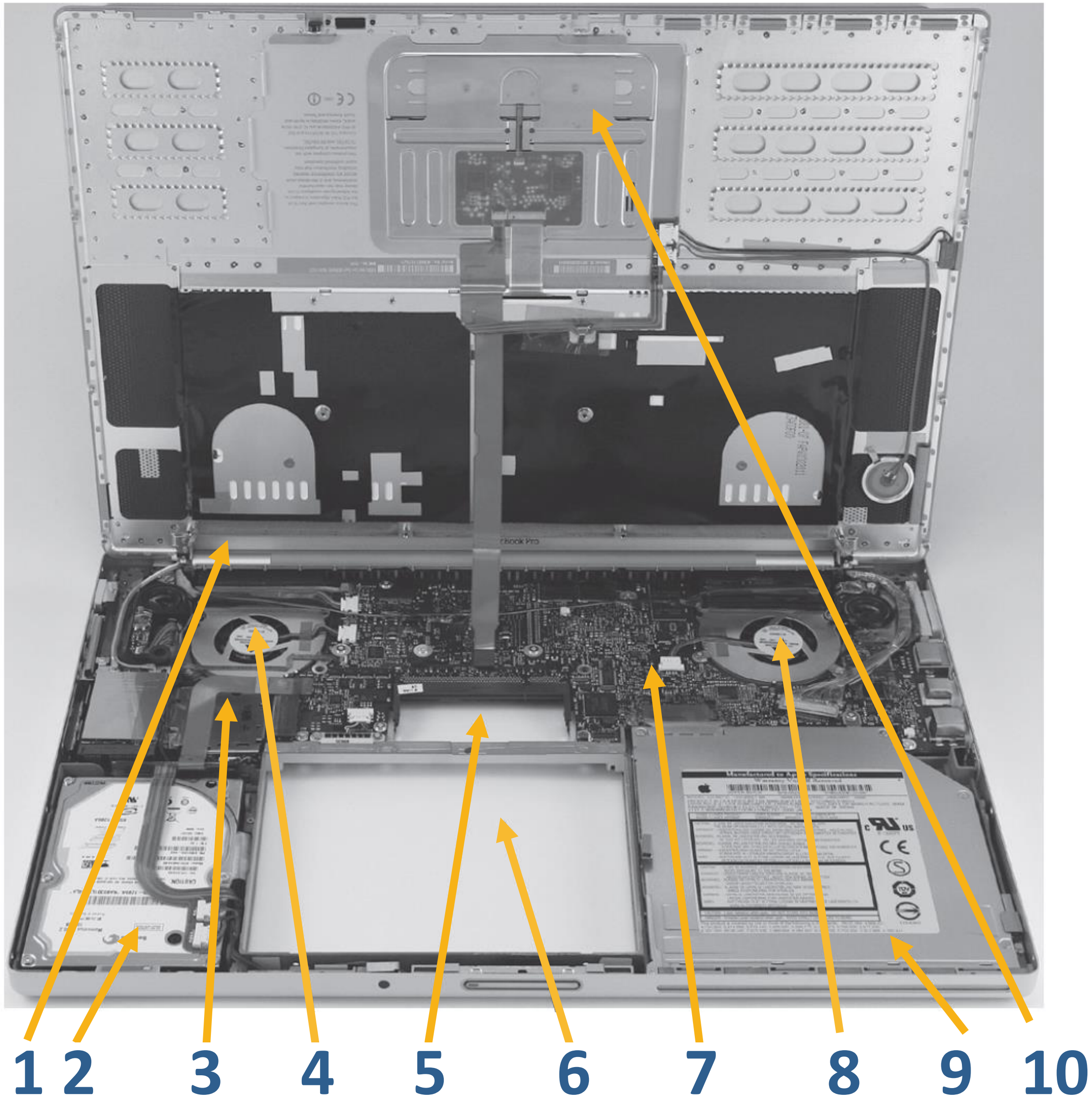
NATIONAL RESEARCH UNIVERSITY

Computer Architecture and Operating Systems
Lecture 3: Computer Architecture

Andrei Tatarnikov

atatarnikov@hse.ru
@andrewt0301

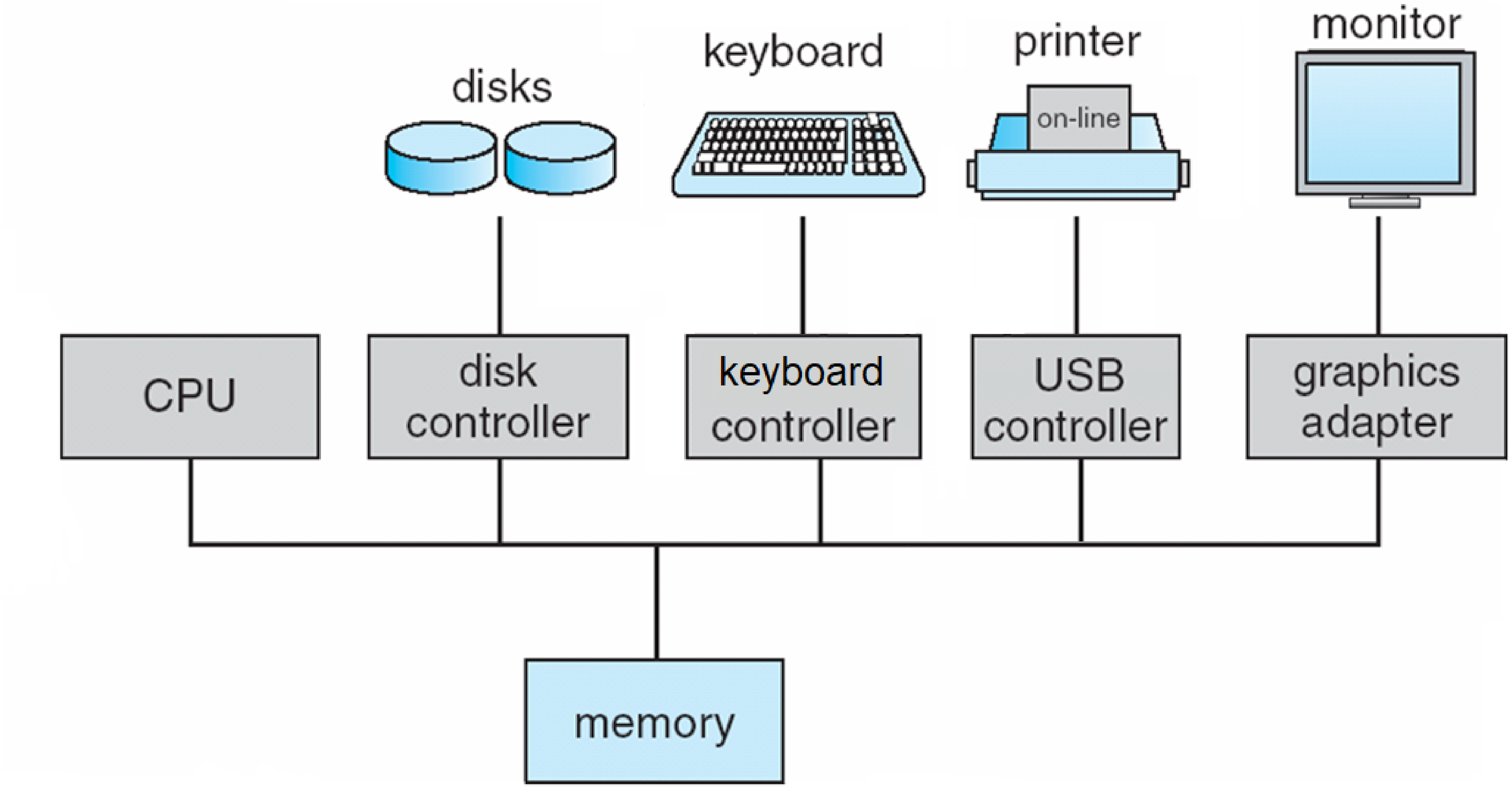
Computer Under Cover



1. Monitor
2. Hard drive
