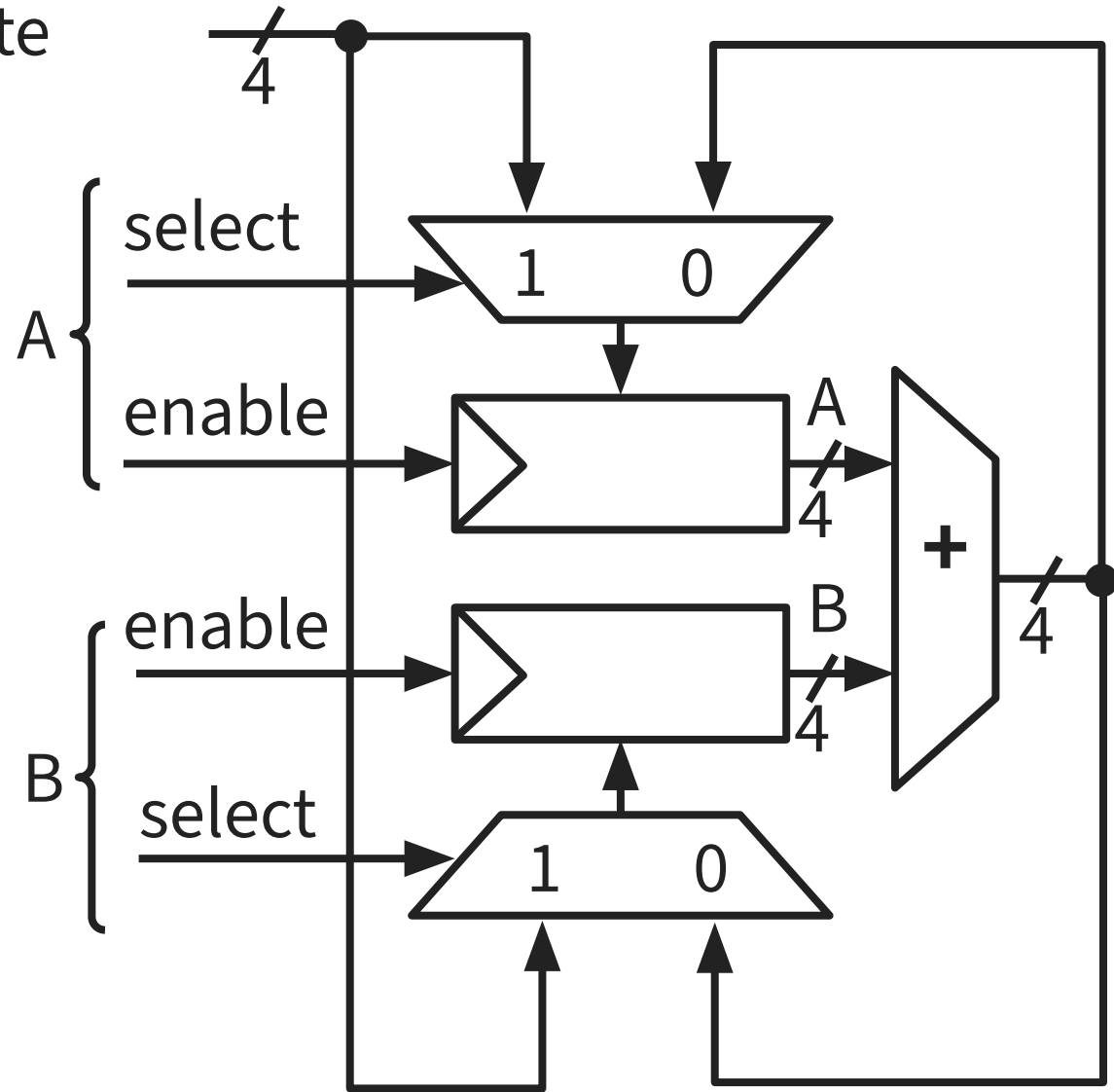


Review: Femto Proc

CS 3410: Computer System Organization and Programming

Fall 2025

immediate

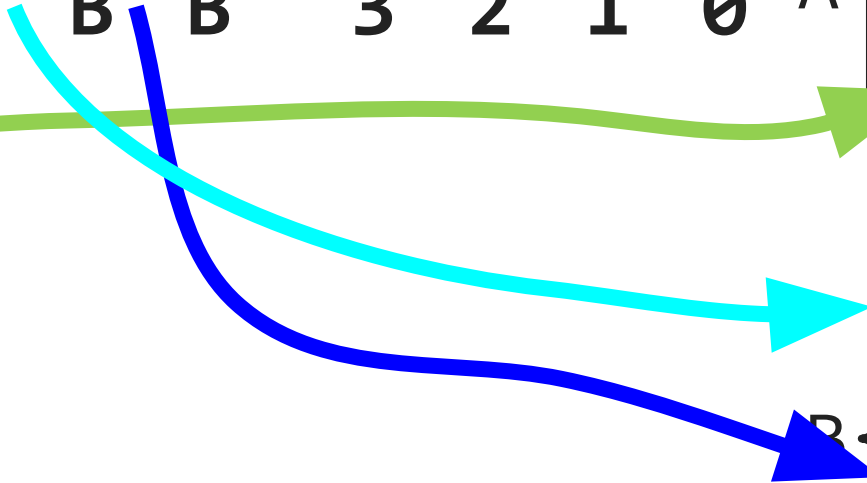


Processor Architecture

Instruction Format:

immediate

E_A **S_A** **E_B** **S_B** **I₃** **I₂** **I₁** **I₀** **A**



select

enable

enable

select

1

0

A

4

B

4

+

4

1

0

Processor Architecture

Instruction Format: immediate

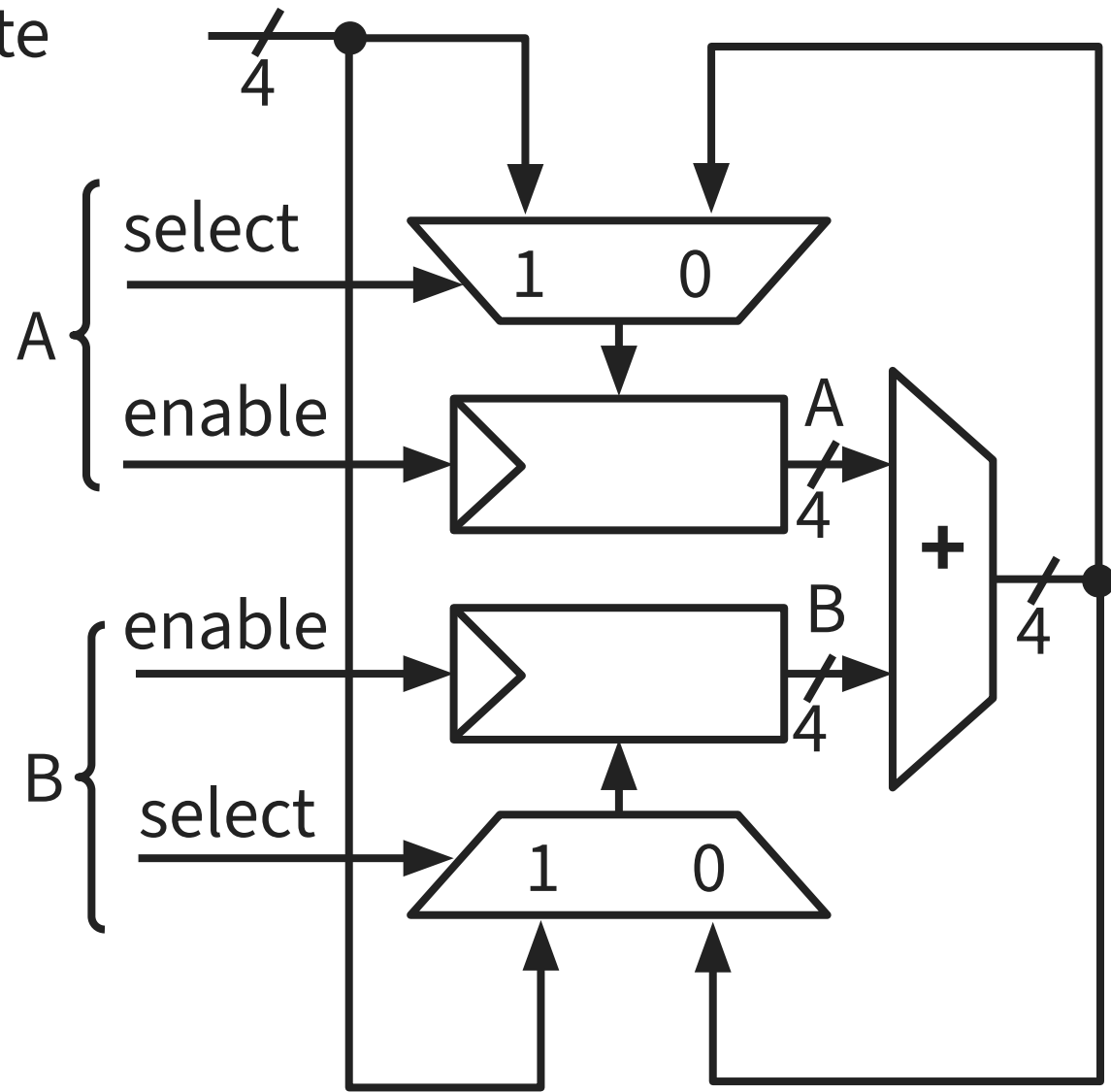
$E_A S_A E_B S_B I_3 I_2 I_1 I_0$

Machine Code

11 0? 0000

0? 11 1000

10 0? ????



Processor Architecture

Instruction Format: immediate

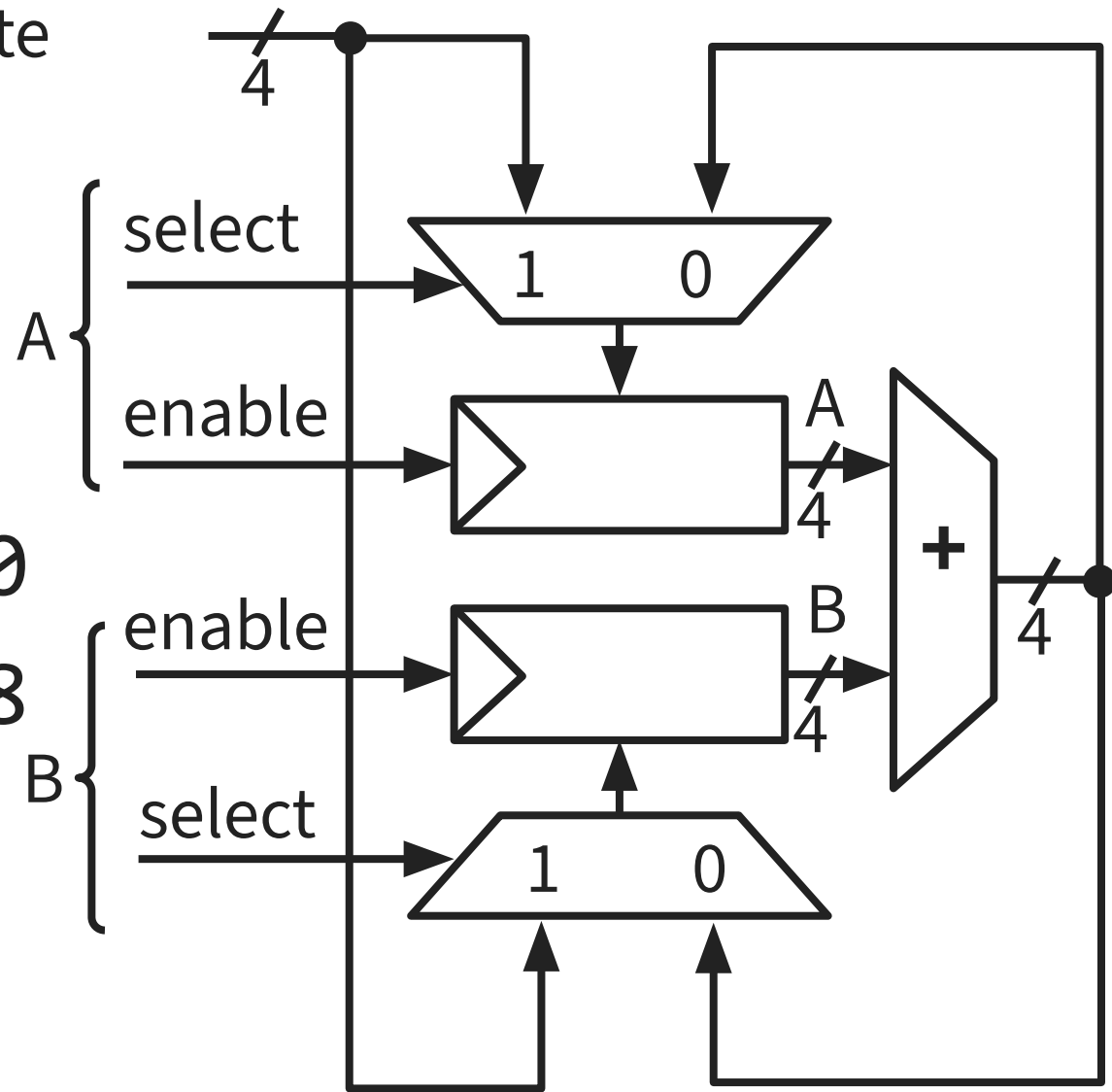
$E_A S_A E_B S_B I_3 I_2 I_1 I_0$

Machine Code Mnemonic

11 0? 0000 wrimm A, 0

0? 11 1000 wrimm B, 8

10 0? ???? add A

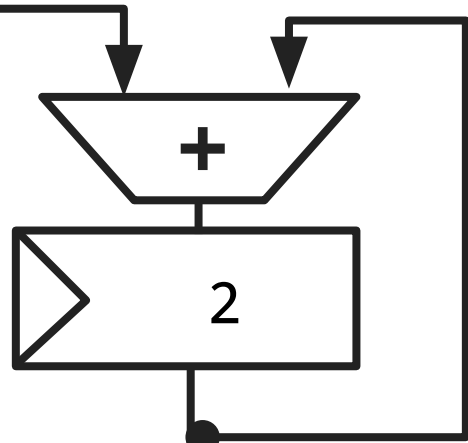


Processor Architecture

Instruction memory

Program Counter

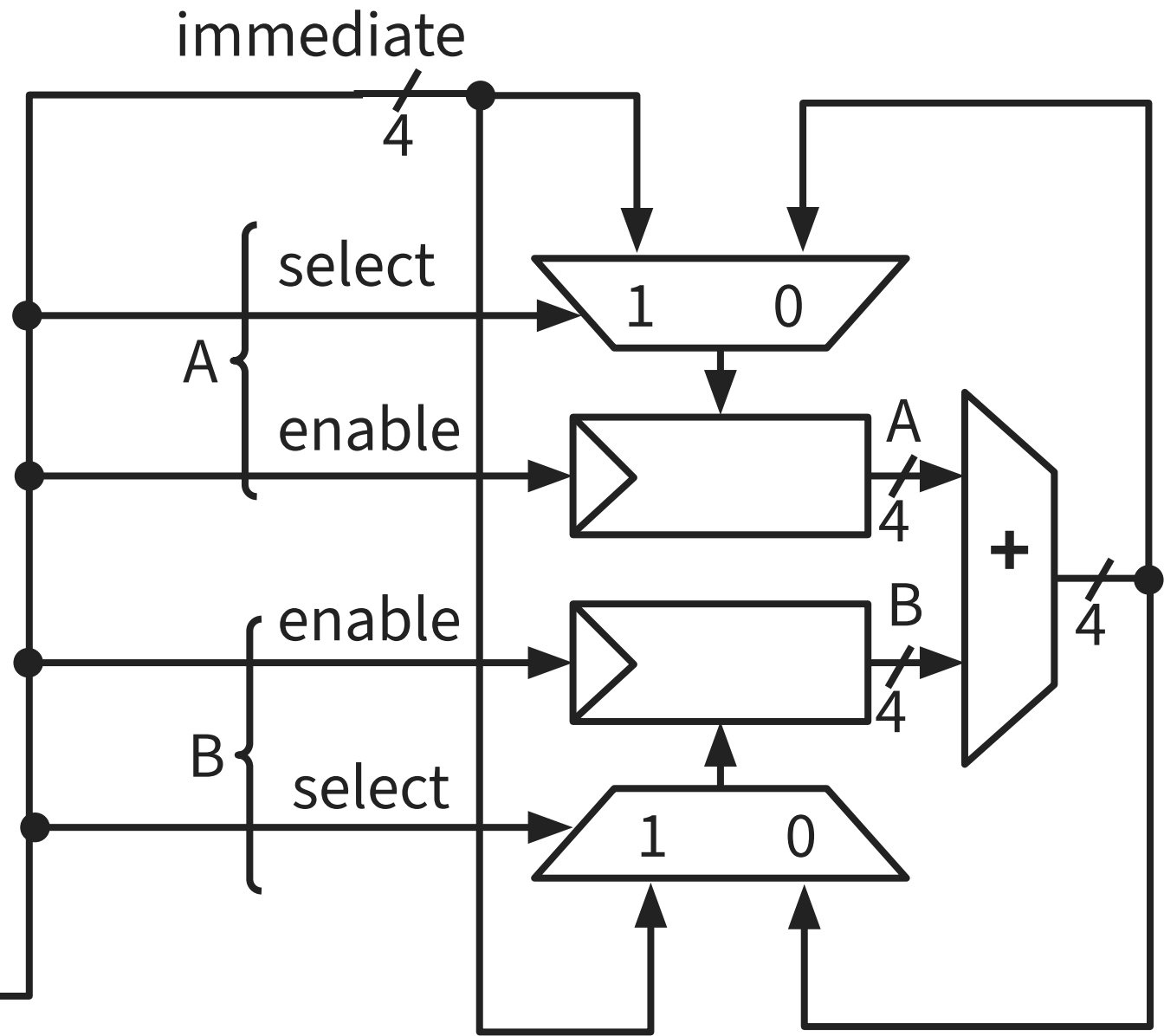
Addr	Data
0	11 00 0000
1	01 11 1000
2	10 00 0101
3	...



data @ addr

8

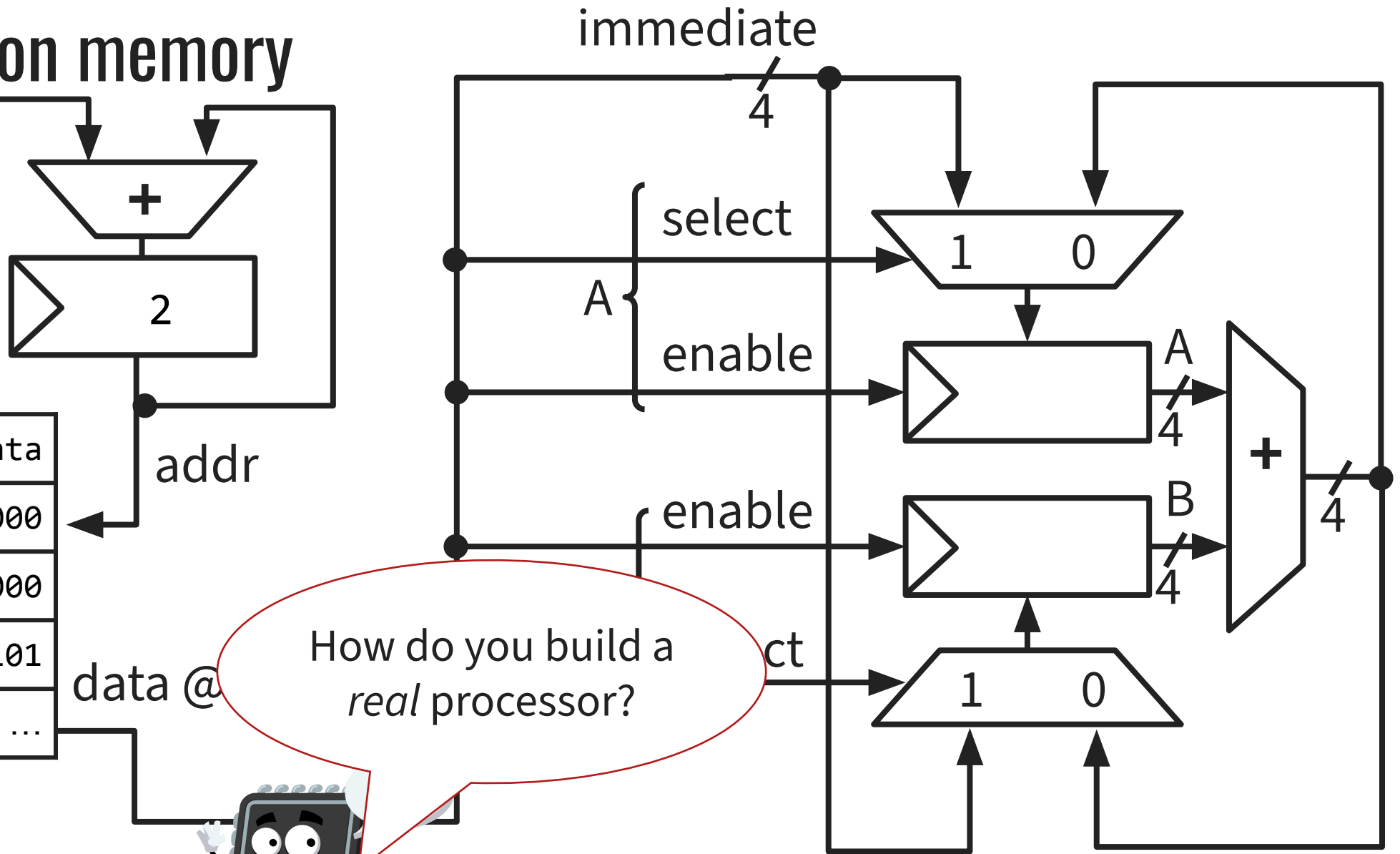
instruction word



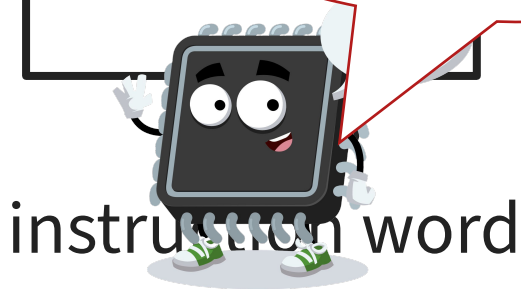
Instruction memory

Program Counter

Addr	Data
0	11 00 0000
1	01 11 1000
2	10 00 0101
3	...



How do you build a real processor?



