

UC Berkeley  
Teaching Professor  
Dan Garcia

# CS61C

## Great Ideas in Computer Architecture (a.k.a. Machine Structures)

### RISC-V Instruction Formats, Part I

# Machine Language

- Machine Language
- R-Format Layout
- I-Format Layout
- I-Format: Load
- S-Format Layout (Store)

# Great Idea #1: Abstraction (Levels of Representation/Interpretation)



High Level Language  
Program (e.g., C)

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

| *Compiler*  
Assembly Language  
Program (e.g., RISC-V)

```
lw    x3, 0(x10)
lw    x4, 4(x10)
sw    x4, 0(x10)
sw    x3, 4(x10)
```

Anything can be represented  
as a number,  
i.e., data or instructions

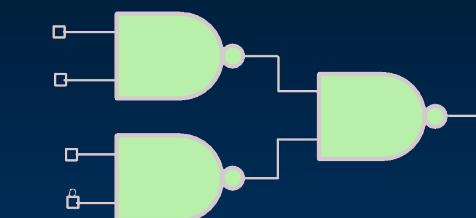
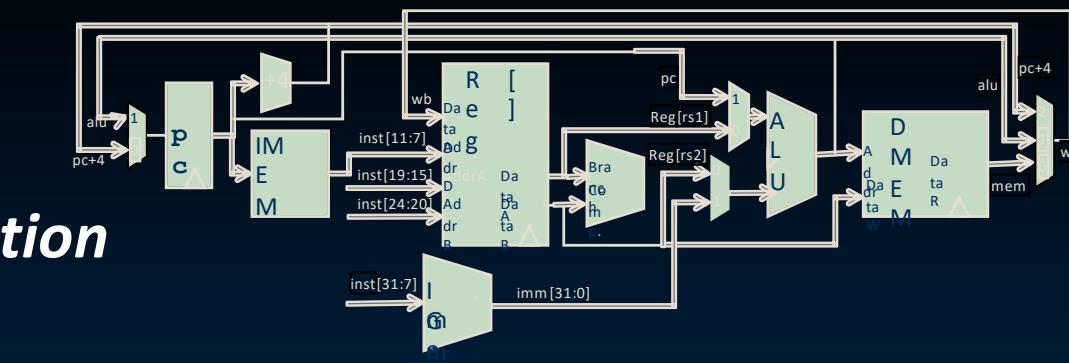
| *Assembler*  
Machine Language  
Program (RISC-V)

1000	1101	1110	0010	0000	0000	0000	0000	0000
1000	1110	0001	0000	0000	0000	0000	0000	0100
1010	1110	0001	0010	0000	0000	0000	0000	0000
1010	1101	1110	0010	0000	0000	0000	0000	0100

Hardware Architecture Description  
(e.g., block diagrams)

| *Architecture Implementation*

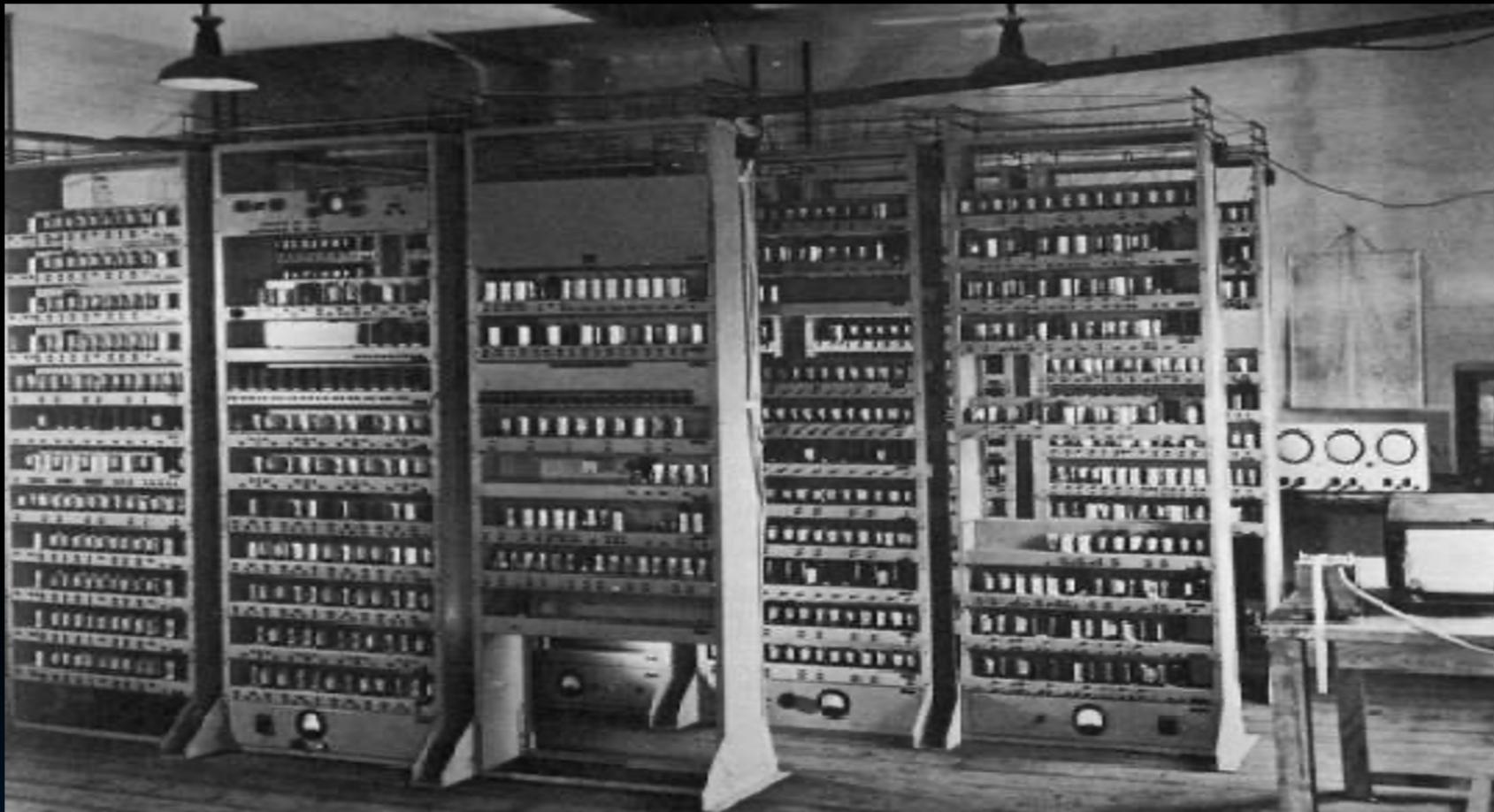
Logic Circuit Description  
(Circuit Schematic Diagrams)



