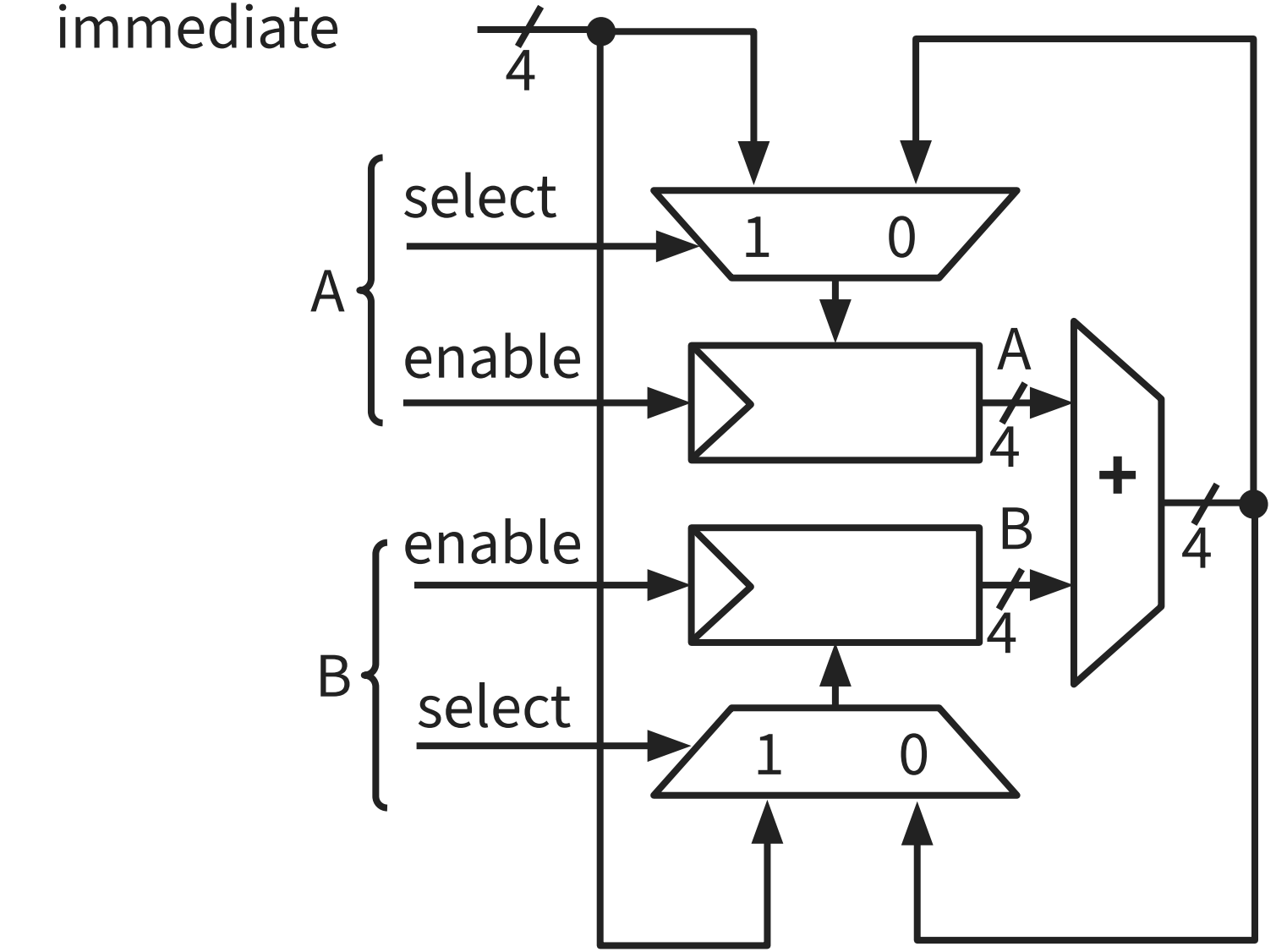
Review: Femto Proc

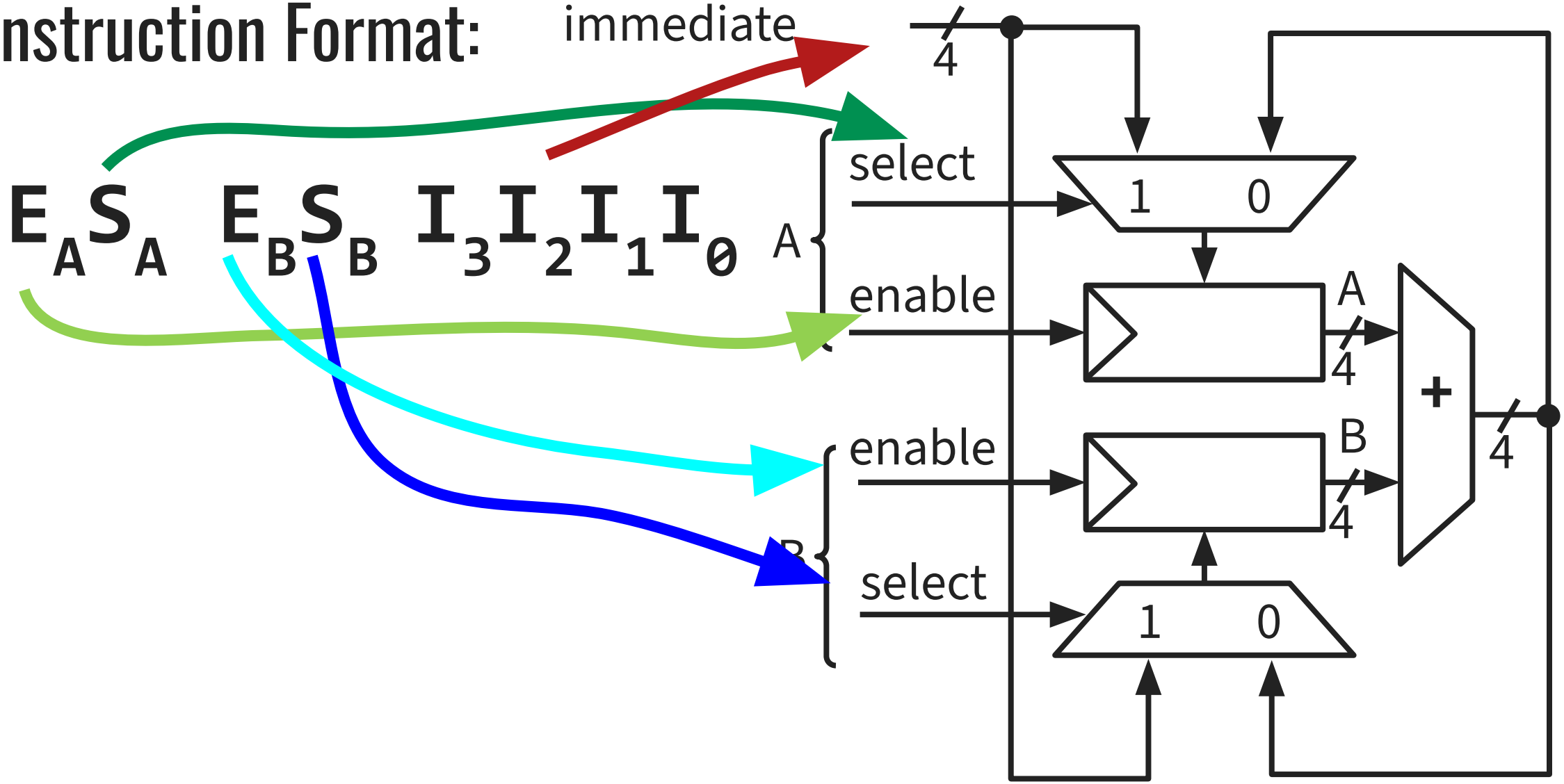
CS 3410: Computer