CS 110
Computer Architecture

Lecture 4: *Intro to Assembly Language, RISC-V Intro*

Instructors:
Sören Schwertfeger & Chundong Wang

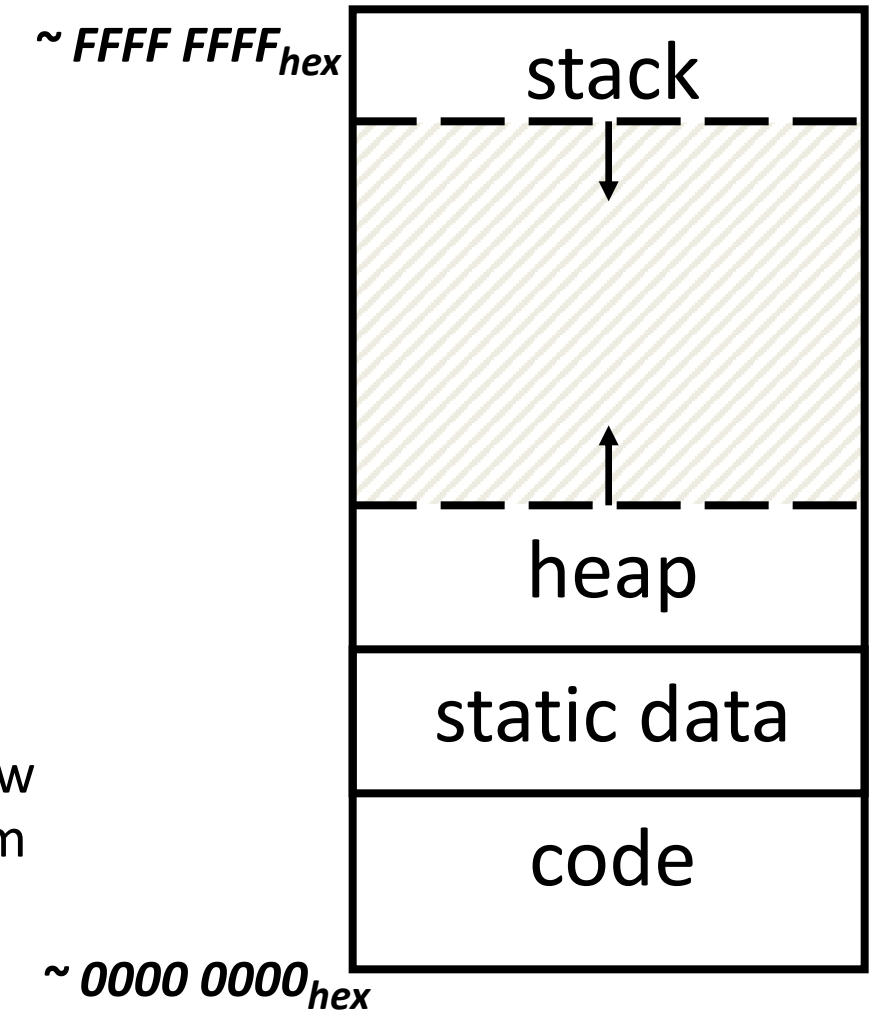
School of Information Science and Technology SIST

ShanghaiTech University

Slides based on UC Berkley's CS61C

C Memory Management

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



Managing the Heap

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
  - careful – it might move!

Malloc()

- **`void *malloc(size_t n)`**:
  - Allocate a block of uninitialized memory
  - NOTE: Subsequent calls might not yield blocks in contiguous addresses
  - **`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
  - Think of pointer as a *handle* that describes the allocated block of memory; Additional control information stored in the heap around the allocated block!

"Cast" operation, changes type of a variable.
Here changes ***`(void *)`*** *to* ***`(int *)`***

- Examples:

```
int *ip;
ip = (int *) malloc(sizeof(int));

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

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 insufficient free memory, **`malloc()`** returns **`NULL`** pointer; **Check for it!**

```
if ((ip = (int *) malloc(sizeof(int))) == NULL){
    printf("\nMemory is FULL\n");
    exit(1); /* Crash and burn! */
}
```

  - When you free memory, you must be sure that you pass the **original address** returned from **`malloc()`** to **`free()`**; Otherwise, system exception (or worse)!

Observations

- 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
- *Managing the heap is tricky*: memory can be allocated / deallocated at any time

```c
#include <libc.h>

/* Takes a string and makes it awesome! */
int make_ca(char * str, size_t length){

    char awesome[] = "CA is so awesome!";

    /* if str is too small we need to get more memory! */
    if(length < strlen(awesome) ){
        str = malloc(sizeof(char) * strlen(awesome));
    }

    strcpy(str, awesome);
}

int main(int argc, char *argv[]){

    char ca[] = "CA is OK.";
    char * CA = malloc(6);
    memcpy(CA, ca, strlen(ca));

    make_ca(ca, strlen(ca));
    make_ca(CA, strlen(CA));
    /* We want to print an awesome string! */
    printf(" %s %s ",ca, CA);

