UC Berkeley
Teaching Professor
Dan Garcia

# CS61C

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

UC Berkeley
Professor
Bora Nikolić

# RISC-V Instruction Representation

# 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)
```

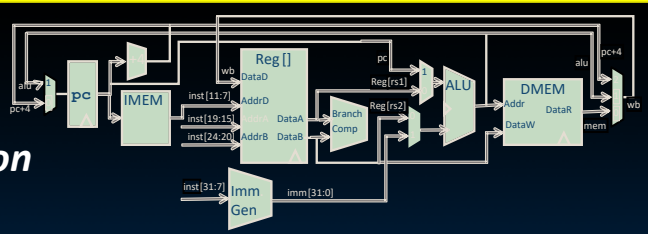
Anything can be represented as a number, i.e., data or instructions

*Assembler*

Machine Language Program (RISC-V)

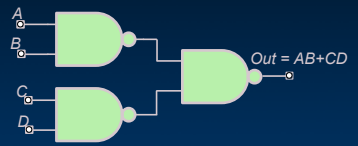
```
1000 1101 1110 0010 0000 0000 0000 0000
1000 1110 0001 0000 0000 0000 0000 0100
1010 1110 0001 0010 0000 0000 0000 0000
1010 1101 1110 0010 0000 0000 0000 0100
```

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

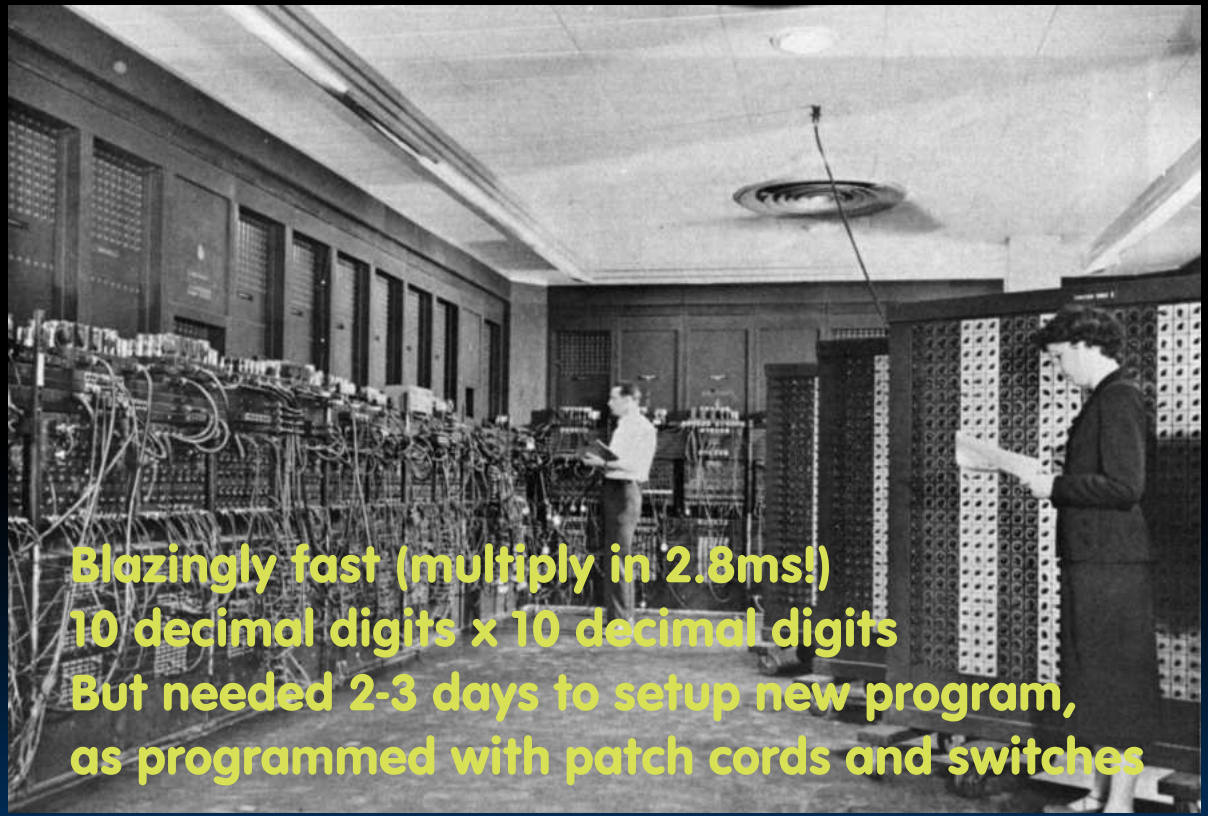



*Architecture Implementation*

Logic Circuit Description (Circuit Schematic Diagrams)

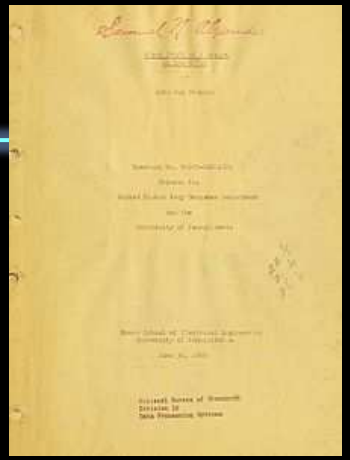


# ENIAC (U.Penn., 1946)
# First Electronic General-Purpose Computer



# Big Idea: Stored-Program Computer

- Instructions are represented as bit patterns – can think of these as numbers
- Therefore, entire programs can be stored in memory to be read or written just like data
- Can reprogram quickly (seconds), don't have to rewire computer (days)
- Known as the "von Neumann" computers after widely distributed tech report on EDVAC project
  - Wrote-up discussions of Eckert and 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