# Quick CS History: ENIAC (UPenn, 1946)



# Quick CS History: EDSAC (Cambridge, 1949)

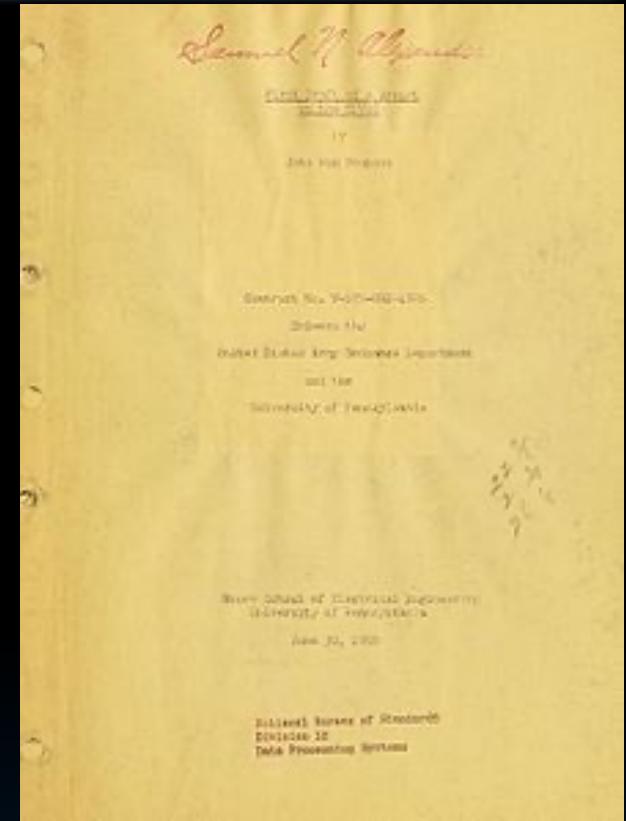


- First General Stored-Program Electronic Computer
- 17-bit words (35-bit double word), binary 2's complement
- Memory: 512 words
- Multiplication: 6ms
- Subroutines

# History's Big Idea: Stored Program Computer

- Instructions are represented as bit patterns (i.e., numbers).
- Therefore, entire programs can be stored to memory to be read or written, just like data.  
This means reprogramming can happen quickly (seconds); don't have to rewire computer (days).
- **Stored-Program Computer AKA the “von Neumann” computer is named after the widely distributed tech report**

Based on discussions of EDVAC (1944-): Eckert, Mauchly  
Anticipated earlier by Turing and Zuse



First Draft of a Report on the EDVAC  
By John von Neumann  
Contract No. W-670-ORD-4926  
Between the  
United States Army Ordnance Department  
and the  
University of Pennsylvania  
Moore School of Electrical Engineering  
University of Pennsylvania  
June 30, 1945



# Stored Program Computer: Implications

## 1. Everything (instructions, data words) has a memory address.

Both branches and jumps use instruction memory addresses.

C pointers are just memory addresses and can point to anything.

Unconstrained use of addresses lead to nasty bugs! Up to C programmer to avoid errors

The Program Counter (PC) register keeps address of instruction being executed.

PC is effectively a pointer to memory; Intel calls it Instruction Pointer (PC).

## 2. Programs are distributed in *binary form*, i.e., as assembled machine code.

Programs are each bound to a specific instruction set.

Different instruction sets, e.g., for phone vs. PCs.

Many instruction sets are backwards-compatible and evolve over time.

E.g., latest PCs with x86 ISA today can still run programs from Intel 8088 (1981)!

# Instructions Are 32 Bits Wide

- ISA defines instructions for CPU down to the bit level:

**add x18, x19, x10** → 00000000101010011000100100110011  
assembly code *machine code*

Remember: Computer only understands 1s and 0s.

Text like “add x18,x19,x10” is meaningless to hardware.

- Most data we work with is in words (32-bit chunks).  
32b registers; **lw**, **sw** access memory one word at a time.
  - Similarly, RISC-V instructions are fixed size, 32-bit words.

RISC-V seeks simplicity: since data is in words, make instructions be fixed-size 32-bit words also.

## Same 32-bit instructions used for RV32, RV64, RV128

# Instruction Formats

- **RISC-V's 32b instruction words are divided into fields.**  
Each field tells processor something about the instruction.
- **The RISC-V ISA defines six basic types of instruction formats.**  
RISC-V seeks simplicity: Avoid defining different fields for each instruction; instead, use same format for similar instructions.

