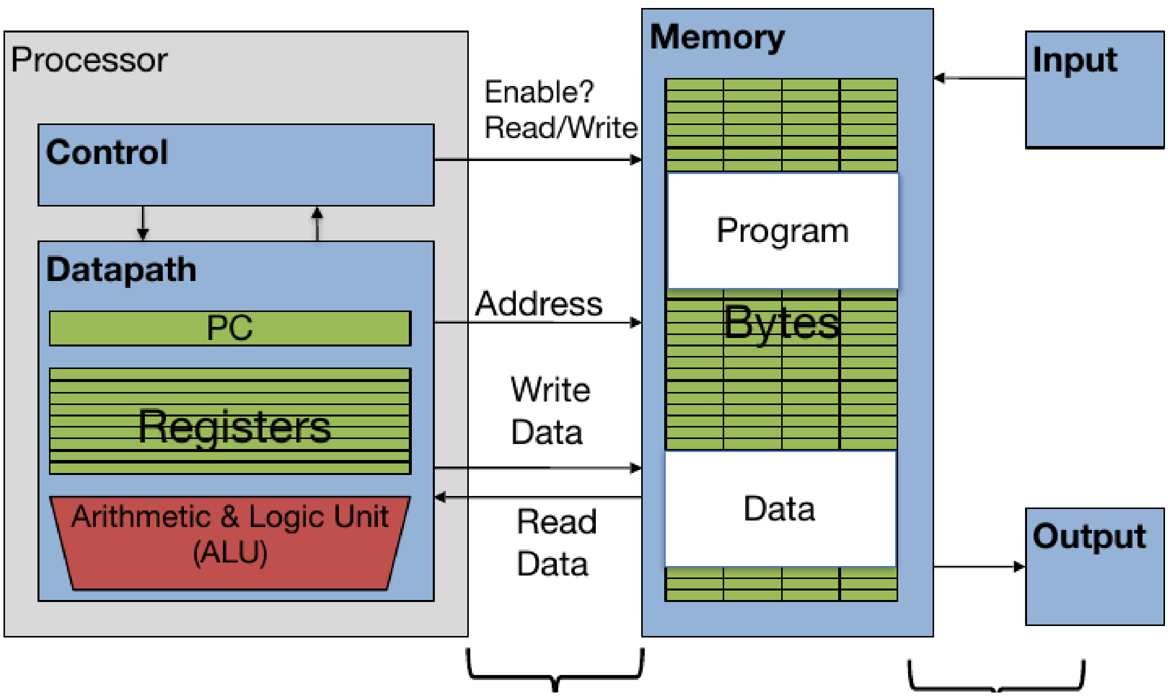
How Program Run, Big Overview

Whole View

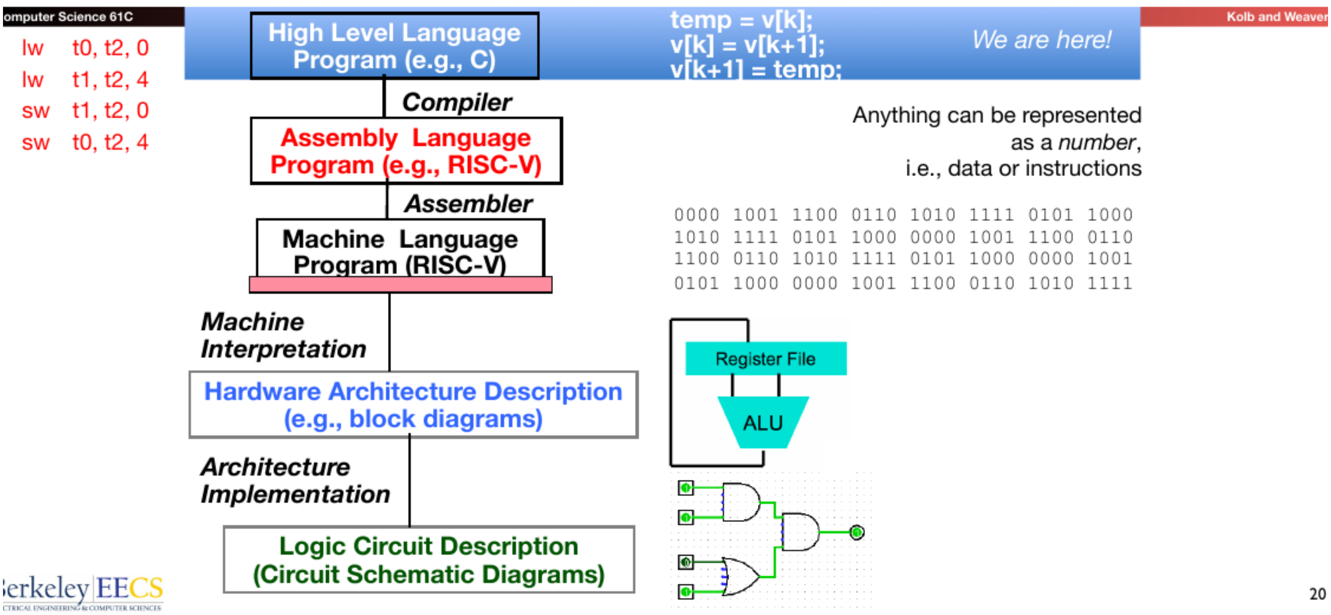


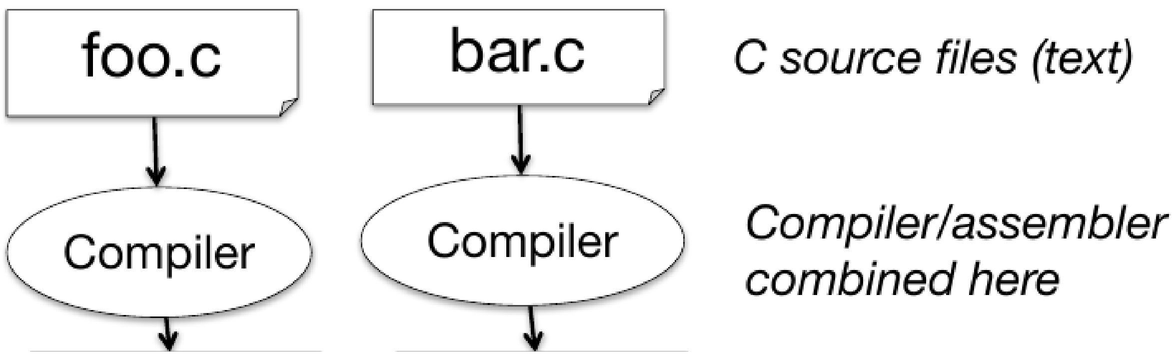
High Level Language

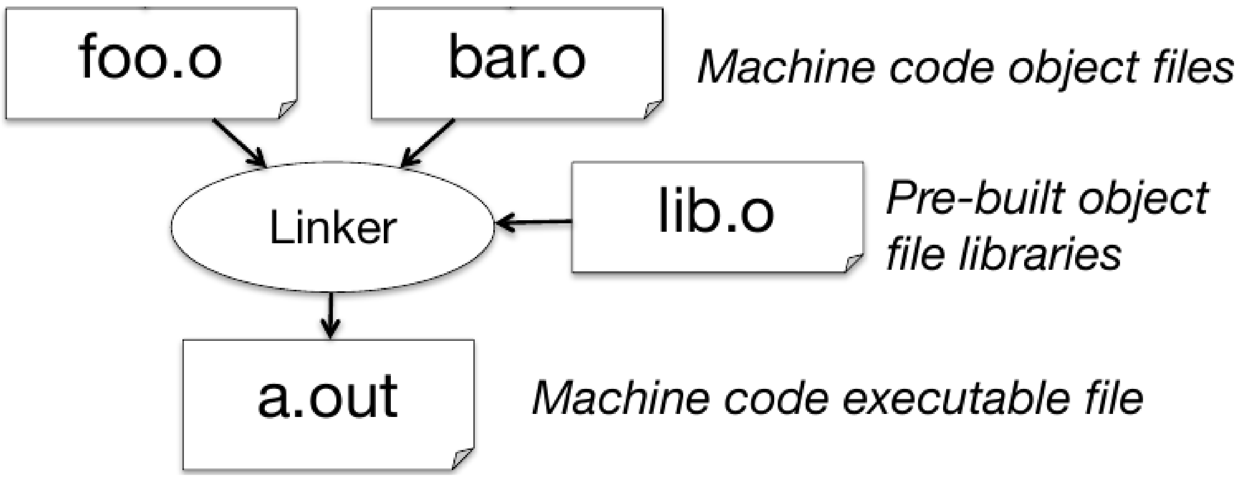


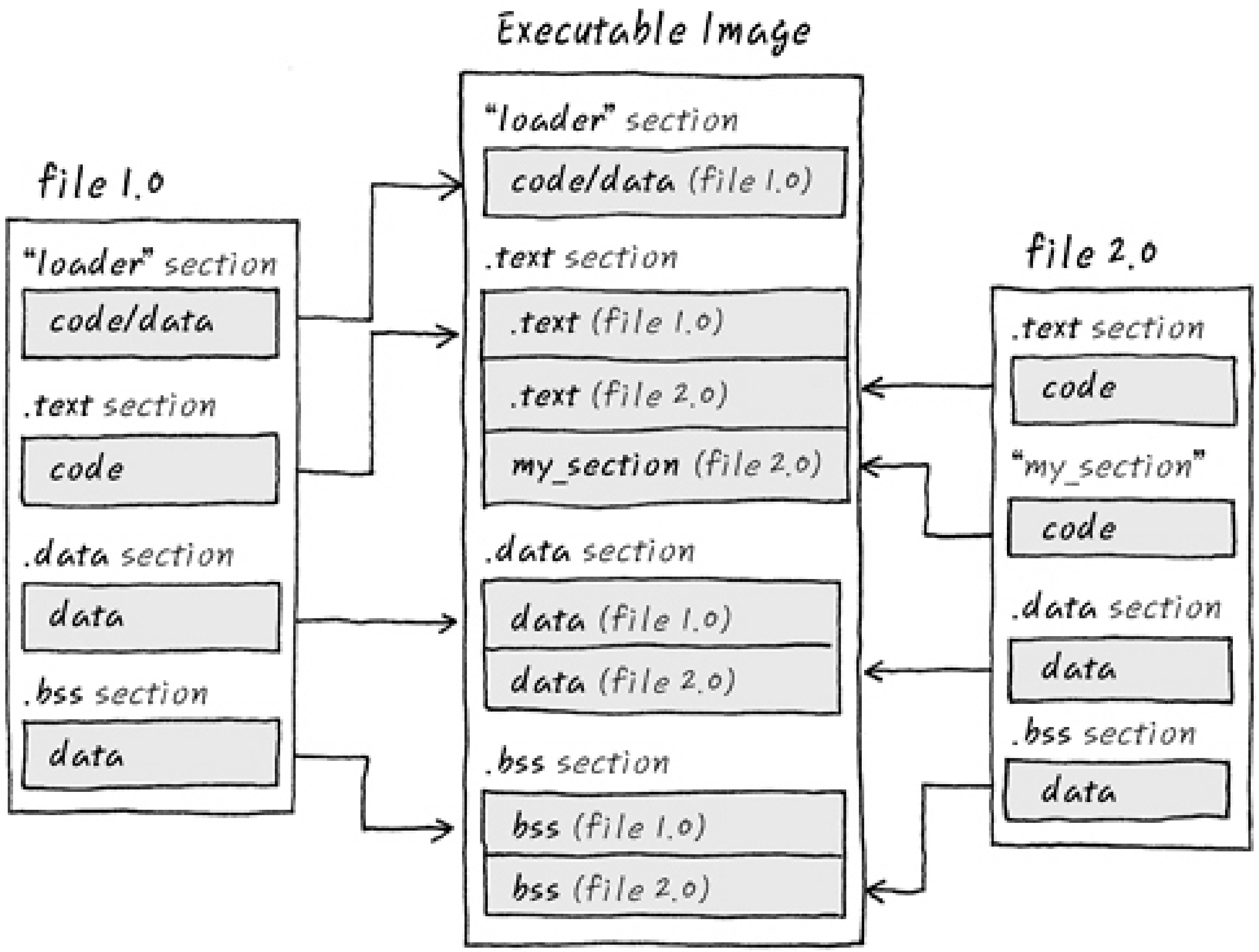
Processor-Memory Interface I/O-Memory Interfaces

Great Idea: Levels of Representation/Interpretation









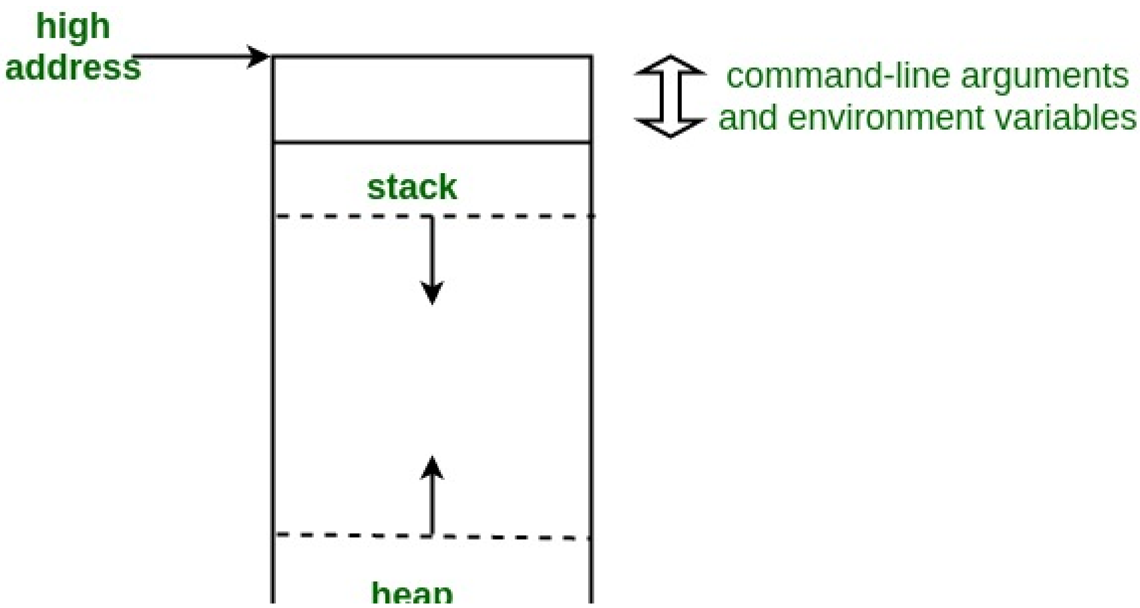
- When C program starts
  - C executable **`a.out`** is loaded into memory by operating system (OS)