System Organization and Programming

Fall 2025



Processor Architecture

Instruction Format:



Processor Architecture

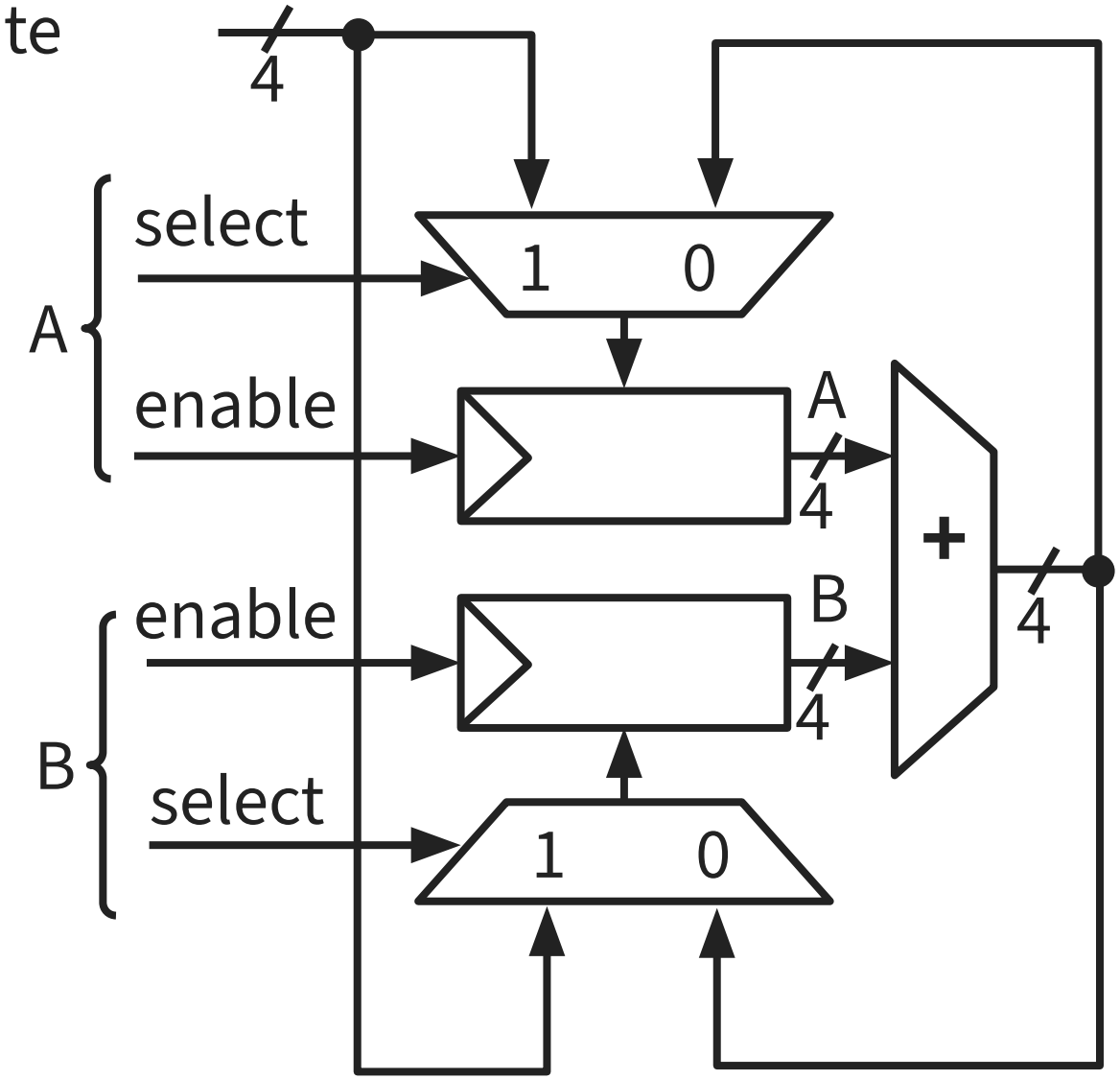
Instruction Format:

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? ????
