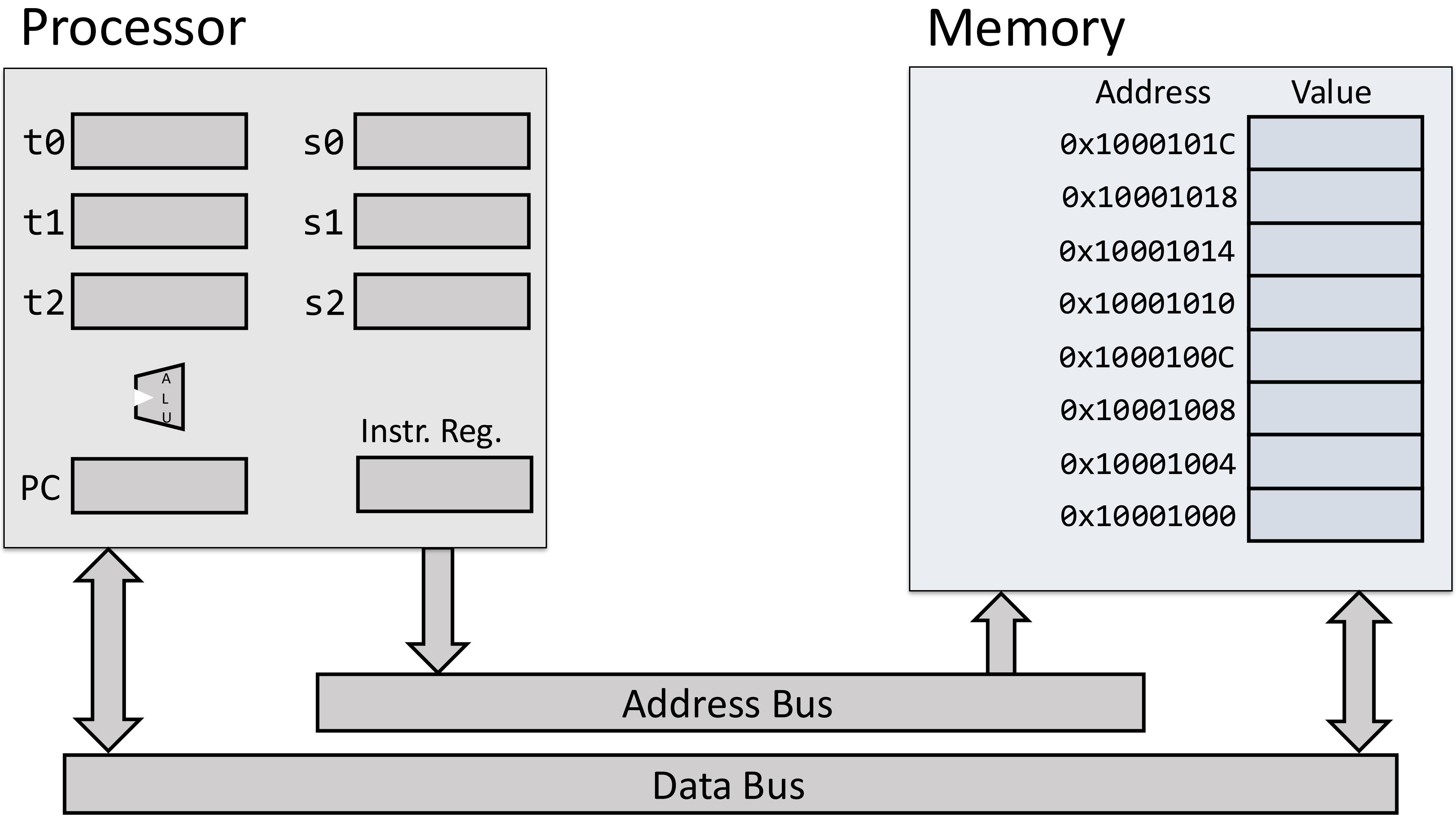
Topic V03

Organization of a Computer
and Memory Addressing

Reading: Section 2.1-2.3

Organization of a Computer



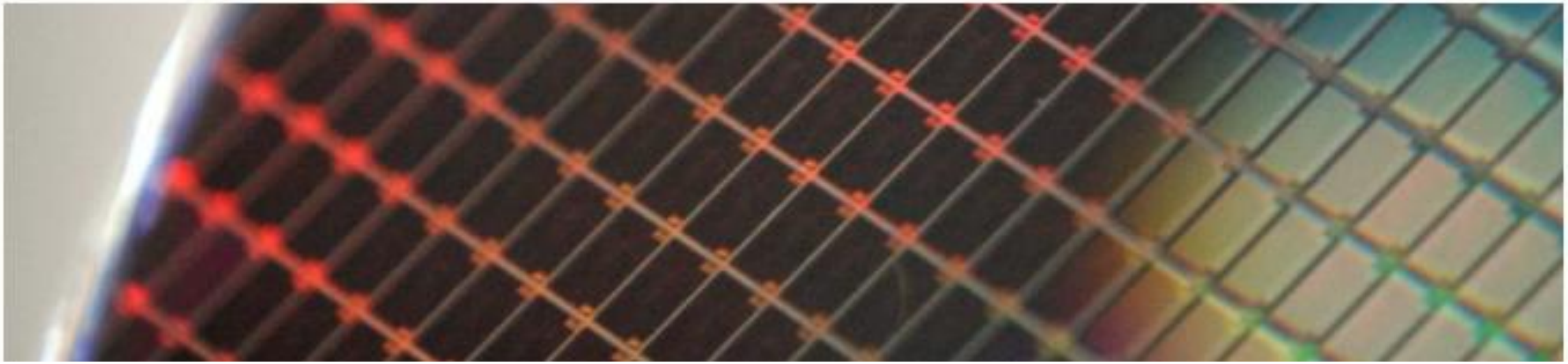
RISC-V®

Krste Asanović at University of California, Berkeley, found many uses for an open-source computer system. In 2010, he decided to develop and publish one in a "short, three-month project over the summer". The plan was to help both academic and industrial users.[10] David Patterson at Berkeley also aided the effort. He originally identified the properties of Berkeley RISC,[11] and RISC-V is one of his long series of cooperative RISC research projects. Early funding was from DARPA.[4]

tps://en.wikipedia.org/wiki/RISC-V

The Difference Between ARM, MIPS, x86, RISC-V And Others In Choosing A Processor Architecture

Jim McGregor Contributor
Tirias Research Contributor Group



https://www.forbes.com/sites/tiriasresearch/2018/04/05/what-you-need-to-know-about-processor-architectures/#7c9ea5c54f57

Arithmetic Operations

Three-operand notation:
two sources and one destination

destination

```
add a, b, c        # a ← b + c
```

sources

```
addi a, b, 20      # a ← b + 20

sub a, b, c        # a ← b - c

addi a, b, -20     # a ← b + -20
```

Compiling a C Assignment

C assignment

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

RISC-V assembly

```
add t0, g, h     # t0 ← g + h
add t1, i, j     # t1 ← i + j
sub f, t0, t1    # f ← t0 - t1
```

How does the processor find f, g, h, i, j?

What do they represent?

Assembly Code for a C Assignment Using Registers

C assignment

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

Must decide which register is allocated for each variable

RISC-V assembly

```
add t0, s1, s2   # t0 ← g + h
add t1, s3, s4   # t1 ← i + j
sub s0, t0, t1   # f ← t0 - t1
```

Assumption

```
f ↔ s0
g ↔ s1
h ↔ s2
i ↔ s3
j ↔ s4
```

Register Operands

Arithmetic instructions use register operands

RISC-V has a 32 x 32-bit register file

- Used for frequently accessed data
- Registers are numbered from 0 to 31
- A ‘word’ is formed by 32 bits