}
```

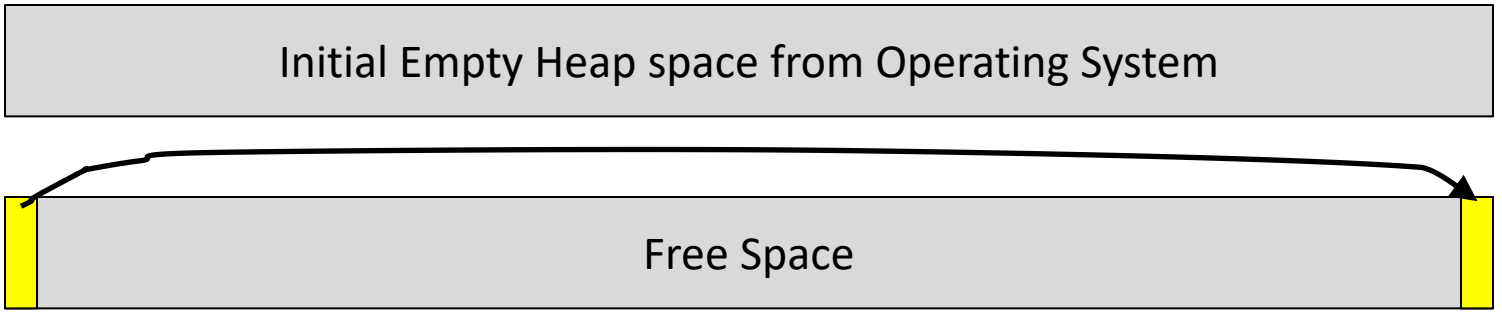
Bugs

- Line 9: comparison with strlen instead of sizeof (for 0-terminator)
- Line 10: strlen instead of sizeof (or +1) for malloc =>
  - Line 13: write past end of array (if malloc was used)
- Line 4: Ownership of pointer str not clear =>
  - Line 10: Potential memory leak
- Line 4: New pointer is not returned/ no pointer to pointer is used
- Line 20: memcpy over length of CA
- Line 20: 0-terminator is not copied!
- Line 22 &23: better: call with array size
- Line 14 & 27: return missing!

How are Malloc/Free implemented?

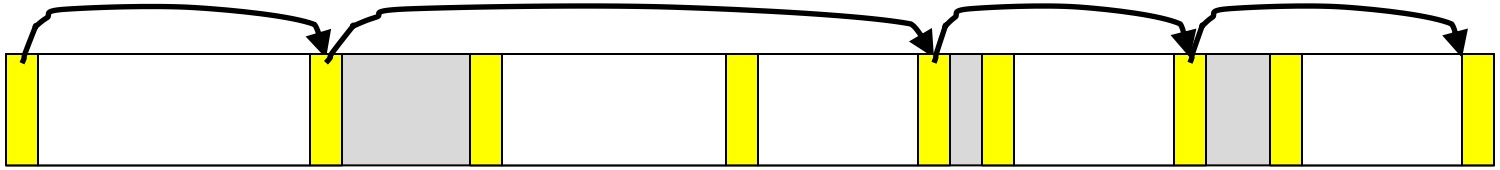
- Underlying operating system allows **`malloc`** library to ask for large blocks of memory to use in heap (e.g., using Unix **`sbrk()`** call)
- C standard **`malloc`** library creates data structure inside unused portions to track free space

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

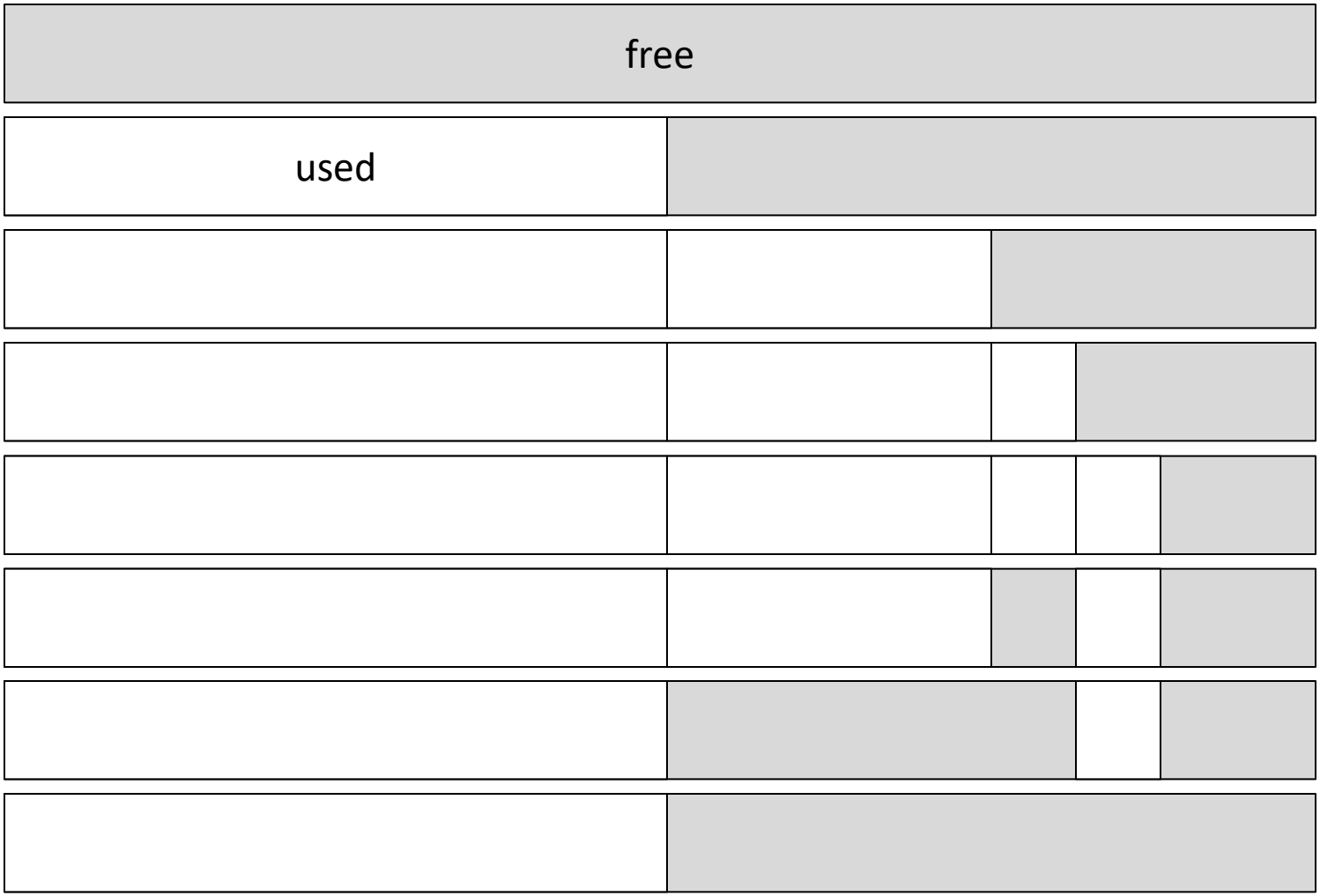


After many mallocs and frees, have potentially long linked list of odd-sized blocks
Frees link block back onto linked list – might merge with neighboring free space

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!

Levels of Representation/Interpretation

High Level Language Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

Compiler

Assembly Language Program (e.g., RISC-V)