```

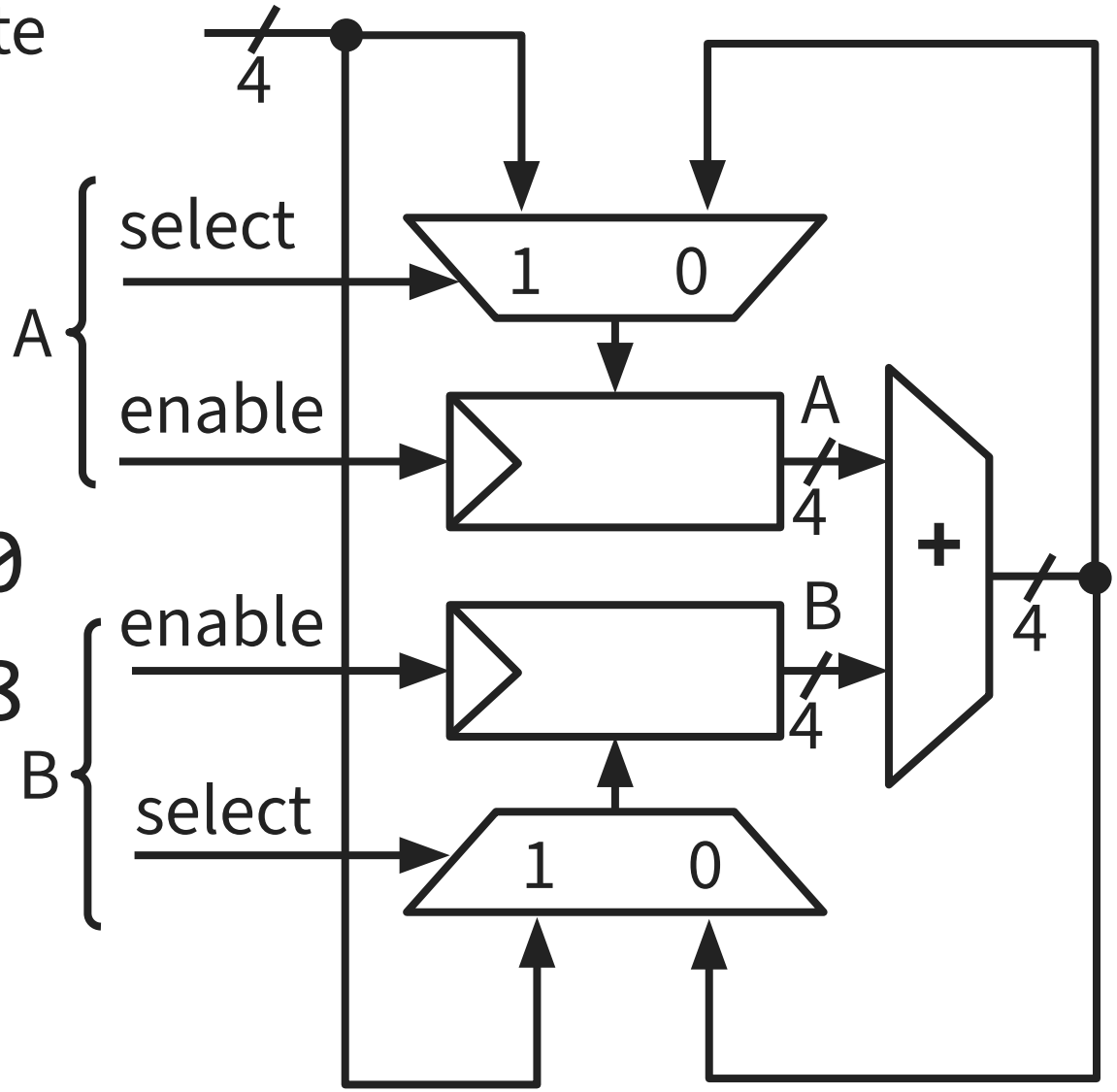


Processor Architecture

Instruction Format: immediate

