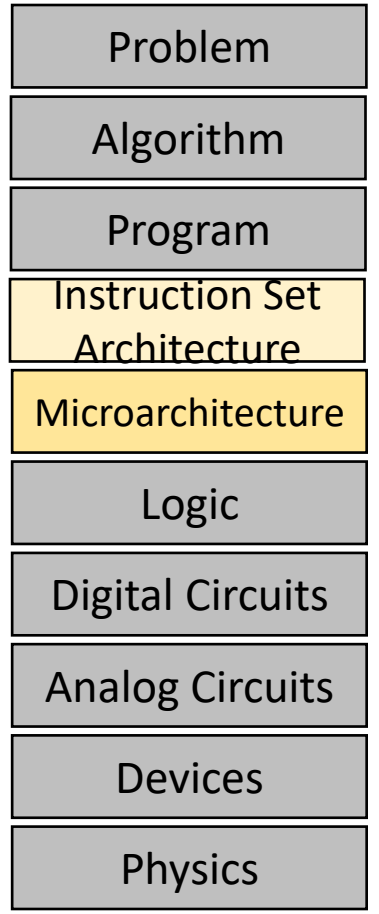
Universität Bremen

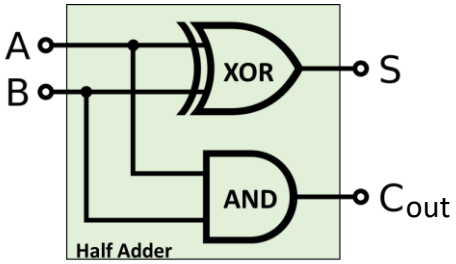
RISC-V Assembly Language

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck

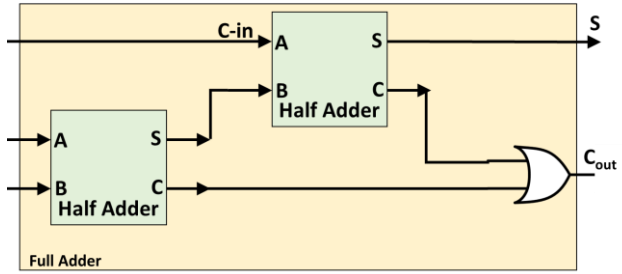


Revision

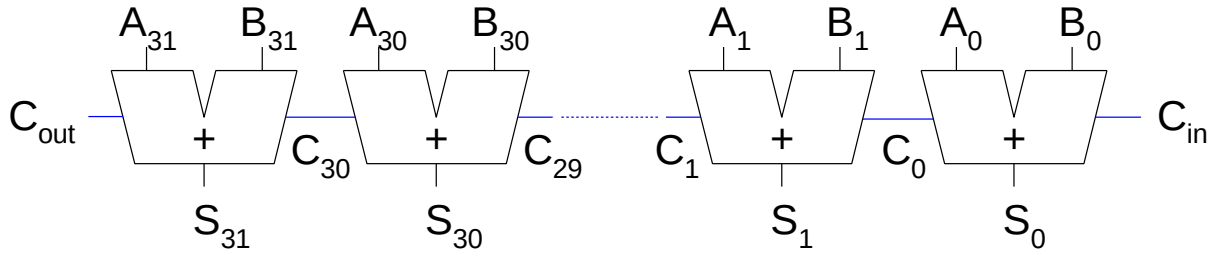
Half Adder

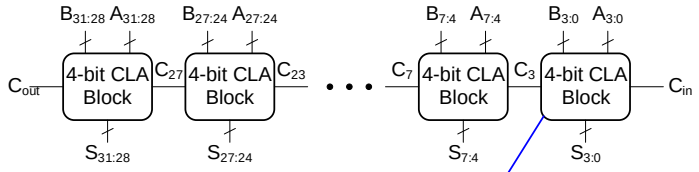


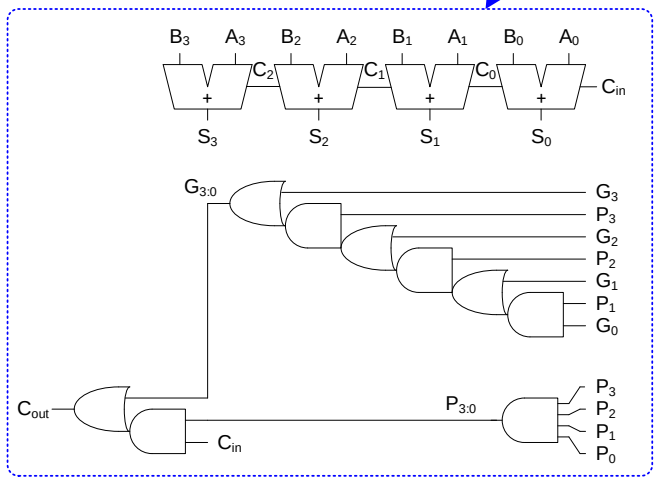
Full Adder



Ripple Carry Adder



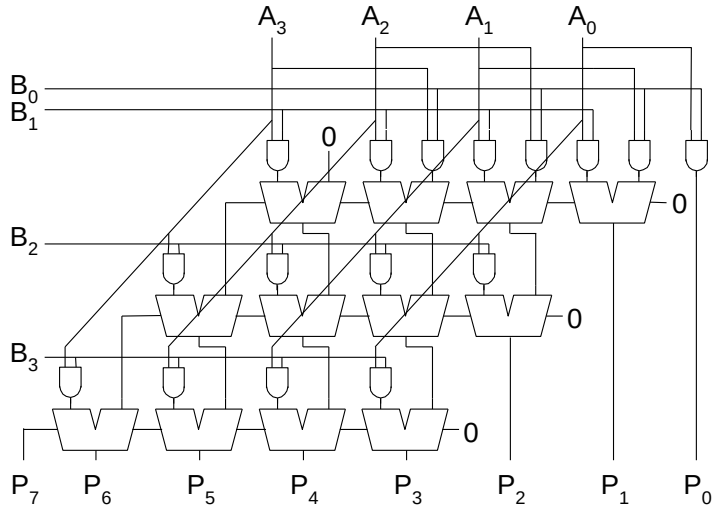




Carry Look-ahead Adder

$$
\begin{array}{cccccccc}
 & & & A_3 & A_2 & A_1 & A_0 & \\
\times & & & B_3 & B_2 & B_1 & B_0 & \\
\hline
 & & & A_3B_0 & A_2B_0 & A_1B_0 & A_0B_0 & \\
 & & A_3B_1 & A_2B_1 & A_1B_1 & A_0B_1 & & \\
 & A_3B_2 & A_2B_2 & A_1B_2 & A_0B_2 & & & \\
+ & A_3B_3 & A_2B_3 & A_1B_3 & A_0B_3 & & & \\
\hline
P_7 & P_6 & P_5 & P_4 & P_3 & P_2 & P_1 & P_0
\end{array}
$$