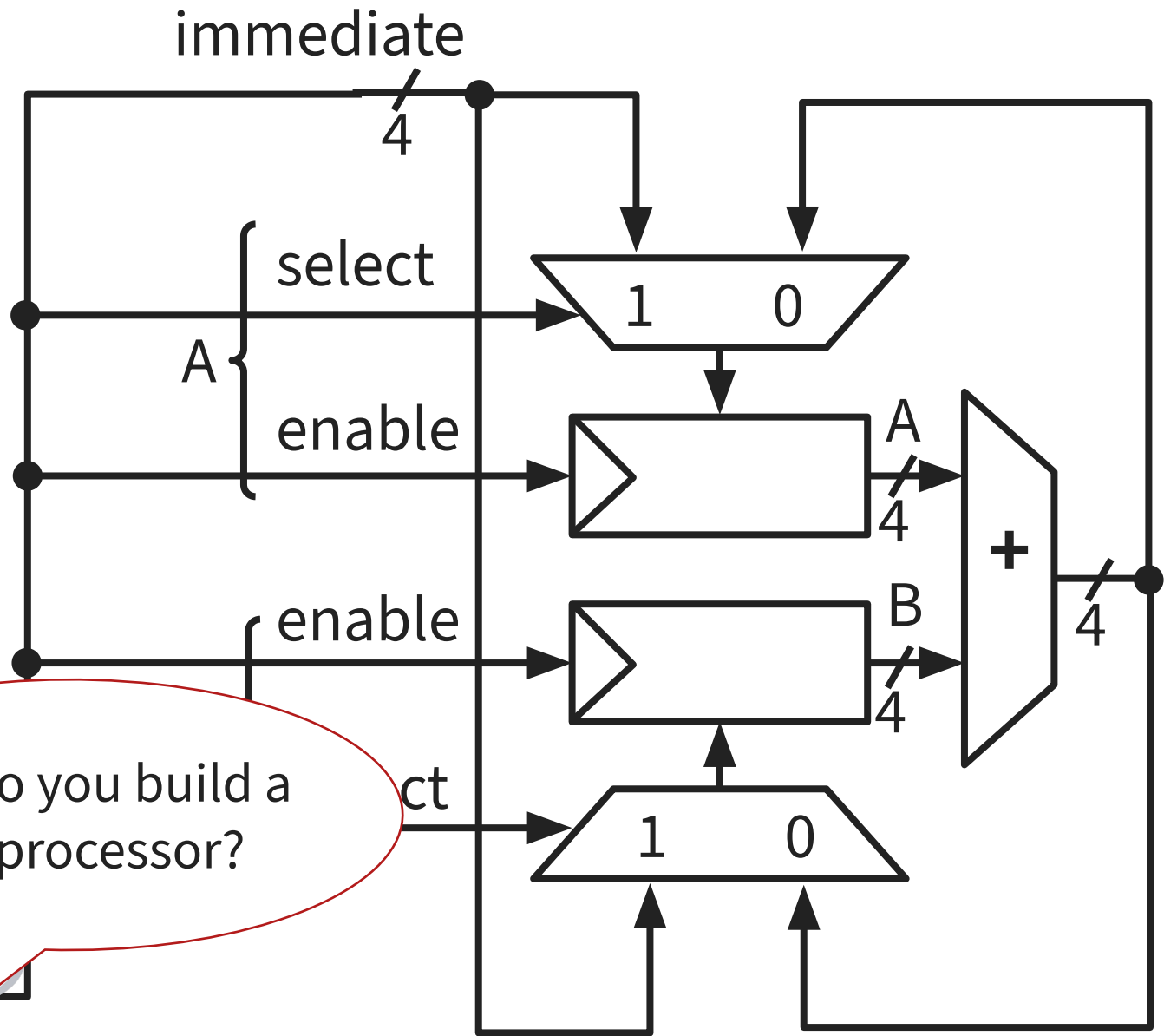
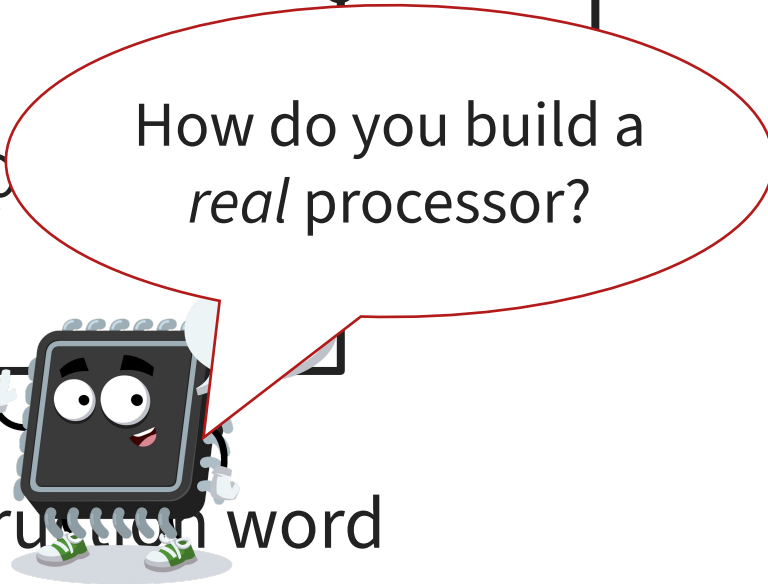
instruction word

Instruction memory

Program Counter

Addr	Data
0	11 00 0000
1	01 11 1000
2	10 00 0101
3	...

instruction word



RISC-V

CS 3410: Computer System Organization and Programming

Fall 2025



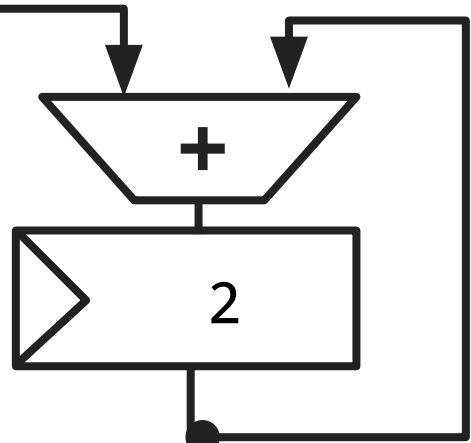
From FemtoProc to RISC-V



Instruction memory

Program Counter

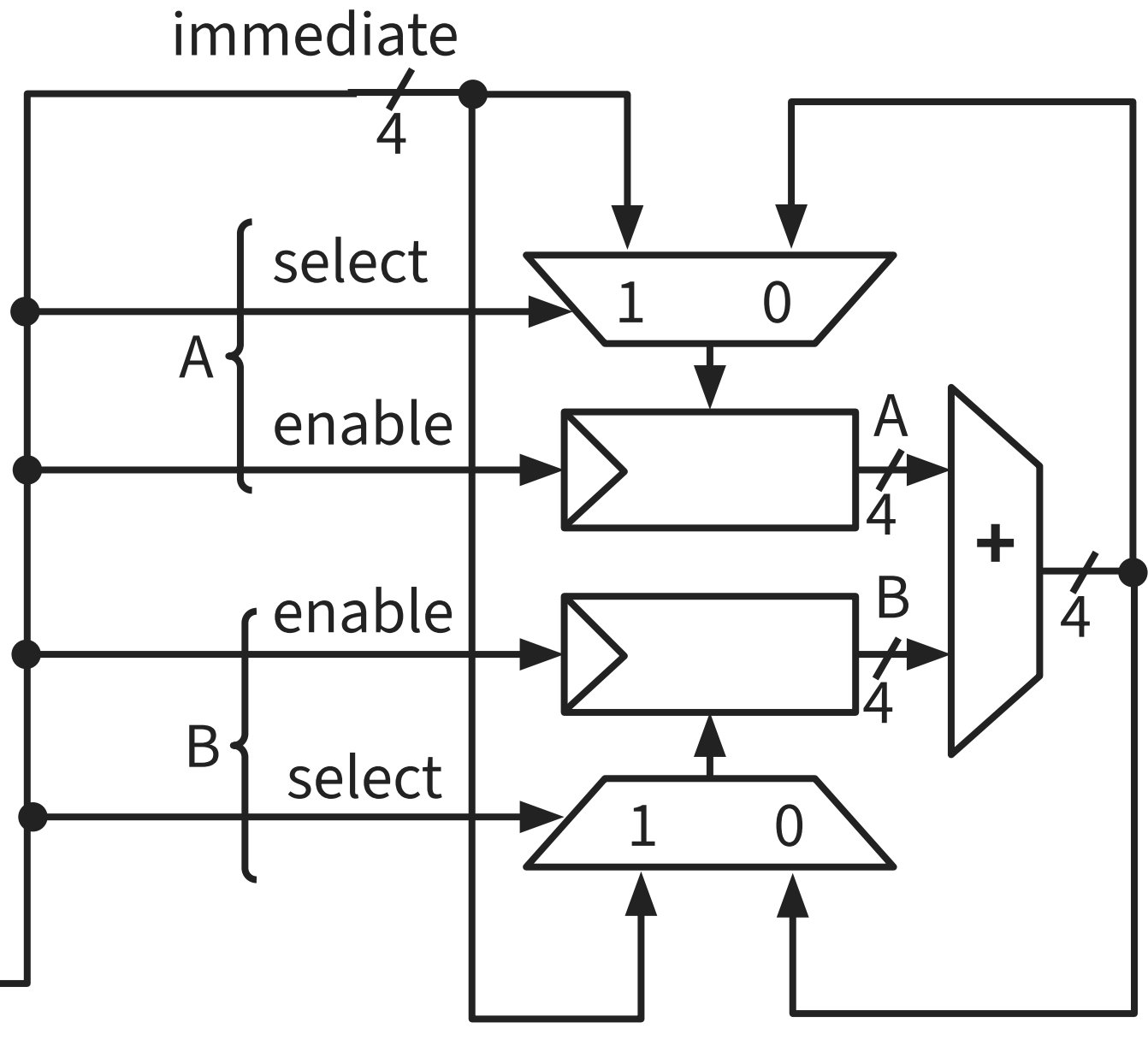
Addr	Data
0	11 00 0000
1	01 11 1000
2	10 00 0101
3	...

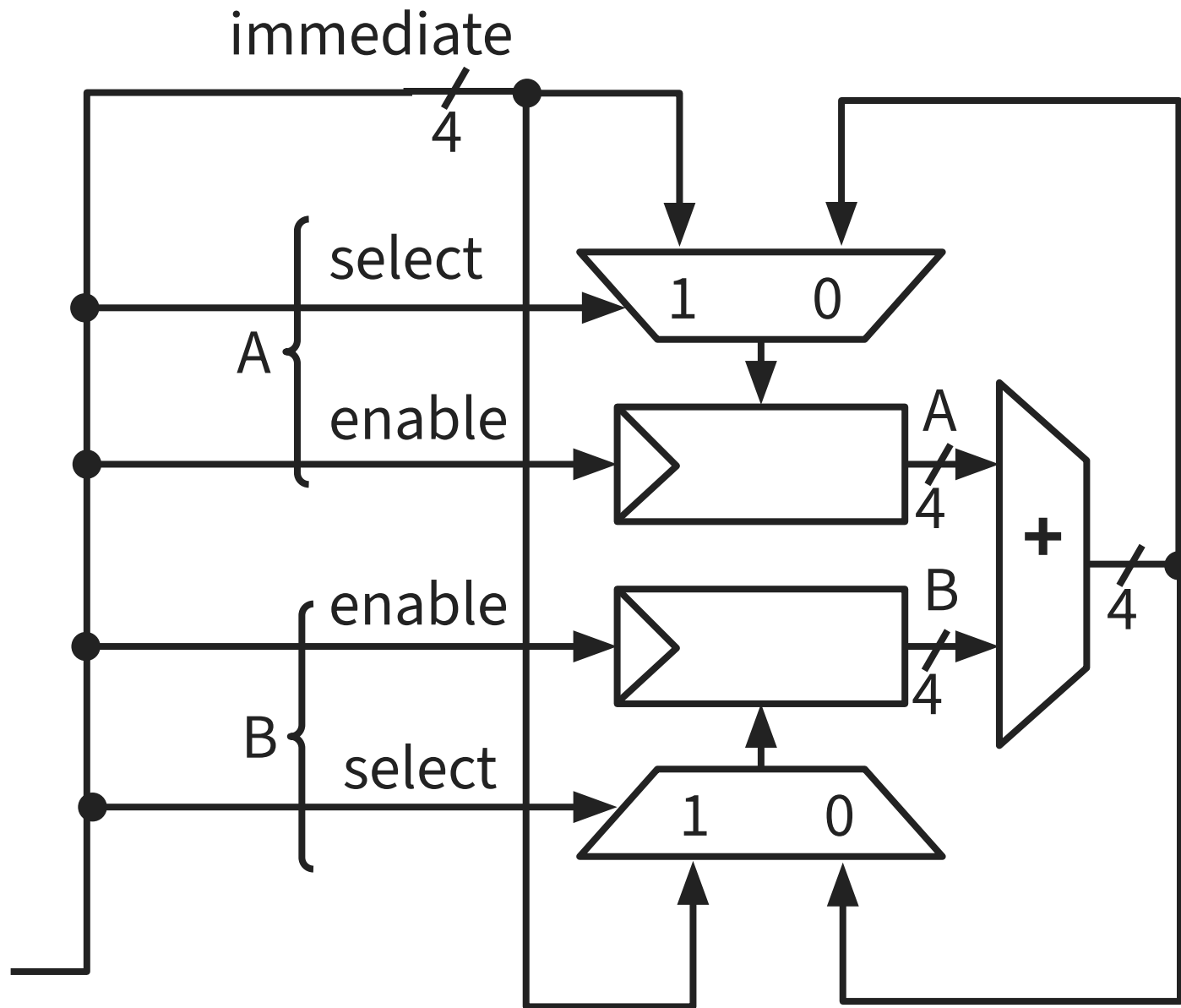
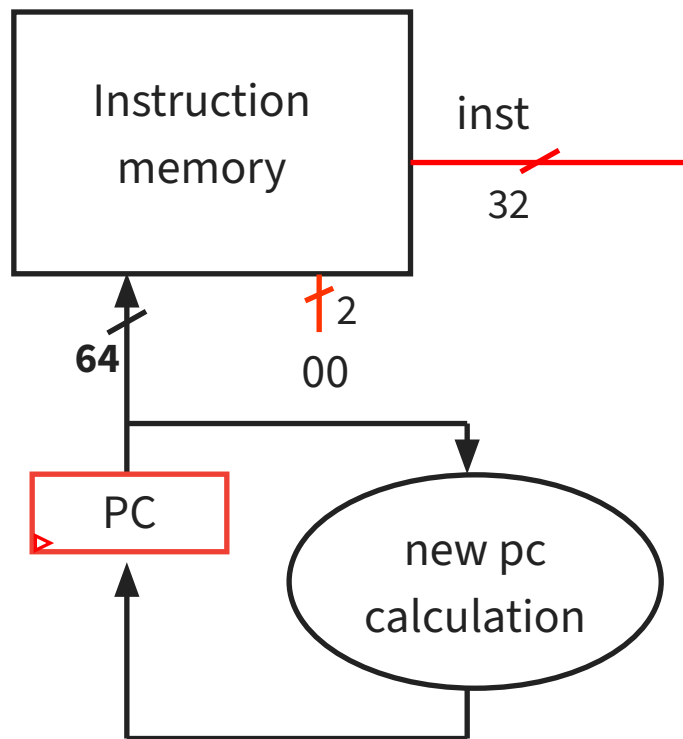


data @ addr

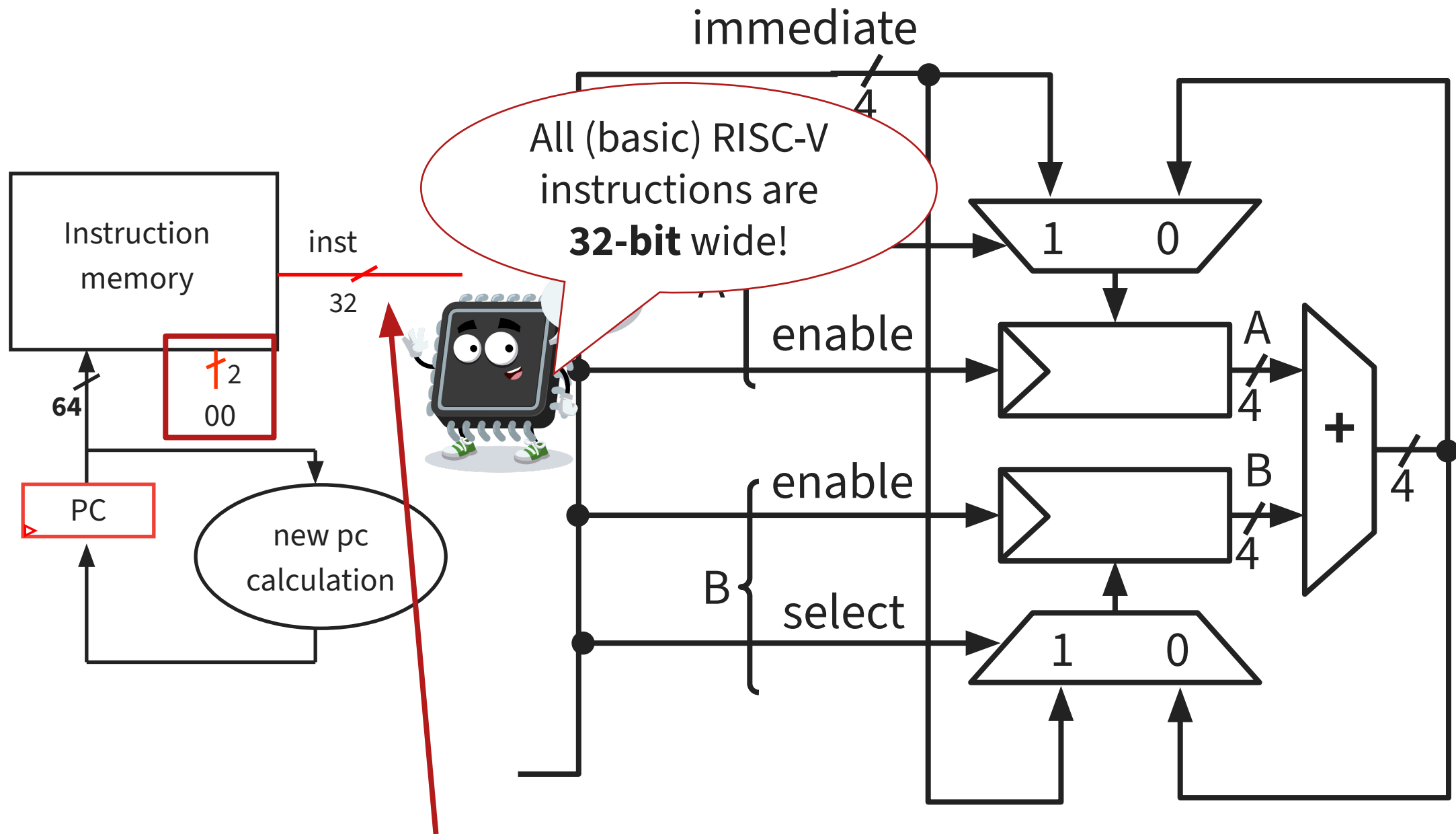
8

instruction word

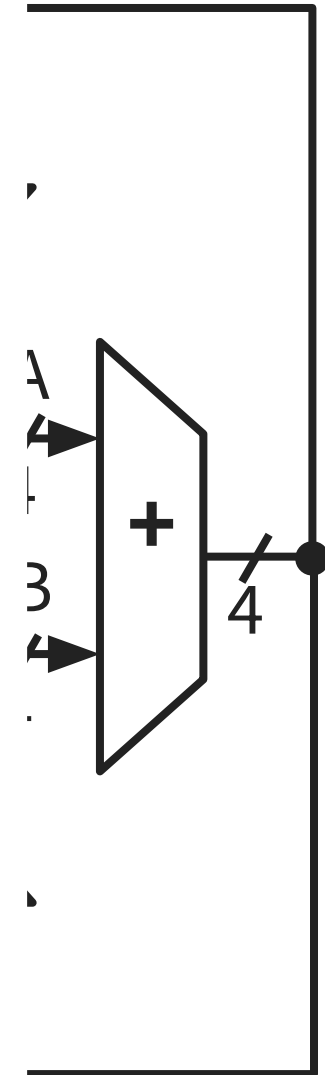
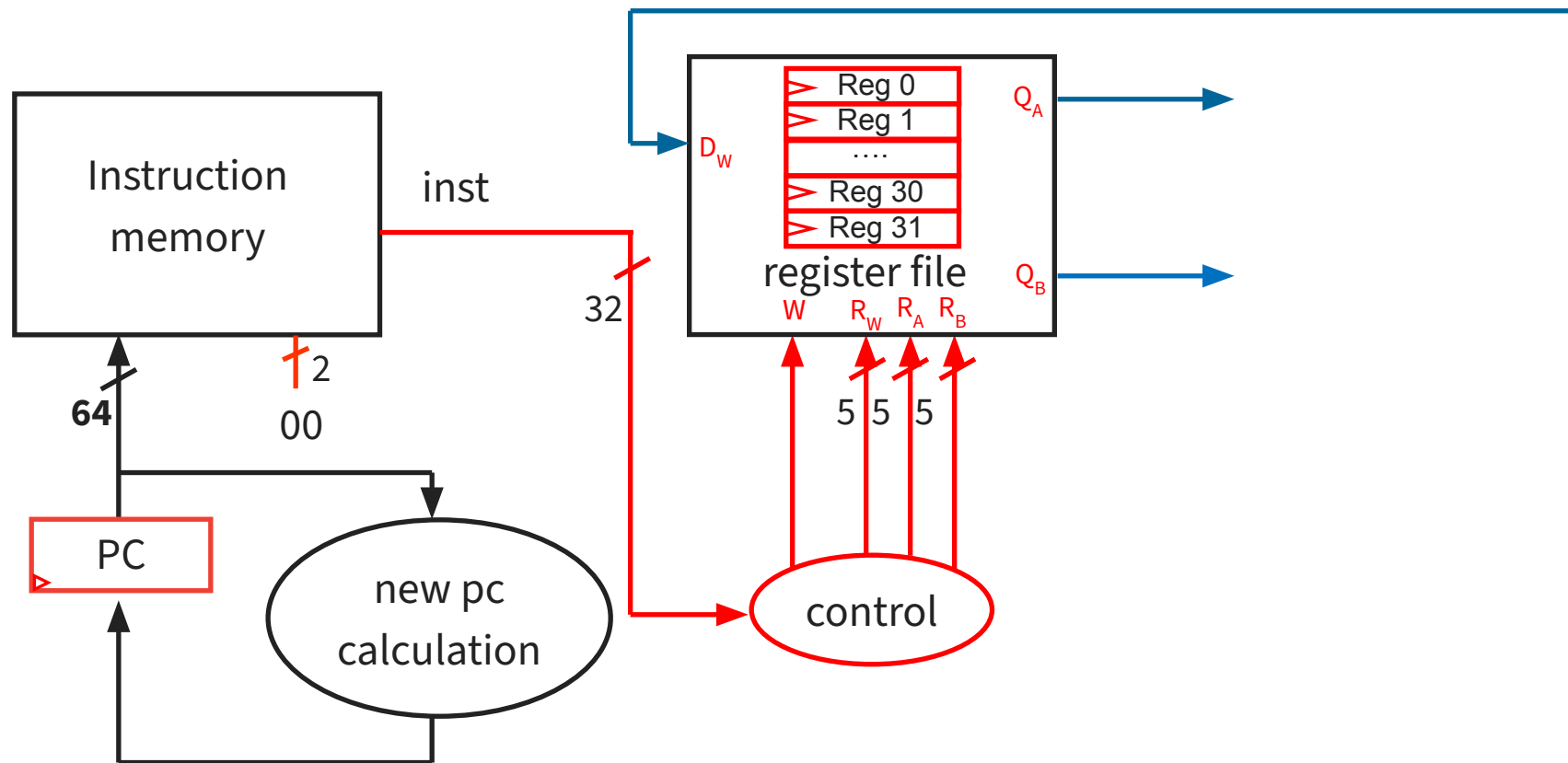