Assembler names

- t0, t1, …, t6 for temporary values
- s0, s1, …, s11 for saved variables
- Textbook uses the notation x0-x31

Addressing an Integer Array

	0	1	2	3	4	5	6	7
A	1	3	5	7	9	11	13	15

Assume that this value

Assume a little-endian machine

Is at this memory address

This address is called the base of array A

Address	Value	Address	Value
0x1000100F	0x00	0x1000101F	0x00
0x1000100E	0x00	0x1000101E	0x00
0x1000100D	0x00	0x1000101D	0x00
0x1000100C	0x07	0x1000101C	0x0F
0x1000100B	0x00	0x1000101B	0x00
0x1000100A	0x00	0x1000101A	0x00
0x10001009	0x00	0x10001019	0x00
0x10001008	0x05	0x10001018	0x0D
0x10001007	0x00	0x10001017	0x00
0x10001006	0x00	0x10001016	0x00
0x10001005	0x00	0x10001015	0x00
0x10001004	0x03	0x10001014	0x0B
0x10001003	0x00	0x10001013	0x00
0x10001002	0x00	0x10001012	0x00
0x10001001	0x00	0x10001011	0x00
0x10001000	0x01	0x10001010	0x09

Addressing an Integer Array

	0	1	2	3	4	5	6	7
A	1	3	5	7	9	11	13	15

Address	Value	Address	Value
0x1000100F	0x00	0x1000101F	0x00
0x1000100E	0x00	0x1000101E	0x00
0x1000100D	0x00	0x1000101D	0x00
0x1000100C	0x07	0x1000101C	0x0F
0x1000100B	0x00	0x1000101B	0x00
0x1000100A	0x00	0x1000101A	0x00
0x10001009	0x00	0x10001019	0x00
0x10001008	0x05	0x10001018	0x0D
0x10001007	0x00	0x10001017	0x00
0x10001006	0x00	0x10001016	0x00
0x10001005	0x00	0x10001015	0x00
0x10001004	0x03	0x10001014	0x0B
0x10001003	0x00	0x10001013	0x00
0x10001002	0x00	0x10001012	0x00
0x10001001	0x00	0x10001011	0x00
0x10001000	0x01	0x10001010	0x09

Addressing an Integer Array

	0	1	2	3	4	5	6	7
A	1	3	5	7	9	11	13	15

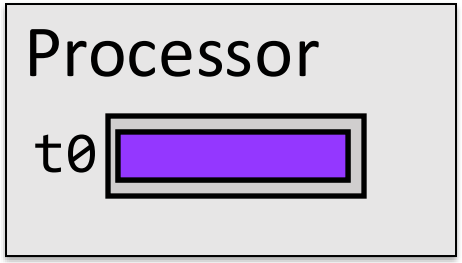
Memory

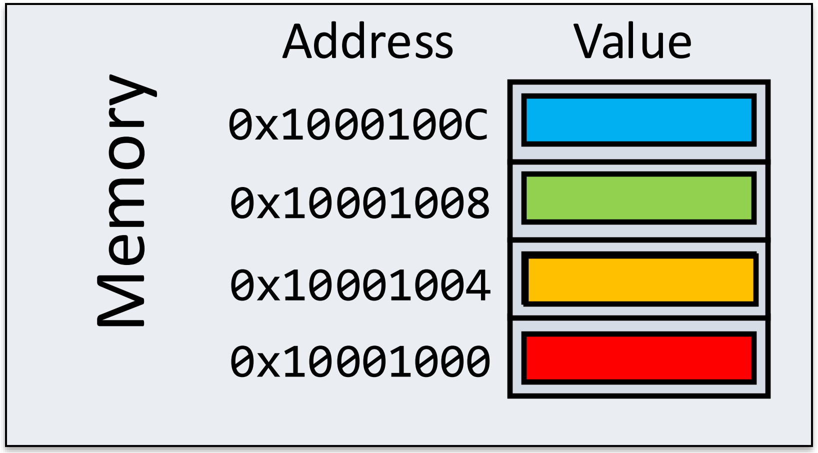
Address	Value
0x1000101C	0x0000000F
0x10001018	0x0000000D
0x10001014	0x0000000B
0x10001010	0x00000009
0x1000100C	0x00000007
0x10001008	0x00000005
0x10001004	0x00000003
0x10001000	0x00000001

