

## Course Description

## Lecture 4: ISAs: The RISC-V ISA

This course covers the design and implementation of computer systems from the perspective of the hardware software interface. The purpose of this course is for students to understand the relationship between the operating system, software, and computer architecture. Students that complete the course will have learned operating system fundamentals, computer architecture fundamentals, compilation to hardware abstractions, and how software actually executes from the perspective of the hardware software/boundary. The course will focus especially on understanding the relationships between software and hardware, and how those relationships influence the design of a computer system's software and hardware. The course will convey these topics through a series of practical, implementation-oriented lab assignments.

**Credit: Brandon Lucia**

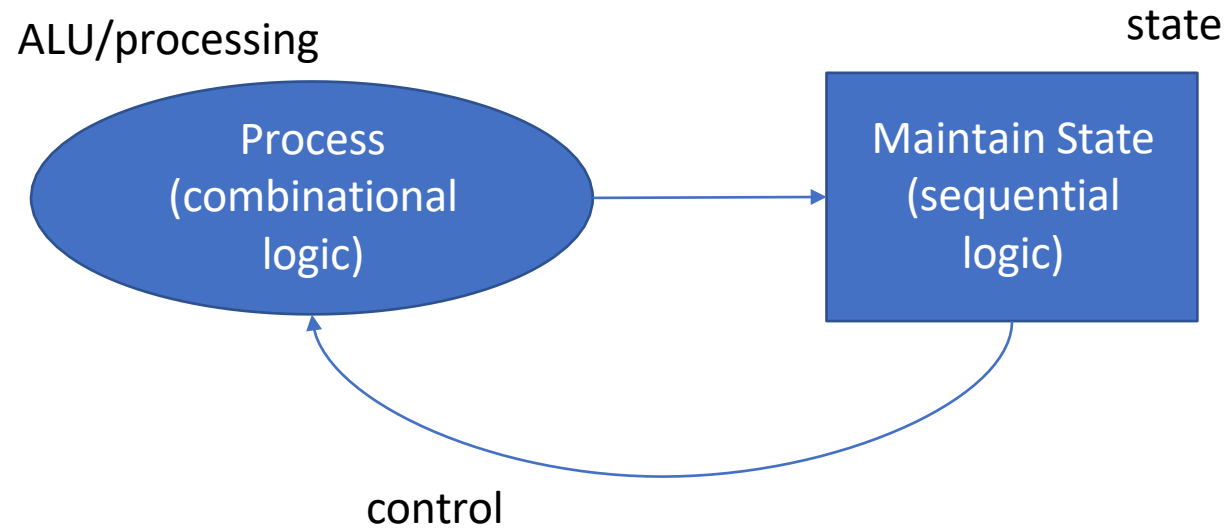
# What is a Computer Architecture?

- Building up to our first architecture
- Defining the ISA: Architecture vs. Microarchitecture
- RISC vs. CISC ISAs
- RISC-V ISA

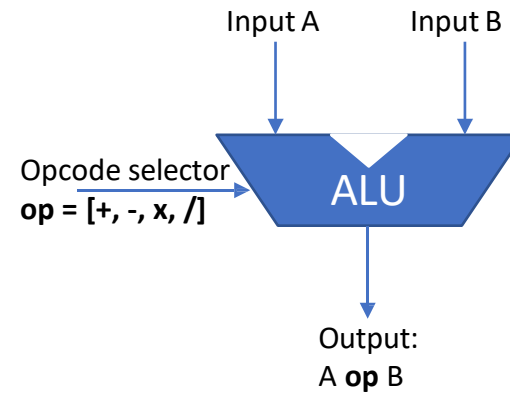
# Recap: What is a Computer Architecture?

- Building up to our first architecture
- Defining the ISA: Architecture vs. Microarchitecture
- RISC vs. CISC ISAs
- RISC-V ISA

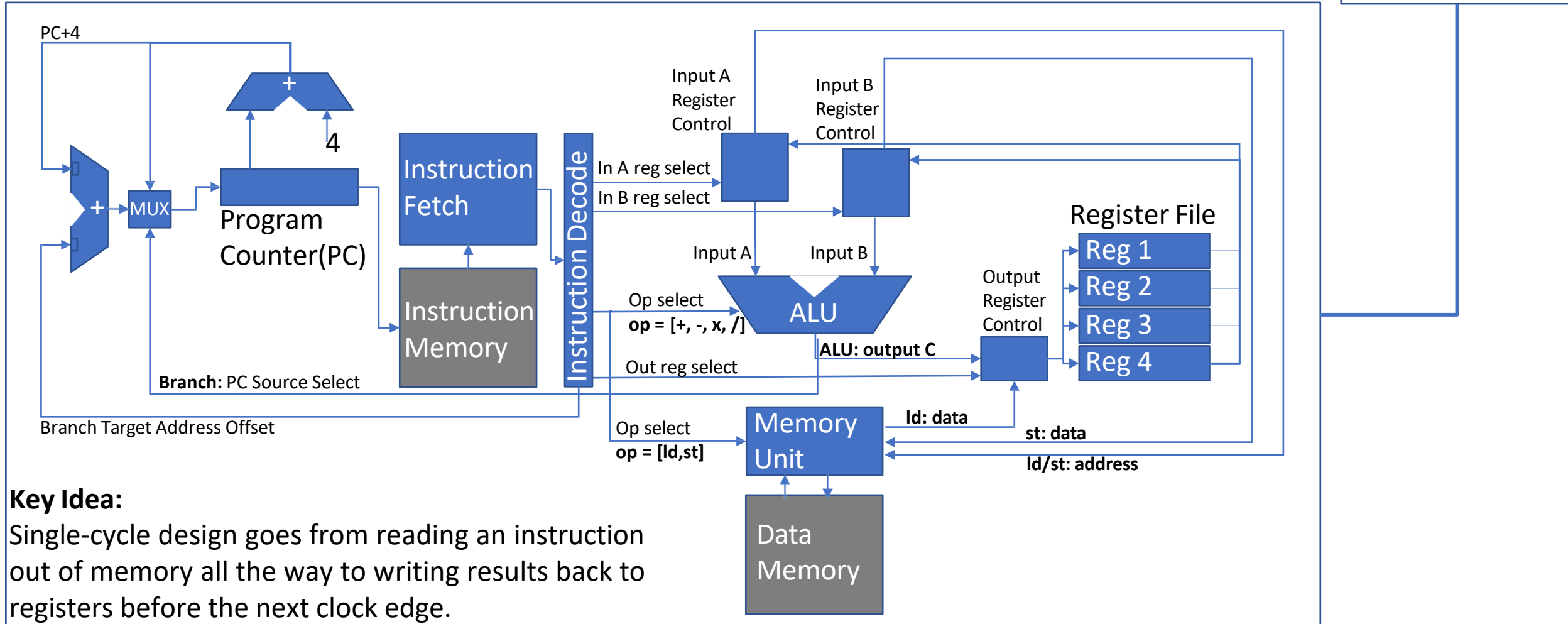
# Basic Architecture: State + processing elements



# Building up to our first architecture: ALU



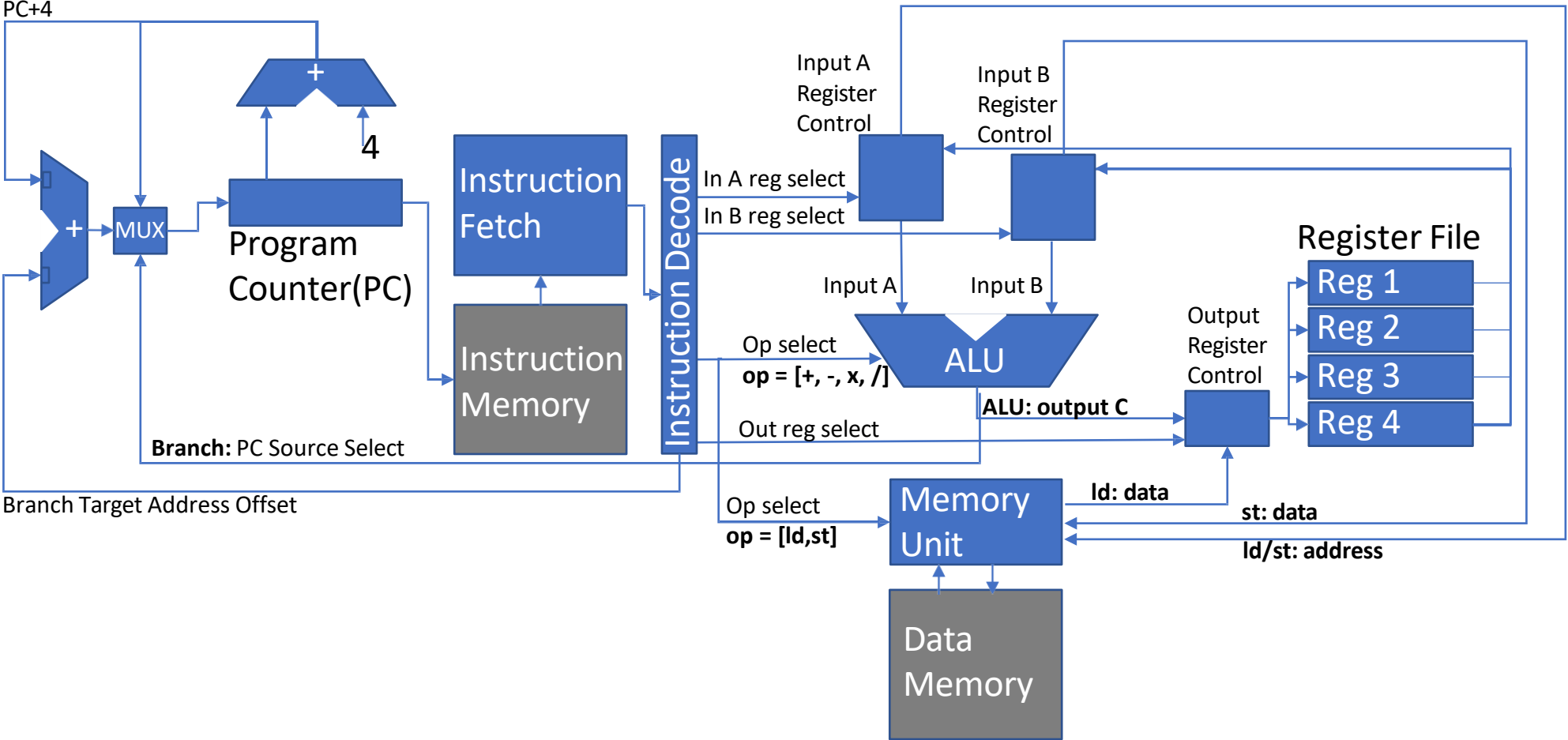
# A “single-cycle” design



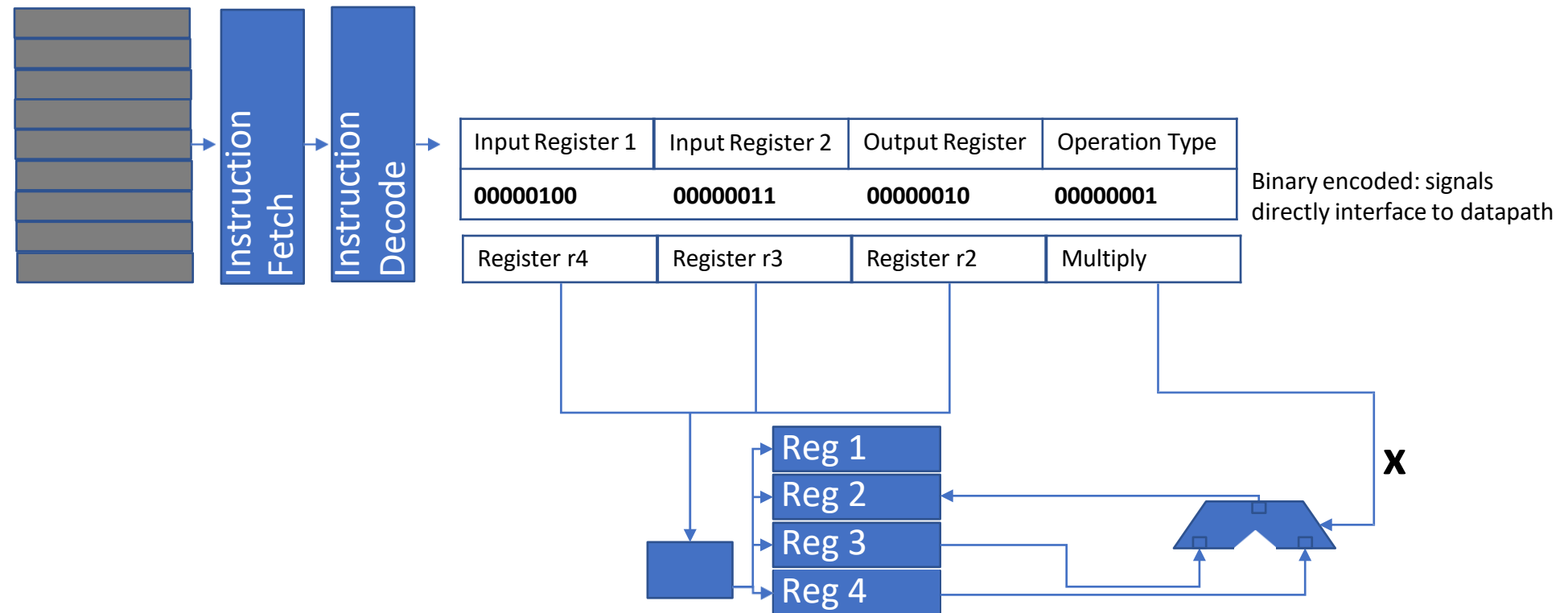
## Key Idea:

Single-cycle design goes from reading an instruction out of memory all the way to writing results back to registers before the next clock edge.