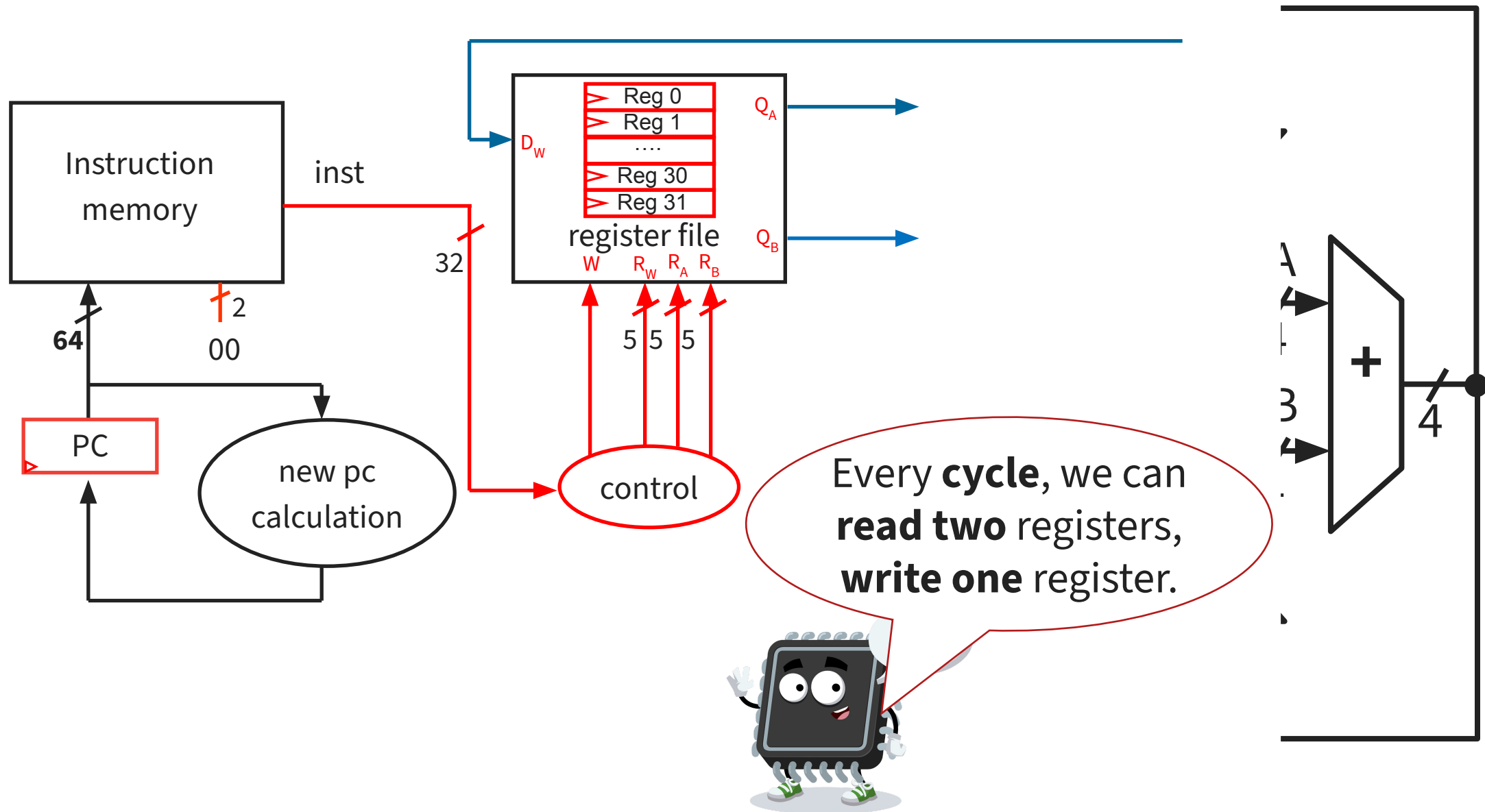


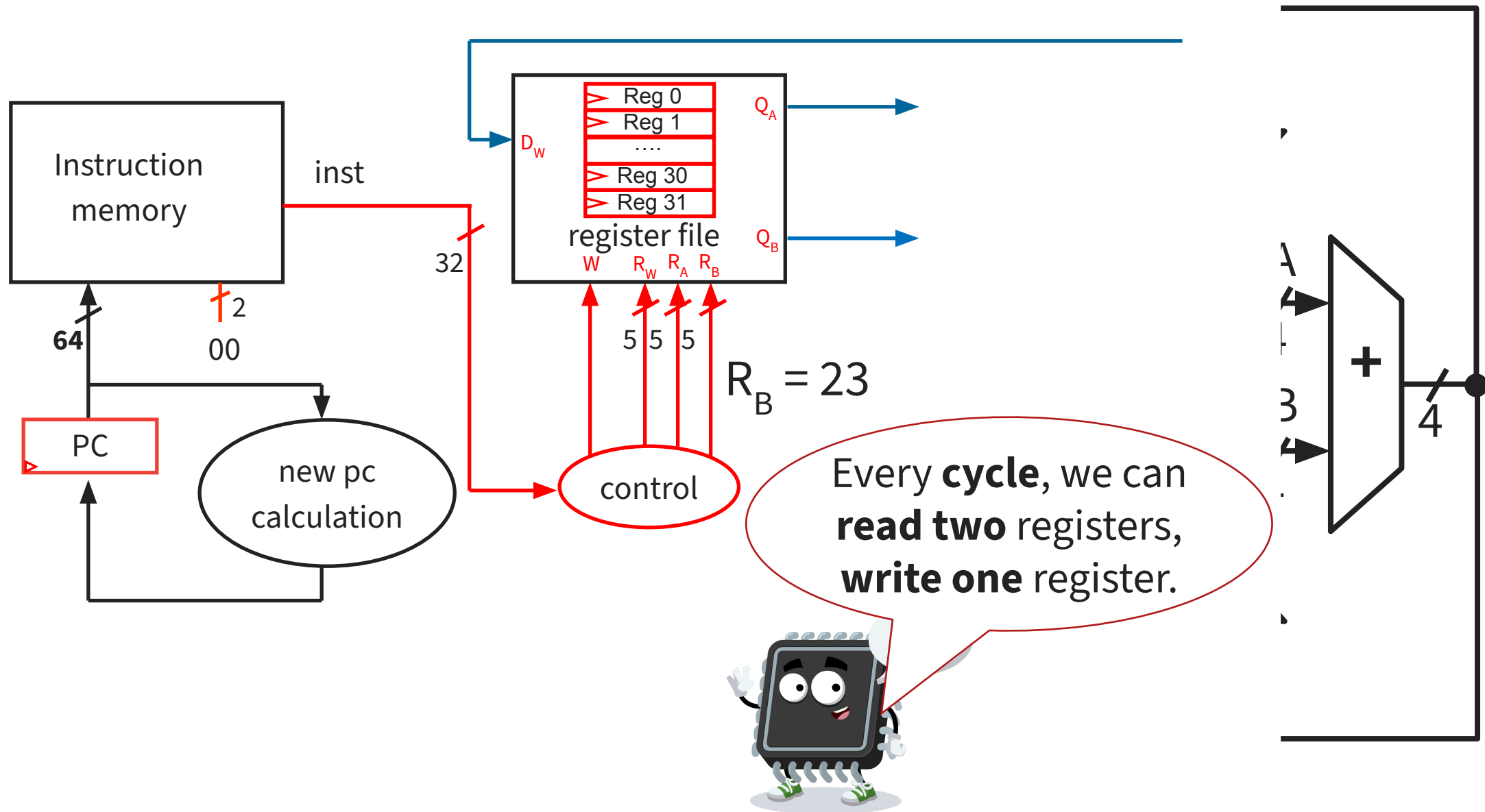
instruction word

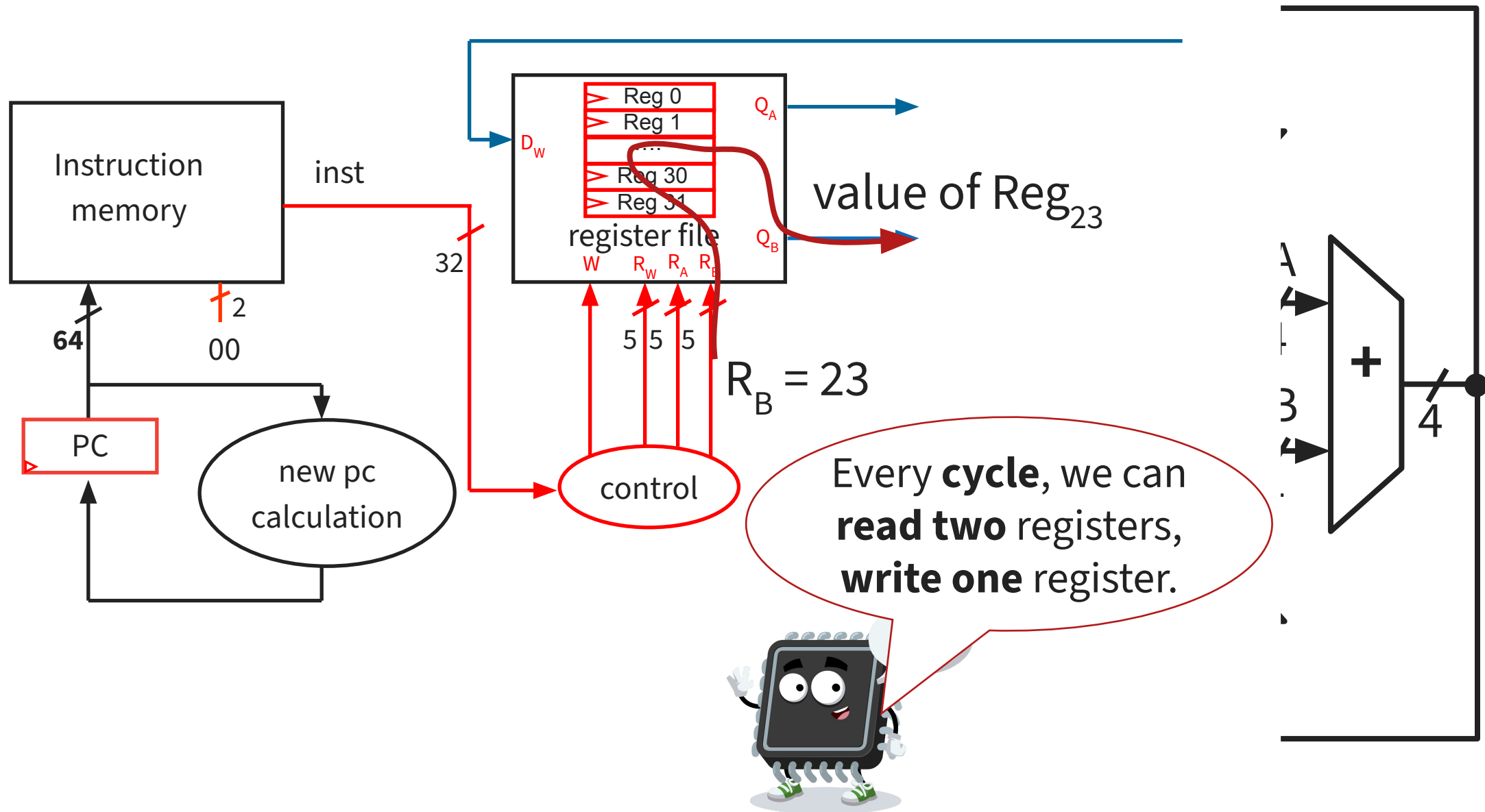


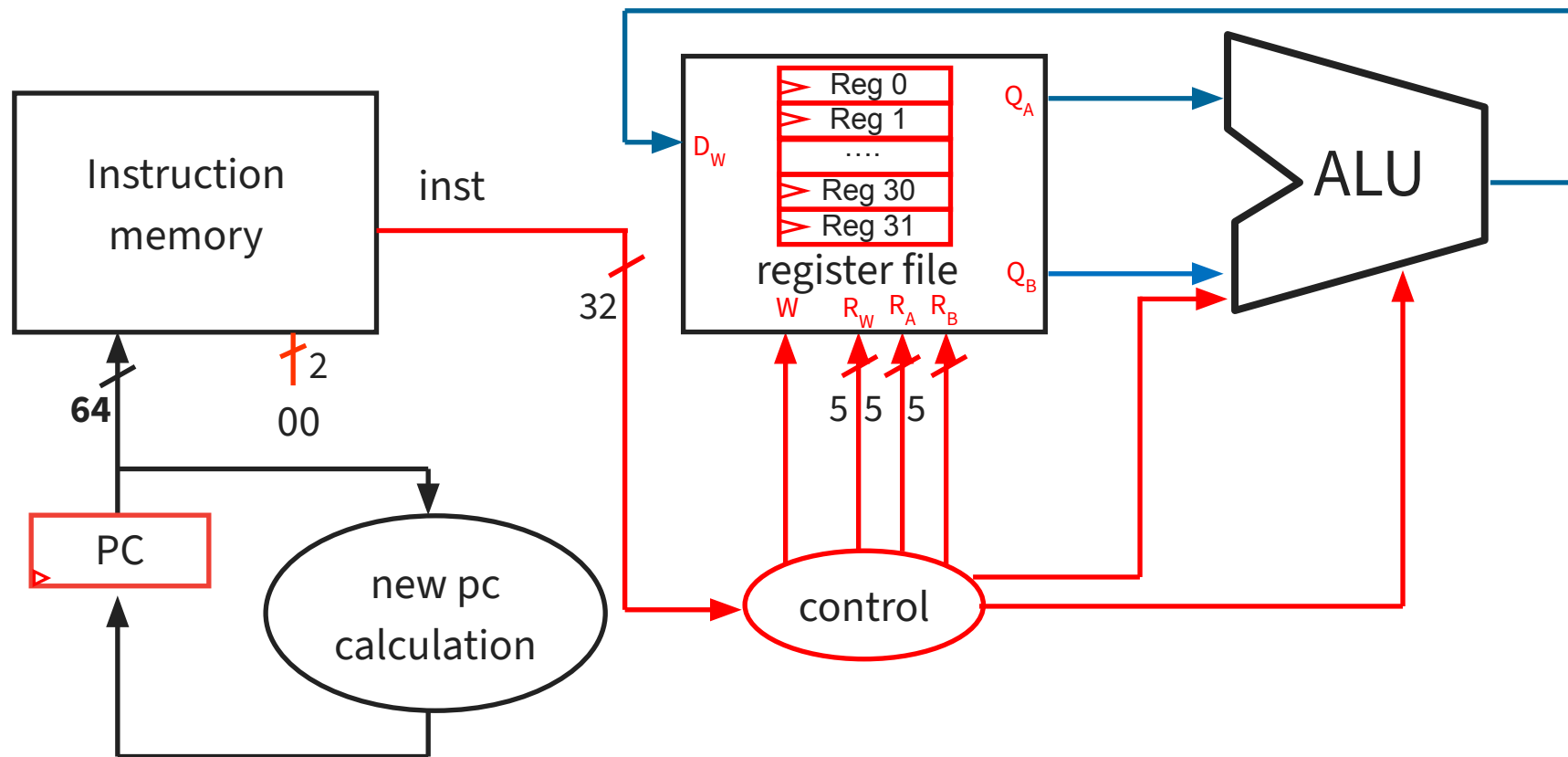
32-bit instruction word

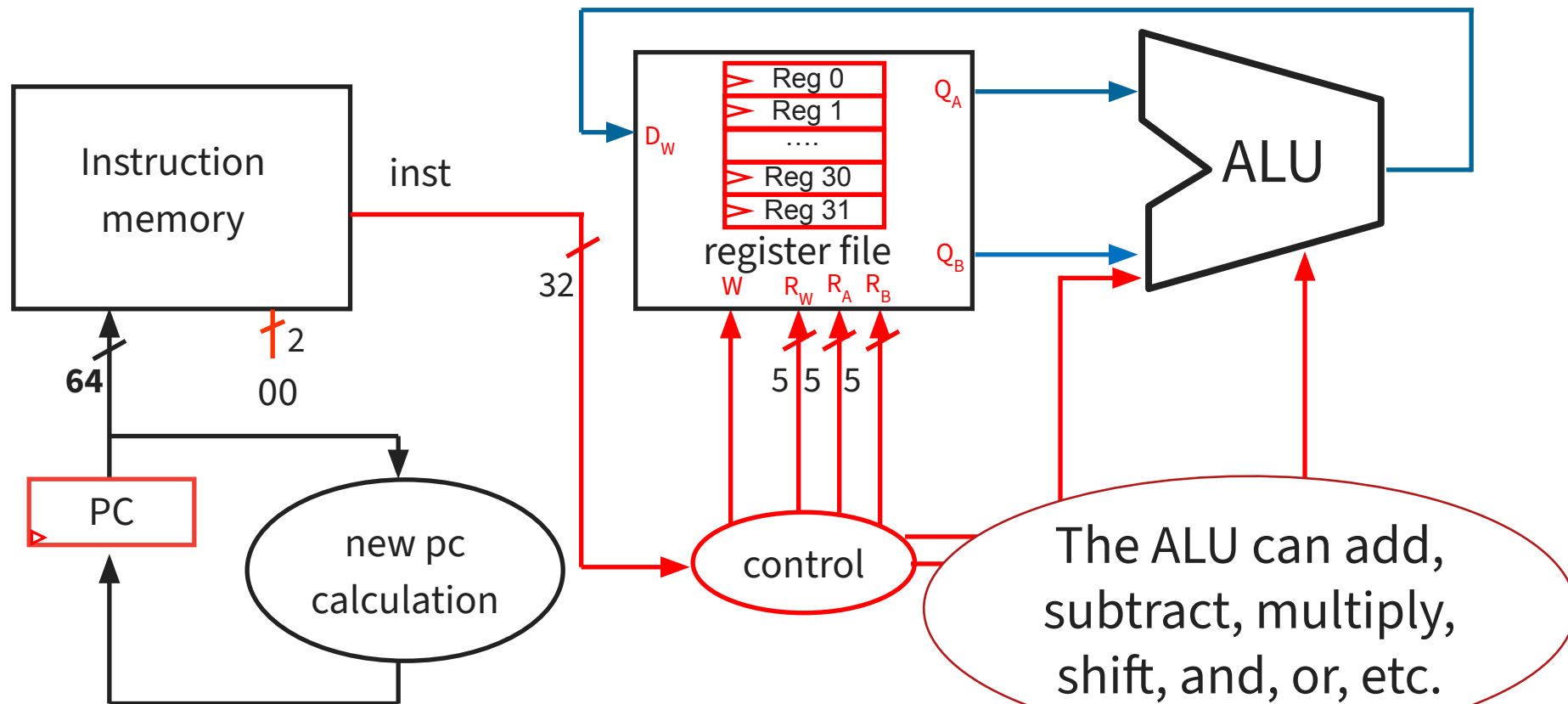












The ALU can add, subtract, multiply, shift, and, or, etc.

The RISC-V Add Instruction

<https://github.com/riscv/riscv-isa-manual>

The RISC-V Instruction Set Manual Volume I

Unprivileged Architecture

Version 20250924: Intermediate Release



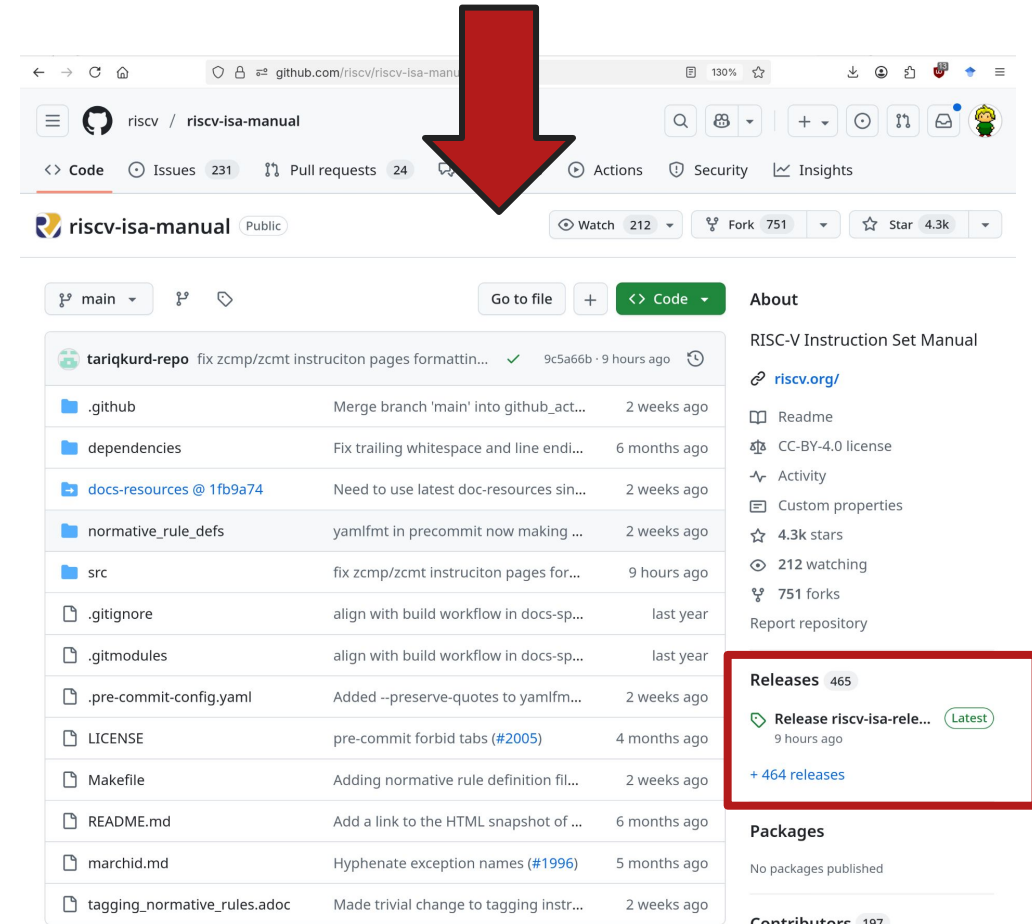
The RISC-V Add Instruction

The RISC-V Instruction Set Manual Volume I

Unprivileged Architecture

Version 20250924: Intermediate Release

<https://github.com/riscv/riscv-isa-manual>



The screenshot shows the GitHub repository page for `riscv/riscv-isa-manual`. A large red arrow points to the repository name. The page displays the repository's file structure, including folders like `.github`, `dependencies`, `docs-resources`, `normative_rule_defs`, `src`, and files like `.gitignore`, `.gitmodules`, `.pre-commit-config.yaml`, `LICENSE`, `Makefile`, `README.md`, `marchid.md`, and `tagging_normative_rules.adoc`. The right sidebar shows the repository's metadata, including the name `riscv-isa-manual`, the description `RISC-V Instruction Set Manual`, the license `CC-BY-4.0`, and the number of stars (4.3k), forks (751), and watchers (212). The `Releases` section is highlighted with a red box, showing the latest release `Release riscv-isa-rele...` from 9 hours ago, with a link to `+ 464 releases`.



The RISC-V Add Instruction

<https://github.com/riscv/riscv-isa-manual>

Release riscv-isa-release-9c5a66b-2025-09-24 Latest Compare

riscv-tech-admin released this 9 hours ago · riscv-isa-relea... · 9c5a66b ✓

This release was created by: aswaterman
Release of RISC-V ISA, built from commit [9c5a66b](#), is now available.

▼ **Assets** 8

riscv-privileged.epub	906 KB	9 hours ago
riscv-privileged.html	2.22 MB	9 hours ago
riscv-privileged.pdf	1.37 MB	9 hours ago
riscv-unprivileged.epub	1.92 MB	9 hours ago
riscv-unprivileged.html	8.54 MB	9 hours ago
riscv-unprivileged.pdf	4.46 MB	9 hours ago
Source code (zip)		9 hours ago
Source code (tar.gz)		9 hours ago

Set

github.com/riscv/riscv-isa-manual

riscv / riscv-isa-manual

Code Issues 231 Pull requests 24 Actions Security Insights

riscv-isa-manual Public

Watch 212 Fork 751 Star 4.3k

main

Go to file + Code

About

RISC-V Instruction Set Manual

[riscv.org/](#)

Readme

CC-BY-4.0 license

Activity

Custom properties

4.3k stars

212 watching

751 forks

Report repository

Releases 465

Release riscv-isa-rele... Latest

9 hours ago

+ 464 releases

Packages

No packages published

Contributors 197

tariqkurd-repo fix zcmp/zcmt instrucion pages formattin... ✓ 9c5a66b · 9 hours ago

.github Merge branch 'main' into github_act... 2 weeks ago

dependencies Fix trailing whitespace and line endi... 6 months ago

docs-resources @ 1fb9a74 Need to use latest doc-resources sin... 2 weeks ago

normative_rule_defs yamlfmt in precommit now making ... 2 weeks ago

src fix zcmp/zcmt instrucion pages for... 9 hours ago

.gitignore align with build workflow in docs-sp... last year

.gitmodules align with build workflow in docs-sp... last year

.pre-commit-config.yaml Added --preserve-quotes to yamlfm... 2 weeks ago

LICENSE pre-commit forbid tabs (#2002) 4 months ago

Makefile Adding normative rule definition fil... 2 weeks ago

README.md Add a link to the HTML snapshot of ... 6 months ago

marchid.md Hyphenate exception names (#1996) 5 months ago

tagging_normative_rules.adoc Made trivial change to tagging instr... 2 weeks ago

The RISC-V Add Instruction

The RISC-V Instruction Set Manual Volume I

Unprivileged Architecture

Version 20250924: Intermediate Release



The RISC-V Add Instruction

2.4.2. Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.

31							25		24		20		19		15		14		12		11		7		6		0				
funct7							rs2					rs1					funct3			rd					opcode						
7							5					5					3			5					7						
0	0	0	0	0	0	0	src2					src1					ADD/SLT[U]			dest					OP						
0	0	0	0	0	0	0	src2					src1					AND/OR/XOR			dest					OP						
0	0	0	0	0	0	0	src2					src1					SLL/SRL			dest					OP						
0	1	0	0	0	0	0	src2					src1					SUB/SRA			dest					OP						

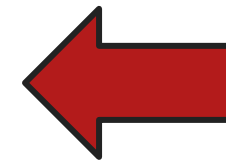
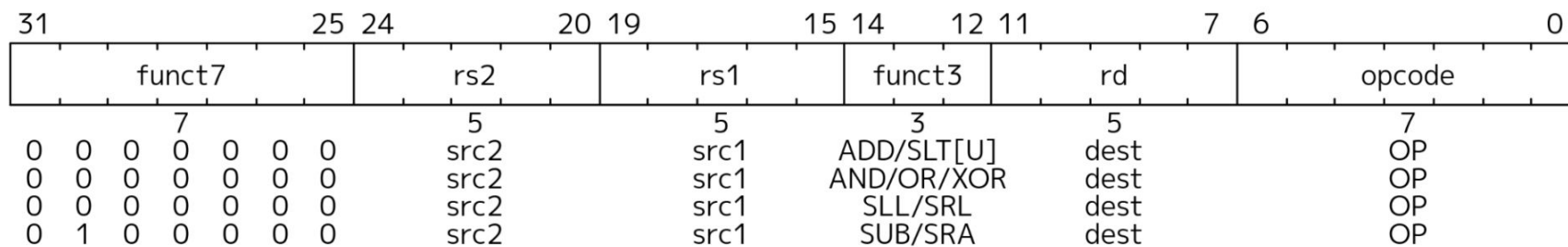
ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, x0, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd*, *rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

The RISC-V Add Instruction

2.4.2. Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.



32-bit
instruction
word

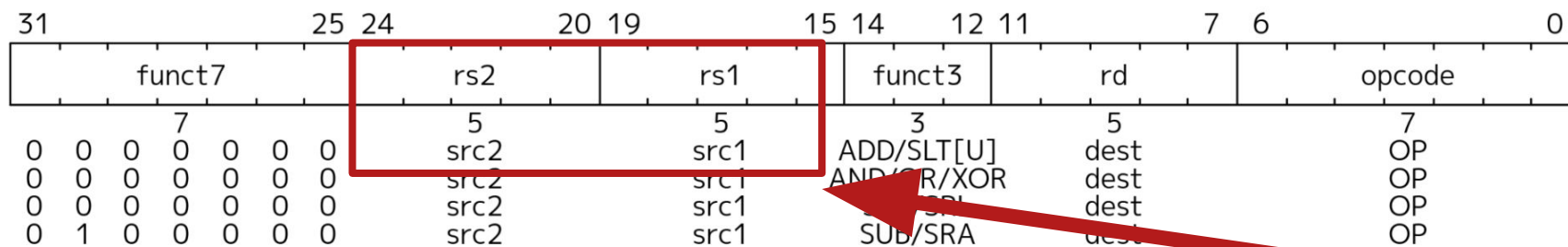
ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, x0, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd*, *rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

The RISC-V Add Instruction

2.4.2. Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.



32-bit
instruction
word

two read
registers
src2: 0..31
src1: 0..31

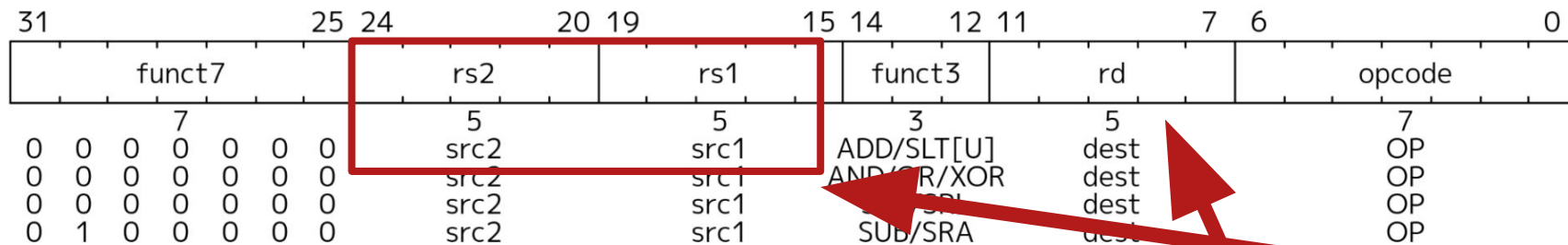
ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, x0, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd*, *rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

The RISC-V Add Instruction

2.4.2. Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.



ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, x0, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd*, *rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

32-bit
instruction
word

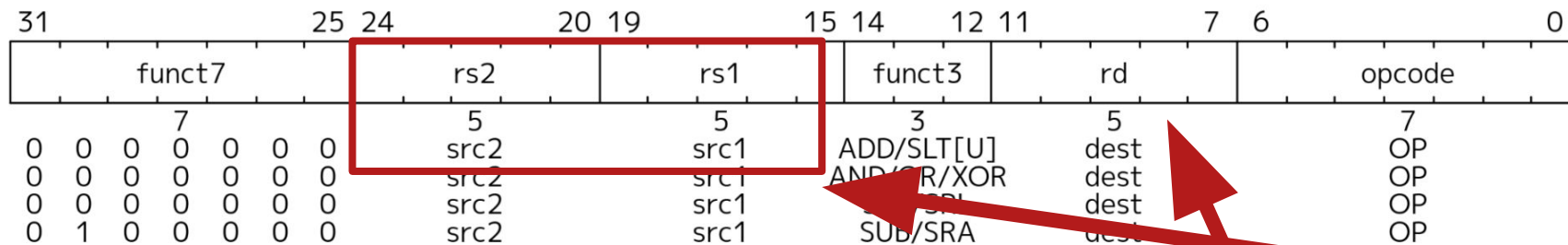
two **read**
registers
src2: 0..31
src1: 0..31

one **write** (destination) **registers**
dest: 0..31

The RISC-V Add Instruction

2.4.2. Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* registers as source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the type of operation.



ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, x0, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd*, *rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

semantics

one write (destination) registers
dest: 0..31

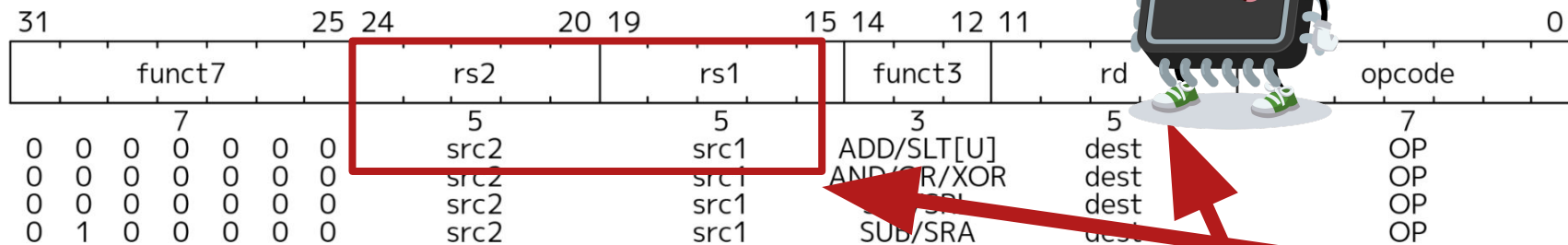
two read registers
src2: 0..31
src1: 0..31

(what the instruction actually does)

The RISC-V Add Instruction

2.4.2. Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* and *rs2* source operands and write the result into register *rd*. The *funct7* and *funct3* fields select the operation.



ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, x0, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd*, *rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

semantics

one write (destination) registers
dest: 0..31

two read registers
src2: 0..31
src1: 0..31

Where is the mnemonic?

32-bit instruction word

(what the instruction actually does)

Where is the mnemonic?
What should funct3 be?

RV32I defines several arithmetic R-type operations. All operations read the register *rs1* as source operand and write the result into register *rd*. The *funct7* and *funct3* fields select the operation.



```
two read
registers
src2: 0..31
src1: 0..31
```

ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. Note, SLTU *rd*, *x0*, *rs2* sets *rd* to 1 if *rs2* is not equal to zero, otherwise sets *rd* to zero (assembler pseudoinstruction SNEZ *rd*, *rs*). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register *rs1* by the shift amount held in the lower 5 bits of register *rs2*.

one **write** (destination) **registers**
dest: 0..31

semantics

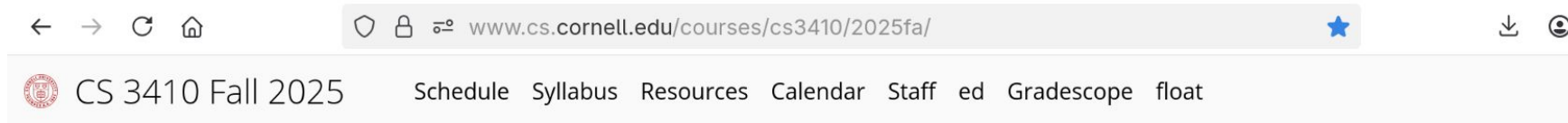
(what the instruction actually does)

The RISC-V Add Instruction: 2nd Try



The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>



Computer System Organization and Programming

CS 3410 - Fall 2025 taught by Giulia Guidi and Kevin Laeuer

Lecture: MoWe 1:25pm-2:40pm ET, Uris Hall G01

Prelim: October 9, 2025 7:30 PM ET

Makeup Prelim: October 16, 2025 5:30 PM ET

Final: TBD

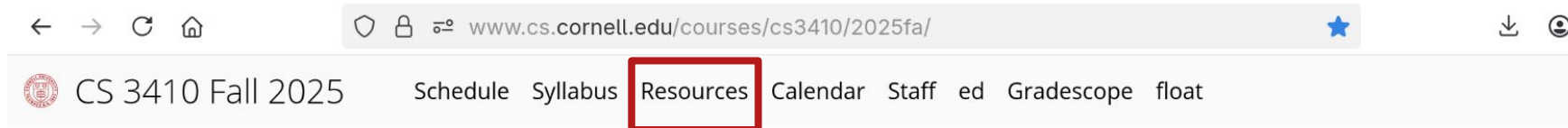
Makeup Final: TBD

Week	Date	Lecture	Lab	Assignment
1	Mon 8/25	• Intro (Notes) • 1+1=2 (Notes)	Lab 1: Nice to C You	• A1: Printf (Due: Wed 9/3) • Introductory Survey due Friday 8/29 • Week 1 Topic Mastery Quiz due Saturday 8/30 • E0 - E4: Course Policies, SSH, Unix & Git, C Compilation, Makefiles due Sunday 9/7
	Wed 8/27	• Numbers (Notes) • C Intro (Notes)		



The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>



Computer System Organization and Programming

CS 3410 - Fall 2025 taught by Giulia Guidi and Kevin Laeuffer

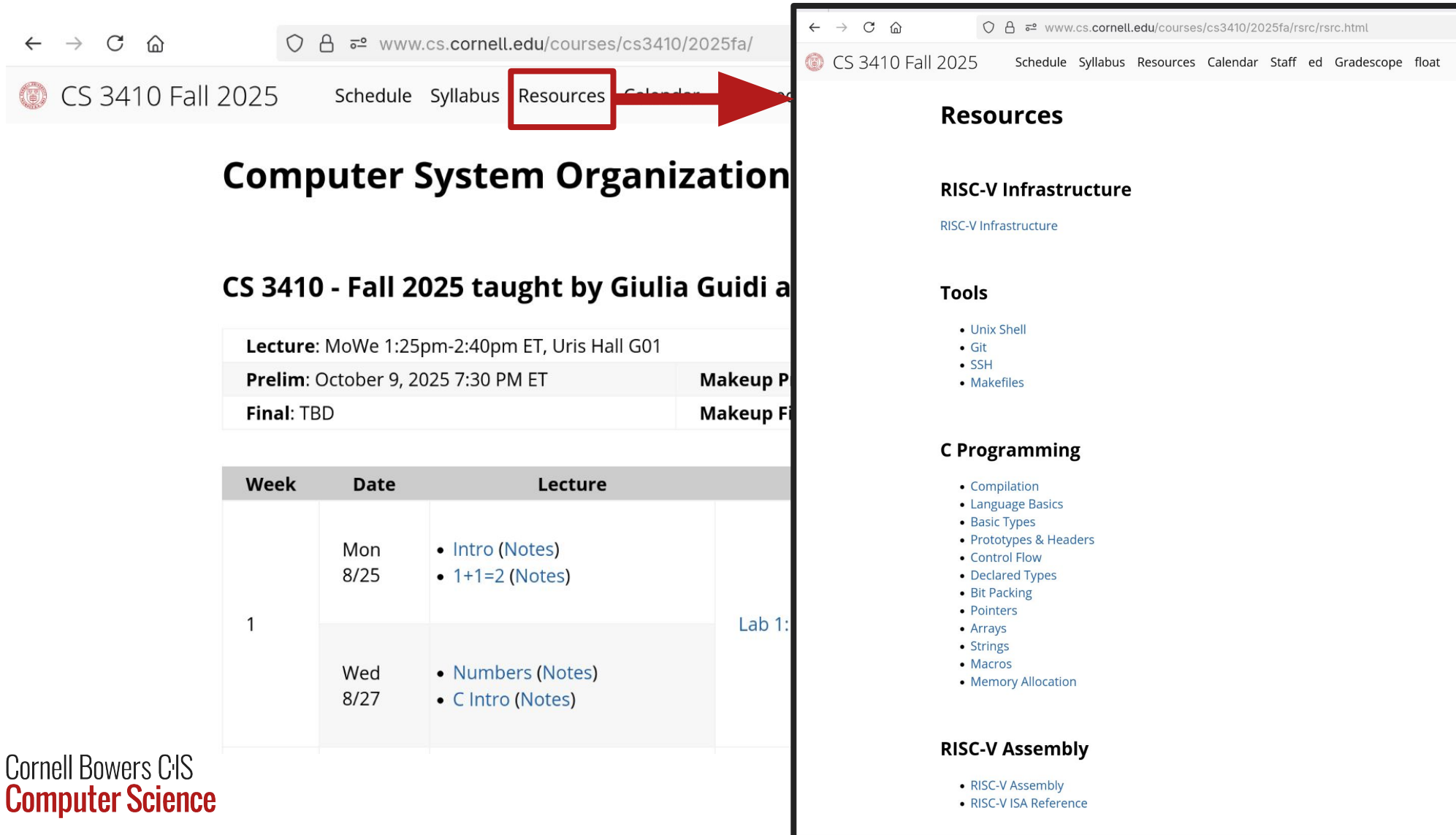
Lecture: MoWe 1:25pm-2:40pm ET, Uris Hall G01	
Prelim: October 9, 2025 7:30 PM ET	Makeup Prelim: October 16, 2025 5:30 PM ET
Final: TBD	Makeup Final: TBD

Week	Date	Lecture	Lab	Assignment
1	Mon 8/25	• Intro (Notes) • 1+1=2 (Notes)	Lab 1: Nice to C You	• A1: Printf (Due: Wed 9/3) • Introductory Survey due Friday 8/29 • Week 1 Topic Mastery Quiz due Saturday 8/30 • E0 - E4: Course Policies, SSH, Unix & Git, C Compilation, Makefiles due Sunday 9/7
	Wed 8/27	• Numbers (Notes) • C Intro (Notes)		



The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>



The screenshot shows the CS 3410 Fall 2025 course page. A red box highlights the 'Resources' link in the top navigation bar, with a red arrow pointing to a detailed view of the Resources page. The detailed view shows a list of resources categorized under 'RISC-V Infrastructure', 'Tools', 'C Programming', and 'RISC-V Assembly'.

CS 3410 Fall 2025 Schedule Syllabus **Resources** Calendar

Computer System Organization

CS 3410 - Fall 2025 taught by Giulia Guidi and

Lecture: MoWe 1:25pm-2:40pm ET, Uris Hall G01

Prelim: October 9, 2025 7:30 PM ET **Makeup P**

Final: TBD **Makeup F**

Week	Date	Lecture
1	Mon 8/25	• Intro (Notes) • 1+1=2 (Notes)
	Wed 8/27	• Numbers (Notes) • C Intro (Notes)

Lab 1:

Resources

RISC-V Infrastructure

[RISC-V Infrastructure](#)

Tools

- [Unix Shell](#)
- [Git](#)
- [SSH](#)
- [Makefiles](#)

C Programming

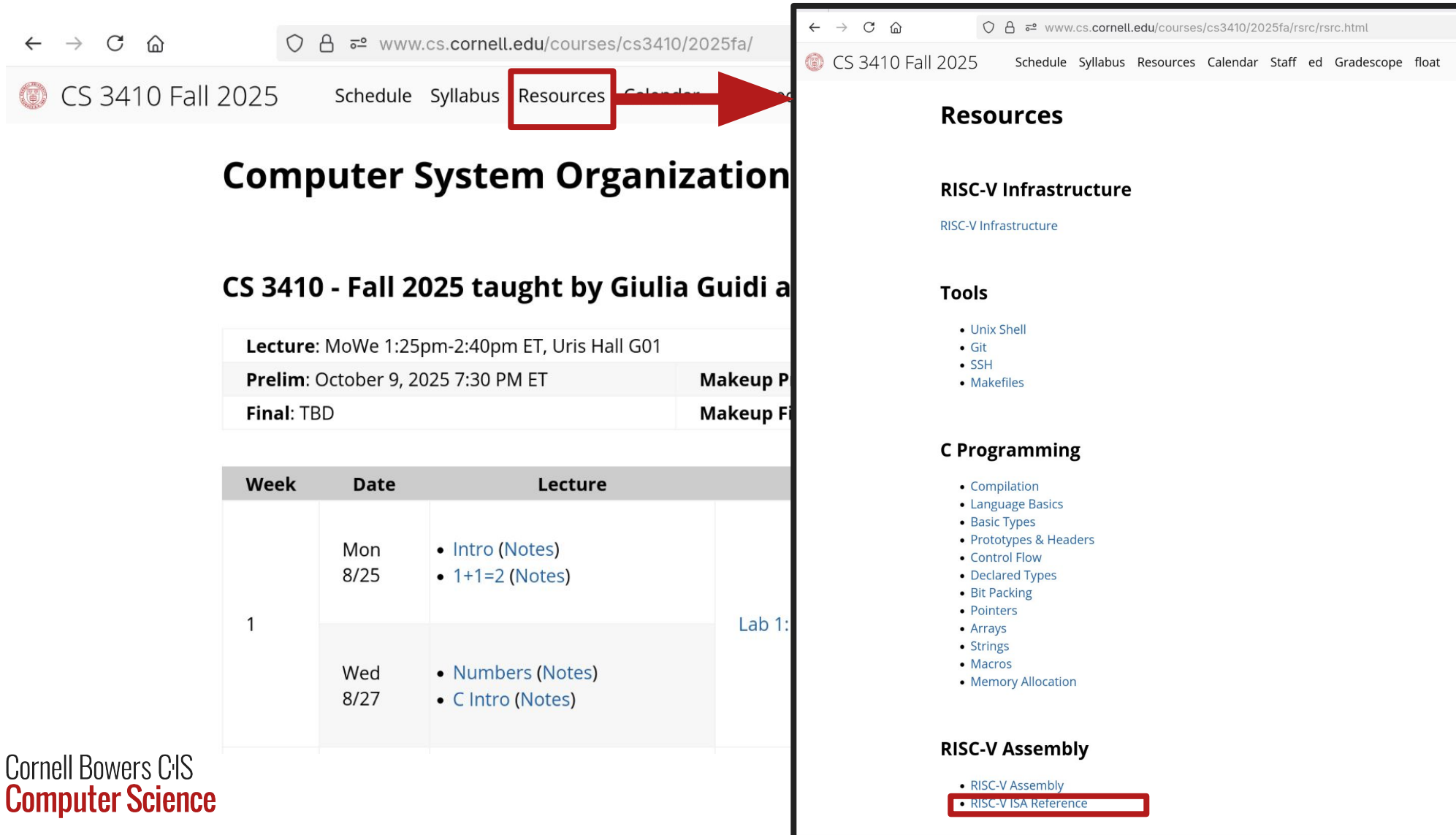
- [Compilation](#)
- [Language Basics](#)
- [Basic Types](#)
- [Prototypes & Headers](#)
- [Control Flow](#)
- [Declared Types](#)
- [Bit Packing](#)
- [Pointers](#)
- [Arrays](#)
- [Strings](#)
- [Macros](#)
- [Memory Allocation](#)

RISC-V Assembly

- [RISC-V Assembly](#)
- [RISC-V ISA Reference](#)

The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>



The image shows a screenshot of the CS 3410 Fall 2025 course page. The 'Resources' link is highlighted with a red box, and a red arrow points from it to a detailed view of the resources page. The detailed view shows a list of resources including RISC-V Infrastructure, Tools, C Programming, and RISC-V Assembly. The 'RISC-V Assembly' section is highlighted with a red box, and the 'RISC-V ISA Reference' link is also highlighted with a red box.

CS 3410 Fall 2025 Schedule Syllabus **Resources** Calendar Staff ed Gradescope float

Computer System Organization

CS 3410 - Fall 2025 taught by Giulia Guidi and others

Lecture: MoWe 1:25pm-2:40pm ET, Uris Hall G01

Prelim: October 9, 2025 7:30 PM ET **Makeup P**

Final: TBD **Makeup F**

Week	Date	Lecture
1	Mon 8/25	• Intro (Notes) • 1+1=2 (Notes)
	Wed 8/27	• Numbers (Notes) • C Intro (Notes)

Lab 1:

Resources

RISC-V Infrastructure

[RISC-V Infrastructure](#)

Tools

- [Unix Shell](#)
- [Git](#)
- [SSH](#)
- [Makefiles](#)

C Programming

- [Compilation](#)
- [Language Basics](#)
- [Basic Types](#)
- [Prototypes & Headers](#)
- [Control Flow](#)
- [Declared Types](#)
- [Bit Packing](#)
- [Pointers](#)
- [Arrays](#)
- [Strings](#)
- [Macros](#)
- [Memory Allocation](#)

RISC-V Assembly

- [RISC-V Assembly](#)
- [RISC-V ISA Reference](#)



RISC-V ISA Reference

Instruction Encoding Templates

	31	25	24	20	19	15	14	12	11	7	6	0
R-type	funct7			rs2		rs1		funct3		rd		opcode
I-type	imm[11:0]				rd							
S-Type	imm[11:5]			rs2						imm[4:0]		
B-type	imm[12 10:5]					imm [4:1 11]						
J-type	imm[20 10:1 11 19:12]								rd			
U-type	imm[31:12]											

RV32I Base Integer Instruction Set

Arithmetic Instructions

Instruction	Name	Description	Opcode	Funct3	Funct7
<code>add rd rs1 rs2</code>	ADD	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000

The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>

Instruction	Name	Description	Opcode	Funct 3	Funct7
<code>add rd rs1 rs2</code>	<i>ADD</i>	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
<code>sub rd rs1 rs2</code>	<i>SUBTRACT</i>	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

	31	25	24	20	19	15	14	12	11	7	6	0
R-type	<i>funct7</i>				<i>rs2</i>	<i>rs1</i>	funct 3		<i>rd</i>	opcode		

The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>

Instruction	Name	Description	Opcode	Funct 3	Funct7
<code>add rd rs1 rs2</code>	<i>ADD</i>	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
<code>sub rd rs1 rs2</code>	<i>SUBTRACT</i>	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

	31	25	24	20	19	15	14	12	11	7	6	0
R-type	<i>funct7</i>				<i>rs2</i>	<i>rs1</i>	<i>funct 3</i>	<i>rd</i>	<i>opcode</i>			

Assembling machine code



The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>

Instruction	Name	Description	Opcode	Funct 3	Funct7
<code>add rd rs1 rs2</code>	<i>ADD</i>	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
<code>sub rd rs1 rs2</code>	<i>SUBTRACT</i>	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

31	25	24	20	19	15	14	12	11	7	6	0
R-type		funct7	rs2	rs1	funct 3		rd	opcode			

Assembling machine code

1. Fill in constants.

The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>

Instruction	Name	Description	Opcode	Funct 3	Funct7
<code>add rd rs1 rs2</code>	<i>ADD</i>	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
<code>sub rd rs1 rs2</code>	<i>SUBTRACT</i>	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

31	25	24	20	19	15	14	12	11	7	6	0
R-type	funct7	rs2	rs1	funct 3	rd	opcode					

Assembling machine code

1. Fill in constants.
2. Read + write register numbers: 0..31
(in binary!)

The RISC-V Add Instruction: 2nd Try

<https://www.cs.cornell.edu/courses/cs3410/2025fa/>

Instruction	Name	Description	Opcode	Funct 3	Funct7
<code>add rd rs1 rs2</code>	<i>ADD</i>	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
<code>sub rd rs1 rs2</code>	<i>SUBTRACT</i>	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

	31	25	24	20	19	15	14	12	11	7	6	0
R-type	<i>funct7</i>				<i>rs2</i>	<i>rs1</i>	funct 3		<i>rd</i>	opcode		

Semantics: How do I execute the instruction?

The RISC-V Add Instruction: 2nd Try

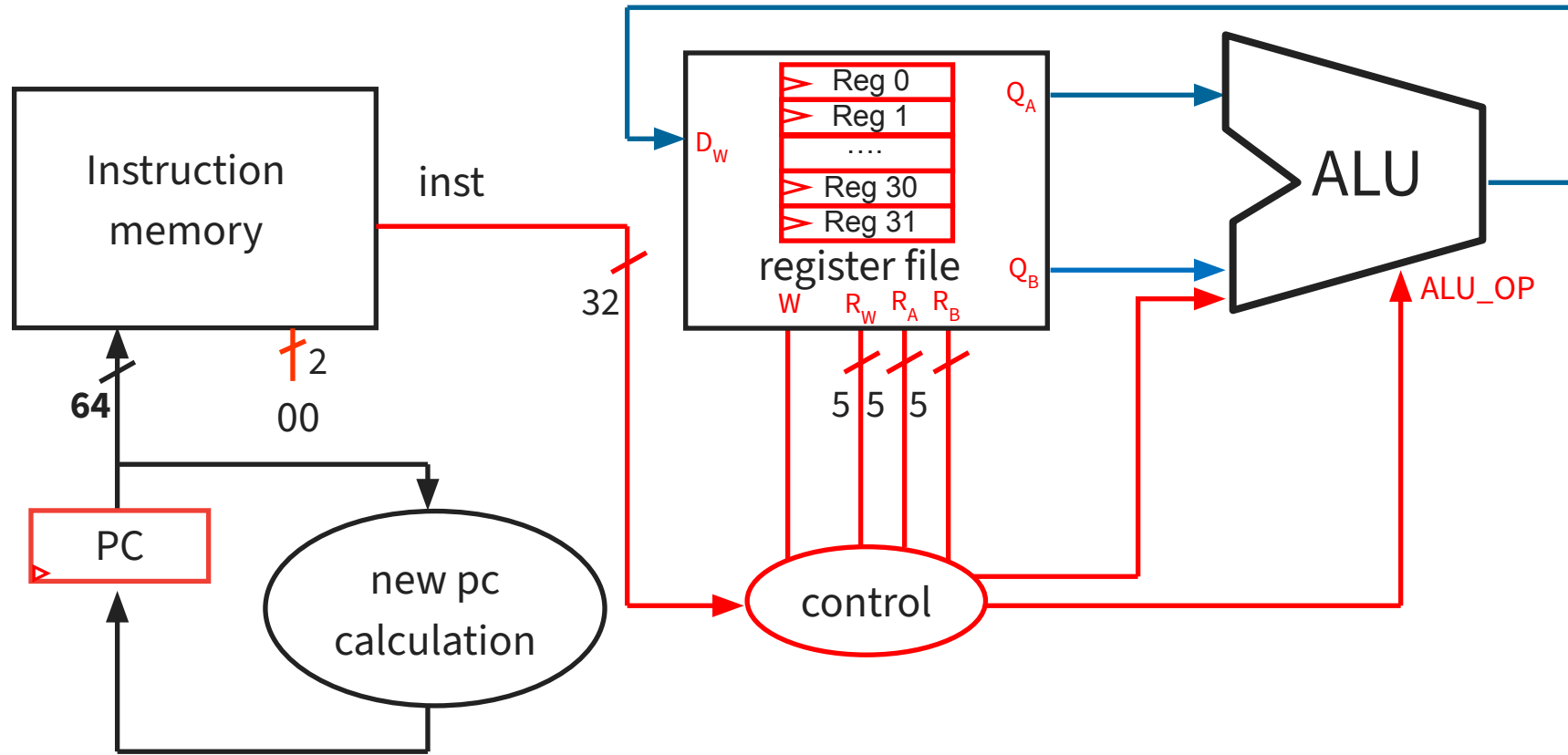
<https://www.cs.cornell.edu/courses/cs3410/2025fa/>

Instruction	Name	Description	Opcode	Funct 3	Funct7
<code>add rd rs1 rs2</code>	<i>ADD</i>	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
<code>sub rd rs1 rs2</code>	<i>SUBTRACT</i>	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

	31	25	24	20	19	15	14	12	11	7	6	0
R-type	<i>funct7</i>				<i>rs2</i>	<i>rs1</i>	funct 3		<i>rd</i>	opcode		

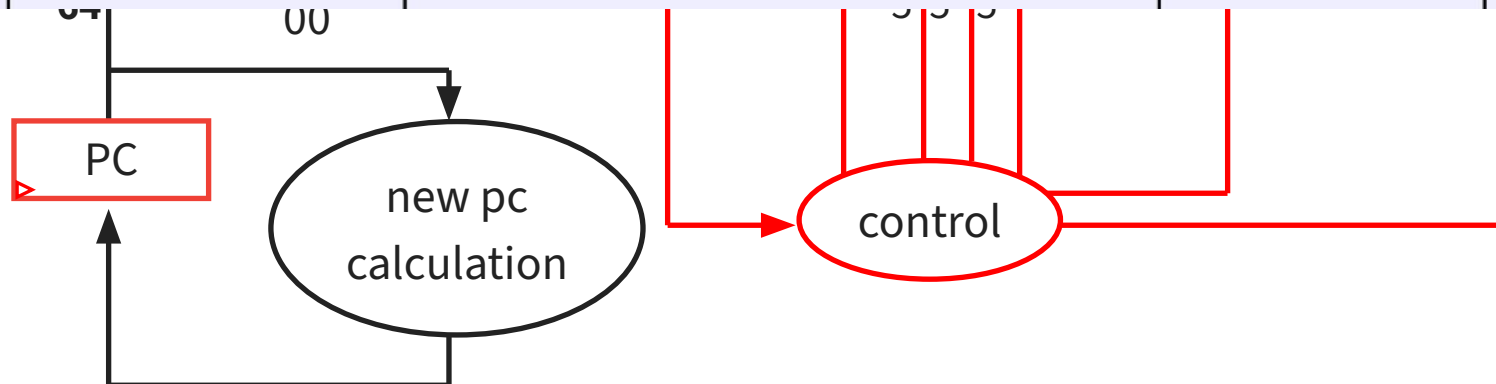
Semantics: How do I execute the instruction?

