RUHR-UNIVERSITÄT BOCHUM

RISC-V ASSEMBLY

Computer Architecture

Agenda

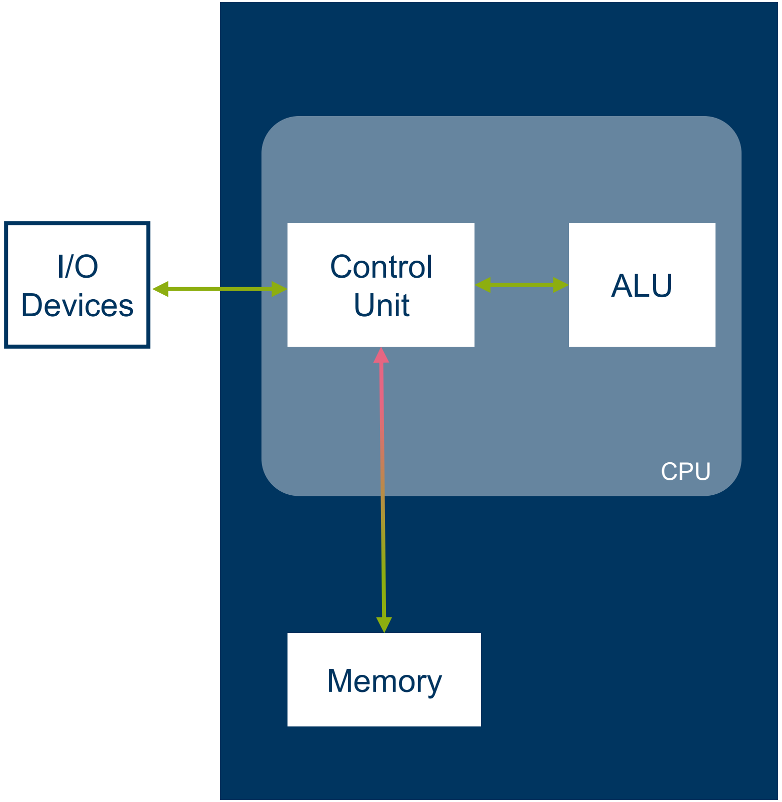
- Registers
- Ripes
- RISC-V Assembly
  - Arithmetic instructions
  - Bit operations
  - Control flow
- Machine code

Recap: the Memory Wall

Von Neumann architecture uses a single bus to access memory – serializes all accesses

Example: Computing a=a+b takes a minimum of four cycles

- 1× Load instruction
- 2× Load data
- 1× Store result



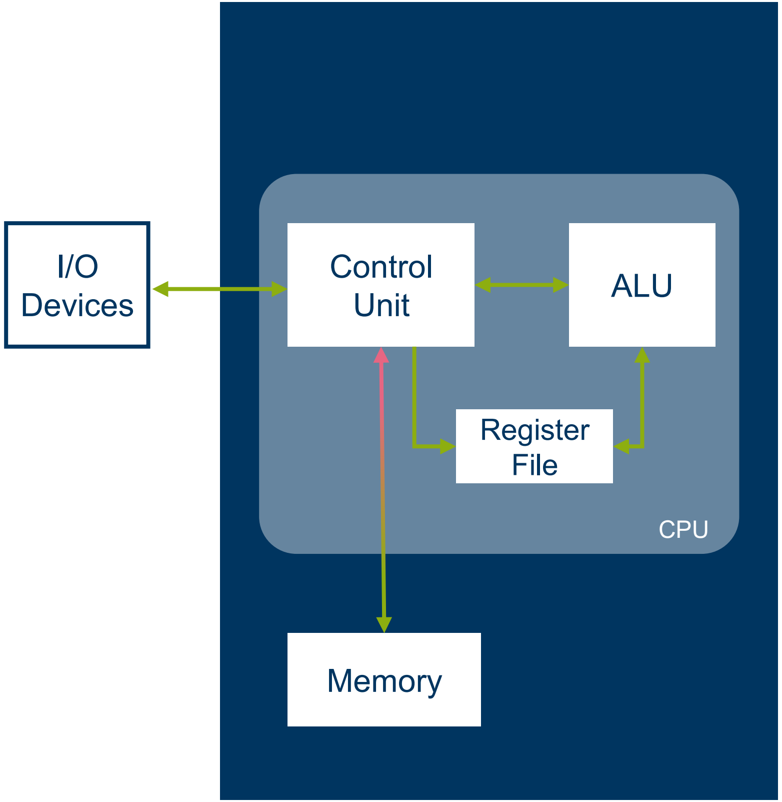
Solution: Registers

Small data elements stored within the CPU

Instructions can load and store data without accessing memory.