R-Format	Register-register arithmetic operations		today
I-Format	Register-immediate arithmetic operations; Loads		
S-Format	Stores		
B-Format	Branches (minor variant of S-format)		
U-Format	20-bit upper immediate instructions		
J-Format	Jumps (minor variant of U-format)		

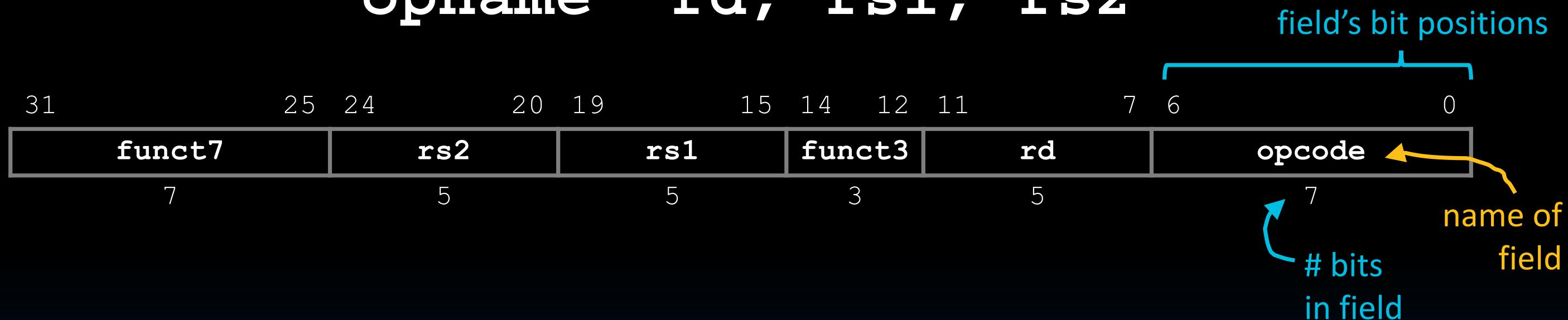
# R-Format Layout

- Machine Language
- R-Format Layout
- I-Format Layout
- I-Format: Load
- S-Format Layout (Store)

# R-Format Instruction Layout

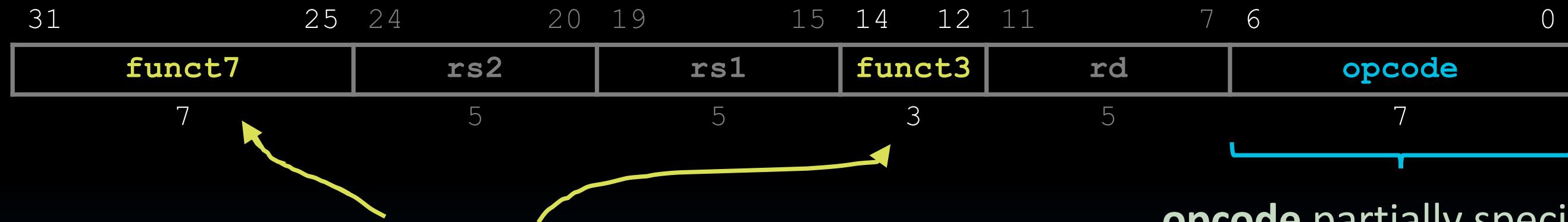
- Register-Register Arithmetic Instructions (add, xor, sll, etc.)

opname      **rd**,    **rs1**,    **rs2**



# R-Format Instruction Layout

- Register-Register Arithmetic Instructions (add, xor, sll, etc.)  
**opname      rd, rs1, rs2**



**funct7, funct3** combined with **opcode** describes what operation to perform.

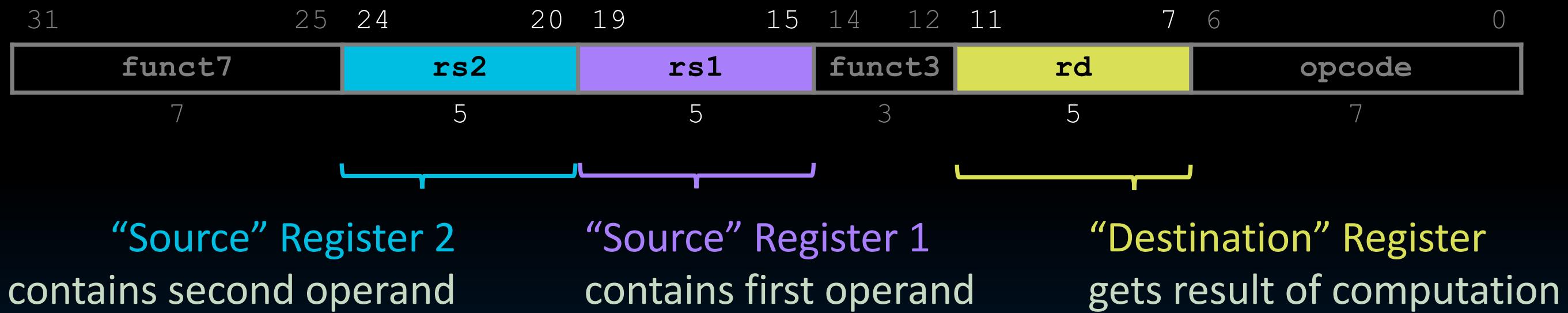
Why not just a 17-bit field for simplicity?  
We'll answer this later...

**opcode** partially specifies which instruction it is.

All R-Format instructions have opcode 0110011.

