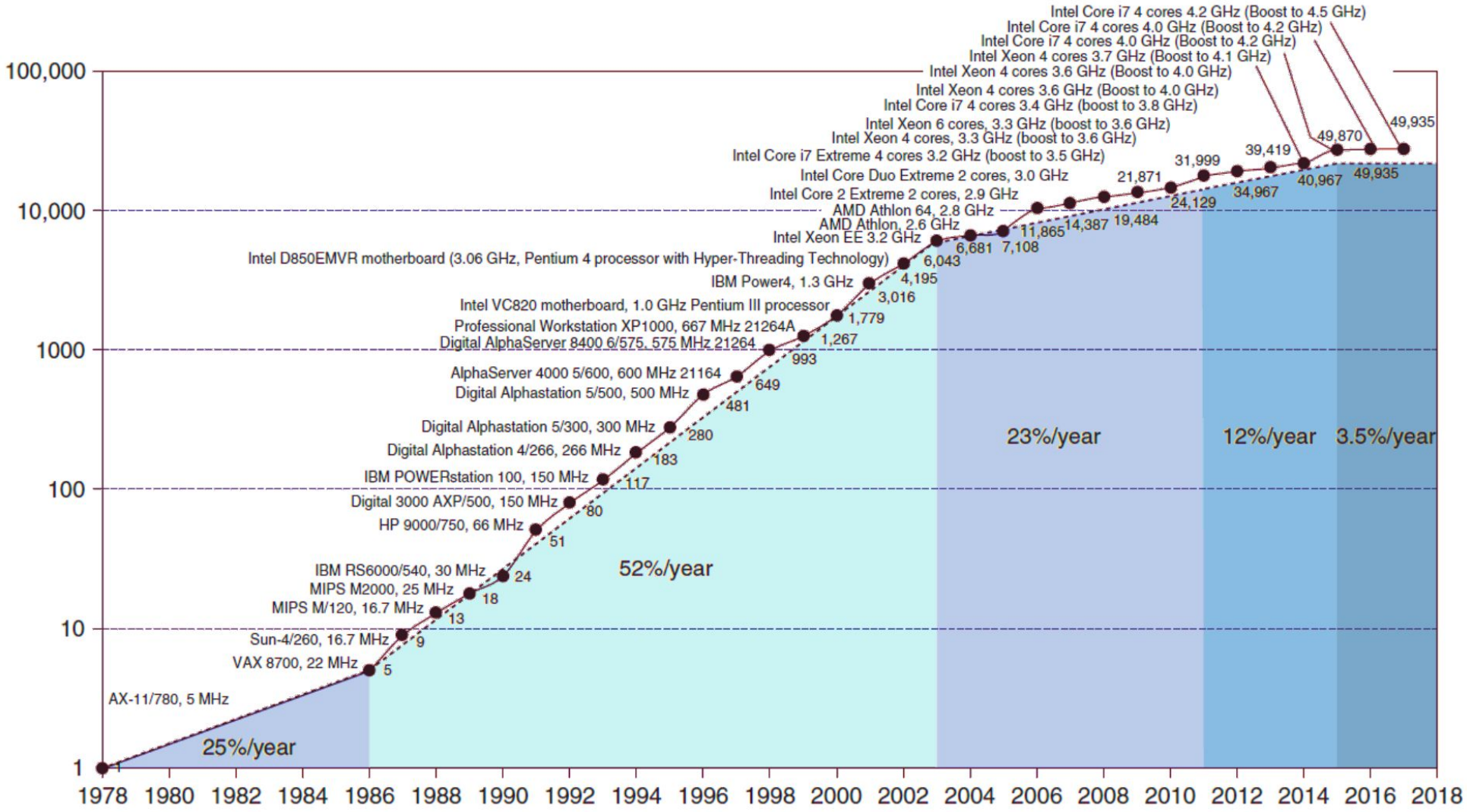
Lecture 2 – ISA

(ECE M116C- CS M151B) Computer Architecture Systems

Nader Sehatbakhsh
Department of Electrical and Computer Engineering
University of California, Los Angeles



*Plot was taken from Hennessy Patterson Book [1].

Is this *it*?

