

8/31/26

GRAB A WORKSHEET FROM THE FRONT OR BACK!

COMP 311:

Computer Organization!

Lecture 4: Loads + Stores, Accessing Memory!



The University
of North Carolina
at Chapel Hill

<https://tinyurl.com/comp311-fa26>

Logistics Update



- First written assignment is due tonight!
 - You will need to do a short reading/tutorial
 - It will show you how to use our **miniRISC-V** simulator 😊
- First quiz next class!
 - First ~35 minutes of class
 - Seating chart will be posted before
- Office hours posted on course webpage

Quiz Topics



- **RISC-V Basics**

- Registers, operands, and instruction syntax
- Read vs. written registers
- R-type vs. I-type instructions
- Arithmetic and immediate instructions
- Shift instructions



- **211 Review**

- Binary $\leftrightarrow$ decimal $\leftrightarrow$ hexadecimal
- Signed vs. unsigned interpretation
- Two's complement
- Basic overflow reasoning

- **Memory**

- Bytes, words, and addresses
- lw and sw
- Base + offset addressing
- Simple array accesses
- Tracing register and memory changes

- **Be able to...**

- Trace a short sequence of RISC-V instructions
- Determine register values after execution
- Reason between simple C expressions/array accesses and RISC-V



*reference
sweet



Plan for Class Today



- 3:35-3:50: R + I type Instruction Review
- 3:50-4:20: Loads and Stores
- 4:20-4:25: BREAK
- 4:25-4:50: Accessing Memory

Guiding Question 😊

How does a program access data that does not fit in registers?



UNIT 1: PROGRAMMING THE MACHINE

R-TYPE VS. I-TYPE INSTRUCTIONS

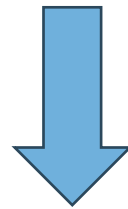
Reminder: How are programs turned into machine code?



 High-level language



Assembly



Machine Code

$c = a + b$



 add x5, x4, x3 



00000000001100100000001010110011

compiling



Translating from C to RISC-V



c = a + b ;
↓ ↓ ↓
x5 x8 x9

add x5, x8, x9
dst src

Translate this line from C to RISC-V



a = b + c + d - e;

↓ ↓ ↓ ↓ ↓

x14 x15 x16 x17 x18

Handwritten annotations: A bracket groups 'b + c + d', and another bracket groups the entire expression 'b + c + d - e'. Arrows point from each variable to its corresponding register: 'a' to x14, 'b' to x15, 'c' to x16, 'd' to x17, and 'e' to x18. A wavy line is drawn under the registers x15, x16, and x17.

You may use additional registers in your solution

add x14, x15, x16
add x14, x14, x17
sub x14, x14, x18

Translate this line from C to RISC-V



a = b + c + d - e;

↓ ↓ ↓ ↓ ↓

x14 x15 x16 x17 x18

add x14, x15, x16

add x14, x14, x17

sub x14, x14, x18

or

add x5, x15, x16

sub x6, x17, x18

add x14, x5, x6



RV32I

Integer Computation

add {immediate}

subtract

{and
or
exclusive or} {immediate}

{shift left logical
shift right arithmetic
shift right logical} {immediate}

load upper immediate

add upper immediate to pc

set less than {immediate} {unsigned}

Control transfer

branch {equal
not equal}

branch {greater than or equal
less than} {unsigned}

jump and link {register}

Loads and Stores

load {byte
halfword
word}

load {byte
halfword} unsigned

Miscellaneous instructions

fence loads & stores

fence.instruction & data

environment {break
call}

control status register {read & clear bit
read & set bit
read & write} {immediate}

be, bne

Figure 2.1: Diagram of the RV32I instructions. The underlined letters are concatenated from left to right to form RV32I instructions. The curly bracket notation { } means each vertical item in the set is a different variation of the instruction. The underscore _ within a set means that one option is simply the instruction name so far without a letter from this set. For example, the notation near the upper left-hand corner represents the following six instructions: and, or, xor, andi, ori, xori.

The miniRISC-V ISA

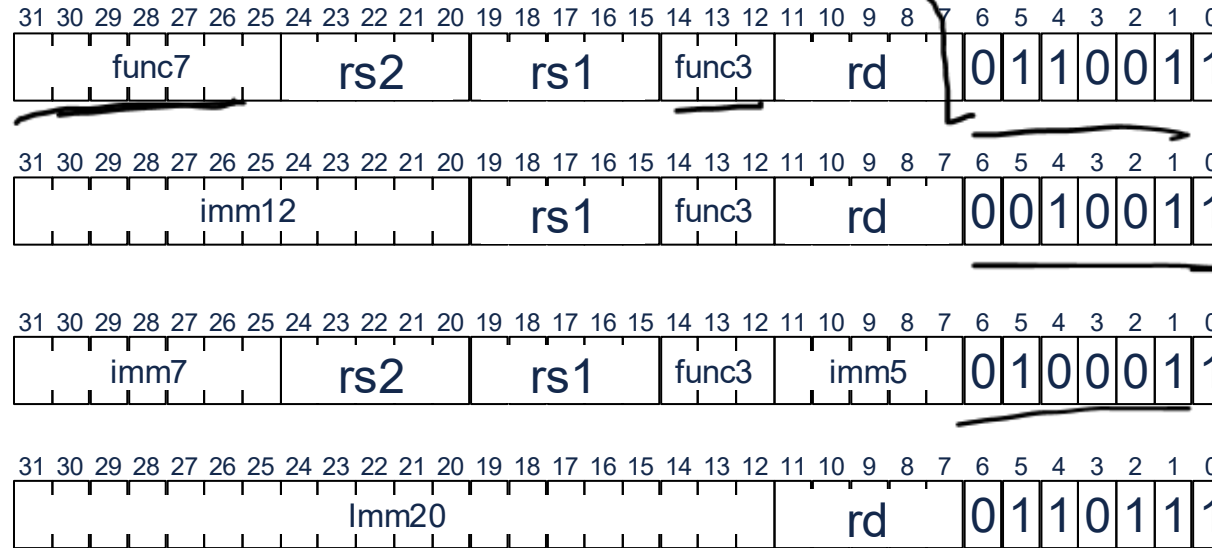


R-type:

(Arithmetic) I-type*:

B/S-type:

U-type:



Four key instruction formats (each one has a corresponding opcode!):

- 1) Arithmetic, Logical or Shift Instruction with two register operands
- 2) Arithmetic, Logical or Shift Instruction with a register and an immediate operand. (LOADS are also I-type! Diff opcode)
- 3) Stores and Branches
- 4) Jumps and large constants (LUI, AUIPC)

31	25	24	20	19	15	14	12	11	7	6	0	
imm[31:12]							rd	0110111			U lui	
imm[31:12]							rd	0010111			U auipc	
imm[20 10:1 11 19:12]							rd	1101111			J jal	
imm[11:0]				rs1	000		rd	1100111			I jalr	
imm[12 10:5]		rs2	rs1	000		imm[4:1 11]	1100011			B beq		
imm[12 10:5]		rs2	rs1	001		imm[4:1 11]	1100011			B bne		
imm[12 10:5]		rs2	rs1	100		imm[4:1 11]	1100011			B blt		
imm[12 10:5]		rs2	rs1	101		imm[4:1 11]	1100011			B bge		
imm[12 10:5]		rs2	rs1	110		imm[4:1 11]	1100011			B bltu		
imm[12 10:5]		rs2	rs1	111		imm[4:1 11]	1100011			B bgeu		
imm[11:0]				rs1	000		rd	0000011			I lb	
imm[11:0]				rs1	001		rd	0000011			I lh	
imm[11:0]				rs1	010		rd	0000011			I lw	
imm[11:0]				rs1	100		rd	0000011			I lbu	
imm[11:0]				rs1	101		rd	0000011			I lhu	
imm[11:5]		rs2	rs1	000		imm[4:0]	0100011			S sb		
imm[11:5]		rs2	rs1	001		imm[4:0]	0100011			S sh		
imm[11:5]		rs2	rs1	010		imm[4:0]	0100011			S sw		
imm[11:0]				rs1	000		rd	0010011			I addi	
imm[11:0]				rs1	010		rd	0010011			I slti	
imm[11:0]				rs1	011		rd	0010011			I sltiu	
imm[11:0]				rs1	100		rd	0010011			I xori	
imm[11:0]				rs1	110		rd	0010011			I ori	
imm[11:0]				rs1	111		rd	0010011			I andi	
0000000		shamt	rs1	001		rd	0010011			I slli		
0000000		shamt	rs1	101		rd	0010011			I srli		
0100000		shamt	rs1	101		rd	0010011			I srai		
0000000		rs2	rs1	000		rd	0110011			R add		
0100000		rs2	rs1	000		rd	0110011			R sub		
0000000		rs2	rs1	001		rd	0110011			R sll		
0000000		rs2	rs1	010		rd	0110011			R slt		
0000000		rs2	rs1	011		rd	0110011			R sltu		
0000000		rs2	rs1	100		rd	0110011			R xor		
0000000		rs2	rs1	101		rd	0110011			R srl		
0100000		rs2	rs1	101		rd	0110011			R sra		
0000000		rs2	rs1	110		rd	0110011			R or		
0000000		rs2	rs1	111		rd	0110011			R and		
0000	pred	succ	00000	000	00000	0001111	I fence					
0000	0000	0000	00000	001	00000	0001111	I fence.i					
000000000000			00000	000	00000	1110011	I ecall					
000000000001			00000	000	00000	1110011	I ebreak					
csr			rs1	001		rd	1110011			I csrrw		
csr			rs1	010		rd	1110011			I csrrs		
csr			rs1	011		rd	1110011			I csrrc		
csr			zimm	101		rd	1110011			I csrrwi		
csr			zimm	110		rd	1110011			I csrrsi		
csr			zimm	111		rd	1110011			I csrrci		

Figure 2.3: RV32I opcode map has instruction layout, opcodes, format type, and names. (Table 19.2 of [Waterman and Asanović 2017] is the basis of this figure.)

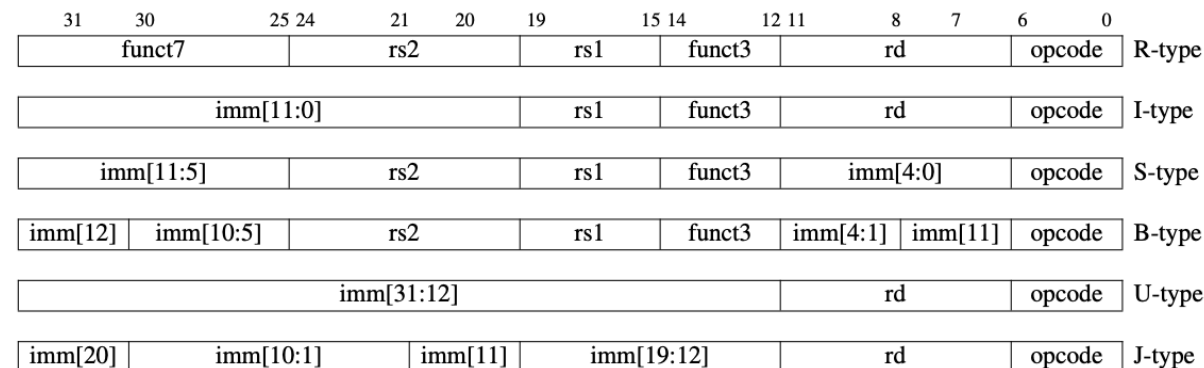


Figure 2.2: RISC-V instruction formats. We label each immediate subfield with the bit position (imm[x]) in the immediate value being produced, rather than the bit position in the instruction's immediate field as is usually done. Chapter 10 explains how the control status register instructions use the I-type format slightly differently. (Figure 2.2 of [Waterman and Asanović 2017] is the basis of this figure).

R-Format Operations



- add
- sub
- AND
- OR
- SLT (set on less than)
- SLTU (Unsigned)
- SLL (shift left logical)
- SRL (shift right logical)
- SRA (shift right arithmetic)

Op/rd.