# Where is the HW/SW Interface?



# Big Idea: Instruction Bits are Control Signals

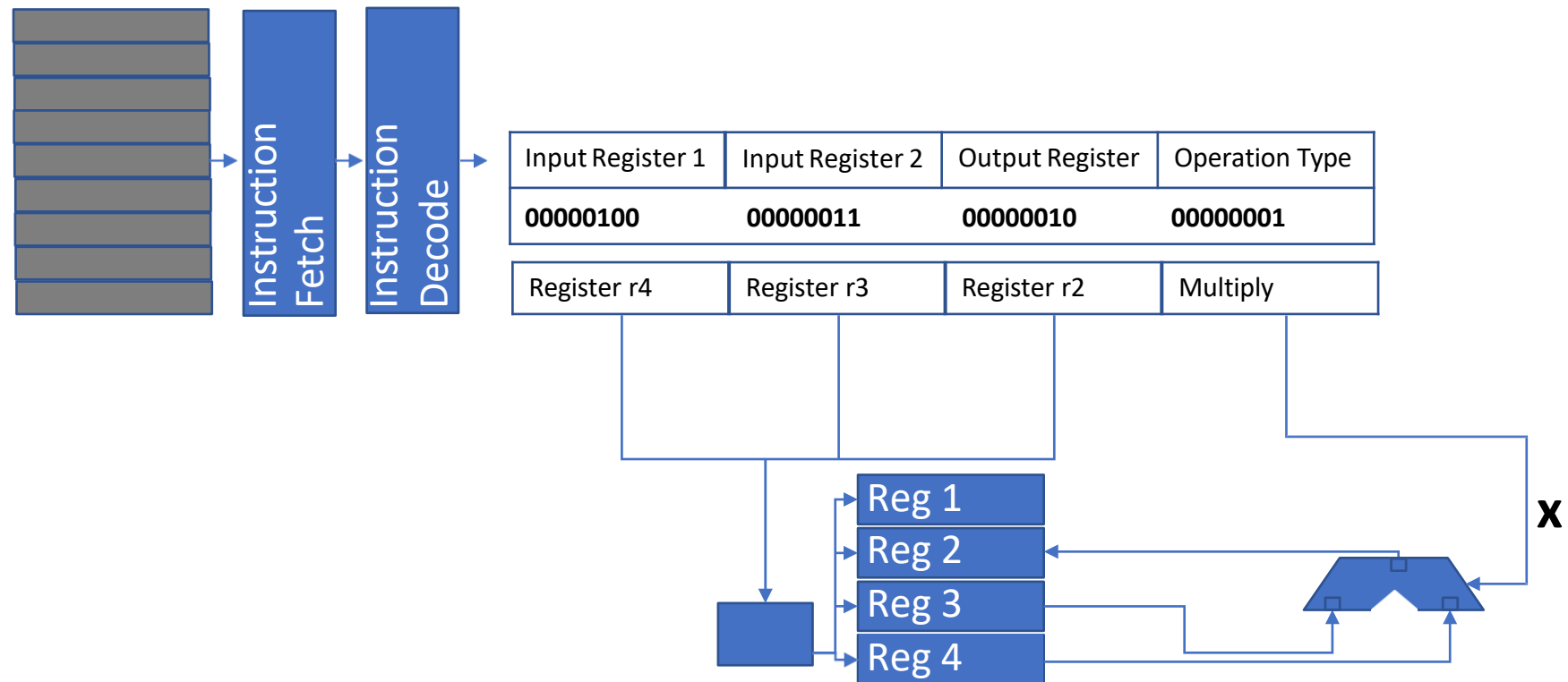




# Architecture vs. Microarchitecture

The ISA defines the **architecture** of the machine

A **microarchitecture** implements the features of the architecture



# Architecture vs. Microarchitecture

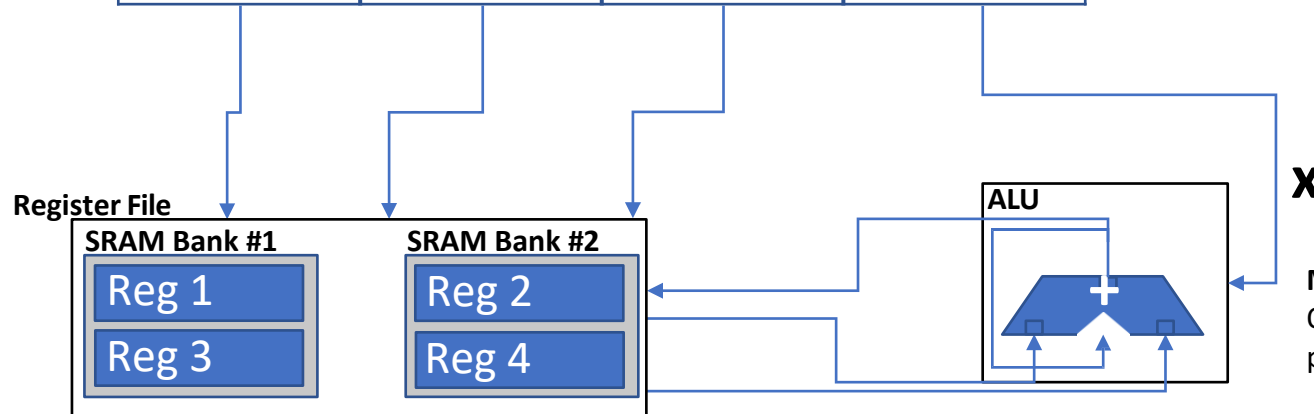
For a given architecture there are **many** perfectly good microarchitectural implementations

## Architecture:

Sequentially-numbered, general-purpose registers

Input Register 1	Input Register 2	Output Register	Operation Type
00000100	00000011	00000010	00000001

Register r4	Register r3	Register r2	Multiply
-------------	-------------	-------------	----------



## Microarchitecture:

Two SRAM banks storing regs based on parity

## Architecture:

Register-register ALU ops, registers numbering 0-4

## Microarchitecture:

One ALU containing an adder; multiply w/ iterated addition, physical register file with registers numbering 0-4

# ISA Design and Diving into RISC-V-RV32I

- What makes an ISA?
- The basics of the RISC-V-RV32I ISA
  - A modern ISA engineered with clear goals from first principles
- More microarchitectural concepts
  - Control hazards & branch prediction
  - Pipelining our microarchitecture & instruction-level parallelism

# What should go in the ISA?

## Reduced Instruction Set Computer

**Simple primitives:**

Let software compose complex operations

**Register operands:**

Decouple functionality from memory accesses

**Few total operations:**

Usually only one way to do something

## Complex Instruction Set Computer

**Simple & complex operations:**

Hardware provides complex functionality

**Many operations:**

Often several ways to do the same thing

**Register and memory operands:**

Operations may directly manipulate memory



# What should go in the ISA?

## Reduced Instruction Set Computer

### Simple primitives:

Let software compose complex operations

### Register operands:

Decouple functionality from memory accesses

### Few total operations:

Usually only one way to do something

```
rd = M[imm]
rd = M[reg]
rd = M[reg + imm]
rd = M[PC + imm]
```

Few cases to map to control signals in microarchitecture

## Complex Instruction Set Computer

### Simple & complex operations:

Hardware must support complex functionality

### Many operations:

Often several ways to do the same thing

### Register and memory operands:

Operations may directly manipulate memory

Many cases to map to control signals in microarchitecture

	Source	Dest	Src, Dest	C Analog
movq	Imm	Reg	movq \$0x4, %rax	temp = 0x4;
		Mem	movq \$-147, (%rax)	*p = -147;
	Reg	Reg	movq %rax, %rdx	temp2 = temp1;
		Mem	movq %rax, (%rdx)	*p = temp;
	Mem	Reg	movq (%rax), %rdx	temp = *p;

Remember this from 18-213?

Plus all of these combinations  
**D(Rb,Ri,S)                      Mem[Reg[Rb]+S\*Reg[Ri]+ D]**

# What should go in the ISA?

## Reduced Instruction Set Computer

### **Simple primitives:**

Let software compose complex operations

### **Register operands:**

Decouple functionality from memory accesses

### **Few total operations:**

Usually only one way to do something

## Complex Instruction Set Computer

### **Simple & complex operations:**

Hardware must support complex functionality

### **Register and memory operands:**

Operations may directly manipulate memory

### **Many operations:**

Often several ways to do the same thing

## **What are the pros and cons of each?**

**How does RISC vs. CISC affect the microarchitecture,  
compiler, program, programmer?**

# Principles of ISA Design

## General Principles

Regularity – “Law of least astonishment”

Orthogonality – keep separable concerns separate

Composability – regular, orthogonal ops combine easily

## Specific Principles

One vs. All – precisely one way to do it, or all ways should be possible

Primitives, not solutions – solve by coding, compiling, & synthesizing

## “Blatant opinions” (matters of taste)

Addressing – not limited to simple arrays, etc.

Environment Support – exceptions, processes, debugging, etc

Deviations – deviate from these rules only in implementation-specific ways

---

**Designing irregular structures at the chip level is very expensive.**

---

---

**Some architectures have provided direct implementations of high-level concepts. In many cases these turn out to be more trouble than they are worth.**

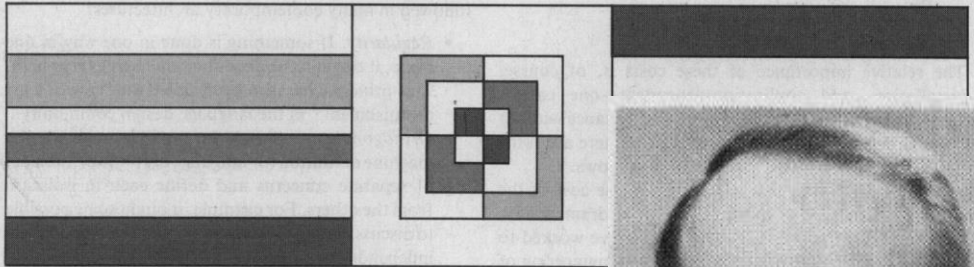
---

*An examination of the relation between architecture and compiler design leads to several principles which can simplify compilers and improve the object code they produce.*

---

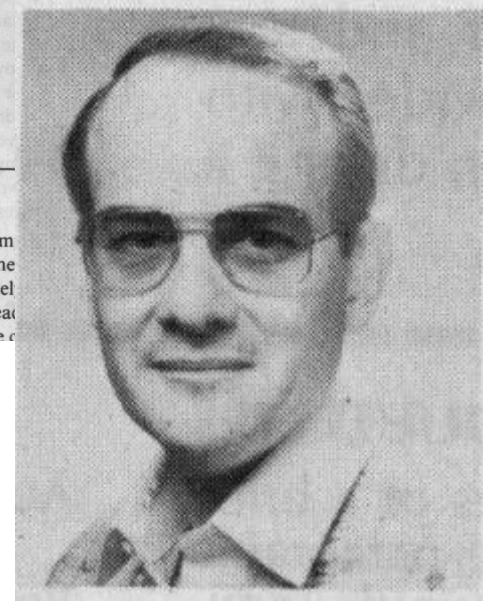
## Compilers and Computer Architecture

William A. Wulf  
Carnegie-Mellon University



The interactions between the design of a computer's instruction set and the design of compilers that generate code for that computer have serious implications for overall computational cost and efficiency. This article, which investigates those interactions, should ideally be

simplify com  
programs the  
are absolutel  
ever, they lea  
people have c



# RISCV ISA

- We will learn about ISA design by learning about RISCV
- Modern, full-featured RISC ISA
- Developed in the last decade at UC Berkeley
  - The fifth in a sequence of RISC ISAs originating in the 80s
  - <https://riscv.org/technical/specifications/>
  - The RISC-V Instruction Set Manual, Volume I: BaseUser-Level ISA, Waterman et al, 2011
- Goals
  - Open-source
  - Free
  - Simple, but full-featured; avoids “over-architecting” for a particular uArch style (FPGA, ASIC,...)
  - Extensible through extension specifications and variants
  - Support heterogeneous & parallel systems efficiently
  - Support 32- and 64-bit variants efficiently
  - Fully virtualizable
  - Supports (but does not require) IEEE 754 Floating Point





# RISCV Variants & extensions

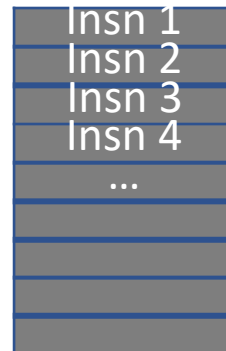
- |  |                  |
|--|------------------|
| <ul style="list-style-type: none"> <li>• RV32I &amp; RV64I are base integer ISA versions</li> <li>• M – Support for HW Multiply &amp; Divide</li> </ul>                                      | Basic Operations |
| <ul style="list-style-type: none"> <li>• F – Support for single-precision Float</li> <li>• D – Support for double-precision Float</li> <li>• Q – Support for quad-precision Float</li> </ul> | Floating Point   |
| <ul style="list-style-type: none"> <li>• A – Support for Atomic instructions</li> <li>• RVWMO – Memory Consistency Model</li> </ul>  | Concurrency      |
| <ul style="list-style-type: none"> <li>• L,B,J,T,P,V – Extra weird stuff (read the spec)</li> </ul>  |                  |

Base	Version	Status
RVWMO	2.0	<b>Ratified</b>
<b>RV32I</b>	<b>2.1</b>	<b>Ratified</b>
<b>RV64I</b>	<b>2.1</b>	<b>Ratified</b>
<i>RV32E</i>	<i>1.9</i>	<i>Draft</i>
<i>RV128I</i>	<i>1.7</i>	<i>Draft</i>
Extension	Version	Status
<b>M</b>	<b>2.0</b>	<b>Ratified</b>
<b>A</b>	<b>2.1</b>	<b>Ratified</b>
<b>F</b>	<b>2.2</b>	<b>Ratified</b>
<b>D</b>	<b>2.2</b>	<b>Ratified</b>
<b>Q</b>	<b>2.2</b>	<b>Ratified</b>
<b>C</b>	<b>2.0</b>	<b>Ratified</b>
<i>Counters</i>	<i>2.0</i>	<i>Draft</i>
<i>L</i>	<i>0.0</i>	<i>Draft</i>
<i>B</i>	<i>0.0</i>	<i>Draft</i>
<i>J</i>	<i>0.0</i>	<i>Draft</i>
<i>T</i>	<i>0.0</i>	<i>Draft</i>
<i>P</i>	<i>0.2</i>	<i>Draft</i>
<i>V</i>	<i>0.7</i>	<i>Draft</i>
<b>Zicsr</b>	<b>2.0</b>	<b>Ratified</b>
<b>Zifencei</b>	<b>2.0</b>	<b>Ratified</b>
<i>Zam</i>	<i>0.1</i>	<i>Draft</i>
<i>Ztso</i>	<i>0.1</i>	<i>Frozen</i>

# RISCV Variants & extensions

- RV32I & RV64I are base integer ISA versions
- XLEN: how many bits in a register?
- Memory:  $2^{XLEN}$  bytes
- Word: 4B, Doubleword: 8B, Halfword: 2B

