

Course Info

- HW2 due Mar. 3rd
- Lab 2 is available and in this week's Lab session
- Discussion this week: Venus (for RISC-V), Memory Management & debug



信息科学与技术学院

School of Information Science and Technology

CS 110

Computer Architecture

C Memory Management

Instructors:

Siting Liu & Chundong Wang

Course website: <https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html>

School of Information Science and Technology (SIST)

ShanghaiTech University

2023/2/6

C Memory Management

- To simplify, assume one program runs at a time

- A program's address space contains 4 regions:

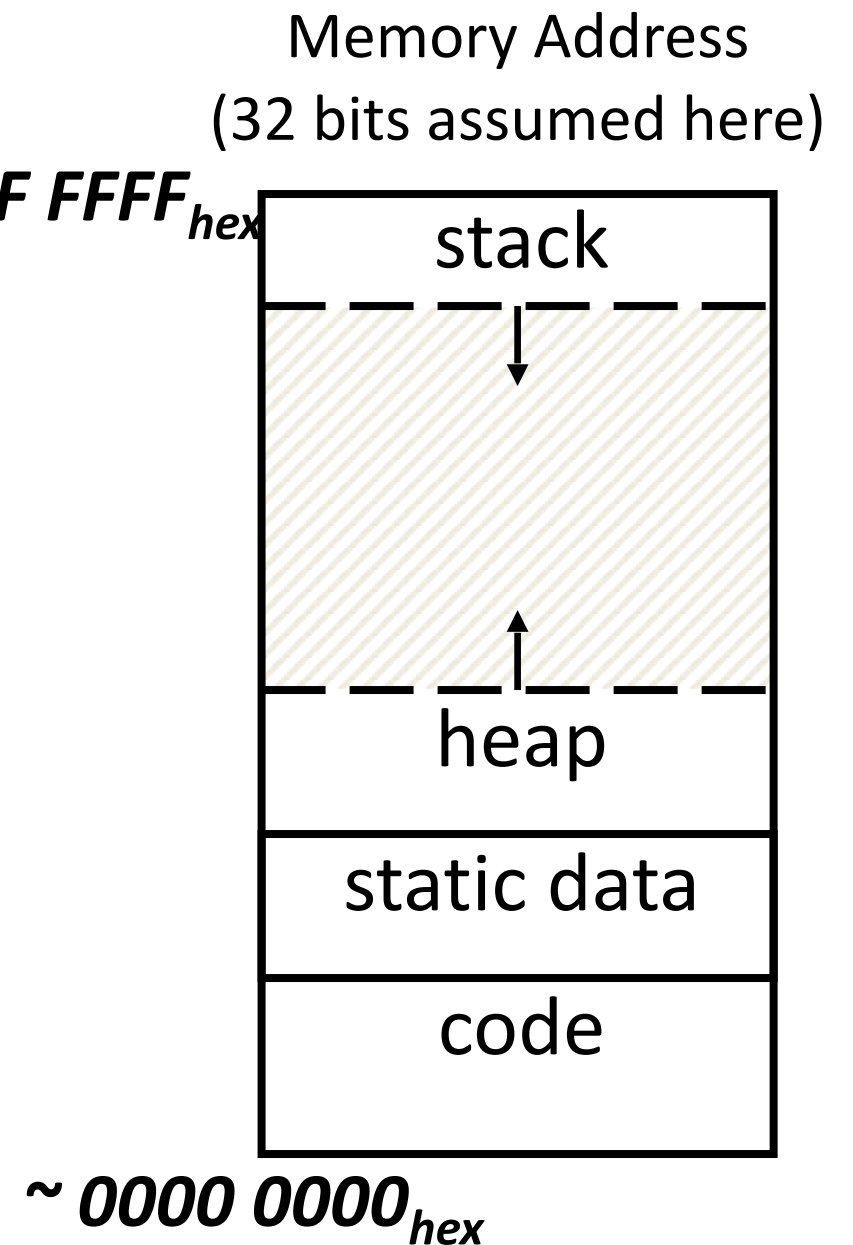
- stack: local variables inside functions, grows downward

- heap: space requested for dynamic data via `malloc()`; resizes dynamically, grows upward

- static {
 - static data: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.

- code (a.k.a. text): loaded when program starts, does not change

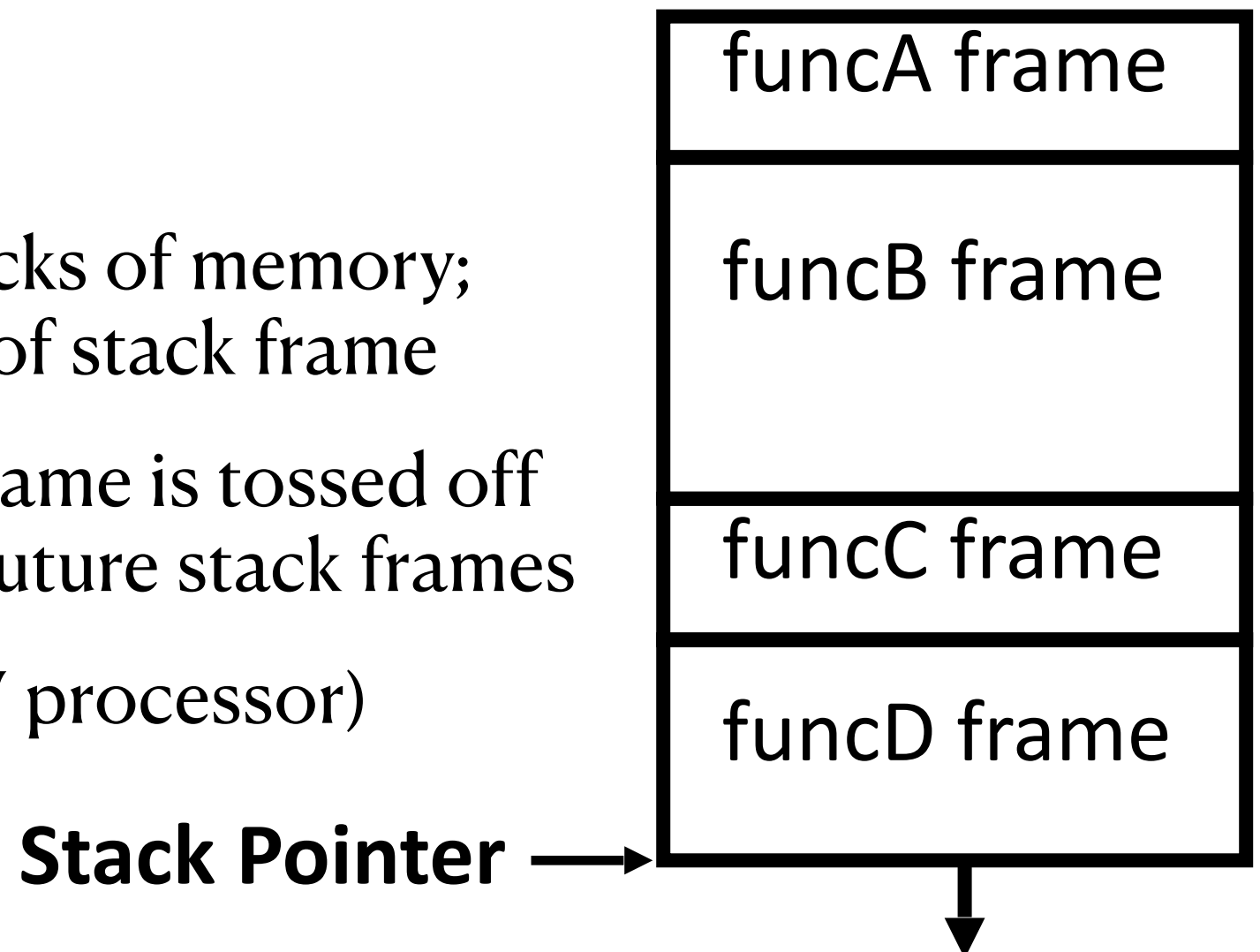
- 0x0 unwritable/unreadable (NULL pointer)



The Stack

- Every time a function is called, a new “stack frame” is allocated on the stack
- Stack frame includes:
 - Return address (who called me?)
 - Arguments
 - Space for local variables
- Stack frames contiguous blocks of memory; stack pointer indicates start of stack frame
- When function ends, stack frame is tossed off the stack; frees memory for future stack frames
- Details covered later (RISC-V processor)

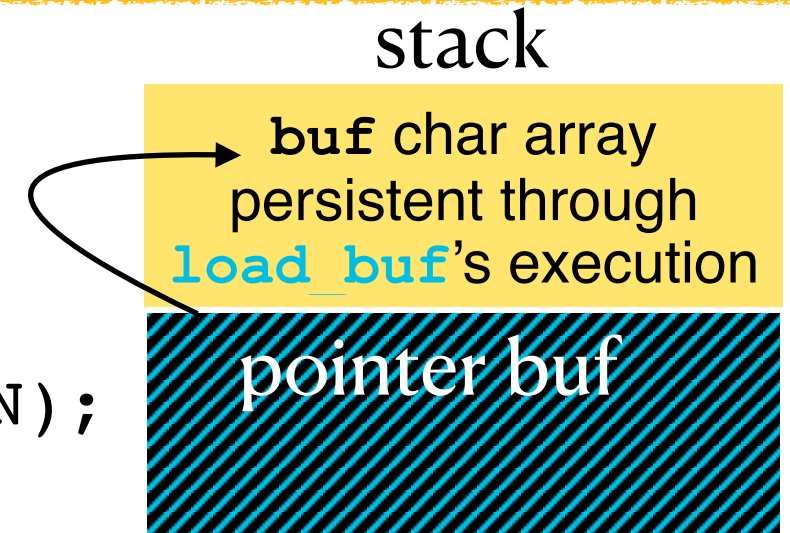
```
funcA() { funcB(); }  
funcB() { funcC(); }  
funcC() { funcD(); }
```



Passing Pointers into the Stack

- It is fine to pass a pointer to stack space further down.