0110011

funct7/funct3
↳ choose the
op

R-format vs I-format Instructions



- R-format
 - Used for operations between two registers
- I-format
 - Used for operations between a register and an immediate value
 - in other words, if you want to use a constant that is not in a register

addi

R-Format vs I-Format Syntax



R-Format

op rd, rs1, rs2

$R[rd] = R[rs1] \text{ op } R[rs2]$

writes to rd

I-Format

op rd, rs1, immediate

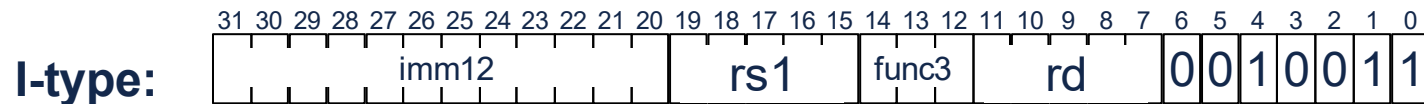
$R[rd] = R[rs1] \text{ op } \text{immediate}$

writes to rd

I-type Data Processing



Instructions that process one register and a constant:



Notice there is no SUBI instruction. Why?



000 - ADDI
001 - SLLI
010 - SLTI
011 - SLTUI
100 - XORI
101 - SRLI/SRAI
110 - ORI
111 - ANDI

andi **x5**, **x10**, 255

I-type instructions
have the following template:

OPfunc3
rd, rs1, imm12

Is encoded as:

0000	1111	1111	0101	0111	0010	1001	0011
------	------	------	------	------	------	------	------

0x0ff57293

Last time...



0x003202b3

Moving to a higher-level of
abstraction!



Disassembly

add x5, x4, x3

Last time...



0x003202b3

Moving to a higher-level of
abstraction!



Disassembly

add x5, x4, x3

addi x5, x0, x12

Moving to a lower-level of
abstraction!



Assembly

0b0000 0000 1100 0000
0000 0010 1001 0011

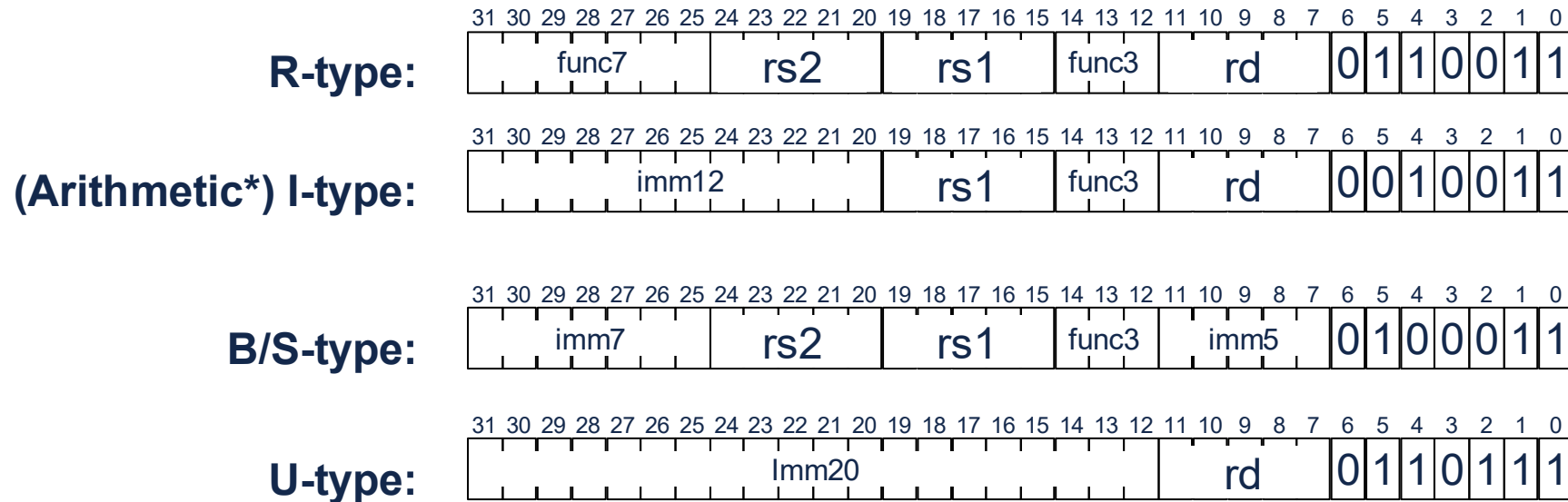
We used our 32-bit instruction
templates to help us 😊



UNIT 1: PROGRAMMING THE MACHINE

Load and Store Instructions!

