

# CSC 411

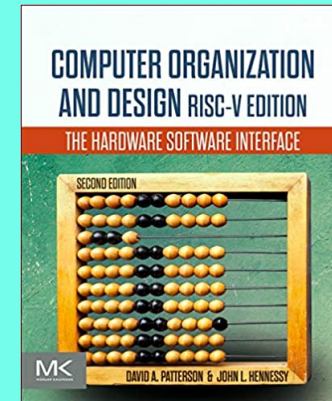
Computer Organization (Spring 2024)  
Lecture 10: RISC-V basics

Prof. Marco Alvarez, University of Rhode Island

## Disclaimer

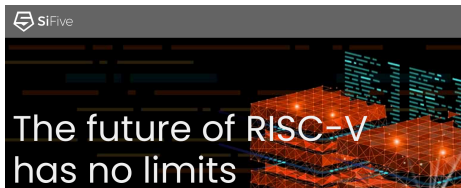
Some figures and slides are adapted from:

Computer Organization and Design (Patterson and Hennessy)  
The Hardware/Software Interface



## The RISC-V instruction set

- We will learn about ISA design by learning RISC-V
  - modern and full featured RISC ISA
  - open source, free, simple, extensible, support heterogeneous and parallel systems, supports 32-bit and 64-bit variants
  - supports (but does not require) IEEE 754
- Developed at UC Berkeley as open ISA (~2010)
  - by 2020, more than 200 companies are members of RISC-V International ([riscv.org](https://riscv.org))
- Similar ISAs have a large share of embedded core market
  - applications in consumer electronics, network/storage equipment, cameras, printers, ...



## RISC-V base and extensions

Name	Description	Version	Status <sup>[A]</sup>	Instruction count
Base				
RVMM0	Weak Memory Ordering	2.0	Ratioed	
RV32I	Base Integer Instruction Set, 32-bit	2.1	Ratioed	40
RV32E	Base Integer Instruction Set (embedded), 32-bit, 16 registers	2.0	Ratioed	40
RV64I	Base Integer Instruction Set, 64-bit	2.1	Ratioed	15
RV64E	Base Integer Instruction Set (embedded), 64-bit	2.0	Ratioed	
RV128I	Base Integer Instruction Set, 128-bit	1.7	Open	15
Extension				
M	Standard Extension for Integer Multiplication and Division	2.0	Ratioed	8 (RV32) 13 (RV64)
A	Standard Extension for Atomic Instructions	2.1	Ratioed	11 (RV32) 22 (RV64)
F	Standard Extension for Single-Precision Floating-Point	2.2	Ratioed	26 (RV32) 30 (RV64)
D	Standard Extension for Double-Precision Floating-Point	2.2	Ratioed	26 (RV32) 32 (RV64)

and many others ... <https://en.wikipedia.org/wiki/RISC-V>

# Instruction set

## Assembly language and ISAs

- assembly language acts as a bridge between the human-readable world and the binary world of machine code
- each assembly language is specific to a particular ISA and reflects its instruction set and architecture
- each line of assembly code represents one instruction for the computer

## Assembly operands are registers

- RISC-V operations can only be performed on registers

```
add x1, x2, x3 # x1 <- x2 + x3
```

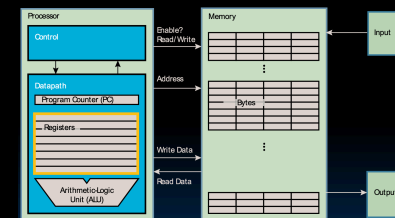
# Registers

## Built directly into the hardware

- limited number of registers available (**register file**)
  - assembly code must be very carefully put together to efficiently use them (mostly done by **compilers**)
- very fast, located within the CPU (0-1 cycles)
  - on a 3 GHz CPU would have an access time of approximately 0.33 ns

## Registers have no type

- contents are just bits, the **“operation”** determines how the bits are interpreted



credit: Berkeley CS61C Great Ideas in Computer Architecture

# RISC-V registers

## RISC-V has 32 integer registers (or 16 in the embedded variant)

- a larger number can lead to an increase in clock cycle time, as the physical distance between them and other components like the ALU increases
- numbered from **x0 to x31** and can be referenced by number or name
  - additional 32 floating-point registers when the floating-point extension is implemented
- each register is 32-bit wide (RV32 variant)

## 32-bit sequences are called a **word** in RV32 and 64-bits sequences a **doubleword**

## RISC-V is a load-store architecture, instructions address only registers,

- load and store instructions convey data to and from memory

## No instructions exist to save and restore multiple registers

# RISC-V registers

Register name	Symbolic name	Description	Saved by
<b>32 integer registers</b>			
x0	Zero	Always zero	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	
x4	tp	Thread pointer	
x5	t0	Temporary / alternate return address	Caller
x6–7	t1–2	Temporary	Caller
x8	s0/fp	Saved register / frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function argument / return value	Caller
x12–17	a2–7	Function argument	Caller
x18–27	s2–11	Saved register	Callee
x28–31	t3–6	Temporary	Caller

<https://en.wikipedia.org/wiki/RISC-V>

## RISC-V registers

32 floating-point extension registers			
f0–7	ft0–7	Floating-point temporaries	Caller
f8–9	fs0–1	Floating-point saved registers	Callee
f10–11	fa0–1	Floating-point arguments/return values	Caller
f12–17	fa2–7	Floating-point arguments	Caller
f18–27	fs2–11	Floating-point saved registers	Callee
f28–31	ft8–11	Floating-point temporaries	Caller

<https://en.wikipedia.org/wiki/RISC-V>

## Arithmetic operations

- Require three register **operands**
- all arithmetic operations have this rigid form: opname, destination, source 1, source 2

opname rd, rs1, rs2

	Addition	Subtraction
c	$a = b + c$	$a = b - c$
RISC-V	<b>add</b> x1, x2, x3	<b>sub</b> x1, x2, x3

## Arithmetic operations

- How would you translate the following C code?
- single line of C may convert into multiple lines in assembly
- some sequences are shorter than others or may use less “temporary” registers

```
// assume these variables are mapped to  
// x5, x1, x2, x3, x4 respectively  
a = b + c + d + e;
```

## Practice

- Translate the following C code into assembly
- if needed, use temporary registers x6, x7

```
// assume these variables are mapped to  
// x5, x1, x2, x3, x4 respectively  
f = (g + h) - (i + j);
```

- optimize the C code to minimize register usage
- good compilers do it all the time

*In fact, if the variables are floating-point values, different sequences of instructions may produce slightly different results. Floating-point operations are not necessarily associative or commutative ...*

## Immediates

- **Immediates are just numerical constants**
  - an immediate operand avoids a **load** instruction
- **Immediate instructions**
  - instructions where constant values may be specified
- **No subtract immediate instruction**
  - there are **add** and **sub** instructions but only **addi** for immediate operands (can use a negative constant)
  - e.g.

```
addi x5, x6, 5  
addi x5, x6, -10
```

## Register **zero**

- **Register **x0** is hardwired to the 0**
  - **cannot be overwritten:** store to the zero register has no effect, and a read always provides 0
- **Useful for common operations**
  - e.g., move between registers, assigning constants to registers