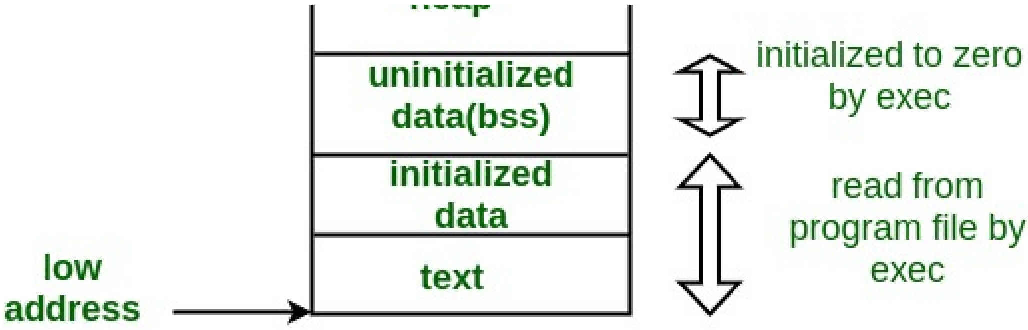
- OS sets up stack, then calls into C runtime library,
- Runtime first initializes memory and other libraries,
- then calls your procedure named **`main()`**

Memory look logically

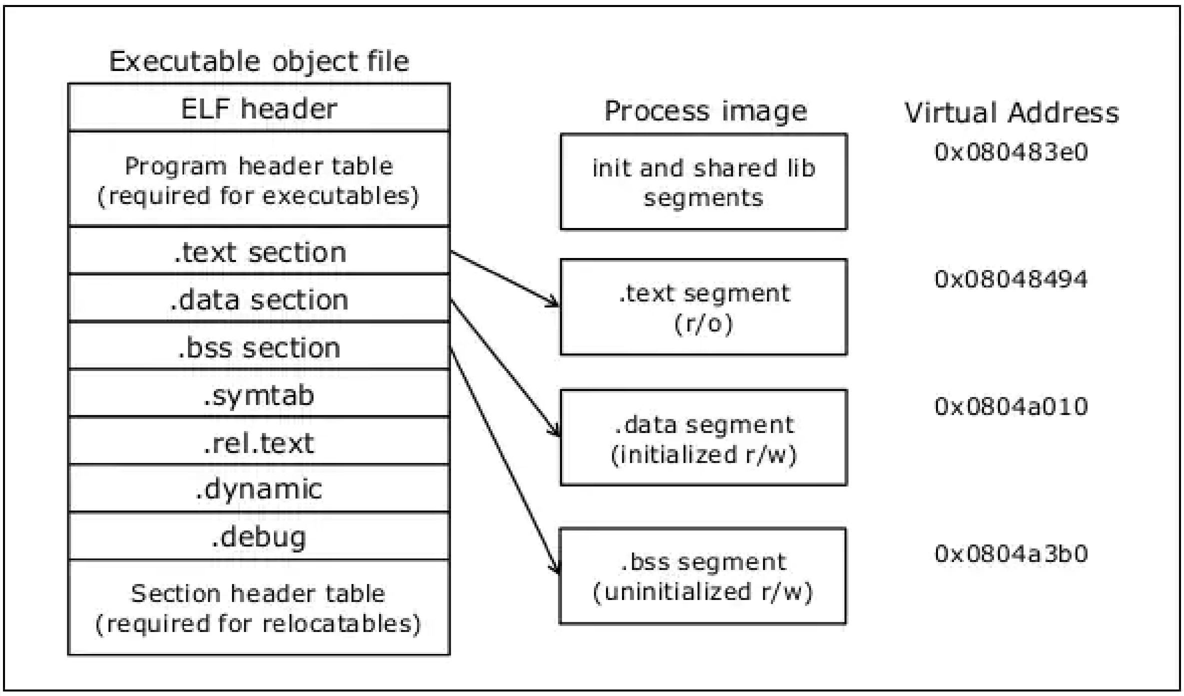
0xFFFFFFFC	xxxx	xxxx	xxxx	xxxx
0xFFFFFFF8	xxxx	xxxx	xxxx	xxxx
0xFFFFFFF4	xxxx	xxxx	xxxx	xxxx
0xFFFFFFF0	xxxx	xxxx	xxxx	xxxx
0xFFFFFFEC	xxxx	xxxx	xxxx	xxxx
...	...	...	...	...
0x14	xxxx	xxxx	xxxx	xxxx
0x10	xxxx	xxxx	xxxx	xxxx
0x0C	xxxx	xxxx	xxxx	xxxx
0x08	xxxx	xxxx	xxxx	xxxx
0x04	xxxx	xxxx	xxxx	xxxx
0x00	xxxx	xxxx	xxxx	xxxx

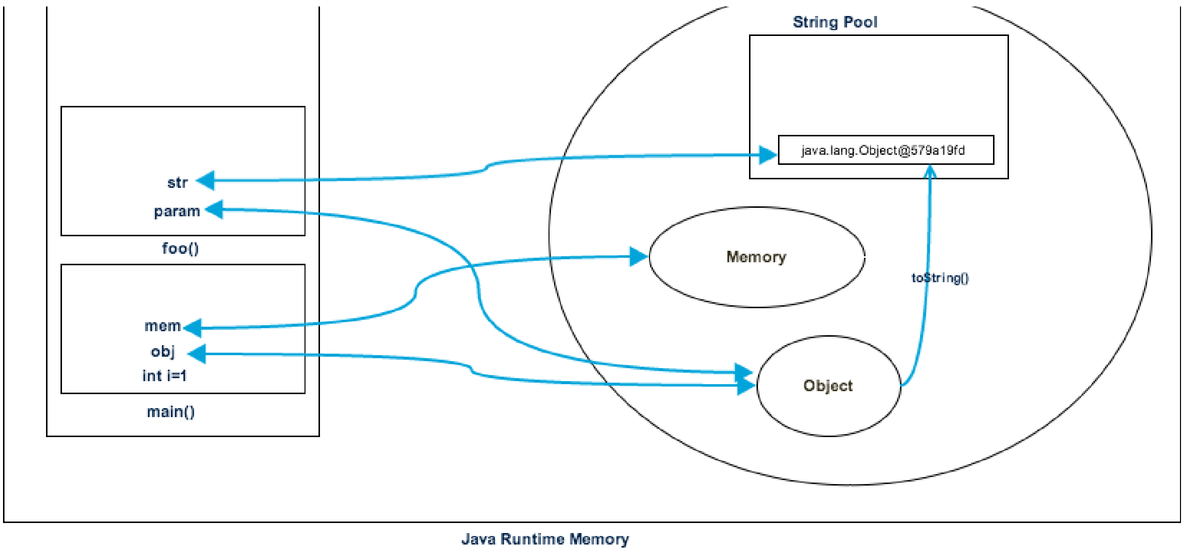
Process Memory Layout



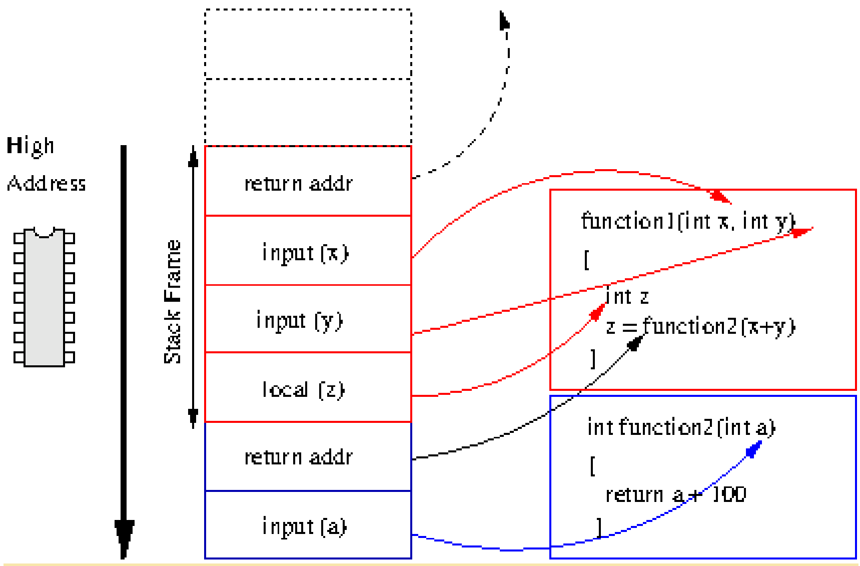


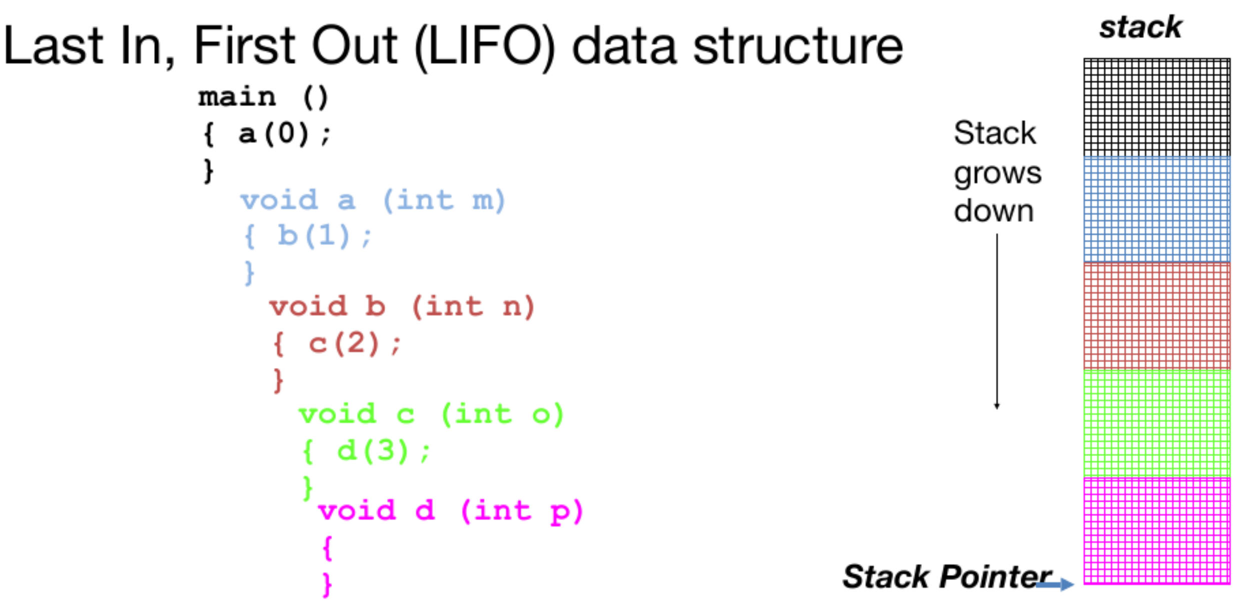
ELF Parsing by Dynamic Linker

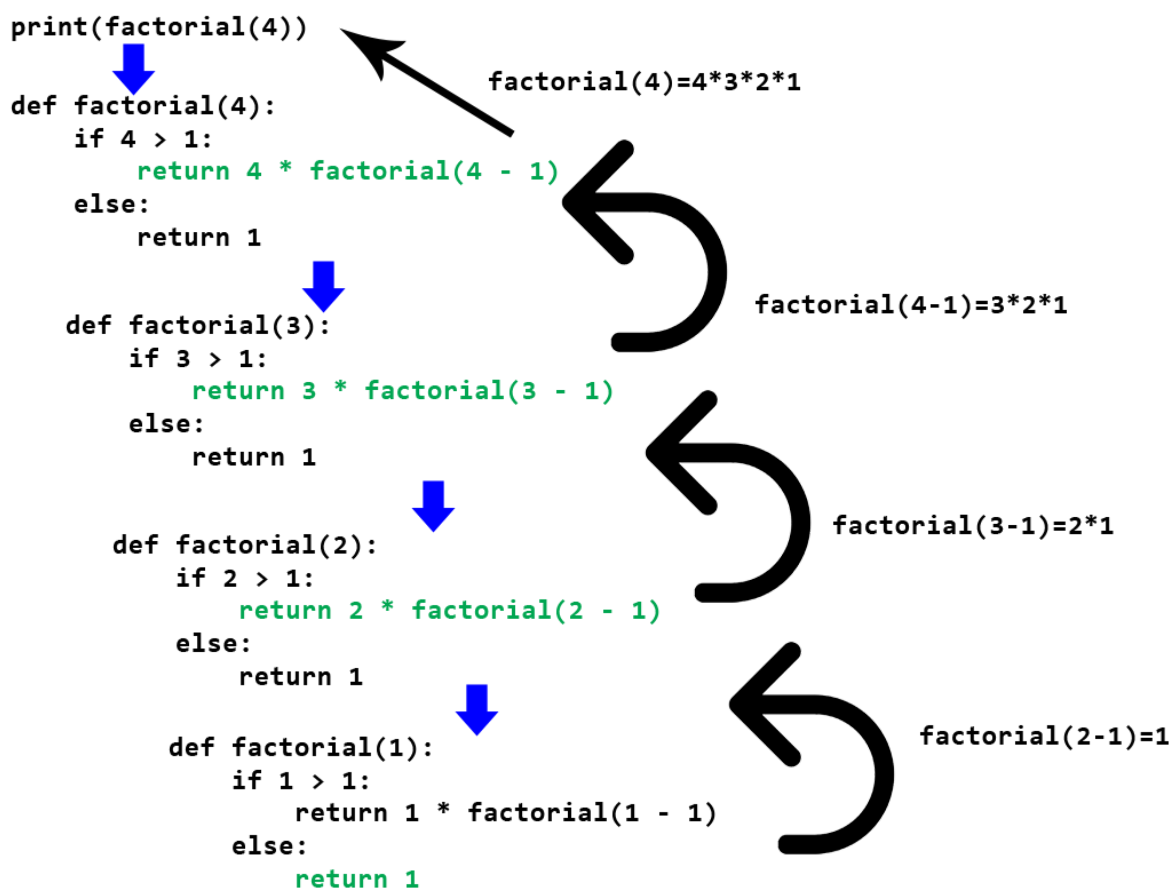


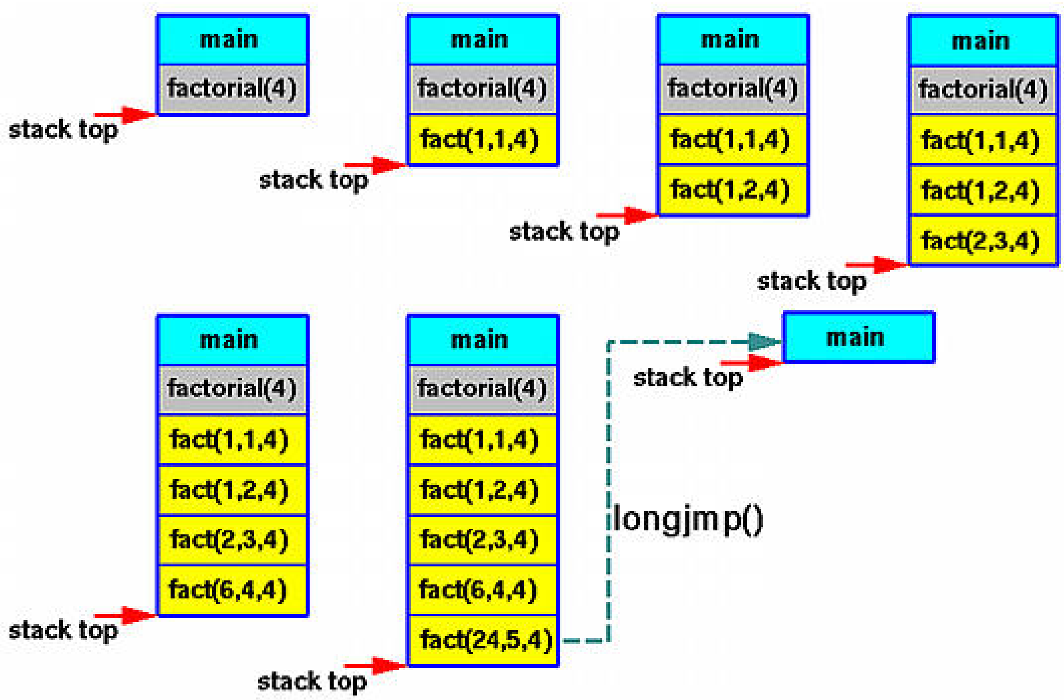

String Pool
java.lang.Object@579a19fd
str
param
foo()
mem
obj
int i=1
main()
Memory
toString()
Object
Java Runtime Memory