Multiplier



Arithmetic Overflow/Underflow Examples

- **Ariane 5 Rocket Explosion (1996):** One of the most famous cases of integer overflow occurred with the Ariane 5 rocket. Shortly after its launch, the rocket exploded due to an integer overflow in the software. A 64-bit floating-point number related to the rocket's horizontal velocity was converted to a 16-bit integer. The number was too large to be represented in 16 bits, leading to an overflow and subsequent failure of the flight control system.
- **Therac-25 Radiation Therapy Machine (1985-1987):** The Therac-25 was a radiation therapy machine that suffered from several critical software errors, including integer overflow. These errors led to patients receiving lethal doses of radiation.

https://www.bugsnag.com/blog/bug-day-ariane-5-disaster/

Universität Bremen

Today's Topics

- Why is RISC-V important?
- Instruction Classes of RISC-V
  - Basic Arithmetic
    - Addition/Subtraction
  - Immediates
    - Immediate Addition
  - Memory Layout
  - Data Transfer
    - Store Word
    - Load Word
  - Decision Making
    - Conditional Branching
    - Non-Conditional Branching

Problem at Hand

```
int A[20];
// fill A with data
int sum = 0;
int a = 0;
for (int i=0; i < 20; i++) {
    a = i + sum;
    sum += A[i];
    A[i] = a;
}
```

Initialize to 0

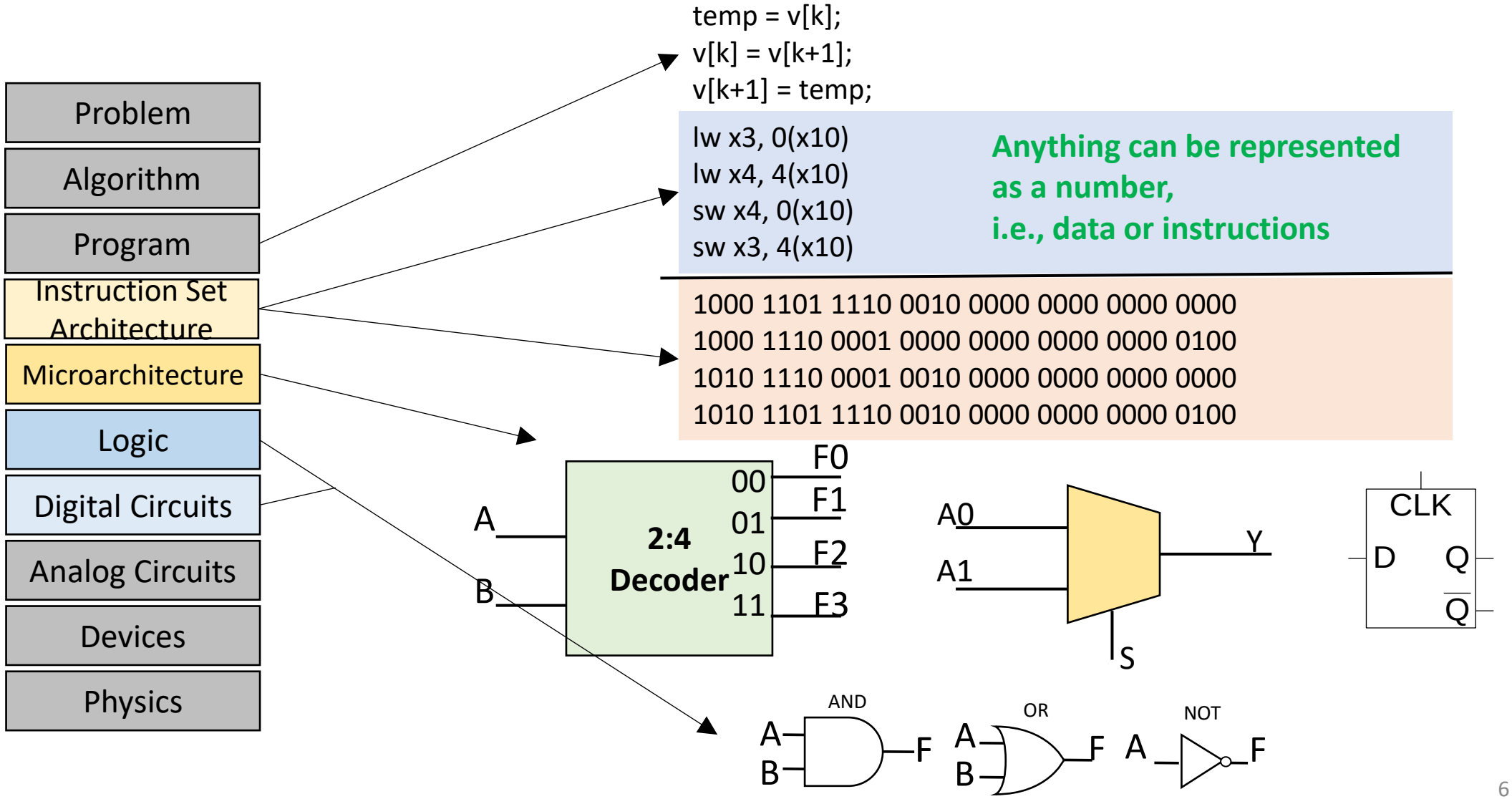
Comparison of two values

addition of two values

Accessing (loading) a value

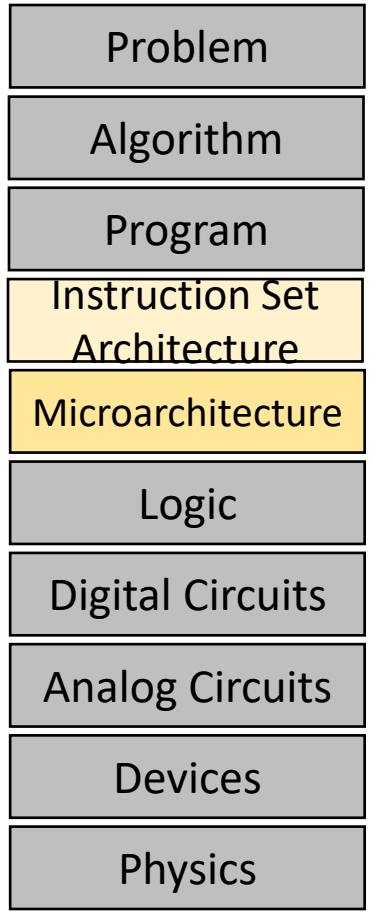
Storing a value

Abstractions



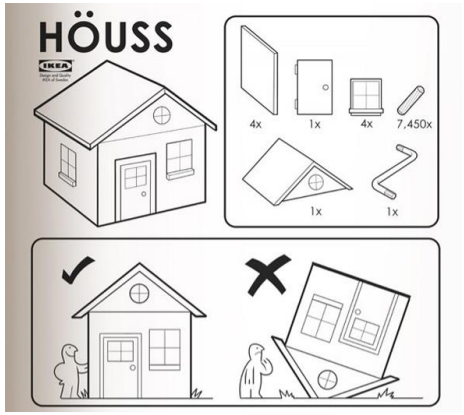
Introduction

- Jumping up a few levels of abstraction
- **Architecture:** programmer's view of computer
  - Defined by instructions & operand locations
  - ***Instruction Set Architecture*** (ISA) is typically seen as the contract between software and hardware.
- **Microarchitecture:** how to implement an architecture in hardware (covered in Lecture 8)