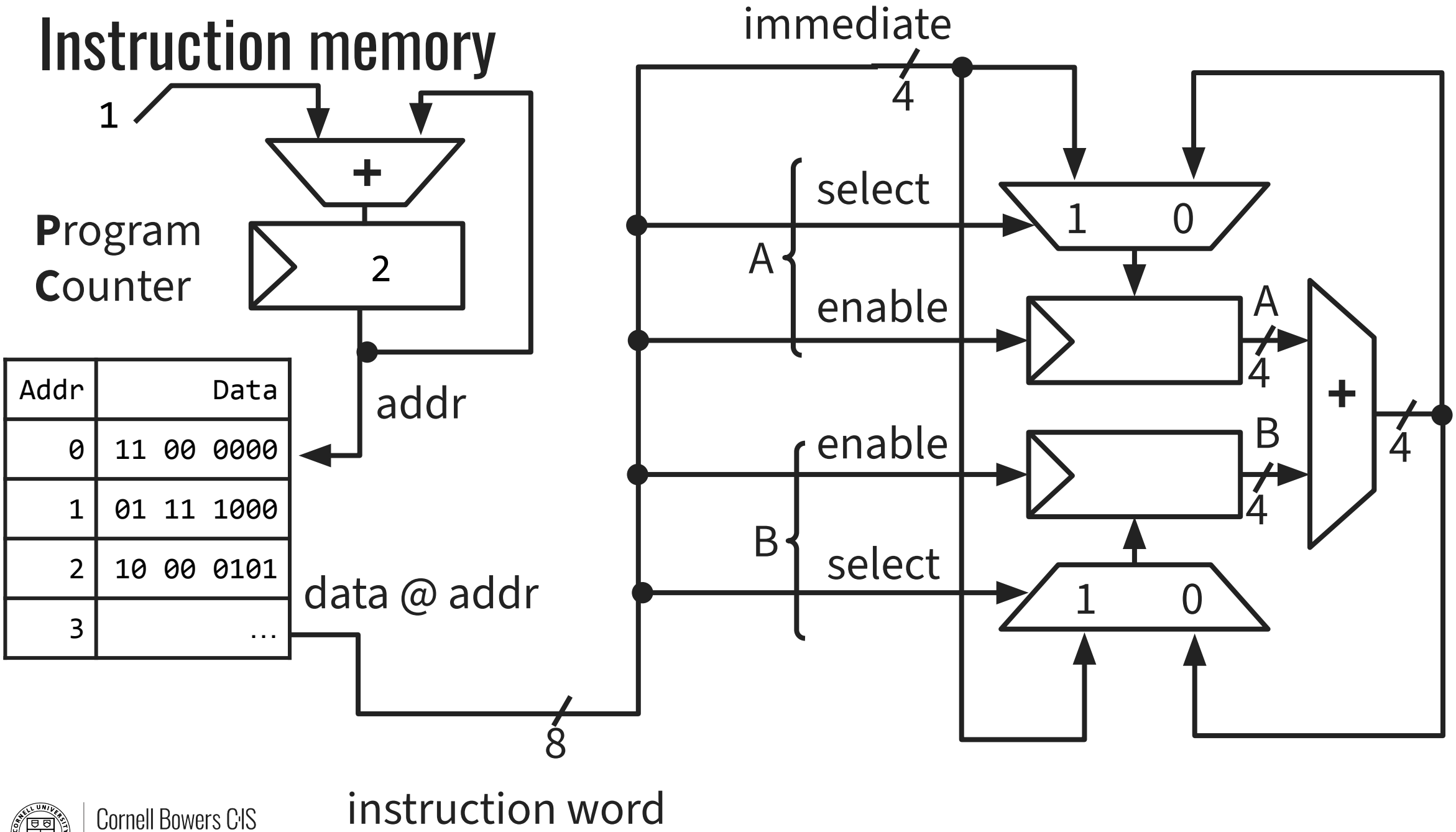
$E_A S_A \; E_B S_B \; I_3 I_2 I_1 I_0$

Machine Code	Mnemonic