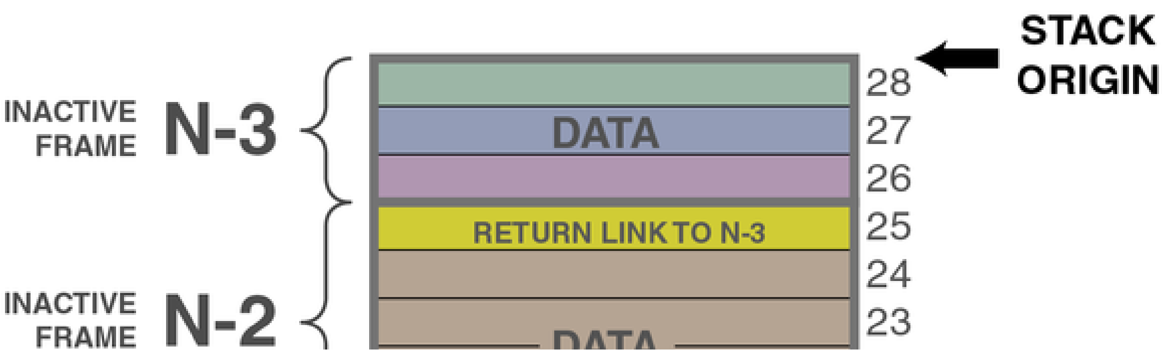
Why stack

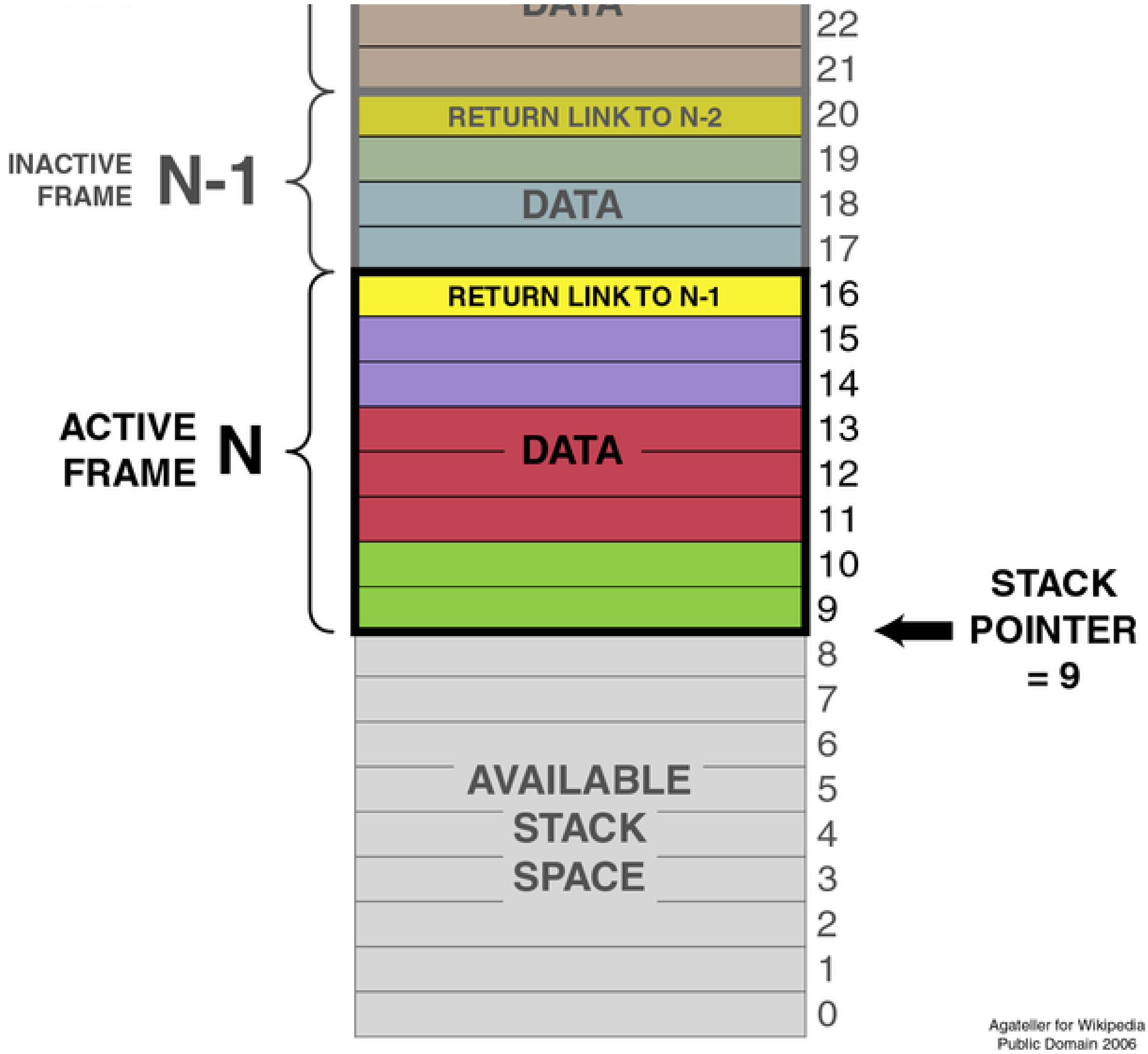

High
Address
Stack Frame
return addr
input (x)
input (y)
local (z)
return addr
input (a)
function1(int x, int y)
[
int z
z = function2(x+y)
]
int function2(int a)
[
return a + 100
]


Last In, First Out (LIFO) data structure
main ()
{ a(0);
}
void a (int m)
{ b(1);
}
void b (int n)
{ c(2);
}
void c (int o)
{ d(3);
}
void d (int p)
{
}
stack
Stack
grows
down
Stack Pointer

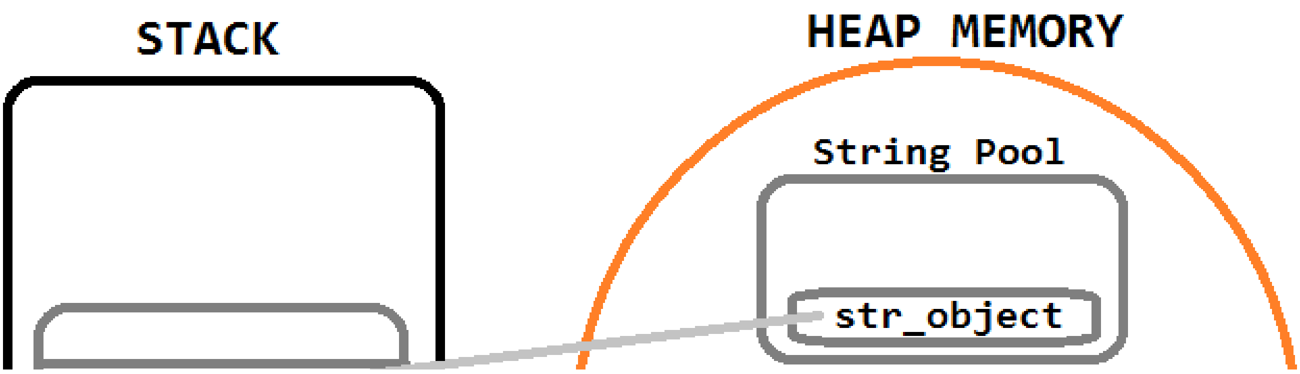

print(factorial(4))
factorial(4)=4*3*2*1
def factorial(4):
if 4 > 1:
return 4 * factorial(4 - 1)
else:
return 1
factorial(4-1)=3*2*1
def factorial(3):
if 3 > 1:
return 3 * factorial(3 - 1)
else:
return 1
factorial(3-1)=2*1
def factorial(2):
if 2 > 1:
return 2 * factorial(2 - 1)
else:
return 1
factorial(2-1)=1
def factorial(1):
if 1 > 1:
return 1 * factorial(1 - 1)
else:
return 1

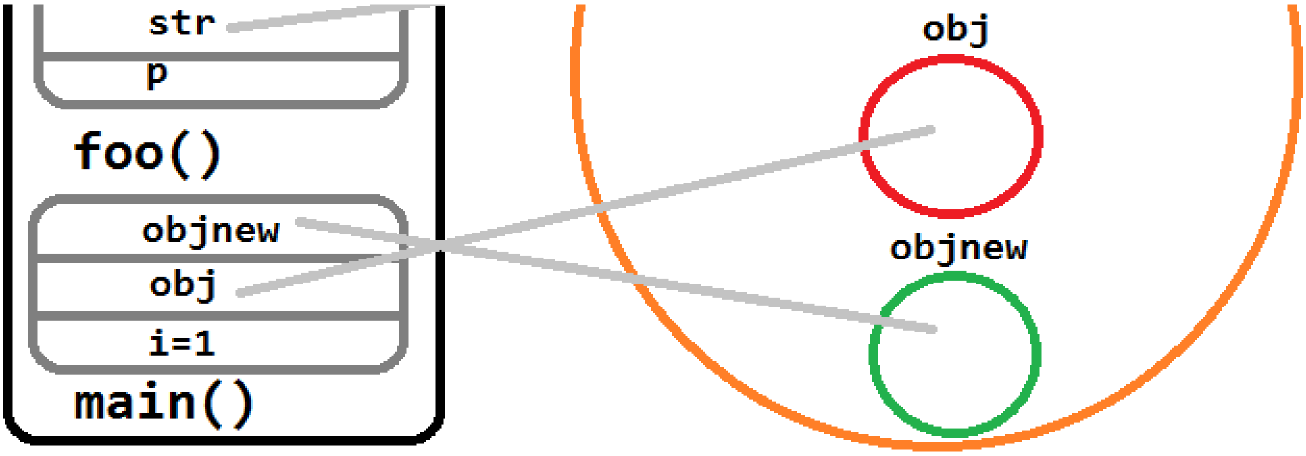

main
factorial(4)
stack top
fact(1,1,4)
fact(1,2,4)
fact(2,3,4)
fact(6,4,4)
fact(24,5,4)
longjmp()

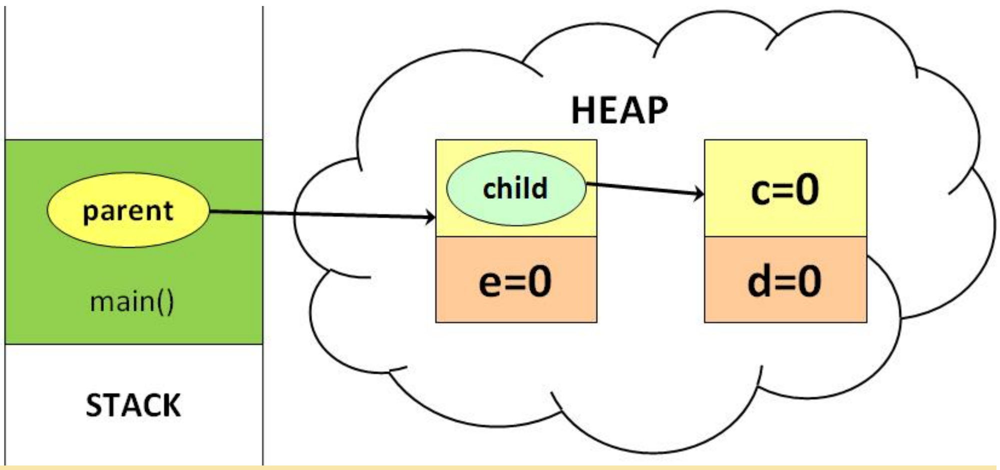

STACK ORIGIN
INACTIVE FRAME
N-3
DATA
RETURN LINK TO N-3
INACTIVE FRAME
N-2
28
27
26
25
24
23

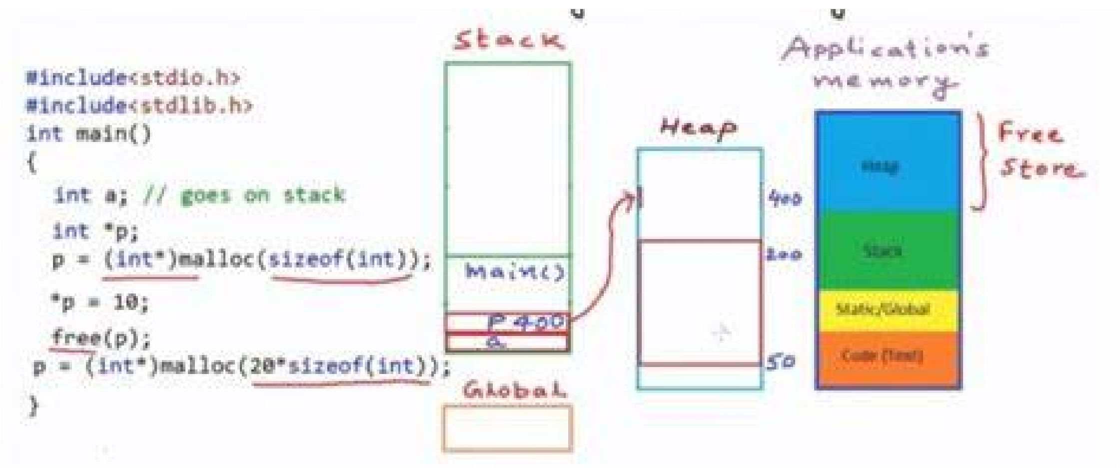

DATA
22
21
RETURN LINK TO N-2
20
INACTIVE FRAME N-1
19
DATA
18
17
RETURN LINK TO N-1
16
15
14
ACTIVE FRAME N
DATA
13
12
11
10
9
STACK POINTER = 9
8
7
6
AVAILABLE STACK SPACE
5
4
3
2
1
0
Agateller for Wikipedia
Public Domain 2006