# R-Format Instruction Layout

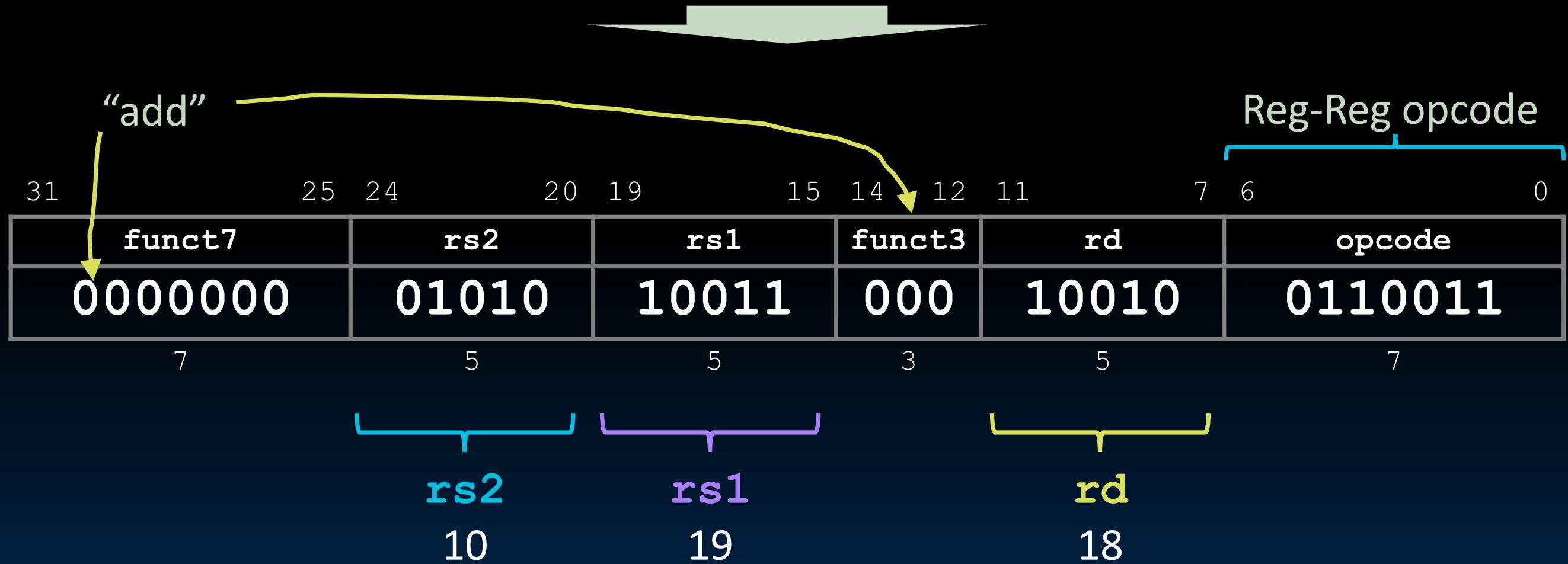
- Register-Register Arithmetic Instructions (add, xor, sll, etc.)  
**opname      rd, rs1, rs2**



**Register field (rs1, rs2, rd) holds a 5-bit unsigned integer [0-31] corresponding to a register number (x0-x31).**

# R-Format Example

add x18, x19, x10



# All ten RV32 R-Format Instructions

Eight funct3 fields for ten instructions

	funct7		funct3		opcode	
2's comp negation of rs2	0000000	rs2	rs1	000	rd	0110011 add
	0100000	rs2	rs1	000	rd	0110011 sub
	0000000	rs2	rs1	001	rd	0110011 sll
	0000000	rs2	rs1	010	rd	0110011 slt
	0000000	rs2	rs1	011	rd	0110011 sltu
	0000000	rs2	rs1	100	rd	0110011 xor
	0000000	rs2	rs1	101	rd	0110011 srl
	0100000	rs2	rs1	101	rd	0110011 sra
	0000000	rs2	rs1	110	rd	0110011 or
	0000000	rs2	rs1	111	rd	0110011 and

} set less than (see refcard)

Different funct7 & funct3 encodings select different operations.

# Your Turn

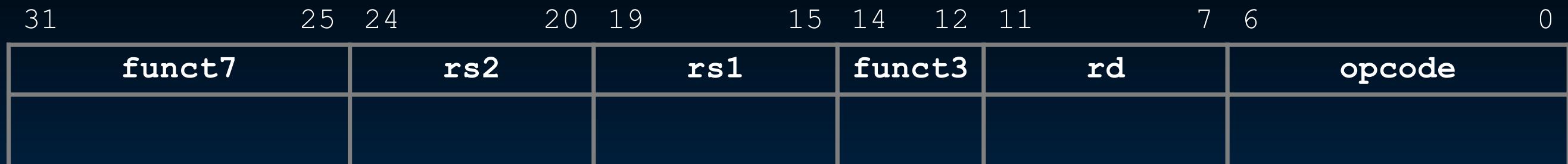
How do we encode

add x4, x3, x2 ?

Lookup table:

funct7	rs2	rs1	funct3	rd	opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	111	rd	0110011	and

- A. 4021 8233<sub>hex</sub>
- B. 0021 82b3<sub>hex</sub>
- C. 4021 82b3<sub>hex</sub>
- D. 0021 8233<sub>hex</sub>
- E. 0021 8234<sub>hex</sub>
- F. Something else



# L11 How do we encode add x4, x3, x2?

0

 Your Turn

How do we encode **add x4, x3, x2 ?**

Lookup table:

	funct7	funct3	opcode			
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	111	rd	0110011	and

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

funct7	rs2	rs1	funct3	rd	opcode

4021 8233

0%

0021 82b3

0%

4021 82b3

0%

0021 8233

0%

0021 8234

0%

Something else

0%

# Your Turn

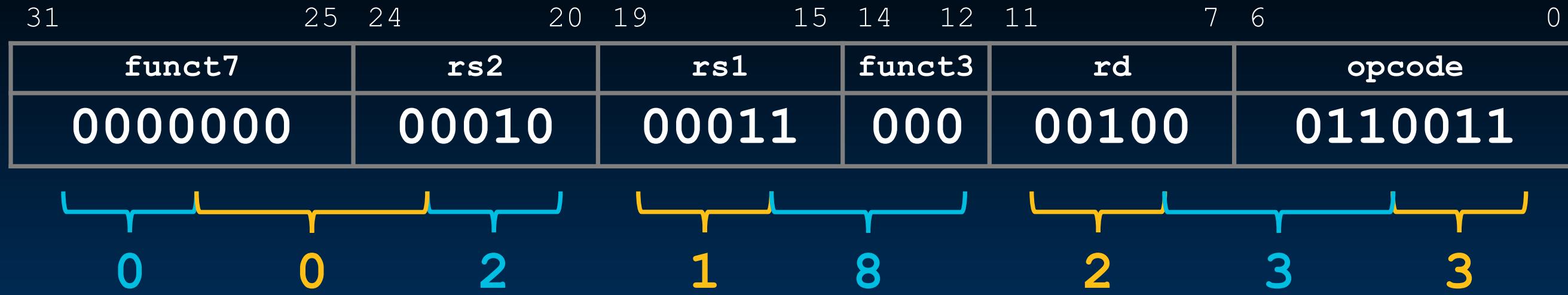
How do we encode

add x4, x3, x2 ?