# EDSAC (Cambridge, 1949): First General Stored-Program Electronic Computer

# Consequence #1: Everything Has a Memory Address

- Since all instructions and data are stored in memory, everything has a memory address: instructions, data words
  - Both branches and jumps use these
- C pointers are just memory addresses: they can point to anything in memory
  - Unconstrained use of addresses can lead to nasty bugs; avoiding errors up to you in C; limited in Java by language design
- One register keeps address of instruction being executed: "Program Counter" (PC)
  - Basically a pointer to memory
  - Intel calls it Instruction Pointer (IP)

IBM 701, 1953
(Image source: Wikipedia)

# Consequence #2: Binary Compatibility

- Programs are distributed in binary form
  - Programs bound to specific instruction set
  - Different version for phones and PCs
- New machines want to run old programs ("binaries") as well as programs compiled to new instructions
- Leads to "backward-compatible" instruction set evolving over time
- Selection of Intel 8088 in 1981 for 1st IBM PC is major reason latest PCs still use 80x86 instruction set; could still run program from 1981 PC today

# Instructions as Numbers (1/2)

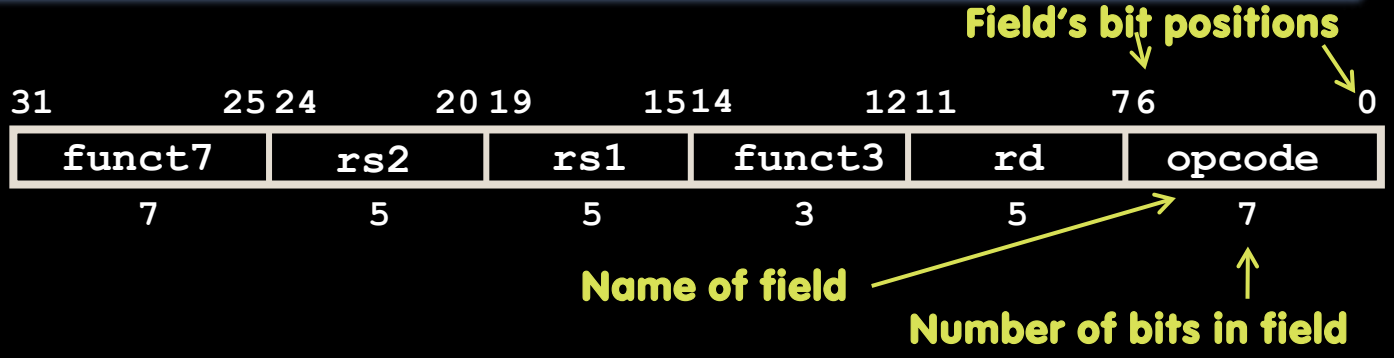
- Most data we work with is in words (32-bit chunks):
  - Each register is a word
  - `lw` and `sw` both access memory one word at a time
- So how do we represent instructions?