The miniRISC-V ISA



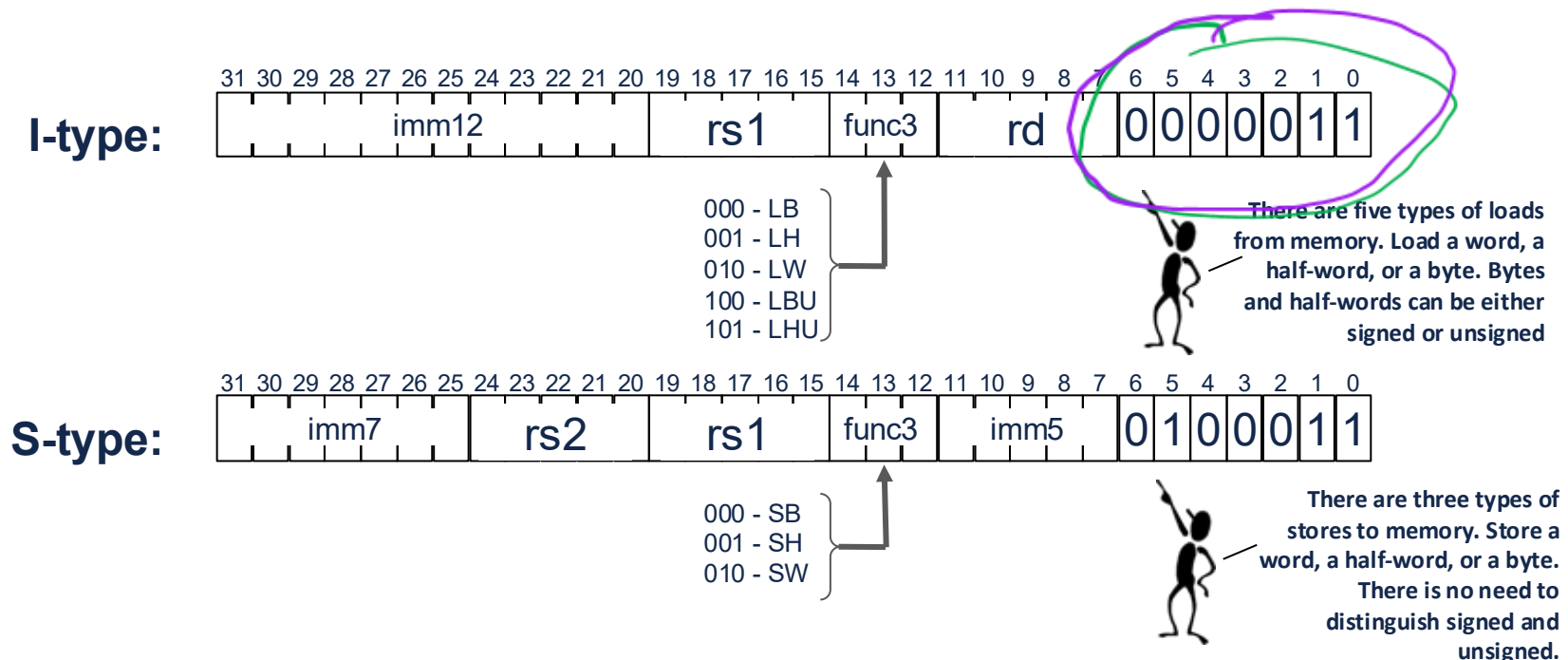
Four key instruction formats (each one has a corresponding opcode!):

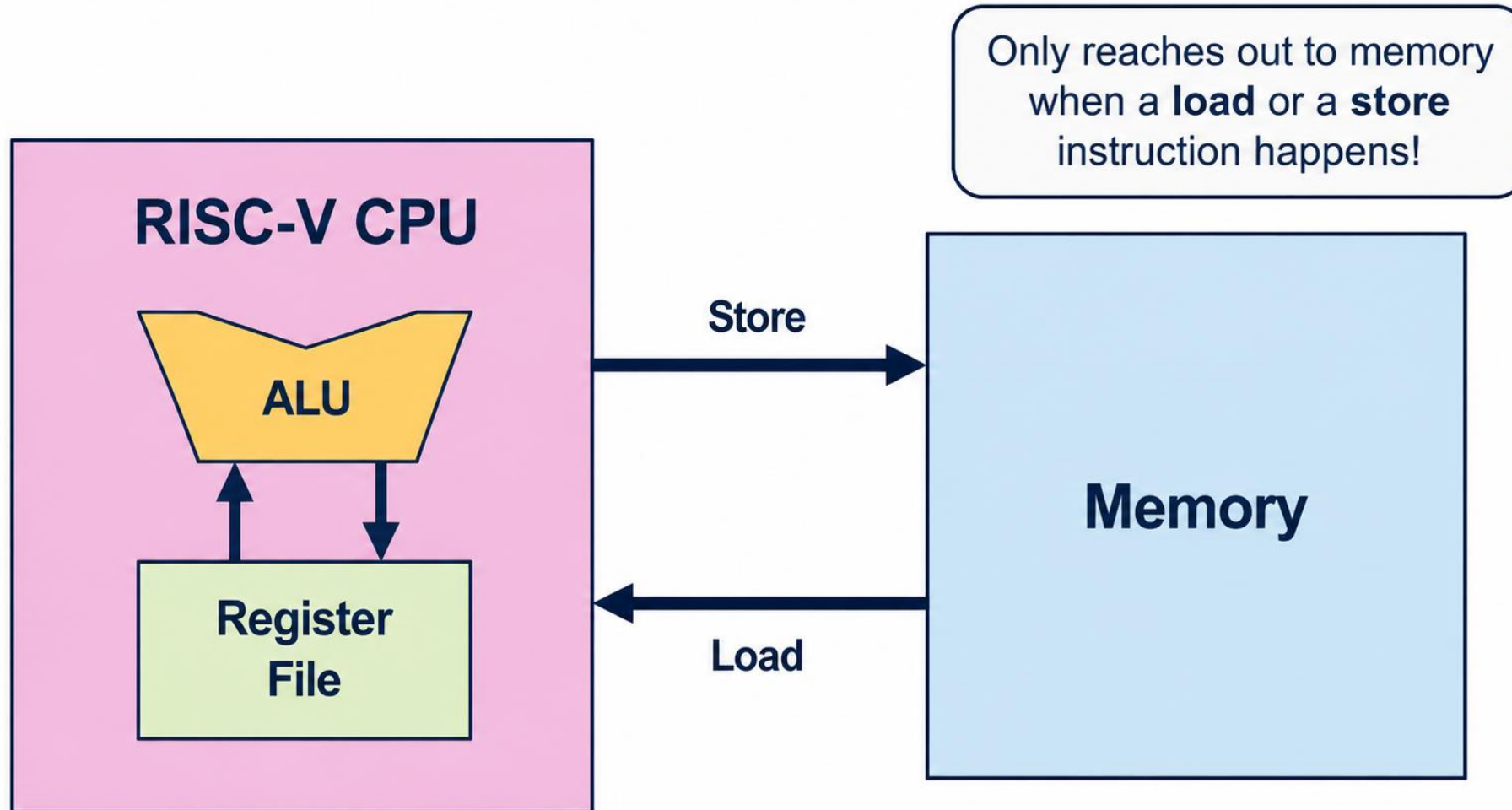
- 1) ALU with two register operands
- 2) ALU with a register and an immediate operand. **(LOADS HAVE DIFF OPCODE!)**
- 3) Stores and Branches
- 4) Jumps and large constants (LUI, AUIPC)