```
lw    xt0, 0(x2)
lw    xt1, 4(x2)
sw    xt1, 0(x2)
sw    xt0, 4(x2)
```

Anything can be represented as a *number*, i.e., data or instructions

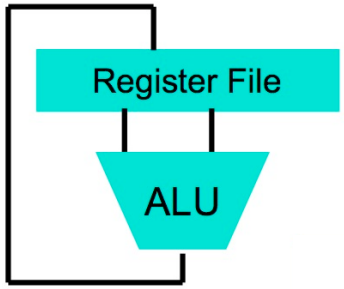
Assembler

Machine Language Program (RISC-V)

```
0000 1001 1100 0110 1010 1111 0101 1000
1010 1111 0101 1000 0000 1001 1100 0110
1100 0110 1010 1111 0101 1000 0000 1001
0101 1000 0000 1001 1100 0110 1010 1111
```

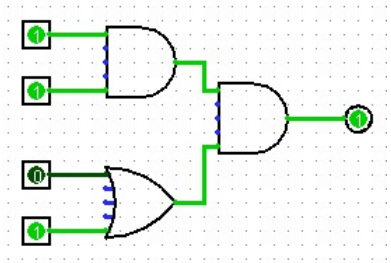
Machine Interpretation

Hardware Architecture Description (e.g., block diagrams)



Architecture Implementation

Logic Circuit Description (Circuit Schematic Diagrams)



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.

- Lead Apollo flight software designer.
- Came up with the idea of naming the discipline, "software engineering"
- https://en.wikipedia.org/wiki/Margaret_Hamilton_%28scientist%29