  - Remember: Computer only understands 1s and 0s, so assembler string "`add x10,x11,x0`" is meaningless to hardware
  - RISC-V 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

# Instructions as Numbers (2/2)

- One word is 32 bits, so divide instruction word into "fields"
- Each field tells processor something about instruction
- We could define different fields for each instruction, but RISC-V seeks simplicity, so define six basic types of instruction formats:
  - R-format for register-register arithmetic operations
  - I-format for register-immediate arithmetic operations and loads
  - S-format for stores
  - B-format for branches (minor variant of S-format)
  - U-format for 20-bit upper immediate instructions
  - J-format for jumps (minor variant of U-format)

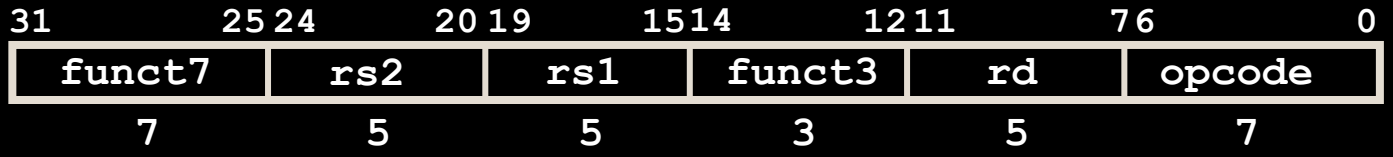
# R-Format Layout

# R-Format Instruction Layout



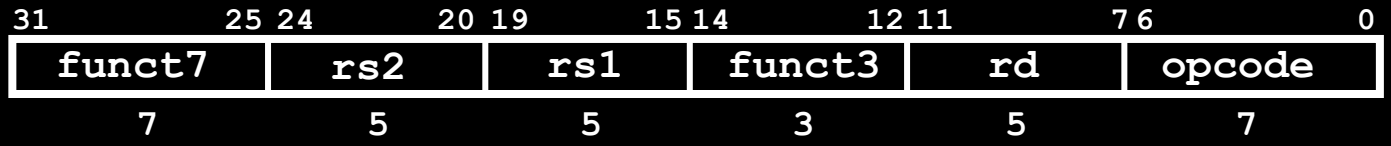

- 32-bit instruction word divided into six fields of varying numbers of bits each: 7+5+5+3+5+7 = 32
- Examples
  - `opcode` is a 7-bit field that lives in bits 6-0 of the instruction
  - `rs2` is a 5-bit field that lives in bits 24-20 of the instruction

# R-Format Instructions opcode/funct Fields




- **opcode**: partially specifies what instruction it is
  - Note: This field is equal to $0110011_{two}$ for all R-Format register-register arithmetic instructions
- **funct7+funct3**: combined with **opcode**, these two fields describe what operation to perform
- **Question: You have been professing simplicity, so why aren't opcode and funct7 and funct3 a single 17-bit field?**