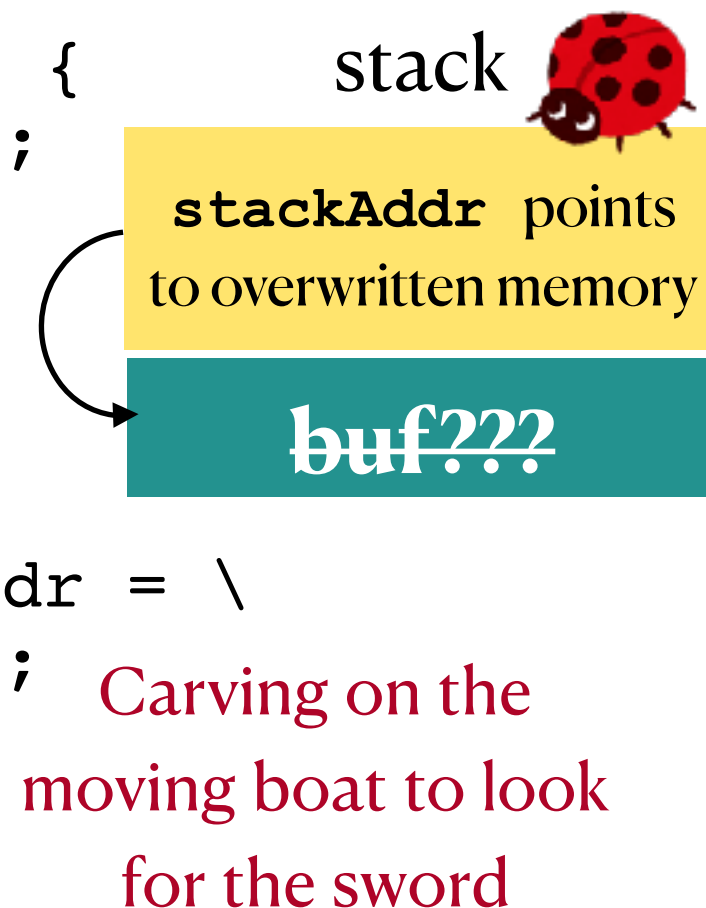
```
#define BUFLen 256
int main() {
    ...
    char buf[BUFLen];
    load_buf(buf, BUFLen);
    ...
}
```



- However, it is bad to return a pointer to something in the **stack**!
- Memory will be overwritten when other functions called!
- So your data would no longer exist, and writes can overwrite key pointers, causing crashes!

```
char *make_buf() {
    char buf[50];
    return buf;
}
```

```
int main() {
    ...
    char *stackAddr = \
        make_buf();
    foo();
    ...
}
```



Solve with slides to come ...

Managing the Heap

- The heap is **dynamic** memory – memory that can be allocated, resized, and freed during program runtime.
 - Useful for persistent memory across function calls
 - But biggest source of pointer bugs, memory leaks, ...
- Large pool of memory, not allocated in contiguous order
 - Back-to-back requests for heap memory could result in blocks very far apart
- C supports four functions for heap management:
 - **malloc()** allocate a block of uninitialized memory
 - **calloc()** allocate a block of zeroed memory
 - **free()** free previously allocated block of memory
 - **realloc()** change size of previously allocated block (might move)
- Read-more: <http://web.archive.org/web/20030222051144/http://home.earthlink.net/~bobbitts/c89.txt> section 4.10.3 memory management functions

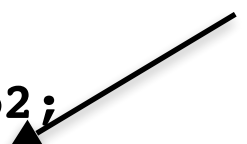
Managing the Heap

- **void *malloc(size_t n):**
 - Allocate a block of uninitialized memory
 - **n** is an integer, indicating size of allocated memory block in bytes
 - **size_t** is an unsigned integer type big enough to “count” memory bytes
 - **sizeof** returns size of given type in bytes, produces more portable code
 - Returns **void*** pointer to block; **NULL** return indicates no more memory; **always check** for return **NULL** (**if (ip)**)
 - Think of pointer as a handle that describes the allocated block of memory; Additional control information stored in the heap around the allocated block! (Including size, etc.)

“Cast” operation, changes type of a variable.

- Examples:

```
int *ip1, *ip2;  
ip1 = (int *) malloc(sizeof(int));  
ip2 = (int *) malloc(20*sizeof(int)); //allocate an array of 20 ints.
```



```
typedef struct { ... } TreeNode;  
TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
```

Assuming size of objects can lead to misleading, unportable code. Use **sizeof()**!

Managing the Heap

- **void free(void *p) :**
 - Releases memory allocated by **malloc()**
 - **p** is pointer containing the address originally returned by **malloc()**

```
int *ip;
ip = (int *) malloc(sizeof(int));
... ..
free((void*) ip); /* Can you free(ip) after ip++ ? */

typedef struct {...} TreeNode;
TreeNode *tp = (TreeNode *) malloc(sizeof(TreeNode));
... ..
free((void *) tp);
```
 - When you free memory, you must be sure that you pass the original address returned from **malloc()** to **free()**; Otherwise, system exception (or worse)!


Managing the Heap

- `void *realloc(void *p, size_t size):`
 - Returns new address of the memory block.
 - In doing so, it may need to copy all data to a new location.
- `realloc(NULL, size); // behaves like malloc`
`realloc(ptr, 0); // behaves like free, deallocates heap block`
- Always check for return NULL

```
int *ip; ip = (int *) malloc(10*sizeof(int));
... .. /* check for NULL */
ip = (int *) realloc(ip, 20*sizeof(int));
/* contents of first 10 elements retained */
... .. /* check for NULL */
realloc(ip, 0); /* equivalent to free(ip); */
```

Keep track of this, since it might change.

Summary

- Code, static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order, avoid “dangling references”
- Managing the heap is tricky: 
 - Memory can be allocated/deallocated at any time
 - “Memory leak”: If you forget to deallocate memory
 - “Use after free”: If you use data after calling free
 - “Double free”: If you call free 2x on same memory
 - Free stack: useless

Using Dynamic Memory—Linked List

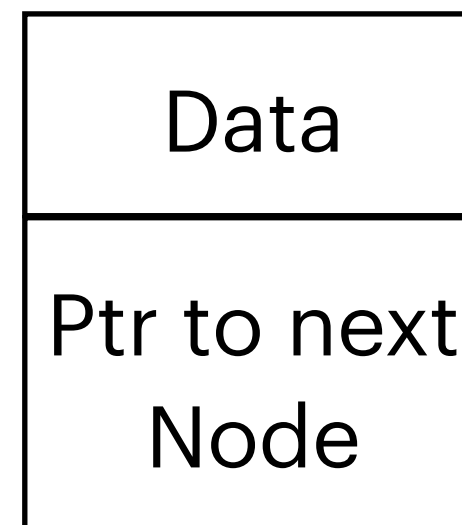
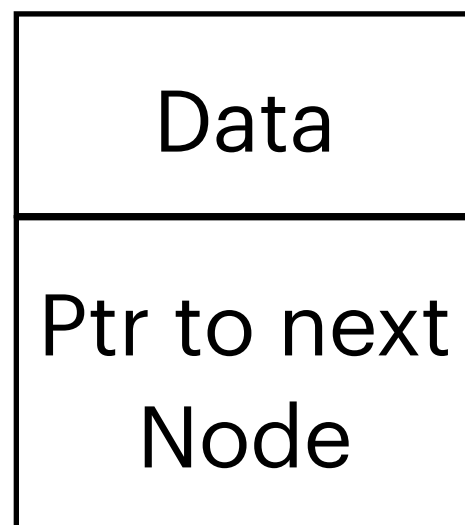
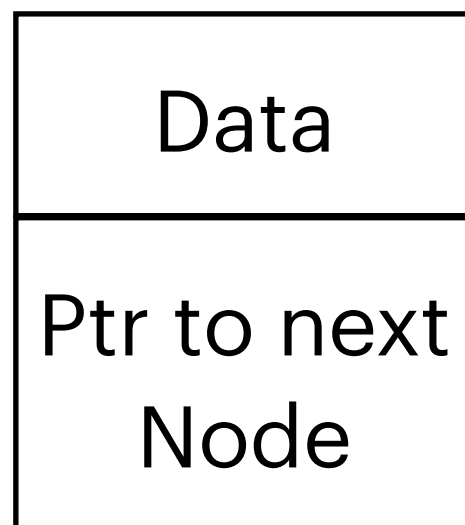
```
typedef struct Node
{
    int val;
    struct Node *next;
} node;
```

```
node * head = NULL;
head = (node *) malloc(sizeof(node));
if(head == NULL){
    return 1;
}
head -> val = 1;
head -> next = NULL;
```