If a is in a register, computing a=a+b requires only two accesses to memory

With b also in a register, computing a=a+b only requires a single memory access.



RISC-V Registers

The RISC-V architecture supports 32 registers named x0-x31

Each register stores a 32-bit number

Register x0 is special – always has the value 0x00000000

Register aliases give more descriptive names to registers

We will mostly use t0-t6 and zero

Name	Alias	Description
x0	zero	hard-wired zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global pointer
x4	tp	thread pointer
x5 - x7	t0 - t2	temporaries
x8 - x9	s0 - s1	saved registers
x10 - x17	a0 - a7	function arguments
x18 - x27	s2 - s11	saved registers
x28 - x31	t3 - t6	temporaries

RISC-V

Modular open-source **I**nstruction **S**et **A**rchitecture

Basic ISA: RV32I

- Instructions are 32-bit long
- Integer instructions are included

Possible Extensions:

- M: Integer multiplication and division
- F/D: Single/Double Precision floating point operations
- C: set of compressed 16-bit instructions is added

What do instructions look like?

```
addi rd, rs1, imm
```

Mnemonic:
Name of the instruction

Destination register:
Where the result will be saved

Source register 1:
The first operand is in this register

Immediate:
The second operand is this number

The `addi` instruction

Placeholders

```
addi rd, rs1, imm
```

```
addi t0, t1, 522
```

The `addi` instruction
- adds the value `imm`
- to the contents of register `rs1` and
- stores the result in register `rd`

`imm` is 12-bit long ($-2048 \leq$ `imm` ≤ 2047)

RUHR UNIVERSITÄT BOCHUM RUB

Introduction to Ripes

Arithmetic instructions

The `add` instruction

```
add rd, rs1, rs2
```

- adds the contents of register `rs1`
- to the contents of register `rs2` and
- stores the result in register `rd`

What does the instruction `add t0, zero, t1` do?

What does the instruction `add t0, t0, t0` do?

A short program

Write a program that uses the `add` instruction to multiply `t0` by 9, with the result in register `t1`. Use less than 8 `add` instructions.

A more complex program

Write a program that uses the `add` and `addi` instructions to set `t0` to 1000000.

Hint: 1000000 = 0xf4240

RUHR UNIVERSITÄT BOCHUM

The `lui` instruction

```
lui rd, imm
```

Load Upper Immediate –

$$\texttt{rd} \leftarrow \texttt{imm} \times 2^{12}$$

`imm` is 20-bit long.

Typical use:

```
lui t0, 0xf4
addi t0, t0, 0x240
```

Pseudo instructions

Short form for common patterns

Pseudo instruction	Equivalent instruction
`nop`	`addi x0, x0, 0`
`mv rd, rs`	`addi rd, rs, 0`
`li rd, imm`	`lui rd, imm1` `addi rd, rd, imm2`

The `sub` instruction

```
sub rd, rs1, rs2
```

Why is there no `subi` instruction?

Multiplication

What is the range for the product of two 32-bit numbers?

Unsigned: $0 \leq a, b \leq 2^{32} - 1$

Hence: $0 \leq a \times b \leq 2^{64} - 2^{33} + 1$

Signed: $-2^{31} \leq a, b \leq 2^{31} - 1$

Hence: $-2^{62} + 2^{31} \leq a \times b \leq 2^{62}$

Problem: Either way, the result does not fit in 32 bits.

Solution: multiply in half

The `mul` instruction

```
mul rd, rs1, rs2
```

Computes the 32 LSBs of the product of `rs1` and `rs2`

Same result for signed and unsigned numbers

Suppose a is a negative number.
When treated as unsigned, we get the value $2^{32} + a$
Hence when multiplying as unsigned by b, we get $2^{32}b + ab$,
which differs from ab by a multiple of 2^{32}

Multiplication of the Upper Half

Instruction	rs1	rs2
`mulh`	signed	signed
`mulhsu`	signed	unsigned
`mulhu`	unsigned	unsigned

Division

```
div rd, rs1, rs2
divu rd, rs1, rs2
```

Divides `rs1` by `rs2` (signed and unsigned)

Rounding towards 0

What does the following program produce in t0?

```
li t0, -5
li t1, 3
div t0, t0, t1
```