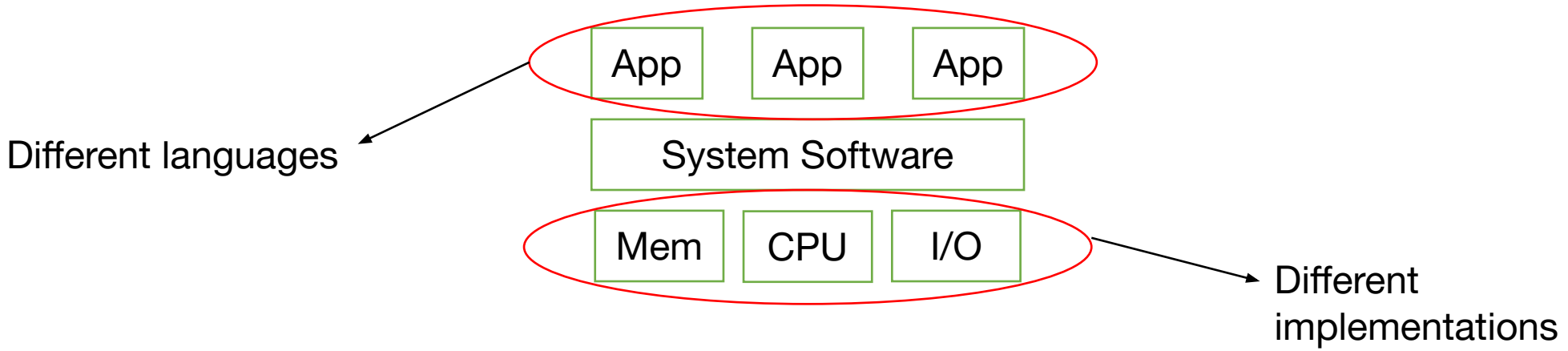
- *NO*

- New Techniques:
  - More Moore
  - More than Moore
  - Beyond CMOS

➔ We will talk about these in W10.

Instruction Set Architecture (ISA)

Running an Application



Challenge: How to maintain compatibility?

Challenge: How to maintain compatibility?

SW

*Image was taken from daria.fandom,com

Let's make this ***contract***:
We (SW and Sys. SW) promise to always give you a program with ***only a set of known instructions***, and you (HW) promise to be able to execute those!

Sys. SW

HW

*Image was taken from Shutterstock.com

ISA

- The contract between software and hardware. Typically described by giving all the programmer-visible state (registers + memory) plus the semantics of the instructions that operate on that state.
- IBM 360 was the first line of machines to separate ISA from implementation (aka. *microarchitecture*).
- Many implementations possible for a given ISA
  - AMD, Intel, VIA processors run the AMD64 ISA
  - many cellphones use the ARM ISA with implementations from many different companies including Apple, Qualcomm, Samsung, Huawei, etc.
- We use RISC-V as standard ISA in class (www.riscv.org)

Design Methodology

- ISA often designed with particular microarchitectural style in mind, e.g.,
  - Accumulator → hardwired, unpipelined
  - CISC → microcoded
  - RISC → hardwired, pipelined
  - VLIW → fixed-latency in-order parallel pipelines
  - JVM → software interpretation
- But can be implemented with any microarchitectural style
  - Intel Ivy Bridge: hardwired pipelined CISC (x86) machine (with some microcode support)
  - Spike: Software-interpreted RISC-V machine
  - ARM Jazelle: A hardware JVM processor

→ *Our Focus: RISC-V (RISC) hardwired, pipelined machine*

What is RISC-V?

What is RISC-V?

- An *open-source* "RISC"-based ISA (*royalty-free*).
- Developed in the 2010s at Berkeley.
- Mostly maintained by the open-source community.
- Different extensions and models. We will focus on the 32-bit "base" mode (i.e., "RV32I").

“RISC”

- “Reduced Instruction Set Computers”
- Alternative: “CISC”

"RISC"

- "Reduced Instruction Set Computers"
- Alternative: "CISC"
- Difference:
  - Instruction size: fixed vs. variable
  - Simple (one-by-one) operation vs. packed operation
  - Complexity

“RISC”

- “Reduced Instruction Set Computers”
- Alternative: “CISC”
- Di
  - Simple (on—by-one) operation vs. packed operation
  - Complexity

Read more here:
https://cs.stanford.edu/people/eroberts/courses/soco/projects/risc/risccisc/

Widely-Used ISAs

- All "RISC" except x86 (Intel's ISA)

- Most popular RISC ISAs:
  - *MIPS, ARM, PowerPC, and RISC-V*

How computers run instructions?

Stored Program Computer (von Neumann)

- Computer hardware is a machine that reads instructions one-by-one and executes them sequentially. It continues this until the program finishes.

➔ *We will break this down in the next few lectures.*

Stored Program Computer (von Neumann)

- Memory holds *both* program and data
  - Instructions and data in a linear memory array
  - Instructions can be modified as data

- Sequential instruction processing
  1. Program Counter (PC) identifies current instruction.
  2. Fetch instruction from memory.
  3. Update state (e.g. PC and memory) as a function of current state according to instruction.
  4. Repeat.