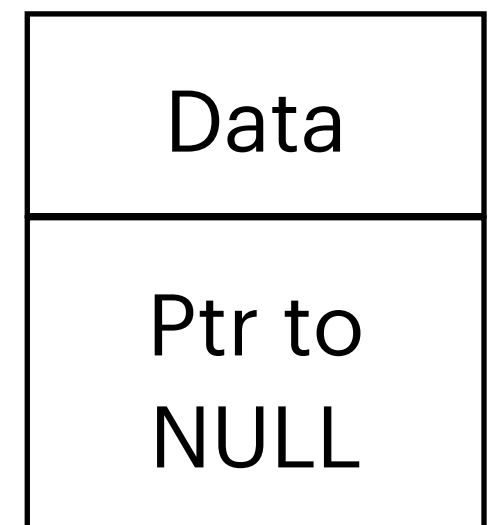
*check malloc
是否成功*

Create the first node

The first node



The last node



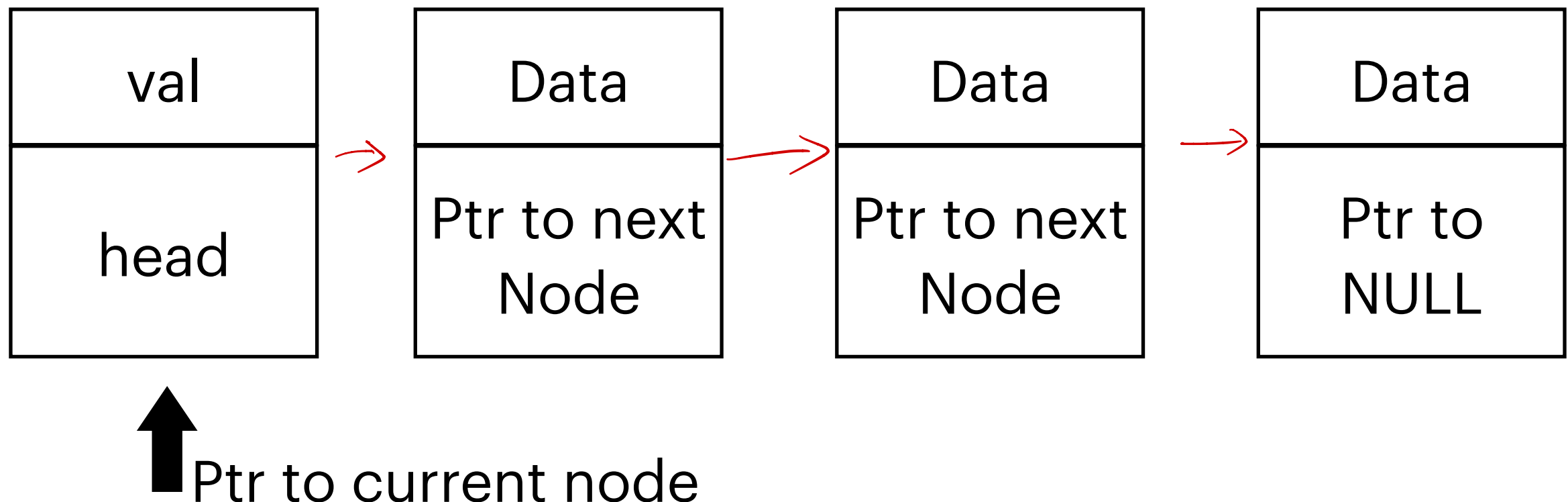
↑
Ptr to head

Using Dynamic Memory—Iterate

```
typedef struct Node
{
    int val;
    struct Node *next;
} node;
```

```
void print_list(node *head){
    node * current = head;
    while (current != NULL){
        printf("%d\t", current -> val);
        current = current -> next;
    }
    printf("\n");
}
```

The first node

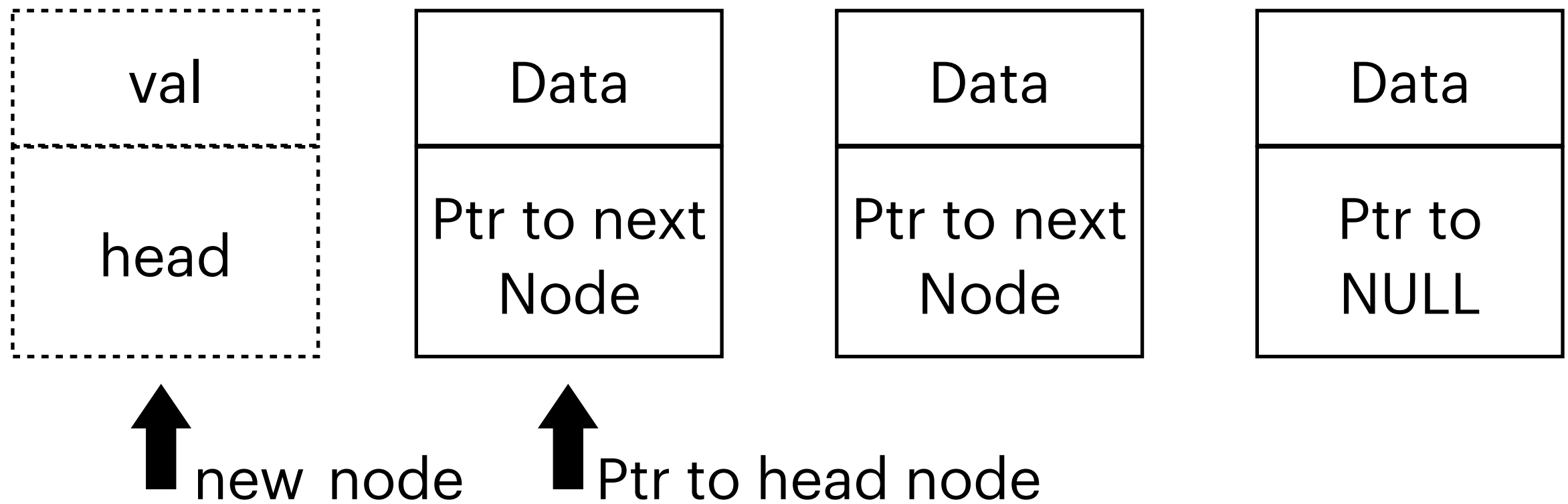


Using Dynamic Memory—Push

```
typedef struct Node
{
    int val;
    struct Node *next;
} node;
```

```
void push_node(node ** head, int val){
    node * new_node;
    new_node = (node *) malloc (sizeof
(node));
    new_node -> val = val;
    new_node -> next = *head;
    *head = new_node;
    printf("Node %d push succeeds!\n",
(*head) -> val);
}
```

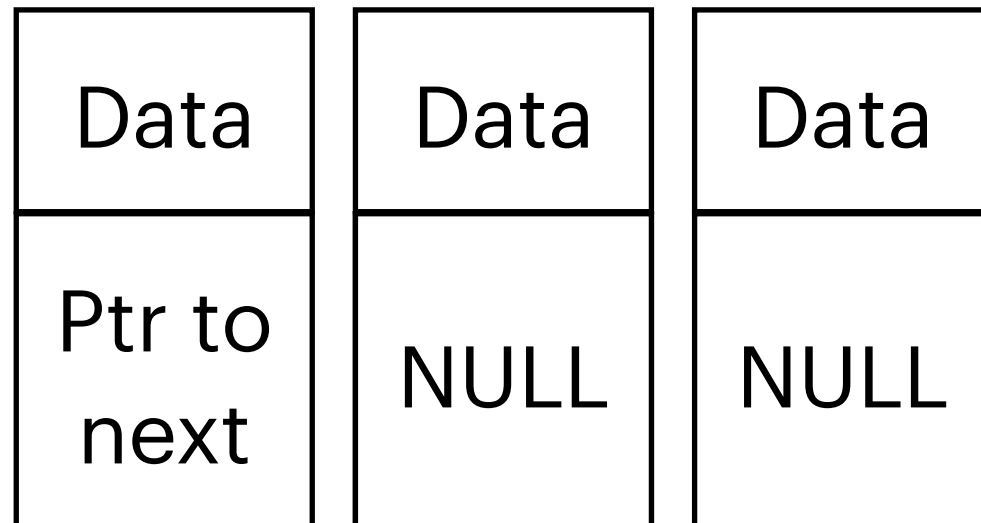
The first node



Using Dynamic Memory—Remove Last

```
typedef struct Node
{
    int val;
    struct Node *next;
} node;
```