```
179             TC      BANKCALL        # TEMPORARY, I HOPE HOPE HOPE
180             CADR    STOPRATE        # TEMPORARY, I HOPE HOPE HOPE
181             TC      DOWNFLAG        # PERMIT X-AXIS OVERRIDE
```

Assembly Language

- Basic job of a CPU: execute lots of *instructions*.
- Instructions are the primitive operations that the CPU may execute.
- Different CPUs implement different sets of instructions. The set of instructions a particular CPU implements is an *Instruction Set Architecture* (*ISA*).
  - Examples: ARM, Intel x86, MIPS, RISC-V, IBM/Motorola PowerPC (quite old Mac), Intel IA64, ...

Instruction Set Architectures

- 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 (Cocke IBM, Patterson, Hennessy, 1980s)

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

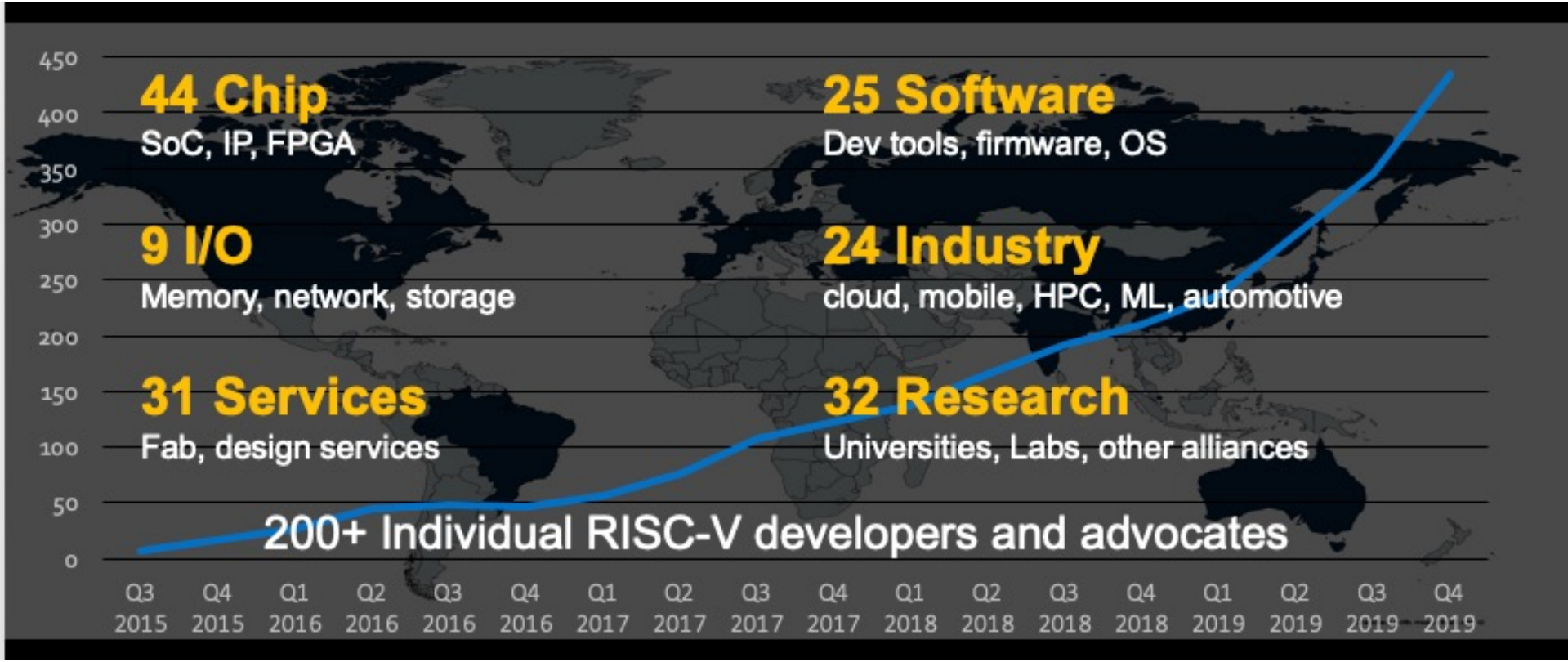
RISC-V Architecture

- New open-source, license-free RISC ISA spec
  - Supported by growing shared software ecosystem
  - Appropriate for all levels of computing system, from microcontrollers to supercomputers
  - 32-bit, 64-bit, and 128-bit variants (we're using 32-bit in class, textbook uses 64-bit)

- RISC-V standard maintained by non-profit RISC-V Foundation

More than 435 RISC-V Members

across 33 Countries Around the World



RISC-V in China

- 33 Chinese members in the global RISC-V Foundation
- 500 attendees at the China RISC-V Forum in Nov 2019

- RISC-V International Open Source Laboratory (RIOS Laboratory) research at Tsinghua-Berkeley Shenzhen Institute (TBSI) June 2019

- Alibaba processor achieves 7.1 Coremark/MHz at a frequency of 2.5GHz on a 12nm process node, which is 40 percent more powerful than any RISC-V processor produced to date. – EE/Times July 2019

- GigaDevice launched world's first general-purpose microcontroller based on RISC-V for the IOT market. – EE Times 26 Aug 2019

- Huami's upcoming Huangshan 1S Processor in 7nm
  Huami one of the top wearable manufacturers; August 27, 2019

Premier Members

Alibaba Cloud

ANDES TECHNOLOGY
FOUNDING

成为资本 CHENGWEI CAPITAL

HUAWEI

ICT
FOUNDING
Institute of Computing Technology of the Chinese Academy of Sciences

ISCAS
Institute of Software, Chinese Academy of Sciences

RIOS
RISC-V International Open Source Laboratory
Tsinghua-Berkeley Shenzhen Institute

ZTE

StarFive 赛昉科技

SiFive
FOUNDING

希姆计算 STREAM COMPUTING

UNISOC

Western Digital.
FOUNDING

Ruppin Academic Center

San Diego State University

SFAL

上海交通大學 Shanghai Jiao Tong University

Technische Universität Darmstadt

Uppsala Universitet

FreeBSD

The Programming Foundation

UEC Tokyo

Universitas Complutensis Matritensis

Universidad de Alcalá

University of Bristol

University of Engineering & Technology Peshawar

University of Minnesota Driven to Discover

UNSW Sydney

University of the Witwatersrand, Johannesburg

東京大学 The University of Tokyo

Universitas Studiorum Zagrabiensis MDCLXIX

https://riscv.org/members/

Intel invests in open-source RISC-V processors, creates billion-dollar fund

Intel and RISC-V working together is a game-changer, and today is the day that RISC-V becomes a chip power.

RISC
has

Inte
proc
CPU

Why? Because Intel sees a future in which ARM, x86, and RISC-V all play major roles. In particular, Intel has already seen strong demand for more RISC-V intellectual property (IP) and chip offerings.

Dr. P