IKEA Furniture

Similar to computer



Similar to assembly language

Aspect	IKEA Furniture	Assembly Language
Detailed Instructions	Precise, step-by-step instructions in the manual for assembling furniture.	Gives precise, step-by-step instructions to the computer.
Precision and Accuracy	Ensuring each part fits perfectly by following the manual.	Allows precise control over the computer's operations.
Efficiency	Building furniture efficiently using the manual.	Helps computers perform tasks quickly and efficiently.
Understanding Mechanics	Understanding furniture design through the assembly process.	Reveals how computers work internally.
Complete Control	Controlling the assembly process of the furniture.	Gives programmers complete control over how the computer operates.

https://www.ikea.com/ph/en/this-is-ikea/about-us/the-story-of-ikea-flatpacks-puba710ccb0
https://blog.ted.com/8-ikea-memes-and-homages/

Assembly Language

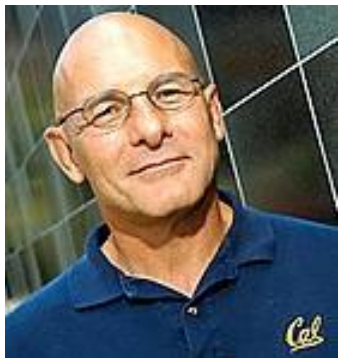
- Basic job of a CPU: execute lots of **instructions**
- **Instructions:** commands in a computer's language
  - Assembly language: human-readable format of instructions
  - Machine language: computer-readable format (1's and 0's)

Different CPUs implement different sets of instructions

Instruction Set Architecture (ISA).
Examples: ARM (cell phones), Intel x86 (i9, i7, i5, i3), IBM Power, IBM/Motorola PowerPC (old Macs), MIPS, RISC-V, ...

- **RISC-V architecture:**

  Developed by Kristе Asanovic, David Patterson, and their colleagues at UC Berkeley in 2010.


  - First widely accepted open-source computer architecture
- Once you've learned one architecture, it's easier to learn others

Instruction Set Architectures

- Early trend was to add more and more instructions to new CPUs to do elaborate operations
  - VAX [CISC](Complex Instruction Set Computing) architecture by Digital Equipment Corporation 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.
  - This went against the convention wisdom of the time (he who laughs last, laughs best)

By Emiliano Russo, Associazione Culturale VerdeBinario - http://it.wikipedia.org/wiki/Immagine:Microvax_3600.jpg, Public Domain, https://commons.wikimedia.org/w/index.php?curid=4807104

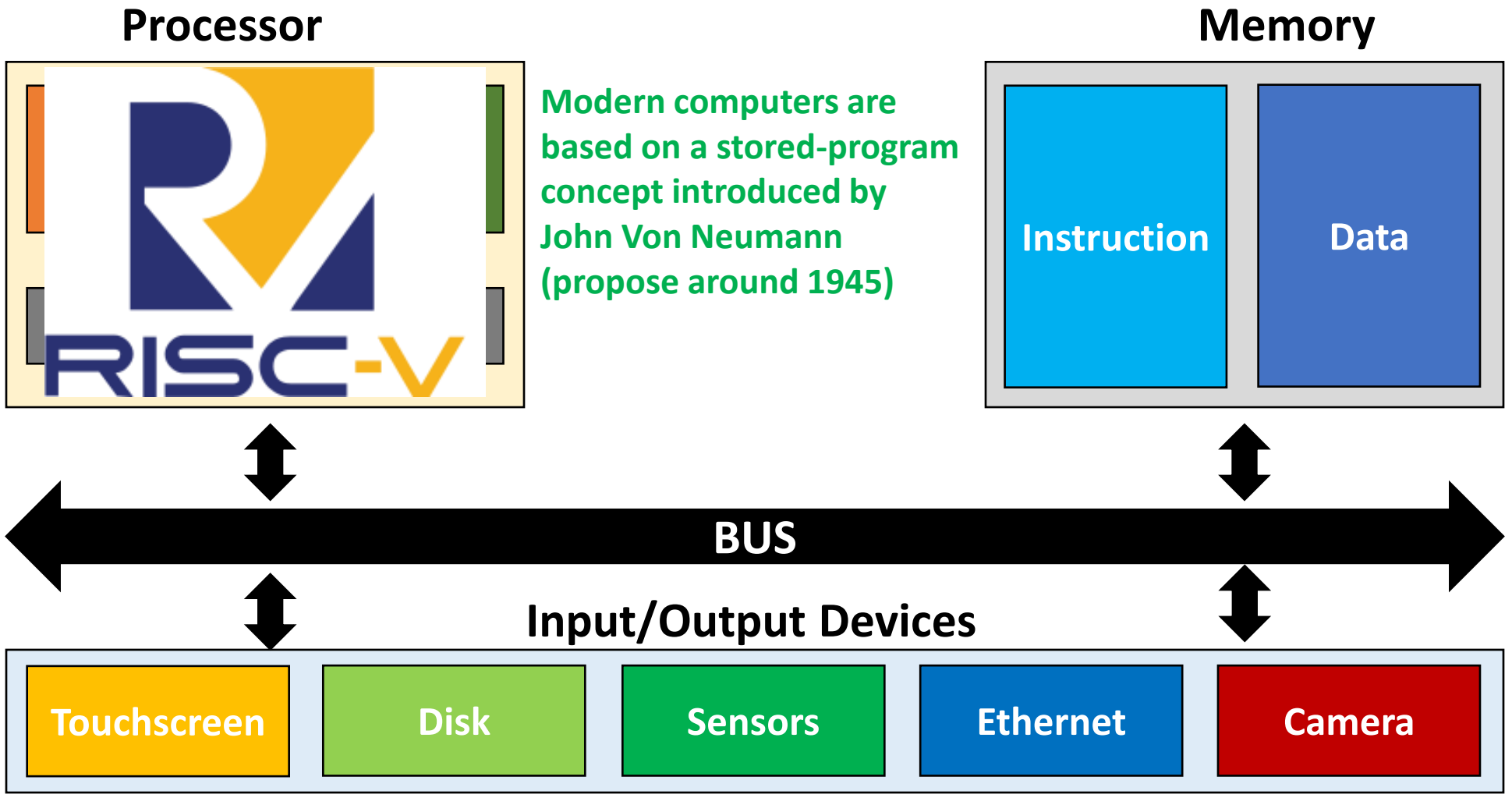
Universität Bremen

RISC-V Architecture

- New open-source, license-free 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
- Why RISC-V instead of Intel 80x86?
  - RISC-V is simple, elegant. Don’t want to get bogged down in gritty details.
  - RISC-V has exponential adoption
- Read more:
  - https://riscv.org/risc-v-history/
  - https://riscv.org/risc-v-genealogy/