3. CPU (Processor)
4. Fan with cover
5. Spot for memory DIMMs
6. Spot for battery
7. Motherboard
8. Fan with cover
9. DVD drive
10. Keyboard

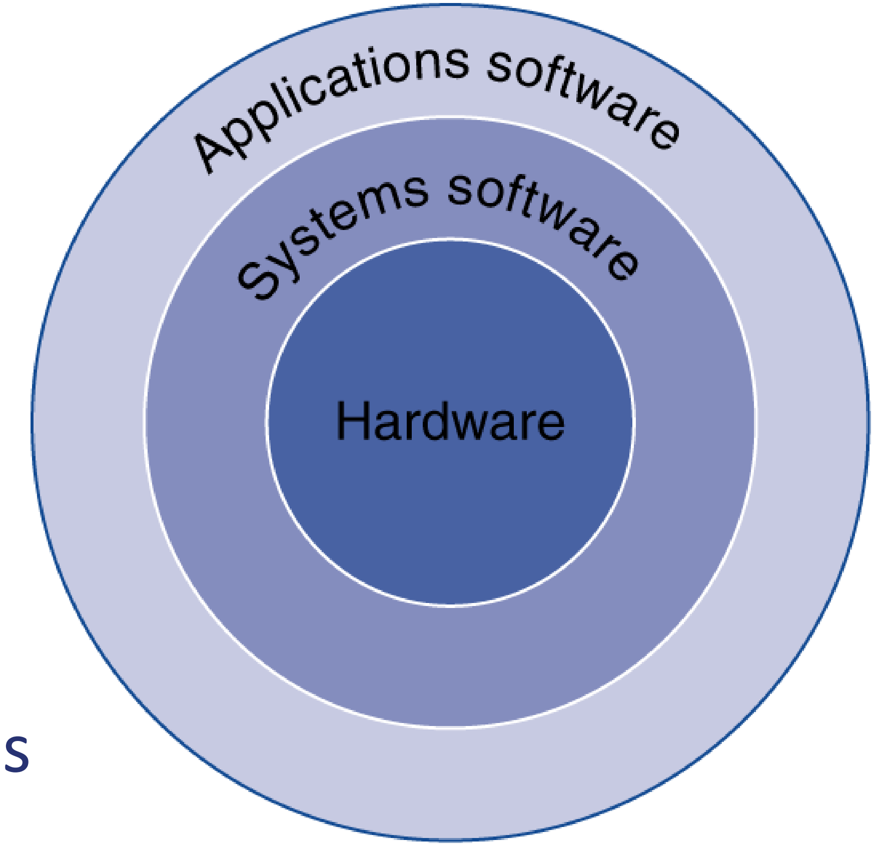
Computer Organization

- One or more CPUs and device controllers connected through a bus providing access to shared memory