lingua franca for computer chips, a set of instructions that would be used by all chipmakers and owned by none. Today, Patterson said, "I'm delighted that Intel, the company that pioneered the microprocessor 50 years ago, is now a member of RISC-V International."

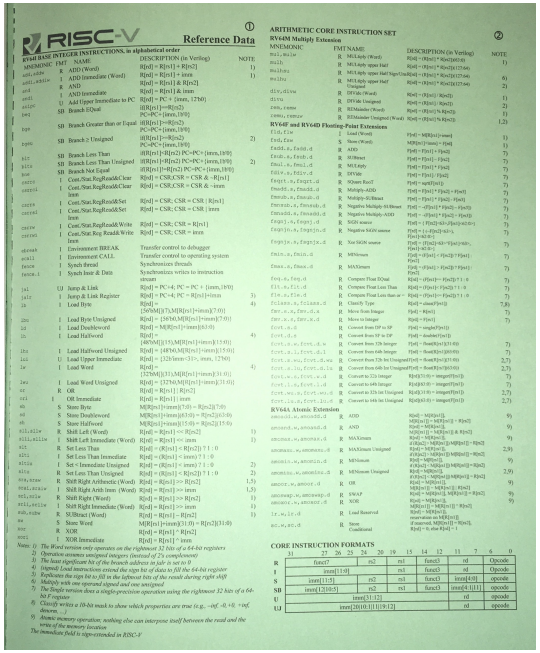
Intel plans for largest chip manufacturing site in the world

Intel Q4 revenue crushes consensus on data center growth

https://www.zdnet.com/article/intel-invests-in-open-source-risc-v-processors-with-a-billion-dollars-in-new-chip-foundries/

Why RISC-V in CS110?

- Why RISC-V instead of Intel 80x86?
  - 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
  - Only exception is two different atomic operation options: Load Reserved/Store Conditional Atomic swap/add/etc...


RISC-V Reference Data

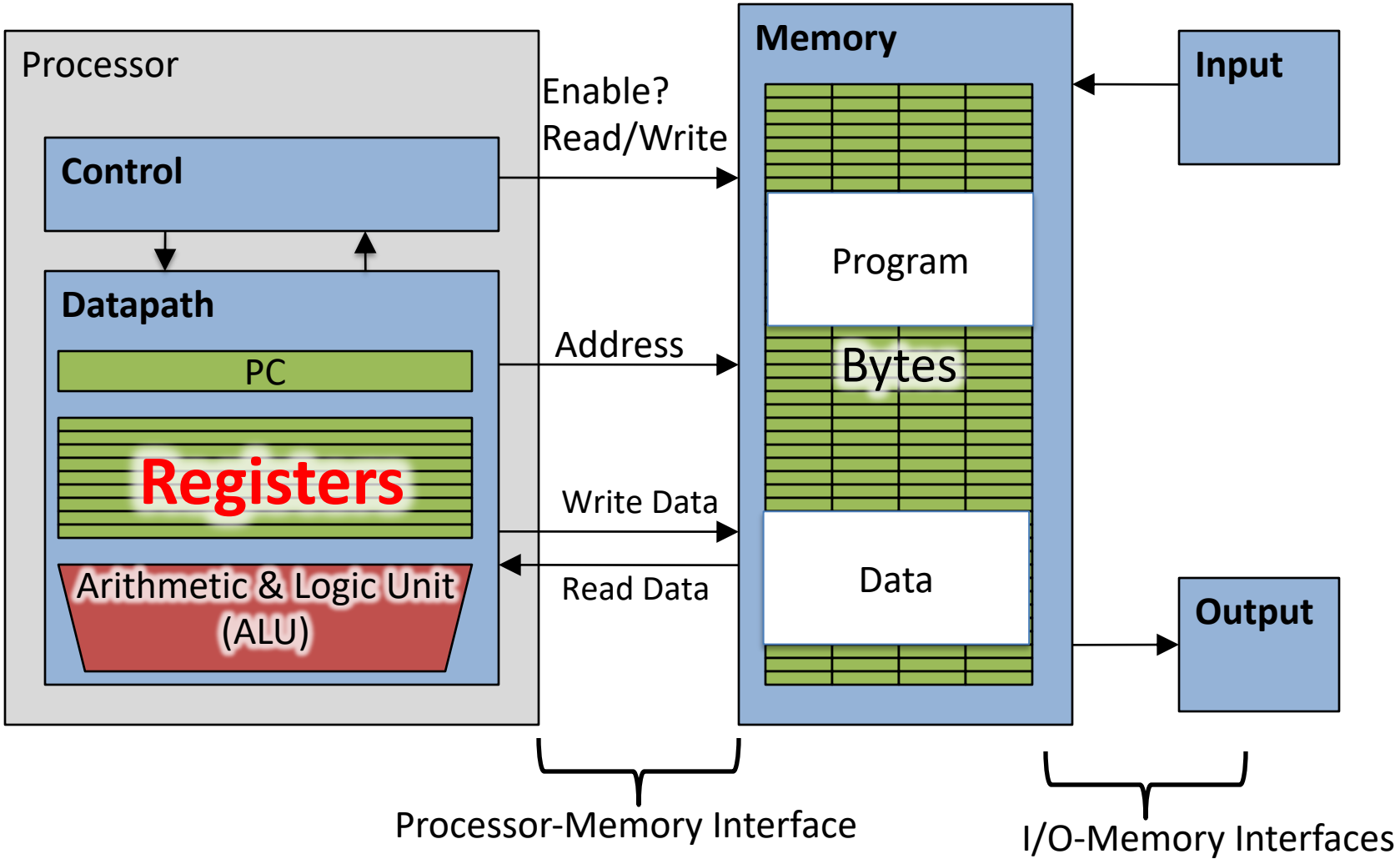
RISC-V Green Card

Assembly Variables: Registers

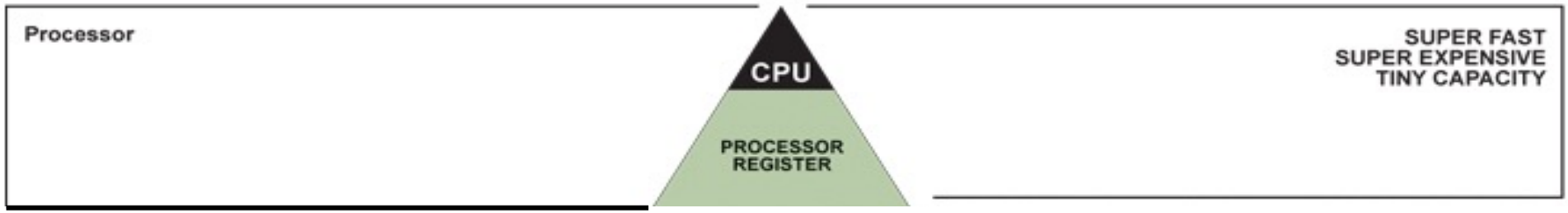
- Unlike HLL like C or Java, assembly cannot use variables
  - Why not? Keep Hardware Simple
- Assembly Operands are registers
  - Limited number of special locations built directly into the hardware
  - Operations can only be performed on these!

- Benefit: Since registers are directly in hardware, they are very fast
  (faster than 1 ns - light travels 30cm in 1 ns!!!)

Registers, inside the Processor



Great Idea #3: Principle of Locality / Memory Hierarchy



Number of Registers

- Drawback: Since registers are in hardware, there is a predetermined number of them
  - Solution: Assembly code must be very carefully put together to efficiently use registers
- 32 registers in RISC-V
  - Why 32? Smaller is faster, but too small is bad.
- 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)

RISC-V Registers

- Registers are numbered from 0 to 31
- Number references:
  – x0, x1, x2, … x30, x31
- x0 : special: always holds value zero
  => only 31 registers to hold variable values
- Each register can be referred to by number or name
  – Cover names later

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; operation determines how register contents are treated

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
- Instructions are related to operations (=, +, -, *, /) in C or Java

Comments in Assembly

- 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
  - This is just like the C99 //
- Note: Different from C.
  - C comments have format
    /* comment */
    so they can span many lines

RISC-V Addition and Subtraction (1/4)

- Syntax of Instructions:
  - One two, three, four **`add x1, x2, x3`**

  where:
  - One = operation by name
  - two = operand getting result (“destination”)
  - three = 1st operand for operation (“source1”)
  - four = 2nd operand for operation (“source2”)
- Syntax is rigid:
  - 1 operator, 3 operands
  - Why? Keep Hardware simple via regularity

Addition and Subtraction of Integers (2/4)

- Addition in Assembly
  - Example: `add x1,x2,x3` (in RISC-V)
  - Equivalent to: a = b + c (in C)
  - where C variables ⇔ RISC-V registers are:
    a ⇔ `x1`, b ⇔ `x2`, c ⇔ `x3`
- Subtraction in Assembly
  - Example: `sub x3,x4,x5` (in RISC-V)
  - Equivalent to: d = e - f (in C)
  - where C variables ⇔ RISC-V registers are:
    d ⇔ `x3`, e ⇔ `x4`, f ⇔ `x5`

Addition and Subtraction of Integers (3/4)

- How to do the following C statement?

  a = b + c + d - e;

- Break into multiple instructions