Lookup table:

funct7	rs2	rs1	funct3	rd	opcode	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	111	rd	0110011	and

- A. 4021 8233<sub>hex</sub>
- B. 0021 82b3<sub>hex</sub>
- C. 4021 82b3<sub>hex</sub>
- D. 0021 8233<sub>hex</sub>
- E. 0021 8234<sub>hex</sub>
- F. Something else

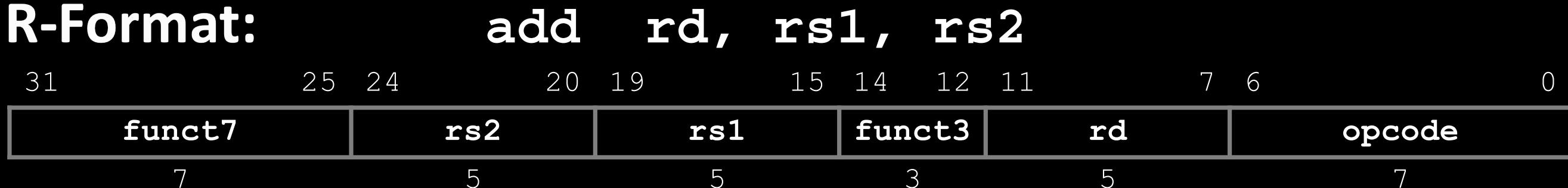


# I-Format Layout

- Machine Language
- R-Format Layout
- I-Format Layout
- Load (I-Format)
- S-Format Layout (Store)

# Immediate Fields Need to be Wider

- R-Format:



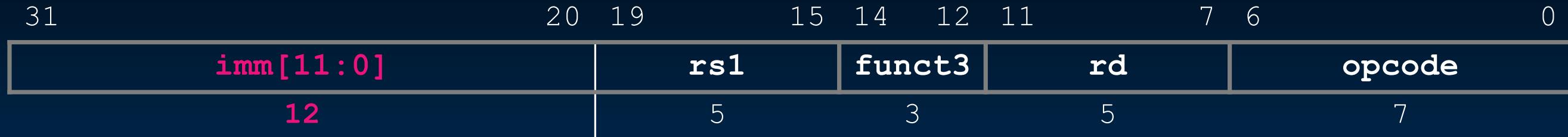
- What about instructions with immediates?

**addi rd, rs1, imm**

If we used R-Format, we'd only be able to represent 32 values in the 5-bit **rs2** field.

Immediates may be (and are often) much bigger!

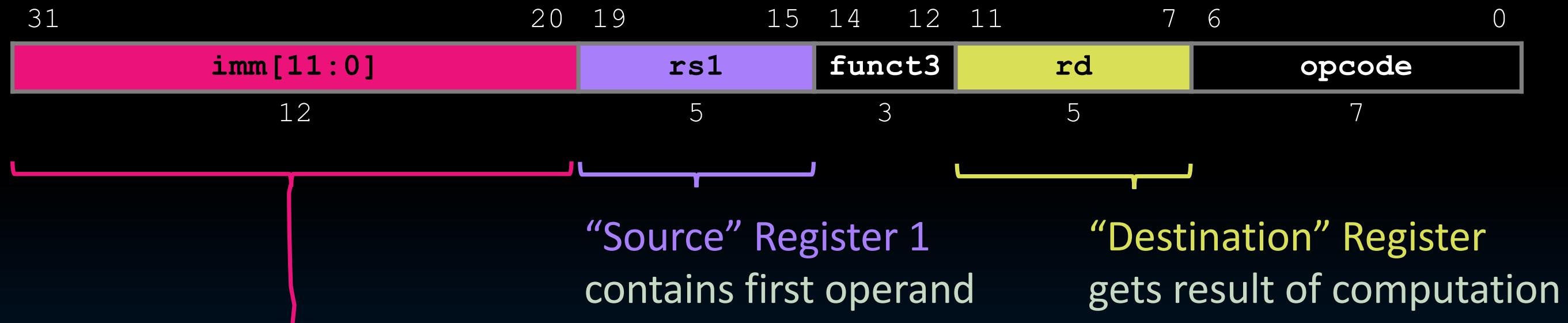
- Enter the I-Format:



I-Format is *mostly* consistent with R-Format. Simplify how the CPU processes instructions!

# I-Format Instruction Layout

- Register-Immediate Arithmetic Instructions (`addi`, `xori`, etc.)  
**opname    rd, rs1, imm**