Program Under Hood

- Application software
  - Written in high-level language
- System software
  - Compiler: translates high-level language code to machine code
  - Operating System: service code
    - Handling input/output
    - Managing memory and storage
    - Scheduling tasks & sharing resources
- Hardware
  - CPU, memory, I/O controllers



Levels of Program Code

High-level language program (in C)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

Compiler

Assembly language program (for RISC-V)

```
swap:
      slli x6, x11, 3
      add  x6, x10, x6
      ld   x5, 0(x6)
      ld   x7, 8(x6)
      sd   x7, 0(x6)
      sd   x5, 8(x6)
      jalr x0, 0(x1)
```

Assembler

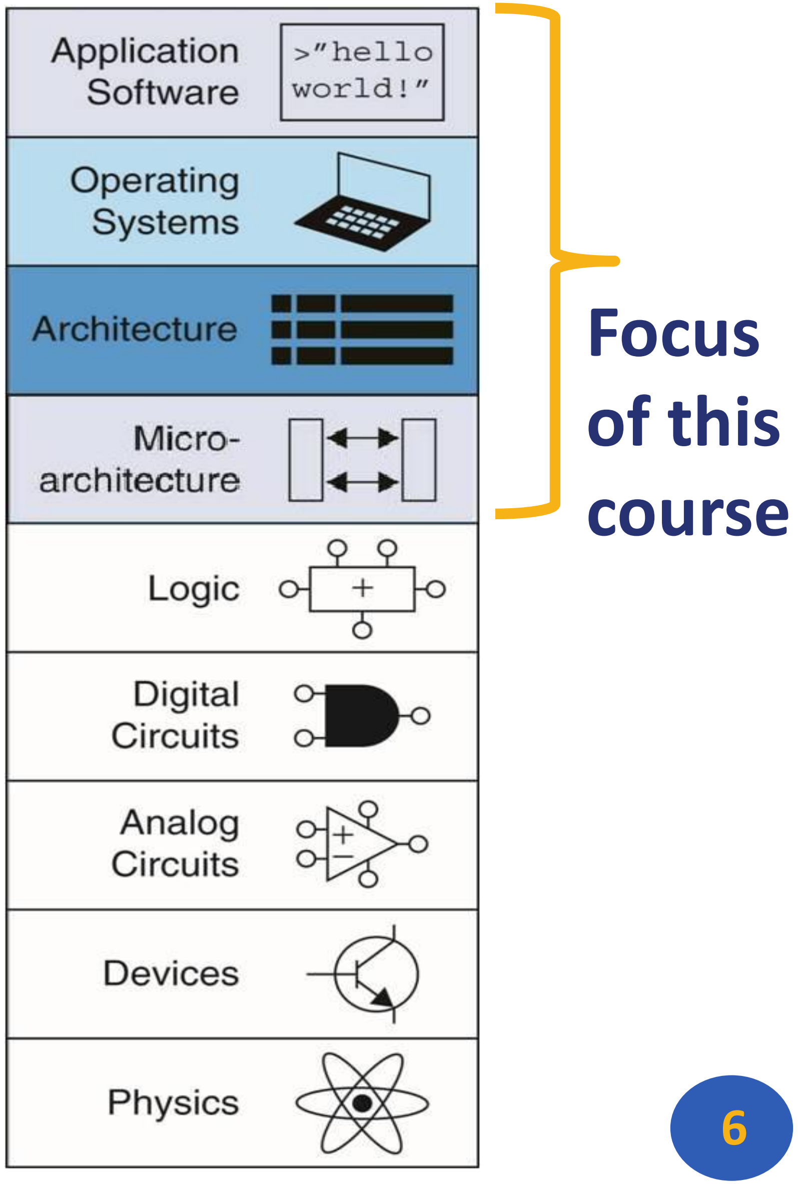
Binary machine language program (for RISC-V)

```
00000000001101011001001100010011
00000000011001010000001100110011
00000000000000110011001010000011
00000000100000110011001110000011
00000000011100110011000000100011
00000000010100110011010000100011
00000000000000001000000001100111
```

- High-level language
  - Level of abstraction closer to problem domain
  - Provides productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data

Abstractions

- Abstraction helps us deal with complexity
  - Hide lower-level detail
- Instruction set architecture (ISA)
  - The hardware/software interface
- Application binary interface (ABI)
  - The ISA plus system software interface
- Implementation (microarchitecture)
  - The details underlying the interface