How to build an ISA?

- Transition from HLL to assembly
- No discussion (yet) on how
  a. to generate machine code?
  b. to upload this to hardware?
  c. does hardware run this?

How to build an ISA?

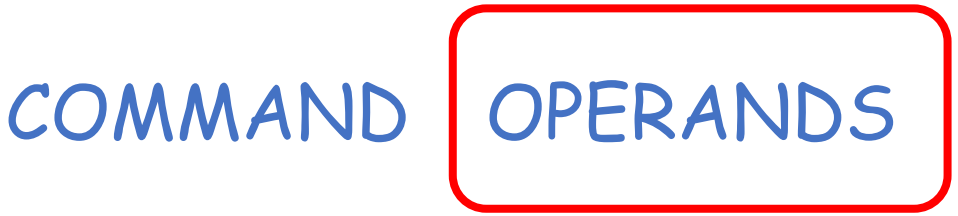
- Transition from HLL to assembly
- No discussion (yet) on how
  a. to generate machine code?
  b. to upload this to hardware?
  c. does hardware run this?

How instructions look like?

COMMAND OPERANDS

In 32-bit RISC-V, each instruction is fixed-size and is 32-bit.

How instructions look like?



In 32-bit RISC-V, each instruction is fixed-size and is 32-bit.

Possible Types of Operands

1. Registers
2. Immediate
3. Memory

What is a register?

- A small storage unit **inside** the processor to quickly access data.

What is a register?

- A small storage unit **inside** the processor to quickly access data.
  - Faster than memory but much smaller (smaller is faster!)
  - Can be seen by Software*!
    - This is something that we will clarify later.
  - Typically between 16 and 64 registers in modern ISAs.

Register Width and Size

- Larger width means easier data transfer but more power.
- More registers means less access to the main memory but requires more area and more power.

Register Width and Size

- Larger width means easier data transfer but more power.
- More registers means less access to the main memory but requires more area and more power.

→ Low-end processors use smaller registers. High-Performance processors use wider and more registers.

RISC-V Registers

- 32 registers, 32-bit each (called word).
  (32 registers, 64-bit each → called double-word).

This is called RV32.

This is called RV64.

RISC-V Registers

- 32 registers, 32-bit each (called word).
  (32 registers, 64-bit each → called double-word).

- Each register shown as *xi*.
- *x0* is hardwired to zero.
- Registers are stored in a data structure called Register File (this is a hardware unit that will be discussed later).

This is called RV32.

This is called RV64.

RISC-V Registers

- 32 registers, 32-bit each (called word).
  (32 registers, 64-bit each → called double-word).

- All 32 registers store values in integers.

- For more high-end processors, there is a separate set of registers for floating point operations.

This is called RV32.

This is called RV64.

Possible Types of Operands

1. Registers
2. Immediate
3. Memory

Immediate

- Constant numbers to be used in an instruction.

- Example:

```
addi x2, x1, 5
```


Nader Sehatbakhsh <nsehat@ee.ucla.edu>

Immediate

- Constant numbers to be used in an instruction.
- Example:

```
addi x2, x1, 5
```

- Registers are 32-bit but immediates *may not. (why?)*
  - ➔ *What should we do then?*

Immediate

- Constant numbers to be used in an instruction.
- *Sign-Extension??*

Immediate: Sign-Extension and Padding

- Many operations in RISC-V are signed. So proper sign-extension is needed when necessary.
- For LSBs, padding zeros is sufficient *(why?)*
- (Today is a good time to review 2's complement…).