The first node



Ptr to cur. node

```
int remove_last(node * head) {
    int retval = 0;

    if (head->next == NULL) {
        retval = head->val;
        free(head);
        return retval;
    }

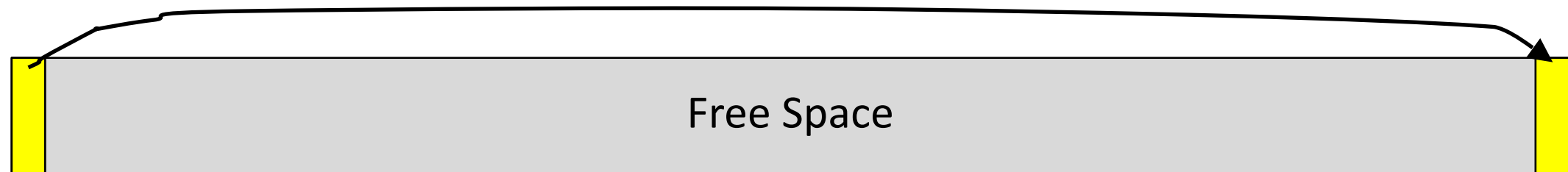
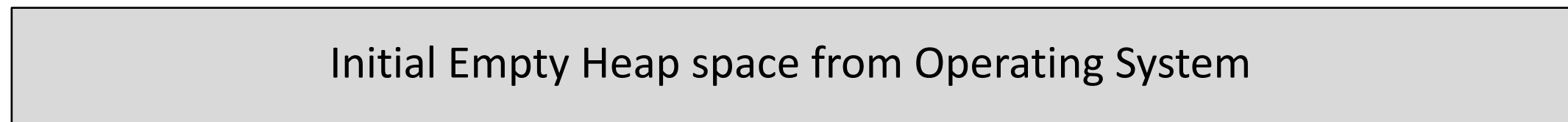
    node * current = head;
    while (current->next->next != NULL) {
        current = current->next;
    }

    retval = current->next->val;
    free(current->next);
    current->next = NULL;
    printf("%d is removed.\n", retval);
    return retval;
}
```

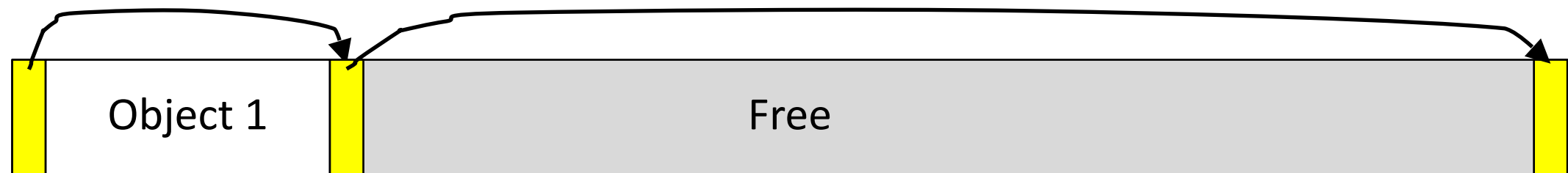
How are Malloc/Free implemented?

- Underlying operating system allows **malloc** library to ask for large blocks of memory to use in heap (e.g., using Unix **sbrk()** call)
- C standard **malloc** library creates data structure inside unused portions to track free space

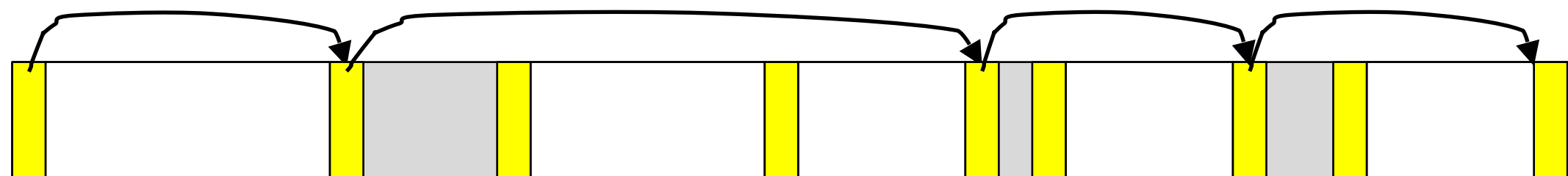
Simple Slow Malloc Implementation



Malloc library creates linked list of empty blocks (one block initially)



First allocation chews up space from start of free space

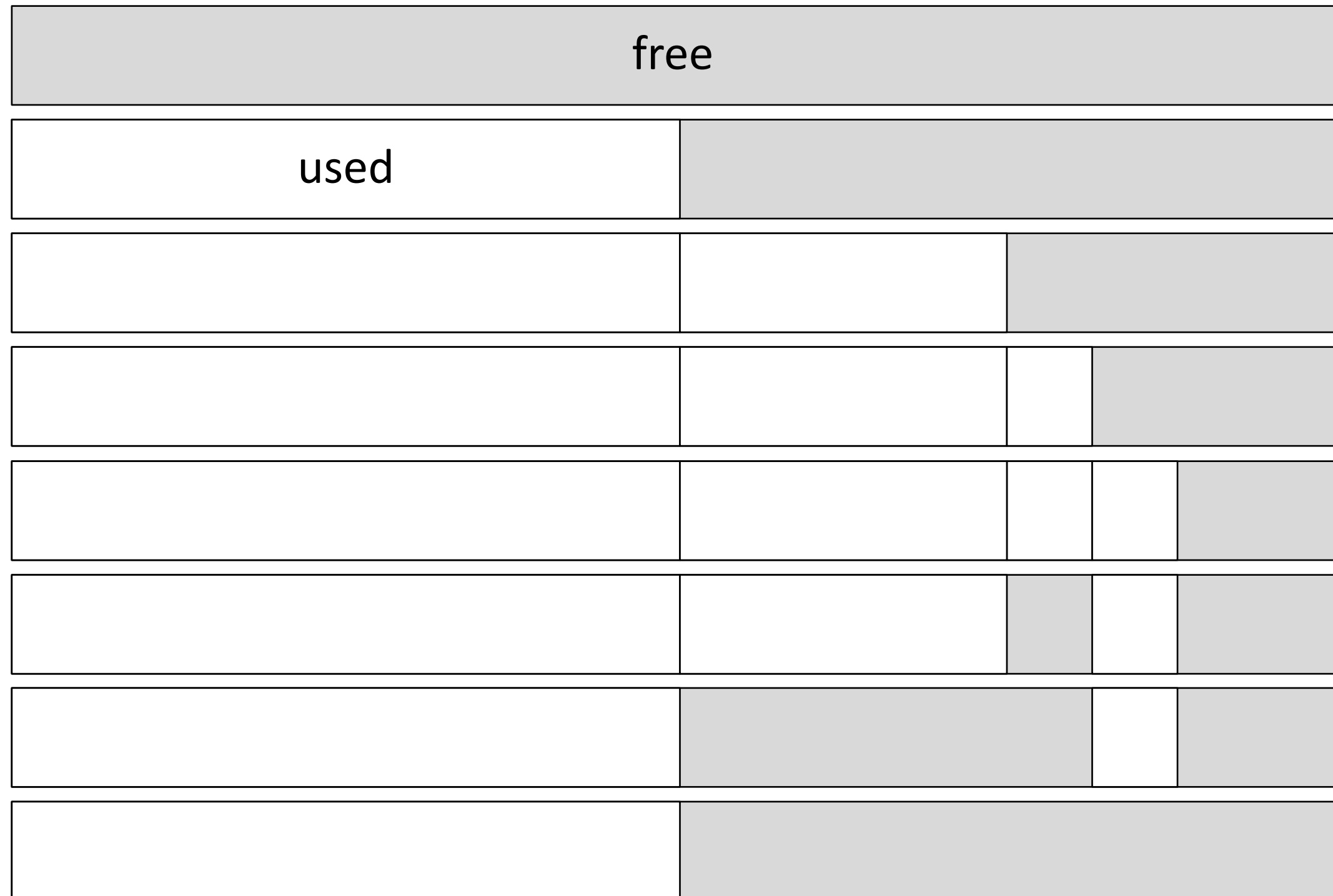


After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list – might merge with neighboring free space

Faster malloc implementations

- Keep separate pools of blocks for different sized objects
- “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:

Power-of-2 “Buddy Allocator”



Malloc Implementations

- All provide the same library interface, but can have radically different implementations
- Uses headers at start of allocated blocks and space in unallocated memory to hold **malloc**'s internal data structures
- Rely on programmer remembering to free with same pointer returned by **malloc**
- Rely on programmer not messing with internal data structures accidentally!