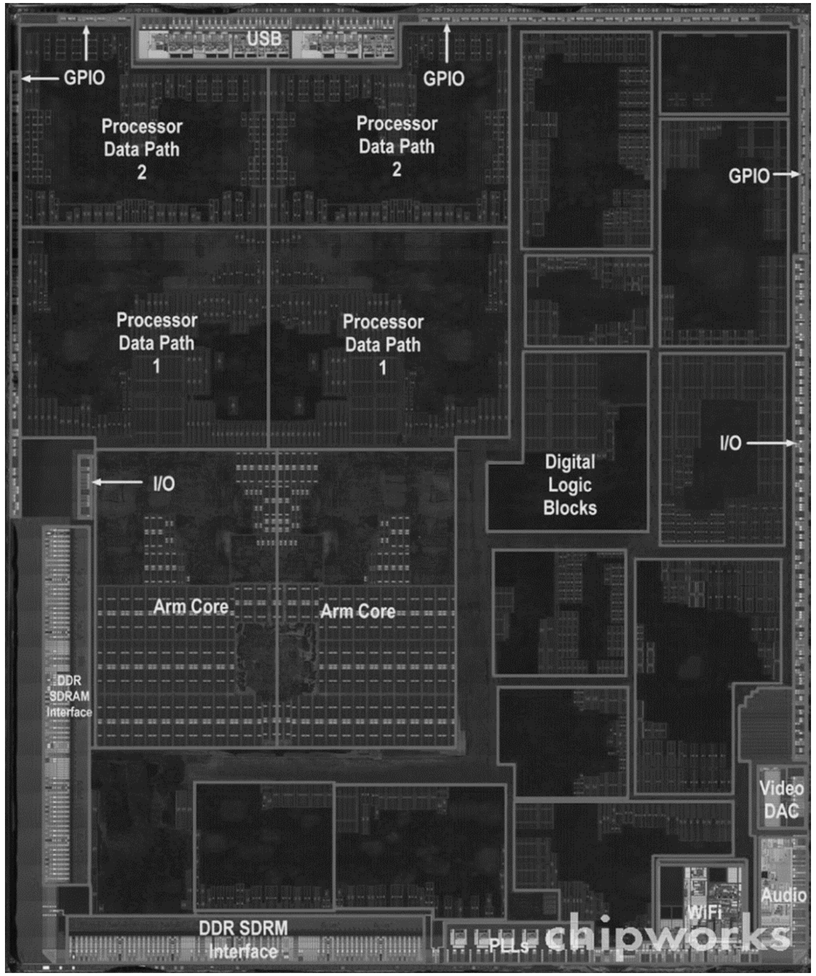


Inside the Processor (CPU)

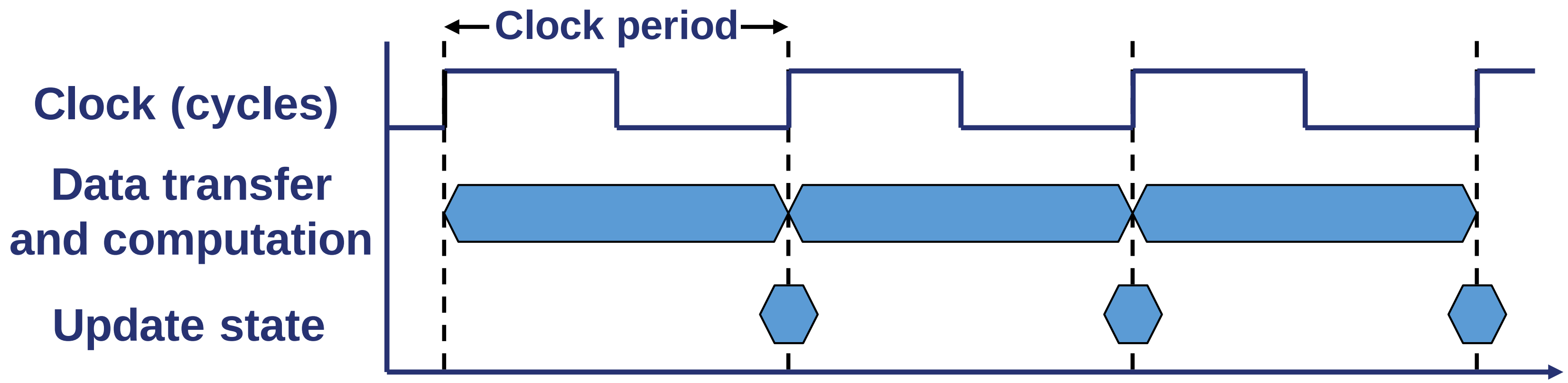
Central Processing Unit (CPU) is the heart of any computer system.

Main components:

- **Register file:** small fast memory for immediate access to data
- **Datapath:** performs operations on data
- **Control unit**: sequences datapath, memory, etc.



CPU Clocking



- Operation of digital hardware governed by a constant-rate clock
- Clock period: duration of a clock cycle
  - e.g., 250 ps = 0.25 ns = 250×10^{-12} s
- Clock frequency (rate): cycles per second
  - e.g., 4.0 GHz = 4000 MHz = 4.0×10^{9} Hz

CPU Time

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Instruction Set Architecture (ISA)

Instruction set architecture (ISA) is the interface between the hardware and the lowest-level software. This is one of the most important abstractions.

ISA Classification

- Complex instruction set computer (CISC)
  - x86/x64 (Intel and AMD)
- Reduced instruction set computer (RISC)
  - ARM, PowerPC, MIPS, RISC-V
- Very long instruction word (VLIW)
  - Itanium, Elbrus

Reduced Instruction Set Computing (RISC)

Reduced Instruction Set Computing (RISC) concept was proposed by teams of researchers at **Stanford University** (John Hennessy) and **University of California Berkeley** (David Paterson) in **early 1980s** as an alternative of Complex Instruction Set Computing (CISC) dominating at that time.