  - **We'll answer this later**

# R-Format Instructions Register Specifiers

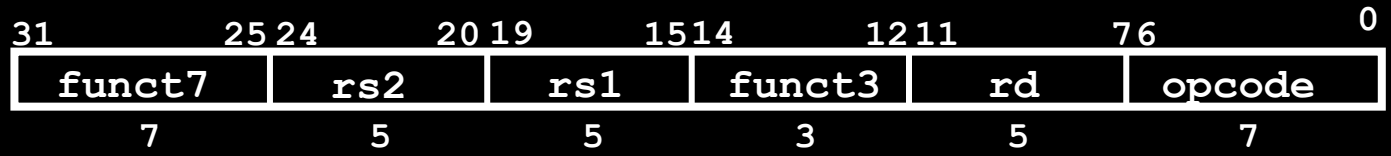


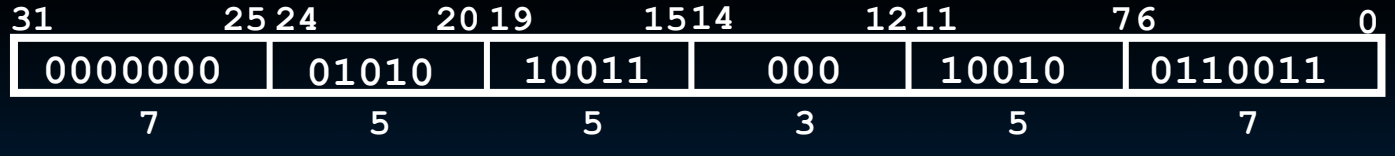

- **rs1** (Source Register #1): specifies register containing first operand
- **rs2** : specifies second register operand
- **rd** (Destination Register): specifies register which will receive result of computation
- Each register field holds a 5-bit unsigned integer (0-31) corresponding to a register number (**x0**-**x31**)

# R-Format Example

- RISC-V Assembly Instruction:

```
add   x18,x19,x10
```




add rs2=10 rs1=19 add rd=18 Reg-Reg OP

# Your Turn

- What is correct encoding of `add x4, x3, x2` ?

1) $4021\ 8233_{hex}$
2) $0021\ 82b3_{hex}$
3) $4021\ 82b3_{hex}$
4) $0021\ 8233_{hex}$
5) $0021\ 8234_{hex}$

| 31 25 | 24 20 | 19 15 | 14 12 | 11 7 | 6 0 | |
|---|---|---|---|---|---|---|
| 0000000 | rs2 | rs1 | 000 | rd | 0110011 | add |
| 0100000 | rs2 | rs1 | 000 | rd | 0110011 | sub |
| 0000000 | rs2 | rs1 | 100 | rd | 0110011 | xor |
| 0000000 | rs2 | rs1 | 110 | rd | 0110011 | or |
| 0000000 | rs2 | rs1 | 111 | rd | 0110011 | and |

# All RV32 R-format Instructions

| | | | | | | |
|---|---|---|---|---|---|---|
| 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 |

Different encoding in funct7 + funct3 selects different operations

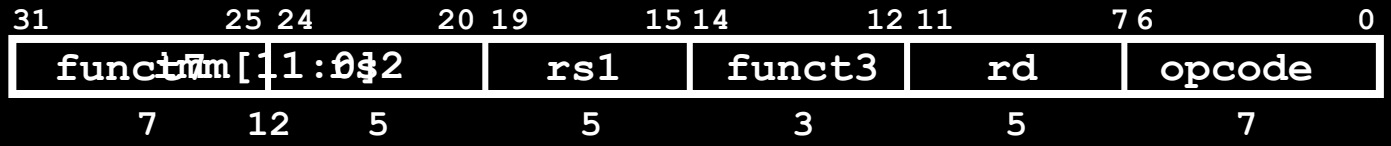
Can you spot two new instructions?

# I-Format Layout

# I-Format Instructions

- What about instructions with immediates?