Executing `add x10, x10, x11`



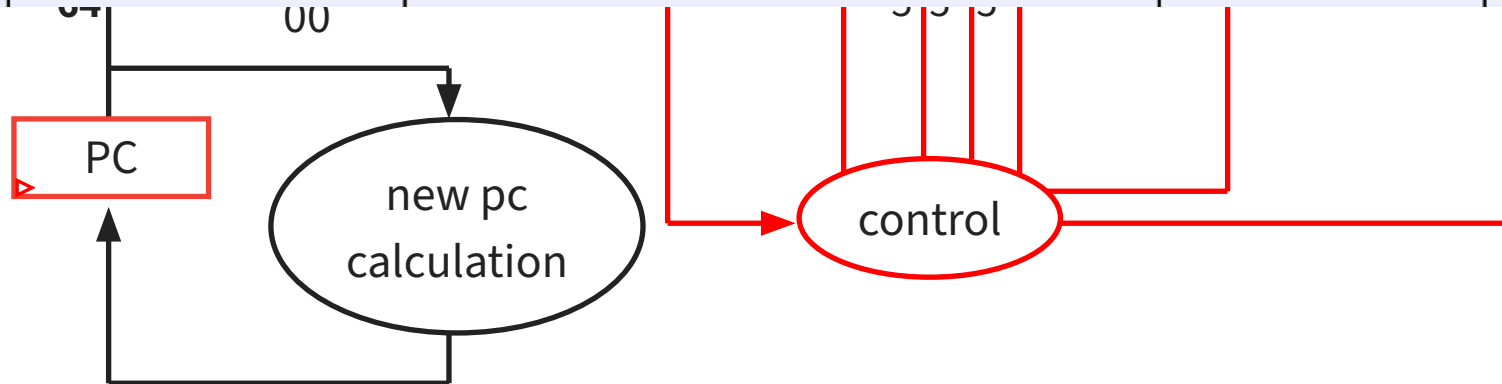
Executing `add x10, x10, x11`

Instruction	Name	Description	Opcode	Funct3	Funct7
<code>add rd rs1 rs2</code>	<i>ADD</i>	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
<code>sub rd rs1 rs2</code>	<i>SUBTRACT</i>	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000



Executing **add** **x10**, **x10**, **x11**

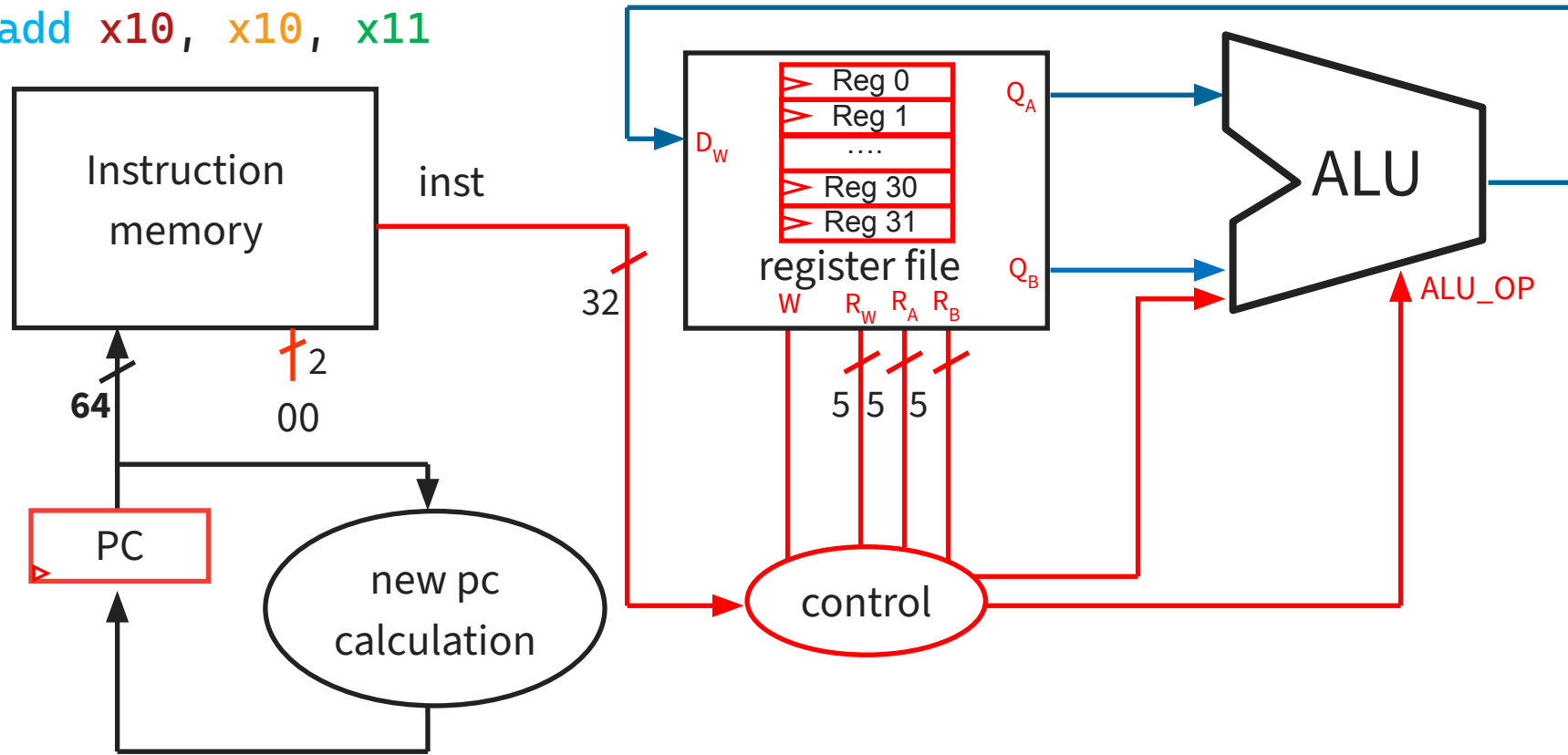
Instruction	Name	Description	Opcode	Funct3	Funct7
<code>add rd rs1 rs2</code>	<i>ADD</i>	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
<code>sub rd rs1 rs2</code>	<i>SUBTRACT</i>	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

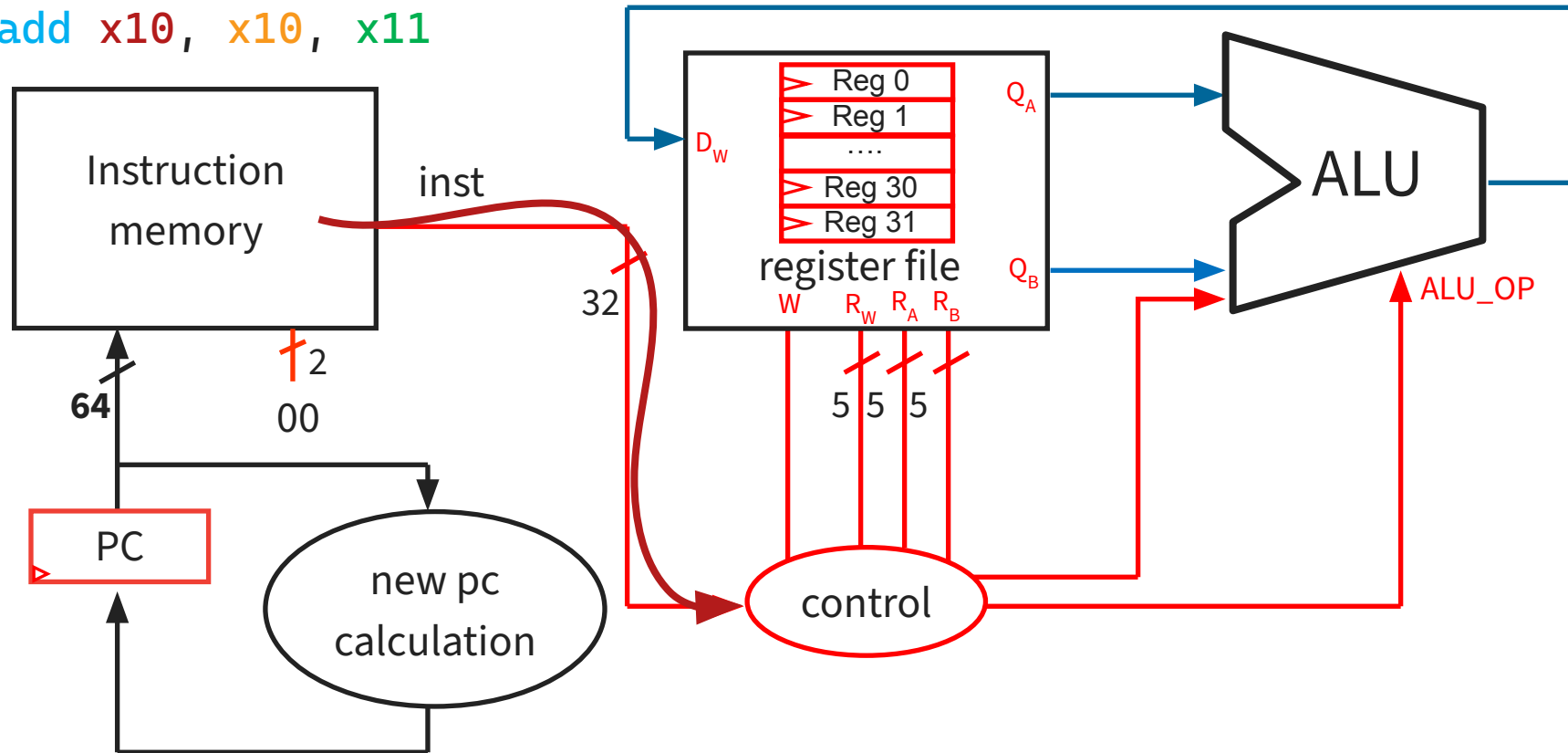
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

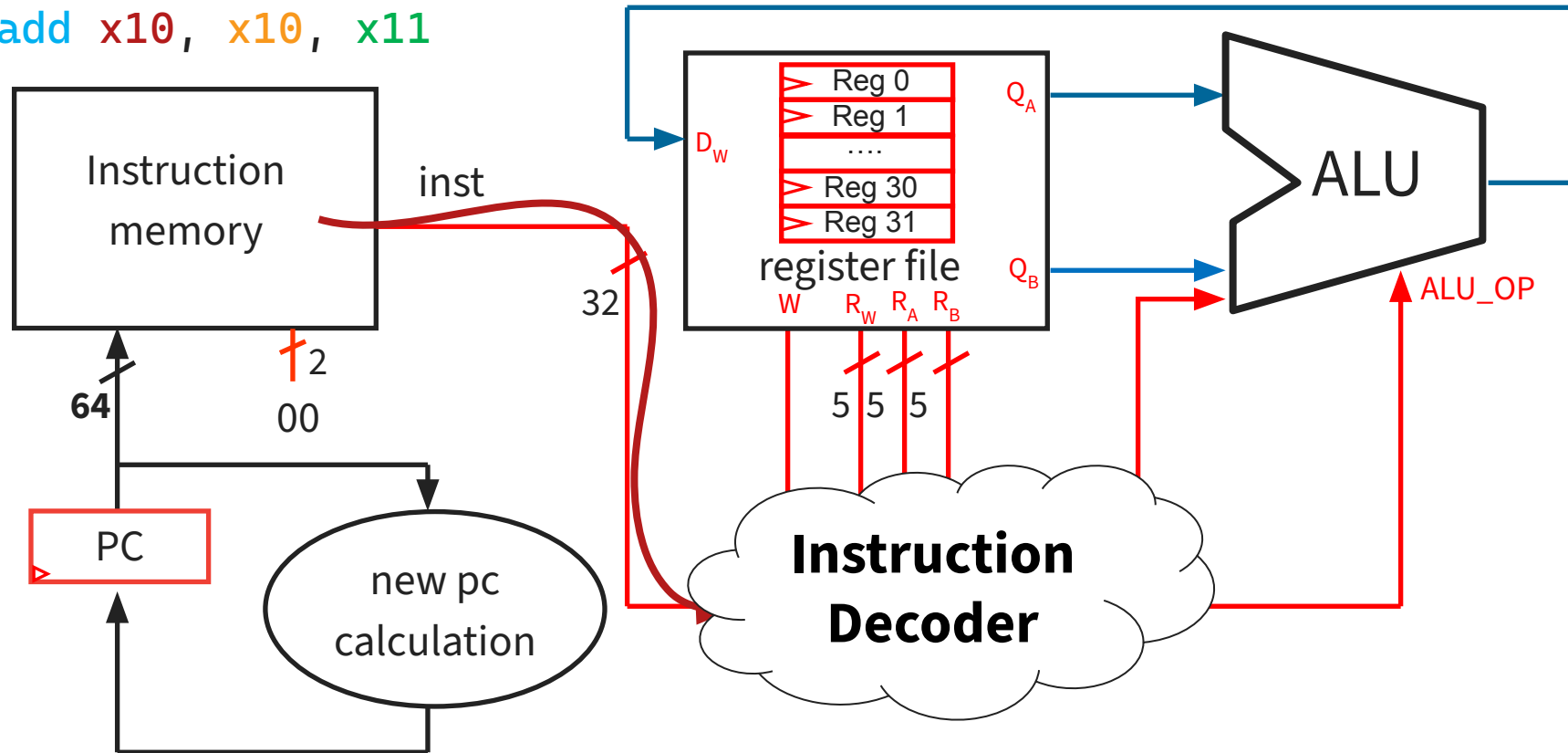
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

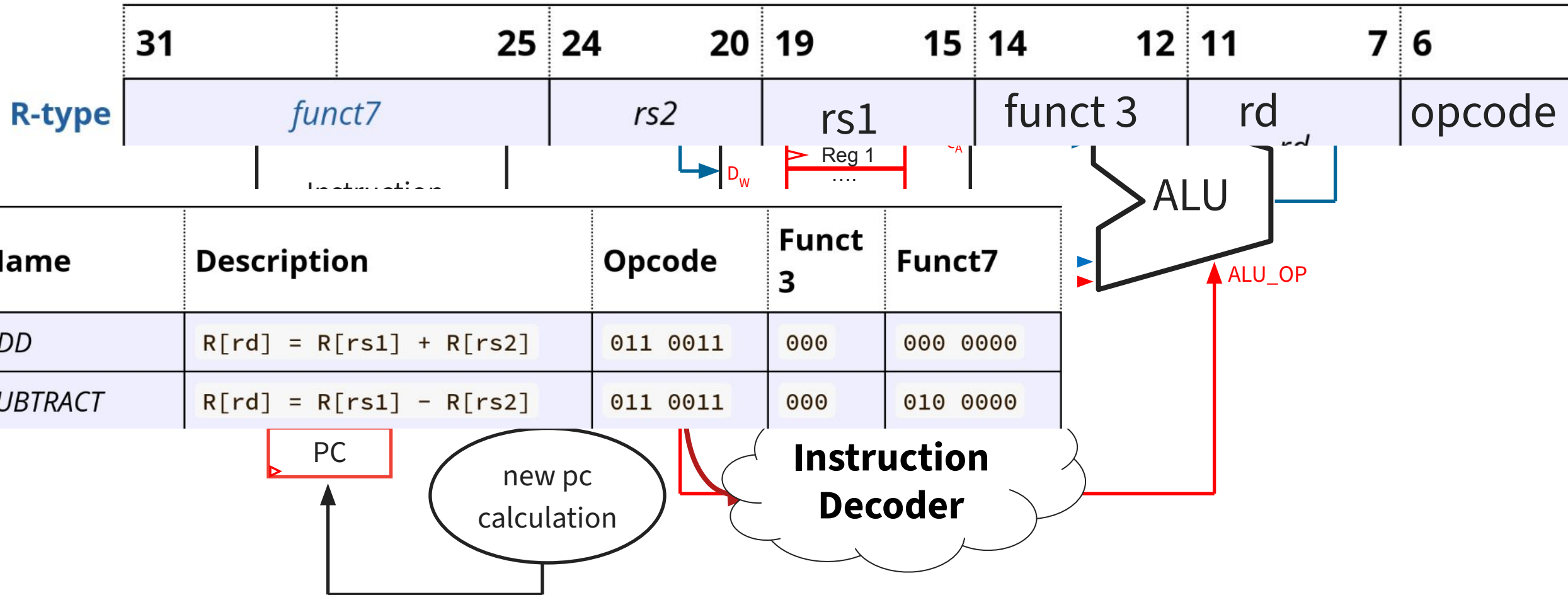
Executing `add x10, x10, x11`

1. `add x10, x10, x11`



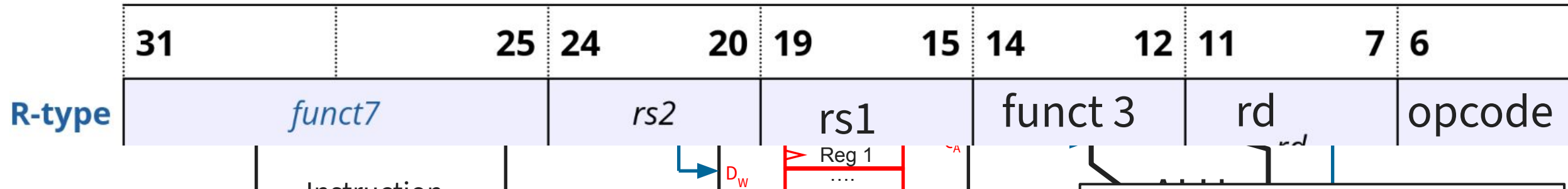
Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

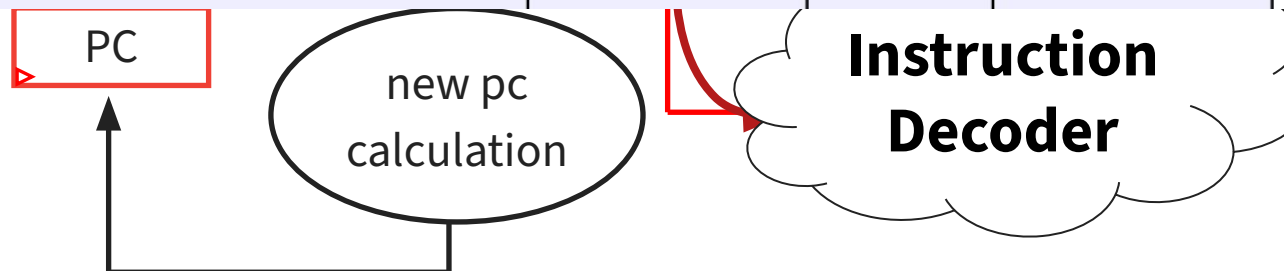


Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`



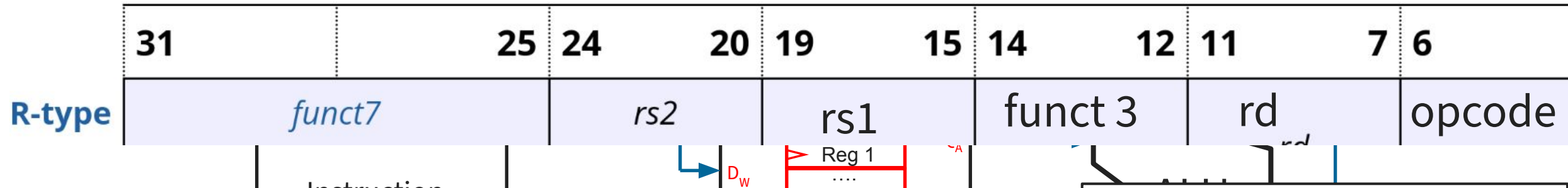
Name	Description	Opcode	Funct3	Funct7
ADD	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
SUBTRACT	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000



Recognize add

Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

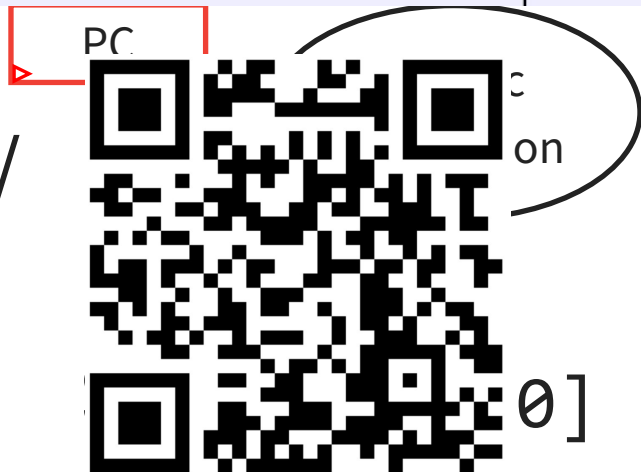


Name	Description	Opcode	Funct 3	Funct7
ADD	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
SUBTRACT	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

Recognize add

Question:
Which bits of the instruction do we need to look at to tell that it is an "R-type" add instruction?

PollEv.com/
cs3410

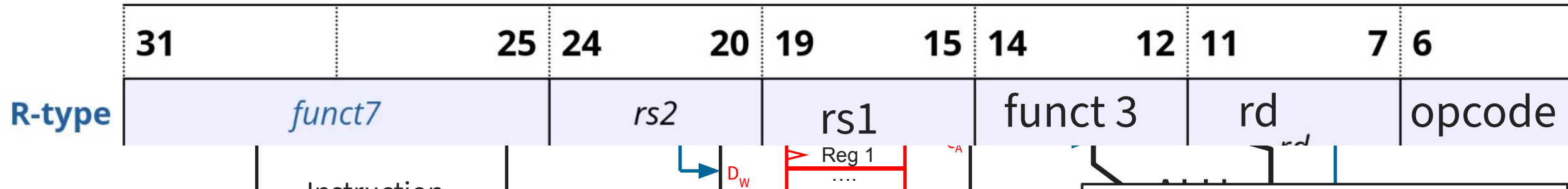


Instruction Decoder

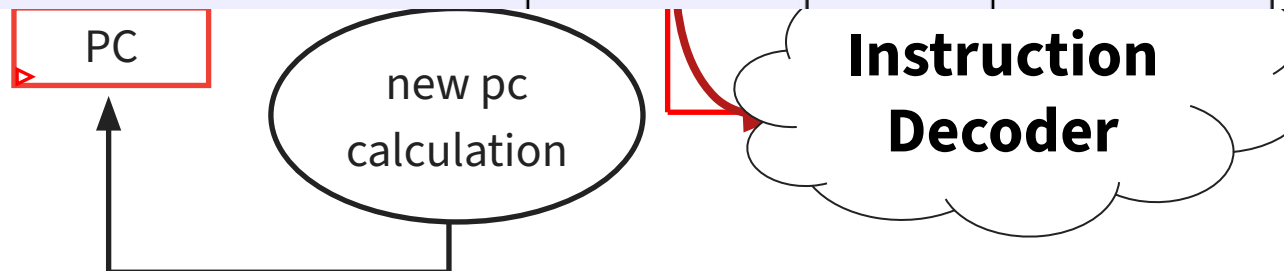
$$R[rd] = R[10] + R[11]$$



Executing `add x10, x10, x11`



Name	Description	Opcode	Funct3	Funct7
ADD	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
SUBTRACT	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

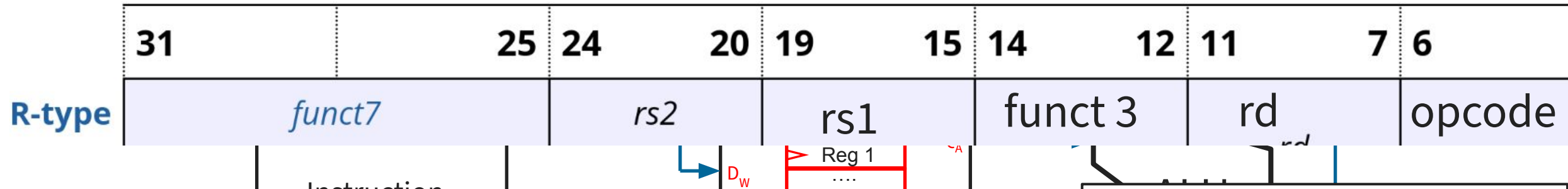


Recognize add

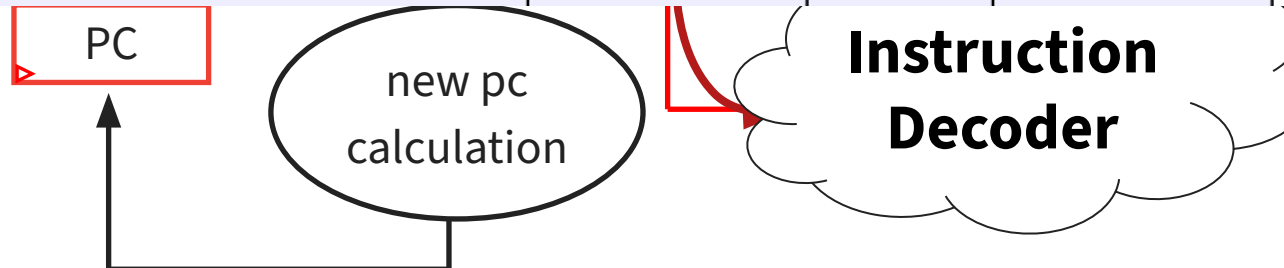
- $inst[31:25] = funct7 = 0$

Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`



Name	Description	Opcode	Funct 3	Funct7
ADD	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
SUBTRACT	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

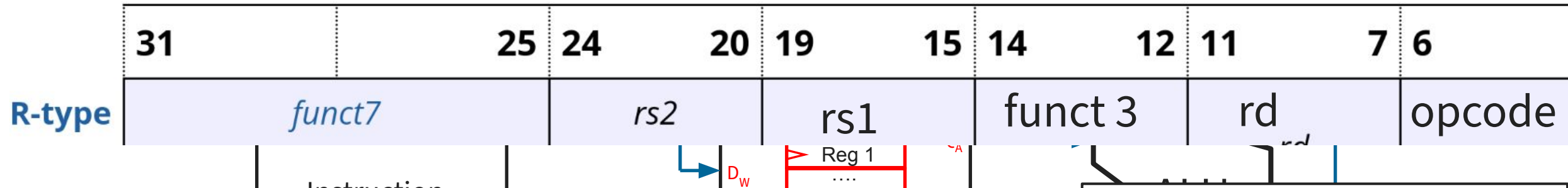


Recognize add

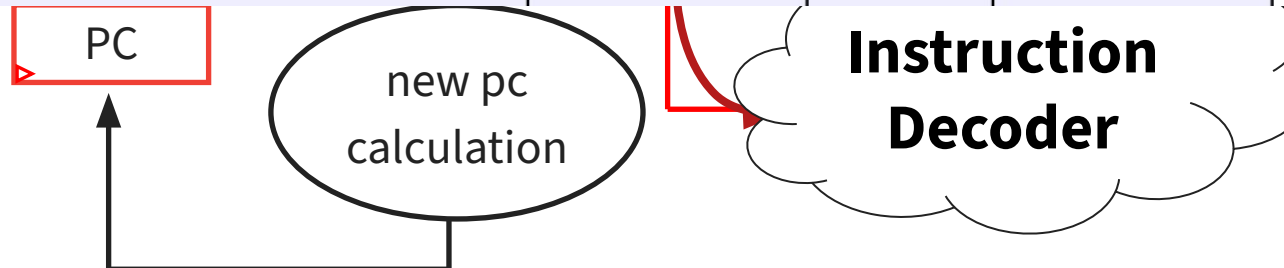
- $inst[31:25] = funct7 = 0$
- $inst[14:12] = funct3 = 0$

Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`



Name	Description	Opcode	Funct3	Funct7
ADD	$R[rd] = R[rs1] + R[rs2]$	011 0011	000	000 0000
SUBTRACT	$R[rd] = R[rs1] - R[rs2]$	011 0011	000	010 0000