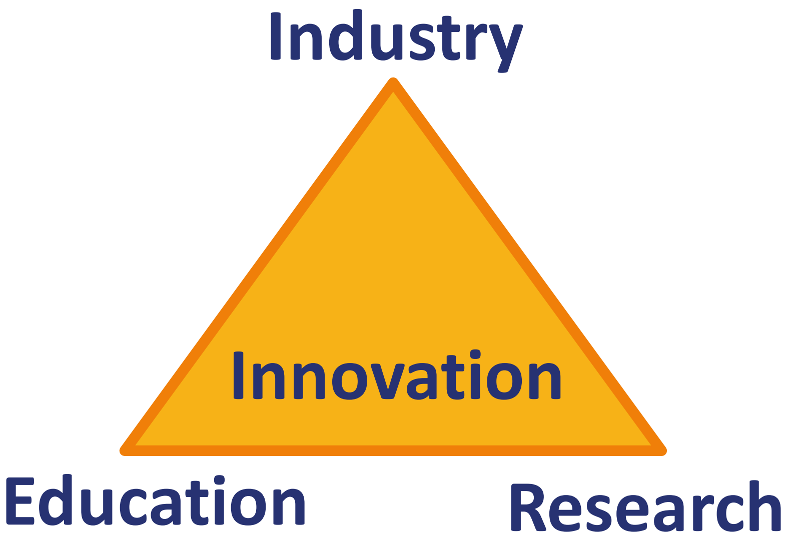
RISC Principles

- All instructions are executed by hardware
- Maximize the rate at which instructions are issued
- Instructions should be easy to decode
- Only loads and stores should reference memory
- Provide plenty of registers

RISC-V ISA

- Simple ISA by UC Berkeley (2010)
- Open and Free
- Wide-Purpose Configurable ISA (from IoT to mainframes)
- Maintained by RISC-V Foundation (moved to Switzerland)
- Supported by many IT Companies and Universities