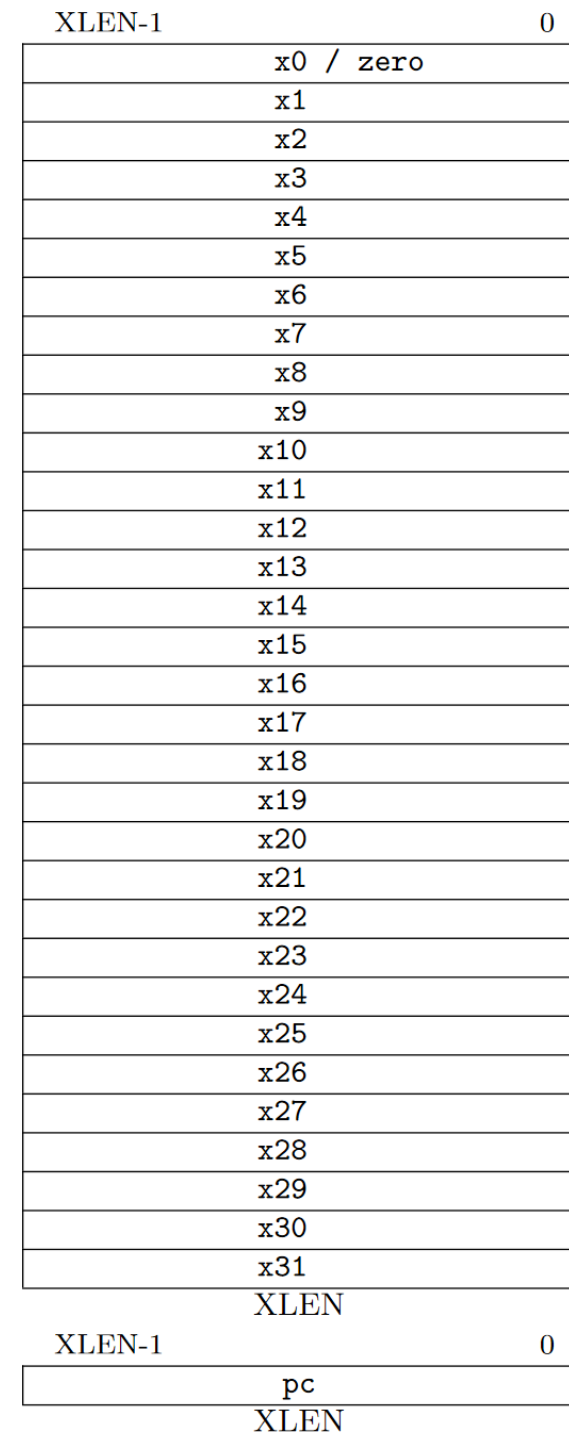
32-bit instructions in base encoding



Base	Version	Status
RVWMO	2.0	<b>Ratified</b>
<b>RV32I</b>	<b>2.1</b>	<b>Ratified</b>
<b>RV64I</b>	<b>2.1</b>	<b>Ratified</b>
<i>RV32E</i>	<i>1.9</i>	<i>Draft</i>
<i>RV128I</i>	<i>1.7</i>	<i>Draft</i>
Extension	Version	Status
<b>M</b>	<b>2.0</b>	<b>Ratified</b>
<b>A</b>	<b>2.1</b>	<b>Ratified</b>
<b>F</b>	<b>2.2</b>	<b>Ratified</b>
<b>D</b>	<b>2.2</b>	<b>Ratified</b>
<b>Q</b>	<b>2.2</b>	<b>Ratified</b>
<b>C</b>	<b>2.0</b>	<b>Ratified</b>
<i>Counters</i>	<i>2.0</i>	<i>Draft</i>
<i>L</i>	<i>0.0</i>	<i>Draft</i>
<i>B</i>	<i>0.0</i>	<i>Draft</i>
<i>J</i>	<i>0.0</i>	<i>Draft</i>
<i>T</i>	<i>0.0</i>	<i>Draft</i>
<i>P</i>	<i>0.2</i>	<i>Draft</i>
<i>V</i>	<i>0.7</i>	<i>Draft</i>
<b>Zicsr</b>	<b>2.0</b>	<b>Ratified</b>
<b>Zifencei</b>	<b>2.0</b>	<b>Ratified</b>
<i>Zam</i>	<i>0.1</i>	<i>Draft</i>
<i>Ztso</i>	<i>0.1</i>	<i>Frozen</i>

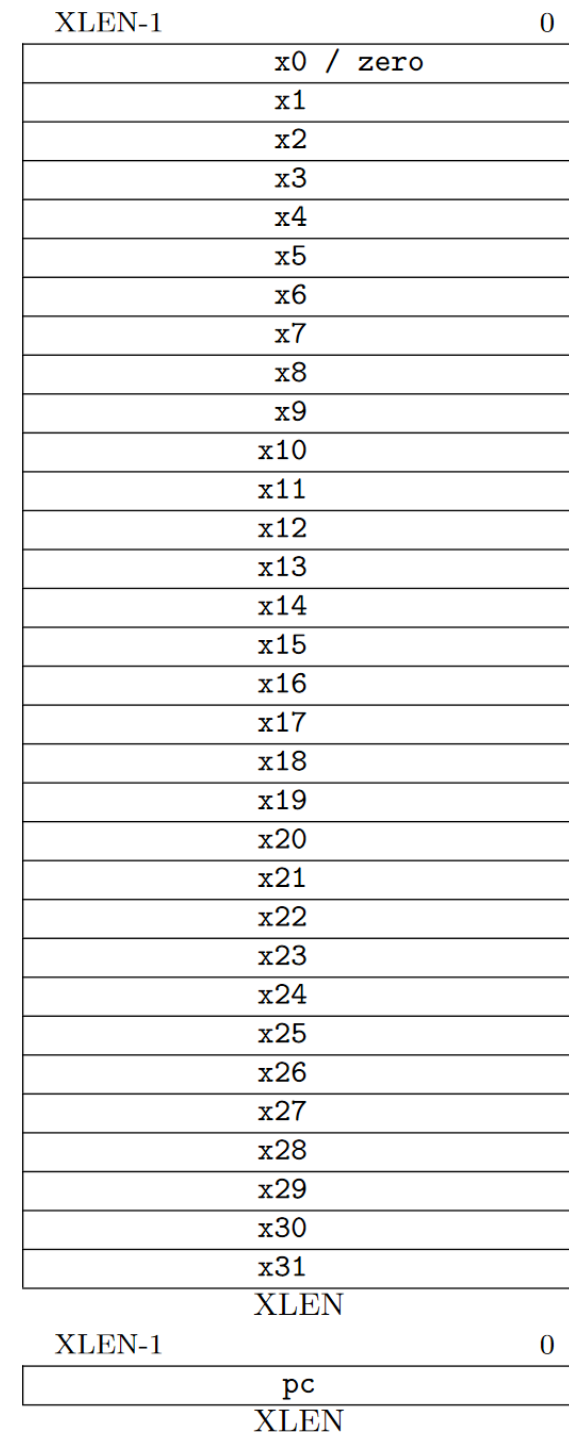
# RISCV-RV32I Specification

- 32 Registers x0-x31 + PC register
- x0 is always zero
- x1 is the return address (by convention)
- x2 is the stack pointer (by convention)
- x5 is used as an “alternate link” register (by convention)
  - E.g., Implementing exceptions / long jumps in software
- **(Micro)architectural implications of this ISA choice?**



# RISCV-RV32I Specification

- 32 Registers x0-x31 + PC register
- x0 is always zero
- x1 is the return address (by convention)
- x2 is the stack pointer (by convention)
- x5 is used as an “alternate link” register (by convention)
  - E.g., Implementing exceptions / long jumps in software
- **(Micro)architectural implications of this ISA choice?**
  - **Why not 16 registers? [RV32-E has 16 regs; why?]**
  - **Why not 16-bit instructions?**
  - **Power / Energy?**
  - **Compilation & optimization?**



# Exercise: What about variable insn/reg size?

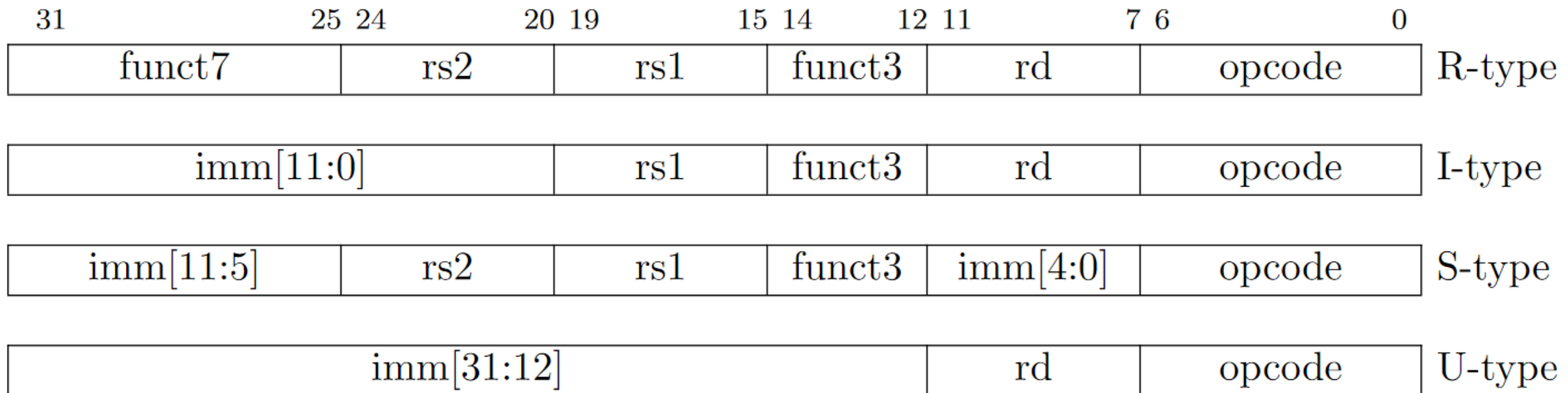
- What defines instruction size? Why? What defines register size? Why?
- Can we support multiple instruction sizes? Why would we support multiple sizes?
- How to support different sizes? Benefits & drawbacks?

# Exercise: What about variable insn/reg size?

- What defines instruction size? Why? What defines register size? Why?
  - ISA defines insn size. Lacking extensions, RV32I & RV64I both 32-bit insn
  - ISA **variant** defines reg size. Programmer must know, datapath must implement; must be ISA-level spec.
- Can we support multiple instruction sizes? Why would we support multiple sizes?
  - Can we? Yes. Why? Code size optimization, longer constant immediates, longer jumps
- How to support different sizes? Benefits & drawbacks?
  - Two options. Option 1: RVC (riscv-compressed) – 16-bit ops blowup at decode into 32bit ones. Code size optimization. Longer jumps possible.
  - Option 2: steal some ISA bits to indicate variable width: 11 for 32, 011111 for 48, 0111111 for 64
  - Costs? Increased decode complexity. Need to figure out how big instruction word is and where important signals are in the instruction word based on size.

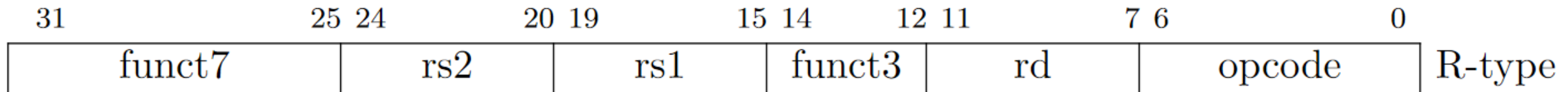
# RISCV-RV32I Specification

- Four base instruction encoding formats
  - R(egister), I(mmediate), S(tore), U(pper Immediate)
  - Mnemonics are non-binding and formats get flexibly used



# RISCV-RV32I Specification

- Four base instruction encoding formats
  - R(egister), I(mmediate), S(tore), U(pper Immediate)
  - Mnemonics are non-binding and formats get flexibly used



R-Type: 2 register input operands, 1 register output operand, opcode type, and function selection bits



# RISCV-RV32I Specification

- Four base instruction encoding formats
  - R(egister), I(mmediate), S(tore), U(pper Immediate)
  - Mnemonics are non-binding and formats get flexibly used

31                      25 24                      20 19                      15 14                      12 11                      7 6                      0

imm[11:0]	rs1	funct3	rd	opcode	I-type
-----------	-----	--------	----	--------	--------

I-Type: 1 register input operands, 1 immediate input operand, 1 register output operand, opcode type, and function selection bits

# RISCV-RV32I Specification

- Four base instruction encoding formats
  - R(egister), I(mmediate), S(tore), U(pper Immediate)
  - Mnemonics are non-binding and formats get flexibly used

31                      25 24                      20 19                      15 14                      12 11                      7 6                      0

S-Type: 2 register input operands, 1 immediate input operand, **[0 output operands]**,  
Opcode type, and function selection bits

imm[11:5]

rs2

rs1

funct3

imm[4:0]

opcode

S-type

# RISCV-RV32I Specification

- Four base instruction encoding formats
  - R(egister), I(mmediate), S(tore), U(pper immediate)
  - Mnemonics are non-binding and formats get flexibly used

31                      25 24                      20 19                      15 14                      12 11                      7 6                      0

U-Type: 1 immediate input operand, 1 register output operand, opcode selection bits

imm[31:12]

rd

opcode

U-type

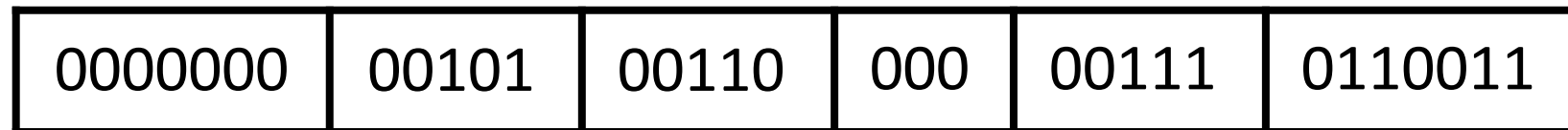
# Example: R-type Arithmetic Operations

## RV32I encoding

<https://metalcode.eu/2019-12-06-rv32i.html>

[31:25] 7	[24:20] 5	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
function 7	source 2	source 1	function 3	destination	opcode
0000000	xR	xL	000 : ADD	xD	0110011 : OP
0100000	xR	xL	000 : SUB	xD	0110011 : OP
0000000	xR	xL	001 : SLL	xD	0110011 : OP
0000000	xR	xL	010 : SLT	xD	0110011 : OP
0000000	xR	xL	011 : SLTU	xD	0110011 : OP
0000000	xR	xL	100 : XOR	xD	0110011 : OP
0000000	xR	xL	101 : SRL	xD	0110011 : OP
0100000	xR	xL	101 : SRA	xD	0110011 : OP
0000000	xR	xL	110 : OR	xD	0110011 : OP
0000000	xR	xL	111 : AND	xD	0110011 : OP

OP



# Example: R-type Arithmetic Operations

## RV32I encoding

<https://metalcode.eu/2019-12-06-rv32i.html>

[31:25] 7	[24:20] 5	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
function 7	source 2	source 1	function 3	destination	opcode
0000000	xR	xL	000 : ADD	xD	0110011 : OP
0100000	xR	xL	000 : SUB	xD	0110011 : OP
0000000	xR	xL	001 : SLL	xD	0110011 : OP
0000000	xR	xL	010 : SLT	xD	0110011 : OP
0000000	xR	xL	011 : SLTU	xD	0110011 : OP
0000000	xR	xL	100 : XOR	xD	0110011 : OP
0000000	xR	xL	101 : SRL	xD	0110011 : OP
0100000	xR	xL	101 : SRA	xD	0110011 : OP
0000000	xR	xL	110 : OR	xD	0110011 : OP
0000000	xR	xL	111 : AND	xD	0110011 : OP

Func 7 = 0    reg x5    reg x6    ADD    reg x7    OP

$$x7 = x5 + x6$$

0000000	00101	00110	000	00111	0110011
---------	-------	-------	-----	-------	---------

# Example: R-type Arithmetic Operations

## RV32I encoding

<https://metalcode.eu/2019-12-06-rv32i.html>

[31:25] 7	[24:20] 5	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
function 7	source 2	source 1	function 3	destination	opcode
0000000	xR	xL	000 : ADD	xD	0110011 : OP
0100000	xR	xL	000 : SUB	xD	0110011 : OP
0000000	xR	xL	001 : SLL	xD	0110011 : OP
0000000	xR	xL	010 : SLT	xD	0110011 : OP
0000000	xR	xL	011 : SLTU	xD	0110011 : OP
0000000	xR	xL	100 : XOR	xD	0110011 : OP
0000000	xR	xL	101 : SRL	xD	0110011 : OP
0100000	xR	xL	101 : SRA	xD	0110011 : OP
0000000	xR	xL	110 : OR	xD	0110011 : OP
0000000	xR	xL	111 : AND	xD	0110011 : OP