```
add x10, x1, x2  # a_temp = b + c
add x10, x10, x3 # a_temp = a_temp + d
sub x10, x10, x4 # a = a_temp - e
```

- Notice: A single line of C may break up into several lines of RISC-V.
- Notice: Everything after the hash mark on each line is ignored (comments).

Addition and Subtraction of Integers (4/4)

- How do we do this?

  f = (g + h) - (i + j);

- Use intermediate temporary register

```
add x5, x20, x21 # a_temp = g + h
add x6, x22, x23 # b_temp = i + j
sub x19, x5, x6  # f = (g + h)- (i + j)
```

Immediates

- Immediates are numerical constants.
- They appear often in code, so there are special instructions for them.
- Add Immediate:
  - `addi x3,x4,10` (in RISC-V)
  - f = g + 10 (in C)
  - where RISC-V registers `x3,x4` are associated with C variables f, g
- Syntax similar to add instruction, except that last argument is a number instead of a register.

Immediates

- There is no Subtract Immediate in RISC-V: Why?
  - –There are add and sub, but no addi counterpart
- Limit types of operations that can be done to absolute minimum
  - – if an operation can be decomposed into a simpler operation, don't include it
  - – addi …, -X = subi …, X => so no subi

    `addi x3,x4,-10` (in RISC-V)

    f = g - 10 (in C)
  - – where RISC-V registers `x3, x4` are associated with C variables f, g

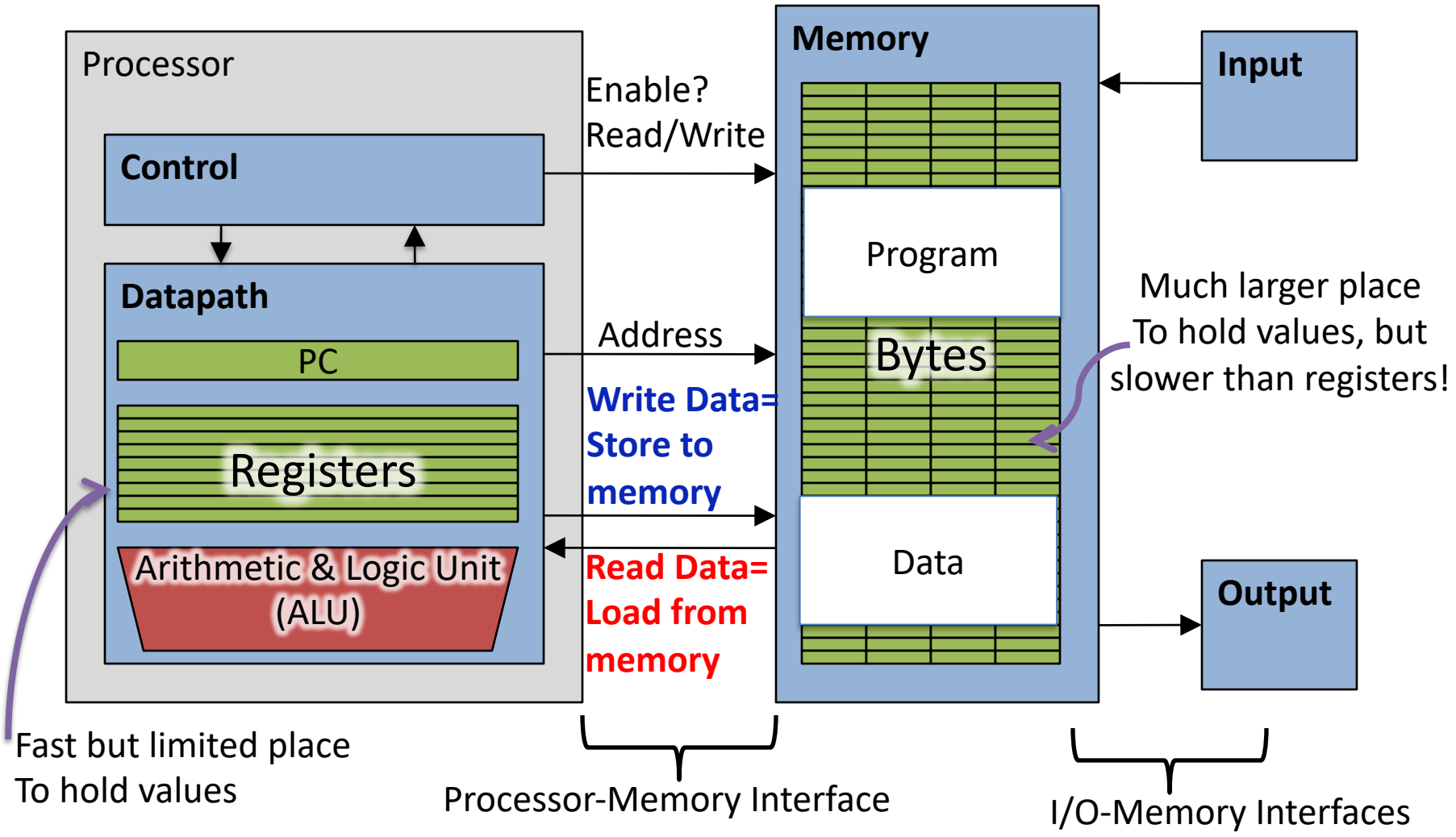
Register Zero

- One particular immediate, the number zero (0), appears very often in code.
- So the register zero (`x0`) is 'hard-wired' to value 0; e.g.
  - `add x3, x4, x0` (in RISC-V)
  - f = g (in C)
  - where RISC-V registers `x3,x4` are associated with C variables f, g
- Defined in hardware, so an instruction
  - `add x0,x3,x4` will not do anything!

No-Op

- A No-op is an instruction that does nothing...
  - Why?
    You may need to replace code later: No-ops can fill space, align data, and perform other options
- By ***convention*** RISC-V has a specific no-op instruction...
  - **`add x0 x0 x0`**
- Why?
  - Writes to x0 are always ignored...
    RISC-V uses that a lot as we will see in the jump-and-link operations
  - Making a "standard" no-op improves the disassembler and can potentially improve the processor

Data Transfer: Load from and Store to memory

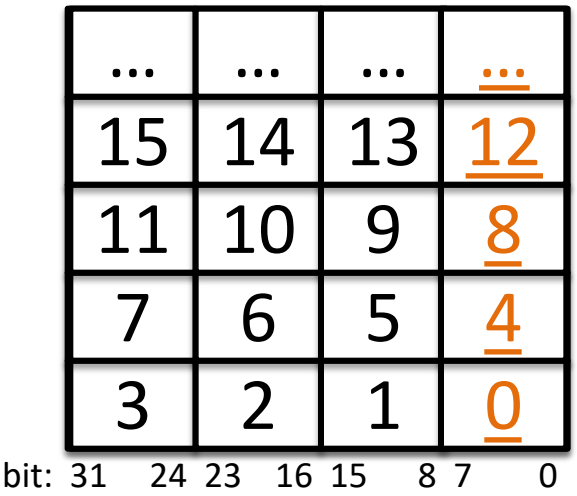


Memory Addresses are in Bytes

- Lots of data is smaller than 32 bits, but rarely smaller than 8 bits – works fine if everything is a multiple of 8 bits
- 8 bit chunk is called a *byte*

  (1 word = 4 bytes)
- Memory addresses are really in *bytes*, not words
- Word addresses are 4 bytes apart
  - Word address is same as address of lowest byte

Little Endian: Start with the small end (little end; Least significant byte of the word)



en.wikipedia.org/wiki/Big_endian

Big Endian vs. Little Endian

Big-endian and little-endian from Jonathan Swift's *Gulliver's Travels*

- **The order in which BYTES are stored in memory**
- **Bits always stored as usual. (E.g., 0xC2=0b 1100 0010)**

Consider the number 1025 as we normally write it:

BYTE3 BYTE2 BYTE1 BYTE0
00000000 00000000 00000100 00000001

Big Endian

ADDR3 ADDR2 ADDR1 ADDR0
BYTE0 BYTE1 BYTE2 BYTE3
00000001 00000100 00000000 00000000

Examples

Names in China (e.g. Schwertfeger, Sören)

Java Packages: (e.g. org.mypackage.HelloWorld)

Dates done correctly ISO 8601 YYYY-MM-DD (e.g. 2020-03-22)

Eating Pizza crust first

Unix file structure (e.g., /usr/local/bin/python)

”Network Byte Order”: most network protocols

IBM z/Architecture; very old Macs

Little Endian

ADDR3 ADDR2 ADDR1 ADDR0
BYTE3 BYTE2 BYTE1 BYTE0
00000000 00000000 00000100 00000001

Examples

Names in the west (e.g. Sören Schwertfeger)

Internet names (e.g. sist.shanghaitech.edu.cn)

Dates written in England DD/MM/YYYY (e.g. 22/03/2020)

Eating Pizza skinny part first (the normal way)

CANopen

Intel x86; RISC-V

bi-endian: ARM (runs mostly little endian), MIPS, IA-64, PowerPC

Example

Memory

Addr. dec	Addr. hex	8-bit Value
…	…	…
15	0x0F	0x77
14	0x0E	0x66
13	0x0D	0x55
12	0x0C	0x44
11	0x0B	0x33
10	0x0A	0x22
9	0x09	0x11
8	0x08	0x00
7	0x07	0xEF
6	0x06	0xCD
5	0x05	0xAB
4	0x04	0x89
3	0x03	0x67
2	0x02	0x45
1	0x01	0x23
0	0x00	0x01

Little Endian

Addresses (hex):

…	…	…	…
F	E	D	C
B	A	9	8
7	6	5	4
3	2	1	0