Why Heap

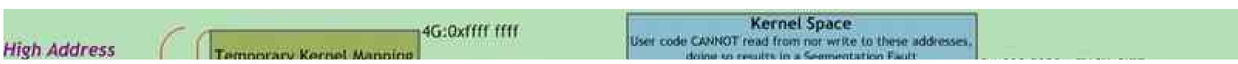

STACK
HEAP MEMORY
String Pool
str_object

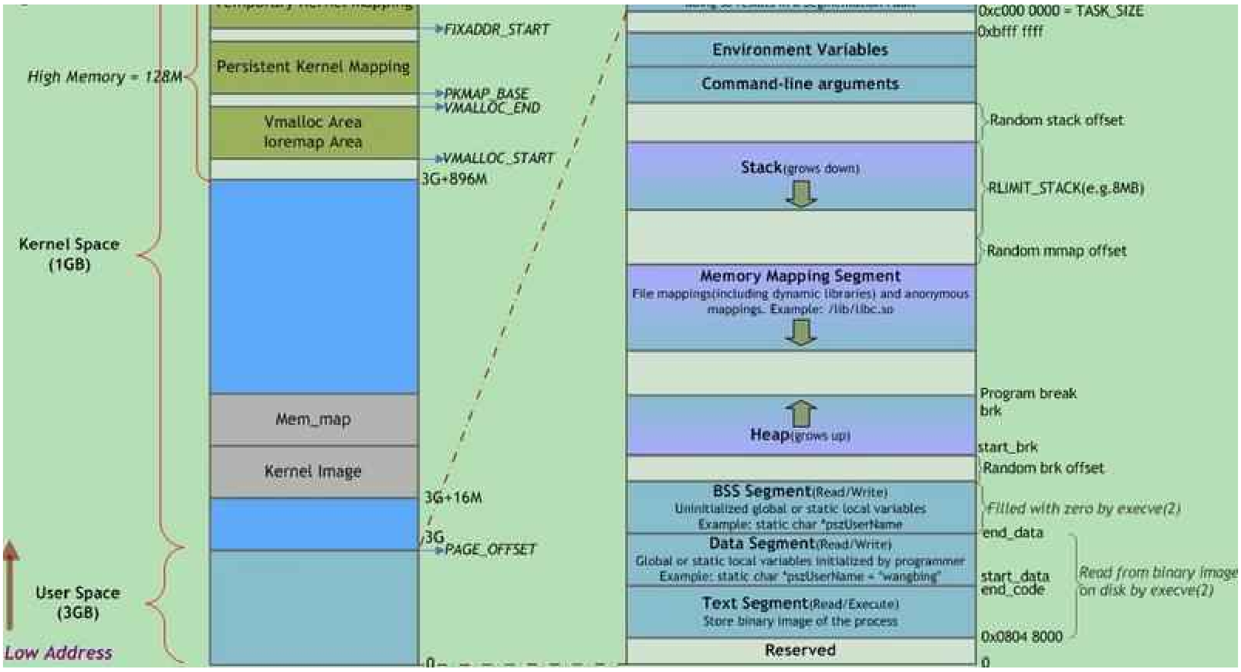


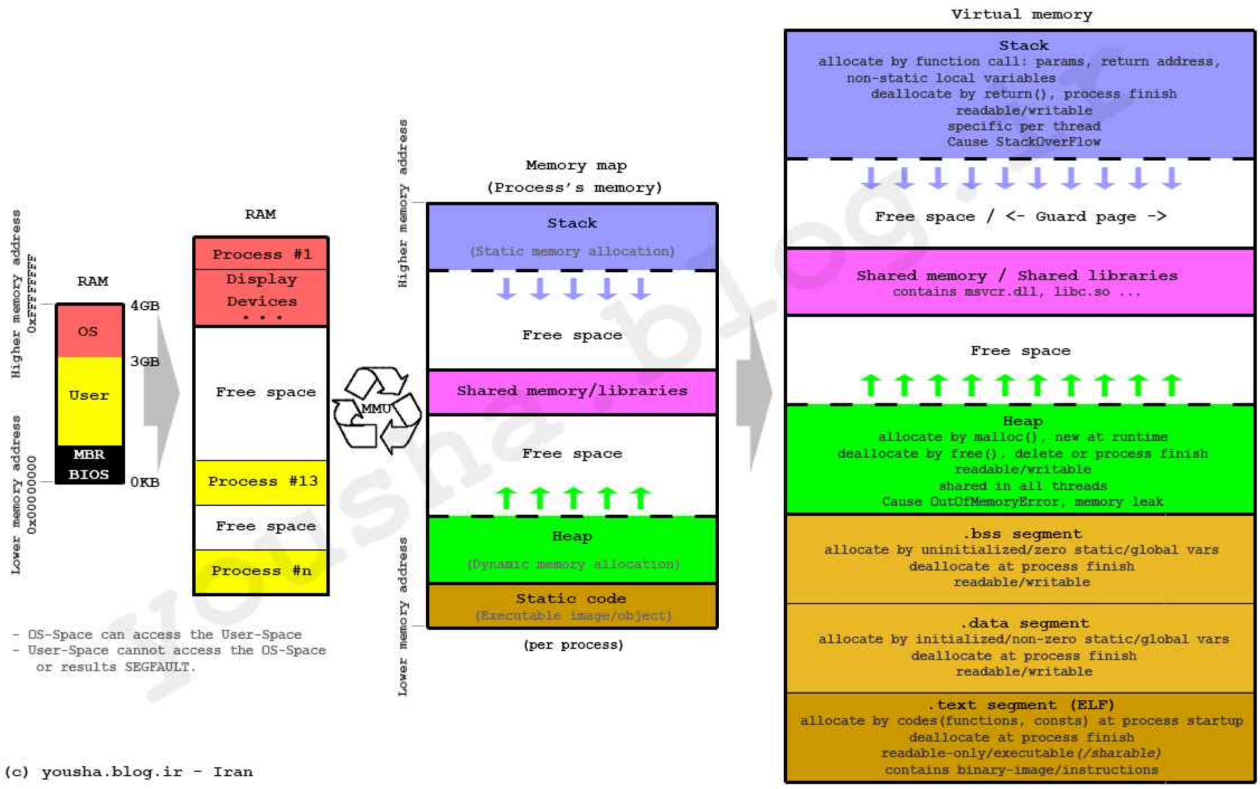




Whole Memory Layout







Assembly Language

What is it

C Loop Mapped to RISC-V Assembly

```c
int A[20];
int sum = 0;
for (int i=0; i<20; i++)
    sum += A[i];
```

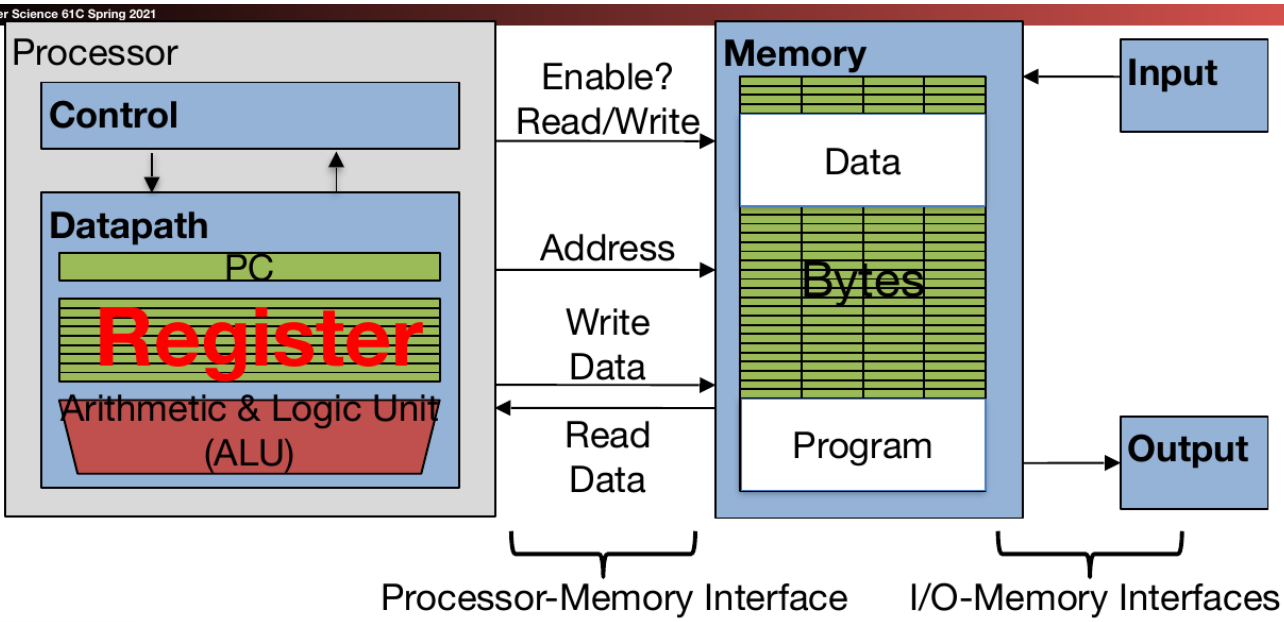
```
# Assume x8 holds pointer to A
# Assign x10=sum, x11=i
add x10, x0, x0 # sum=0
add x11, x0, x0 # i=0
addi x12,x0,20  # x12=20
Loop:
bge x11, x12, exit:
sll x13, x11, 2    # i * 4
add x13, x13, x8   # & of A + i
lw x13, 0(x13)     # *(A + i)
add x10, x10, x13 # increment sum
addi x11, x11, 1  # i++
j Loop             # Iterate
exit:
```

ARM

```
LDR r0,[p_a]
LDR r1,[p_b]
ADD r3,r0,r1
STR r3,[p_w]
LDR r2,[p_c]
ADD r0,r2,r3
STR r0,[p_x]
LDR r0,[p_d]
ADD r3,r2,r0
STR r3,[p_y]
```

x86

```
pushl   %ebp
movl    %esp,%ebp
subl    $0x4,%esp
movl    $0x0,0xfffffffc(%ebp)
cmpl    $0x63,0xfffffffc(%ebp)
jle     08048930
jmp     08048948
```

C, Java Variables vs. Registers

- In C (and most HLLs):
  - Variables declared and given a type
    - Example:
      ```
      int fahr, celsius;
      char a, b, c, d, e;
      ```
  - Each variable can ONLY represent a value of the type it was declared (e.g., cannot mix and match int and char variables)
    - If types are not declared, the object carries around the type with it. EG in python:
      ```
      a = "fubar" # now a is a string
      a = 121 # now a is an integer
      ```
- In Assembly Language:
  - Registers have ***no type***;
  - Operation determines how register contents are interpreted

RISC-V Instructions

- Instructions are fixed, 32b long
  - Must also be word aligned, or half-word aligned if the 16b optional (C) instruction set is also enabled
- Only a few formats (we'll go into detail later)...

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

RISC-V Instruction Assembly Syntax

- Instructions have an opcode and operands

E.g., `add x1, x2, x3 # x1 = x2 + x3`

is assembly comment syntax

Destination register

Second operand register

Operation code (opcode)

First operand register

Addition and Subtraction of Integers Example 1

- How to do the following C statement?

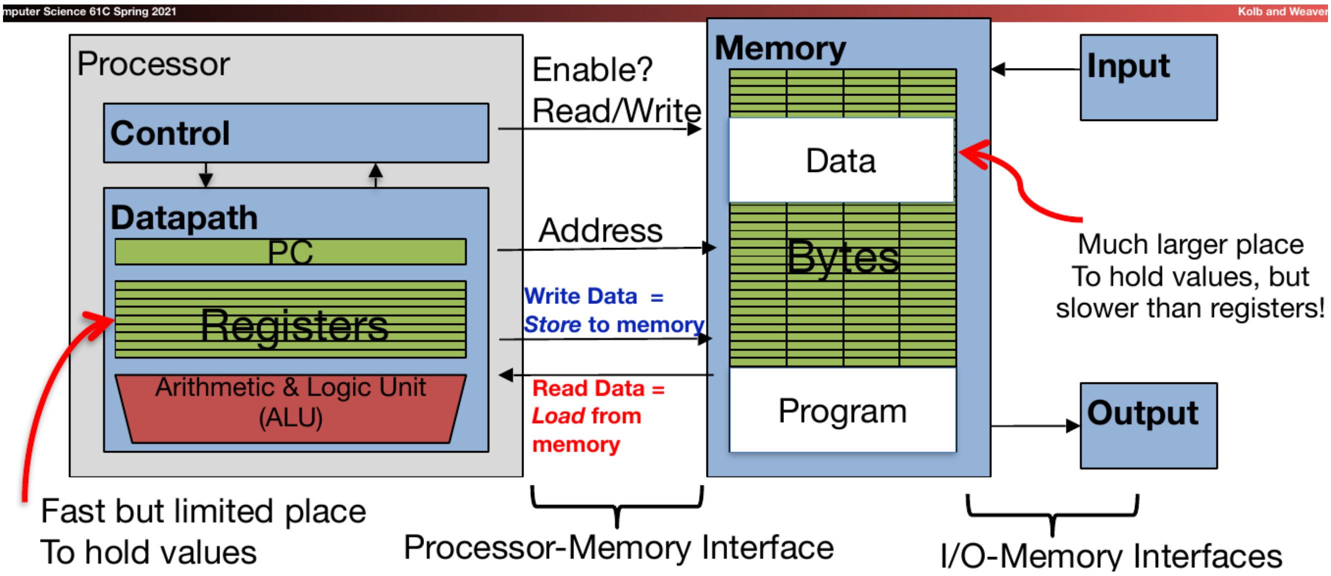
```
a = b + c + d - e;
```

- Break into multiple instructions

```
add x1, x2, x3   # temp = b + c
add x1, x1, x4   # temp = temp + d
sub x1, x1, x5   # a = temp - e
```