11 0? 0000	wrimm A, 0
0? 11 1000	wrimm B, 8
10 0? ????	add A

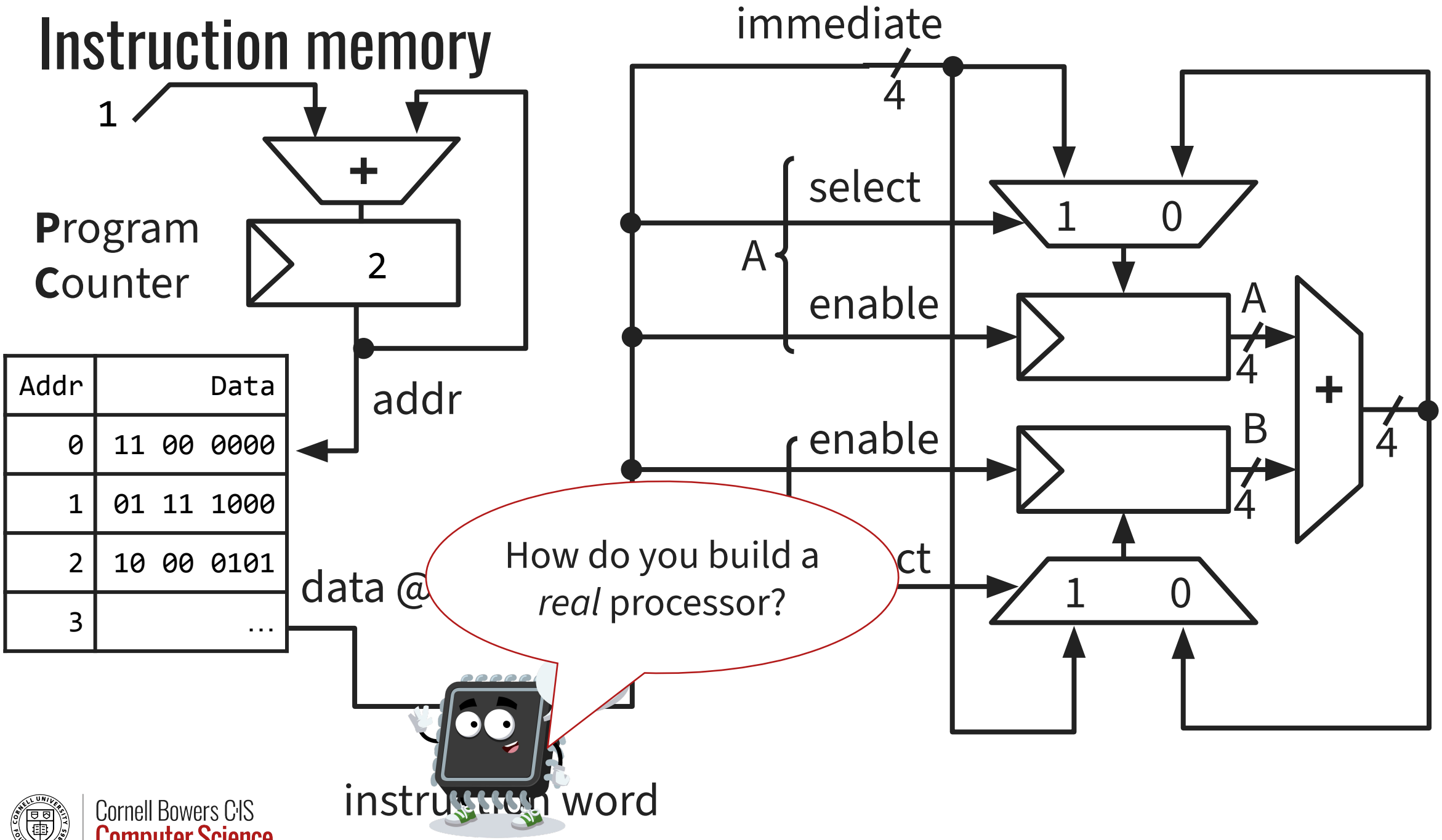