```
Word at address 0x0C: 0x 77 66 55 44
Word at address 0x08: 0x 33 22 11 00
Word at address 0x04: 0x EF CD AB 89
Word at address 0x00: 0x 67 45 23 01
```

…	…	…	…
77	66	55	44
33	22	11	00
EF	CD	AB	89
67	45	23	01

Big Endian

Addresses (hex):

…	…	…	…
C	D	E	F
8	9	A	B
4	5	6	7
0	1	2	3

```
Word at address 0x0C: 0x 44 55 66 77
Word at address 0x08: 0x 00 11 22 33
Word at address 0x04: 0x 89 AB CD EF
Word at address 0x00: 0x 01 23 45 67
```

…	…	…	…
44	55	66	77
00	11	22	33
89	AB	CD	EF
01	23	45	67

RISC-V: Little Endian

(E.g., 1025 = 0x401 = 0b 0100 0000 0001)

ADDR3	ADDR2	ADDR1	ADDR0
BYTE3	BYTE2	BYTE1	BYTE0
00000000	00000000	00000100	00000001

Little Endian
Least significant byte in a word
(numbers are addresses) ↓

...	...	...	...
15	14	13	12
11	10	9	8
7	6	5	4
3	2	1	0

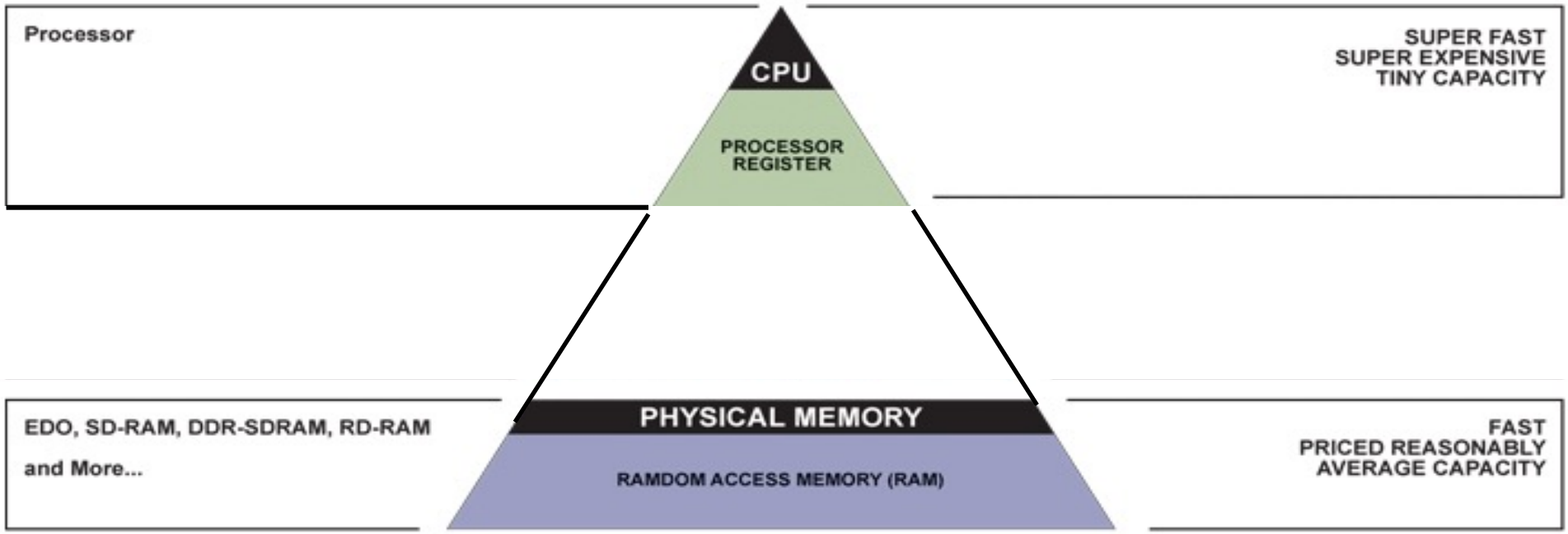
bit: 31 24 23 16 15 8 7 0

- Hexadecimal number:
  0xFD34AB88 ($4{,}248{,}087{,}432_{ten}$) =>
  - Byte 0: 0x88 (136_{ten})
  - Byte 1: 0xAB (171_{ten})
  - Byte 2: 0x34 (52_{ten})
  - Byte 3: 0xFD (253_{ten})

Address:	64 address of word (e.g. int)			
Address:	67	66	65	64
Data:	0xFD	0x34	0xAB	0x88

- Little Endian: Starts with the little end of a word:
  - It starts with the smallest (least significant) Byte

Great Idea #3: Principle of Locality / Memory Hierarchy



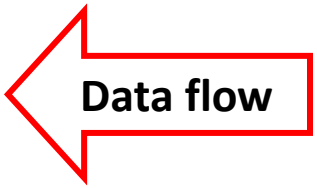
Speed of Registers vs. Memory

- Given that
  - Registers: 32 words (128 Bytes)
  - Memory: Billions of bytes (2 GB to 64 GB on laptop)
- and the RISC principle is…
  - Smaller is faster
- How much faster are registers than memory??
- About 100-500 times faster!
  - in terms of *latency* of one access

Load from Memory to Register

- C code