RISC-V

Main Components of a Modern Computer



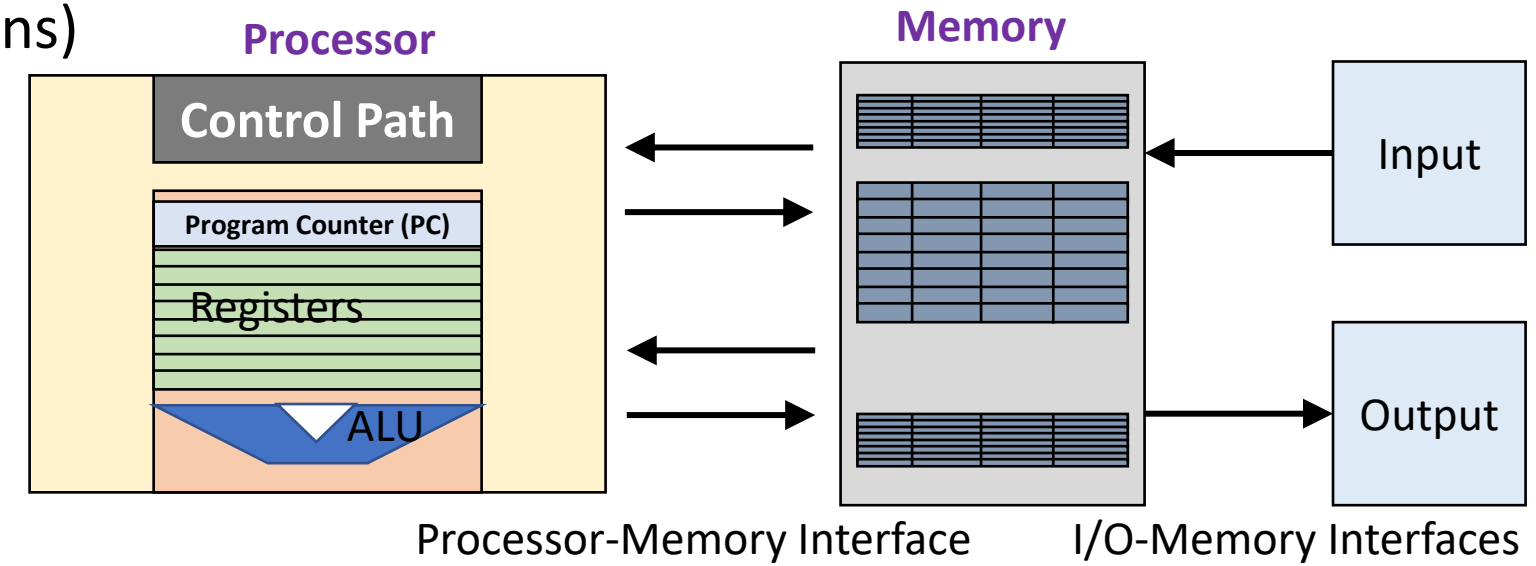
Vectorised by Vulphere from https://riscv.org/wp-content/uploads/2017/05/Tue1100-RISC-V-Foundation-Update.pdf

Assembly Variables – Registers

Variables are an abstraction provided by high-level languages.

- Unlike High-Level Languages 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're very fast (faster than 0.25ns)

Drawback: Since registers are in hardware, there is a predetermined number of them



Each RISC-V register is 32 bits wide (in RV32 variant)

Registers are numbered from 0 to 31

x0 is special, always holds value zero

Universität Bremen

A Simple (32-bit) RISC-V Processor

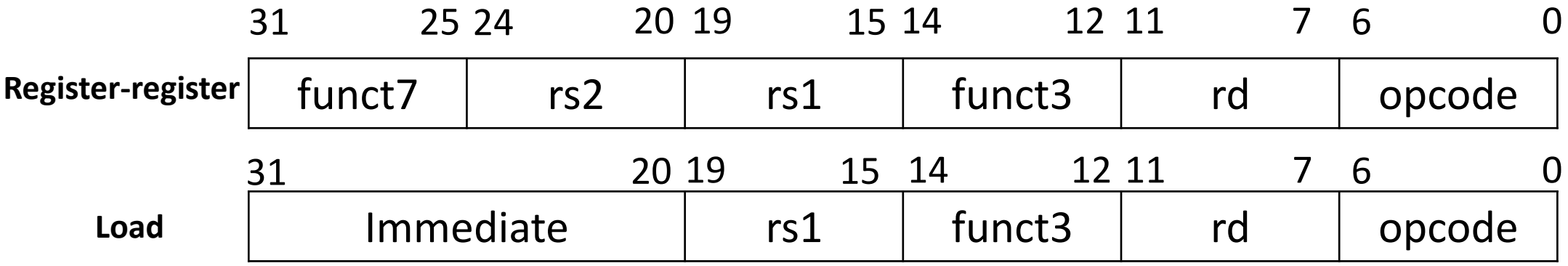
- We will only need a few simple components:
  - **Memories** – to store our program (instructions) and data
  - A **register file** – instructions will read their operands from the register file and also write their results to it.
  - **Registers**, an **ALU** and **adders**
  - **Decode** and **control logic**
- Instructions are encoded in 32-bits.
- Registers and Datapath are also 32-bits wide.
- Memory is accessed with a 32-bit address and returns 32-bit data.
- RISC-V has 32 registers, hence we must use 5-bits to identify a particular register (as 2^5 =32).

A Processor Datapath – Encoding Instructions

- A simple data processing instruction may have the following format, where Operand2 may be a register or immediate value.

Instruction Rd, Rs, Operand2

- RISC-V: 32-bits to encode instructions, e.g. register-register, Load



RISC-V Addition and Subtraction

- Syntax of Instructions:
  - one two, three, four

    add x2, x3, x4

  - where:
    - one = operation by name
    - two = operand getting result (“destination,” x2)
    - three = 1st operand for operation (“source1,” x3)
    - four = 2nd operand for operation (“source2,” x4)
- Syntax is rigid:
  - 1 operator, 3 operands
  - Why?
    - Keep hardware simple via regularity

Universität Bremen

Addition and Subtraction of Integers

- 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 cont …

- How to do the following C statement?

  a = b + c + d - e;

- Break into multiple instructions
  - add x10, x1, x2
  - add x10, x10, x3
  - sub x10, x10, x4

a_temp = b + c
a_temp = a_temp + d
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).