- A single line of C may turn into several RISC-V instructions

`add x3,x4,x0` (in RISC-V) same
`f = g` (in C)



Computer Decision Making

- Based on computation, do something different
- Normal operation on CPU is to execute instructions in sequence
- Need special instructions for programming languages: *if*-statement

- RISC-V: *if*-statement instruction is

  `beq register1,register2,L1`

  means: go to instruction labeled L1
  if (value in register1) == (value in register2)
  ….otherwise, go to next instruction
- **`beq`** stands for *branch if equal*
- Other instruction: **`bne`** for *branch if not equal*

Example, a for loop

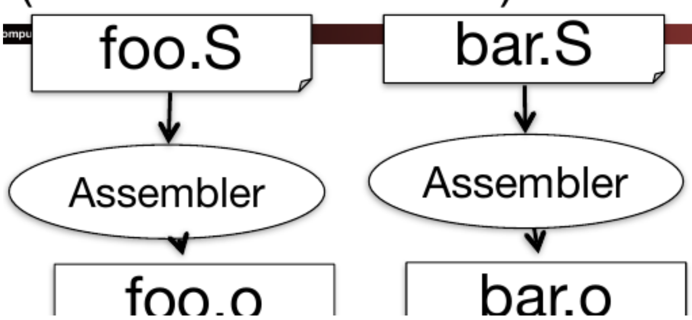

```
• for(i = 0; i < 10; ++i) {
     ....
  }
```

- Assume i is in register x3

```
• add x3 x0 x0        # Set i to 0
  j check             # Jump to the check:
                      # We'll see how this work soon
  loop_start: ....

  addi x3 x3 1
  check:
  li x4 10            # load the constant 10
  blt x3 x4 loop_start
```

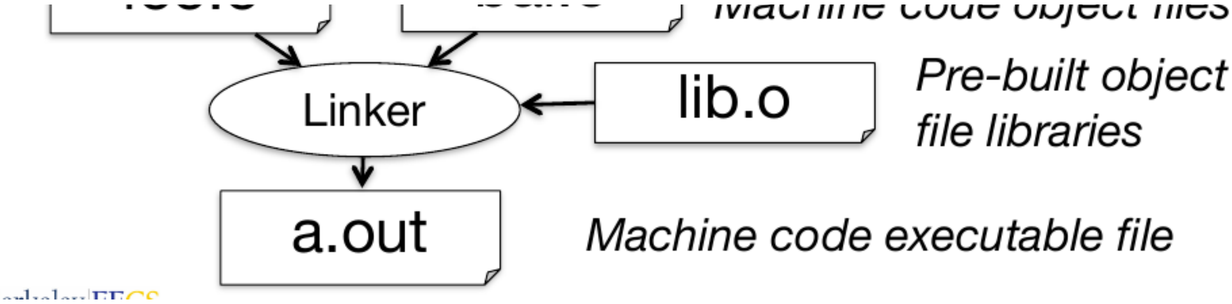
Assembler to Machine Code (more later in course)



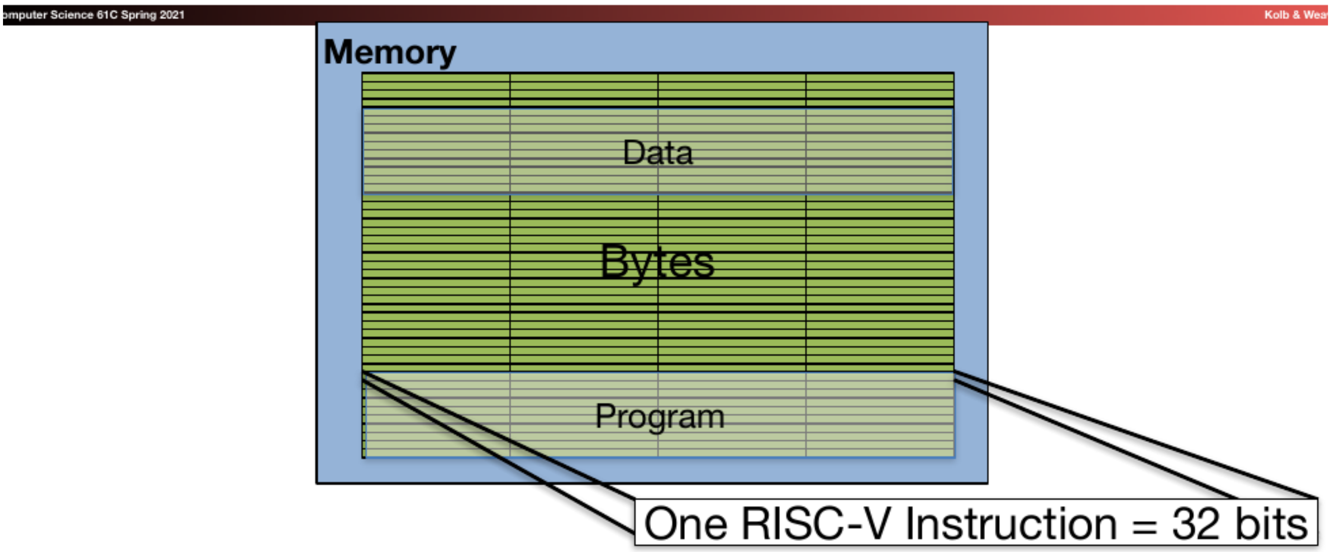
Assembler source files (text)

Assembler converts human-readable assembly code to instruction bit patterns

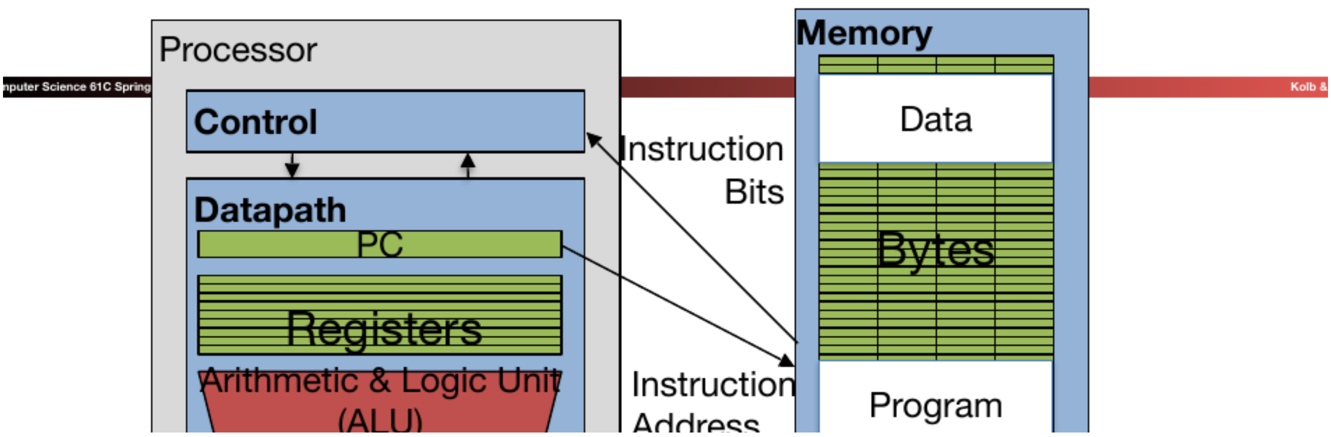
Machine code object files



How Program is Stored



Program Execution



- **PC** (program counter) is special internal 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)

Function Call

Six Fundamental Steps in Calling a Function

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

The Calling Convention: A Contract Between Functions…

- The "Calling Convention" in the ABI is the format/usage of registers in a way between the function ***caller*** and function

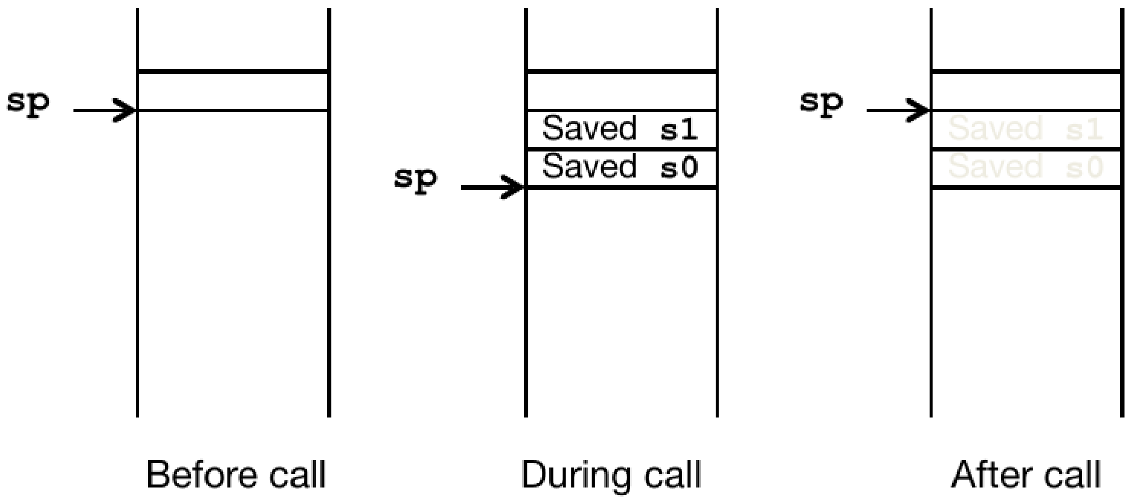
callee, if all functions implement it, everything works out

- It is effectively a contract between functions

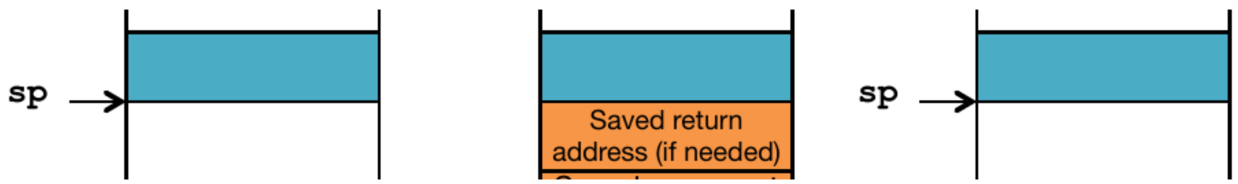
- Registers are two types
  - ***caller-saved***
    - The function invoked (the callee) can do whatever it wants to them!
    - Means that the caller can not count on them not being mangled beyond recognition
  - ***callee-saved***
    - The function invoked must restore them before returning (if used)

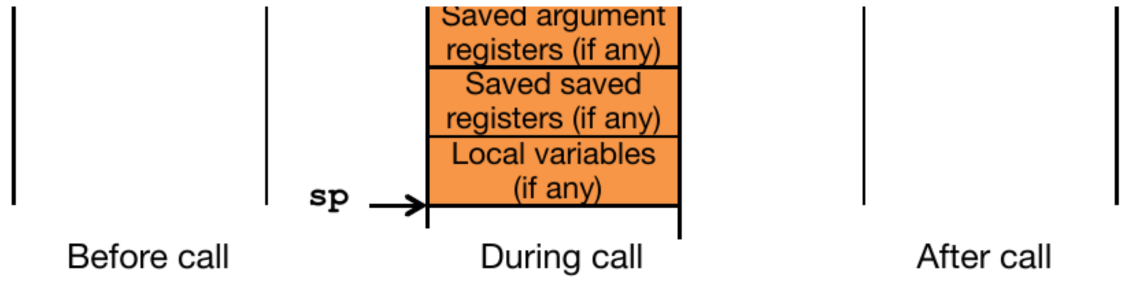
Stack Before, During, After Function

- Need to save old values of `s0` and `s1`



Stack Before, During, After Function





Using the Stack (2/2)

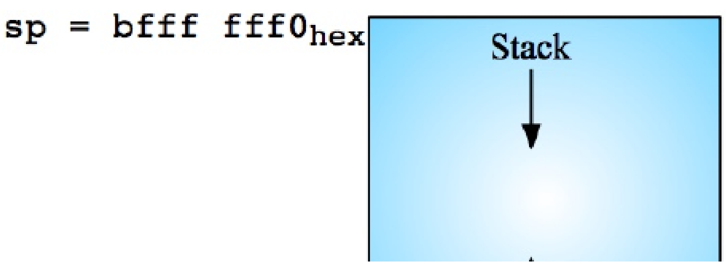
puter Science 61C Spring 2021

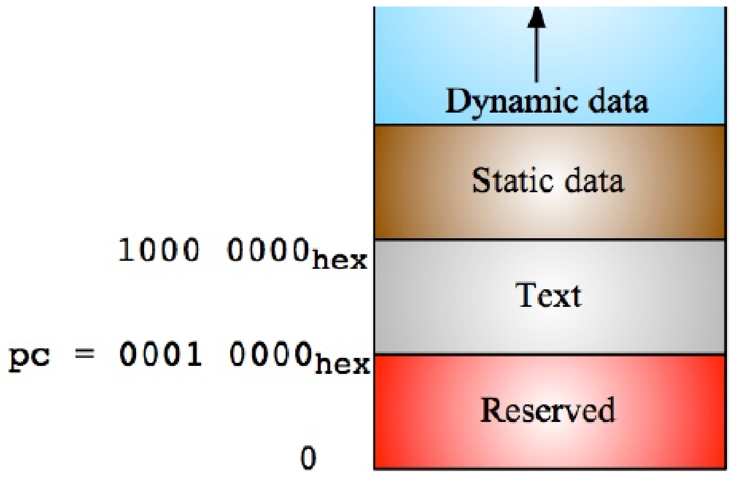
```
int sumSquare(int x, int y) {
  return mult(x,x)+ y; }
```

```
sumSquare:
"push"  addi sp,sp,-8    # reserve space on stack
        sw ra, 4(sp)     # save ret addr
        sw a1, 0(sp)     # save y
        mv a1,a0         # mult(x,x)
        jal mult         # call mult
        lw a1, 0(sp)     # restore y
        add a0,a0,a1     # mult()+y
        lw ra, 4(sp)     # get ret addr
"pop"   addi sp,sp,8     # restore stack
        jr ra
mult: ...
```