Agenda

- C Memory Management
- C Bugs: covered in discussion this week

Summary

- C has several main memory segments in which to allocate data:
 - Static Data: Variables outside functions/code
 - Stack: Variables local to function
 - Heap: Objects explicitly malloc-ed/free-d.
- Heap data is biggest source of bugs in C code



信息科学与技术学院

School of Information Science and Technology

CS 110

Computer Architecture

Intro to RISC-V I

Instructors:

Siting Liu & Chundong Wang

Course website: <https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/index.html>

School of Information Science and Technology (SIST)

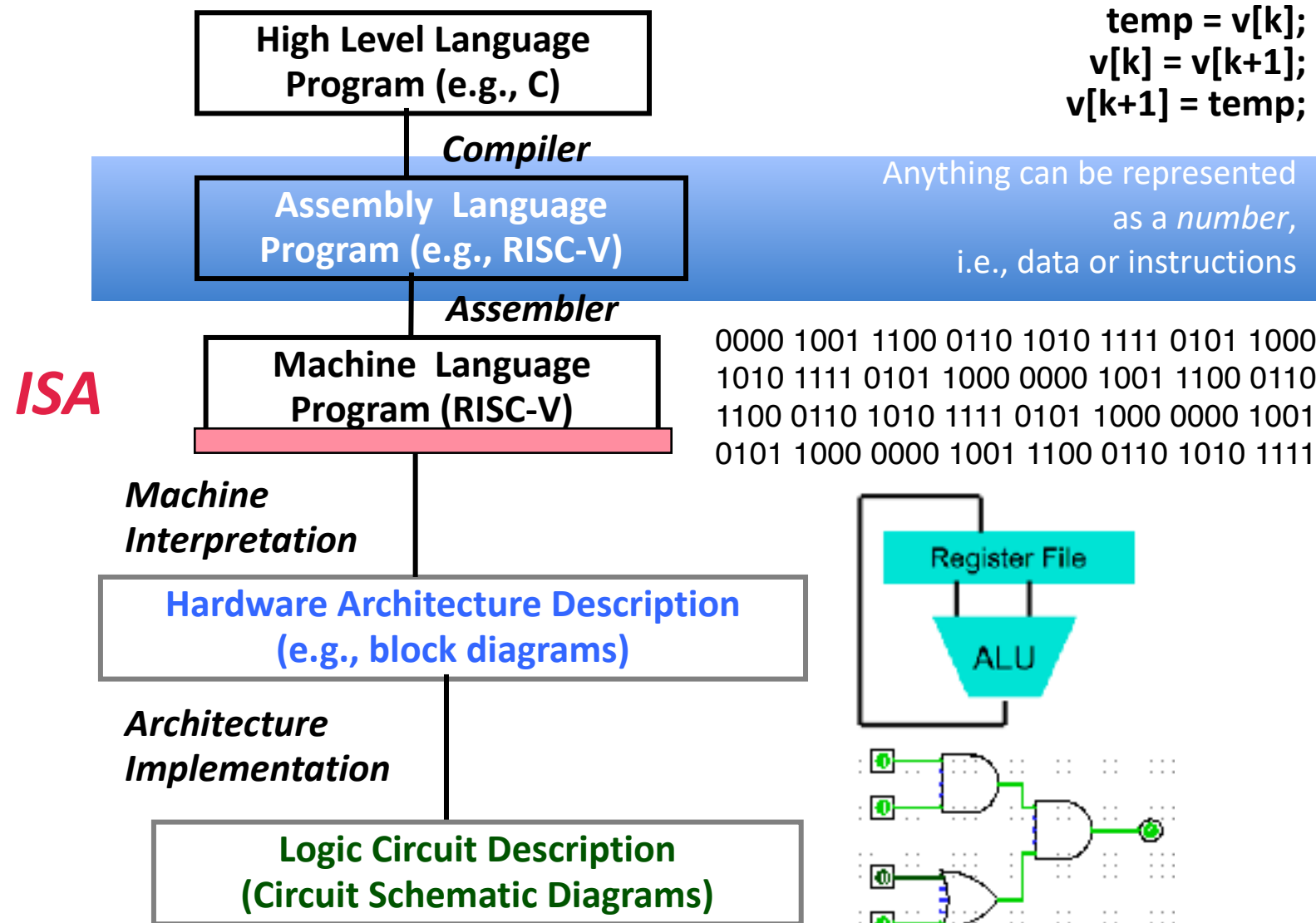
ShanghaiTech University

2023/2/6

Review

- Number representations (Unsigned/Signed)
- How C compiler works
 - C codes are analyzed and break into basic operations
- C usage
 - Pointers & Memory Management
- Overview of Von Neumann Architecture
 - CPU (CA/CC/Registers, etc.) & Memory
- Next introduce how basic operations are implemented
 - **RISC-V Assembly (basic operations can be performed by hardware)**
 - Micro-architecture (hardware, basics on digital circuit)
 - Other number representations (floating-point, IEEE standard 754)

History



**53 years ago:
Apollo Guidance
Computer
programmed in
Assembly**

30x30x30cm, 32 kg.

**10,000 lines of machine
code manually entered –
tons of easter eggs!**

abcnews.go.com/Technology/apollo-11s-source-code-tons-easter-eggs-including/story?id=40515222



**Margaret Hamilton with the
code she wrote.**

Intro to ISA

- Part of the **abstract model** of a computer that defines how the CPU is controlled by the software; **interface** between the hardware and the software;
- **Programmers' manual** because it is the portion of the machine that is visible to the **assembly language programmers**, the compiler writers, and the application programmers.
- Defines the supported data types, the registers, how the hardware manages main memory, key features, **instructions that can be executed (instruction set)**, and the input/output model of multiple ISA implementations
- ISA can be extended by adding instructions or other capabilities

-by ARM

Instruction Set Architecture

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
 - VAX architecture had an instruction to multiply polynomials!
- RISC philosophy (John Cocke IBM, John Hennessy Stanford, David Patterson Berkeley, 1980s)
- Hennessy & Patterson won ACM A.M. Turing Award

Reduced Instruction Set Computing (RISC)

- Keep the instruction set small and simple, makes it easier to build fast hardware.
- Let software do complicated operations by composing simpler ones.

Mainstream ISAs

X86/AMD64	ARM	RISC-V
CISC	RISC	RISC
Fees for ISA (Limited)	Fees for ISA	No fees for ISA
Fees for micro- architecture (Limited)	Fees for micro- architecture	Depending on usage (commercial vs. open-source)
A lot of historical burden	Relatively simple	Simple & can DIY, expandable
Intel/AMD	ARM	non-profit RISC-V foundation

RISC vs. CISC

Disassembly of section __TEXT,__text:

0000000000000000 <ltmp0>:

```

0: ff c3 00 d1 : sub sp, sp, #48
4: fd 7b 02 a9 : stp x29, x30, [sp, #32]
8: fd 83 00 91 : add x29, sp, #32
c: 08 00 80 52 : mov w8, #0
10: e8 0f 00 b9 : str w8, [sp, #12]
14: bf c3 1f b8 : stur wzr, [x29, #-4]
18: 48 9a 80 52 : mov w8, #1234
1c: a8 83 1f b8 : stur w8, [x29, #-8]
20: 28 1c 82 52 : mov w8, #4321
24: a8 43 1f b8 : stur w8, [x29, #-12]
28: a8 83 5f b8 : ldur w8, [x29, #-8]
2c: a9 43 5f b8 : ldur w9, [x29, #-12]
30: 08 01 09 0b : add w8, w8, w9
34: e8 13 00 b9 : str w8, [sp, #16]
38: e9 13 40 b9 : ldr w9, [sp, #16]
3c: e8 03 09 aa : mov x8, x9
40: e9 03 00 91 : mov x9, sp
44: 28 01 00 f9 : str x8, [x9]
48: 00 00 00 90 : adrp x0, 0x0 <ltmp0+0x48>
4c: 00 00 00 91 : add x0, x0, #0
50: 00 00 00 94 : bl 0x50 <ltmp0+0x50>
54: e0 0f 40 b9 : ldr w0, [sp, #12]
58: fd 7b 42 a9 : ldp x29, x30, [sp, #32]
5c: ff c3 00 91 : add sp, sp, #48
60: c0 03 5f d6 : ret

```

定长

Assembly

Compiled on Mac machine using ARM CPU

0000000000000054 <main>:

```

54: 55 : push %rbp
55: 48 89 e5 : mov %rsp,%rbp
58: 48 83 ec 30 : sub $0x30,%rsp
5c: e8 00 00 00 00 : call 61 <main+0xd>
61: c7 45 fc d2 04 00 00 : movl $0x4d2,-0x4(%rbp)
68: c7 45 f8 e1 10 00 00 : movl $0x10e1,-0x8(%rbp)
6f: 8b 55 fc : mov -0x4(%rbp),%edx
72: 8b 45 f8 : mov -0x8(%rbp),%eax
75: 01 d0 : add %edx,%eax
77: 89 45 f4 : mov %eax,-0xc(%rbp)
7a: 8b 45 f4 : mov -0xc(%rbp),%eax
7d: 89 c2 : mov %eax,%edx
7f: 48 8d 05 00 00 00 00 : lea 0x0(%rip),%rax # 86 <main+0x32>
86: 48 89 c1 : mov %rax,%rcx
89: e8 72 ff ff ff : call 0 <printf>
8e: b8 00 00 00 00 : mov $0x0,%eax
93: 48 83 c4 30 : add $0x30,%rsp
97: 5d : pop %rbp
98: c3 : ret
99: 90 : nop
9a: 90 : nop
9b: 90 : nop
9c: 90 : nop
9d: 90 : nop
9e: 90 : nop
9f: 90 : nop