Processor Architecture

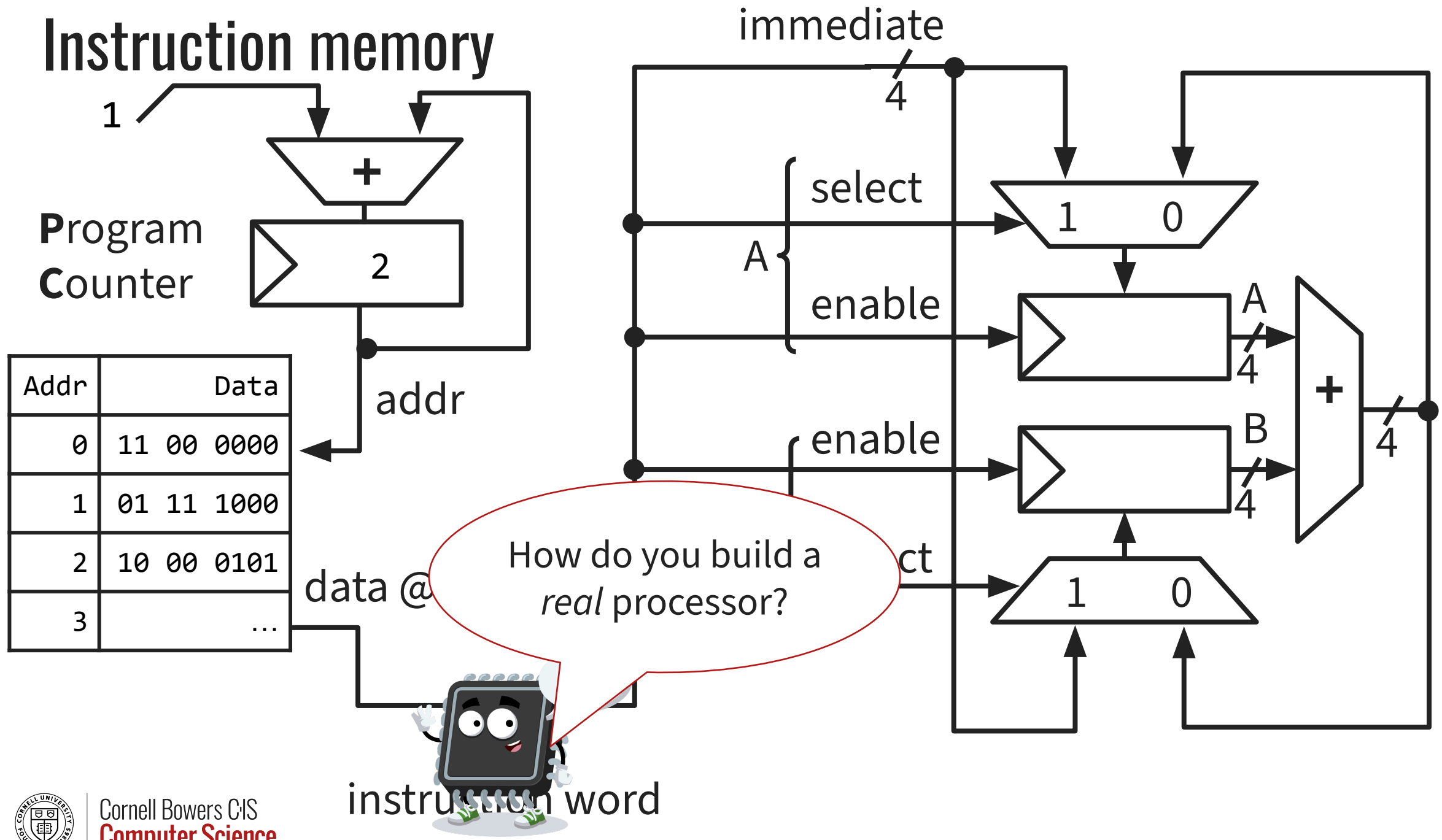
Instruction memory