  - Compare:
    - `add  rd, rs1, rs2`
    - `addi rd, rs1, imm`
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, RISC-V would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is mostly consistent with R-format
  - Notice if instruction has immediate, then uses at most 2 registers (one source, one destination)

# I-Format Instruction Layout

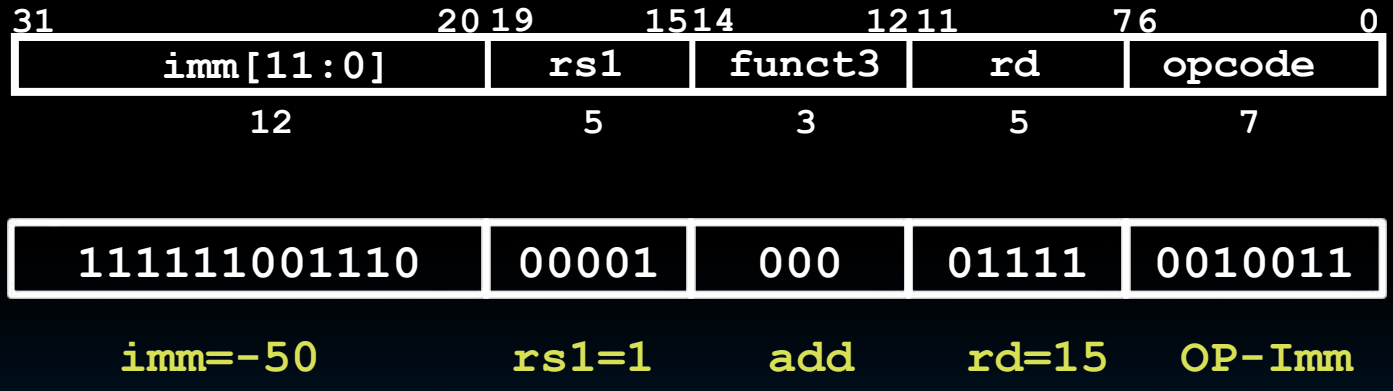



- Only one field is different from R-format, **`rs2`** and **`funct7`** replaced by 12-bit signed immediate, **`imm[11:0]`**
- Remaining fields (**`rs1`**, **`funct3`**, **`rd`**, **`opcode`**) same as before
- **`imm[11:0]`** can hold values in range [$-2048_{ten}$ , $+2047_{ten}$]
- Immediate is always sign-extended to 32-bits before use in an arithmetic operation
- We'll later see how to handle immediates > 12 bits

# I-Format Example

- RISC-V Assembly Instruction:

```
addi   x15,x1,-50
```



# All RV32 I-format Arithmetic Instructions

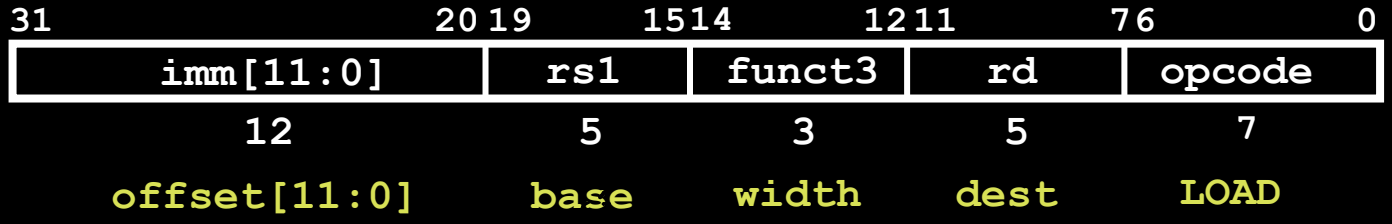
| | | | | | | |
|---|---|---|---|---|---|---|
| imm[11:0] | | rs1 | 000 | rd | 0010011 | addi |
| imm[11:0] | | rs1 | 010 | rd | 0010011 | slti |
| imm[11:0] | | rs1 | 011 | rd | 0010011 | sltiu |
| imm[11:0] | | rs1 | 100 | rd | 0010011 | xori |