**Imm[11:0]** holds 12-bit-wide immediate values:

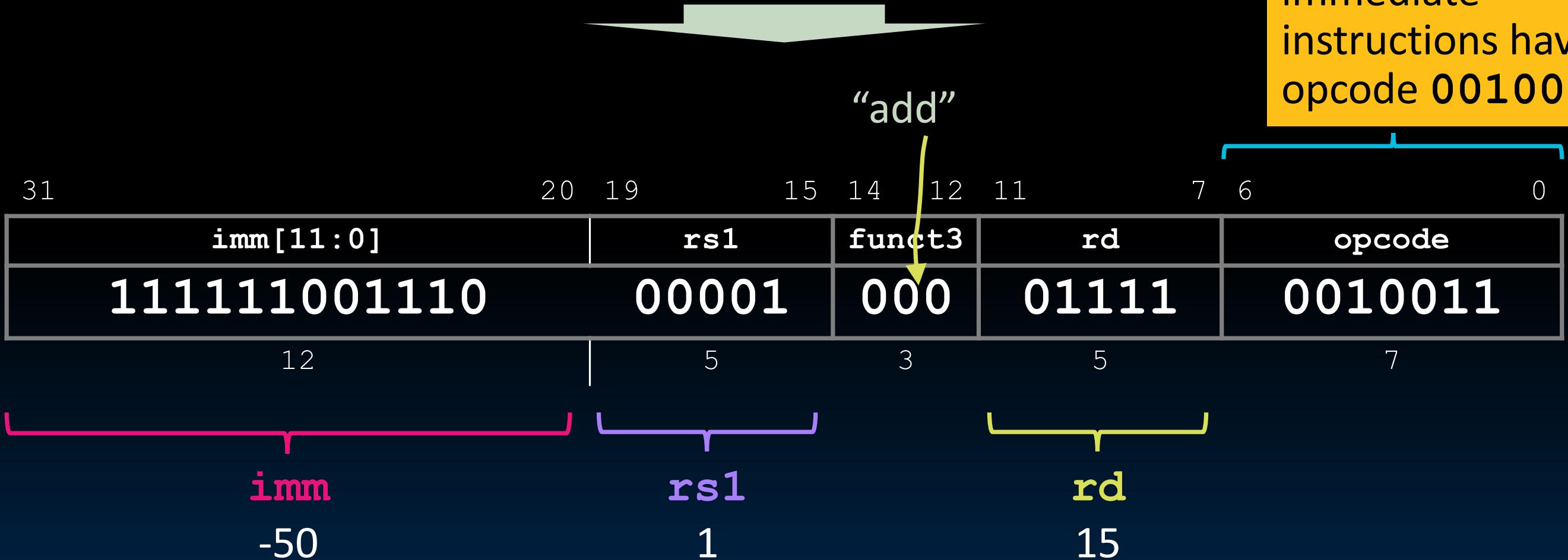
Values in range  $[-2048_{\text{ten}}, +2047_{\text{ten}}]$

CPU sign-extends to 32 bits before use in an arithmetic operation

How to handle immediates > 12 bits? (more next time)

# I-Format Example

addi x15, x1, -50



All arithmetic immediate instructions have opcode `0010011`.

# All nine RV32 I-Format Arithmetic Instructions

Same funct3 fields as corresponding R-format operation (remember, no **subi**)

			funct3			
				opcode		
imm[11:0]	rs1	000	rd	0010011	addi	
imm[11:0]	rs1	010	rd	0010011	slti	
imm[11:0]	rs1	011	rd	0010011	sltiu	
imm[11:0]	rs1	100	rd	0010011	xori	
imm[11:0]	rs1	110	rd	0010011	ori	
imm[11:0]	rs1	111	rd	0010011	andi	
0000000	shamt	001	rd	0010011	slli	
0000000	shamt	rs1	101	rd	0010011	srl
0100000	shamt	rs1	101	rd	0010011	srai

“Shift by Immediate” instructions encode the shift amount in the lower-order 5 bits.

We can only (meaningfully) shift 32-b word by 0-31 positions.

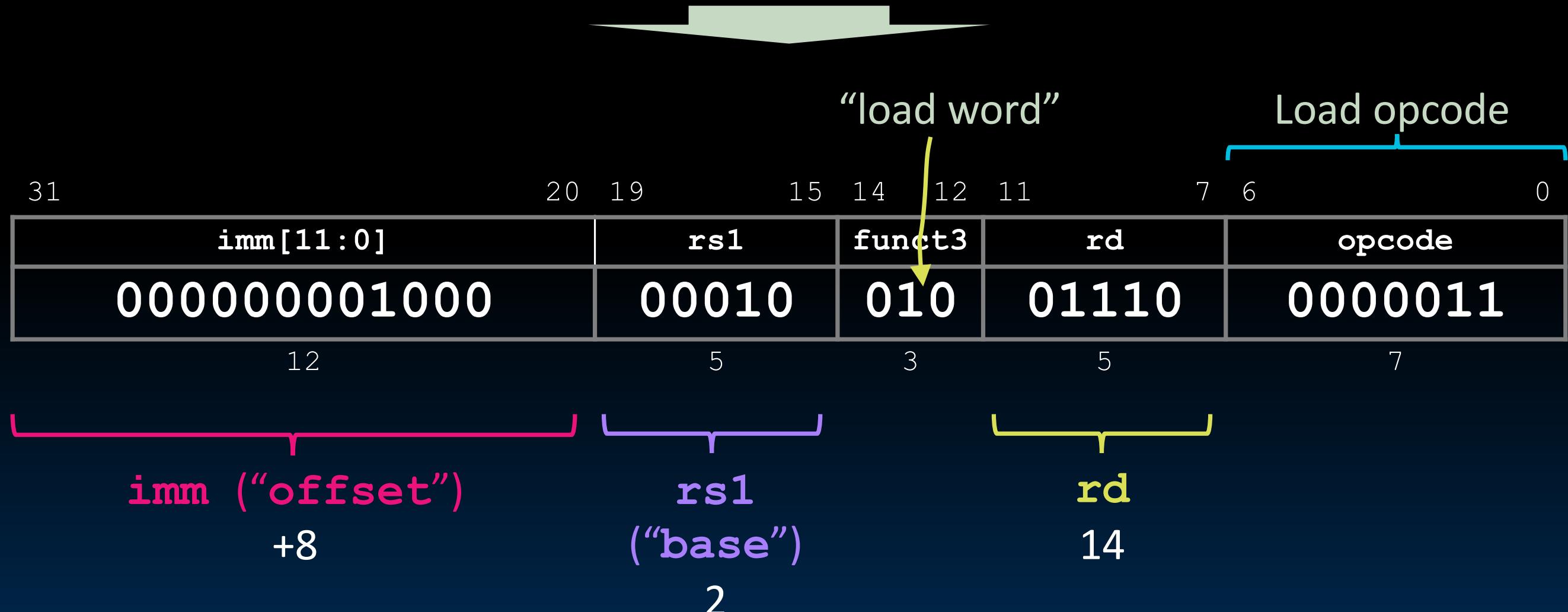
One higher-order immediate bit used for sign extend (**srl** vs. **srai**). Same bit position as in R-Format!