Load and Store Instructions



RISC-V is a “Load/Store architecture”. That means that only a specific class of instructions are used to reference data in memory. As a rule, data is loaded into registers first, then processed, and the results are written back using stores. Load and Store instructions have their own format:





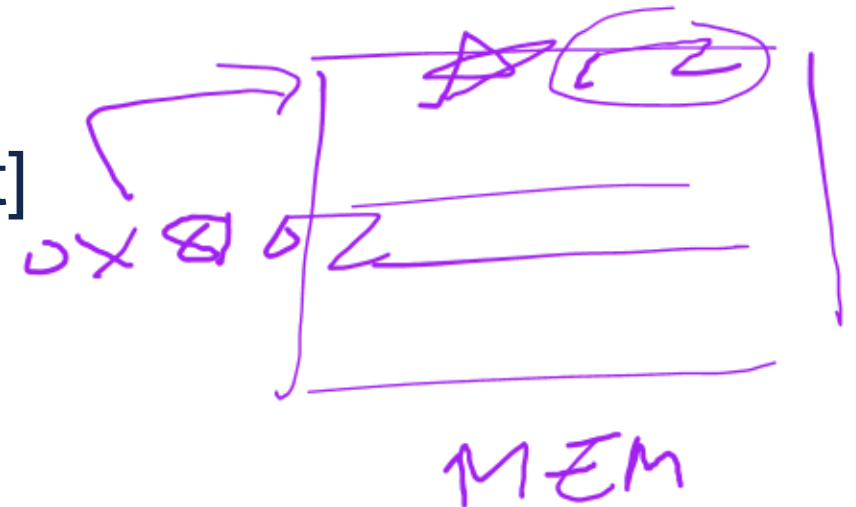
Load Word (lw)



- Used to read 4 bytes (one word) of data from memory
- `lw rd, offset(rs1)`
 - `rs1` is a register that holds a memory address
 - `offset` is a 12 bit immediate value
 - The loaded word will be stored in `rd`

$$R[rd] = M[R[rs1] + offset]$$

`lw x5, 0x8(x3)`
addr



Store Word (sw)



- Used to store 4 bytes (one word) of data to memory
- `sw rs2 offset(rs1)`
 - `rs1` is a register that holds a memory address
 - `offset` is a signed 12-bit immediate value
 - `rs2` is the value that will be written at this address

$$M[R[rs1] + offset] = R[rs2]$$



Load Word vs Store Word

- Load Word
 - Reads 4 bytes (one word) from memory
- Store word
 - Writes 4 bytes (one word) to memory
- Both:
 - Require a base address coming from a reg & small offset encoded in the instruction

Load and Store Options



RISC-V's load and store instructions are versatile. They provide a wide range of addressing modes. Only a subset is shown here.

base

lw rd, imm12(rs)



$Rd \leftarrow \text{Memory}[Rs + \text{imm12}]$

Rd is loaded with the contents of memory at the address found by adding the contents of the base register, Rs, to the supplied constant

lb rd, -4(rs)



$Rd \leftarrow \text{sign_extend}(\text{Memory}[Rs - 4])$

Offsets can be either added or subtracted, as indicated by a negative sign. The byte is signed-extend to fill the 32 bits of

lw rd, (rs)



If no offset is specified it is assumed to be zero

sw rd, 12(rs)



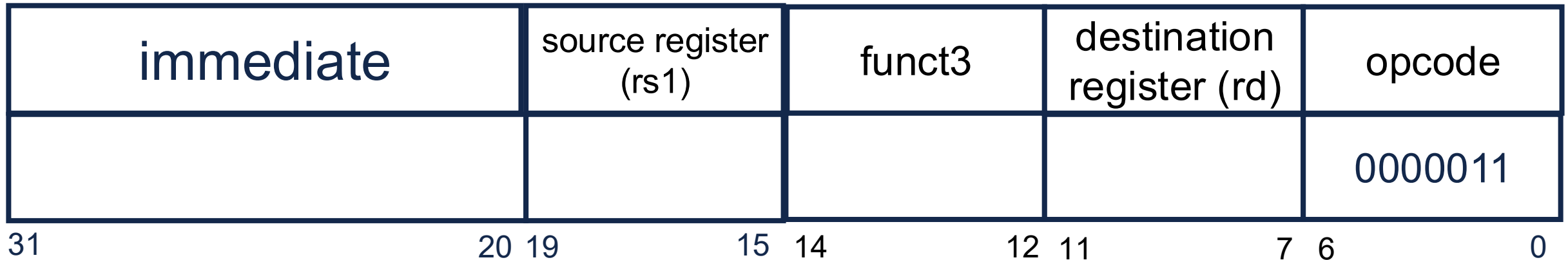
The contents of a register hold the address of the either the data value or a base address for a composite type (structure, object, array, or a stack)

sh rd, (rs)

Load Word Encoding



I-Type, *but slightly different opcode!*



load

1. Compute address
2. Load data from memory
3. Store data into register file

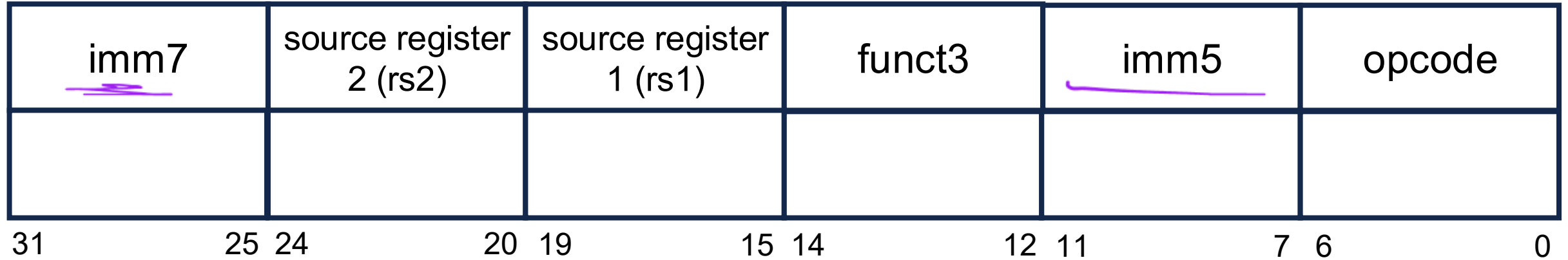
lw rd, imm12(rs1)

$$R[rd] \leftarrow M[R[rs1] + \text{SignExt}(\text{imm12})]$$

Store Word Encoding



S-type



sw rs2, imm(rs1)

$$M[R[rs1] + \text{SignExt}(\text{Imm})] = R[rs2]$$

opcode = 0b0100011

Translate the following instructions into machine code



lw x5, 0x20(x7)