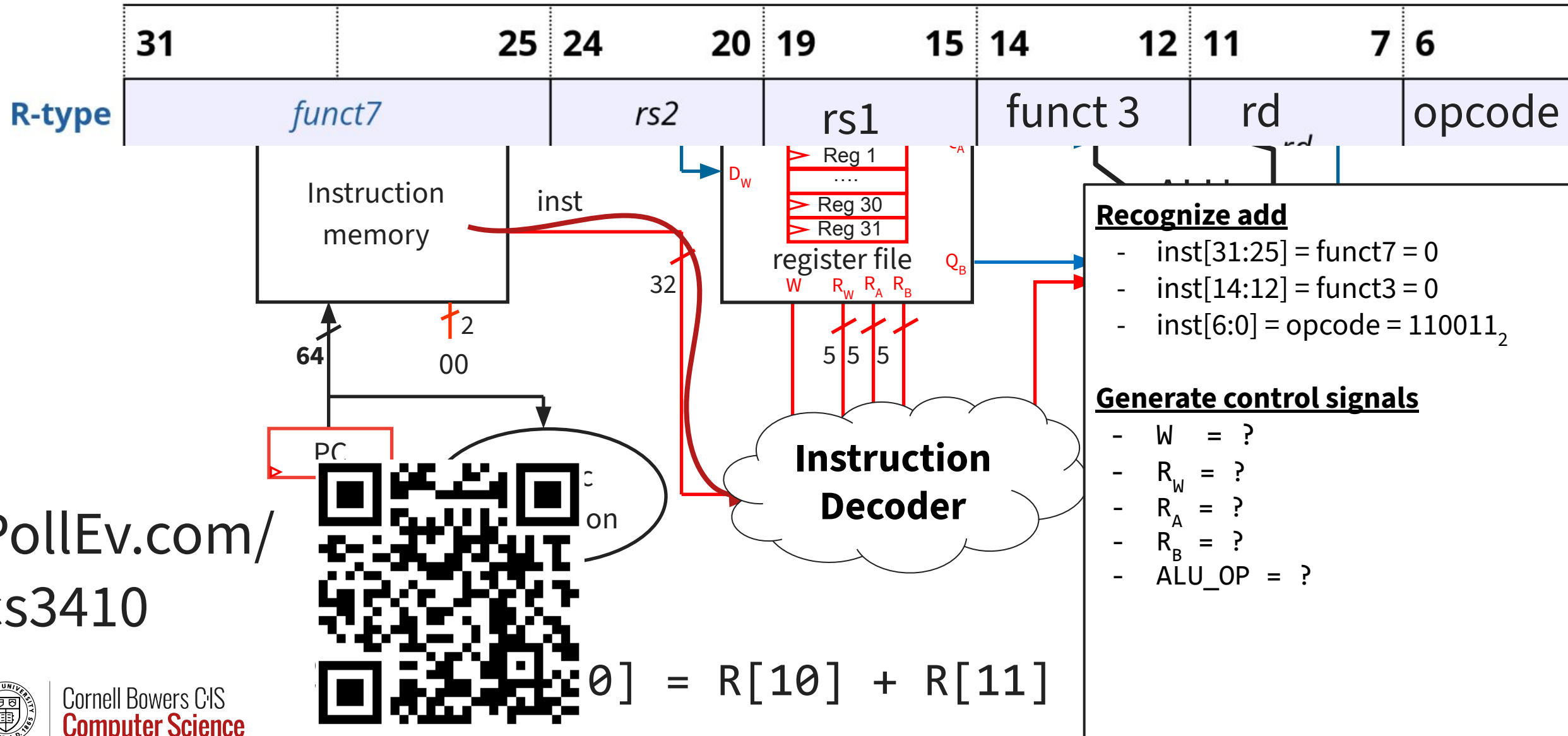


Recognize add

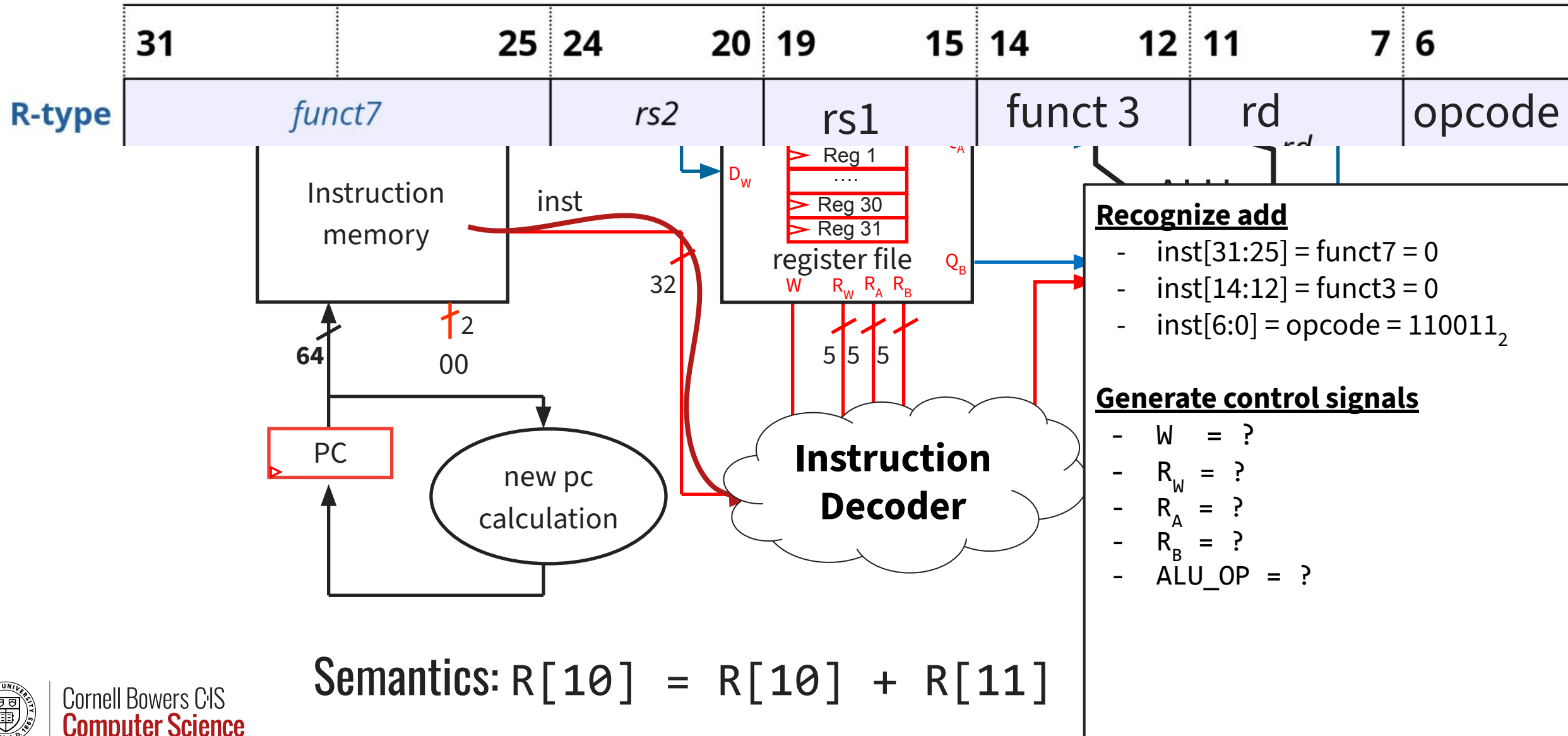
- $inst[31:25] = funct7 = 0$
- $inst[14:12] = funct3 = 0$
- $inst[6:0] = opcode = 110011_2$

Semantics: $R[10] = R[10] + R[11]$

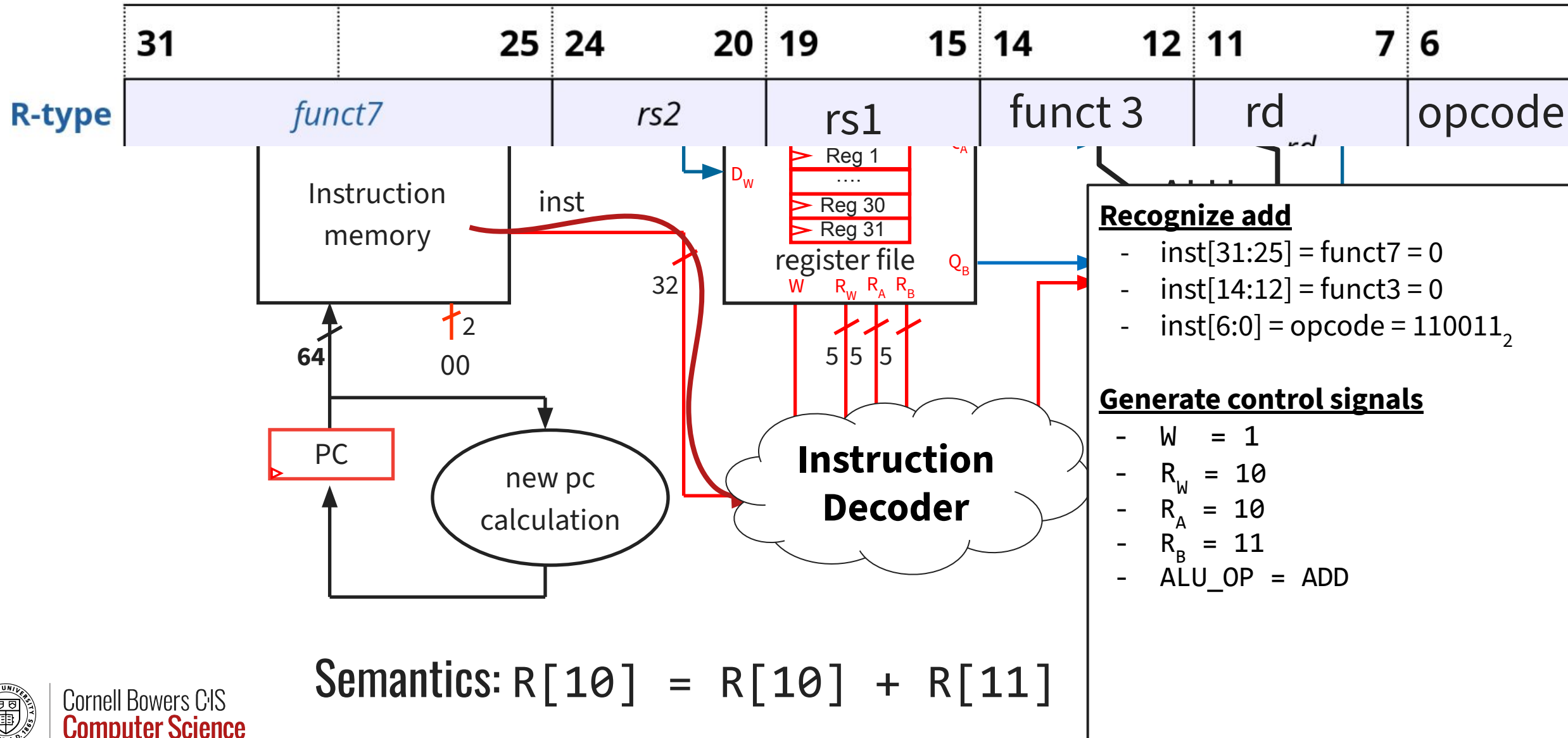
Executing `add x10, x10, x11`



Executing `add x10, x10, x11`

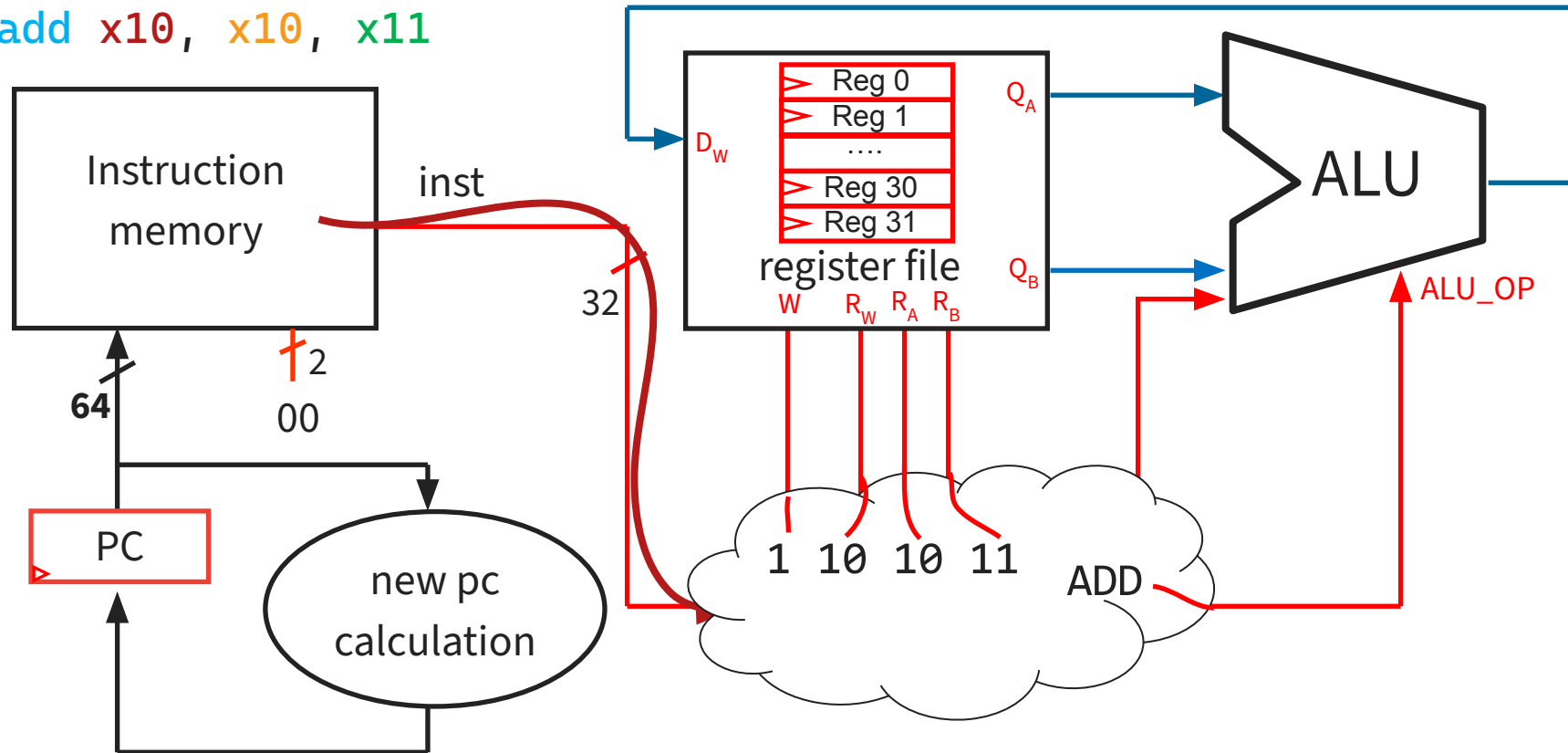


Executing `add x10, x10, x11`



Executing `add x10, x10, x11`

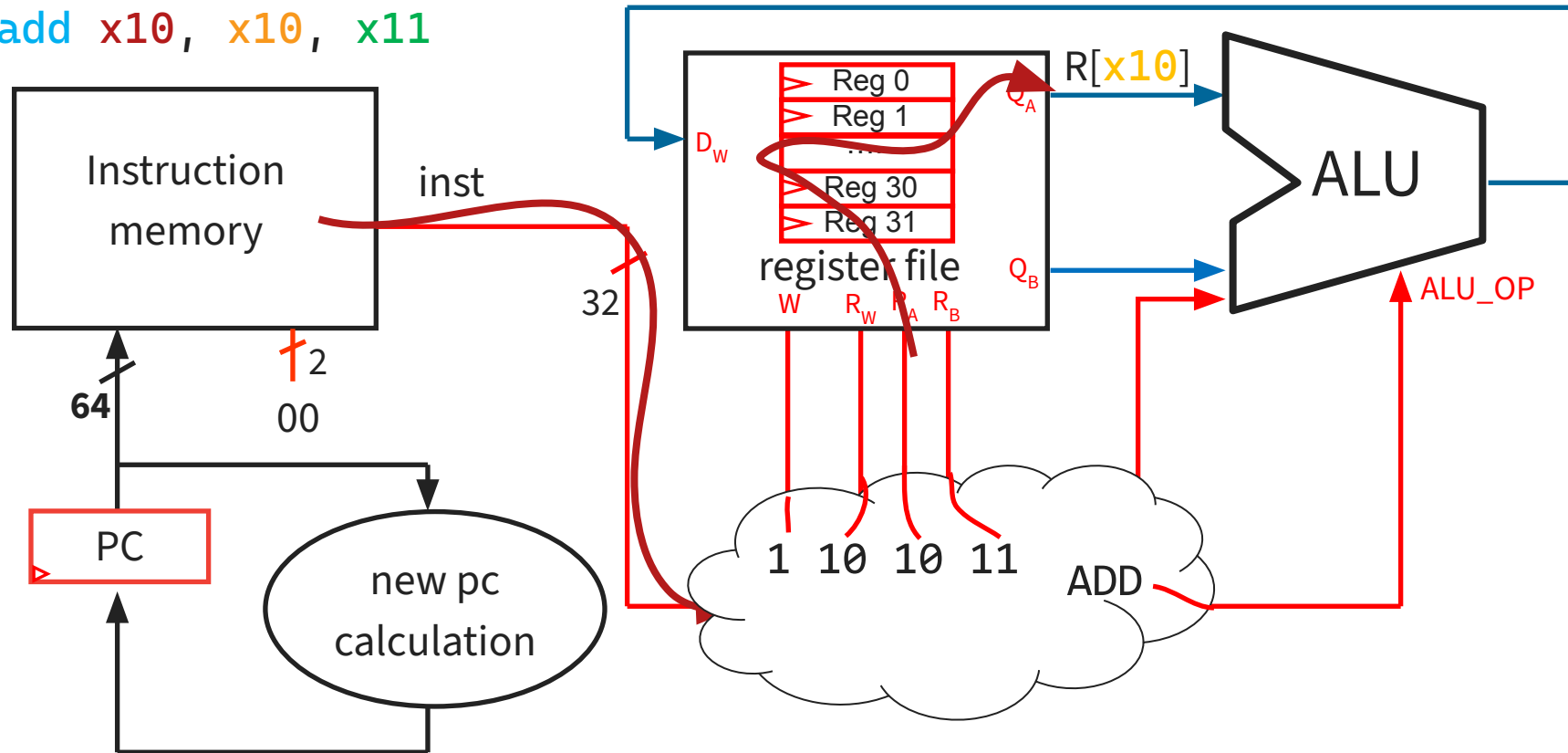
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

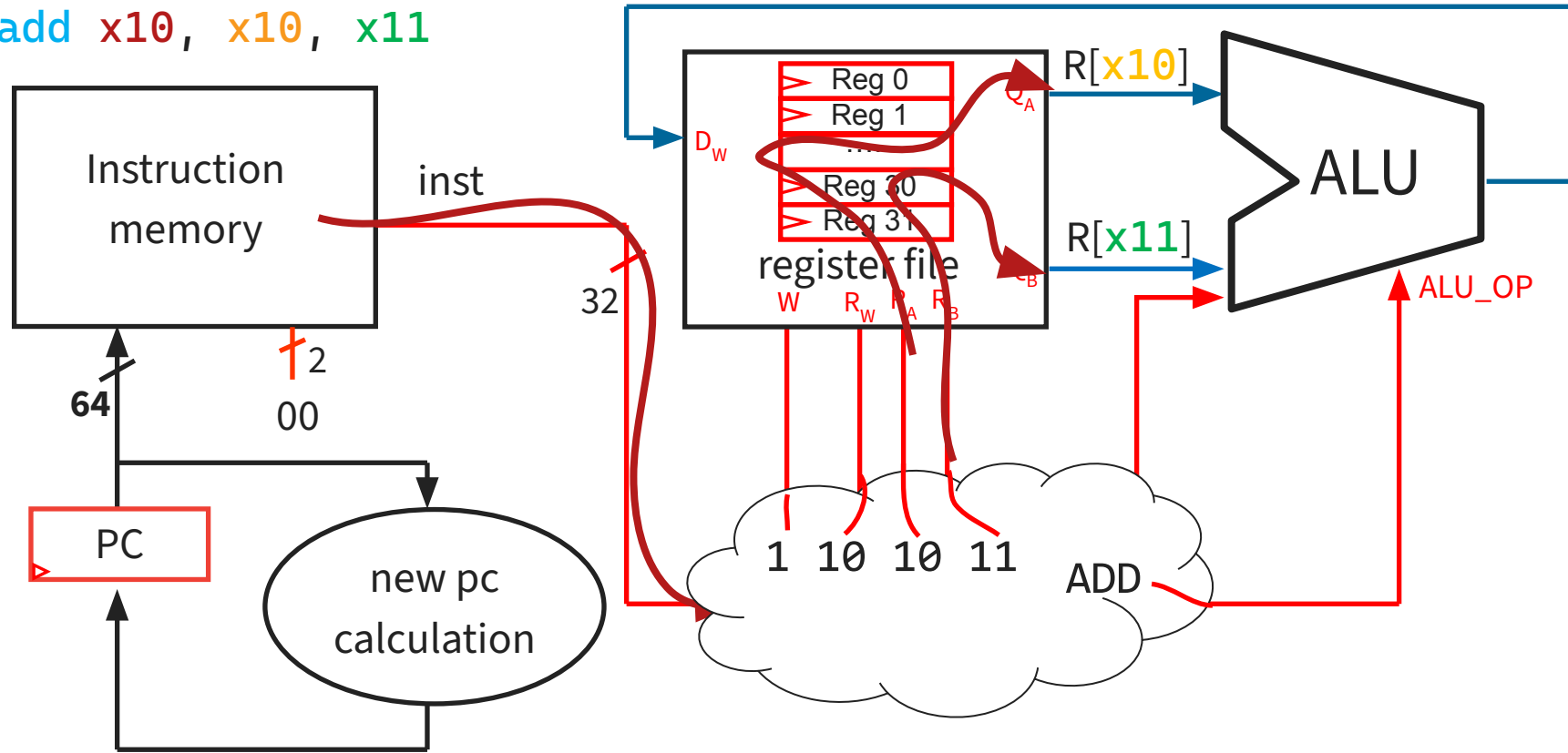
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

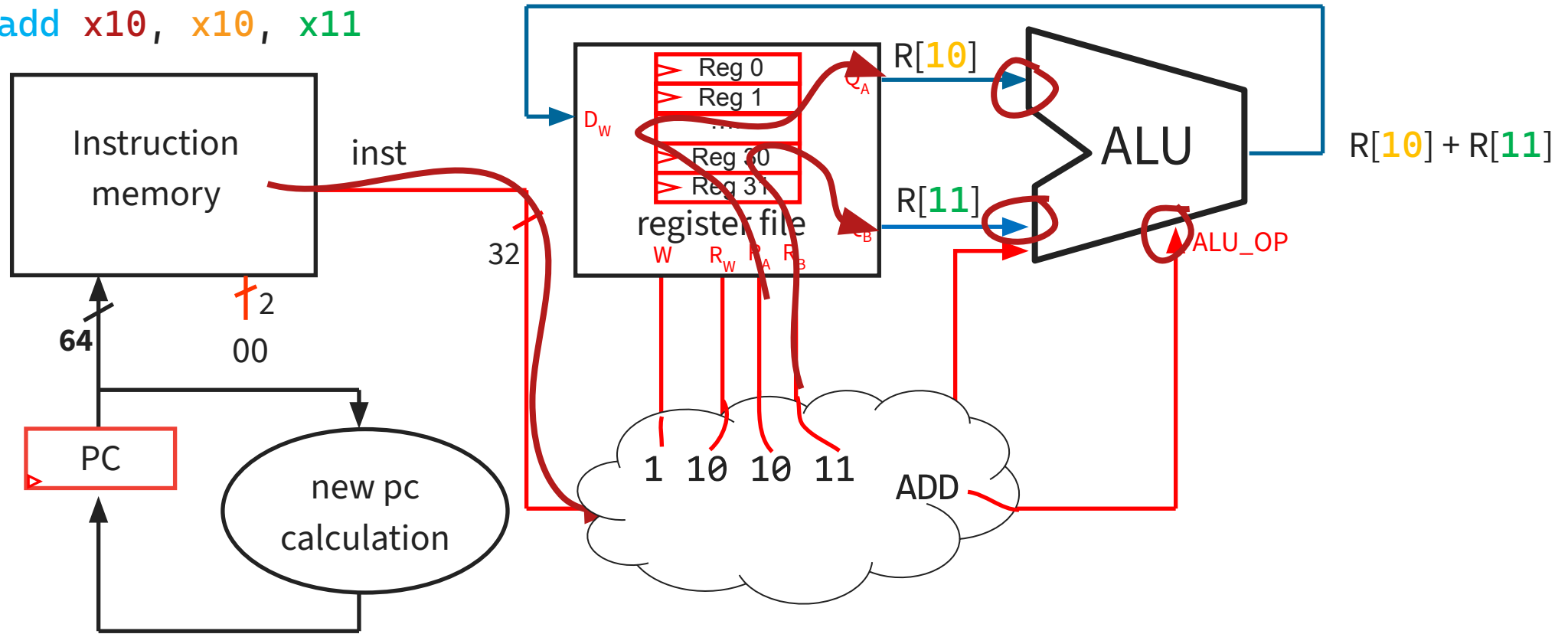
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

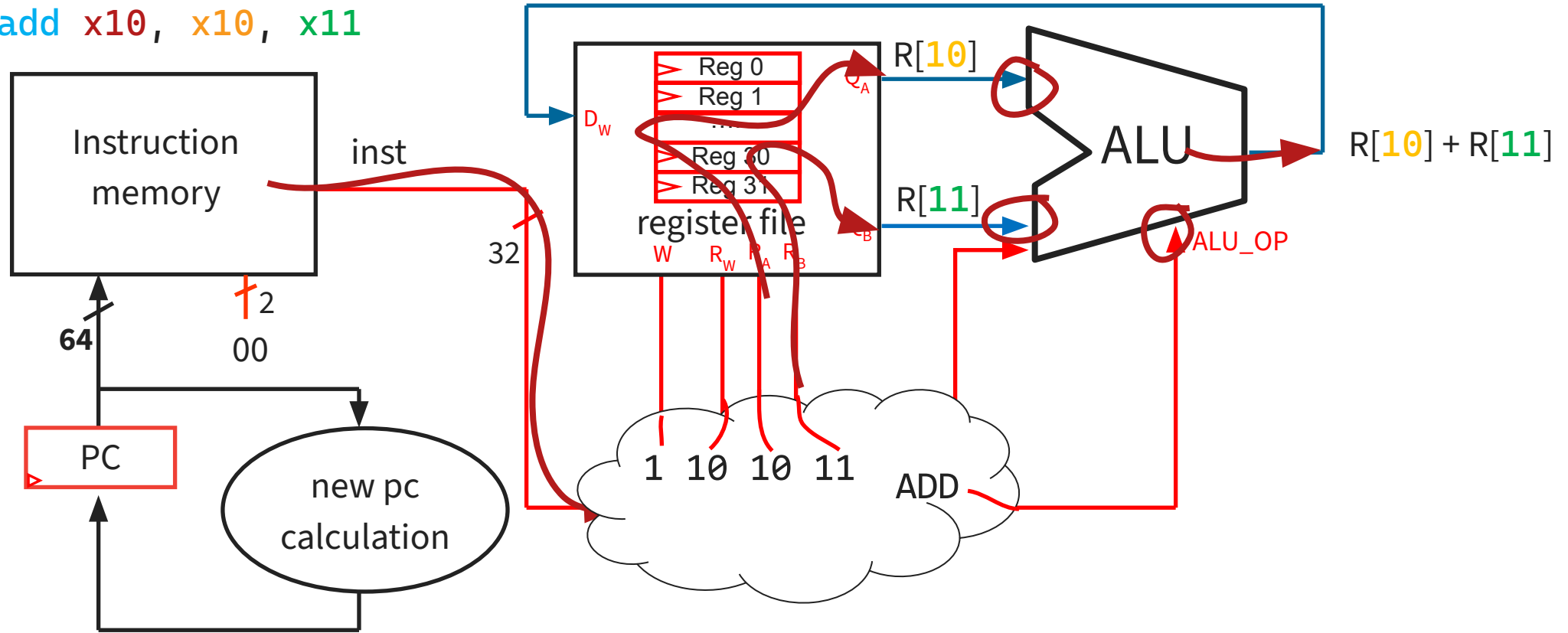
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