| imm[11:0] | | rs1 | 110 | rd | 0010011 | ori |
| imm[11:0]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

One of the higher-order immediate bits is used to distinguish "shift right logical" (SRLI) from "shift right arithmetic" (SRAI)

"Shift-by-immediate" instructions only use lower 5 bits of the immediate value for shift amount (can only shift by 0-31 bit positions)

# RISC-V Loads

# Load Instructions are also I-Type

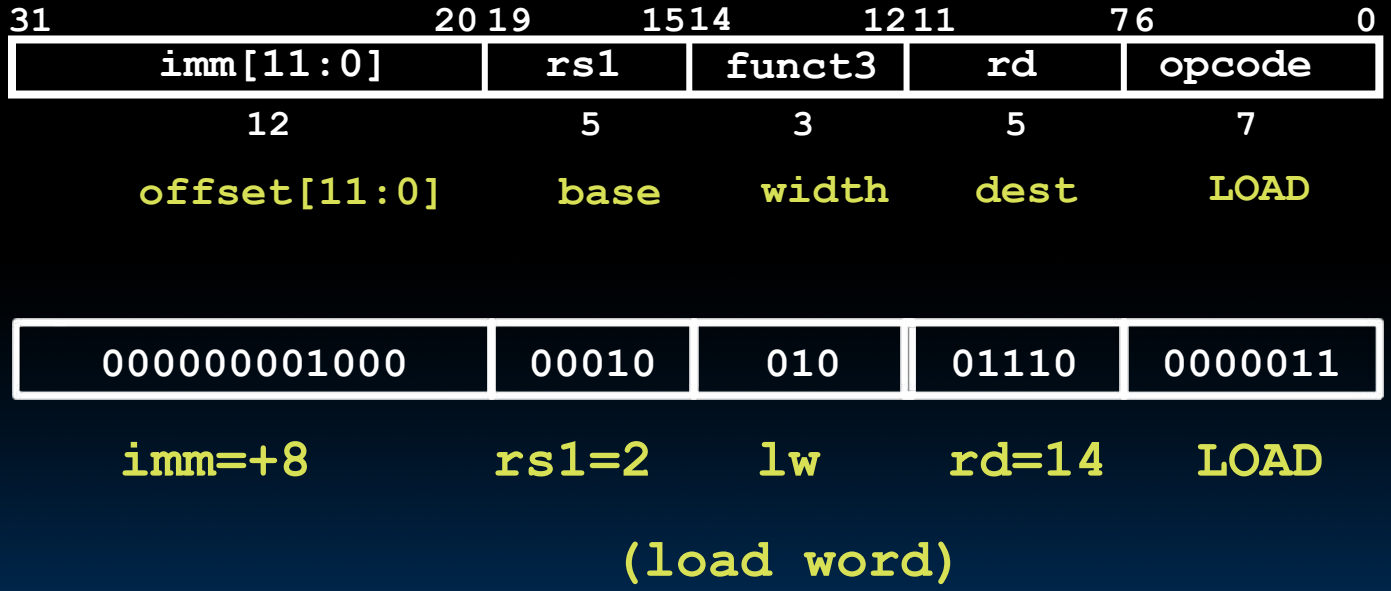



- The 12-bit signed immediate is added to the base address in register **rs1** to form the memory address
  - This is very similar to the add-immediate operation but used to create address not to create final result
- The value loaded from memory is stored in register **rd**

# I-Format Load Example

- RISC-V Assembly Instruction:

```
lw x14, 8(x2)
```



# All RV32 Load Instructions

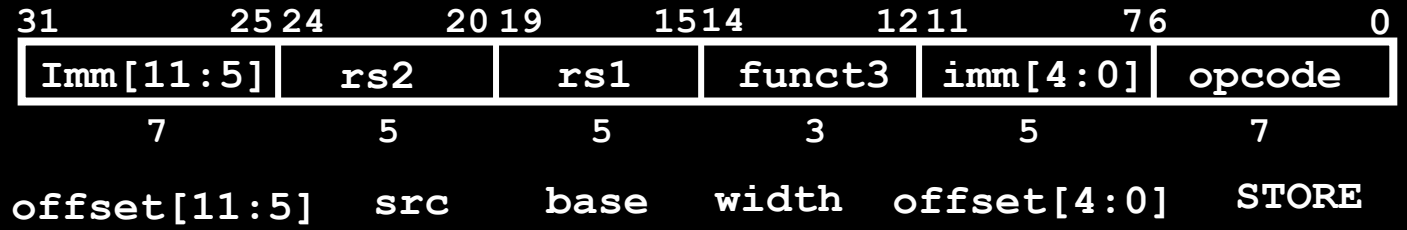
| imm[11:0] | rs1 | 000 | rd | 0000011 | lb |
|---|---|---|---|---|---|
| imm[11:0] | rs1 | 001 | rd | 0000011 | lh |
| imm[11:0] | rs1 | 010 | rd | 0000011 | lw |
| imm[11:0] | rs1 | 100 | rd | 0000011 | lbu |
| imm[11:0] | rs1 | 101 | rd | 0000011 | lhu |

funct3 field encodes size and 'signedness' of load data