Func 7 = 32    reg x5    reg x6    SUB    reg x7    OP

$$x7 = x5 - x6$$

0100000	00101	00110	000	00111	0110011
---------	-------	-------	-----	-------	---------

# Example: R-type Arithmetic Operations

## RV32I encoding

<https://metalcode.eu/2019-12-06-rv32i.html>

[31:25] 7	[24:20] 5	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
function 7	source 2	source 1	function 3	destination	opcode
0000000	xR	xL	000 : ADD	xD	0110011 : OP
0100000	xR	xL	000 : SUB	xD	0110011 : OP
0000000	xR	xL	001 : SLL	xD	0110011 : OP
0000000	xR	xL	010 : SLT	xD	0110011 : OP
0000000	xR	xL	011 : SLTU	xD	0110011 : OP
0000000	xR	xL	100 : XOR	xD	0110011 : OP
0000000	xR	xL	101 : SRL	xD	0110011 : OP
0100000	xR	xL	101 : SRA	xD	0110011 : OP
0000000	xR	xL	110 : OR	xD	0110011 : OP
0000000	xR	xL	111 : AND	xD	0110011 : OP

Func 7 = 0    reg x5    reg x6    OR    reg x7    OP

$x7 = x5 \mid x6$

0000000	00101	00110	110	00111	0110011
---------	-------	-------	-----	-------	---------

# Example: I-type Reg/Imm Arithmetic Operations

[31:20] 12	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
IMMEDIATE[11:0]	source 1	function 3	destination	opcode
CONSTANT[11:0]	xL	000 : ADDI	xD	0010011 : OP-IMM
CONSTANT[11:0]	xL	010 : SLTI	xD	0010011 : OP-IMM
CONSTANT[11:0]	xL	011 : SLTIU	xD	0010011 : OP-IMM
CONSTANT[11:0]	xL	100 : XORI	xD	0010011 : OP-IMM
CONSTANT[11:0]	xL	110 : ORI	xD	0010011 : OP-IMM
CONSTANT[11:0]	xL	111 : ANDI	xD	0010011 : OP-IMM

$x7 = x5 \ll 9$

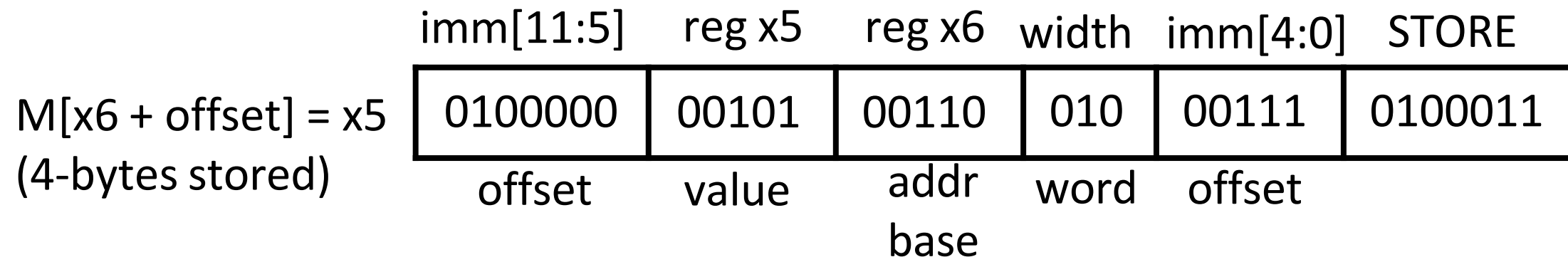
Func 7 = 0	reg x6	SLTI	reg x7	OP-IMM
0000 0000 1001	00110	010	00111	0100011



# Example: S-type Store Operations

**Base + offset: Why? Seem familiar?**

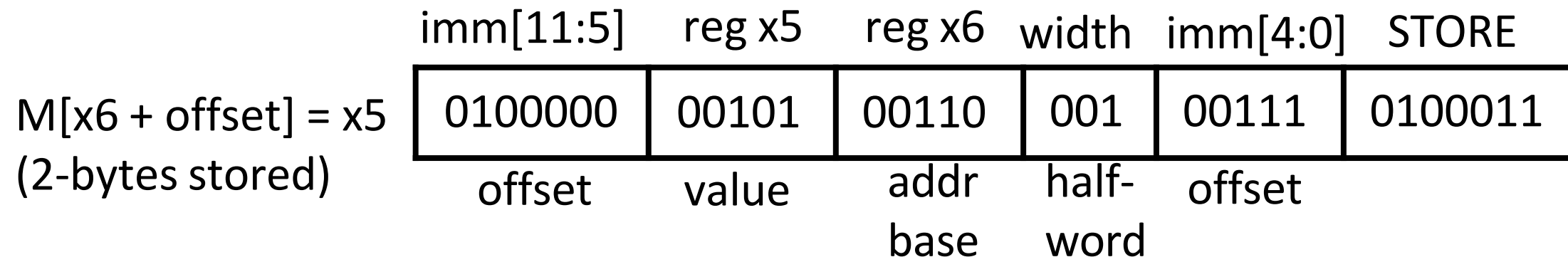
[31:25] 7	[24:20] 5	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
IMMEDIATE[11:5]	source 2	source 1	function 3	IMMEDIATE[4:0]	opcode
OFFSET[11:5]	x5	xB	000 : SB	OFFSET	0100011 : STORE
OFFSET[11:5]	x5	xB	001 : SH	OFFSET	0100011 : STORE
OFFSET[11:5]	x5	xB	010 : SW	OFFSET	0100011 : STORE
OFFSET[11:5]	x5	xB	100 : SBU	OFFSET	0100011 : STORE
OFFSET[11:5]	x5	xB	101 : SHU	OFFSET	0100011 : STORE



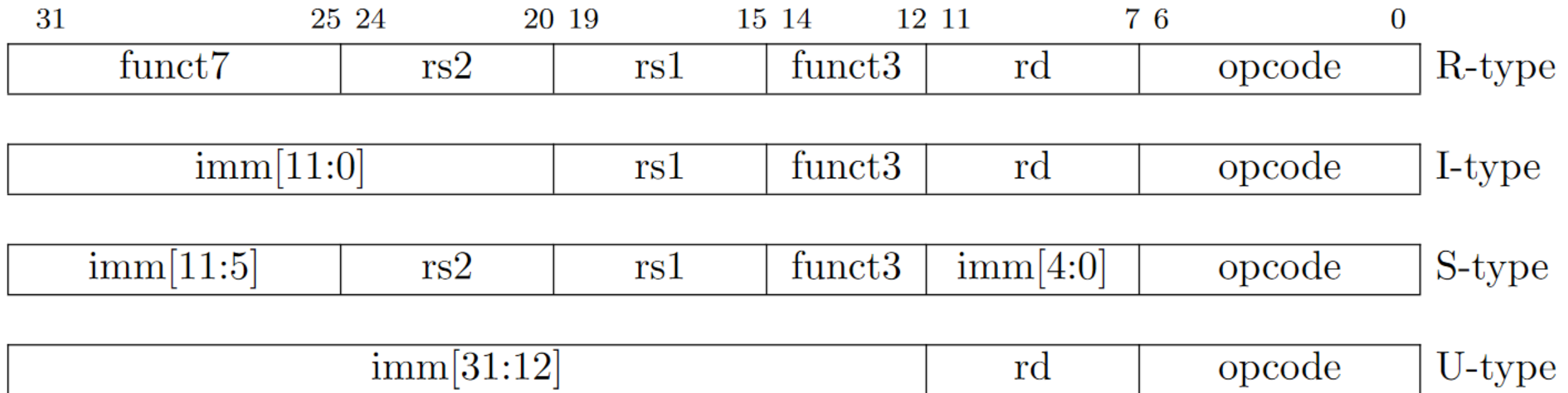
# Example: S-type Store Operations

**Base + offset: Why? Seem familiar?**

[31:25] 7	[24:20] 5	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
IMMEDIATE[11:5]	source 2	source 1	function 3	IMMEDIATE[4:0]	opcode
OFFSET[11:5]	x5	xB	000 : SB	OFFSET	0100011 : STORE
OFFSET[11:5]	x5	xB	001 : SH	OFFSET	0100011 : STORE
OFFSET[11:5]	x5	xB	010 : SW	OFFSET	0100011 : STORE
OFFSET[11:5]	x5	xB	100 : SBU	OFFSET	0100011 : STORE
OFFSET[11:5]	x5	xB	101 : SHU	OFFSET	0100011 : STORE

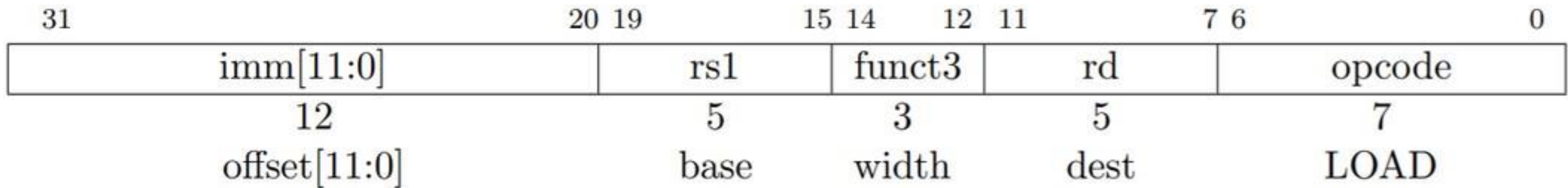


# What type are Load instructions?



What information do we need to encode for a load instruction?

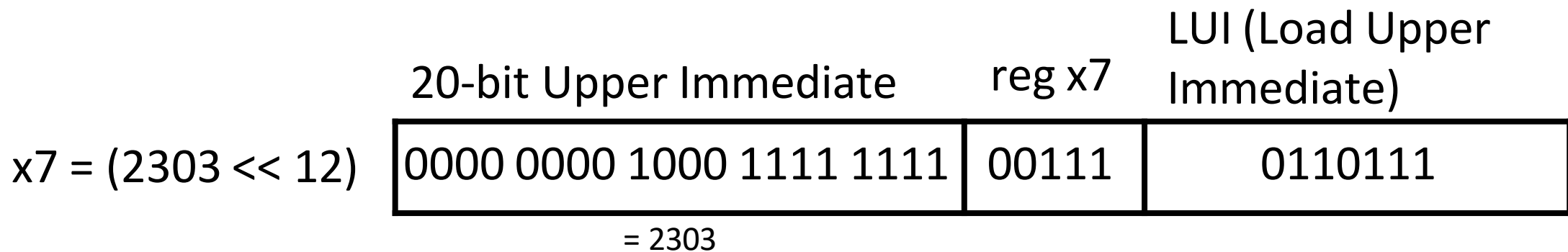
# Load operations are I-Type Instructions



# Example: U-type Upper Immediate Operations

**What in the world is this instruction type used for?**

[31:12] 20	[11:7] 5	[6:0] 7
IMMEDIATE[31:12]	destination	opcode
UPPER[31:12]	xD	0110111 : LUI
UPPER[31:12]	xD	0010111 : AUIPC

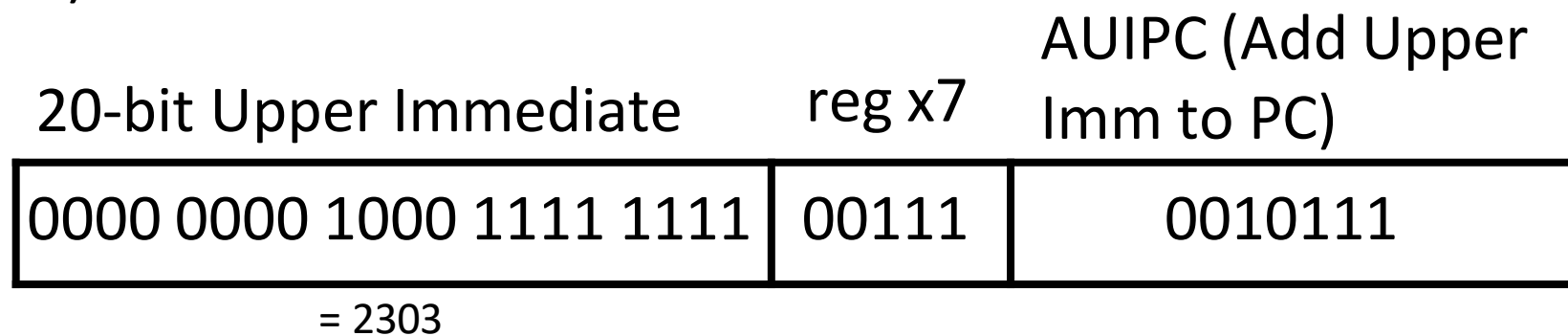


# Example: U-type Upper Immediate Operations

**Why bring the PC into the picture?**

[31:12] 20	[11:7] 5	[6:0] 7
IMMEDIATE[31:12]	destination	opcode
UPPER[31:12]	xD	0110111 : LUI
UPPER[31:12]	xD	0010111 : AUIPC

$$x7 = PC + (2303 \ll 12)$$



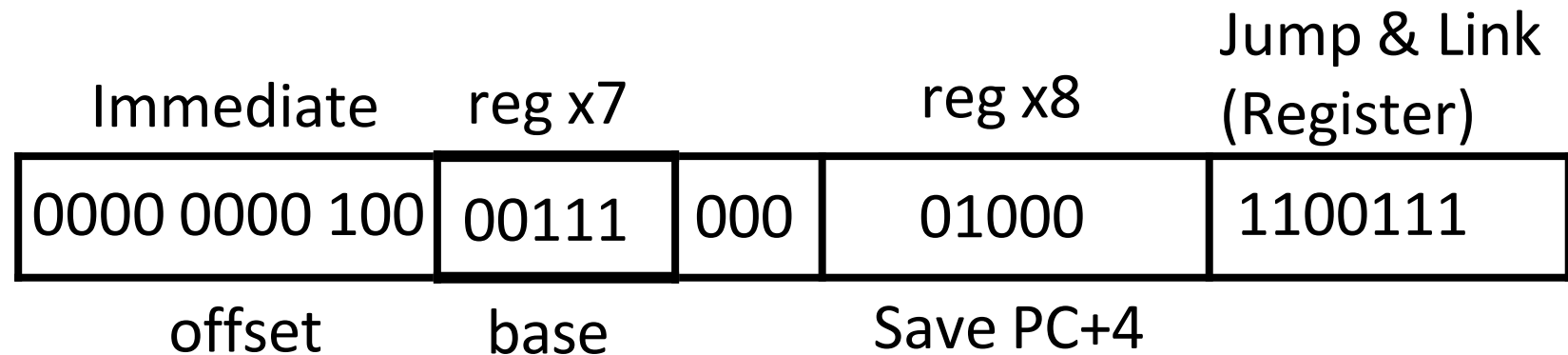
# Control Flow: Jump-and-link

[31]	[30:21]	[20]	[19:12]	[11:7]	[6:0]
1	10	1	8	5	7
I[20]	I[10:1]	I[11]	IMMEDIATE[19:12]	destination	opcode
O[20]	O[10:1]	O[11]	O[19:12]	xD	1101111 : JAL

[31:20]	[19:15]	[14:12]	[11:7]	[6:0]
12	5	3	5	7
IMMEDIATE[11:0]	source 1	function 3	destination	opcode
OFFSET[11:0]	xL	0	xD	1100111 : JALR

x8=PC+4;

jump to  
((base+offset) &  
0xffffffff)