Word

Most used data unit in a processor

In a 32-bit RISC-V a word has 4 bytes, or 32 bits





load word

lw loads a word <u>from</u> a specified <u>memory</u> location <u>into</u> a <u>register</u> in the processor

store word

sw stores a word <u>from</u> a <u>register</u> in the processor <u>into</u> a specified <u>memory</u> location

Addressing Memory

What is the RISC-V assembly code for the following C statement?

```
A[0] = h + A[2]
```

RISC-V assembly

```
lw    t0, 8(s1)        # t0 ← A[2]
add   t0, s2, t0       # t0 ← h + A[2]
sw    t0, 0(s1)        # A[0] ← t0
```

Assumption

h ↔ s2
A ↔ s1

Processor

t0	[redacted]

Memory

Address	Value
0x1000100C	
0x10001008	[redacted]
0x10001004	
0x10001000	

Addressing Memory

destination
register for lw

RISC-V assembly

address of
source memory
location for lw

```
lw   t0, 8(s1)     # t0 ← A[2]
add  t0, s2, t0    # t0 ← h + A[2]
sw   t0, 0(s1)     # A[0] ← t0
```

source register
for sw

address of
destination memory
location for sw

Displacement Addressing Mode

RISC-V assembly

```
lw    t0, 8(s1)          # t0 ← A[2]
add   t0, s2, t0         # t0 ← h + A[2]
sw    t0, 0(s1)          # A[0] ← t0
```

Get the value stored in register s1
Add 8 (2 words) to this value

The result is the memory address for the load instruction

Registers vs. Memory

Registers are faster to access than memory

Operating on memory data requires loads and stores

More instructions to be executed

Compiler must use registers for variables as much as possible

Only spill to memory for less frequently used variables

Register optimization is important!

Immediate Operands

Constant data specified in an instruction

```
addi s3, s3, 4      # s3 ← s3 + 4
```

There is no subtract immediate instruction

Just use a negative constant

```
addi s2, s1, -1     # s2 ← s1 + -1
```

Design Principle

Make the common case fast

Small constants are common

Immediate operands avoids a load instruction

Memory Operands

Main memory used for composite data

```
int vec[100];
```

Arrays

```
char *city = “Edmonton”;
```

Strings

```
struct person {
      int age, salary;
      char department[50];
      char name[12];
      char address[100];
};
```

Structures

Memory is Byte Addressed

Address	Value	Address	Value
0x1000100F	0x00	0x1000101F	0x00
0x1000100E	0x00	0x1000101E	0x00
0x1000100D	0x00	0x1000101D	0x00
0x1000100C	0x07	0x1000101C	0x0F
0x1000100B	0x00	0x1000101B	0x00
0x1000100A	0x00	0x1000101A	0x00
0x10001009	0x00	0x10001019	0x00
0x10001008	0x05	0x10001018	0x0D
0x10001007	0x00	0x10001017	0x00
0x10001006	0x00	0x10001016	0x00
0x10001005	0x00	0x10001015	0x00
0x10001004	0x03	0x10001014	0x0B
0x10001003	0x00	0x10001013	0x00
0x10001002	0x00	0x10001012	0x00
0x10001001	0x00	0x10001011	0x00
0x10001000	0x01	0x10001010	0x09