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

There is no Subtract Immediate in RISC-V: Why?
There are add and sub, but no addi counterpart

- 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

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

- Syntax similar to add instruction, except that last argument is a number instead of a register.

addi …, -x = "subi …, x" => so no "subi"

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

f = g - 10 (in C)

Universität Bremen

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)
- Defined in hardware, so an instruction

  add x0,x3,x4 will not do anything!

RISC-V Register Set

Name	Register Number	Usage
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0-2	x5-7	Temporaries
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0-1	x10-11	Function arguments / return values
a2-7	x12-17	Function arguments
s2-11	x18-27	Saved registers
t3-6	x28-31	Temporaries

Data Transfers: Load from and Store to memory

- Addition/subtraction

  add rd, rs1, rs2

  R[rd] = R[rs1] + R[rs2]

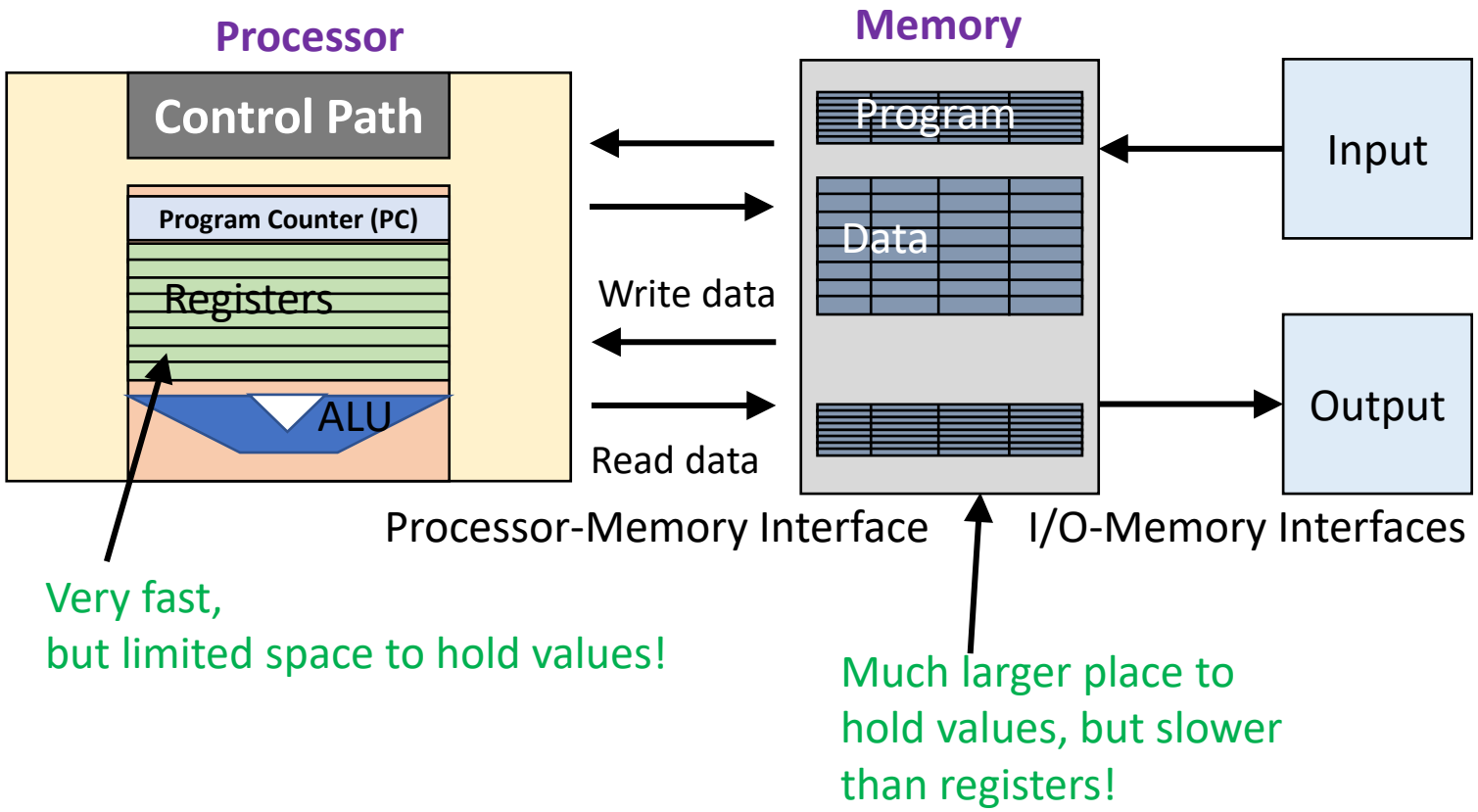
  sub rd, rs1, rs2

  R[rd] = R[rs1] - R[rs2]