# I-Format: Load

- Machine Language
- R-Format Layout
- I-Format Layout
- I-Format: Load
- S-Format Layout (Store)

# Load Instructions Are Also I-Format: Example

lw x14, 8(x2)



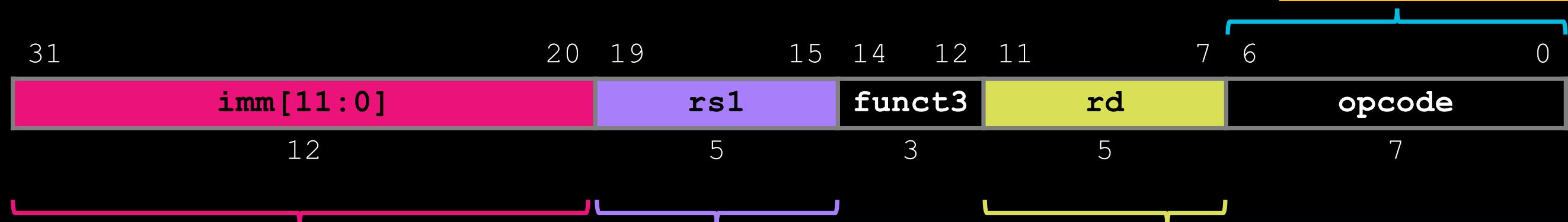
$$\text{Load address} = (\text{Base Register}) + (\text{Immediate Offset})$$

# I-Format Instruction Layout: Loads

- Load instructions use I-Format:

**loadop    rd,    imm (rs1)**

All load instructions have opcode **0000011**.



Immediate “**offset**” added to base address to form memory address

“**base**” Register  
Base address of load

“Destination” Register receives value loaded from memory

# All five RV32 Load Instructions

Encodes data size and “signedness” of load operation

**funct3**

**opcode**

imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu

- **lb:** “load byte,” **lh:** “load halfword” (16 bits)

*Sign extend* to fill upper bits of destination 32-bit register.

- **lbu:** “load unsigned byte,” **lhu:** “load unsigned halfword”

*Zero extend* to fill upper bits of destination 32-bit register.

- Note no **lwu** instruction in RISC-V.

Simplicity: No need to sign/zero extend when copying 32 bits into 32-bit register.



# S-Format Layout (Store)

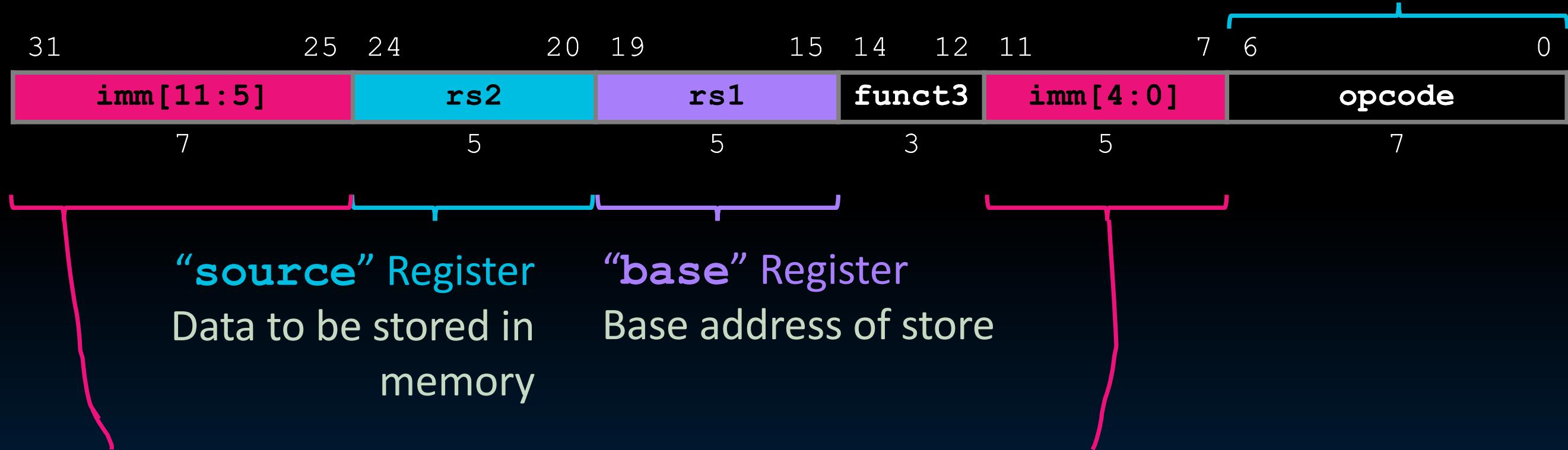
- Machine Language
- R-Format Layout
- I-Format Layout
- I-Format: Load
- S-Format Layout (Store)

# S-Format Instruction Layout

- Store instructions have their S-Format.

**storeop      rs2, imm (rs1)**

All store instructions have opcode 0100011.



Immediate “**offset**” added to base address to form memory address.

Store address = (Base Register) + (Immediate Offset)

The immediate’s higher 7 bits and lower 5 bits are in separate fields!

# S-Format Simplifies Hardware Design

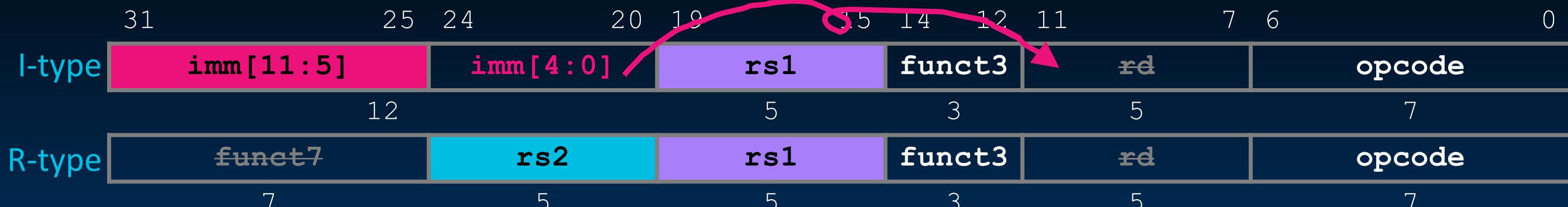
- Why split up the immediate field?