Possible Types of Operands

1. Registers
2. Immediates
3. Memory

Possible Types of Operands

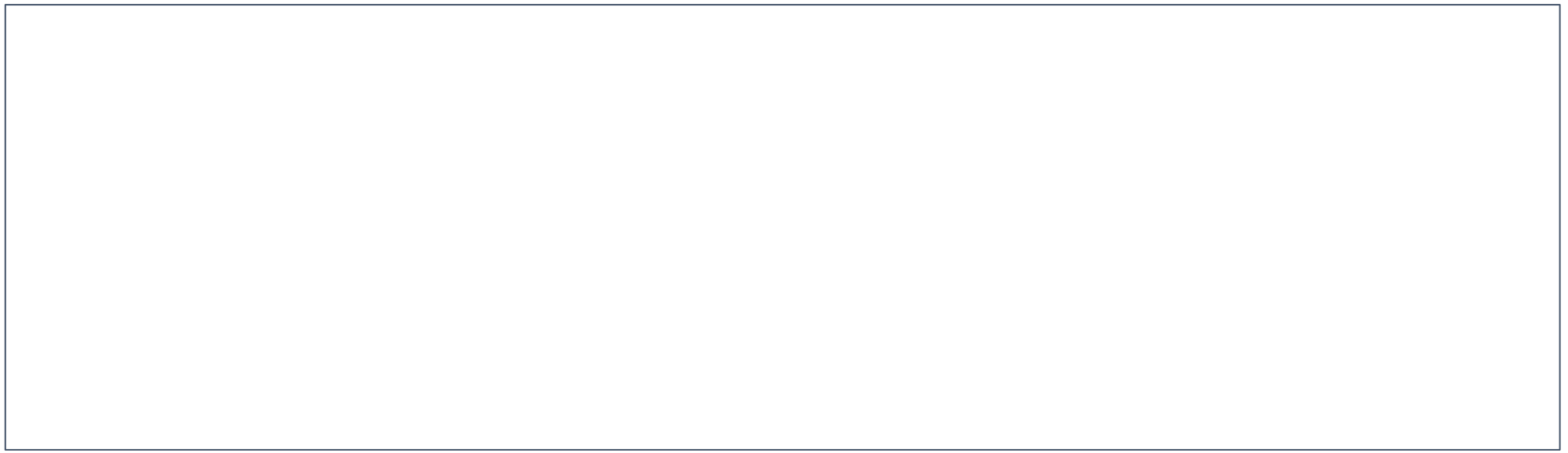
1. Registers

2. Immediates

3. Memory
   - Values are stored in a memory and could be accessed.
   - Memory should be byte-addressable (More about memory later ...)