I-type



immediate	source register (rs1)	funct3	destination register (rd)	opcode
000000100000	00111	010	00101	000111
31	20 19	15 14	12 11	7 6

S-type

sw x20, -12(x9)

1100 → 0111

imm7	source register 2 (rs2)	source register 1 (rs1)	funct3	imm5	opcode
1111111	10100	01001	010	10100	0100011
31	25 24	20 19	15 14	12 11	7 6

data base address

Translate the following instructions into machine code



lw x5, 0x20(x7)

0x0203A283

immediate							source register (rs1)			funct3		destination register (rd)				opcode	
000000100000							00111			010		00101				0000011	
31	20						19	15		14	12		11	7		6	0

sw x20, -12(x9)

0xFF44AA23

imm7			source register 2 (rs2)			source register 1 (rs1)			funct3		imm5				opcode	
1111111			10100			01001			010		10100				0100011	
31	25		24	20		19	15		14	12		11	7		6	0



UNIT 1: PROGRAMMING THE MACHINE

Accessing Memory

Memory



- The register file only contains 32, 32-bit registers
 - There be other registers in the hardware, i.e., pipeline registers, but the programmer doesn't have access to these!
- We can't store all of our data in the register file. So where does it all go?
- ***In the main memory***

RISC-V Memory Model



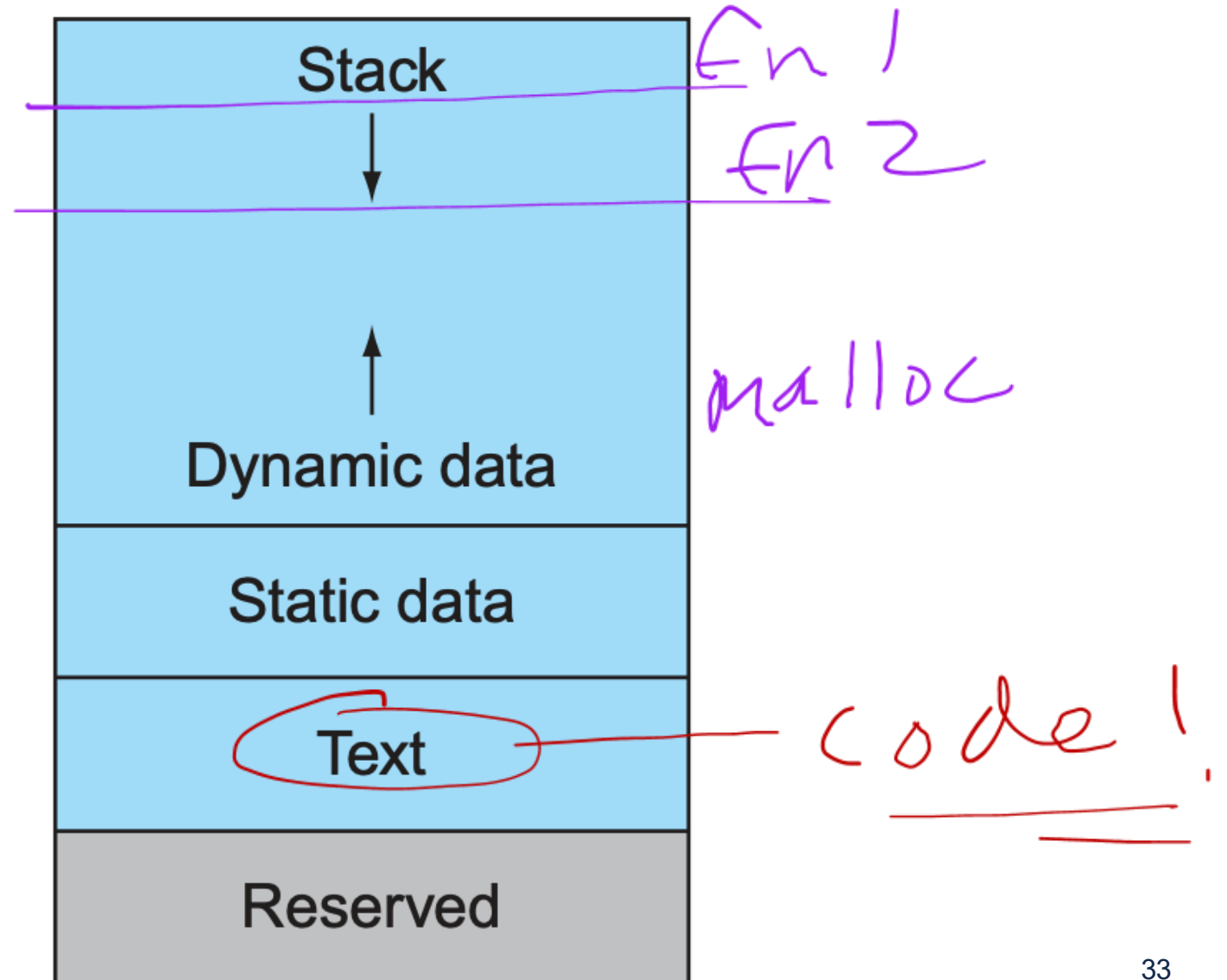
Used to manage function calls and local variables