RISC-V



Berkeley
UNIVERSITY OF CALIFORNIA

RISC-V Community

Wide Support of IT Companies (except Intel and ARM) and Universities

Alibaba Cloud

HUAWEI

ANDES TECHNOLOGY

StarFive 赛昉科技

Western Digital

NVIDIA

SAMSUNG

NXP

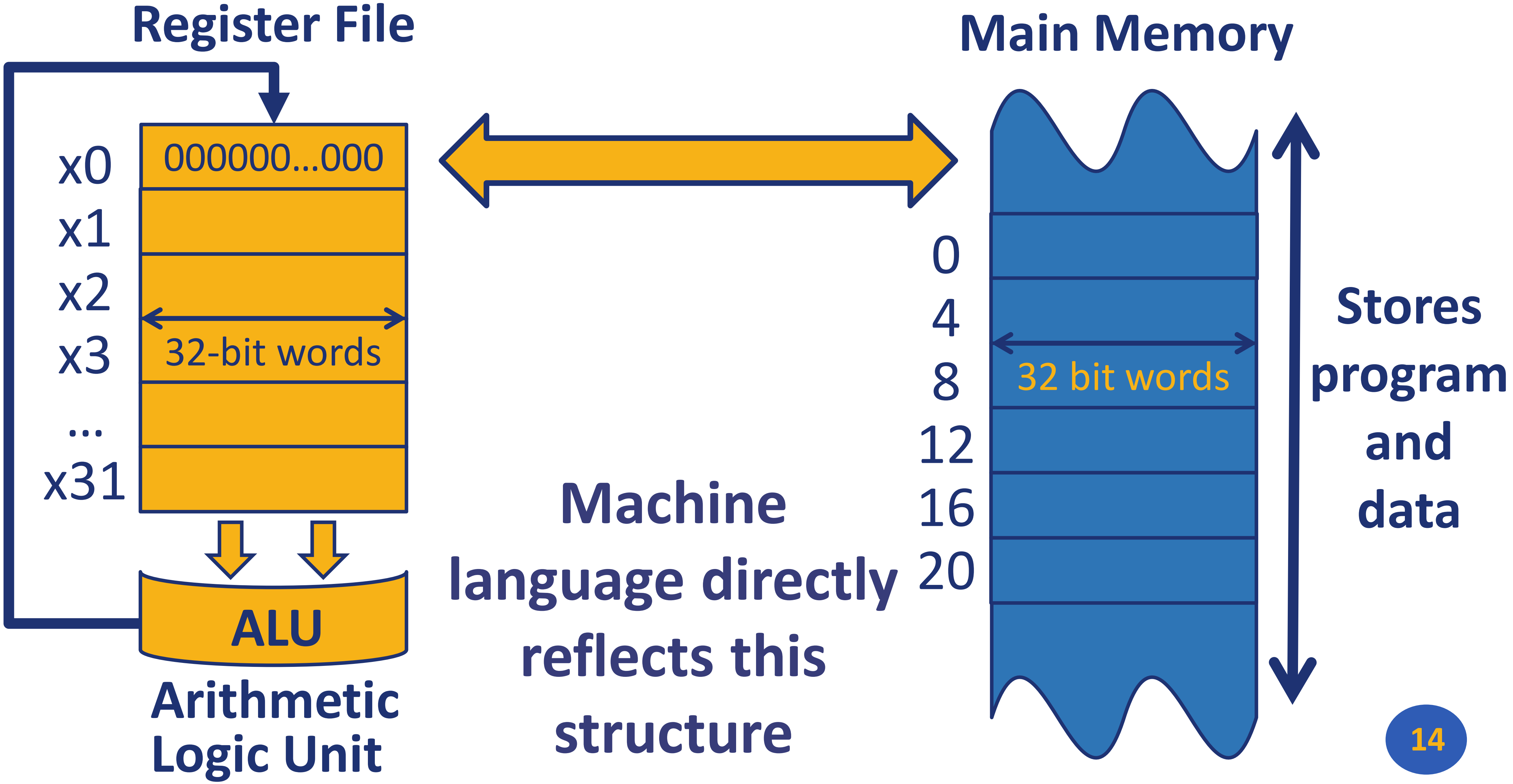
ETH zürich

IBM

Google

and many others…

How CPU Works

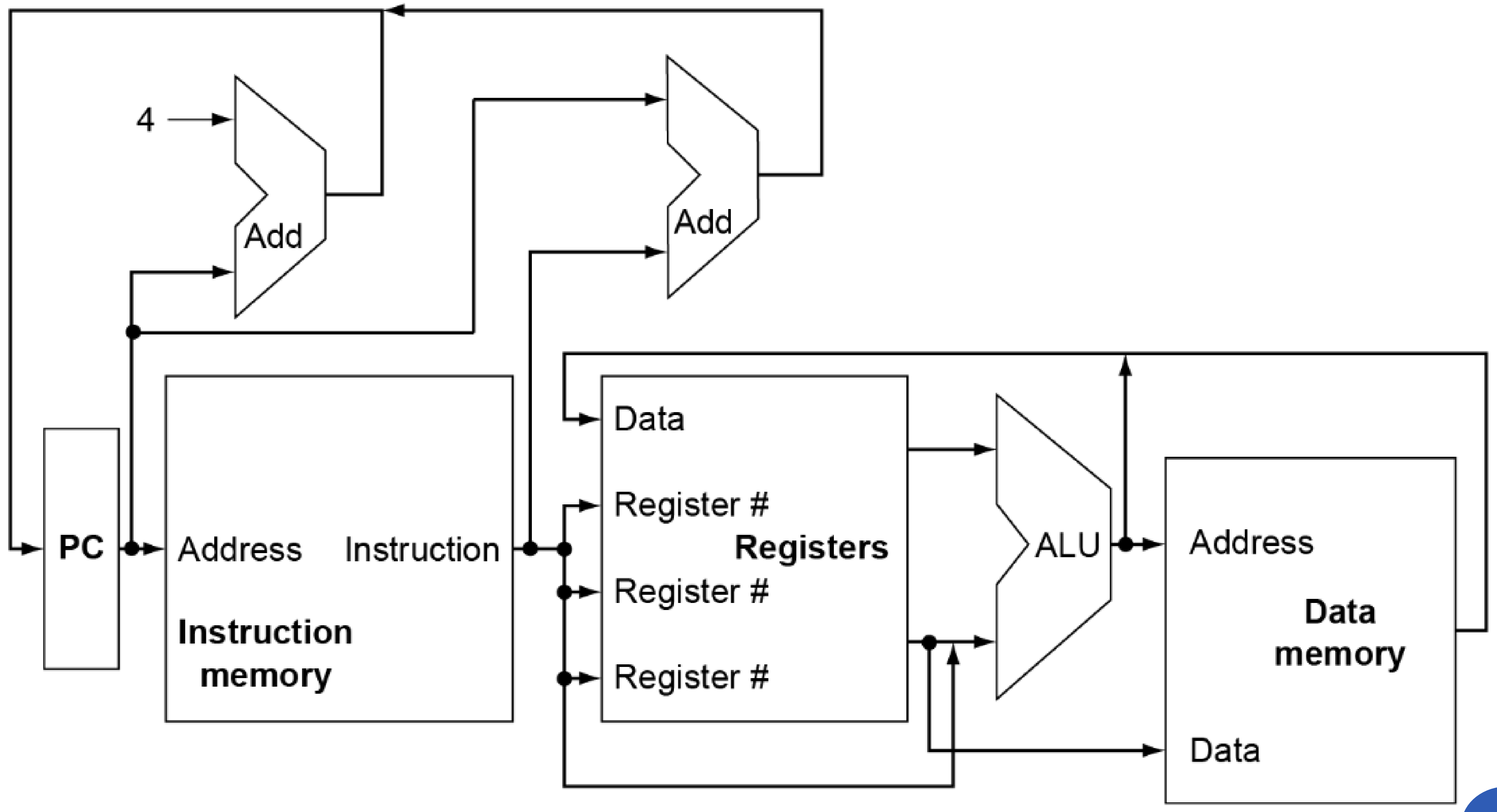


Machine language directly reflects this structure

Instruction Execution

1. Fetch next instruction from memory into instruction register
2. Change program counter to point to next instruction
3. Determine type of instruction just fetched
4. If instructions uses word in memory, determine where Fetch word, if needed, into CPU register
5. Execute the instruction
6. Go to step 1 to begin executing following instruction

RISC-V CPU Scheme



RISC-V General-Purpose Registers

Register	Name	Use	Saver
x0	zero	constant 0	n/a
x1	ra	return addr	caller
x2	sp	stack ptr	callee
x3	gp	gbl ptr	
x4	tp	thread ptr	
x5-x7	t0-t2	temporaries	caller
x8	s0/fp	saved/ frame ptr	callee
x9	s1	saved	callee
x10-x17	a0-a7	arguments	caller
x18-x27	s2-s11	saved	callee
x28-x31	t3-t6	temporaries	caller

32 Registers

32 (or 64) Bits Wide

RISC-V Instructions

- Fixed-size 32 bit instructions
- Always three operands: d -> op(s, t)
- Instruction types
  - Computational instructions
  - Load-store instructions
  - Control-transfer instructions
  - System instructions
- All operations done with registers

Assembly Programming

High Level Language vs Assembly Language

High Level Language	Assembly Language
1. Primitive arithmetic and logical operations	1. Primitive arithmetic and logical operations
2. Complex data types and data structures	2. Primitive data structures – bits and integers
3. Complex control structures – conditional statements, loops and procedures	3. Control transfer instructions
4. Not suitable for direct implementation in hardware	4. Designed to be directly implementable in hardware ***tedious programming!***