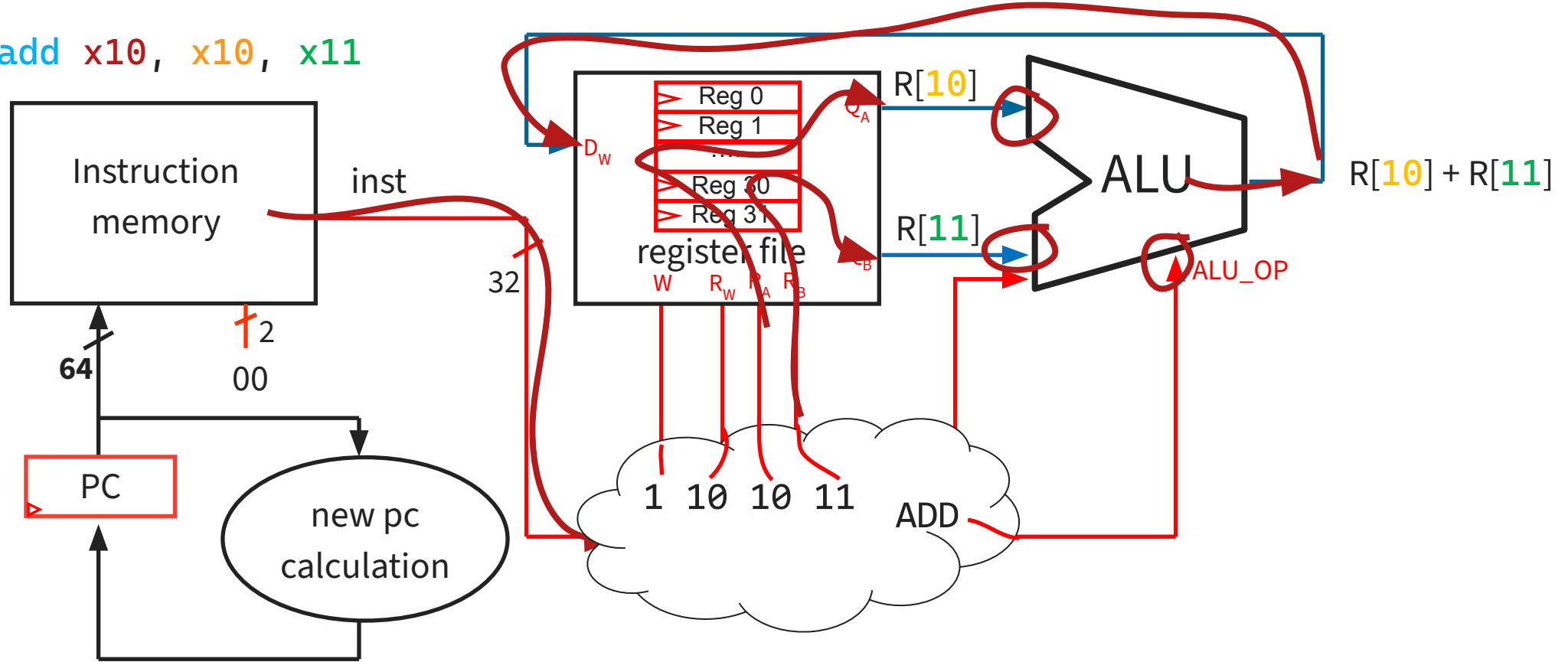
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

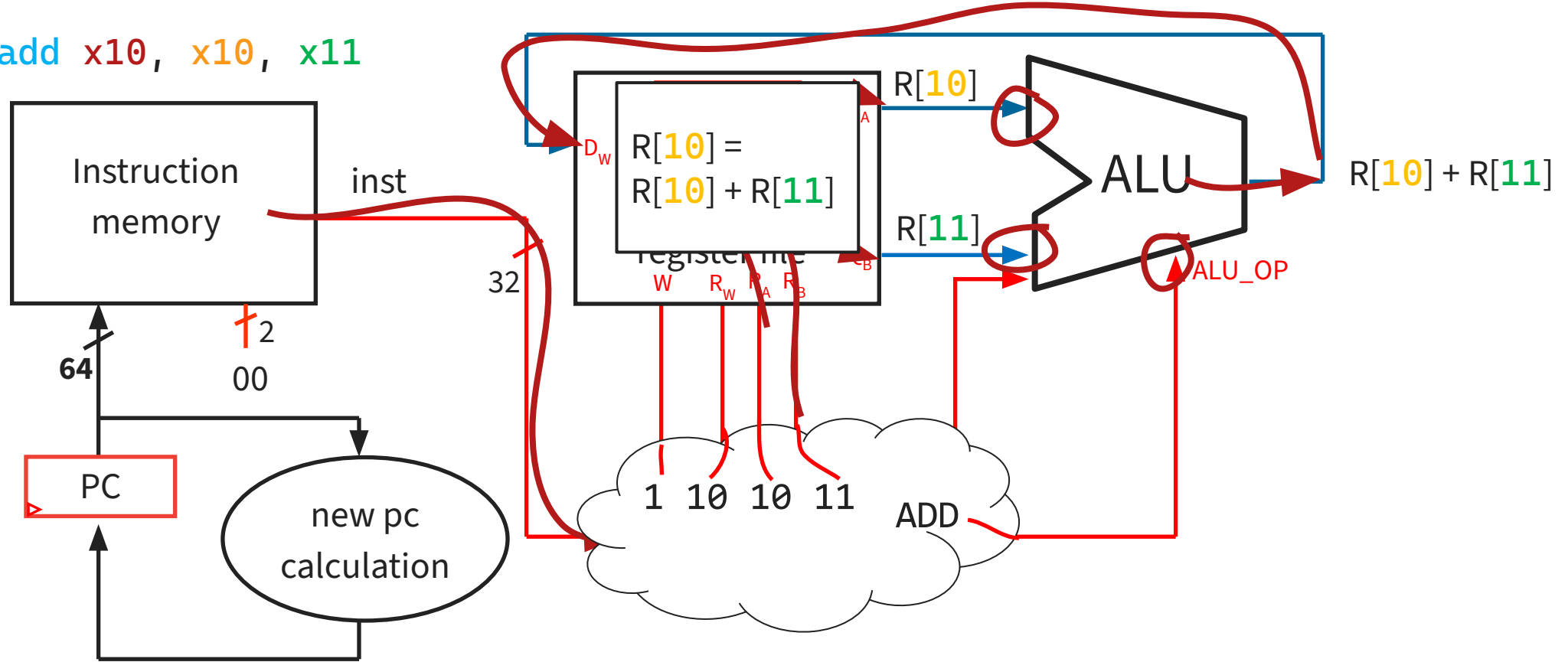
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

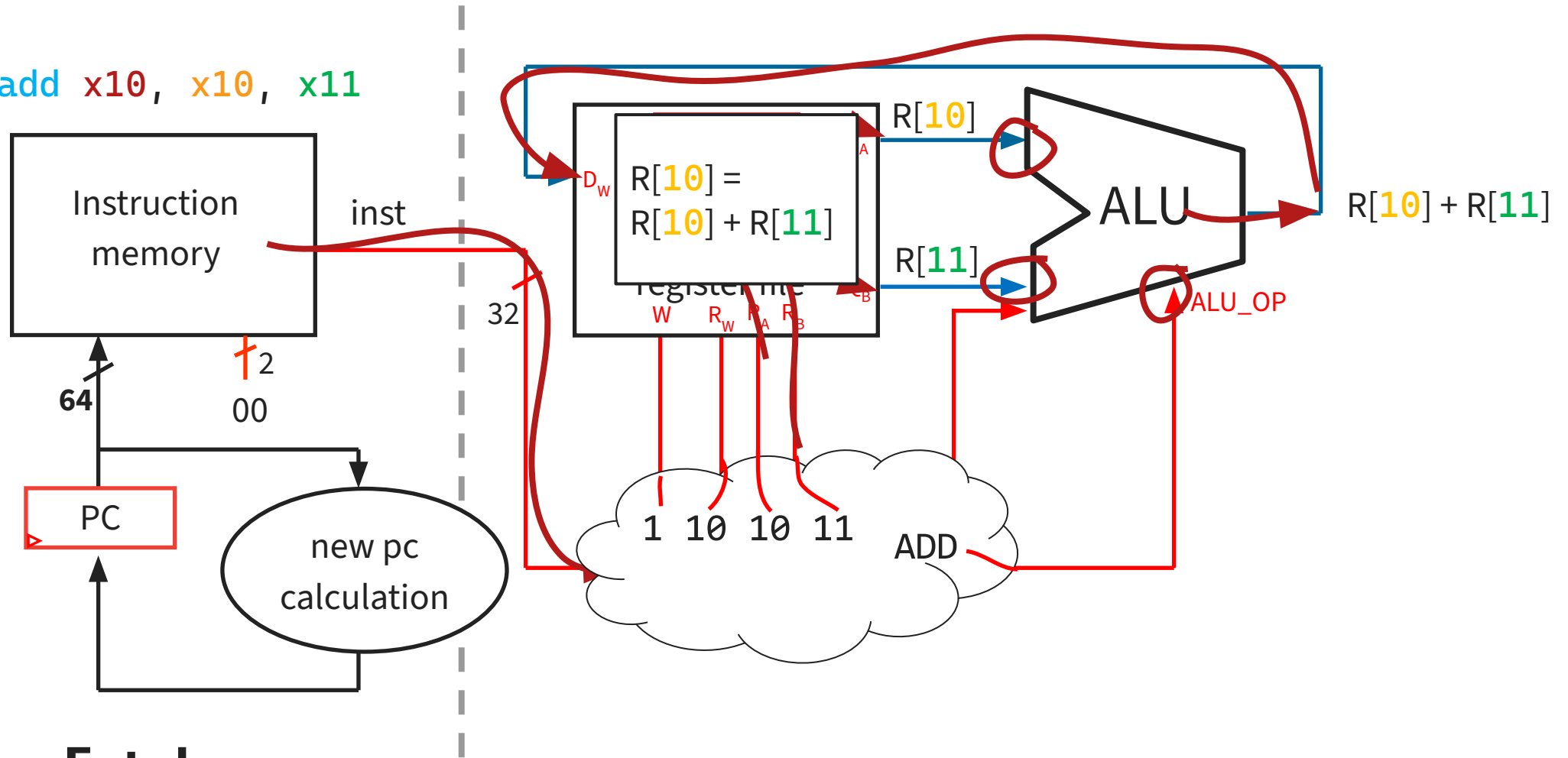
1. `add x10, x10, x11`



Semantics: $R[10] = R[10] + R[11]$

Executing `add x10, x10, x11`

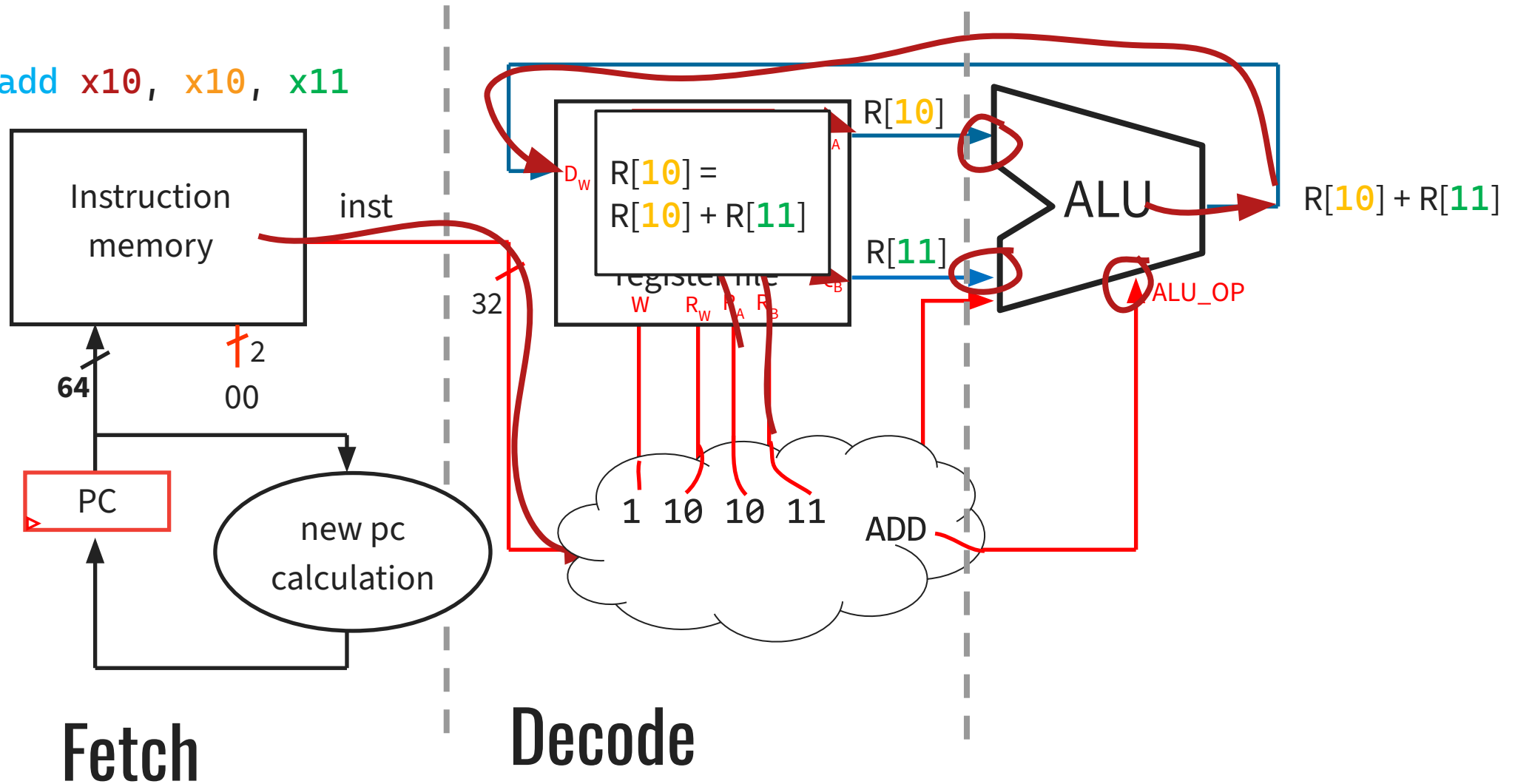
1. `add x10, x10, x11`



Fetch

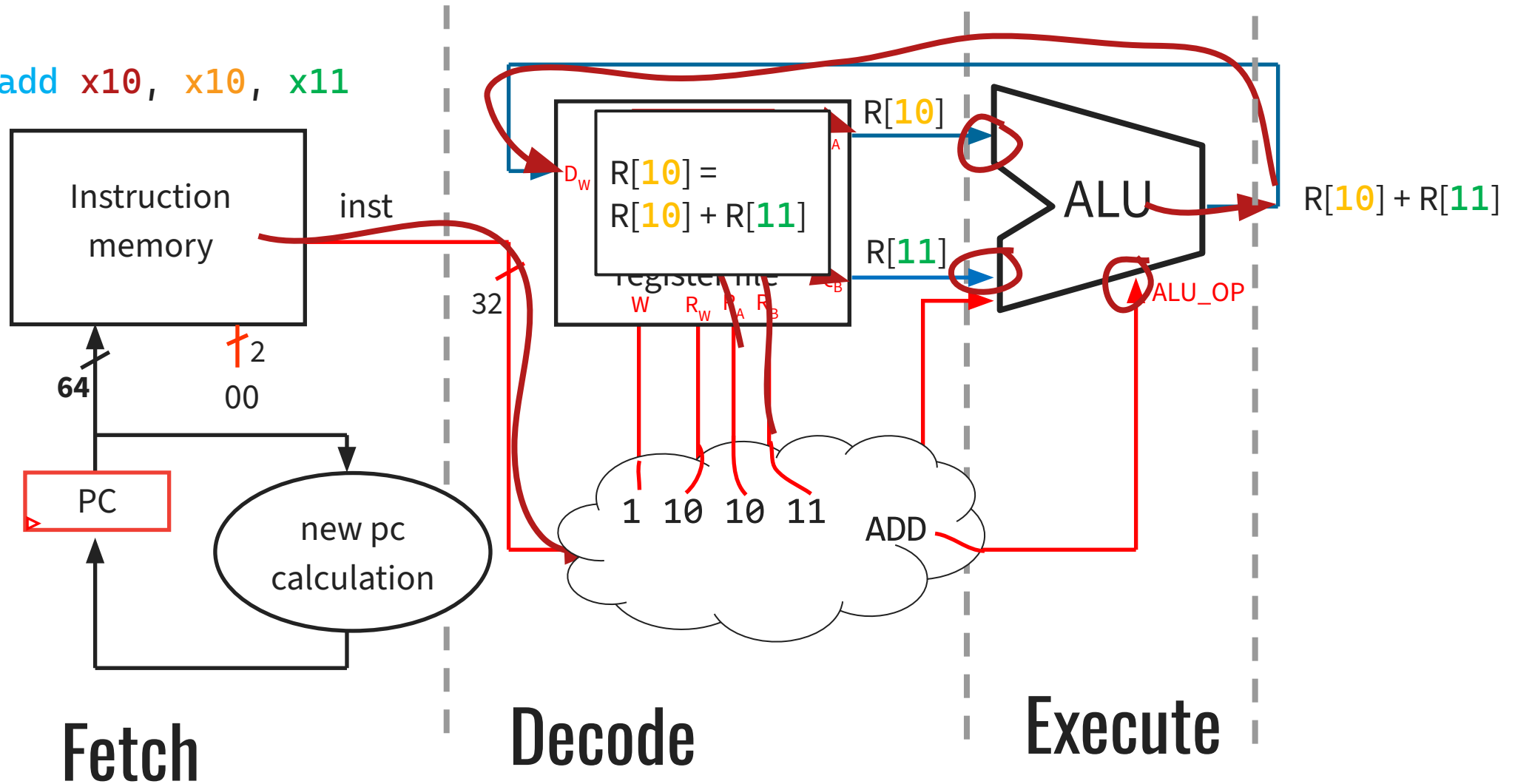
Executing `add x10, x10, x11`

1. `add x10, x10, x11`



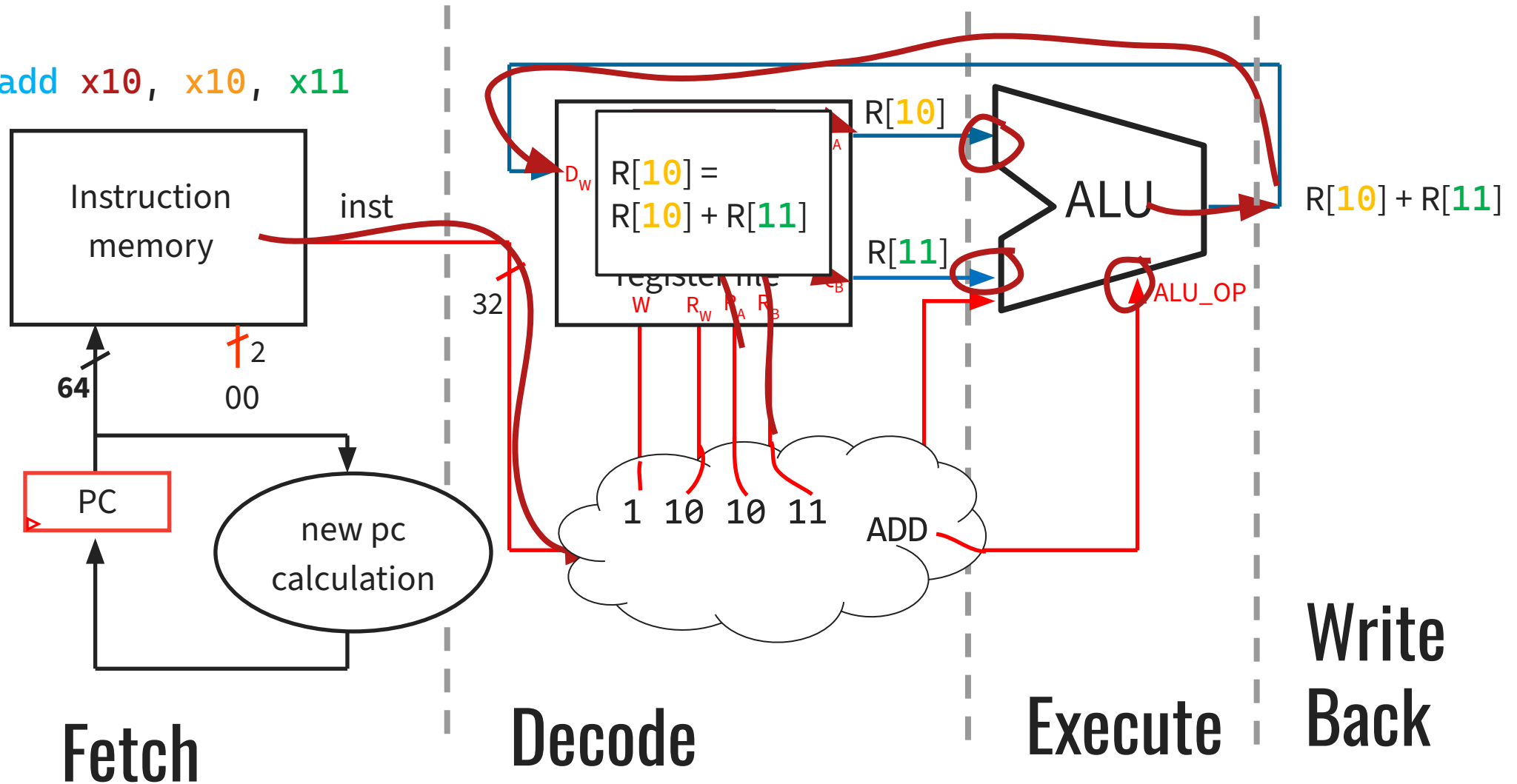
Executing `add x10, x10, x11`

1. `add x10, x10, x11`



Executing `add x10, x10, x11`

1. `add x10, x10, x11`



Instruction

add x10, x10, x11

Instruction

Operands

add x10, x10, x11

Instruction

Destination

Operands

add x10, x10, x11

Instruction

Destination

Operands

add x10, x10, x11

Can reuse registers as source and destination

Registers

- RISC-V has 32 registers
 - Each stores 64-bit integers

Register name	Symbolic name	Description
32 integer registers		
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments / return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

Registers

- RISC-V has 32 registers
 - Each stores 64-bit integers
- Different registers are used for different purposes

Register name	Symbolic name	Description
32 integer registers		
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments / return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

Registers

- RISC-V has 32 registers
 - Each stores 64-bit integers
- Different registers are used for different purposes
 - mostly just by convention

Register name	Symbolic name	Description
32 integer registers		
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments / return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

Registers

- RISC-V has 32 registers
 - Each stores 64-bit integers
- Different registers are used for different purposes
 - mostly just by convention
 - will cover details when discussing “calling convention”

Register name	Symbolic name	Description
32 integer registers		
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments / return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

Registers

- RISC-V has 32 registers
 - Each stores 64-bit integers
- Different registers are used for different purposes
 - mostly just by convention
 - will cover details when discussing “calling convention”
 - **x0** is always zero!

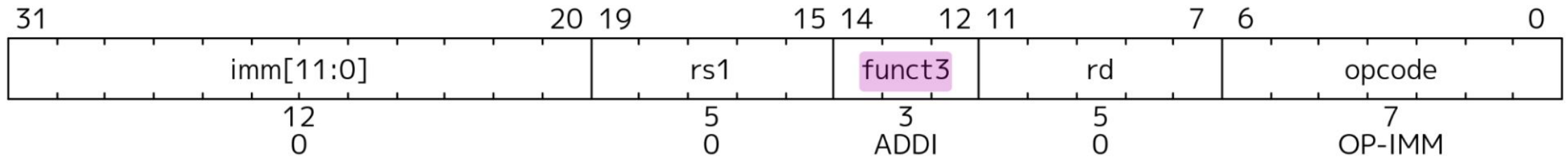
Register name	Symbolic name	Description
32 integer registers		
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function arguments / return values
x12–17	a2–7	Function arguments
x18–27	s2–11	Saved registers
x28–31	t3–6	Temporaries

Registers

- RISC-V has 32 registers

Register name	Symbolic name	Description
32 integer registers		
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer

2.4.3. NOP Instruction



The NOP instruction does not change any architecturally visible state, except for advancing the **pc** and incrementing any applicable performance counters. NOP is encoded as `ADDI x0, x0, 0`.

convention

- **x0** is always zero!