Each address identifies a single 8-bit byte

Words are Aligned in Memory

Address	Value
0x1000100F	0x00
0x1000100E	0x00
0x1000100D	0x00
0x1000100C	0x07
0x1000100B	0x00
0x1000100A	0x00
0x10001009	0x00
0x10001008	0x05
0x10001007	0x00
0x10001006	0x00
0x10001005	0x00
0x10001004	0x03
0x10001003	0x00
0x10001002	0x00
0x10001001	0x00
0x10001000	0x01

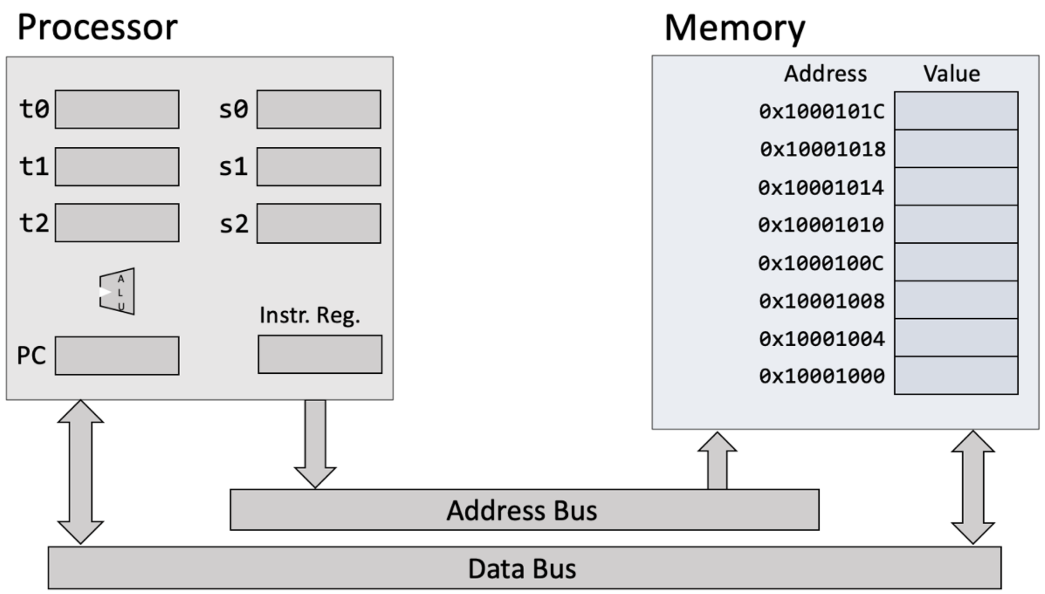
Address	Value
0x1000101F	0x00
0x1000101E	0x00
0x1000101D	0x00
0x1000101C	0x0F
0x1000101B	0x00
0x1000101A	0x00
0x10001019	0x00
0x10001018	0x0D
0x10001017	0x00
0x10001016	0x00
0x10001015	0x00
0x10001014	0x0B
0x10001013	0x00
0x10001012	0x00
0x10001011	0x00
0x10001010	0x09

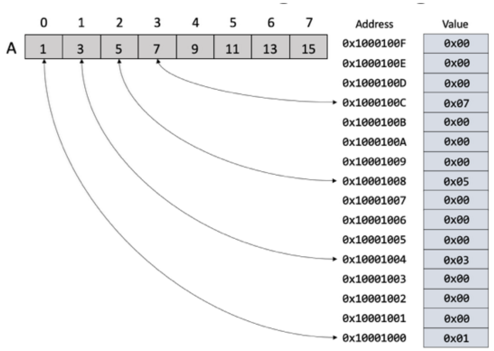
The address of a word must be a multiple of 4

Memory

Address	Value
0x1000101C	0x0000000F
0x10001018	0x0000000D
0x10001014	0x0000000B
0x10001010	0x00000009
0x1000100C	0x00000007
0x10001008	0x00000005
0x10001004	0x00000003
0x10001000	0x00000001

What we learned






```
lw   t0, 8(s1)    # t0 ← A[2]
add  t0, s2, t0   # t0 ← h + A[2
sw  t0, 0(s1)     # A[0] ← t0
```