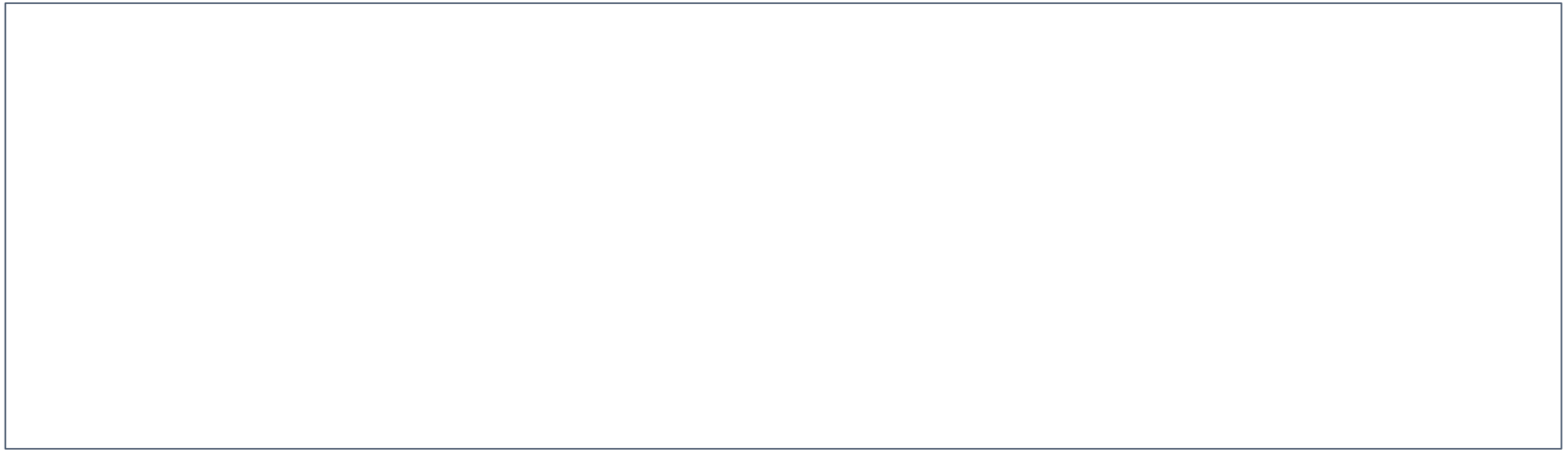
Memory

- Format and addressing
  - Load and store



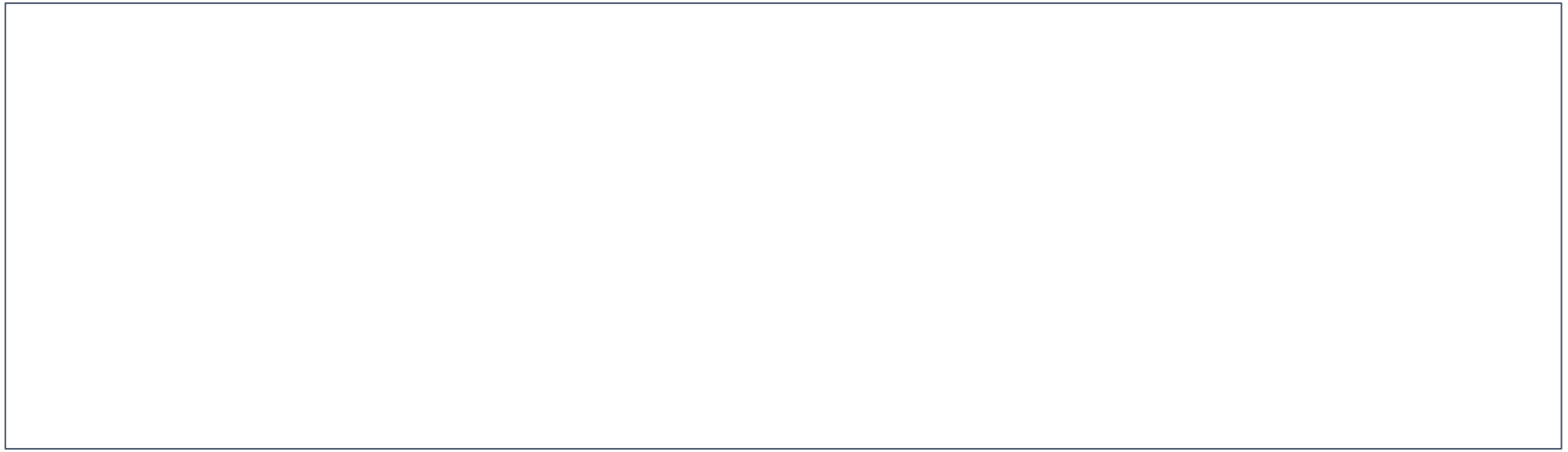
Memory

- Format and addressing
  - Base and offset (base displacement addressing)



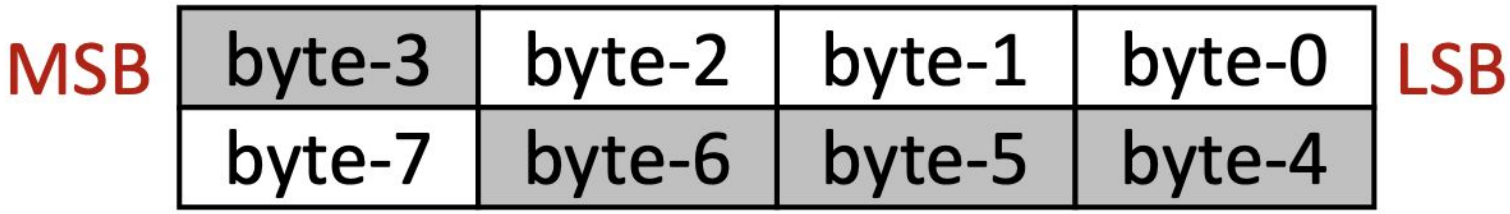
Memory

- Format and addressing
  - Byte-addressability vs. 32-bit data (little endian)



Memory

- Format and addressing
  - Alignment



Memory

- Load and Store Variants
  - LW, LB, LH
  - LBU, LHU

  - SW, SB, SH (what about the sign?)

Possible Types of Operands

1. Registers
2. Immediates
3. Memory

How many operands?

- At most two (for RISC-V). Sometime one, sometimes none.

- Why?
  - Most efficient → Tradeoff between size, complexity, and efficiency

How instructions look like?

COMMAND OPERANDS

In 32-bit RISC-V, each instruction is fixed-size and is 32-bit.

Instruction Types

- Arithmetic/ALU
- Memory
- Control-Flow

Arithmetic/ALU Instruction

- Do an arithmetic operation on
  - two registers or
  - one register and an immediate

  and save the result in another register.