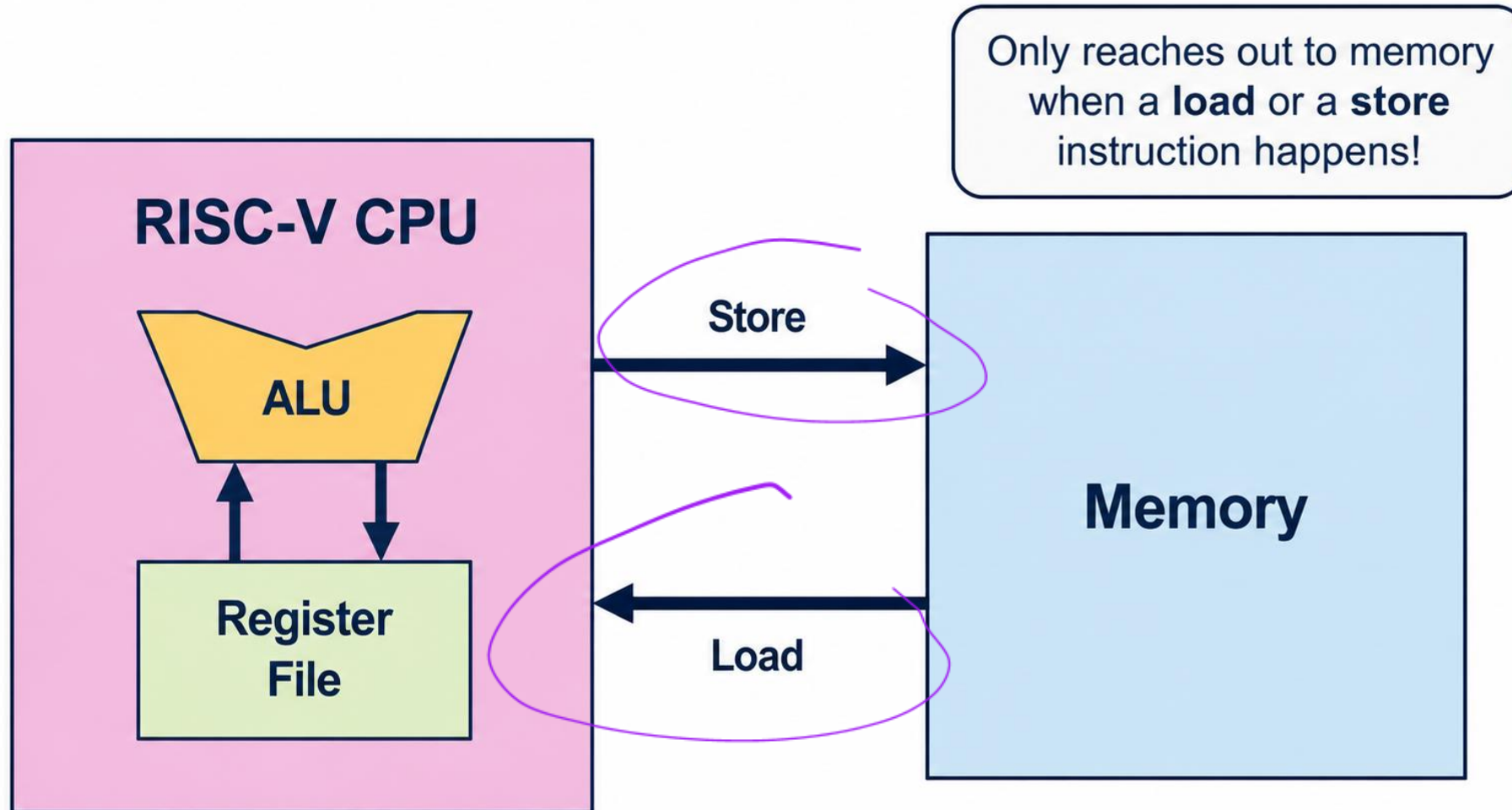
Used for dynamic memory allocation

Variables allocated at compile-time

Programs

Reserved for the OS







Memory vs. Register File

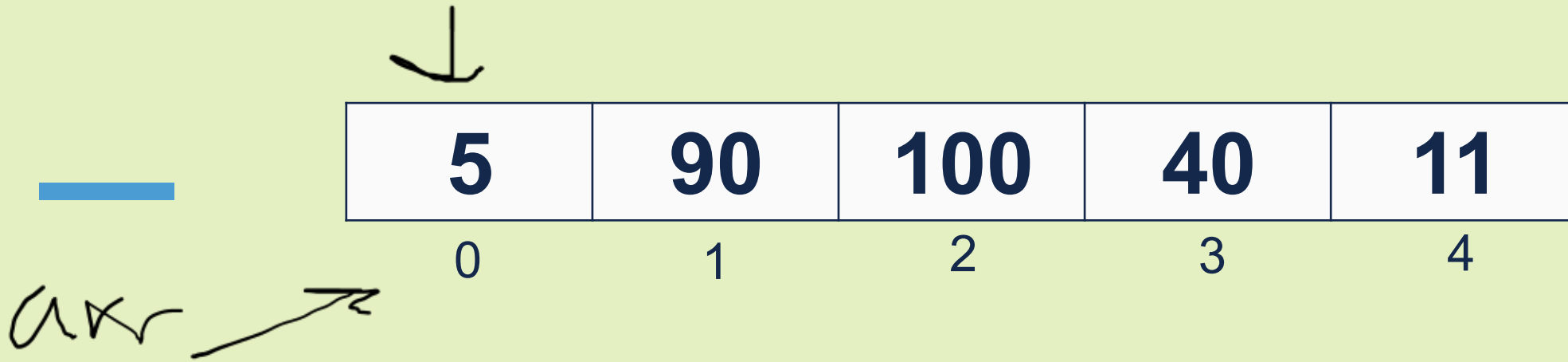
- So what's the point of the register file?
- It is MUCH, much..! faster to access the register file
- We'll store the values that we are currently working with in the register file

Memory

1. large
2. takes longer to access
3. not part of the CPU

Register File

1. small
2. quick access
3. part of the CPU



- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x11 after executing the following instruction?

```
lw x11, 0(x10)
```



5	90	100	40	11
0	1	2	3	4

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x11 after executing the following instruction?

```
lw x11 0(x10)
```

5



5	90	100	40	11
---	----	-----	----	----

0

1

2

3

4

avr

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x12 after executing the following instruction?