- Add immediate

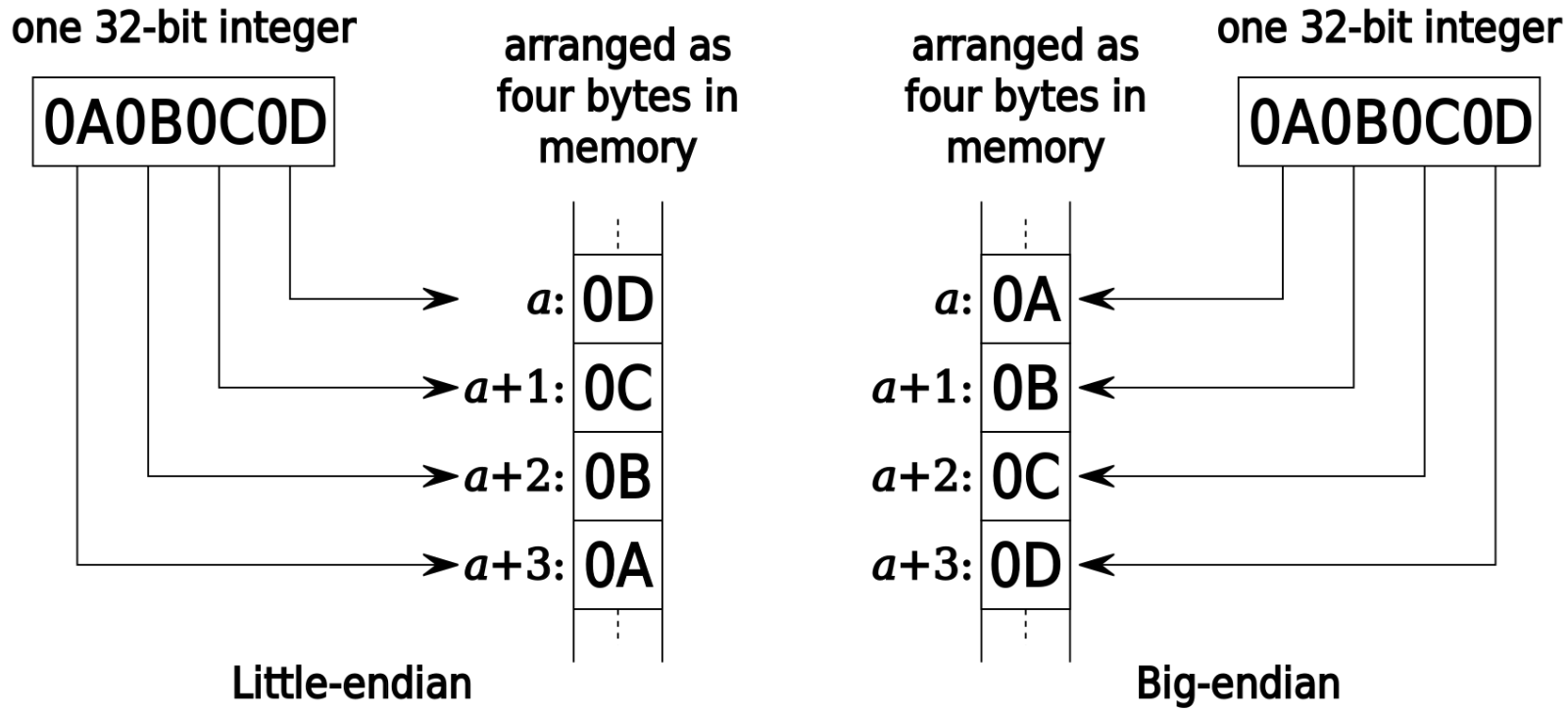
  addi rd, rs1, imm

  R[rd] = R[rs1] + imm



Endianness

Endianness refers to the order in which bytes are arranged and accessed in computer memory.

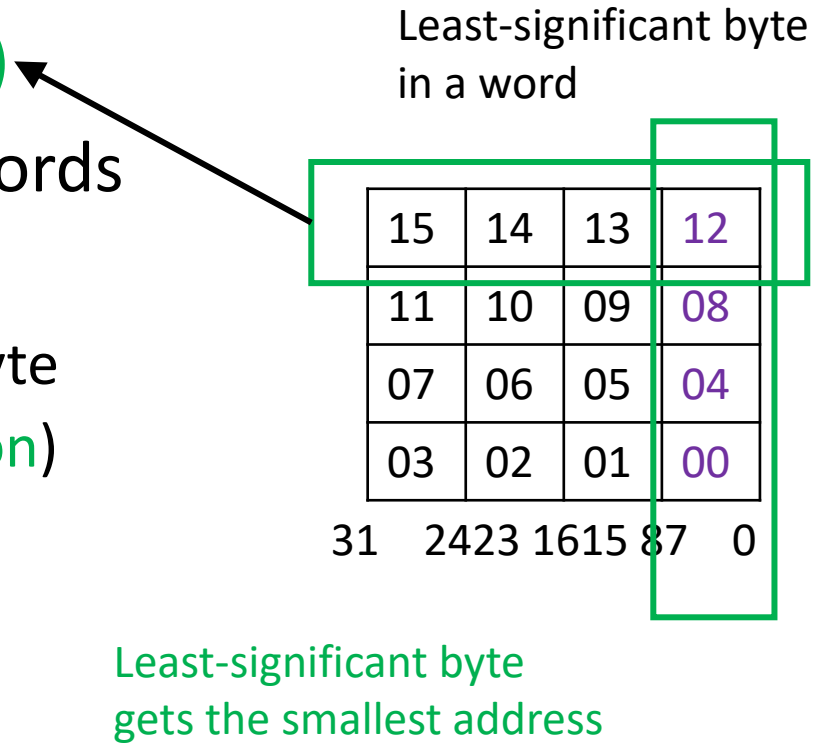


Little-endian is favored in arithmetic operations on low-level hardware because the least significant byte is readily accessible

https://en.wikipedia.org/wiki/Endianness#/media/File:32bit-Endianess.svg

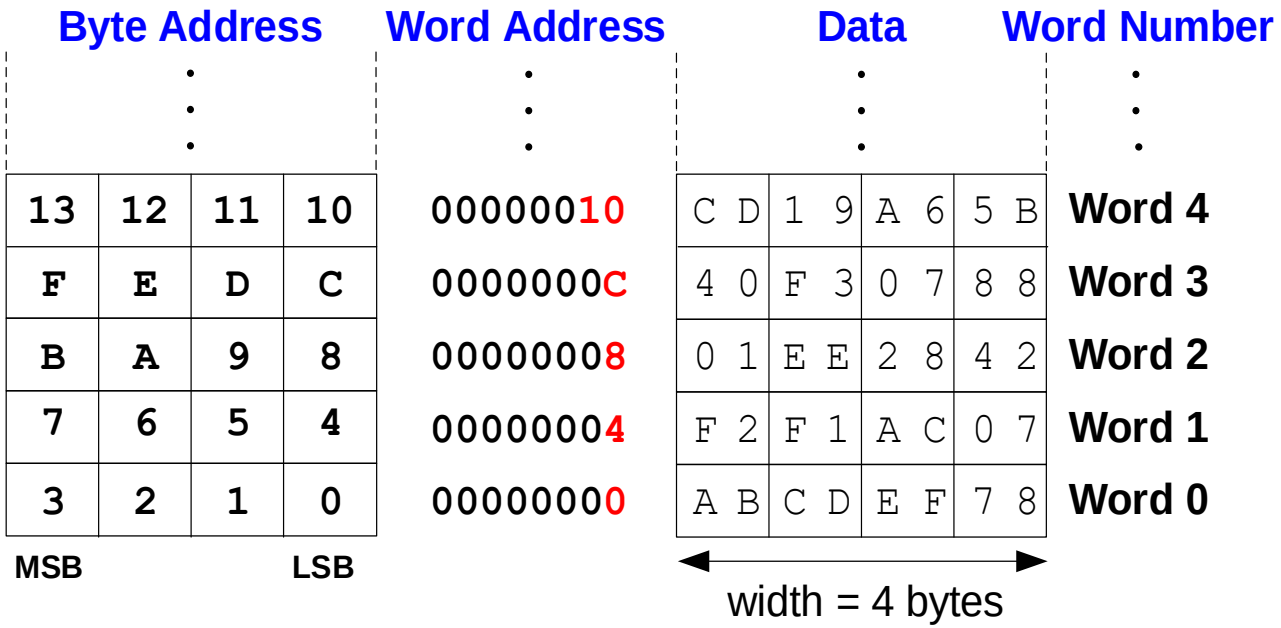
RISC-V Memory Addresses

- Data typically smaller than 32 bits, but rarely smaller than 8 bits (e.g., char type) – 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 rightmost byte
  - least-significant byte (i.e. Little-endian convention)



Byte-Addressable Memory

- Each data byte has a unique address
- Load/store words or single bytes: load byte (lb) and store byte (sb)
- 32-bit word = 4 bytes, so word address **increments by 4**



Reading Byte-Addressable Memory

- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is 2 × 4 = 8
  - the address of memory word 10 is 10 × 4 = 40 (0x28)