```

Assembly

Compiled on Windows machine using Intel CPU

More than 3,100 RISC-V Members

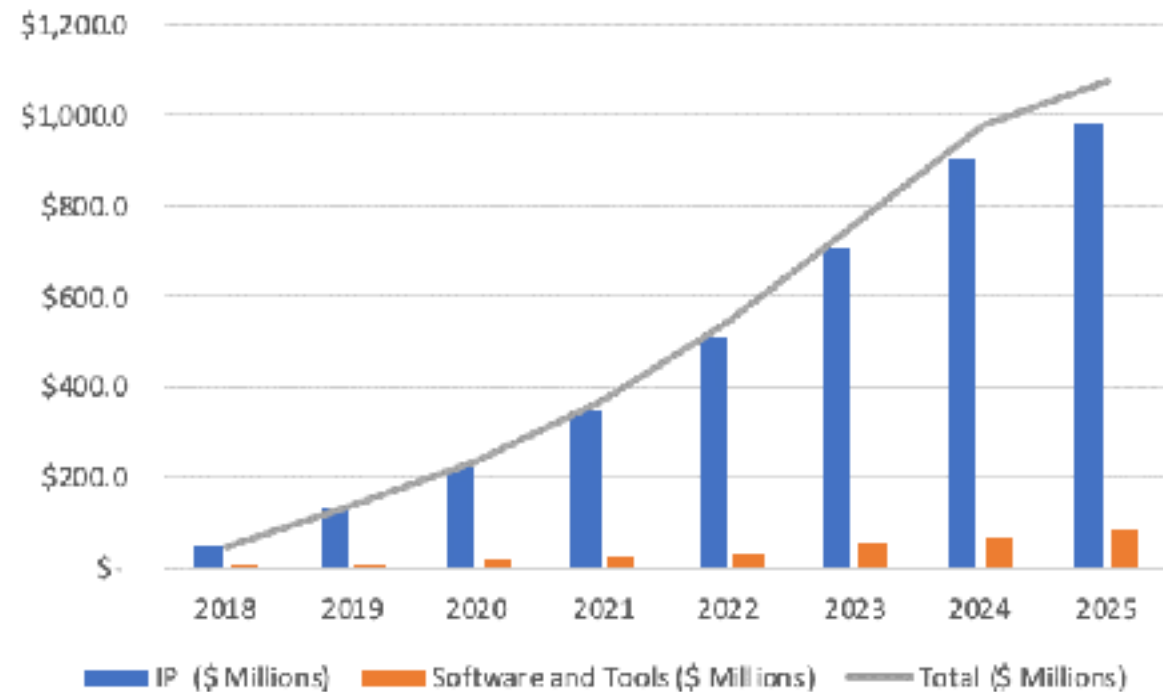
- Alibaba Cloud: T-Head 玄铁 C series; E series, and R series
- Huawei: Hi3861V100 SoC for IoT/smart home
- Tencent: recently become a premier member
- Intel, Google, Meta, SiFive, AMD/Xilinx, etc.

ShanghaiTech hold several RISC-V Summits China recent years!

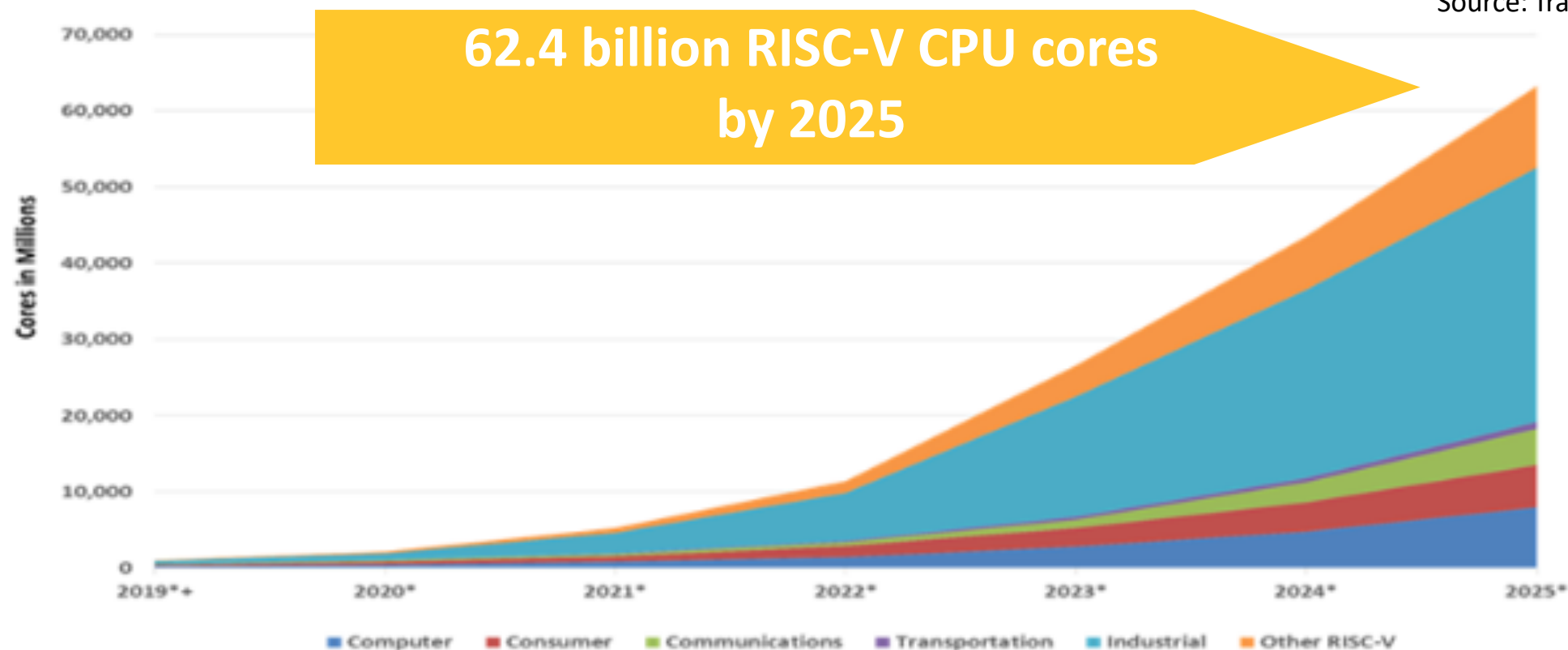
- Can Linux OS work on RISC-V CPU?

More than 3,100 RISC-V Members

The total market for RISC-V IP and Software is expected to grow to \$1.07 billion by 2025 at a CAGR of 54.1%



Source: Tractica

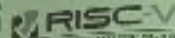


Assembly Language

- Basic job of a CPU: execute lots of instructions.
- Instructions are the primitive operations that the CPU may execute.
 - Other examples: MIPS, IBM/Motorola PowerPC (quite old Mac), Intel IA64, ...

Why RISC-V in CS110?

- Why RISC-V instead of Intel x86?
 - RISC-V is simple, elegant. Don't want to get bogged down in gritty details.
- It is a very very clean RISC
 - No real additional "optimizations"
- Generally only one way to do any particular thing
- https://toast-lab.sist.shanghaitech.edu.cn/courses/CS110@ShanghaiTech/Spring-2023/lecture_notes/riscvcard.pdf