Remainder

```
rem rd, rs1, rs2
remu rd, rs1, rs2
```

What's left after dividing an integer by another integer

27 % 5 = 2
39 % 4 = 3
20 % 2 = 0

What does the following program produce in t0?

```
li t0, -5
li t1, 3
rem t0, t0, t1
```

Bit operations

Logic operations

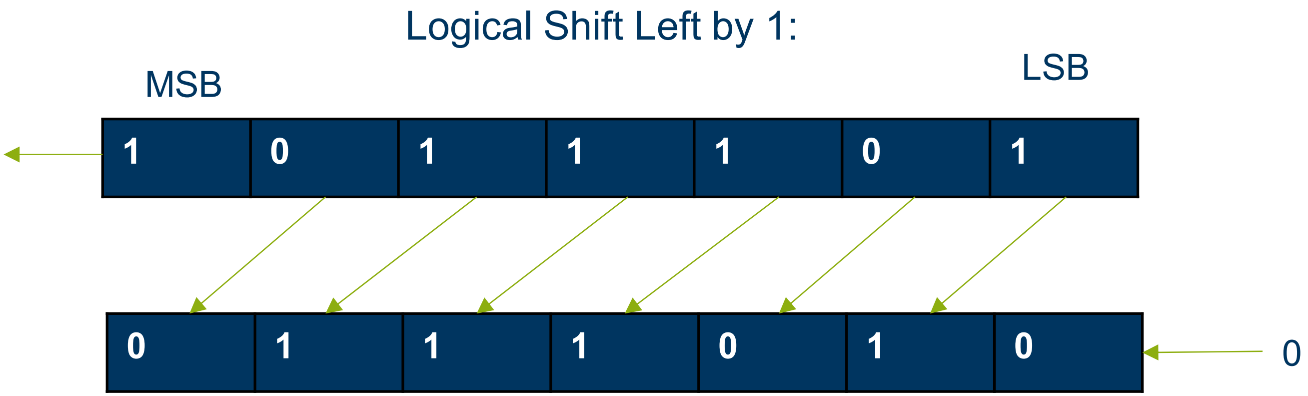
```
and rd, rs1, rs2
or rd, rs1, rs2
xor rd, rs1, rs2
```

Compute the bitwise and, or, and xor operations.

The immediate variants `andi`, `ori`, `xori` are also supported

What about not?
Pseudo instruction – same as `xori` with -1

Shift operations



Each bit is moved to the left by 1 position and the free space is filled with a zero

Logical Shifts

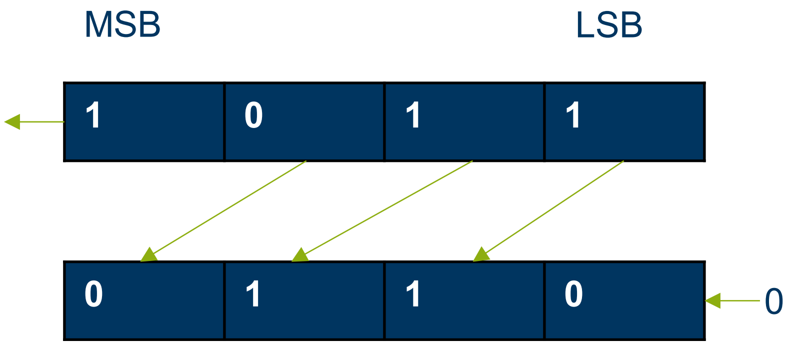
Shift Left:

Push all bits to the left by a given number from the register/immediate and fill all free spaces with zeroes

Same as multiplication by 2^i

Assembly instructions:

- **S**hift **L**eft (**L**ogical)
- **S**hift **L**eft (**L**ogical) **I**mmediate



Logical Shifts

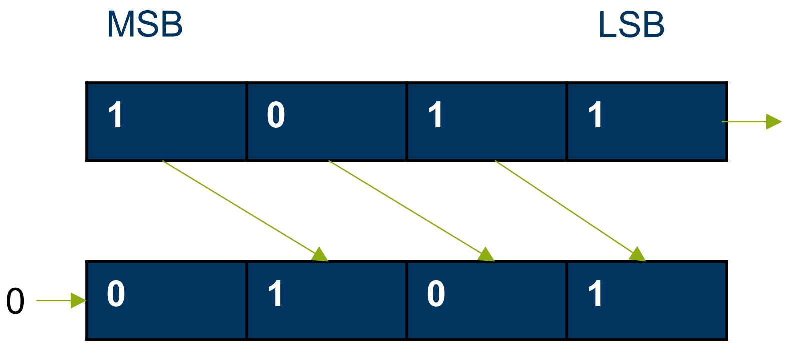
Shift Right:

Push all bits to the right by a given number from the register/immediate and fill the free space with zeroes

Same as unsigned division by 2^i

Assembly instructions:

- **S**hift **R**ight (**L**ogical)
- **S**hift **R**ight (**L**ogical) **I**mmediate



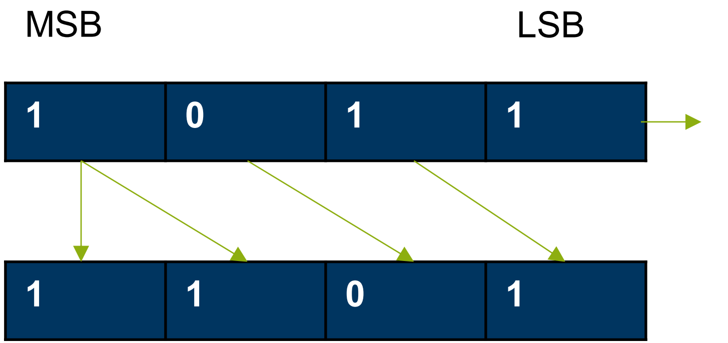
Arithmetic Shift Right

Signed division by 2^i

Push all bits to the right by a given number from the register/immediate and fill the MSB with the sign

Assembly instructions:

- **S**hift **R**ight **A**rithmetic
- **S**hift **R**ight **A**rithmetic **I**mmediate



RUHR UNIVERSITÄT BOCHUM RUB

Questions:

What does `slli t0, t0, 5` do?

What instruction divides `t0` by 8?

```
srai t0, t0, 3
```

More questions:

What does `srai t0, t0, 31` do?

The function ReLU(), which is central to deep learning is defined as

$$ReLU(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$$

Write a program that computes ReLU(`t0`). Use only instructions we've learnt in this class.

RUHR UNIVERSITÄT BOCHUM RUB

Control flow

Jump (unconditional)

`j imm`

Pseudo instruction. Sets PC to PC+`imm`

Address calculations are complex – use labels.

```
  li t0, 10
  j end
  addi t0, t0, 5
end:
  nop
```


RUHR UNIVERSITÄT BOCHUM RUB

Branches (conditional)

`beq rs1, rs2, imm`	(Branch Equal)
`bne rs1, rs2, imm`	(Branch Not Equal)
`blt rs1, rs2, imm`	(Branch Less Then)
`bge rs1, rs2, imm`	(Branch Greater or Equal)
`bltu rs1, rs2, imm`	(Branch LT Unsigned)
`bgeu rs1, rs2, imm`	(Branch GE Unsigned)

Conditionally sets PC to PC+`imm`

For example, `bne` branches if `rs1` ≠ `rs2`

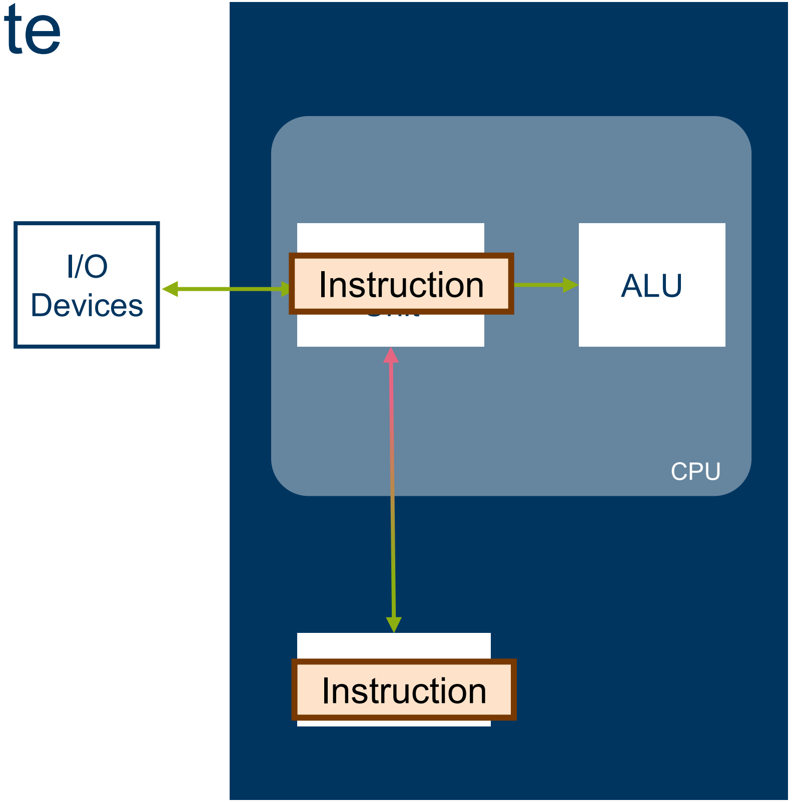
Machine Code

Recap: Fetch-Decode-Execute

Program execution follows three main steps:

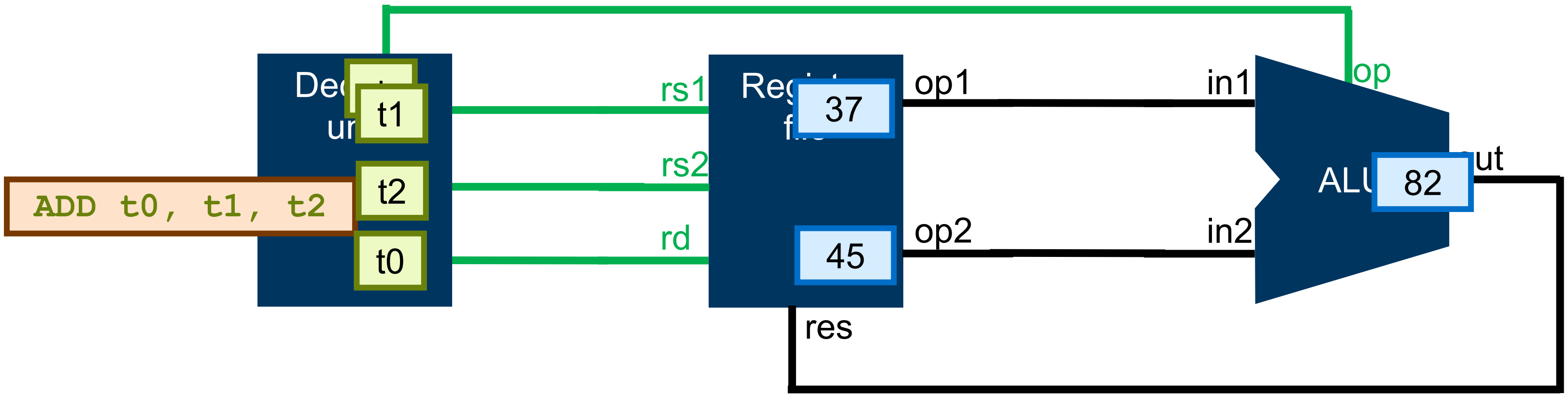
- **Fetch:** bring the next instruction from memory to control unit
- **Decode:** “understand” what the instruction needs to do and set up for execution
- **Execute:** perform the instruction

Repeat



RUHR UNIVERSITÄT BOCHUM RUB

A Deeper Look



Instruction Formats

Different formats for different operations

R-Type: 1 destination register, 2 source registers (e.g. `add rd, rs1, rs2`)

I-Type: 1 destination register, 1 source register, 1 immediate (e.g. `addi rd, rs1, imm`)

…

<table>
<tr><td>31 27</td><td>26 25</td><td>24 20</td><td>19 15</td><td>14 12</td><td>11 7</td><td>6 0</td><td></td></tr>
<tr><td colspan="2">funct7</td><td>rs2</td><td>rs1</td><td>funct3</td><td>rd</td><td>opcode</td><td>R-type</td></tr>
<tr><td colspan="3">imm[11:0]</td><td>rs1</td><td>funct3</td><td>rd</td><td>opcode</td><td>I-type</td></tr>
<tr><td colspan="2">imm[11:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:0]</td><td>opcode</td><td>S-type</td></tr>
<tr><td colspan="2">imm[12|10:5]</td><td>rs2</td><td>rs1</td><td>funct3</td><td>imm[4:1|11]</td><td>opcode</td><td>B-type</td></tr>
<tr><td colspan="5">imm[31:12]</td><td>rd</td><td>opcode</td><td>U-type</td></tr>
<tr><td colspan="5">imm[20|10:1|11|19:12]</td><td>rd</td><td>opcode</td><td>J-type</td></tr>
</table>

And now it's your turn…

How many bits are reserved for the immediate value of the `addi` instruction?

Summary

- What are registers
- How to use Ripes
- Some RISC-V Assembly
- What is machine code