Computational Instructions

- Arithmetic, comparison, logical, and shift operations.
- Register-Register Instructions:
  - 2 source operand registers
  - 1 destination register
  - Format: op dest, src1, src2

Arithmetic	Comparisons	Logical	Shifts
add, sub	slt, sltu	and, or, xor	sll, srl, sra

add x3, x1, x2 — x3 <- x1 + x2

slt x3, x1, x2 — if x1 < x2 then x3 = 1 else x3 = 0

and x3, x1, x2 — x3 <- x1 & x2

sll x3, x1, x2 — x3 <- x1 << x2

Register-Immediate Instructions

- One operand comes from a register and the other is a small constant that is encoded into the instruction.
  - Format: op dest, src1, src2

Format	**Arithmetic**	**Comparisons**	**Logical**	**Shifts**
Register-Register	add, sub	slt, sltu	and, or, xor	sll, srl, sra
Register-Immediate	addi	slti, sltiu	andi, ori, xori	slli, srli, srai

```
addi x3, x1, 3     x3 <- x1 + 3
andi x3, x1, 3     x3 <- x1 & 3
slli x3, x1, 3     x3 <- x1 << 3
addi x3, x1, -3    x3 <- x1 - 3
```

No subi, instead use negative constant.

Compound Computations

- Execute a = ((b+3) >> c) - 1;
  - Break up complex expression into basic computations.
    - Our instructions can only specify two source operands and one destination operand (also known as three address instruction).
  - Assume a, b, c are in registers x1, x2, and x3 respectively. Use x4 for t0, and x5 for t1.

```
addi x4, x2, 3        t0 = b + 3;
srl  x5, x4, x3       t1 = t0 >> c;
addi x1, x5, -1       a = t1 - 1;
```

Control Flow Instructions

- Need Conditional branch instructions:
  - Format: comp src1, src2, label
  - First performs comparison to determine if branch is taken or not: src1 comp src2
  - If comparison returns True, then branch is taken, else continue executing program in order.

```
    bge  x1, x2, else
    addi x3, x1, 1
    beq  x0, x0, end
else:
    addi x3, x2, 2
end:
```

```
if (a < b):  c = a + 1
else:        c = b + 2
```

assume

x1=a; x2=b; x3=c;

Unconditional Control Instructions: Jumps

- jal: Unconditional jump and link
  - Example: jal x3, label
  - Jump target specified as label
  - label is encoded as an offset from current instruction
  - Link (to be discussed later): is stored in x3
- jalr: Unconditional jump via register and link
  - Example: jalr x3, 4(x1)
  - Jump target specified as register value plus constant offset
  - Example: Jump target = x1 + 4
  - Can jump to any 32 bit address – supports long jumps

Constants and Instruction Encoding Limits

- Instructions are encoded as 32 bits
  - Need to specify operation (10 bits)
  - Need to specify 2 source registers (10 bits) or 1 source register (5 bits) plus a small constant
  - Need to specify 1 destination register (5 bits)
- The constant in register-immediate instructions has to be smaller than 12 bits; bigger constants have to be stored in the memory or a register and then used explicitly
- The constant in a jal instruction is 20 bits wide (7 bits for operation, and 5 bits for register)

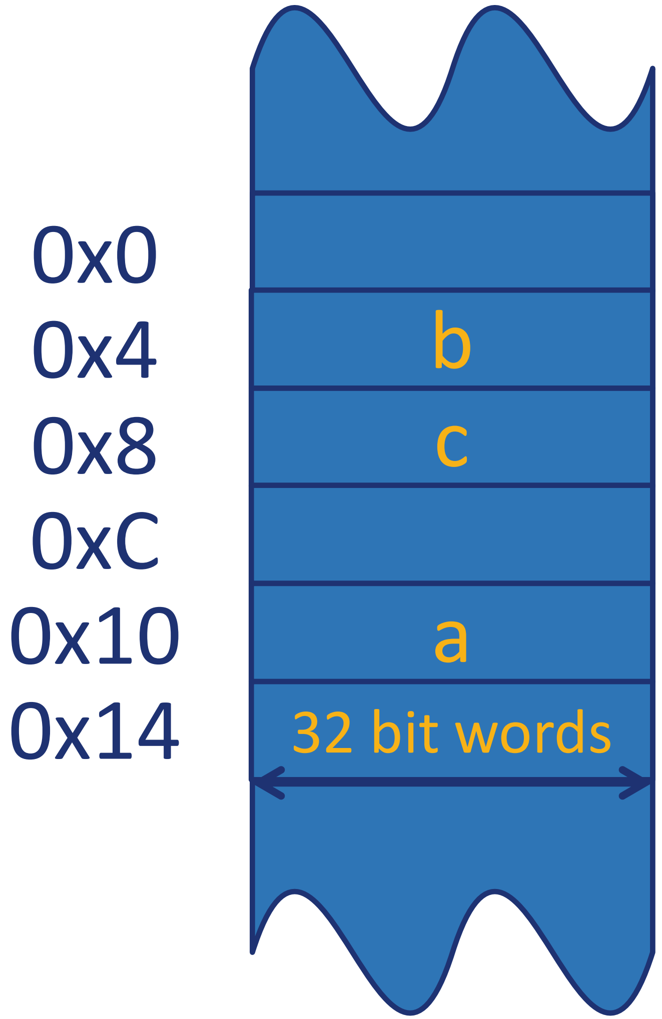
Computations on Values in Memory

a = b + c

x1 <- load(Mem[b])
x2 <- load(Mem[c])
x3 <- x1 + x2
store(Mem[a]) <- x3

x1 <- load(0x4)
x2 <- load(0x8)
x3 <- x1 + x2
store(0x10) <- x3

Main Memory



Load and Store Instructions

- Address is specified as a <base address, offset> pair:
  - Base address is always stored in a register
  - Offset is specified as a small constant
  - Format: lw dest, offset(base)   sw src, offset(base)