`lw x12, 4(x10)`



5	90	100	40	11
0	1	2	3	4

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x12 after executing the following instruction?

```
lw x12 4(x10)
```

90

0x4000 0x4004 0x4008 0x400C 0x4010



5	90	100	40	11
---	----	-----	----	----

0

1

2

3

4

avr 2 →

x10 ↑

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000 ~~4~~
- What is the value of x12 after executing the following instructions?

→ addi x10, x10, 4

→ lw x12, 8(x10)



40



5	90	100	40	11
0	1	2	3	4

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of x12 after executing the following instructions?

```
addi x10 x10 4  
lw x12 8(x10)
```

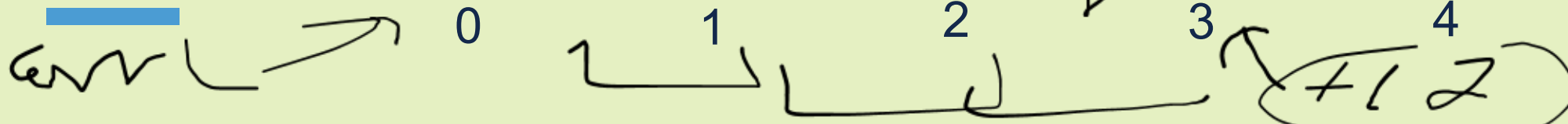
40

0x4000

8



5	90	100	40	11
---	----	-----	---------------	----



- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume $x10 = 0x00004000$ and $x7 = 8$.
- What are the contents of the array after executing the following instruction?

→ `sw(x7, 12(x10))`



5	90	100	40	11
0	1	2	3	4

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000 and x7 = 8.
- What are the contents of the array after executing the following instruction?

`sw x7, 12(x10)`

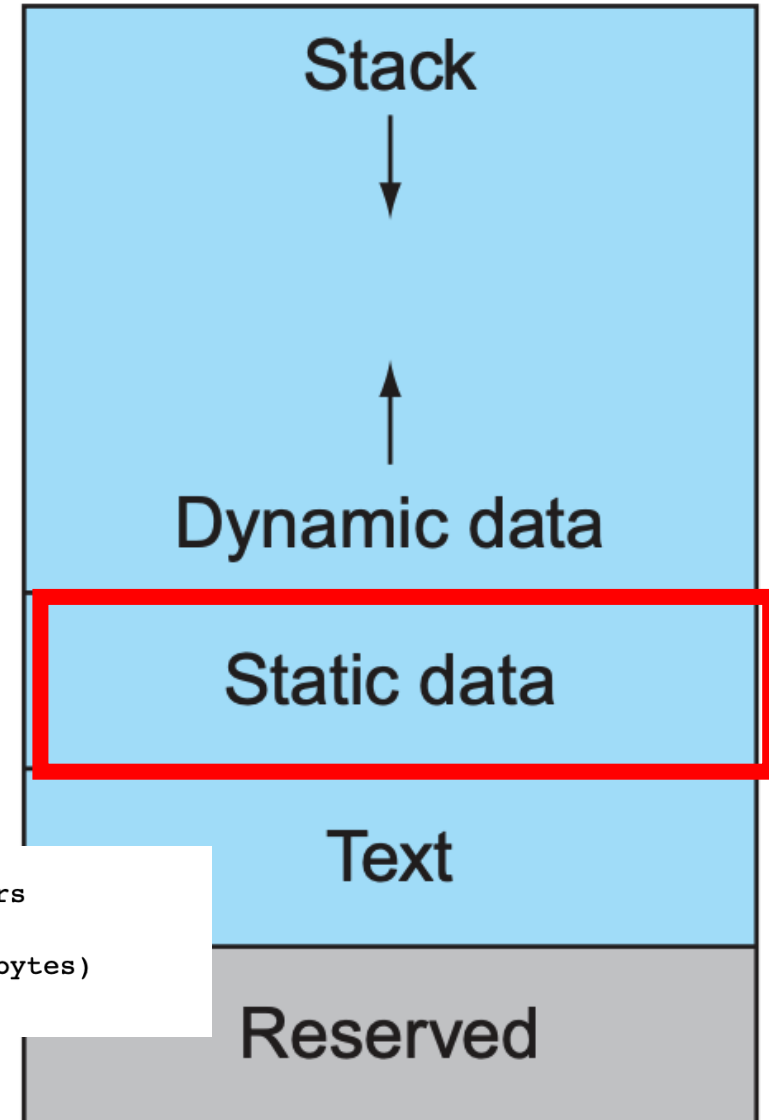
5	90	100	8	11
0	1	2	3	4

Storing Arrays in Memory



- We will store our statically allocated arrays in the static data segment
- To store data in the static data segment, we use an assembler directive.
 - The miniRISCV assembler and simulator provides a set of directives for allocating space for data and initializing variables

```
Fib: .word 1,1,2,3,5,8,13,21           # first 8 Fibonacci numbers
masks: .word 0x000000ff,0x0000ff00,0x00ff0000,0xff000000 # byte masks
Name: .string "Leonard"              # 0-terminated string (8 bytes)
array: .space 20                      # 20 uninitialized words
```



Load and Stores in action

Array access in C → in RISC-V:
compute addr, load value, then
use it!



An example of how loads and stores are used to access arrays. What C code produced this?

Hints:

- sizeof(int) in C is 4 bytes.
- What does slli x6, x5, 2 do to the value in x5?
Why might that be useful before a load?
- Instructions that help us with the loop pattern?
- Which instructions access memory?
- What does the add x7, x7, x6 suggest is happening repeatedly?
- Use the simulator! Use the docs!

Assembly:

```
                addi    x31, x0, 10
                addi    x5, x0, 0
loop:           slli    x6, x5, 2
                lw      x6, values(x6)
                lw      x7, sum(x0)
                add     x7, x7, x6
                sw      x7, sum(x0)
                addi    x5, x5, 1
                blt     x5, x31, loop
*halt:         jal     x0, halt
sum:           .word   0
values:        .word   1, 3, 5, 7, 9, 11, 13, 15, 17, 19
```