RISC-V

Reference Manual

UNOFFICIAL TRANSLATION OF THE
RISC-V SPECIFICATION




Table of Contents

1 Introduction

2 Architecture Overview

3 Instruction Set Architecture

4 Base Integer Instruction Set

5 Base Floating-Point Instruction Set

6 Base Atomic Instruction Set

7 Base Cryptographic Instruction Set

8 Base Vector Instruction Set

9 Base Neural Network Instruction Set

10 Base Machine Learning Instruction Set

11 Base Machine Learning Extension

12 Base Machine Learning Extension

13 Base Machine Learning Extension

14 Base Machine Learning Extension

15 Base Machine Learning Extension

16 Base Machine Learning Extension

17 Base Machine Learning Extension

18 Base Machine Learning Extension

19 Base Machine Learning Extension

20 Base Machine Learning Extension

21 Base Machine Learning Extension

22 Base Machine Learning Extension

23 Base Machine Learning Extension

24 Base Machine Learning Extension

25 Base Machine Learning Extension

26 Base Machine Learning Extension

27 Base Machine Learning Extension

28 Base Machine Learning Extension

29 Base Machine Learning Extension

30 Base Machine Learning Extension

31 Base Machine Learning Extension

32 Base Machine Learning Extension

33 Base Machine Learning Extension

34 Base Machine Learning Extension

35 Base Machine Learning Extension

36 Base Machine Learning Extension

37 Base Machine Learning Extension

38 Base Machine Learning Extension

39 Base Machine Learning Extension

40 Base Machine Learning Extension

41 Base Machine Learning Extension

42 Base Machine Learning Extension

43 Base Machine Learning Extension

44 Base Machine Learning Extension

45 Base Machine Learning Extension

46 Base Machine Learning Extension

47 Base Machine Learning Extension

48 Base Machine Learning Extension

49 Base Machine Learning Extension

50 Base Machine Learning Extension

51 Base Machine Learning Extension

52 Base Machine Learning Extension

53 Base Machine Learning Extension

54 Base Machine Learning Extension

55 Base Machine Learning Extension

56 Base Machine Learning Extension

57 Base Machine Learning Extension

58 Base Machine Learning Extension

59 Base Machine Learning Extension

60 Base Machine Learning Extension

61 Base Machine Learning Extension

62 Base Machine Learning Extension

63 Base Machine Learning Extension

64 Base Machine Learning Extension

65 Base Machine Learning Extension

66 Base Machine Learning Extension

67 Base Machine Learning Extension

68 Base Machine Learning Extension

69 Base Machine Learning Extension

70 Base Machine Learning Extension

71 Base Machine Learning Extension

72 Base Machine Learning Extension

73 Base Machine Learning Extension

74 Base Machine Learning Extension

75 Base Machine Learning Extension

76 Base Machine Learning Extension

77 Base Machine Learning Extension

78 Base Machine Learning Extension

79 Base Machine Learning Extension

80 Base Machine Learning Extension

81 Base Machine Learning Extension

82 Base Machine Learning Extension

83 Base Machine Learning Extension

84 Base Machine Learning Extension

85 Base Machine Learning Extension

86 Base Machine Learning Extension

87 Base Machine Learning Extension

88 Base Machine Learning Extension

89 Base Machine Learning Extension

90 Base Machine Learning Extension

91 Base Machine Learning Extension

92 Base Machine Learning Extension

93 Base Machine Learning Extension

94 Base Machine Learning Extension

95 Base Machine Learning Extension

96 Base Machine Learning Extension

97 Base Machine Learning Extension

98 Base Machine Learning Extension

99 Base Machine Learning Extension

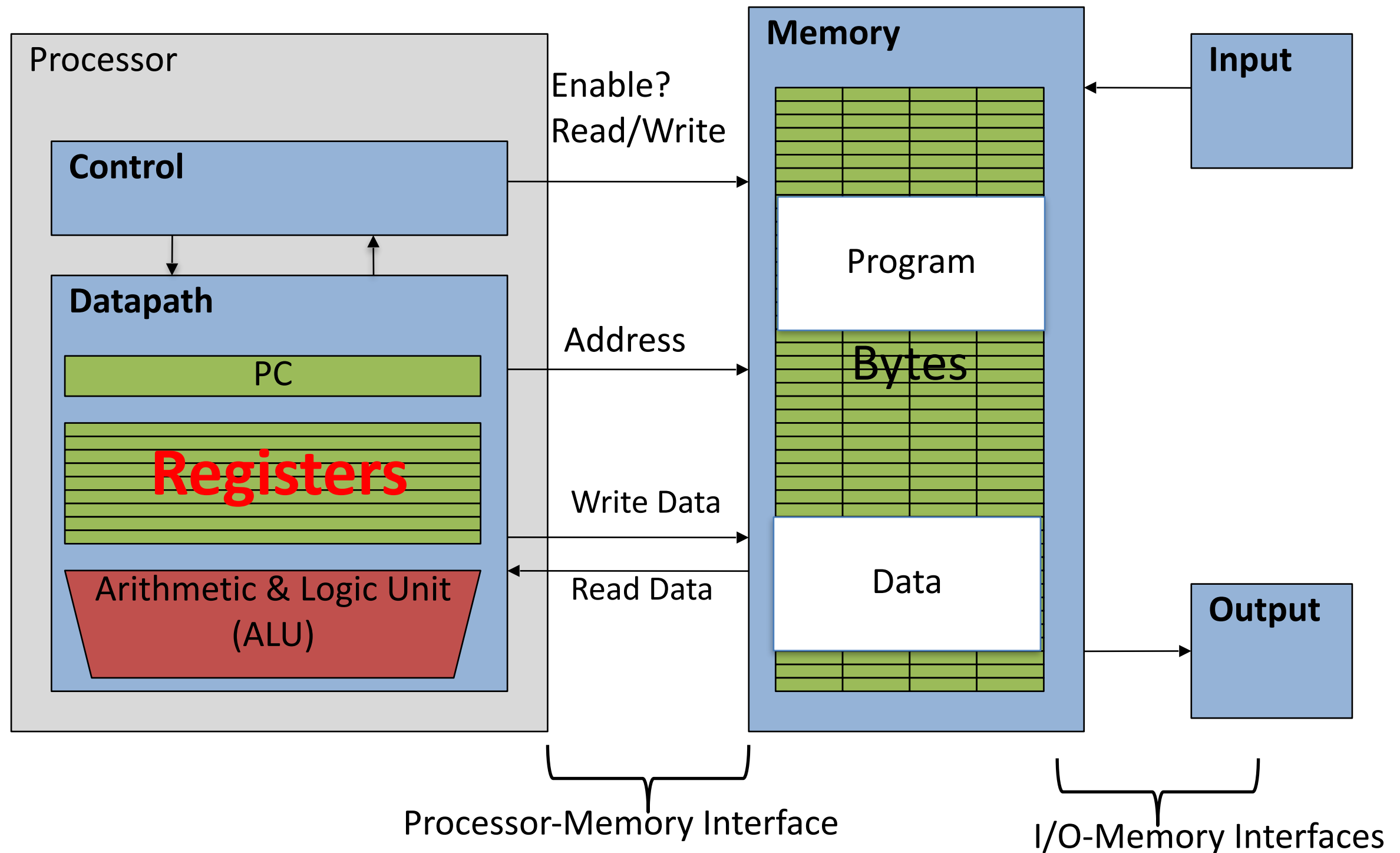
100 Base Machine Learning Extension

RISC-V Green Card

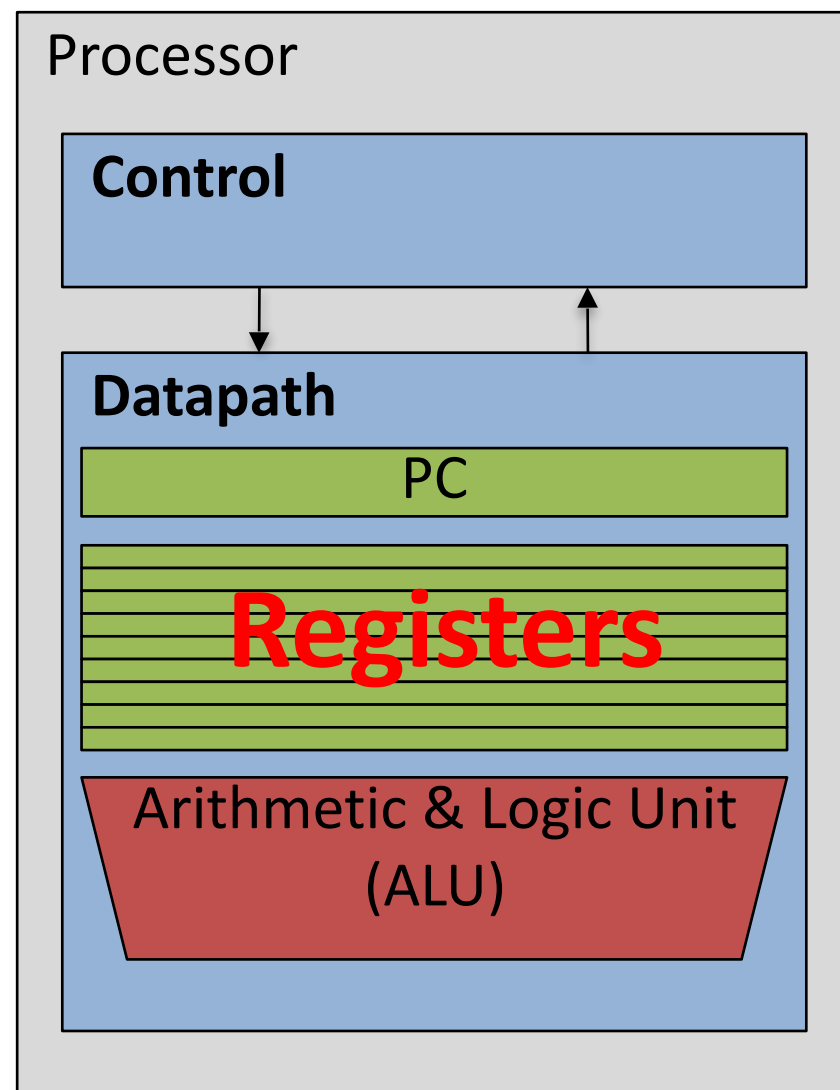
Assembly Registers (hardware/variable)

- Unlike C or Java, assembly cannot use variables
 - Keep assembly/computer hardware abstract simple
- Assembly operands are registers
 - Limited number of special locations/memory built directly into the CPU
 - Operations can only be performed on these registers in RISC-V
- Benefit: Since registers are directly in hardware (CPU), they are very fast

Registers, inside the Processor



Registers, inside the Processor

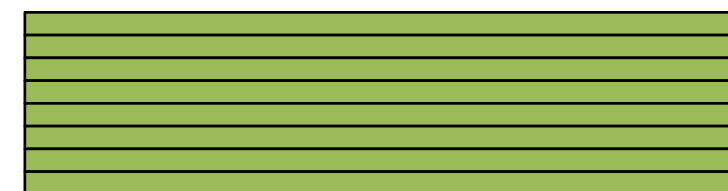


Registers



x0始终为0
 x0 / zero
 x1
 x2

...