Arithmetic/ALU Instruction

- Source and Destination registers
- Example:

```
SUB x3, x2, x1
ADD x5, x1, 10
```

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »$_u$ constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »$_s$ constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »$_u$ reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »$_s$ reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »$_u$ constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »$_s$ constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »$_u$ reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »$_s$ reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »$_u$ constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »$_s$ constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »$_u$ reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »$_s$ reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <s constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <u constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »u constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »s constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <s reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <u reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »u reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »s reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <s constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <u constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »u constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »s constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <s reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <u reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »u reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »s reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <s constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <u constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »u constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »s constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <s reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <u reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »u reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »s reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »$_u$ constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »$_s$ constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »$_u$ reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »$_s$ reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »$_u$ constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »$_s$ constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »$_u$ reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »$_s$ reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <s constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <u constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »u constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »s constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <s reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <u reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »u reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »s reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

ADDI	`addi rd, rs1, constant`	Add Immediate	`reg[rd] <= reg[rs1] + constant`
SLTI	`slti rd, rs1, constant`	Compare < Immediate (Signed)	`reg[rd] <= (reg[rs1] <$_s$ constant) ? 1 : 0`
SLTIU	`sltiu rd, rs1, constant`	Compare < Immediate (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ constant) ? 1 : 0`
XORI	`xori rd, rs1, constant`	Xor Immediate	`reg[rd] <= reg[rs1] ^ constant`
ORI	`ori rd, rs1, constant`	Or Immediate	`reg[rd] <= reg[rs1] | constant`
ANDI	`andi rd, rs1, constant`	And Immediate	`reg[rd] <= reg[rs1] & constant`
SLLI	`slli rd, rs1, constant`	Shift Left Logical Immediate	`reg[rd] <= reg[rs1] « constant`
SRLI	`srli rd, rs1, constant`	Shift Right Logical Immediate	`reg[rd] <= reg[rs1] »$_u$ constant`
SRAI	`srai rd, rs1, constant`	Shift Right Arithmetic Immediate	`reg[rd] <= reg[rs1] »$_s$ constant`
ADD	`add rd, rs1, rs2`	Add	`reg[rd] <= reg[rs1] + reg[rs2]`
SUB	`sub rd, rs1, rs2`	Subtract	`reg[rd] <= reg[rs1] - reg[rs2]`
SLL	`sll rd, rs1, rs2`	Shift Left Logical	`reg[rd] <= reg[rs1] « reg[rs2]`
SLT	`slt rd, rs1, rs2`	Compare < (Signed)	`reg[rd] <= (reg[rs1] <$_s$ reg[rs2]) ? 1 : 0`
SLTU	`sltu rd, rs1, rs2`	Compare < (Unsigned)	`reg[rd] <= (reg[rs1] <$_u$ reg[rs2]) ? 1 : 0`
XOR	`xor rd, rs1, rs2`	Xor	`reg[rd] <= reg[rs1] ^ reg[rs2]`
SRL	`srl rd, rs1, rs2`	Shift Right Logical	`reg[rd] <= reg[rs1] »$_u$ reg[rs2]`
SRA	`sra rd, rs1, rs2`	Shift Right Arithmetic	`reg[rd] <= reg[rs1] »$_s$ reg[rs2]`
OR	`or rd, rs1, rs2`	Or	`reg[rd] <= reg[rs1] | reg[rs2]`
AND	`and rd, rs1, rs2`	And	`reg[rd] <= reg[rs1] & reg[rs2]`

What is the final value in x3?

```
addi    x1, x0, 3
addi    x2, x0, 5
add     x3, x2, x1
sra     x3, x3, x1
```

Write this in RISC-V Assembly

- Sum of Y= {1,6,8,9}, left-shifted 2 bits and subtracted from its 2-bits right-shifted. (`X0 = 0`)

```
Y= SUM{1,6,8,9}
Z = Y<<2 - Y>>2
```

Instruction Types

- Arithmetic/ALU
- Memory
- Control-Flow

Instruction Types

- Arithmetic/ALU
- Memory
- Control-Flow

Memory-Type Instructions

- Load and Store
- Addressing modes:
  - Base (register) + Offset (immediate)
  - (Two other)
- Data Size and format
  - Word, half-word, byte
  - Signed and unsigned

Memory-Type Instructions

LB	`lb rd, offset(rs1)`	Load Byte	`reg[rd] <= signExtend(mem[addr])`
LH	`lh rd, offset(rs1)`	Load Half Word	`reg[rd] <= signExtend(mem[addr + 1: addr])`
LW	`lw rd, offset(rs1)`	Load Word	`reg[rd] <= mem[addr + 3: addr]`
LBU	`lbu rd, offset(rs1)`	Load Byte (Unsigned)	`reg[rd] <= zeroExtend(mem[addr])`
LHU	`lhu rd, offset(rs1)`	Load Half Word (Unsigned)	`reg[rd] <= zeroExtend(mem[addr + 1: addr])`
SB	`sb rs2, offset(rs1)`	Store Byte	`mem[addr] <= reg[rs2][7:0]`
SH	`sh rs2, offset(rs1)`	Store Half Word	`mem[addr + 1: addr] <= reg[rs2][15:0]`
SW	`sw rs2, offset(rs1)`	Store Word	`mem[addr + 3: addr] <= reg[rs2]`

Instruction Types

- Arithmetic/ALU
- Memory
- Control-Flow

What is control-flow

- Program counter and PC register
- Next PC?