RV32 Memory Allocation

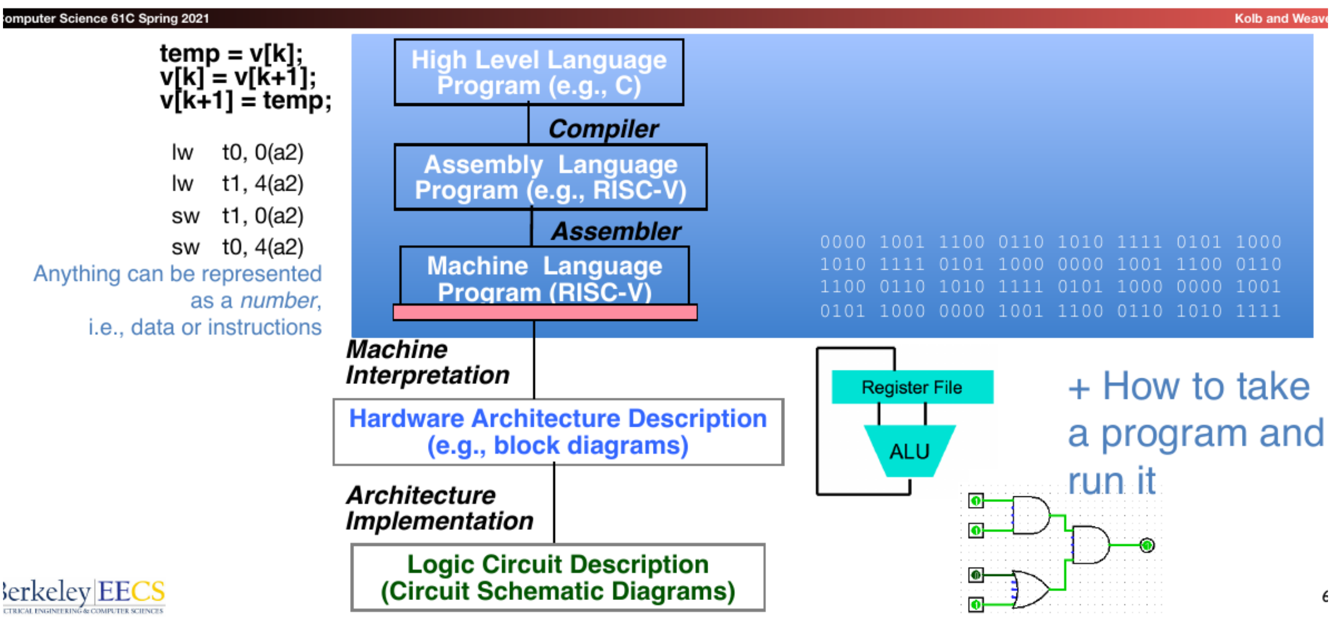
omputer Science 61C Spring 2021





CALL(Compiler/Assembler/Linker/Loader)

Levels of Representation/Interpretation

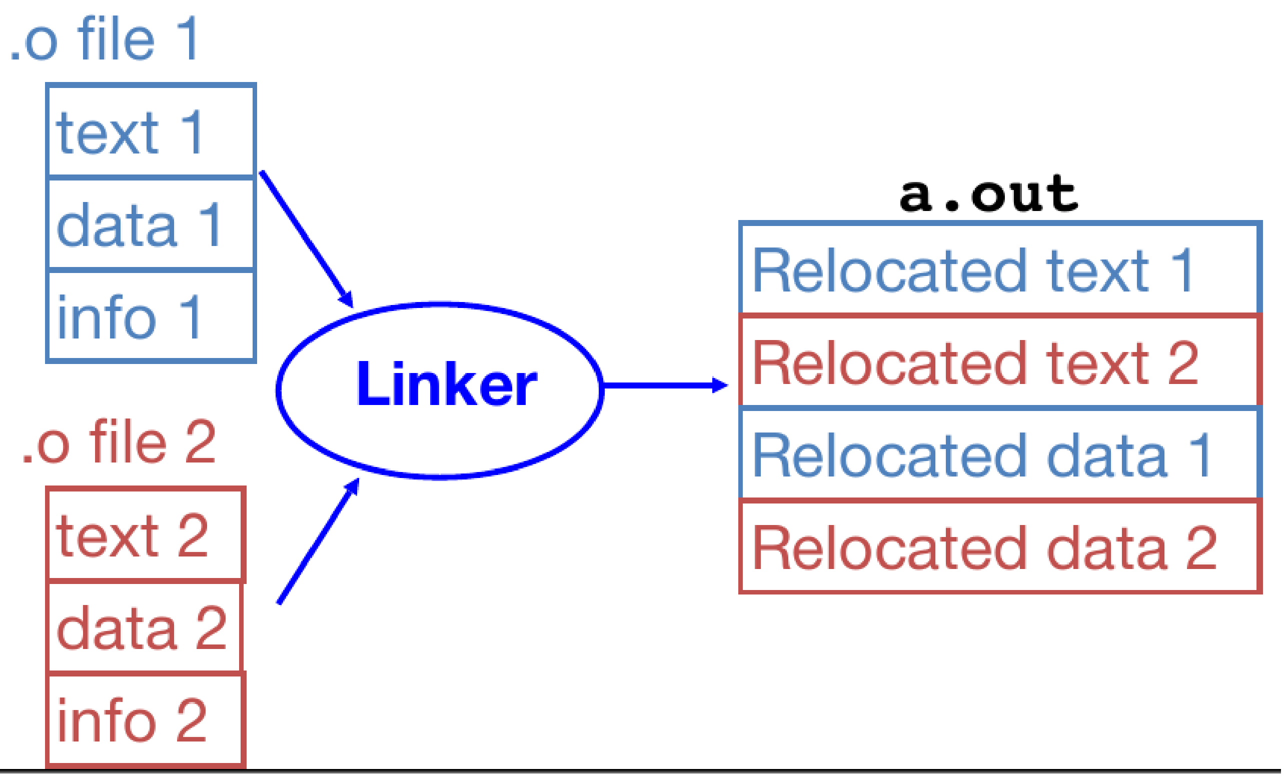


Scheme Java C++ C Java bytecode Assembly Machine code

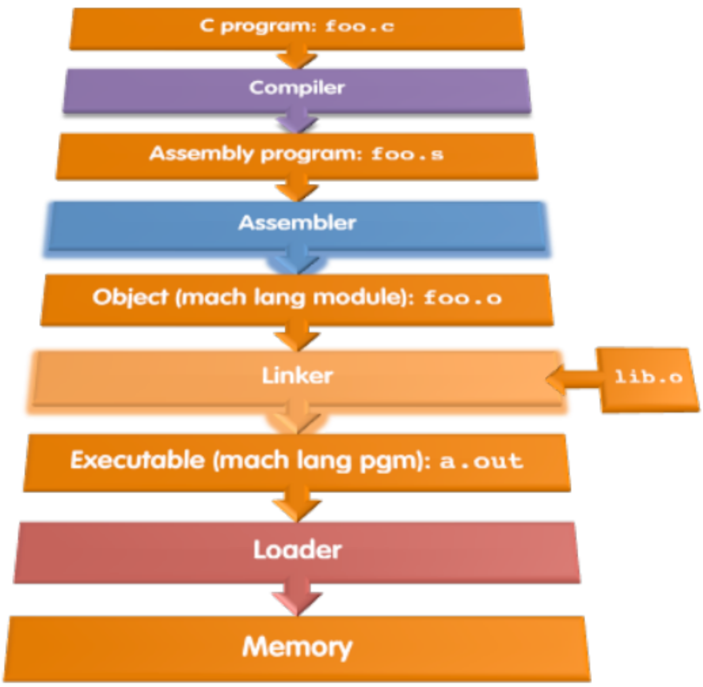
Easy to program Difficult to program

Inefficient to interpret

Efficient to interpret



- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A `.s` file becomes a `.o` file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.



Loader … what does it do?

- 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
  - If main routine returns, start-up routine terminates program with the exit system call

Example: C ⇒ Asm ⇒ Obj ⇒ Exe ⇒ Run

C Program Source Code: *prog.c*

```
#include <stdio.h>
int main (int argc, char *argv[]) {
 int i, sum = 0;
 for (i = 0; i <= 100; i++)
    sum = sum + i * i;
 printf ("The sum of sq from 0 .. 100 is %d\n", sum);
}
```

"*printf*" lives in "*libc*"

Compilation: Assembly Language: i = t0, sum = a1


```
  .text
  .align 2
  .globl main
main:
  addi sp, sp, -4
  sw ra, 0(sp)
  mv t0, x0
  mv a1, x0
  li t1, 100
  j check
loop:
  mul t2, t0, t0
  add a1, a1, t2
```

Pseudo-Instructions?

```
check:
  blt t0, t1 loop:
  la $a0, str
  jal printf
  mv a0, x0
  lw ra, 0(sp)
  addi sp, sp 4
  ret
  .data
  .align  0
str:
  .asciiz "The sum of sq from 0
  .. 100 is %d\n"
```

```
    addi t0, t0, 1
```

Assembly step 1:
Remove Pseudo Instructions, assign jumps


```
   .text
   .align 2
   .globl main
main:
   addi sp, sp, -4
   sw ra, 0(sp)
   addi t0, x0, 0
   addi a1, x0, 0
   addi t1, x0, 100
   jal x0, 12
loop:
   mul t2, t0, t0
   add a1, a1, t2
   addi t0, t0, 1
```

Pseudo-Instructions? Underlined

```
check:
   blt t0, t1 -16
   lui a0, l.str
   addi a0, a0, r.str
   jal printf
   addi a0, x0, 0
   lw ra, 0(sp)
   addi sp, sp 4
   jalr x0, ra
   .data
   .align  0
str:
   .asciiz "The sum of sq from 0
   .. 100 is %d\n"
```

Assembly step 2

Create relocation table and symbol table

- Symbol Table

Label	address (in module)	type
main:	**0x00000000**	**global text**
loop:	**0x00000014**	**local text**
str:	**0x00000000**	**local data**

- Relocation Information

Address	Instr. type	Dependency
`0x0000002c`	`lui`	`l.str`
`0x00000030`	`addi`	`r.str`
`0x00000034`	`jal`	`printf`

Assembly step 3

- Generate object (`.o`) file:
  - Output binary representation for
    - text segment (instructions)
    - data segment (data)
    - symbol and relocation tables
  - Using dummy "placeholders" for unresolved absolute and external references
- And then… We link!

Linking Just Resolves References...

- So take all the .o files
  - Squish the different segments together
- For each entry in the relocation table:
  - Replace it with the actual address for the symbol table of the item you are linking to

- Result is a single binary

```c
#include <stdio.h>
int main()
{
  printf("Hello, %s\n", "world");
  return 0;
}
```



```
 .text                          # Directive: enter text section
   .align 2                     # Directive: align code to 2^2 bytes
   .globl main                  # Directive: declare global symbol main
 main:                          # label for start of main
   addi sp,sp,-16               # allocate stack frame
   sw   ra,12(sp)               # save return address
   lui  a0,%hi(string1)         # compute address of
   addi a0,a0,%lo(string1)      #   string1
   lui  a1,%hi(string2)         # compute address of
   addi a1,a1,%lo(string2)      #   string2
   call printf                  # call function printf
   lw   ra,12(sp)               # restore return address
   addi sp,sp,16                # deallocate stack frame
   li   a0,0                    # load return value 0
   ret                          # return
   .section .rodata             # Directive: enter read-only data section
   .balign 4                    # Directive: align data section to 4 bytes
 string1:                       # label for first string
   .string "Hello, %s!\n"       # Directive: null-terminated string
 string2:                       # label for second string
   .string "world"              # Directive: null-terminated string
```

Berkeley EECS

Assembled `Hello.s`: Linkable `Hello.o`

```
00000000 <main>:
0:   ff010113 addi sp,sp,-16
4:   00112623 sw ra,12(sp)
8:   00000537 lui a0,0x0      # addr placeholder
c:   00050513 addi a0,a0,0    # addr placeholder
10:  000005b7 lui a1,0x0      # addr placeholder
14:  00058593 addi a1,a1,0    # addr placeholder
18:  00000097 auipc ra,0x0    # addr placeholder
1c:  000080e7 jalr ra         # addr placeholder
20:  00c12083 lw ra,12(sp)
24:  01010113 addi sp,sp,16
28:  00000513 addi a0,a0,0
2c:  00008067 jalr ra
```