# Control Flow: ~~Call~~ Jump-and-link

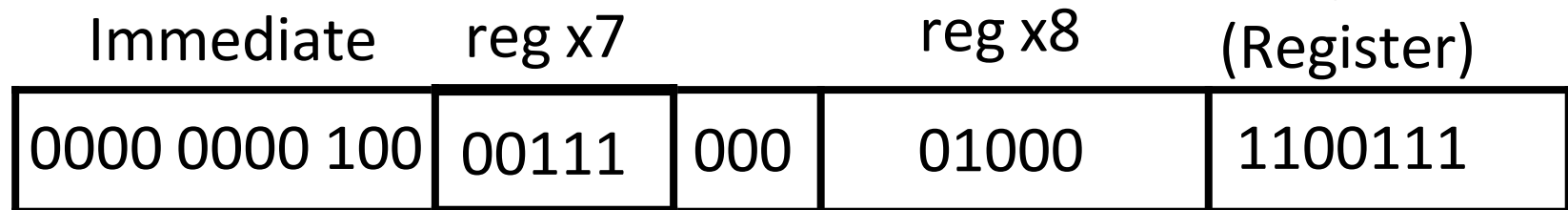
[31]	[30:21]	[20]	[19:12]	[11:7]	[6:0]
1	10	1	8	5	7
I[20]	I[10:1]	I[11]	IMMEDIATE[19:12]	destination	opcode
O[20]	O[10:1]	O[11]	O[19:12]	xD	1101111 : JAL

[31:20]	[19:15]	[14:12]	[11:7]	[6:0]
12	5	3	5	7
IMMEDIATE[11:0]	source 1	function 3	destination	opcode
OFFSET[11:0]	xL	0	xD	1100111 : JALR

x8=PC+4;

jump to  
((base+offset) &  
0xffffffff)

Jump & Link  
(Register)



offset

base

Save PC+4

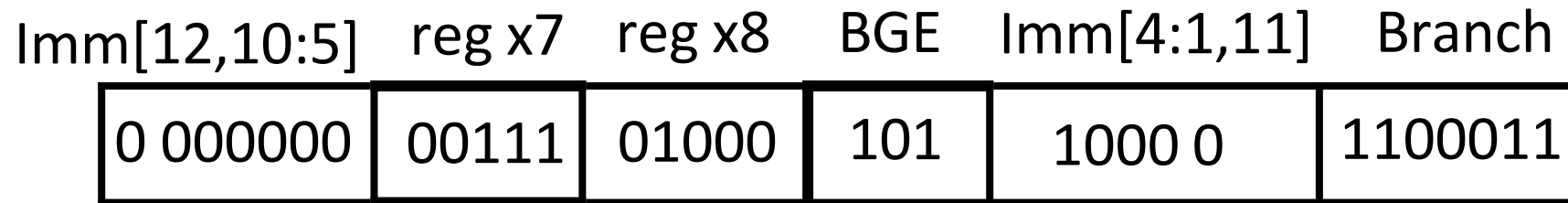
**Why the mask?**



# Control Flow: Compare & Branch

[31] 1	[30:25] 6	[24:20] 5	[19:15] 5	[14:12] 3	[11:8] 4	[7] 1	[6:0] 7
I[12]	I[10:5]	source 2	source 1	function 3	I[4:1]	I[11]	opcode
O[12]	O[10:5]	xR	xL	000 : BEQ	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	001 : BNE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	100 : BLT	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	101 : BGE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	110 : BLTU	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	111 : BGEU	O[4:1]	O[11]	1100011 : BRANCH

If(x8 >= x7)  
PC=PC + 16

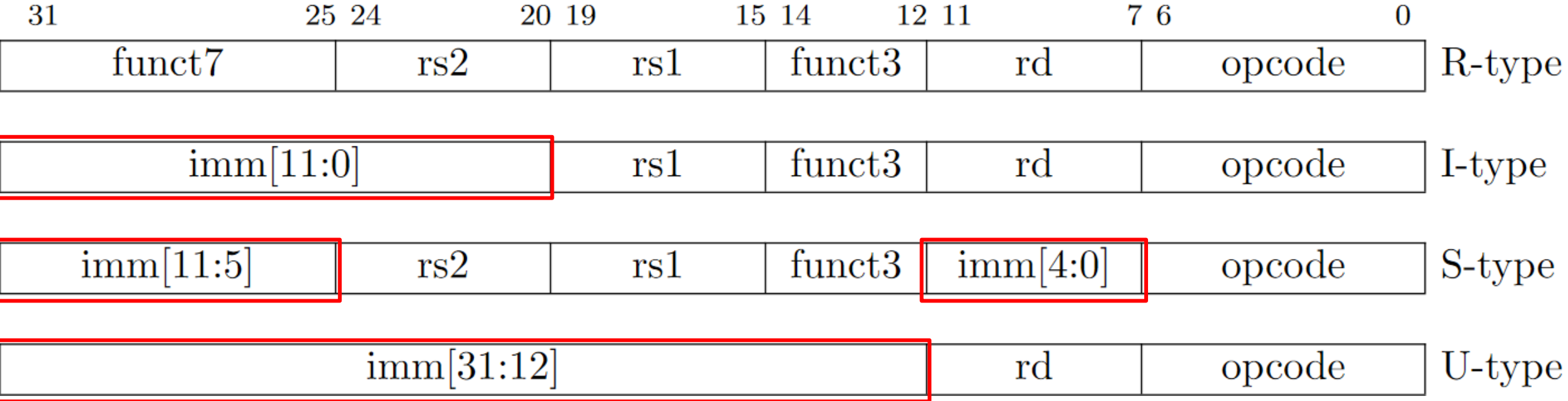


**Imm: PC-relative branch target**

**Where is bit 0?**

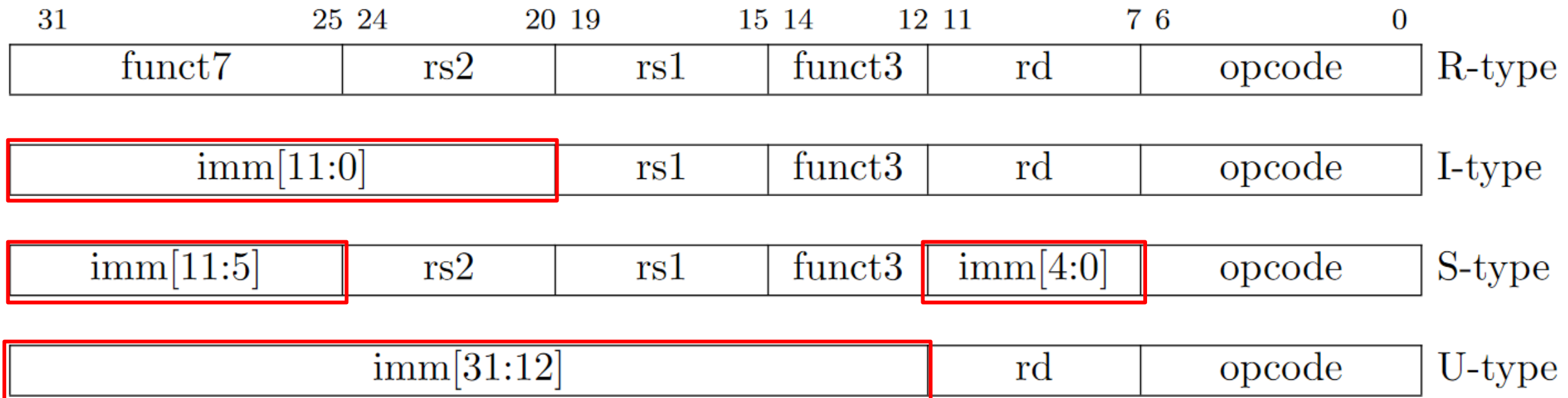
# Design Goals Behind Instruction Encoding

- Why are all of the immediate bits all over the place?
- Why are the immediates apparently different sizes?
- Why are some immediate bits left unspecified?



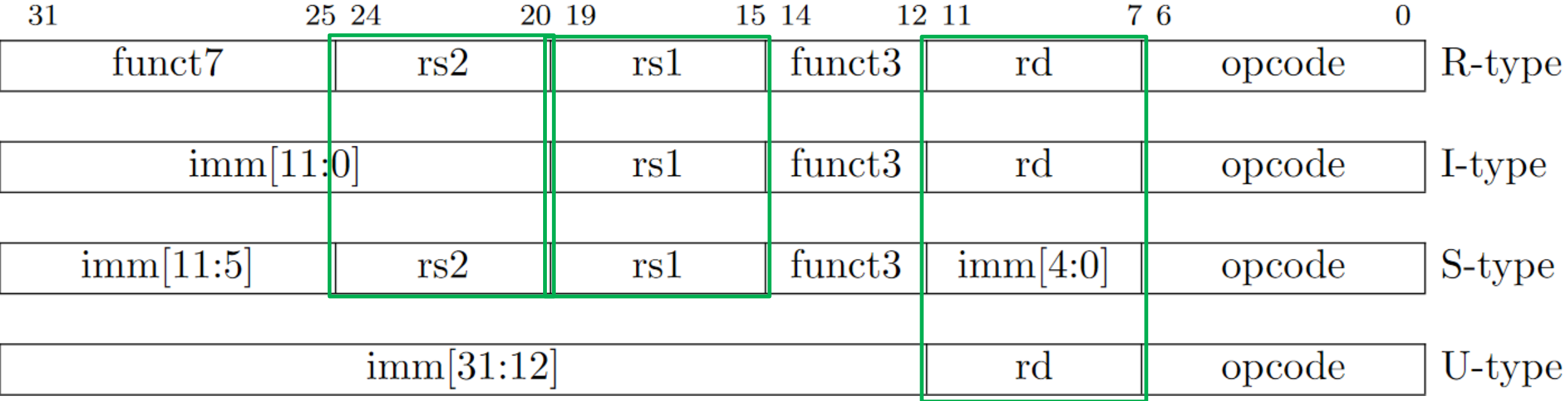
# Design Goals Behind Instruction Encoding

- Why are all of the immediate bits all over the place?