...

x31

- Similar to memory, use “address” to refer to specific location



- Hold address of the current instruction

- 32 registers in RISC-V (in RV32 variant)
 - Why 32? Smaller is faster, but too small is bad.

寄存器共32个, 每个32位

- Each RISC-V register is 32 bits wide (in RV32 variant)
 - Groups of 32 bits called a word in RV32; P&H textbook uses 64-bit variant RV64 (doubleword)

```
1c: a8 83 1f b8  stur w8, [x29, #-8]
20: 28 1c 82 52  mov w8, #4321
24: a8 43 1f b8  stur w8, [x29, #-12]
28: a8 83 5f b8  ldur w8, [x29, #-8]
2c: a9 43 5f b8  ldur w9, [x29, #-12]
30: 08 01 09 0b  add w8, w8, w9
```

RISC-V Manual, RTFM

- <https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf>
- <https://github.com/riscv-non-isa/riscv-asm-manual/blob/master/riscv-asm.md>

Base	Version	Frozen?
RV32I	2.0	Y
RV32E	1.9	N
RV64I	2.0	Y
RV128I	1.7	N
Extension	Version	Frozen?
M	2.0	Y
A	2.0	Y
F	2.0	Y
D	2.0	Y
Q	2.0	Y
L	0.0	N
C	2.0	Y
B	0.0	N
J	0.0	N
T	0.0	N
P	0.1	N
V	0.2	N
N	1.1	N

Number indicates address/pointer/register width
I: Integer (integer arith., load, store and control-flow instructions)

M: Integer multiplication & division extension

A: Atomic instruction (read-modify-write)

F: single-precision floating-point (FP) extension (FP registers/arith./load/store)

D: double-precision ... (similar to F, with more bits)

RV32 + IMAFD extension = RV32G

RV64 + IMAFD extension = RV64G

C, Java variables vs. registers

- In C (and most High Level Languages) variables declared first 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 as (cannot mix and match int and char variables).
- In Assembly Language, registers have no type, **simply stores 0s and 1s**; operation determines how register contents are treated (think about the hardware)

Assembly Instructions

- In assembly language, each statement (called an instruction), executes exactly one of a short list of simple commands
- Unlike in C (and most other High Level Languages), each line of assembly code contains at most 1 instruction
- Another way to make your code more readable: comments!
- Hash (#) is used for RISC-V comments
 - anything from hash mark to end of line is a comment and will be ignored

Assembly Instructions

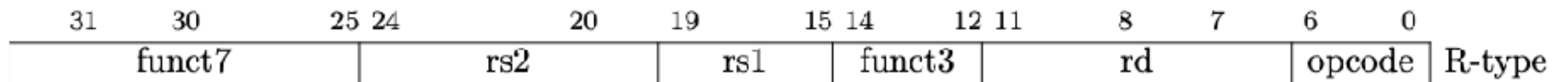
- Different types of instructions (4 core types + B/J based on the handling of immediate)

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		

- Different types have different format but “rs1”, “rs2” and “rd” are at the same position (hardware friendly)
- As an ID number, the machine code of the instructions has different fields; format depends on their operands/type

Assembly Instructions

- Different types of instructions (4 core types + B/J based on the handling of immediate)



- R-type
 - Register-register operation, mainly for arithmetic & logic
 - Has two operands (accessed from the source registers, rs1 & rs2) and one output (saved to the destination register, rd)
 - Cannot access main memory (instruction executed by CPU alone, no data exchange with main memory)

RV32I R-type Arithmetic

- Syntax of instructions: assembly language, two register operands

- Addition: `add rd, rs1, rs2` (operation `rd,rs1,rs2`)

Adds the value stored in register `rs1` to that of `rs2` and stores the sum into register `rd`, similar to $a = b + c$, $a \Leftrightarrow \text{rd}$, $b \Leftrightarrow \text{rs1}$, $c \Leftrightarrow \text{rs2}$

- Example: `add x5, x2, x1` $x_5 = x_2 + x_1$

`add x6, x0, x5`

`add x4, x1, x3`

$$\begin{array}{r}
 x_1 \quad 0x12340000 \\
 + x_3 \quad 0xFFFFFFFF \\
 \hline
 x_4 \quad 0x1233FFFF
 \end{array}$$

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x1233FFFF	x4
0x12346789	x5
0x12346789	x6
	x7

RV32I R-type Arithmetic

- Syntax of instructions: assembly language, two register operands

- Subtraction: `sub rd, rs1, rs2`

Subtract the value stored in register `rs2` from that of `rs1` and stores the difference into register `rd`, equivalent to $a = b - c$, $a \Leftrightarrow \text{rd}$, $b \Leftrightarrow \text{rs1}$, $c \Leftrightarrow \text{rs2}$

- Example: `sub x5, x2, x1` $x_5 = x_2 - x_1$
`sub x6, x0, x5`

Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
	x3
	x4
	x5
	x6
	x7

RV32I R-type Logic Operation

- Syntax of instructions: assembly language, two register operands

- AND/OR/XOR: `and/or/xor rd, rs1, rs2`

Logically **bit**-wise and/or/xor the value stored in register `rs1` and that of `rs2` and stores the result into register `rd`, equivalent to $a = b \text{ (&/||/^)} c, a \Leftrightarrow \text{rd}, b \Leftrightarrow \text{rs1}, c \Leftrightarrow \text{rs2}$

- Example: `and x5, x2, x1`
`xor x6, x1, x5`
`and x4, x1, x3`

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
	x4
	x5
	x6
	x7

RV32I R-type Logic Operation

- Syntax of instructions: assembly language, two register operands

- AND/OR/XOR: `and/or/xor rd, rs1, rs2`

Logically **bit**-wise and/or/xor the value stored in register `rs1` and that of `rs2` and stores the result into register `rd`, equivalent to $a = b \ (\&/\|/\wedge) \ c, a \Leftrightarrow rd, b \Leftrightarrow rs1, c \Leftrightarrow rs2$

- Used for bit-mask

`and x5, x7, x4`

`or x6, x7, x4` *$x_6 = x_7 \text{ or } x_4$*

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0xFFFF0000	x4
	x5
<i>0xFFFF5678</i>	x6
0x12345678	x7

RV32I R-type Compare

- Syntax of instructions: assembly language, two register operands
 - SLT/SLTU: ^(signed) slt/^(unsigned) sltu rd, rs1, rs2
- Compare the value stored in register rs1 and that of rs2, sets rd=1, if rs1 < rs2 otherwise rd=0, equivalent to $a = b < c ? 1 : 0$, $a \Leftrightarrow \text{rd}$, $b \Leftrightarrow \text{rs1}$, $c \Leftrightarrow \text{rs2}$. Treat the numbers as signed/unsigned with slt/sltu.
- Handwritten notes:* $rs_1 < rs_2 \Rightarrow rd = 1$
else $\Rightarrow rd = 0$

- Example:


```

      slt x5, x2, x1
      slt x4, x3, x1
      sltu x5, x3, x1
      
```

Handwritten notes: 2^{32}

- Overflow detection (unsigned)

add x5, x3, x3

sltu x6, x5, x3

- Overflow detection (signed)?

- Try yourself/RTFM

Handwritten notes: 0001

Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x1	x4
0x1 \Rightarrow 0x0	x5
	x6
	x7

RV32I R-type Shift

shift left logic - right -

rs1 移 rs2 位, 存 rd 中

- Syntax of instructions: assembly language, two register operands

- Shift left/right (arithmetic): `sll/srl/sra rd, rs1, rs2`

Left/Right shifts the value stored in register `rs1` by that of `rs2`, equivalent to $a = b \ll / \gg c$, $a \Leftrightarrow \text{rd}$, $b \Leftrightarrow \text{rs1}$, $c \Leftrightarrow \text{rs2}$. Arithmetic: sign extended.

- Example: `sll x5, x2, x4`
`srl x6, x3, x4`
`sra x7, x3, x4`

Shift right arithmetic

空位按最高位填充

Registers

0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
0xFFFFFFFF	x7

1000 1001

← sll 1

0001 0010

srl 1

0100 0100

sra 1

1100 0100

RV32I R-type Shift

- Syntax of instructions: assembly language, two register operands

- Shift left/right (arithmetic): `sll/srl/sra rd, rs1, rs2`

Left/Right shifts the value stored in register `rs1` by that of `rs2`, equivalent to $a = b \ll / \gg c$, $a \Leftrightarrow \text{rd}$, $b \Leftrightarrow \text{rs1}$, $c \Leftrightarrow \text{rs2}$.

arithmetic: sign extended.

- Example: `sll x5, x2, x4`
`srl x6, x1, x4`
`sra x7, x3, x4`

- What is the arithmetic effect by shifting?

Registers	
0	x0/zero
0x12340000	x1
0x00006789	x2
0xFFFFFFFF	x3
0x4	x4
	x5
	x6
	x7