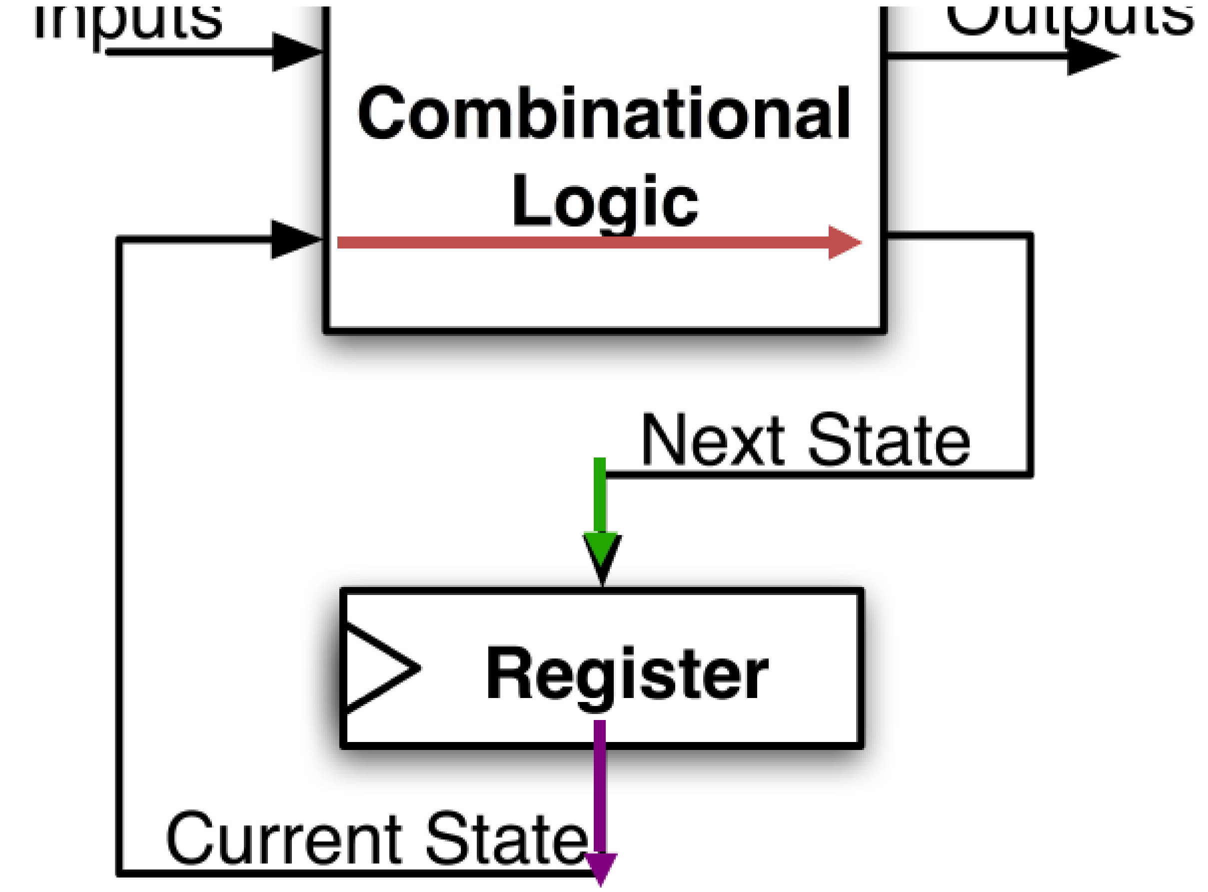
Linked `Hello.o`: `a.out`

mputer Science 61C Spring 2021

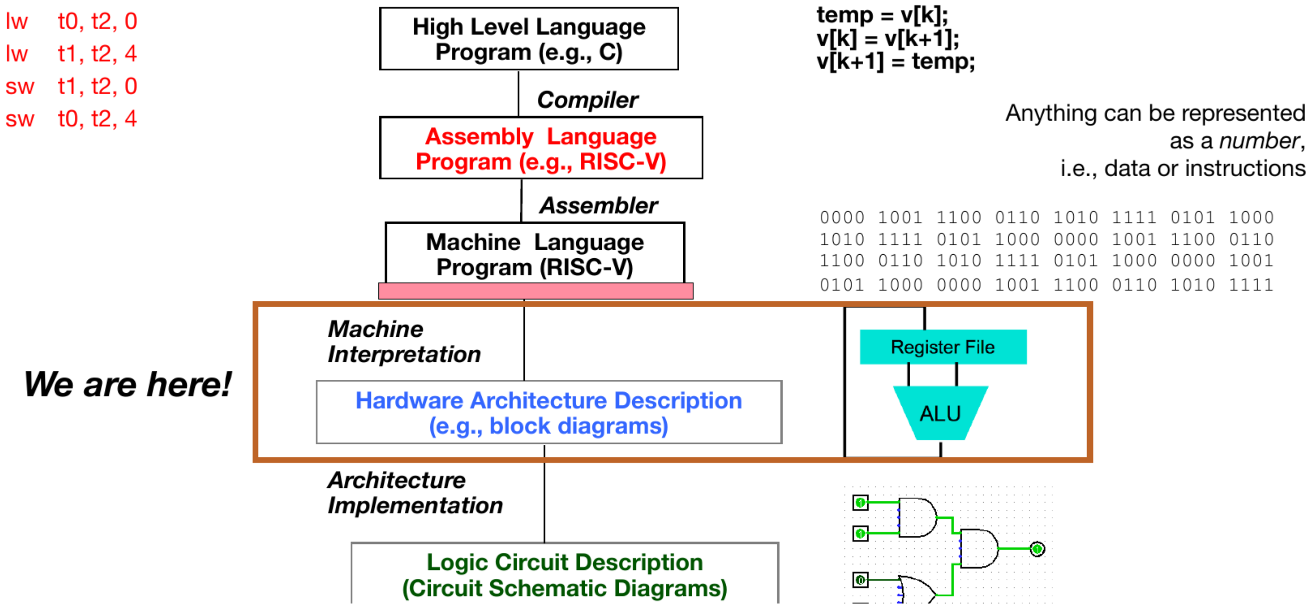
```
000101b0 <main>:
  101b0: ff010113 addi sp,sp,-16
  101b4: 00112623 sw   ra,12(sp)
  101b8: 00021537 lui  a0,0x21
  101bc: a1050513 addi a0,a0,-1520 # 20a10 <string1>
  101c0: 000215b7 lui  a1,0x21
  101c4: a1c58593 addi a1,a1,-1508 # 20a1c <string2>
  101c8: 288000ef jal  ra,10450    # <printf>
  101cc: 00c12083 lw   ra,12(sp)
  101d0: 01010113 addi sp,sp,16
  101d4: 00000513 addi a0,0,0
  101d8: 00008067 jalr  ra
```

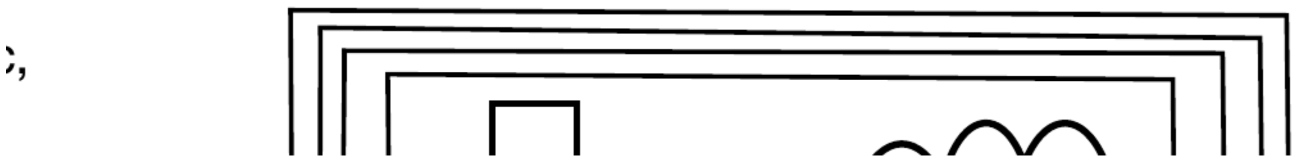
Hardware

Data Path



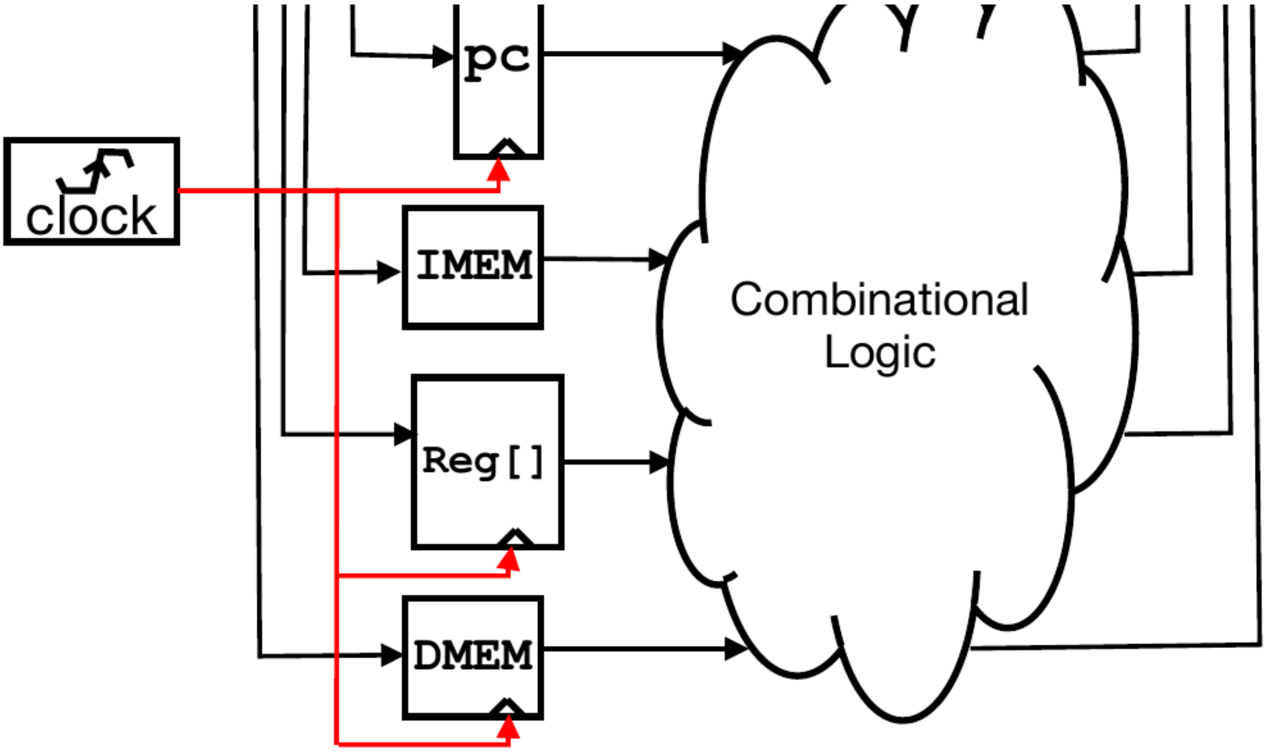
Great Idea #1: Abstraction (Levels of Representation/Interpretation)