lw x1, 0x4(x0)	x1 <- load(Mem[x0 + 0x4])
lw x2, 0x8(x0)	x2 <- load(Mem[x0 + 0x8])
add x3, x1, x2	x3 <- x1 + x2
sw x3, 0x10(x0)	store(Mem[x0 + 0x10]) <- x3

Pseudoinstructions

- Aliases to other actual instructions to simplify assembly programming.

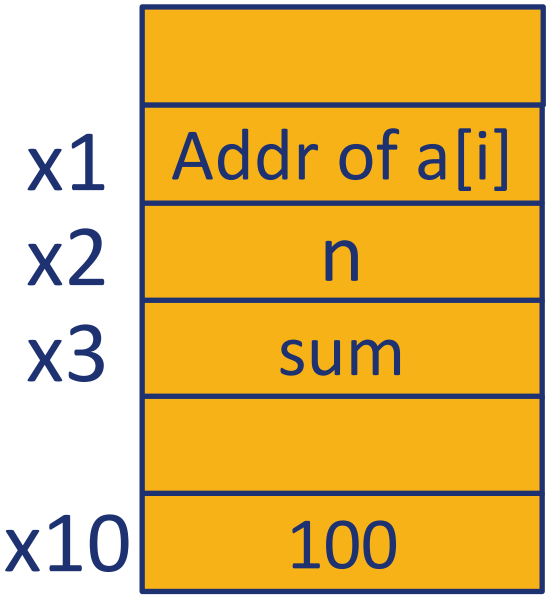
Pseudoinstruction:	Equivalent Assembly Instruction:
mv x2, x1	addi x2, x1, 0
li x2, 3	addi x2, x0, 3
ble x1, x2, label	bge x2, x1, label
beqz x1, label	beq x1, x0, label
bnez x1, label	bne x1, x0, label
j label	jal x0, label

Example: Program to Sum Array Elements

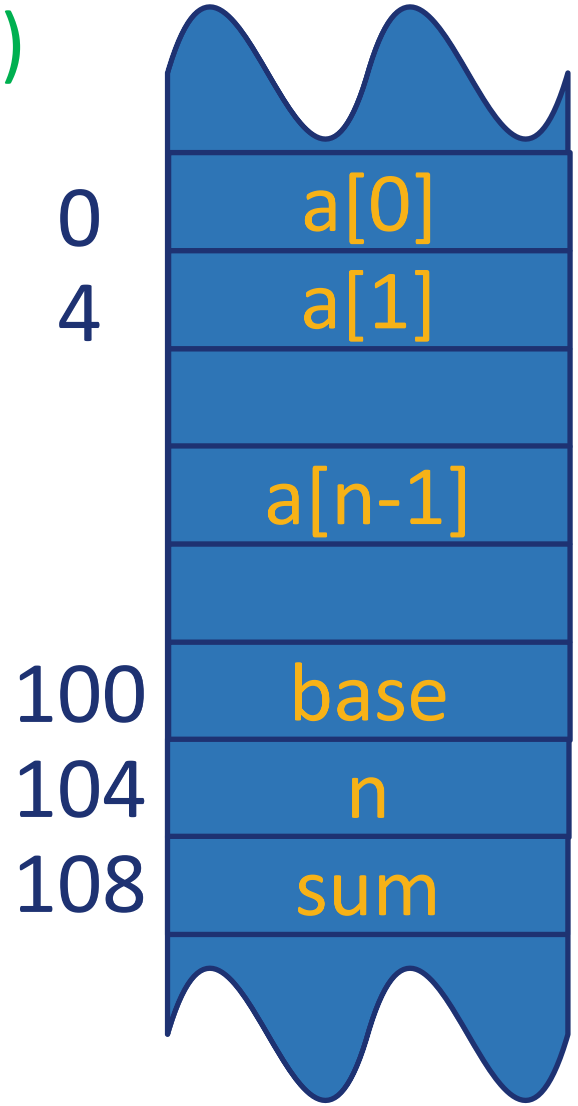
sum = a[0] + a[1] + a[2] + ... + a[n-1]
(assume base address 100 is already in x10)

```
    lw    x1, 0x0(x10)
    lw    x2, 0x4(x10)
    add   x3, x0, x0
loop:
    lw    x4, 0x0(x1)
    add   x3, x3, x4
    addi  x1, x1, 4
    addi  x2, x2, -1
    bnez  x2, loop
    sw    x3, 0x8(x10)
```

Register File



Main Memory



Any Questions?

```
            .text
__start:    addi t1, zero, 0x18
            addi t2, zero, 0x21
cycle:      beq t1, t2, done
            slt t0, t1, t2
            bne t0, zero, if_less
            nop
            sub t1, t1, t2
            j cycle
            nop
if_less:    sub t2, t2, t1
            j cycle
done:       add t3, t1, zero
```