Control-Flow

- Why we need it?
  - Conditions
  - Function calls and returns
  - ...
- Example

Jump Instruction

- Need a destination address (i.e., where to jump?)

➔ PC-relative addressing

JUMP vs. Branch

- Jump always change the PC!

- Branch changes the PC *only if* the condition is TRUE!

JUMP and Link

- Why do we need linking?

- What if we don't need to link?

Control-Flow

JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if $=$	`pc <= (reg[rs1] == reg[rs2]) ? label` `: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if $\neq$	`pc <= (reg[rs1] != reg[rs2]) ? label` `: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if $<$ (Signed)	`pc <= (reg[rs1] <`$_s$` reg[rs2]) ? label` `: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if $\geq$ (Signed)	`pc <= (reg[rs1] >=`$_s$` reg[rs2]) ? label` `: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if $<$ (Unsigned)	`pc <= (reg[rs1] <`$_u$` reg[rs2]) ? label` `: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if $\geq$ (Unsigned)	`pc <= (reg[rs1] >=`$_u$` reg[rs2]) ? label` `: pc + 4`

For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e., pc + imm = label).

Control-Flow

PC-relative vs. Register addressing

JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label` `: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label` `: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <s reg[rs2]) ? label` `: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=s reg[rs2]) ? label` `: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <u reg[rs2]) ? label` `: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=u reg[rs2]) ? label` `: pc + 4`

For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e., pc + imm = label).

Control-Flow

JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label` `: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label` `: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <s reg[rs2]) ? label` `: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=s reg[rs2]) ? label` `: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <u reg[rs2]) ? label` `: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=u reg[rs2]) ? label` `: pc + 4`

For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e., pc + imm = label).

Control-Flow

JAL	`jal rd, label`	Jump and Link	`reg[rd] <= pc + 4` `pc <= label`
JALR	`jalr rd, offset(rs1)`	Jump and Link Register	`reg[rd] <= pc + 4` `pc <= {(reg[rs1] + offset)[31:1], 1'b0}`
BEQ	`beq rs1, rs2, label`	Branch if =	`pc <= (reg[rs1] == reg[rs2]) ? label` `: pc + 4`
BNE	`bne rs1, rs2, label`	Branch if ≠	`pc <= (reg[rs1] != reg[rs2]) ? label` `: pc + 4`
BLT	`blt rs1, rs2, label`	Branch if < (Signed)	`pc <= (reg[rs1] <s reg[rs2]) ? label` `: pc + 4`
BGE	`bge rs1, rs2, label`	Branch if ≥ (Signed)	`pc <= (reg[rs1] >=s reg[rs2]) ? label` `: pc + 4`
BLTU	`bltu rs1, rs2, label`	Branch if < (Unsigned)	`pc <= (reg[rs1] <u reg[rs2]) ? label` `: pc + 4`
BGEU	`bgeu rs1, rs2, label`	Branch if ≥ (Unsigned)	`pc <= (reg[rs1] >=u reg[rs2]) ? label` `: pc + 4`

For JAL and branch instructions (BEQ, BNE, BLT, BGE, BLTU, BGEU), the immediate encodes the target address as an offset from the current pc (i.e., pc + imm = label).

Pseudo Instructions

- Instructions that are not in the ISA but can be easily converted to one or two.

- Example: *(load immediate)*

```
li rd, constant
```

Pseudo Instructions

- Instructions that are not in the ISA but can be easily converted to one or two.