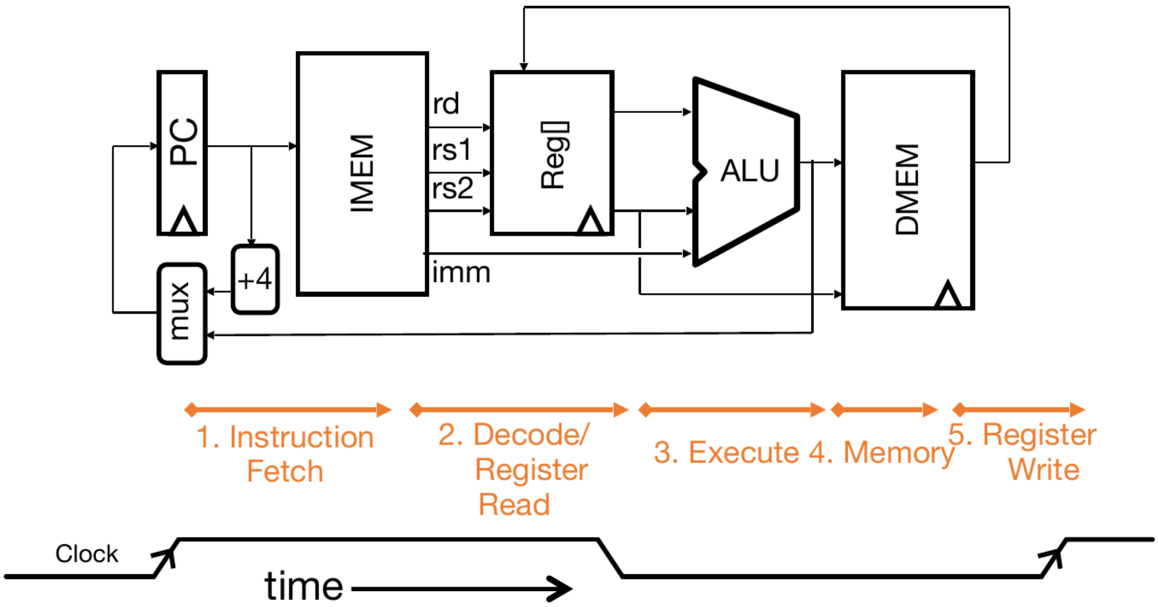


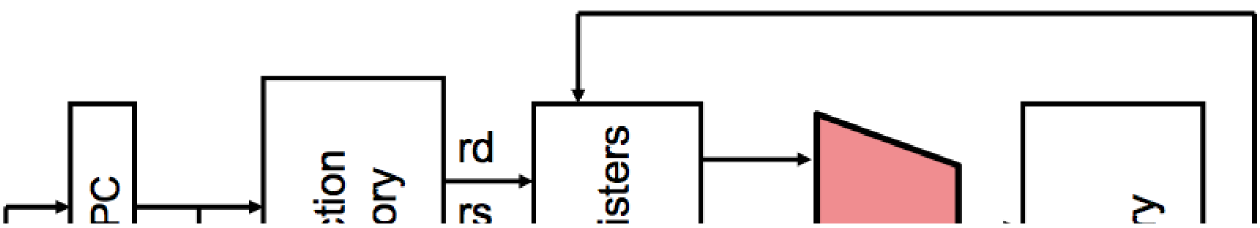
t

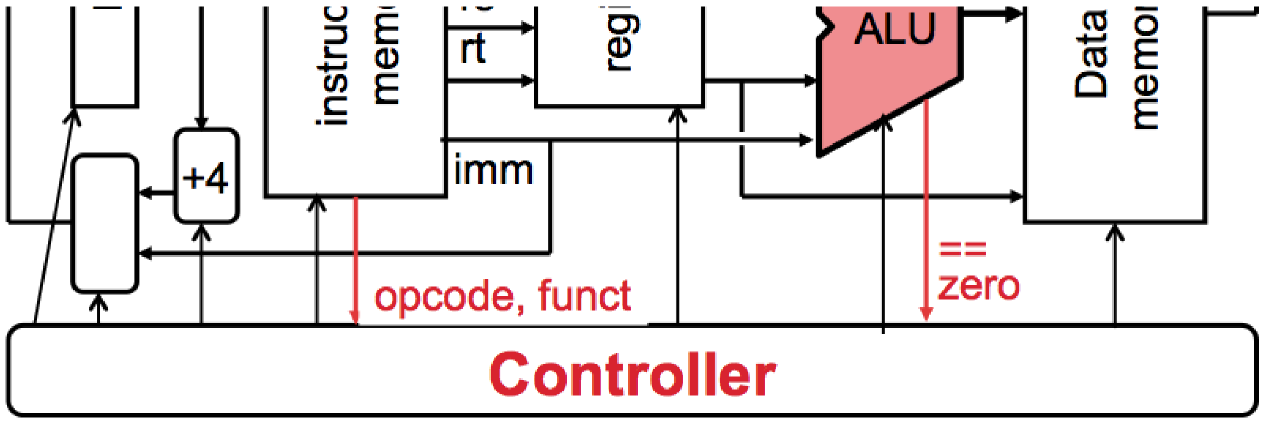
),



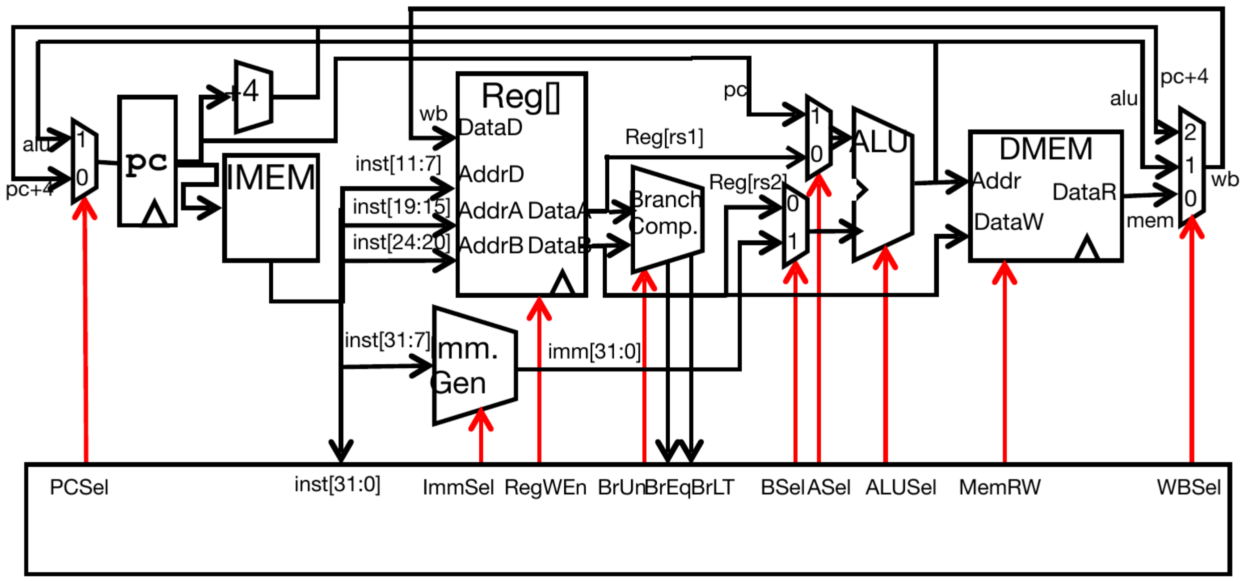
Basic Phases of Instruction Execution





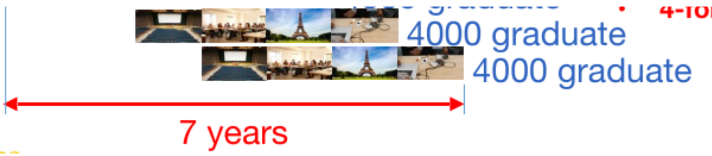


Single-Cycle RISC-V RV32I Datapath



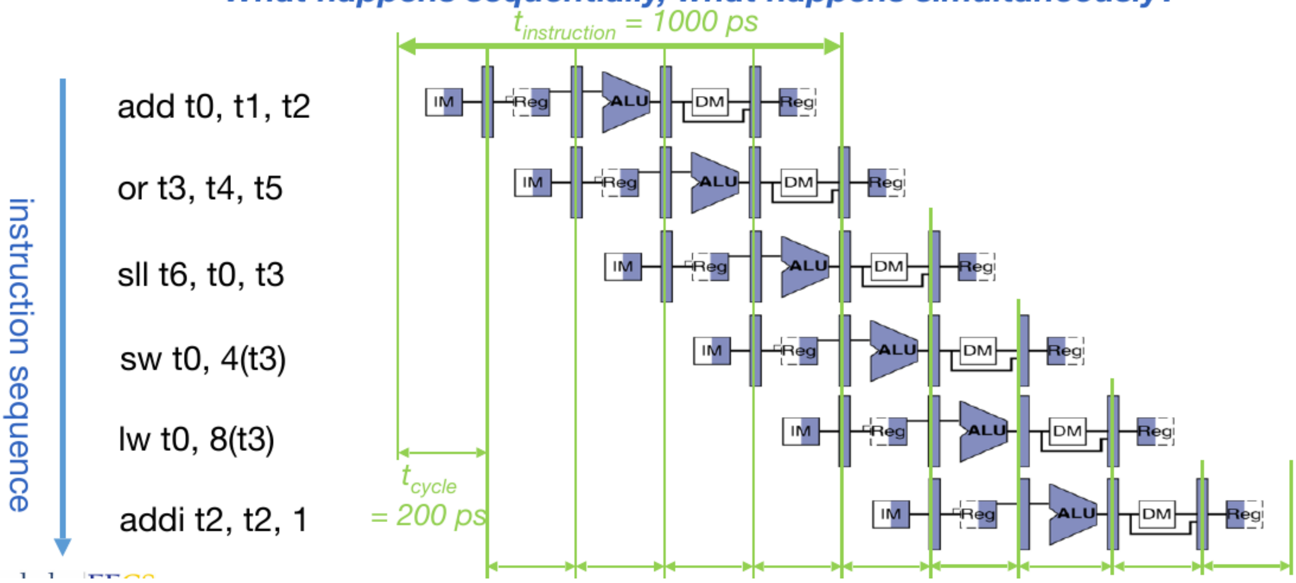
Computer Scientist Education

- 4-fold improvement over serial education



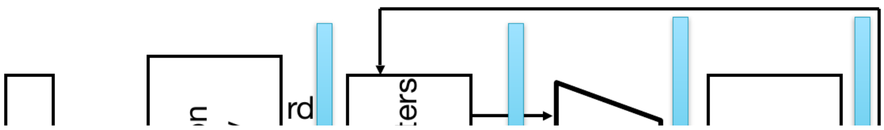
Sequential vs Simultaneous

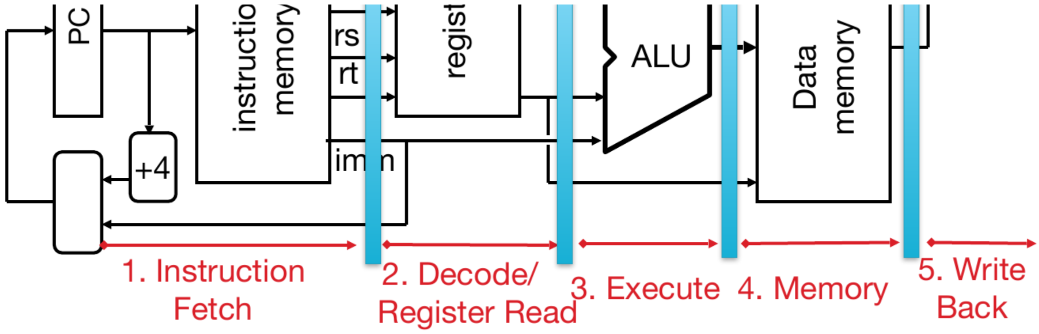
What happens sequentially, what happens simultaneously?



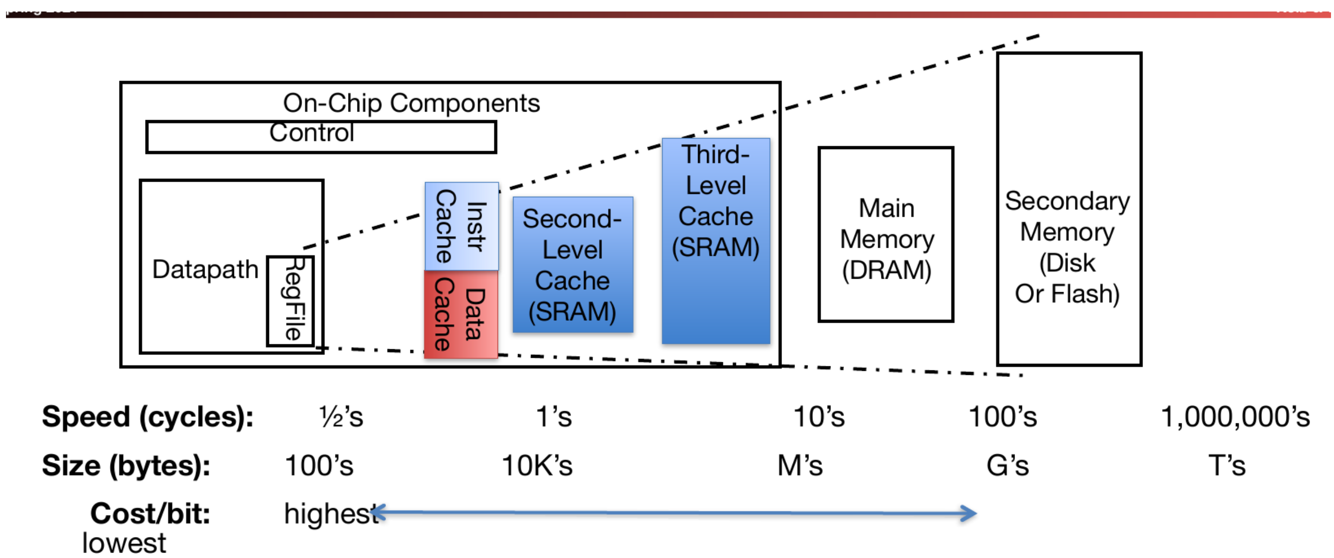
Pipeline registers

- Need registers between stages
  - To hold information produced in previous cycle

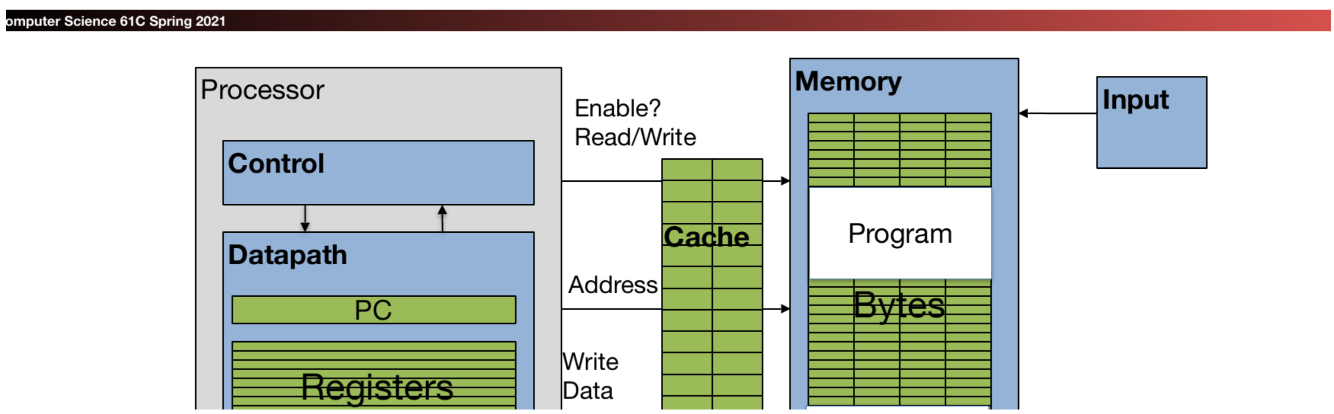


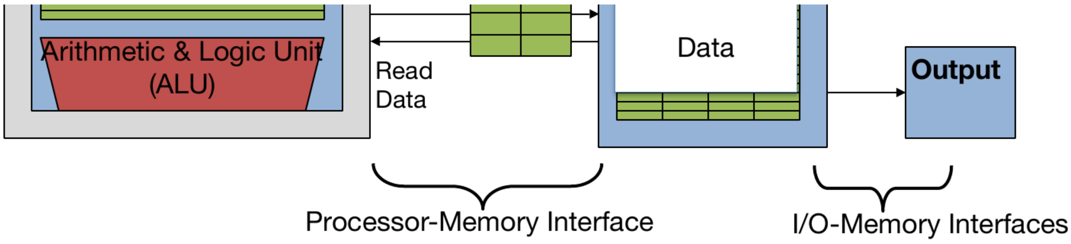


Cache



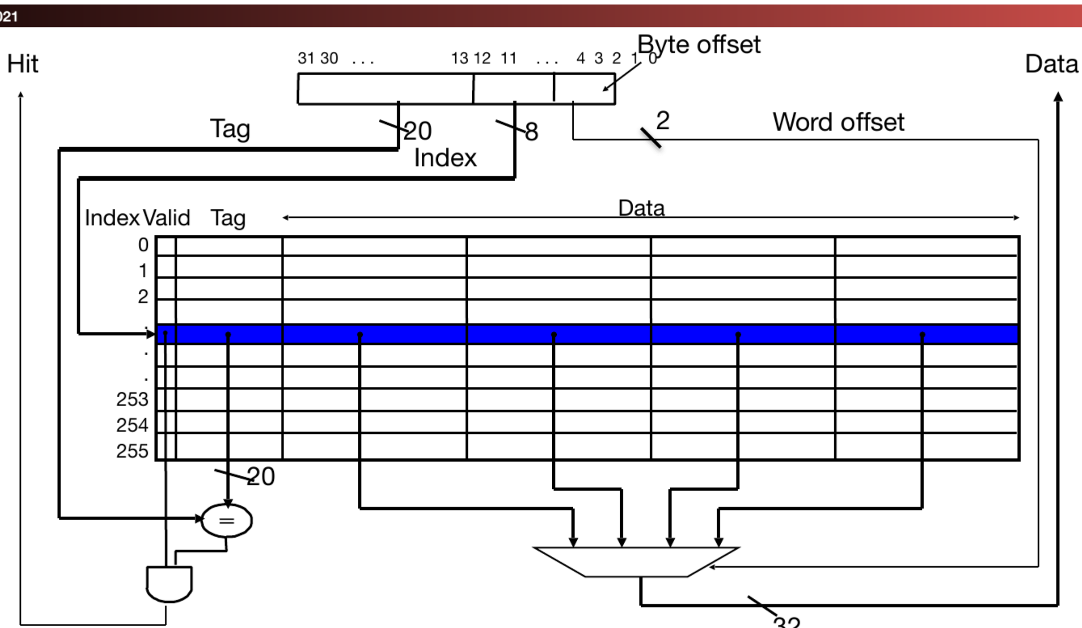
Adding Cache to Computer





5 4	3 2	1 0
Tag	*Index*	*Byte Offset*
Mem Block Within $ Block	Block Within $	Byte Within Block

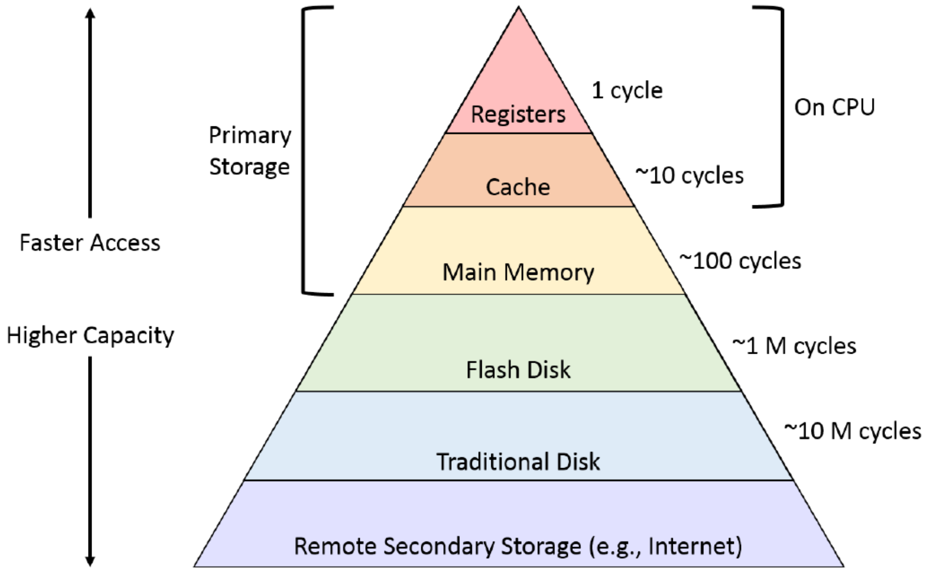
Multiword-Block Direct-Mapped Cache: 16B block size, 4 kB data



What kind of locality are we taking advantage of?

What kind of locality are we taking advantage of?

The Memory Hierarchy



The Memory Hierarchy

Next, Operating System!