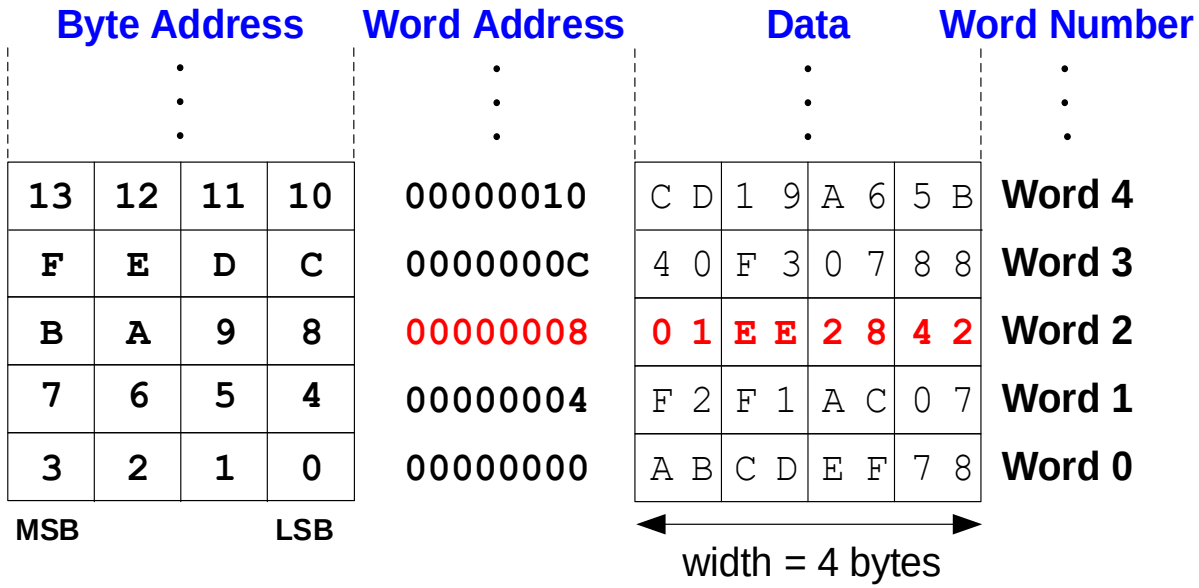
- RISC-V is **byte-addressed**, not word-addressed
  - Each byte can be individually addressed and manipulated

Reading Byte-Addressable Memory

- Example: Load a word of data at memory address 8 into s3.

RISC-V assembly code

```
lw s3, 8(zero)  # read word at address 8 into s3
```



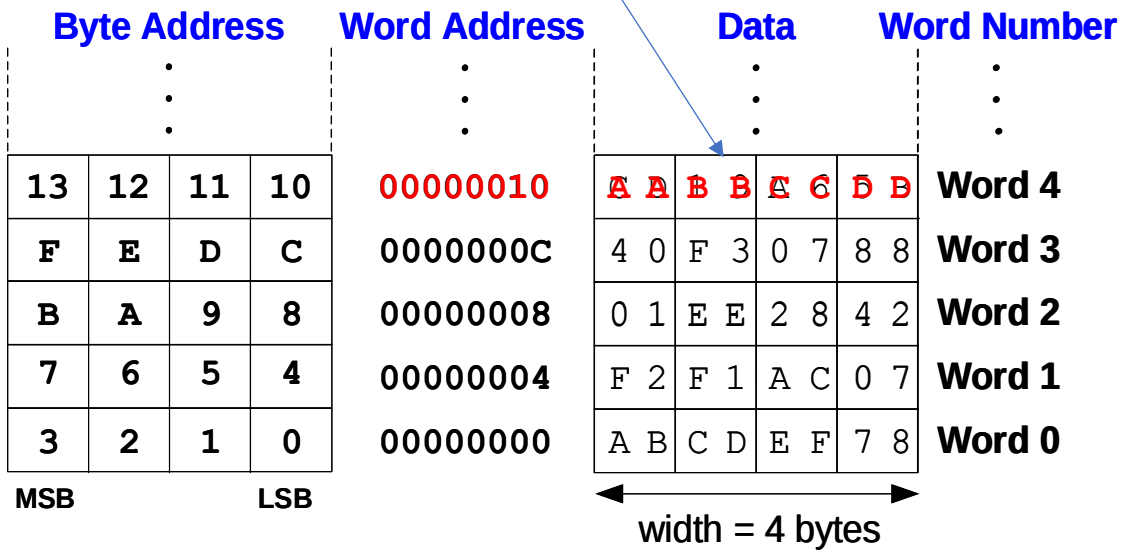
s3 holds the value 0x1EE2842 after load

Writing Byte-Addressable Memory

- Example: store the value held in t7 into memory address 0x10 (16)
  - if t7 holds the value 0xAABBCCDD, then after the sw completes, word 4 (at address 0x10) in memory will contain that value

RISC-V assembly code

```
sw t7, 0x10(zero)  # write t7 into address 16
```



Branching

- Execute instructions out of sequence
- Types of branches:
- Conditional
  - branch if equal (beq)
  - branch if not equal (bne)
  - branch if less than (blt)
  - branch if greater than or equal (bge)
- Unconditional
  - jump (j)
  - jump register (jr)
  - jump and link (jal)
  - jump and link register (jalr)

Based on computation, do something different
In programming languages: if-statement

RISC-V: if-statement instruction is
beq reg1,reg2,L1
means: go to statement labeled L1
if (value in reg1) == (value in reg2)
….otherwise, go to next statement

Change of control flow

We'll talk about these when discuss function calls

Example *if* Statement

- Assuming translations below, compile if block

  f → x10 g → x11 h → x12

  i → x13 j → x14

- if (i == j) bne x13,x14,Exit

  f = g + h; add x10,x11,x12

  Exit:

- May need to negate branch condition

Example:

```
if (i == j)
  f = g + h;
```

Example *if* Statement

- Assuming translations below, compile if block

f → x10 g → x11 h → x12

i → x13 j → x14

```
if (i == j)       bne x13,x14,Else
  f = g + h;      add x10,x11,x12
else              j Exit
  f = g – h;      Else:sub x10,x11,x12
                  Exit:
```

Universität Bremen

Magnitude Compares in RISC-V

- General programs need to test < and > as well.
- RISC-V magnitude-compare branches:

  “Branch on Less Than”

  Syntax: blt reg1,reg2, Label

  Meaning: if (reg1 < reg2) goto Label;

  “Branch on Less Than Unsigned”

  Syntax: bltu reg1,reg2, Label

  Meaning: if (reg1 < reg2)// treat registers as unsigned integers
  goto label;
- Also “Branch on Greater or Equal” bge and bgeu
- Note: No ‘bgt’ or ‘ble’ instructions