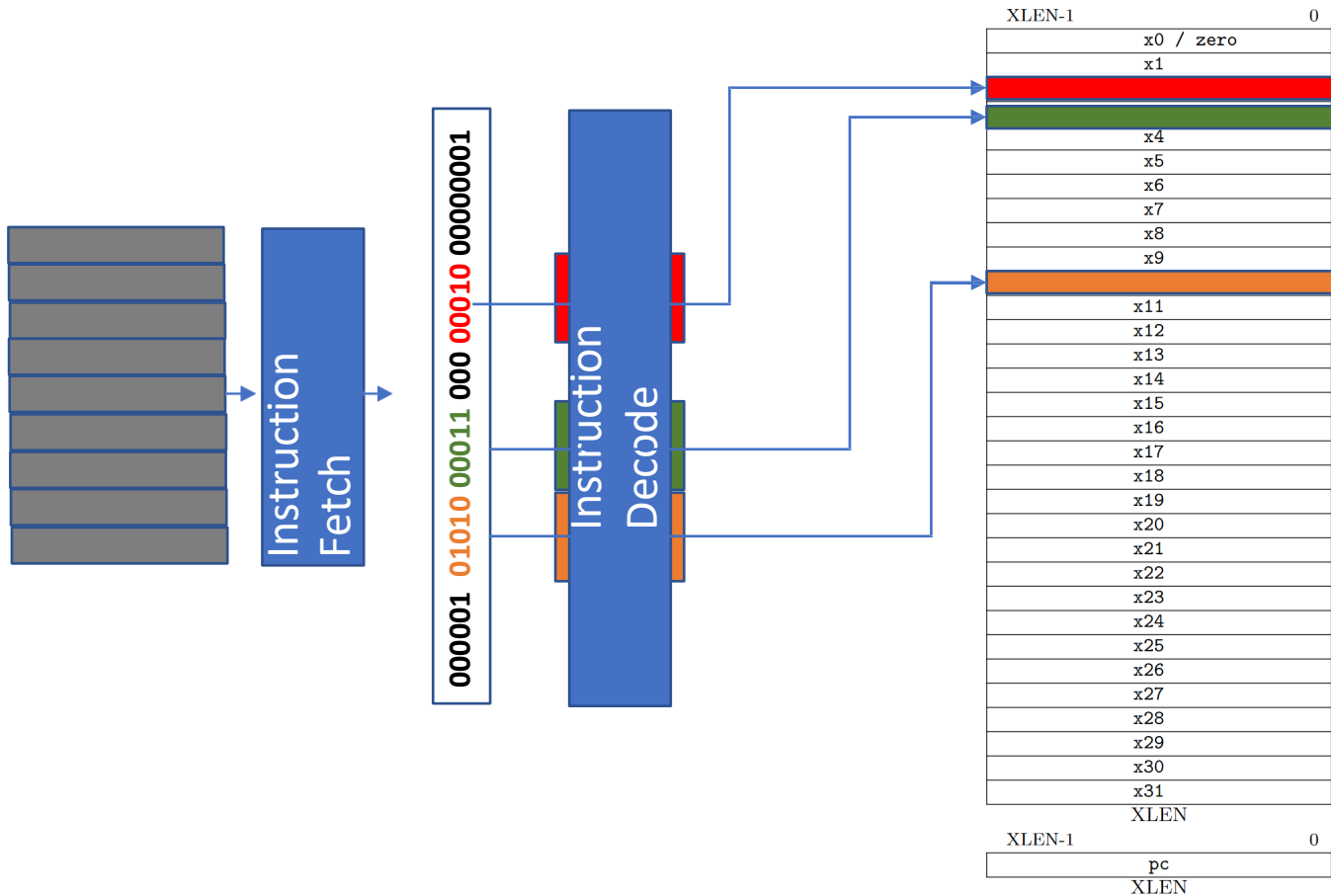


# Design Goals Behind Instruction Encoding

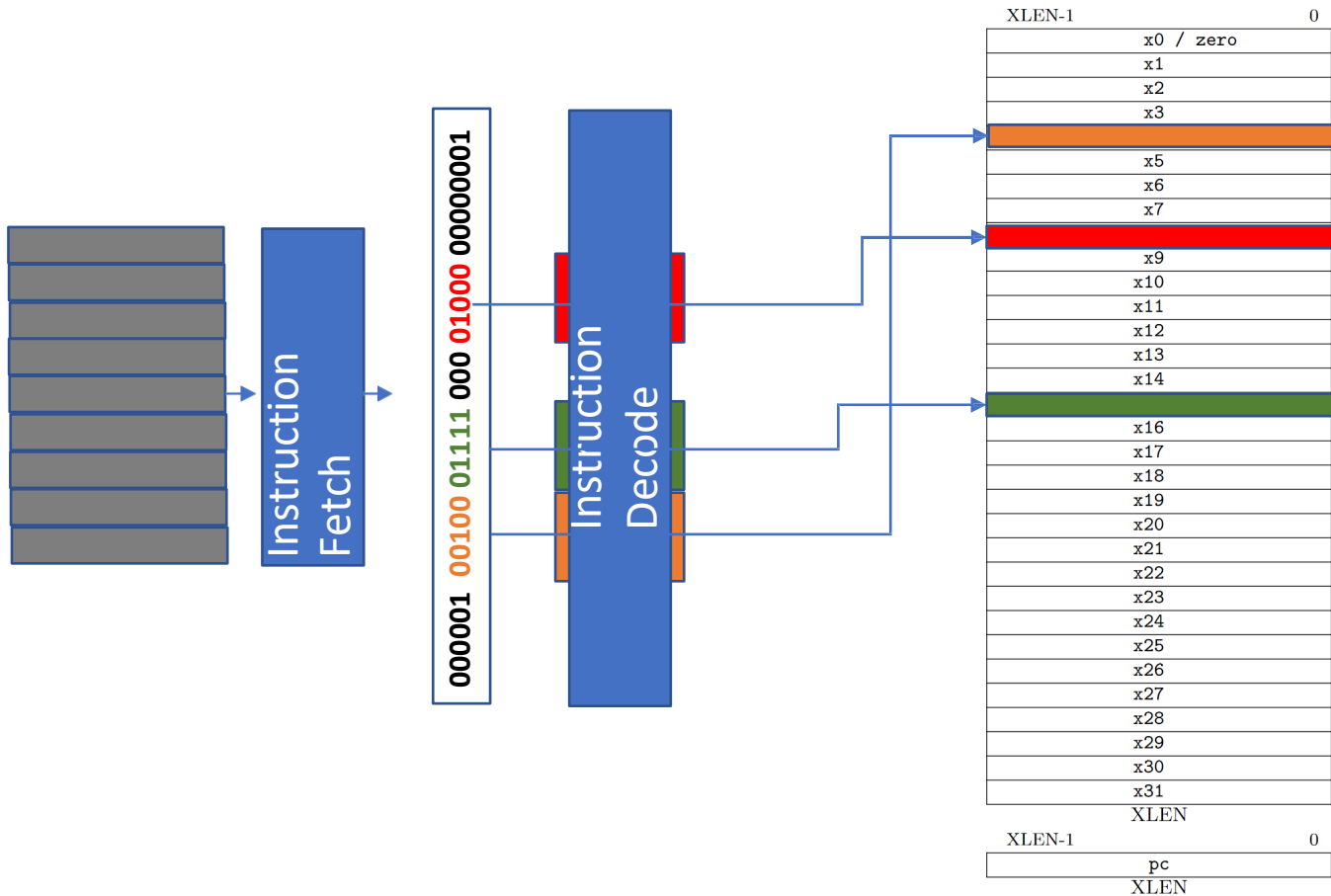
- **Why are all of the immediate bits all over the place?**
  - Instruction decode is expensive & **always** performance critical
  - Registers always in same place makes decoder simpler



# Fixed register position, simple decode logic

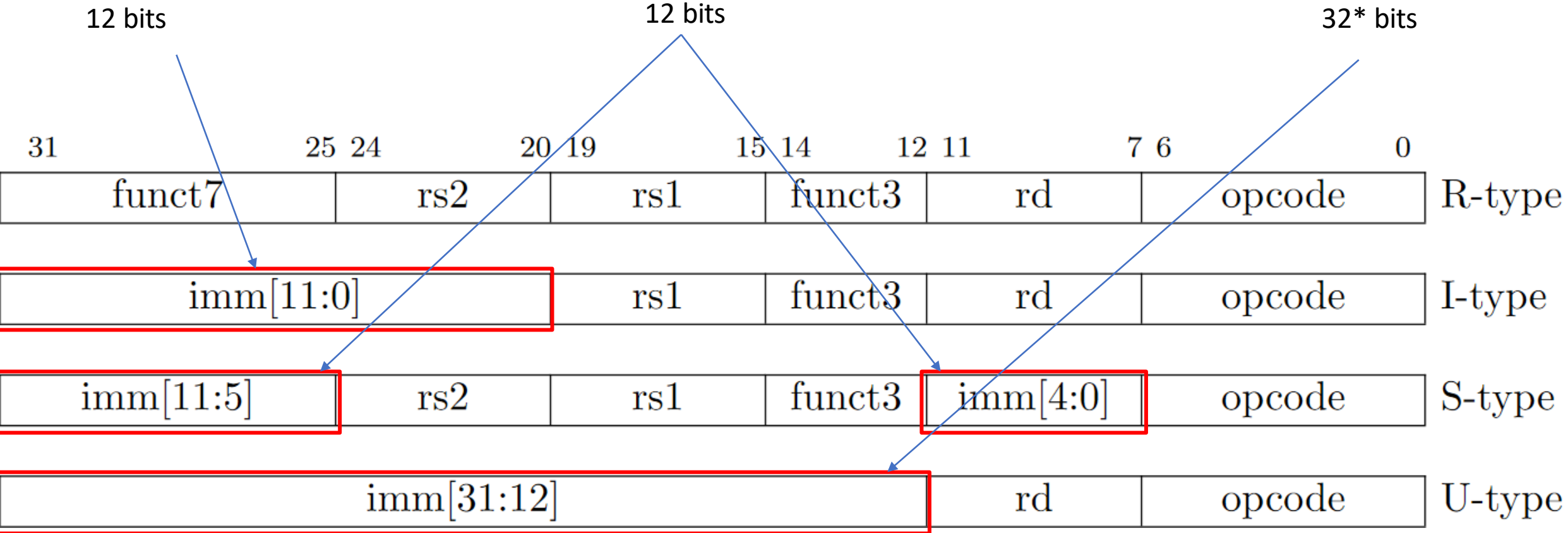


# Fixed register position, simple decode logic



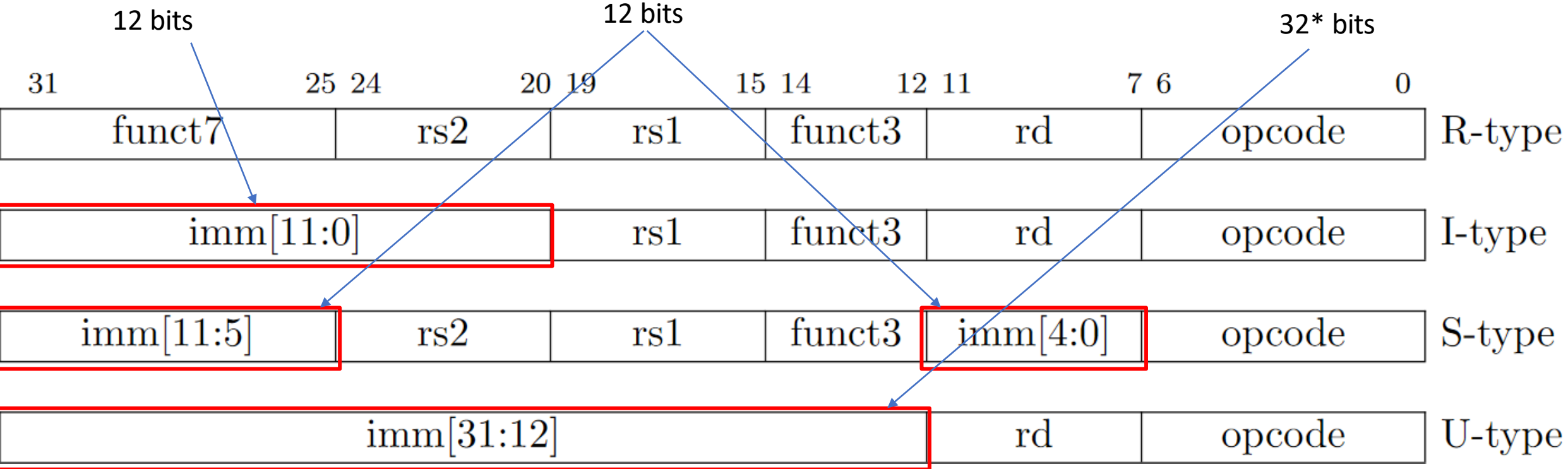
# Design Goals Behind Instruction Encoding

- Why are the immediates apparently different sizes?



# Design Goals Behind Instruction Encoding

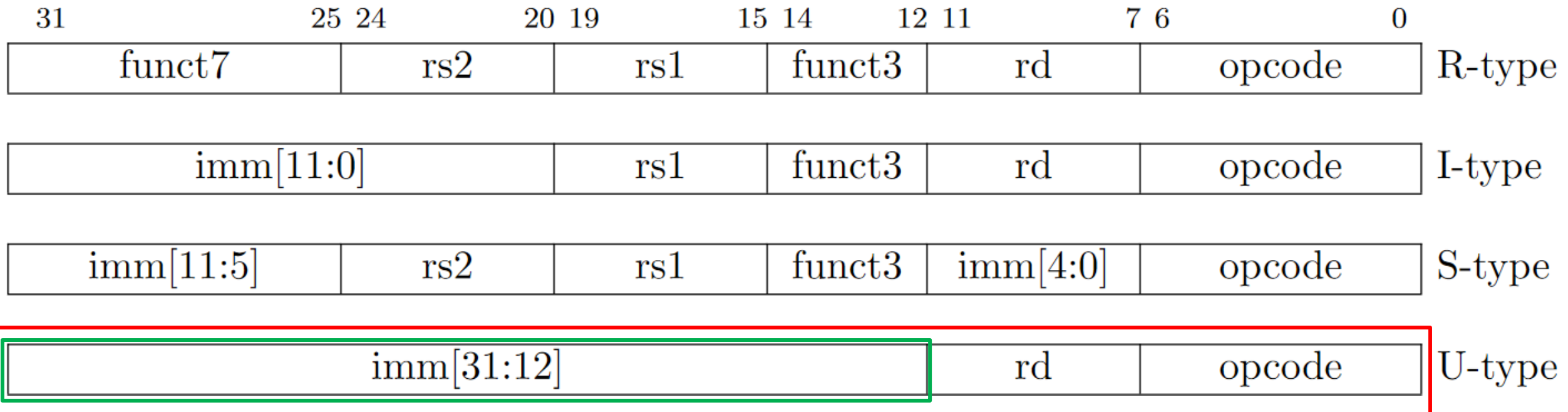
- **Why are the immediates apparently different sizes?**
  - Different immediates have different uses!
  - E.g., 12-bits are enough for PC-relative jump targets





# Design Goals Behind Instruction Encoding

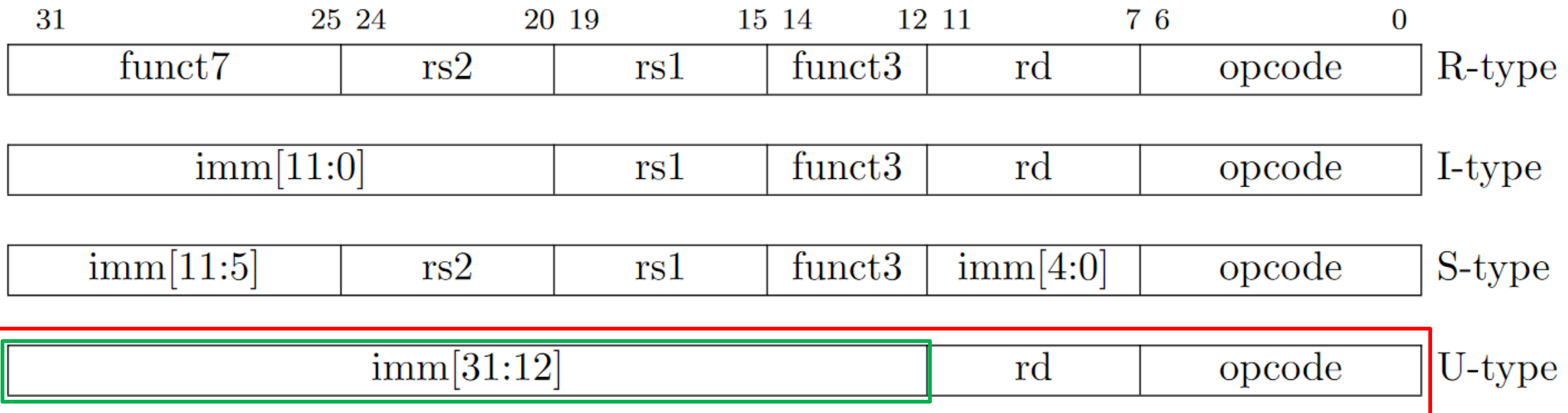
- **Why are some immediate bits left unspecified?**



# Design Goals Behind Instruction Encoding

- **Why are some immediate bits left unspecified?**

- To make large (like, 32-bit) values in a 32-bit insn., need to leave stuff out
- E.g., LUI: store high order bits of a large constant into a register



# Implications of Control Flow Design

- **Benefits & limitations of PC-relative offsets?**

[31] 1	[30:21] 10	[20] 1	[19:12] 8	[11:7] 5	[6:0] 7
I[20]	I[10:1]	I[11]	IMMEDIATE[19:12]	destination	opcode
O[20]	O[10:1]	O[11]	O[19:12]	xD	1101111 : JAL

[31:20] 12	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
IMMEDIATE[11:0]	source 1	function 3	destination	opcode
OFFSET[11:0]	xL	0	xD	1100111 : JALR

[31] 1	[30:25] 6	[24:20] 5	[19:15] 5	[14:12] 3	[11:8] 4	[7] 1	[6:0] 7
I[12]	I[10:5]	source 2	source 1	function 3	I[4:1]	I[11]	opcode
O[12]	O[10:5]	xR	xL	000 : BEQ	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	001 : BNE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	100 : BLT	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	101 : BGE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	110 : BLTU	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	111 : BGEU	O[4:1]	O[11]	1100011 : BRANCH

# Implications of Control Flow Design

- **Benefits & limitations of PC-relative offsets?**

- Compact encoding
- Easy to support position independent code (PIC) if all jumps PC-relative
- Limited reach for jump targets

[31]	[30:21]	[20]	[19:12]	[11:7]	[6:0]
1	10	1	8	5	7
I[20]	I[10:1]	I[11]	IMMEDIATE[19:12]	destination	opcode
O[20]	O[10:1]	O[11]	O[19:12]	xD	1101111 : JAL

[31:20]	[19:15]	[14:12]	[11:7]	[6:0]
12	5	3	5	7
IMMEDIATE[11:0]	source 1	function 3	destination	opcode
OFFSET[11:0]	xL	0	xD	1100111 : JALR

[31]	[30:25]	[24:20]	[19:15]	[14:12]	[11:8]	[7]	[6:0]
1	6	5	5	3	4	1	7
I[12]	I[10:5]	source 2	source 1	function 3	I[4:1]	I[11]	opcode
O[12]	O[10:5]	xR	xL	000 : BEQ	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	001 : BNE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	100 : BLT	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	101 : BGE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	110 : BLTU	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	111 : BGEU	O[4:1]	O[11]	1100011 : BRANCH

# Implications of Control Flow Design

- **Benefits of combined compare & branch?**

[31] 1	[30:21] 10	[20] 1	[19:12] 8	[11:7] 5	[6:0] 7
I[20]	I[10:1]	I[11]	IMMEDIATE[19:12]	destination	opcode
O[20]	O[10:1]	O[11]	O[19:12]	xD	1101111 : JAL

[31:20] 12	[19:15] 5	[14:12] 3	[11:7] 5	[6:0] 7
IMMEDIATE[11:0]	source 1	function 3	destination	opcode
OFFSET[11:0]	xL	0	xD	1100111 : JALR