**storeop    rs2 , imm (rs1)**



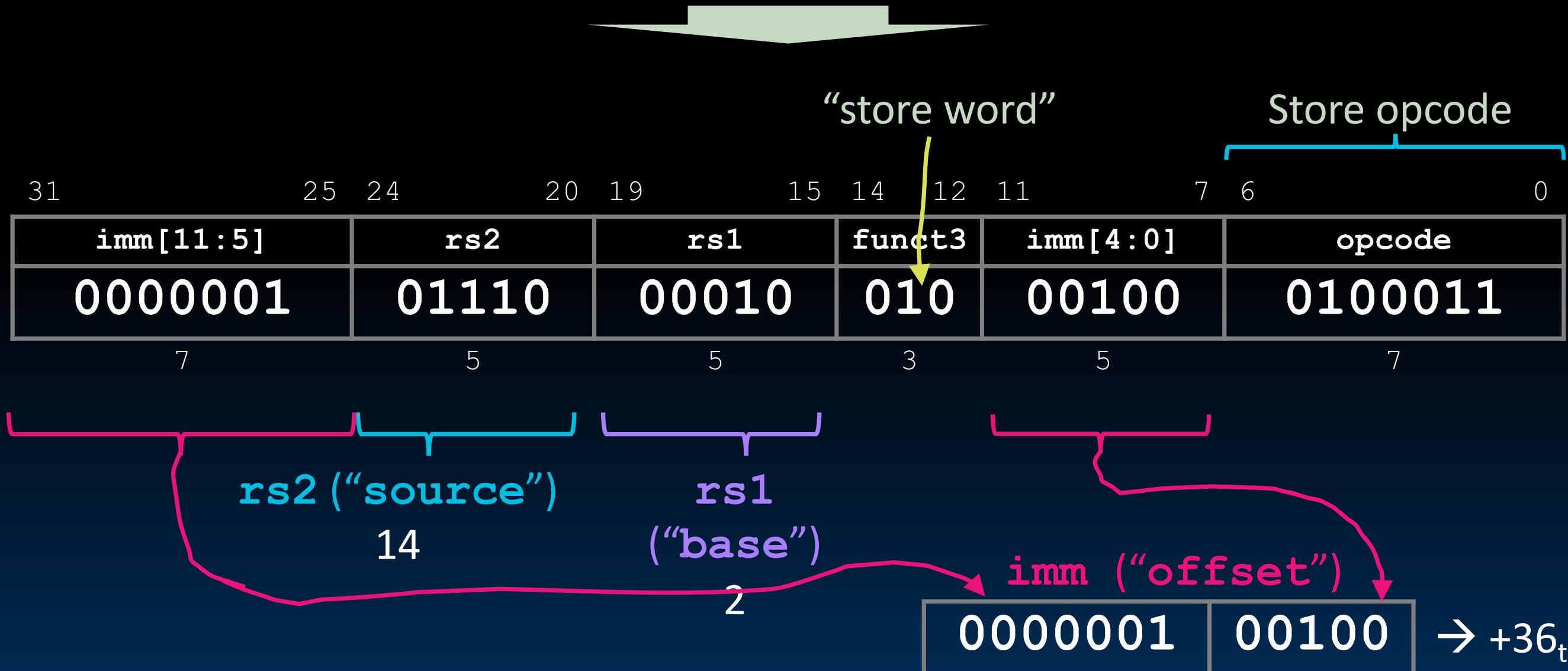
RISC-V design prioritizes keeping *register fields* in the same places. Immediates are less critical to hardware.

- Store needs immediate, two read registers, no destination register!
- Move lower 5 bits of immediate to rd field location in other formats:



# S-Format Example

**sw x14, 36(x2)**



# All three RV32 Store Instructions

			funct3	opcode		
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	sb
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	sh
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	sw

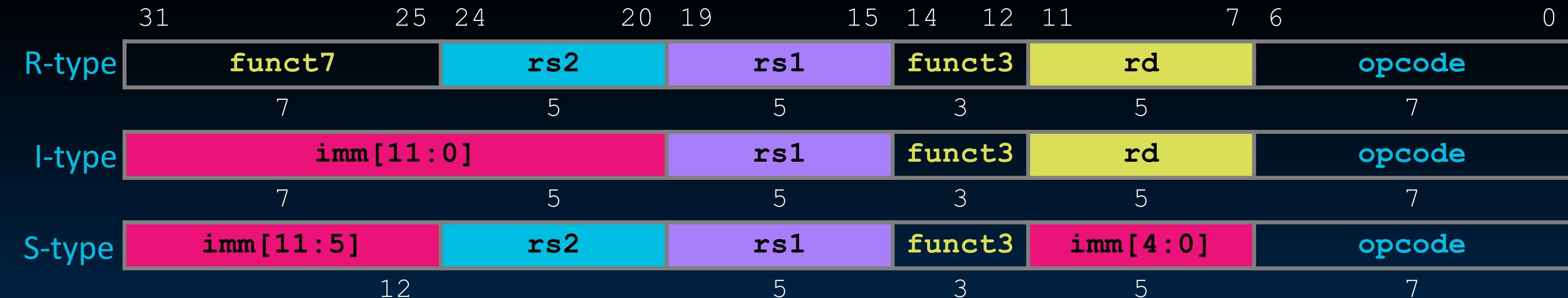
No sign/zero extending for store!  
We only write to memory  
the data width specified.

Same data width fields as  
load instructions.

# “And in Conclusion...”

- Simplification works for RISC-V: Instructions are same size as data word (32 bits) so that they use the same memory.
- Computer program stored as a series of these 32-bit numbers.
- RISC-V machine language instruction formats:

Today:



Next time: B-type, U-type, J-type