```
int  A[100];          /* A => x15 */
g = h + A[3];         /* h => x13 */
```



- Using Load Word (`lw`) in RISC-V:

```
lw   x10,12(x15)  # Reg x10 gets A[3]
add  x11,x13,x10  # g = h + A[3]
```

Note: `x15` – base register (pointer to A[0])

`12` – offset in bytes

Offset must be a constant known at assembly time

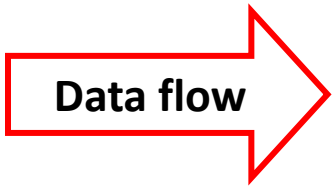
Store from Register to Memory

- C code

```
int  A[100];          /* A => x15 */
A[10] = h + A[4]; /* h => x13 */
```

- Using Store Word (sw) in RISC-V:

```
lw   x10,16(x15)    # Temp reg x10 gets A[4]
add  x10,x13,x10    # Temp reg x10 gets h + A[4]
sw   x10,40(x15)    # A[10] = h + A[4]
```



Note: x15 – base register (pointer)

16,40 – offsets in bytes

Memory Alignment

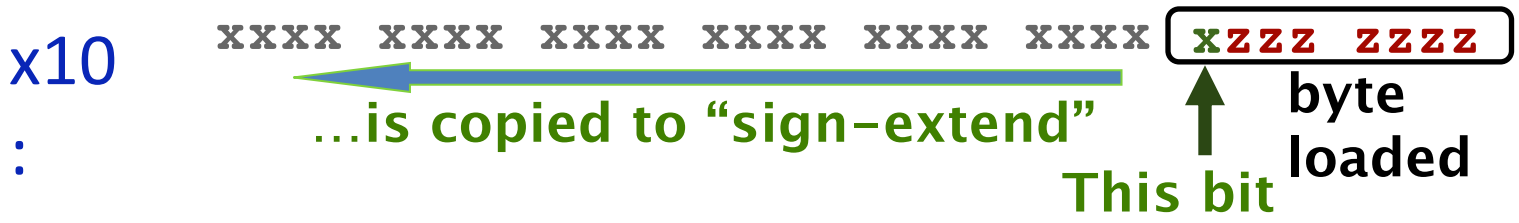
- RISC-V does not ***require*** that integers be word aligned...
  - But it can be very **very bad** if you don't make sure they are...
- Consequences of unaligned integers
  - Slowdown: The processor is allowed to be a lot slower when it happens
    - In fact, a RISC-V processor may natively only support aligned accesses, and do unaligned-access in ***software***!
      An unaligned load could take ***hundreds of times longer***!
    - Lack of **atomicity**: The whole thing doesn't happen at once... can introduce lots of very subtle bugs
- So in ***practice***, RISC-V requires integers to be aligned on 4- byte boundaries

Loading and Storing Bytes

- In addition to word data transfers (**`lw`**, **`sw`**), RISC-V has byte data transfers:
  - load byte: **`lb`**
  - store byte: **`sb`**
- Same format as **`lw`**, **`sw`**

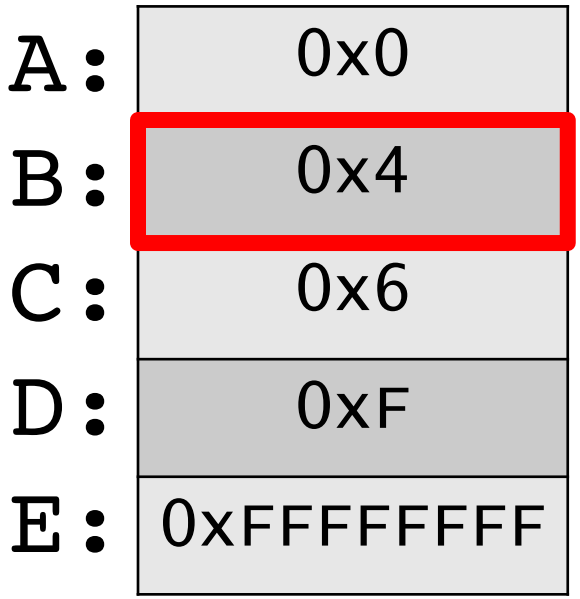
RISC-V also has "unsigned byte" loads (**`lbu`**) which zero extends to fill register. Why no unsigned store byte **`sbu`**?

- E.g., **`lb x10,3(x11)`**
  - contents of memory location with address = sum of "3" + contents of register x11 is copied to the low byte position of register x10.



Question! What's in x12?

```
addi x11,x0,0x4F6
sw x11,0(x5)
lb x12,1(x5)
```



Question! What's in x12?

```
addi x11,x0,0x85BCF6
sw x11,0(x5)
lb x12,2(x5)
```

Actually:
invalid instruction:
immediate is too large;
details covered later

A:	0x8
B:	0x85
C:	0xC
D:	0xBC
E:	0XFFFFFF85
F:	0XFFFFFFF8
G:	0XFFFFFFC
H:	0XFFFFFFBC

Question!
Which of the following is TRUE?

- A: `add x10, x11, 4(x12)` is valid in RV32
- B: can byte address 4GB of memory with an RV32 word
- C: imm must be multiple of 4 for lw x10, imm(x10) to be valid
- D: None of the above

“And in Conclusion…”

- In RISC-V Assembly Language:
  - Registers replace C variables
  - One instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- In RV32, words are 32b
- RISC-V is Little Endian
- Instructions:
  `add`, `addi`, `sub`, `lw`, `sw`, `lb`, `lbu`, `sw`
- Registers:
  - 32 registers, referred to as `x0` – `x31`
  - Zero: `x0`