- Example:

```
li rd, constant
→ addi rd, x0, constant
```

Pseudo Instructions

Pseudoinstruction	Description	Execution
`li rd, constant`	Load Immediate	`reg[rd] <= constant`
`mv rd, rs1`	Move	`reg[rd] <= reg[rs1] + 0`
`not rd, rs1`	Logical Not	`reg[rd] <= reg[rs1] ^ -1`
`neg rd, rs1`	Arithmetic Negation	`reg[rd] <= 0 - reg[rs1]`
`j label`	Jump	`pc <= label`
`jal label` `call label`	Jump and Link (with `ra`)	`reg[ra] <= pc + 4` `pc <= label`
`jr rs`	Jump Register	`pc <= reg[rs1] & ~1`
`jalr rs`	Jump and Link Register (with `ra`)	`reg[ra] <= pc + 4` `pc <= reg[rs1] & ~1`
`ret`	Return from Subroutine	`pc <= reg[ra]`
`bgt rs1, rs2, label`	Branch $>$ (Signed)	`pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4`
`ble rs1, rs2, label`	Branch $\leq$ (Signed)	`pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4`
`bgtu rs1, rs2, label`	Branch $>$ (Unsigned)	`pc <= (reg[rs1] >s reg[rs2]) ? label : pc + 4`
`bleu rs1, rs2, label`	Branch $\leq$ (Unsigned)	`pc <= (reg[rs1] <=s reg[rs2]) ? label : pc + 4`
`beqz rs1, label`	Branch $= 0$	`pc <= (reg[rs1] == 0) ? label : pc + 4`
`bnez rs1, label`	Branch $\neq 0$	`pc <= (reg[rs1] != 0) ? label : pc + 4`
`bltz rs1, label`	Branch < 0 (Signed)	`pc <= (reg[rs1] <s 0) ? label : pc + 4`
`bgez rs1, label`	Branch ≥ 0 (Signed)	`pc <= (reg[rs1] >=s 0) ? label : pc + 4`
`bgtz rs1, label`	Branch > 0 (Signed)	`pc <= (reg[rs1] >s 0) ? label : pc + 4`
`blez rs1, label`	Branch ≤ 0 (Signed)	`pc <= (reg[rs1] <=s 0) ? label : pc + 4`

RET and CALL

- How do these instructions work?

Input: x10=5 **and** x11=3

Output: x10=? x11=?

```
Label 1:
 LI   x20, 0
Label 2:
 ADD  x20, x20, x10
 ADDI x11, x11, -1
 BGT  x11, x0, label 2
 MV  x10, x20
 RET
```

```
int foo(int a, int b, int c) {
    int d;
    d=a;
    if (b<d)                    (L1)
        d=b;
    if (c<d)                    (L2)
        d=c;
    return d;                (L3)
```

Calling Convention

- When starting a new mechanism, a set of rules have to be followed to guarantee correctness.

Calling Convention

- Major rules:
  - The callee promises to leave some registers unchanged for the caller. If needs to be modified, the callee saves these on the stack first and recovers in the end.
  - On the call, the return has to be saved on the stack.
  - Before returning, the frame pointer has to be recovered.

Stack and Calling Convention

- Caller
  - Puts arguments on the stack
  - Invokes callee by using call instruction
- Callee
  - Saves reserved registers for caller
  - Saves old based pointer
  - Makes room for local variables and executes the code
  - Puts return value into register
  - Restores stack frame and key registers
  - Returns

Calling Convention

Registers	Symbolic names	Description	Saver
x0	zero	Hardwired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5-x7	t0-t2	Temporary registers	Caller
x8-x9	s0-s1	Saved registers	Callee
x10-x11	a0-a1	Function arguments and return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporary registers	Caller

Data Size

C type	Description	Bytes in RV32
`char`	Character value/byte	1
`short`	Short integer	2
`int`	Integer	4
`long`	Long integer	4
`long long`	Long long integer	8
`void*`	Pointer	4
`float`	Single-precision float	4
`double`	Double-precision float	8
`long double`	Extended-precision float	16

We have our assembly code, now what?

So how does hardware understand these instructions?

- Machine don't understand assembly, so computers use a specific tool called *assembler* to generate machine code (ones and zeroes).

- To generate the machine code, the assembler uses a table.

Learn More

- Checkout these resources:
  - https://6004.mit.edu/web/_static/test/resources/references/6004_isa_reference.pdf
  - https://inst.eecs.berkeley.edu/~cs152/sp21/greencard.pdf
  - https://github.com/jameslzhu/riscv-card/blob/master/riscv-card.pdf

Things we didn't (and won't) cover

- RV64 or RV16
- Other extensions of RV32 (e.g., M, A)
- Floating point instructions and registers
- Privilege and system instructions
- Instructions beyond what were shown in this lecture.

Things we might cover

- CISC and microcode
- VLIW
- …

Participation

- (first time) Create an account here:
  - Link: https://gavel-for-nader.web.app/login
  - or Scan the QR code.

- Use this password: [redacted]

Summary

End of Presentation

Acknowledgement

- This course is partly inspired by the following courses created by my colleagues:
  - CS152, Krste Asanovic (UCB)
  - 18-447, James C. Hoe (CMU)
  - CSE141, Steven Swanson (UCSD)
  - CIS 501, Joe Devietti (Upenn)
  - CS4290, Tom Conte (Goergia Tech)
  - 252-0028-00L, Onur Mutlu (ETH)