[31] 1	[30:25] 6	[24:20] 5	[19:15] 5	[14:12] 3	[11:8] 4	[7] 1	[6:0] 7
I[12]	I[10:5]	source 2	source 1	function 3	I[4:1]	I[11]	opcode
O[12]	O[10:5]	xR	xL	000 : BEQ	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	001 : BNE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	100 : BLT	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	101 : BGE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	110 : BLTU	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	111 : BGEU	O[4:1]	O[11]	1100011 : BRANCH

# Implications of Control Flow Design

- **Benefits of combined compare & branch?**

- No management of implicit (& explicit) condition codes like x86, ARM, SPARC...
- Higher code density, reduced instruction fetch traffic,
- Execution reaches branch sooner in instruction stream allowing earlier prediction

[31]	[30:21]	[20]	[19:12]	[11:7]	[6:0]
1	10	1	8	5	7
I[20]	I[10:1]	I[11]	IMMEDIATE[19:12]	destination	opcode
O[20]	O[10:1]	O[11]	O[19:12]	xD	1101111 : JAL

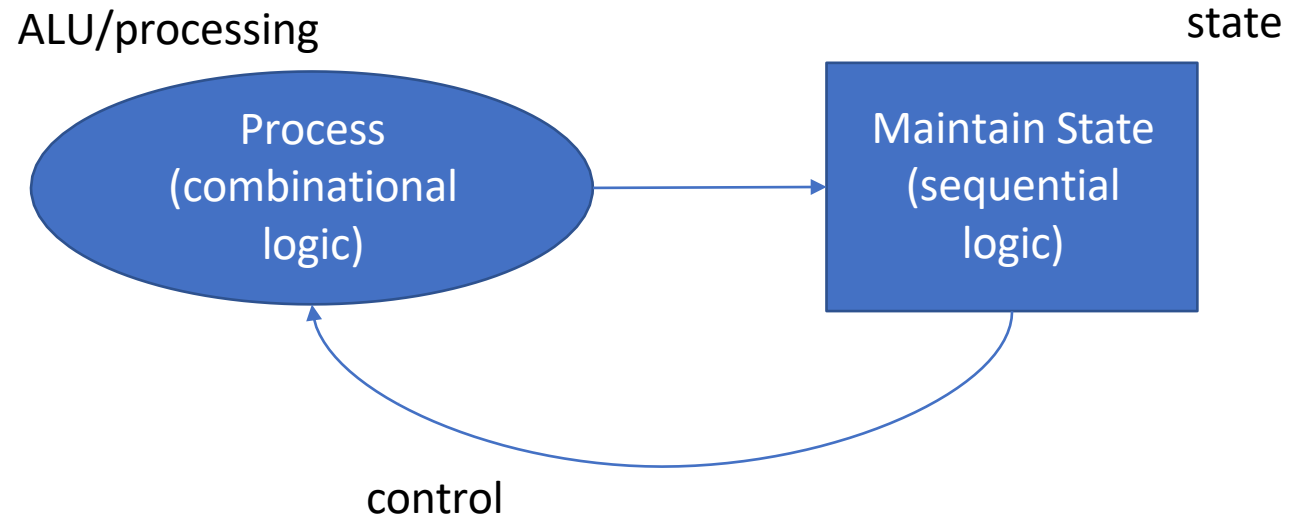
[31:20]	[19:15]	[14:12]	[11:7]	[6:0]
12	5	3	5	7
IMMEDIATE[11:0]	source 1	function 3	destination	opcode
OFFSET[11:0]	xL	0	xD	1100111 : JALR

[31]	[30:25]	[24:20]	[19:15]	[14:12]	[11:8]	[7]	[6:0]
1	6	5	5	3	4	1	7
I[12]	I[10:5]	source 2	source 1	function 3	I[4:1]	I[11]	opcode
O[12]	O[10:5]	xR	xL	000 : BEQ	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	001 : BNE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	100 : BLT	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	101 : BGE	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	110 : BLTU	O[4:1]	O[11]	1100011 : BRANCH
O[12]	O[10:5]	xR	xL	111 : BGEU	O[4:1]	O[11]	1100011 : BRANCH

# Other Design Dissiderata

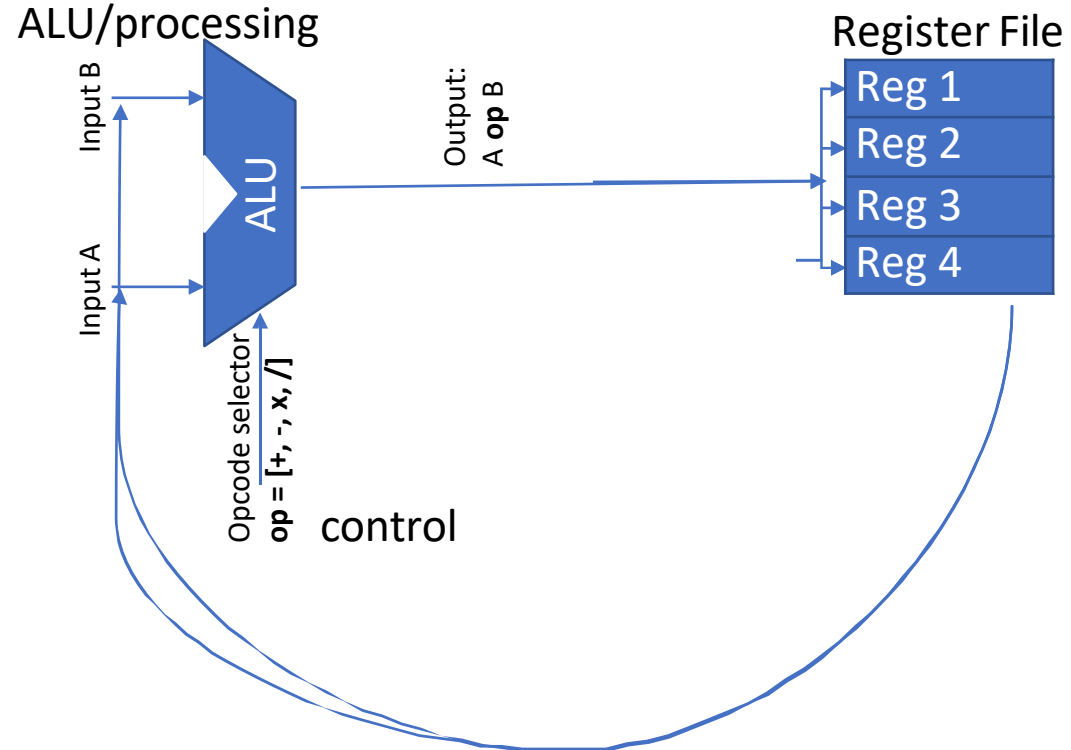
- **Differences from x86 (for better and for worse)**
  - No complex addressing modes
  - No conditional move (like x86 cmov) or predicated execution
  - Much smaller basic ISA implementation (<http://ref.x86asm.net/coder.html>)
- **Memory Model Considerations (more on this toward end of semester)**
  - **Alignment Issues:** *“The base ISA supports misaligned accesses, but these might run extremely slowly depending on the implementation. Furthermore, naturally aligned loads and stores are guaranteed to execute atomically, whereas misaligned loads and stores might not, and hence require additional synchronization to ensure atomicity.”*
  - **Memory Consistency Model:** *“We chose a relaxed memory model to allow high performance from simple machine implementations, however a completely relaxed memory model is too weak to support programming language memory models and so the memory model is being tightened.” [this point is basically the riscv team apologizing for having punted on concurrency to begin with and walking back to something that makes sense]*

# What *kind* of ISA are we studying?



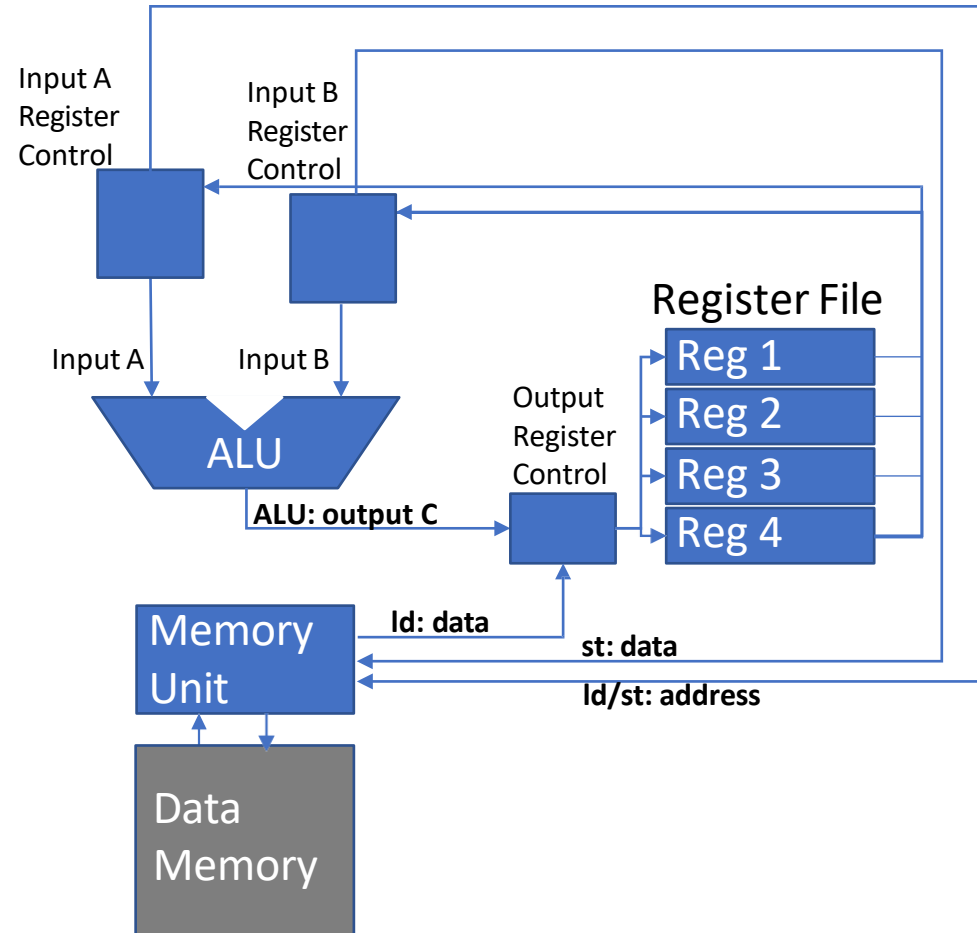


# RISCV is a “Register / Register” ISA



state – inputs & outputs are in register file

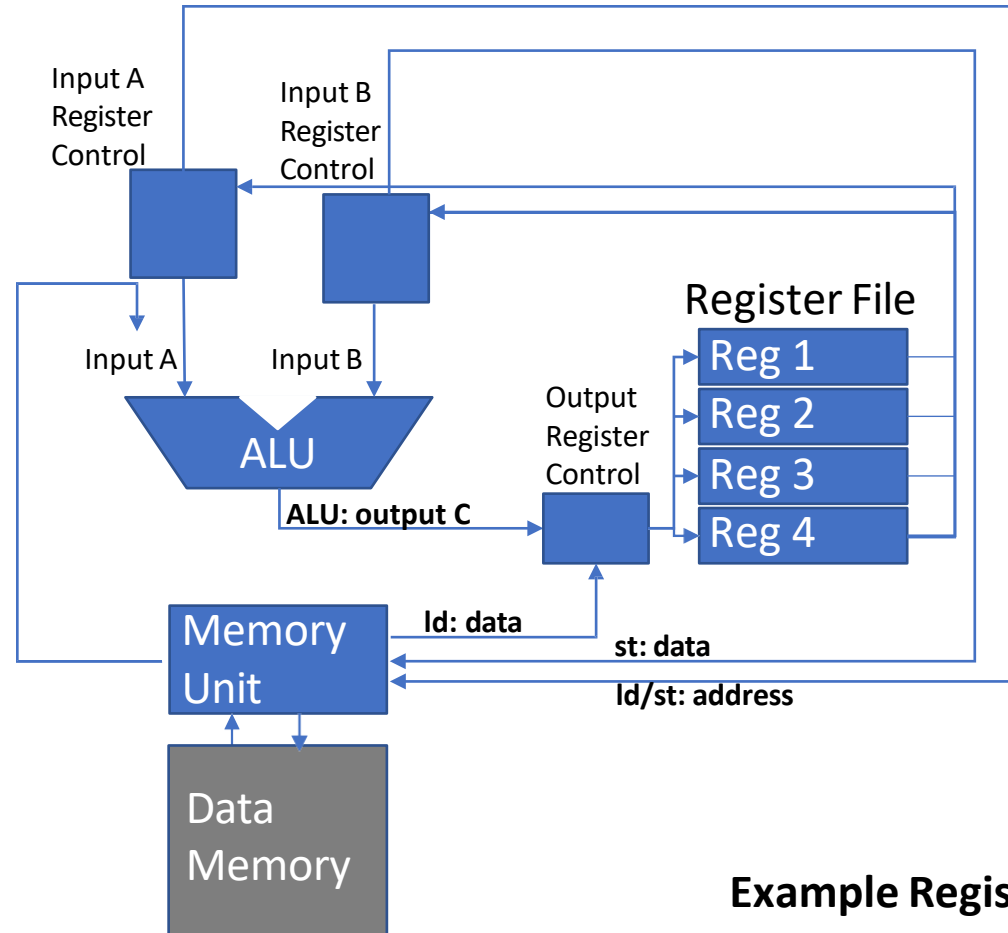
# RISCV is a “Register / Register” ISA



state – inputs & outputs are in register file  
**Register / Register? we have memory tho?**

# Alternative: “Register / Memory” ISA

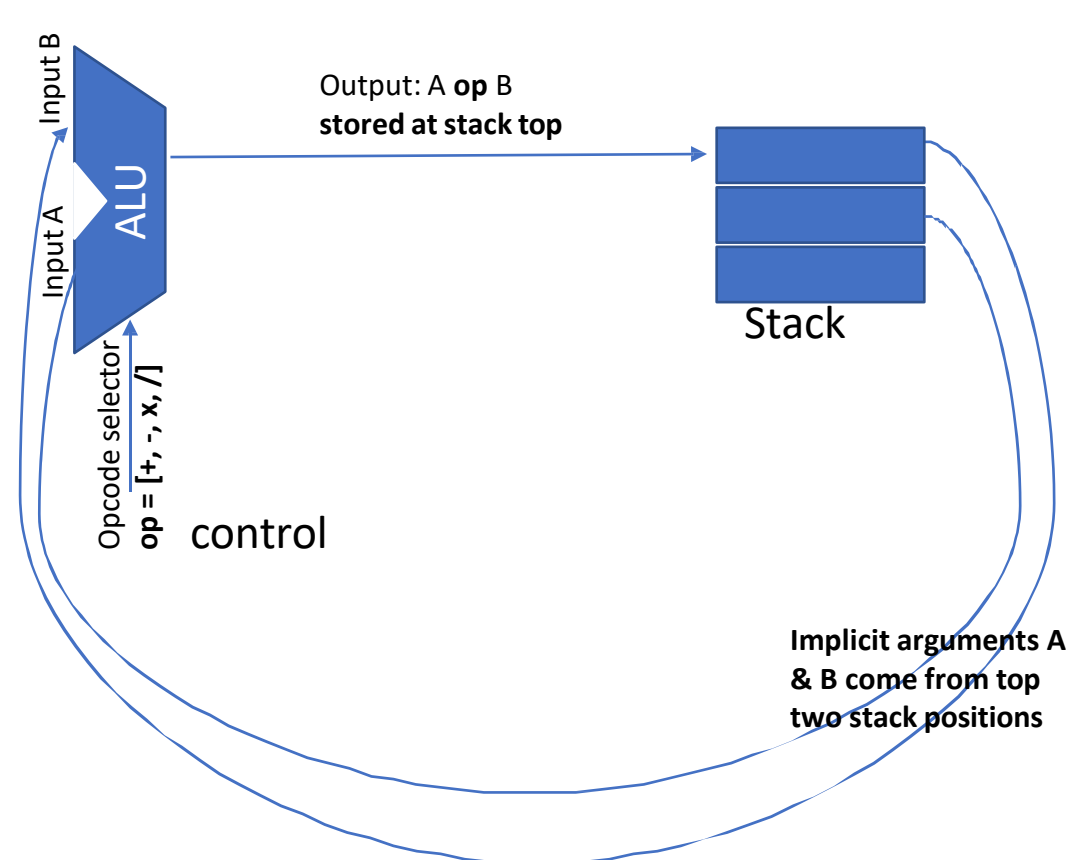
Zero or one inputs comes from memory in a R/M architecture



state – inputs & outputs are in register file  
**Register / Register?**  
**we have memory tho?**

Example Register / Memory architectures?