Instructions beyond “add”



Instruction Types

Arithmetic

- add, subtract, shift left, shift right, multiply, divide

Memory

- load value from memory to a register
- store value to memory from a register

Control flow

- conditional jumps (branches)
- jump and link (subroutine call)

Many other instructions are possible

- vector add/sub/mul/div, string operations
- manipulate coprocessor
- I/O

RISC-V Instruction Types

Arithmetic/Logical

- **R-type**: result and two source registers, shift amount
- **I-type**: result and source register, shift amount in 12-bit immediate with sign/zero extension
- **U-type**: result register, 20-bit immediate with sign/zero extension

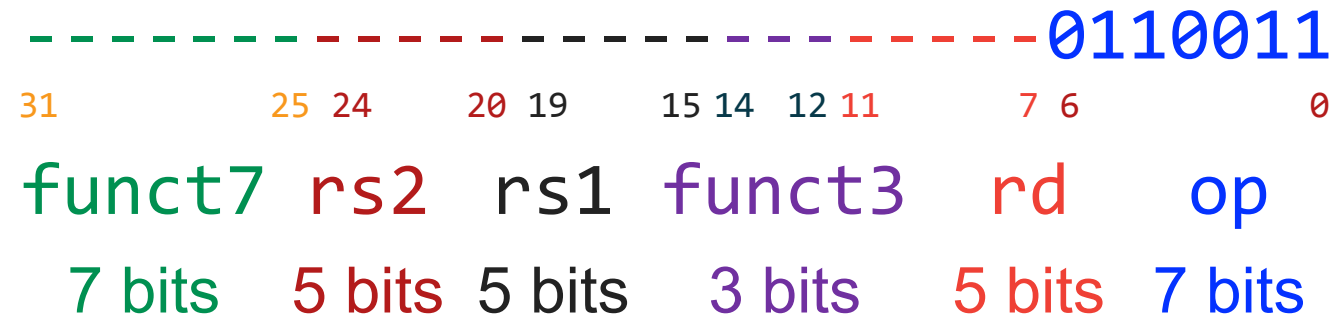
Memory Access

- **I-type** for loads and **S-type** for stores
- load/store between registers and memory
- word, half-word and byte operations

Control flow

- **S-type**: conditional branches: pc-relative addresses
- **U-type**: jump-and-link
- **I-type**: jump-and-link register

R-Type (1): Arithmetic and Logic



funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
0100000	000	SUB rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
0000000	110	OR rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
0000000	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$

R-Type (1): Arithmetic and Logic

00000000 0011001000100010000100001000110011
31 25 24 20 19 15 14 12 11 7 6 0

funct7 rs2 rs1 funct3 rd op
7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
0100000	000	SUB rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
0000000	110	OR rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
0000000	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$

R-Type (1): Arithmetic and Logic

00000000 0011001000100010000100001000110011
31 25 24 20 19 15 14 12 11 7 6 0

funct7 rs2 rs1 funct3 rd op
7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
0100000	000	SUB rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
0000000	110	OR rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
0000000	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$



R-Type (1): Arithmetic and Logic

00000000 0011001000100010000100001000110011
31 25 24 20 19 15 14 12 11 7 6 0

funct7 rs2 rs1 funct3 rd op
7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
0100000	000	SUB rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
0000000	110	OR rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
0000000	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$



#XOR

R-Type (1): Arithmetic and Logic

00000000 00110010001000100000100001000110011
31 25 24 20 19 15 14 12 11 7 6 0

funct7 rs2 rs1 funct3 rd op
7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
0100000	000	SUB rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
0000000	110	OR rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
0000000	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$



#XOR x4
rd

R-Type (1): Arithmetic and Logic

00000000 00110010001000100000100001000110011
31 25 24 20 19 15 14 12 11 7 6 0

funct7 rs2 rs1 funct3 rd op
7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
0100000	000	SUB rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
0000000	110	OR rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
→ 0000000	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$

#XOR x4, x8
rd, rs1

R-Type (1): Arithmetic and Logic

00000000 00110010001000100000100001000110011
31 25 24 20 19 15 14 12 11 7 6 0

funct7 rs2 rs1 funct3 rd op
7 bits 5 bits 5 bits 3 bits 5 bits 7 bits

funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
0100000	000	SUB rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
0000000	110	OR rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
0000000	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$



#XOR x4, x8, x6
rd, rs1, rs2

R-Type (1): Arithmetic and Logic

00000000011001000100001000010001000110011

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

funct7	rs2	rs1	funct3	rd	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

funct7	funct3	mnemonic	description
0000000	000	ADD rd, rs1, rs2	$R[rd] = R[rs1] + R[rs2]$
0100000	000	SUB rd, rs1, rs2	$R[rd] = R[rs1] - R[rs2]$
0000000	110	OR rd, rs1, rs2	$R[rd] = R[rs1] \mid R[rs2]$
0000000	100	XOR rd, rs1, rs2	$R[rd] = R[rs1] \oplus R[rs2]$

example: x4 = x8 ⊕ x6 #XOR x4, x8, x6
 rd, rs1, rs2

Aside: Truncation

- Suppose we want to convert an 8-bit value into a 4-bit value

0000 0111 = 7

Aside: Truncation

- Suppose we want to convert an 8-bit value into a 4-bit value

0000 0111 = 7 = 0111

Aside: Truncation

- Suppose we want to convert an 8-bit value into a 4-bit value

$$0000 \ 0111 = 7 = 0111$$

$$0000 \ 1111 = 15$$

Aside: Truncation

- Suppose we want to convert an 8-bit value into a 4-bit value

$$0000 \ 0111 = 7 = 0111$$

$$0000 \ 1111 = 15 \neq 1111 \ (-1)$$

Aside: Zero-Extension

- Suppose we want to convert a 4-bit number into an 8-bit number

$$1 = 0001$$

Aside: Zero-Extension

- Suppose we want to convert a 4-bit number into an 8-bit number

$$1 = 0001 = 0000 \ 0001$$

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

$$-1 = 1111$$

Remember negative numbers are encoded using **Two's Complement!**

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

$$-1 = 1111$$

$$-1 = 1111 \ 1111$$

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

$$-1 = 1111$$

$$-1 = 1111 \ 1111$$

$$-(-1) = \neg(1111 \ 1111) + 1$$

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

$$-1 = 1111$$

$$-1 = 1111 \ 1111$$

$$\begin{aligned} -(-1) &= \neg(1111 \ 1111) + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

$$-1 = \textcircled{1}111$$

$$-1 = 1111 \ 1111$$

$$\begin{aligned} -(-1) &= \neg(1111 \ 1111) + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

$$-1 = \textcircled{1}111$$

$$-1 = 1111 \textcircled{1}111$$

$$\begin{aligned} -(-1) &= \neg(1111 \ 1111) + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

$$-1 = \textcircled{1}111$$

$$-1 = \textcircled{1111}\textcircled{1}111$$

$$\begin{aligned} -(-1) &= \neg(1111 \ 1111) + 1 \\ &= 0 + 1 \\ &= 1 \end{aligned}$$

Aside: Sign-Extension

- Suppose we want to convert a 4-bit *negative* number into an 8-bit number

$$-1 = 1111 = 1111 \ 1111$$

The MSB bit (i.e., the sign-bit!) is copied!

Aside: Truncation & Extension

- **Truncation** *decreases* the size of a value

Aside: Truncation & Extension

- **Truncation** *decreases* the size of a value
- **Extension** *increases* the size of a value

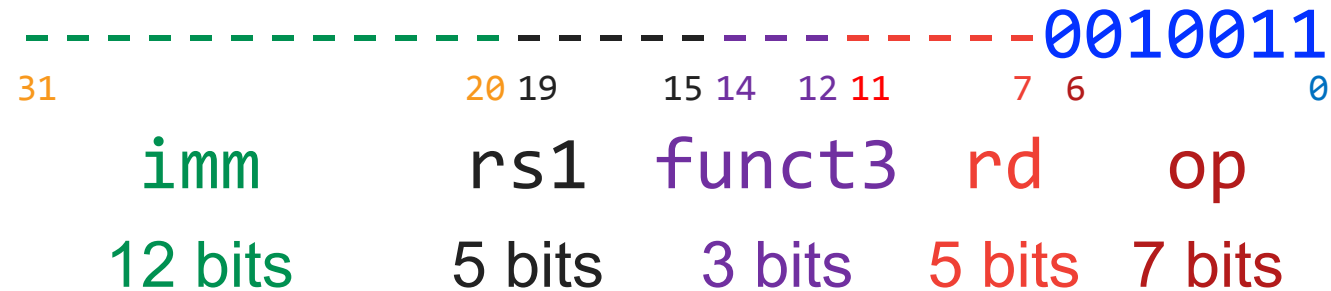
Aside: Truncation & Extension

- **Truncation** *decreases* the size of a value
- **Extension** *increases* the size of a value
 - **Zero-extension** fills upper bits with 0
 - Used to extend *unsigned* numbers

Aside: Truncation & Extension

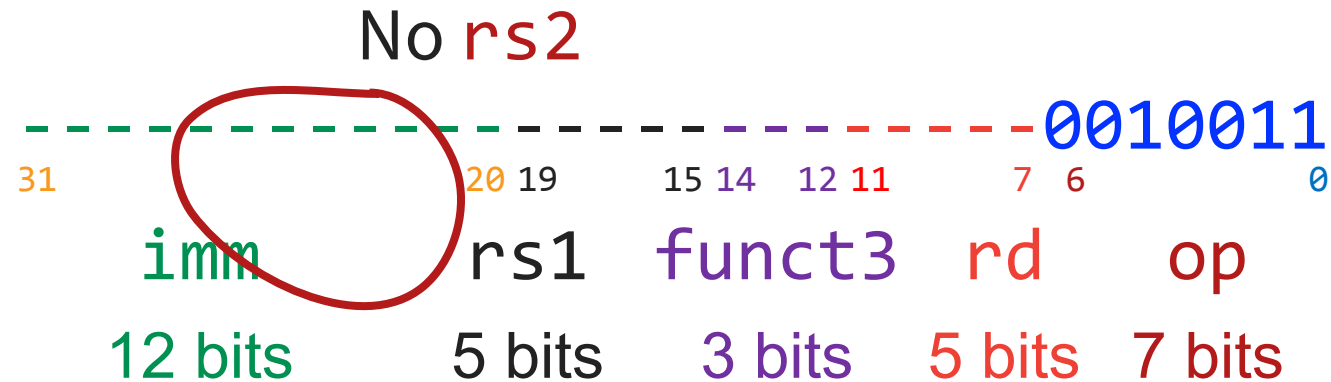
- **Truncation** *decreases* the size of a value
- **Extension** *increases* the size of a value
 - **Zero-extension** fills upper bits with 0
 - Used to extend *unsigned* numbers
 - **Sign-extension** fills upper bits with *copies of the most-significant bit*
 - Used to extend *signed* numbers

I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + \text{sign_extend}(\text{imm})$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(\text{imm})$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(\text{imm})$

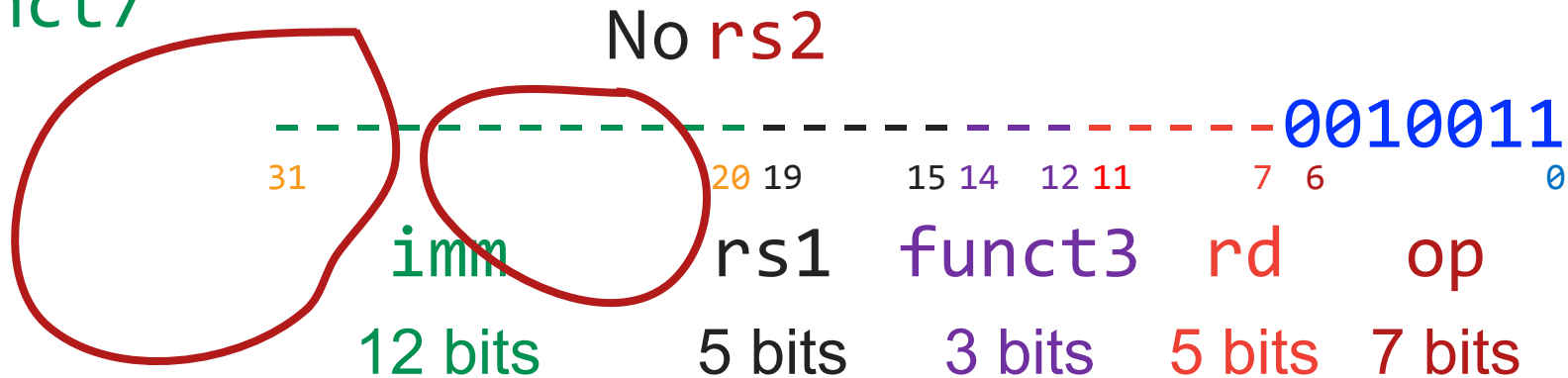
I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + \text{sign_extend}(\text{imm})$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(\text{imm})$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(\text{imm})$

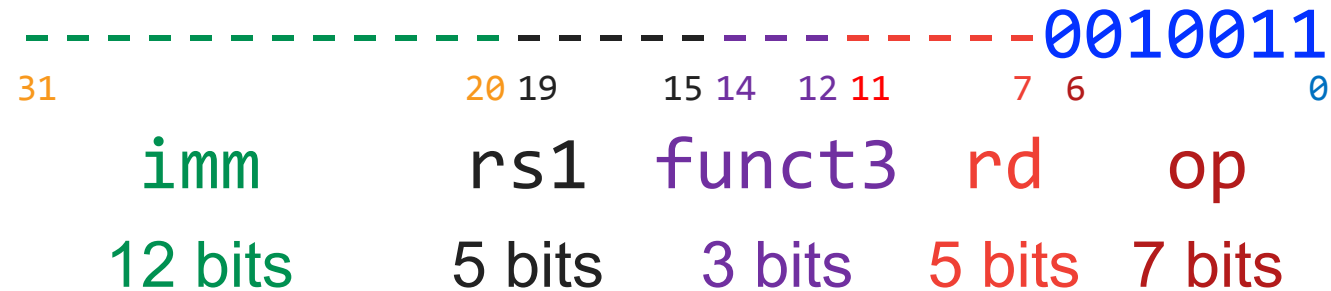
I-Type (1): Arithmetic w/ immediates

No **funct7**



funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + \text{sign_extend}(\text{imm})$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(\text{imm})$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(\text{imm})$

I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + \text{sign_extend}(\text{imm})$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(\text{imm})$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(\text{imm})$

I-Type (1): Arithmetic w/ immediates

00000000010100110000001100010011
31 20 19 15 14 12 11 7 6 0
imm rs1 funct3 rd op
12 bits 5 bits 3 bits 5 bits 7 bits

funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + imm$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(imm)$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(imm)$

I-Type (1): Arithmetic w/ immediates

00000000010100110000001100010011
31 20 19 15 14 12 11 7 6 0
imm rs1 funct3 rd op
12 bits 5 bits 3 bits 5 bits 7 bits

funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + imm$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(imm)$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(imm)$

ADDI

I-Type (1): Arithmetic w/ immediates

00000000010100110000001100010011
31 20 19 15 14 12 11 7 6 0
imm rs1 funct3 rd op
12 bits 5 bits 3 bits 5 bits 7 bits

funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + imm$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(imm)$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(imm)$

ADDI x6,
rd,

I-Type (1): Arithmetic w/ immediates

00000000010100110000001100010011
31 20 19 15 14 12 11 7 6 0
imm rs1 funct3 rd op
12 bits 5 bits 3 bits 5 bits 7 bits

funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + imm$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(imm)$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(imm)$

ADDI x6, x6
rd, rs1

I-Type (1): Arithmetic w/ immediates

00000000010100110000001100010011
31 20 19 15 14 12 11 7 6 0
imm rs1 funct3 rd op
12 bits 5 bits 3 bits 5 bits 7 bits

funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + imm$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(imm)$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(imm)$

ADDI x6, x6, 5
rd, rs1, imm

I-Type (1): Arithmetic w/ immediates

00000000010100110000001100010011
31 20 19 15 14 12 11 7 6 0
imm rs1 funct3 rd op
12 bits 5 bits 3 bits 5 bits 7 bits

funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + imm$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(imm)$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(imm)$

x6 = x6 + 5 # ADDI x6, x6, 5
rd, rs1, imm

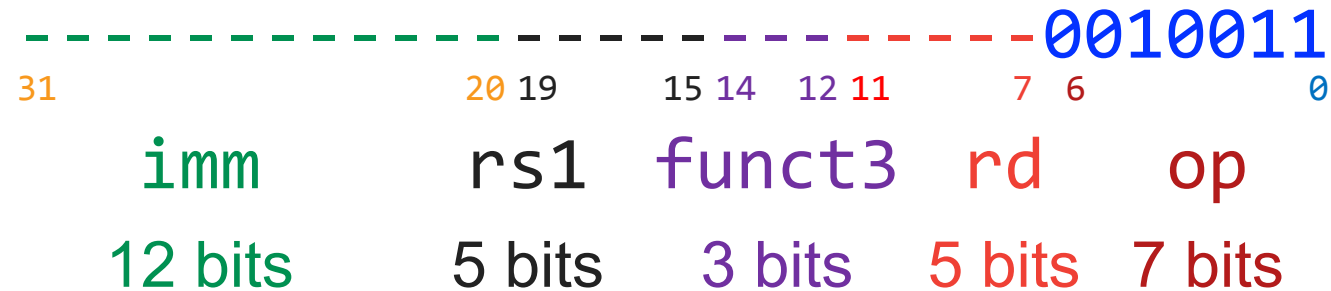
I-Type (1): Arithmetic w/ immediates

00000000010100110000001100010011
31 20 19 15 14 12 11 7 6 0
imm rs1 funct3 rd op
12 bits 5 bits 3 bits 5 bits 7 bits

funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + imm$
111	ANDI rd, rs1, imm	$R[rd] = R[rs1] \& \text{sign_extend}(imm)$
110	ORI rd, rs1, imm	$R[rd] = R[rs1] \mid \text{sign_extend}(imm)$

x6 = x6 + 5 # ADDI x6, x6, 5
x6 += 5 rd, rs1, imm

I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	$R[rd] = R[rs1] + \text{sign_extend}(\text{imm})$

PollEv.com/cs3410

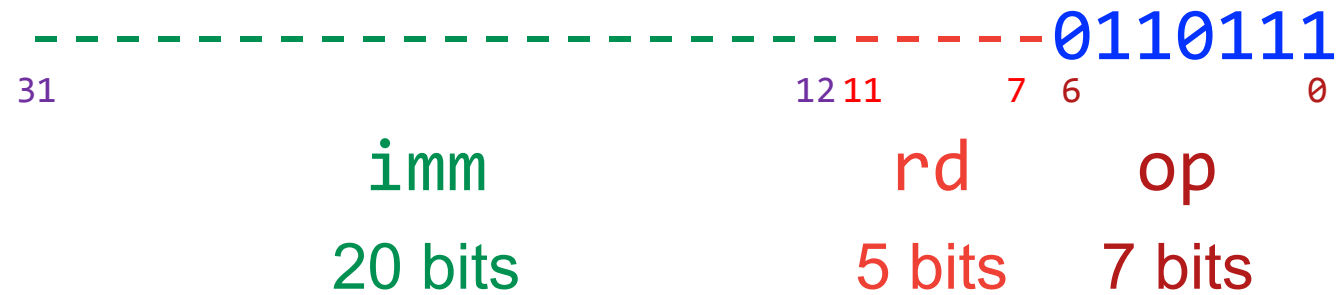


What is the largest immediate value you can add with ADDI?

Loading larger immediate values

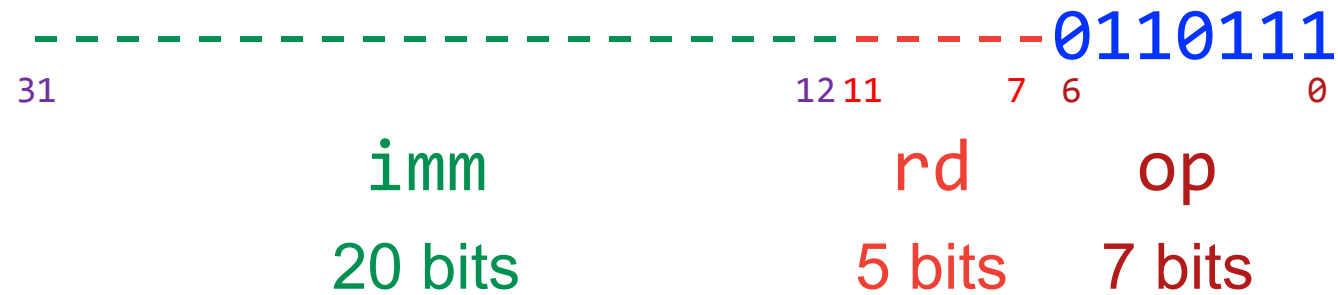


U-Type (1): Load Upper Immediate



mnemonic	description
LUI rd , imm	$R[\text{rd}] = \text{imm} \ll 12$

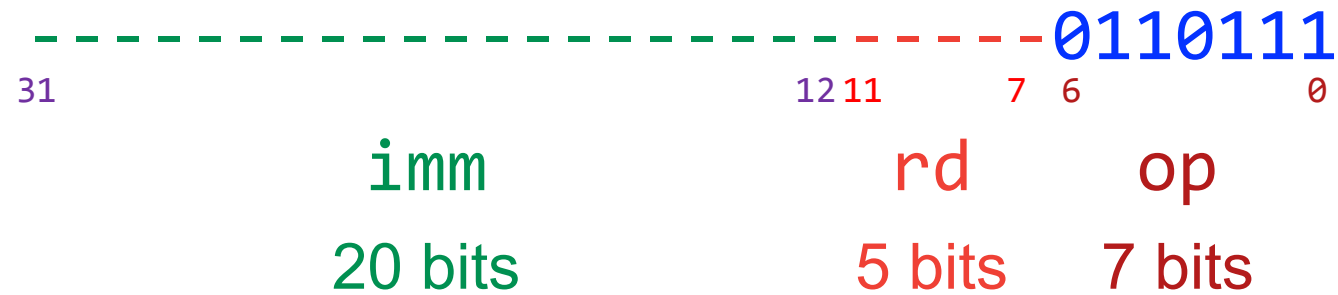
U-Type (1): Load Upper Immediate



mnemonic	description
LUI rd , imm	$R[\text{rd}] = \text{imm} \ll 12$

example: x5 = 0x5000 # LUI x5, 5
 rd, imm

U-Type (1): Load Upper Immediate

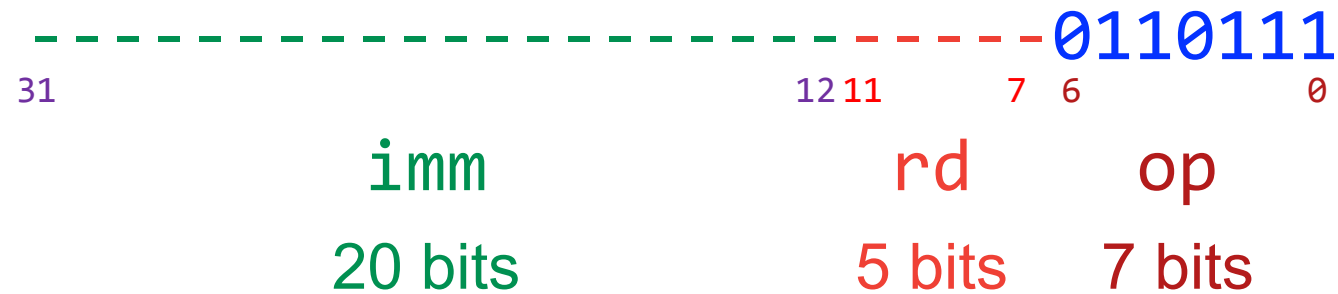


mnemonic	description
LUI rd , imm	$R[\text{rd}] = \text{imm} \ll 12$

example: x5 = 0x5000 # LUI x5, 5
 rd, imm

Typical Usage Pattern: LUI x5, 0x12345
 ADDI x5, x5 0x678

U-Type (1): Load Upper Immediate



mnemonic	description
LUI rd , imm	$R[\text{rd}] = \text{imm} \ll 12$

example: x5 = 0x5000 # LUI x5, 5
 rd, imm

Typical Usage Pattern: LUI x5, 0x12345
 ADDI x5, x5 0x678
 → x5 = 0x12345678

Assembly Programming

Pseudocode

`a0 = 34`

`a1 = a0 - 13`

`a2 = a1 * 2`

Assembly

Assembly Programming

Pseudocode

a0 = 34

a1 = a0 - 13

a2 = a1 * 2

Assembly



Assembly Programming

Pseudocode

a0 = 34

a1 = a0 - 13

a2 = a1 * 2

Assembly

addi a0, x0, 34



How do we put
34 into register
a0?

Assembly Programming

Pseudocode

a0 = 34

a1 = a0 - 13

a2 = a1 * 2

Assembly

addi a0, x0, 34

Always zero!

How do we put
34 into register
a0?

Assembly Programming

Pseudocode

$a0 = 0 + 34$

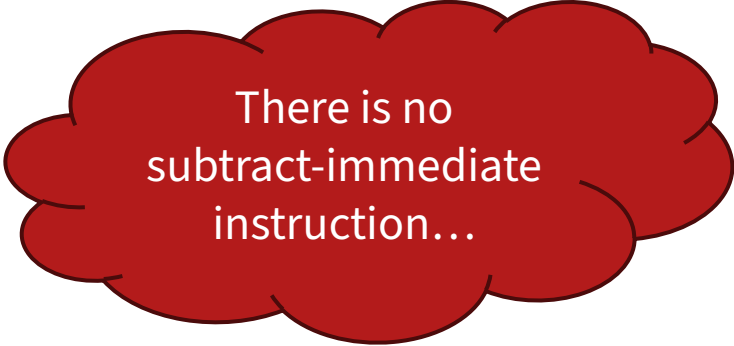
$a1 = a0 - 13$

$a2 = a1 * 2$

Assembly

`addi a0, x0, 34`

`addi a1, a0, -13`



There is no
subtract-immediate
instruction...

Assembly Programming

Pseudocode

$a0 = 0 + 34$

$a1 = a0 - 13$

$a2 = a1 * 2$

Assembly

```
addi a0, x0, 34
```

```
addi a1, a0, -13
```

Assembly Programming

Pseudocode

$a0 = 0 + 34$

$a1 = a0 - 13$

$a2 = a1 * 2$

Assembly

```
addi a0, x0, 34
```

```
addi a1, a0, -13
```



Multiplying by 2
is the same as
shifting left by 1!

Assembly Programming

Pseudocode

$a0 = 0 + 34$

$a1 = a0 - 13$

$a2 = a1 \ll 1$

Assembly

```
addi a0, x0, 34
```

```
addi a1, a0, -13
```

```
slli a2, a1, 1
```



Multiplying by 2
is the same as
shifting left by 1!

Next Monday

- RISC-V Control Flow
- RISC-V Memory