- **lbu** is "load unsigned byte"
- **lh** is "load halfword", which loads 16 bits (2 bytes) and sign-extends to fill destination 32-bit register
- **lhu** is "load unsigned halfword", which zero-extends 16 bits to fill destination 32-bit register
- There is no '**lwu**' in RV32, because there is no sign/zero extension needed when copying 32 bits from a memory location into a 32- bit register

# S-Format Layout

# S-Format Used for Stores

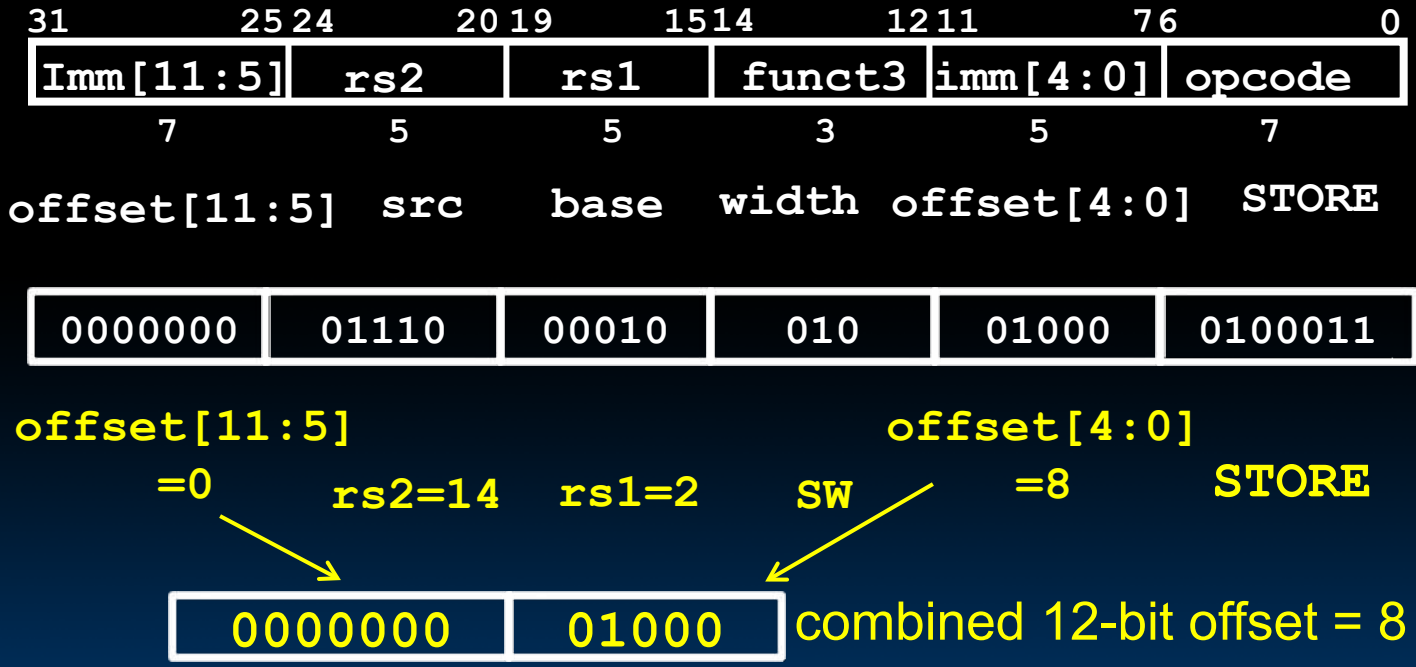



- Store needs to read two registers, **`rs1`** for base memory address, and **`rs2`** for data to be stored, as well immediate offset!
- Can't have both **`rs2`** and immediate in same place as other instructions!
- Note that stores don't write a value to the register file, ***no `rd`***!
- RISC-V design decision is to move low 5 bits of immediate to where **`rd`** field was in other instructions – keep **`rs1`**/**`rs2`** fields in same place
  - Register names more critical than immediate bits in hardware design

# S-Format Example

- RISC-V Assembly Instruction:

```
sw x14, 8(x2)
```



# All RV32 Store Instructions

- Store byte, halfword, word

| 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 |

width