# Alternative: Stack Machine Architecture



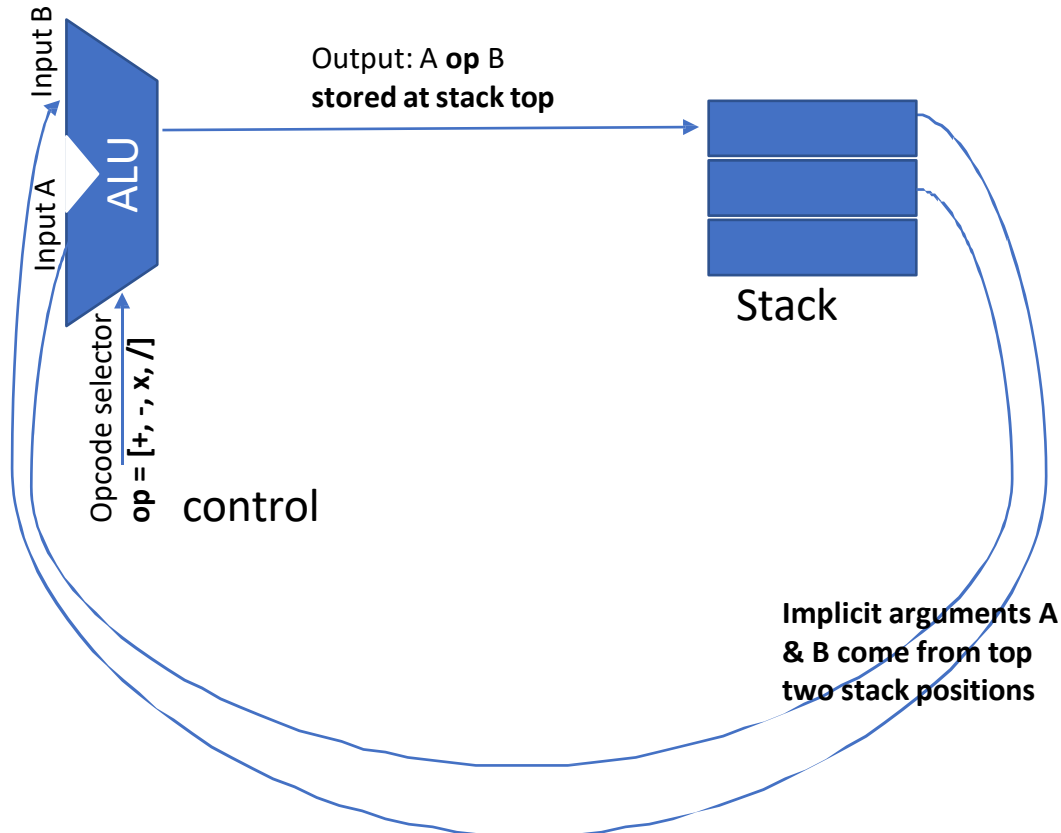
state – in/out  
operands from stack  
only. there are **no**  
**registers** in this  
architecture!

**(Access memory via  
ld/st to stack top)**

# Alternative: Stack Machine Architecture

```
push (0xabc)
push (0xac0)
addw
push (0x123)
push $10 //imm. 10
mulw
addw
pop (0xdeadbeef)
```

What does this do?



state – in/out  
operands from stack  
only. there are no  
registers in this  
architecture!

**Memory is weird:**  
**push <addr> loads**  
**<addr> and pushes**  
**to stack top, pop**  
**<addr> stores stack**  
**top to <addr>**

# Alternative: Stack Machine Architecture

## The Z4 digital computer (1942 + later)

```
push (0xabc)
push (0xac0)
addw
push (0x123)
push $10 //imm. 10
mulw
addw
pop (0xdeadbeef)
```

What does this do?



state – in/out  
operands from stack  
only. there are no  
registers in this  
architecture!

**Memory is weird:**  
**push <addr> loads**  
**<addr> and pushes**  
**to stack top, pop**  
**<addr> stores stack**  
**top to <addr>**

# Alternative: Stack Machine Architecture

## The Java Virtual Machine is a stack machine

```
push (0xabc)
push (0xac0)
addw
push (0x123)
push $10 //imm. 10
mulw
addw
pop (0xdeadbeef)
```

What does this do?



# Java™

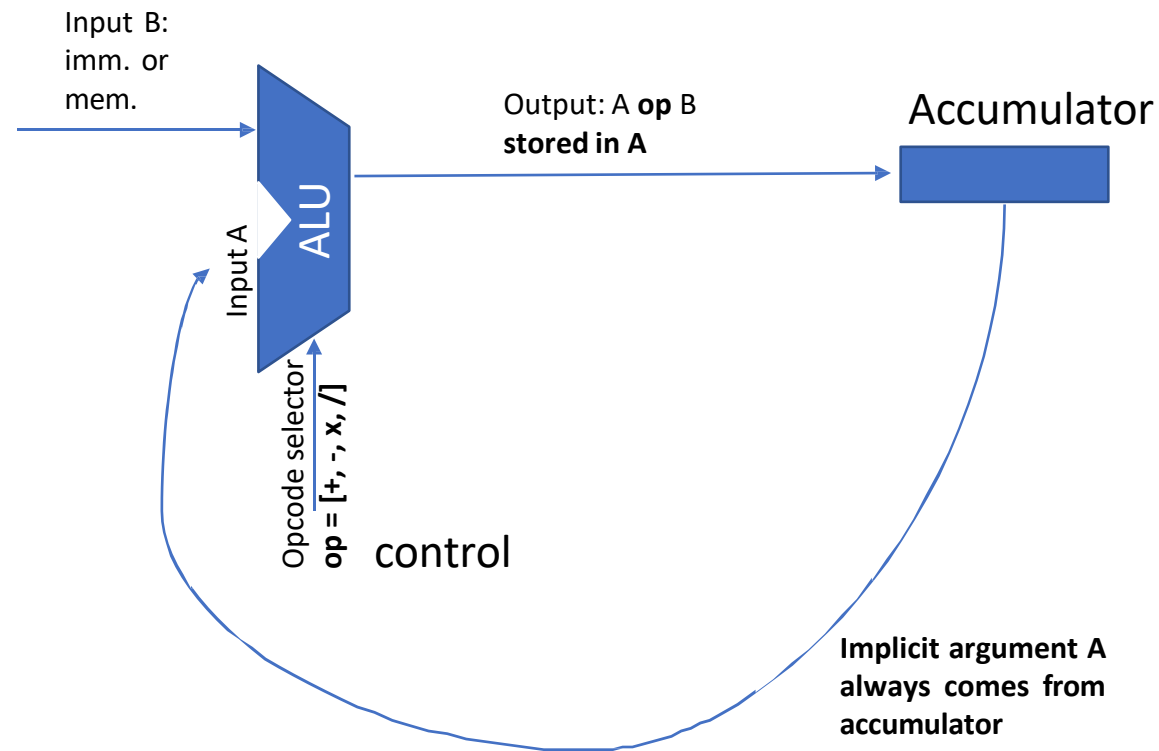
state – in/out  
operands from stack  
only. there are no  
registers in this  
architecture!

**Memory is weird:**  
**push <addr> loads**  
**<addr> and pushes**  
**to stack top, pop**  
**<addr> stores stack**  
**top to <addr>**

# Alternative: Accumulator Machine Architecture

```
load (0xabc)
add (0xac0)
store (0xf00)
load (0x123)
mul 10
add (0xf00)
store (0xdeadbeef)
```

What does this do?



state – no registers, implicit argument is **A** (the accumulator), other arg is imm. / mem.

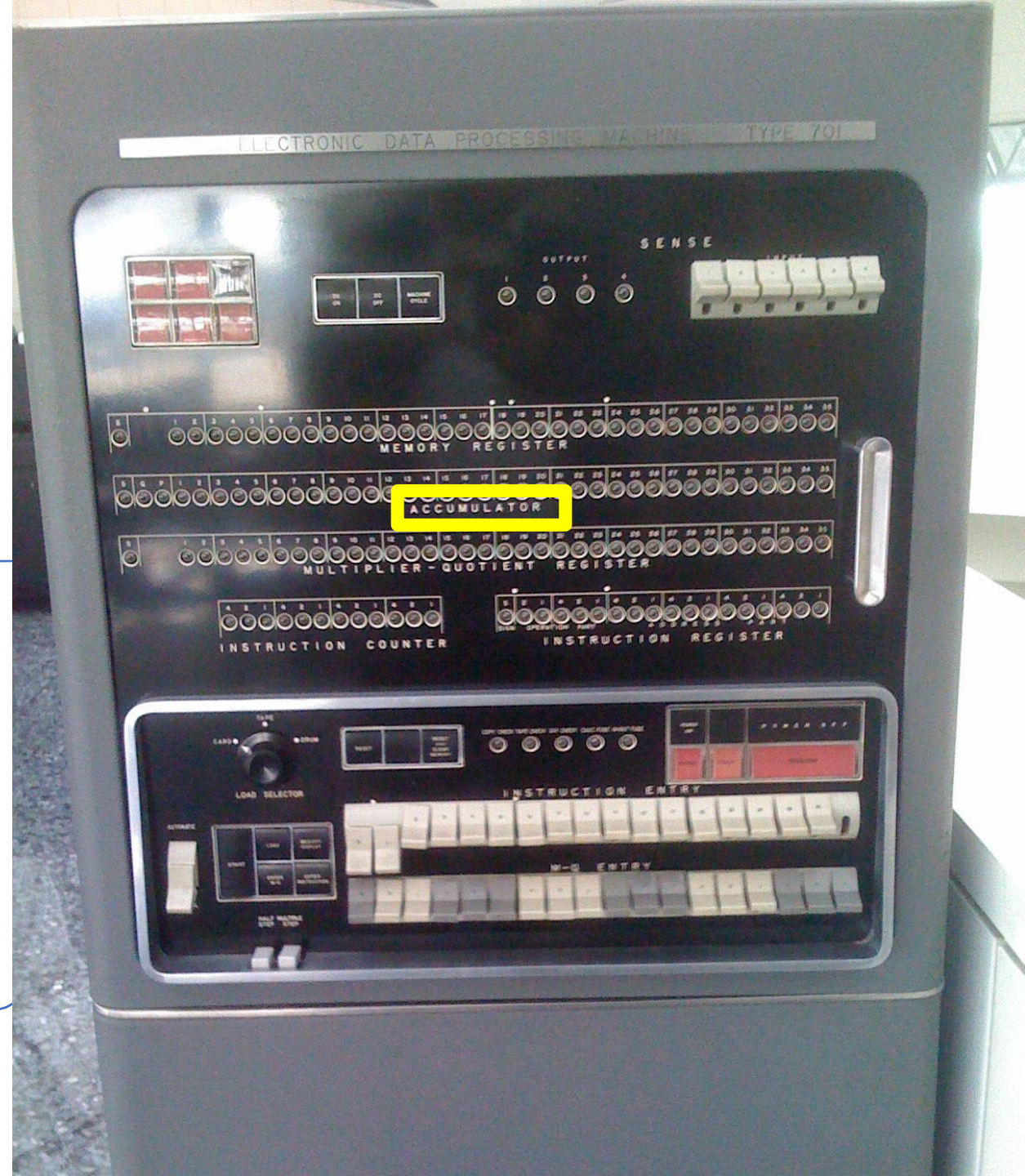
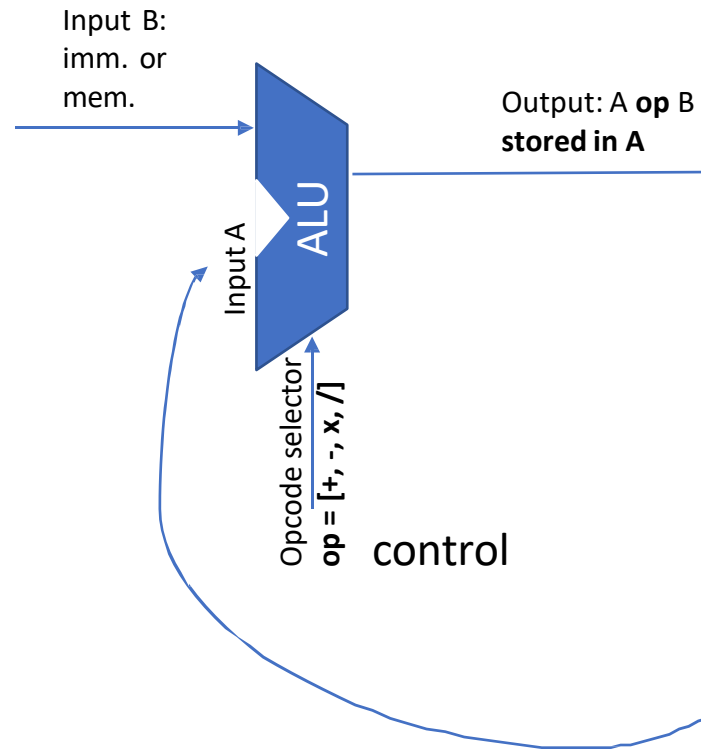
**Memory is less weird:** load (0xabc) puts the contents of 0xabc in A, store (0xabc) puts A into memory at 0xabc



# IBM 701 (ca. 1952)

```
load (0xabc)
add (0xac0)
store (0xf00)
load (0x123)
mul 10
add (0xf00)
store (0xdeadbeef)
```

What does this do?



# What did we just learn?

- A deep-ish dive into the RISC-V ISA as a vehicle for learning about ISA design
- A look at how ISA design choices influence other aspects of the system's design
- How to we cross the hardware/software interface to go from software to hardware

# What to think about next?

- More microarchitectural concepts (next time)
  - Pipelining our microarchitecture & instruction-level parallelism
  - Control hazards & branch prediction
- Caches as a microarchitectural optimization (next next time)
  - Implementation of cache hierarchies
  - Cache design tradeoffs
- Performance Evaluation (looking forward)
  - Design spaces, Pareto Frontiers, and design space exploration