C Loop Mapped to RISC-V Assembly

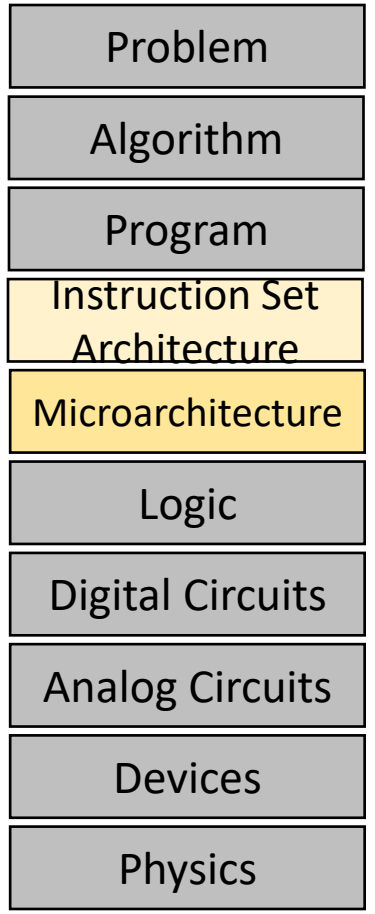
```
int A[20];
// fill A with data
int sum = 0;
for (int i=0; i < 20; i++)
    sum += A[i];
```

```
    add x9, x8, x0          # x9=&A[0]
    add x10, x0, x0         # sum=0
    add x11, x0, x0         # i = 0
    addi x13,x0, 20         # x13 = 20
Loop:
    bge x11,x13,Done
    lw x12, 0(x9)           # x12=A[i]
    add x10,x10,x12         # sum+=x12
    addi x9, x9,4           # &A[i+1]
    addi x11,x11,1          # i++
    j Loop
Done:
```

Universität Bremen

RISC-V Assembly Language

Prof. Dr. Rolf Drechsler
Dr. Muhammad Hassan
M.Sc. Jan Zielasko
M.Sc. Milan Funck



RISC-V

Universität Bremen

Literature

- D. Patterson, J. Hennessy: Computer Organization and Design RISC-V Edition – The Hardware Software Interface, Elsevier, 2020.
- S. L. Harris, D. Harris: Digital Design and Computer Architecture, RISC-V Edition, Elsevier, 2021.
- ARM University program: Introduction-to-Computer-Architecture-Education.
- J. Teich, C. Haubelt: Digitale Hardware/Software-Systeme – Synthese und Optimierung, Springer Verlag, 2. Auflage, 2007.
- G. De Micheli: Synthesis and Optimization of Digital Circuits, McGraw-Hill, 1994.
- G. D. Hachtel, F. Somenzi: Logic Synthesis and Verification Algorithms, Kluwer, 1996.

Universität Bremen

Literature

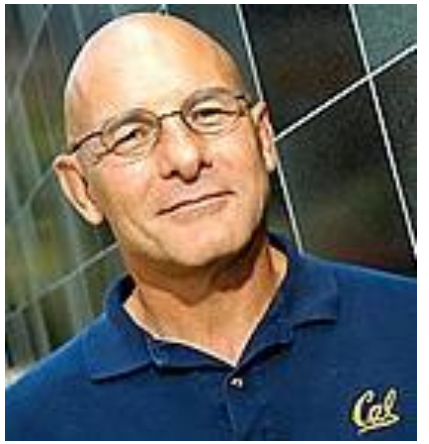
- H. Bähring, J. Dunkel, G. Rademacher: Mikrorechner-Systeme: Mikroprozessoren, Speicher, Peripherie, Springer-Verlag, 2. Auflage, 1994.
- J. P. Hayes: Computer Architecture and Organization, McGraw-Hill, 1998.
- M. G. Arnold: Verilog Digital Computer Design: Algorithms to Hardware, Prentice Hall, 1998.
- A. Tanenbaum: Structured Computer Organization, Prentice Hall, 5th Edition, 2006.
- D. A. Patterson, J. L. Hennessy: Computer Organization and Design - The Hardware/Software-Interface, Morgan Kaufmann, 3. Auflage, 2007.
- J. L. Hennessy, D. A. Patterson: Computer Architecture - A Quantitative Approach, Morgan Kaufmann, 4. Auflage, 2007.
- H. Kaeslin: Digital Integrated Circuit Design, From VLSI to CMOS Fabrication, Cambridge University Press, 2008.
- A. Biere, D. Kroening, G. Weissenbacher, C. M. Wintersteiger: Digitaltechnik – Eine praxisnahe Einführung, Springer-Verlag, 2008.

Kriste Asanovic

- Professor of Computer Science at the University of California, Berkeley
- Developed RISC-V during one summer
- Chairman of the Board of the RISC-V Foundation
- Co-Founder of SiFive, a company that commercializes and develops supporting tools for RISC-V

David Patterson

- Professor of Computer Science at the University of California, Berkeley since 1976
- Coinvented the Reduced Instruction Set Computer (RISC) with John Hennessy in the 1980's
- Founding member of RISC-V team.
- Was given the Turing Award (with John Hennessy) for pioneering a quantitative approach to the design and evaluation of computer architectures.



John Hennessy

- President of Stanford University from 2000 - 2016.
- Professor of Electrical Engineering and Computer Science at Stanford since 1977
- Coinvented the Reduced Instruction Set Computer (RISC) with David Patterson in the 1980’s
- Was given the Turing Award (with David Patterson) for pioneering a quantitative approach to the design and evaluation of computer architectures.

Conditional Branching

```
# RISC-V assembly
addi   s0,    zero,   4        # s0 = 0 + 4 = 4
addi   s1,    zero,   1        # s1 = 0 + 1 = 1
slli   s1,    s1,     2        # s1 = 1 << 2 = 4
beq    s0,    s1,     target   # branch is taken
addi   s1,    s1,     1        # not executed
sub    s1,    s1,     s0       # not executed

target:                       # label
add    s1,    s1,     s0       # s1 = 4 + 4 = 8
```

Labels indicate instruction location. They can't be reserved words and must be followed by a colon (:)

The Branch Not Taken (bne)

```
# RISC-V assembly
 addi   s0,    zero,   4        # s0 = 0 + 4 = 4
 addi   s1,    zero,   1        # s1 = 0 + 1 = 1
 slli   s1,    s1,     2        # s1 = 1 << 2 = 4
 bne    s0,    s1,     target   # branch not taken
 addi   s1,    s1,     1        # s1 = 4 + 1 = 5
 sub    s1,    s1,     s0       # s1 = 5 – 4 = 1

target:                         # label
 add    s1,    s1,     s0       # s1 = 1 + 4 = 5
```

Universität Bremen

Unconditional Branching (j)

```
# RISC-V assembly
j       target          # jump to target
srai    s1, s1, 2       # not executed
addi    s1, s1, 1       # not executed
sub     s1, s1, s0      # not executed

target:
add     s1, s1, s0      # s1 = 1 + 4 = 5
```