Instruction memory



Instruction memory



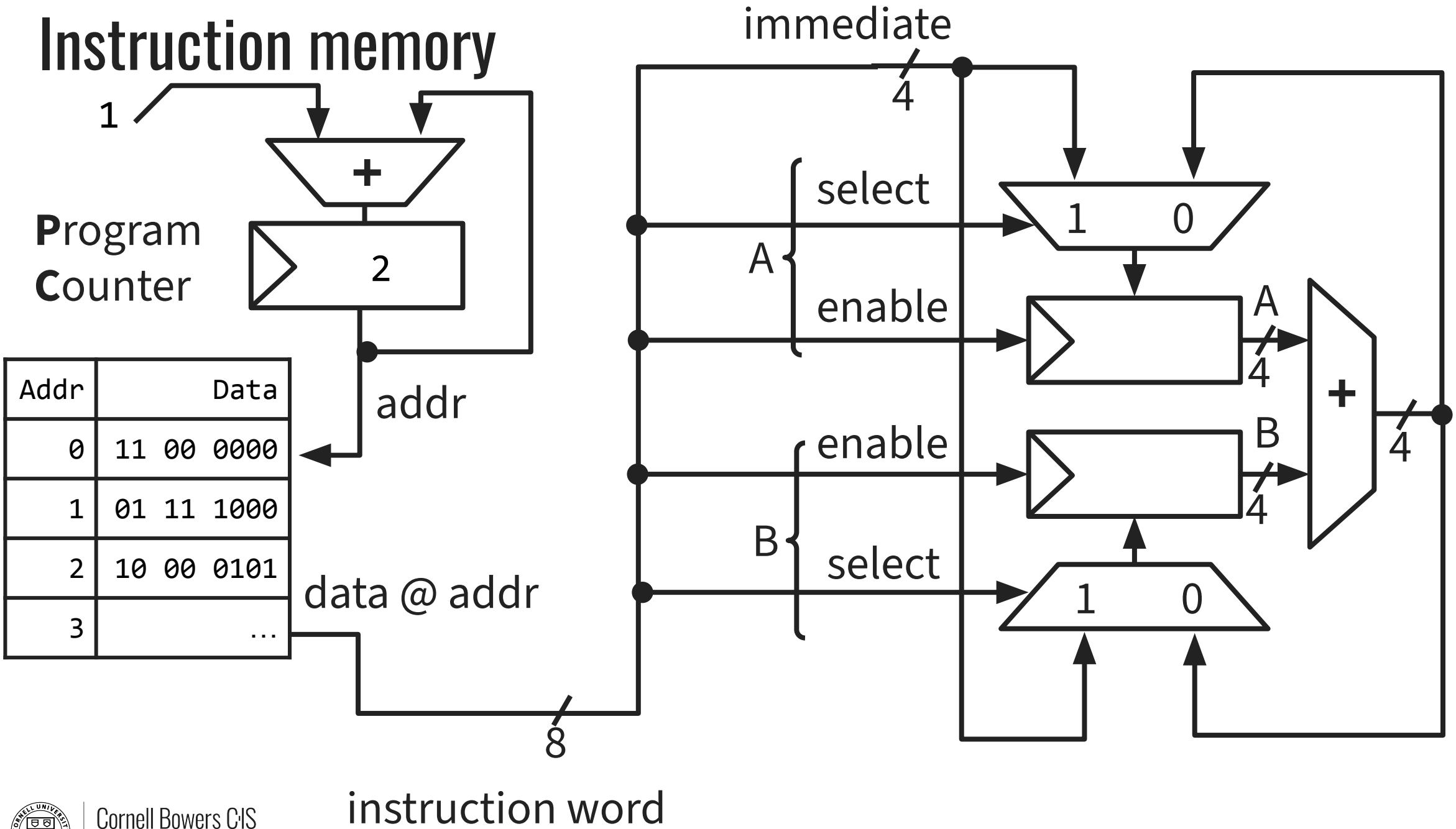
RISC-V

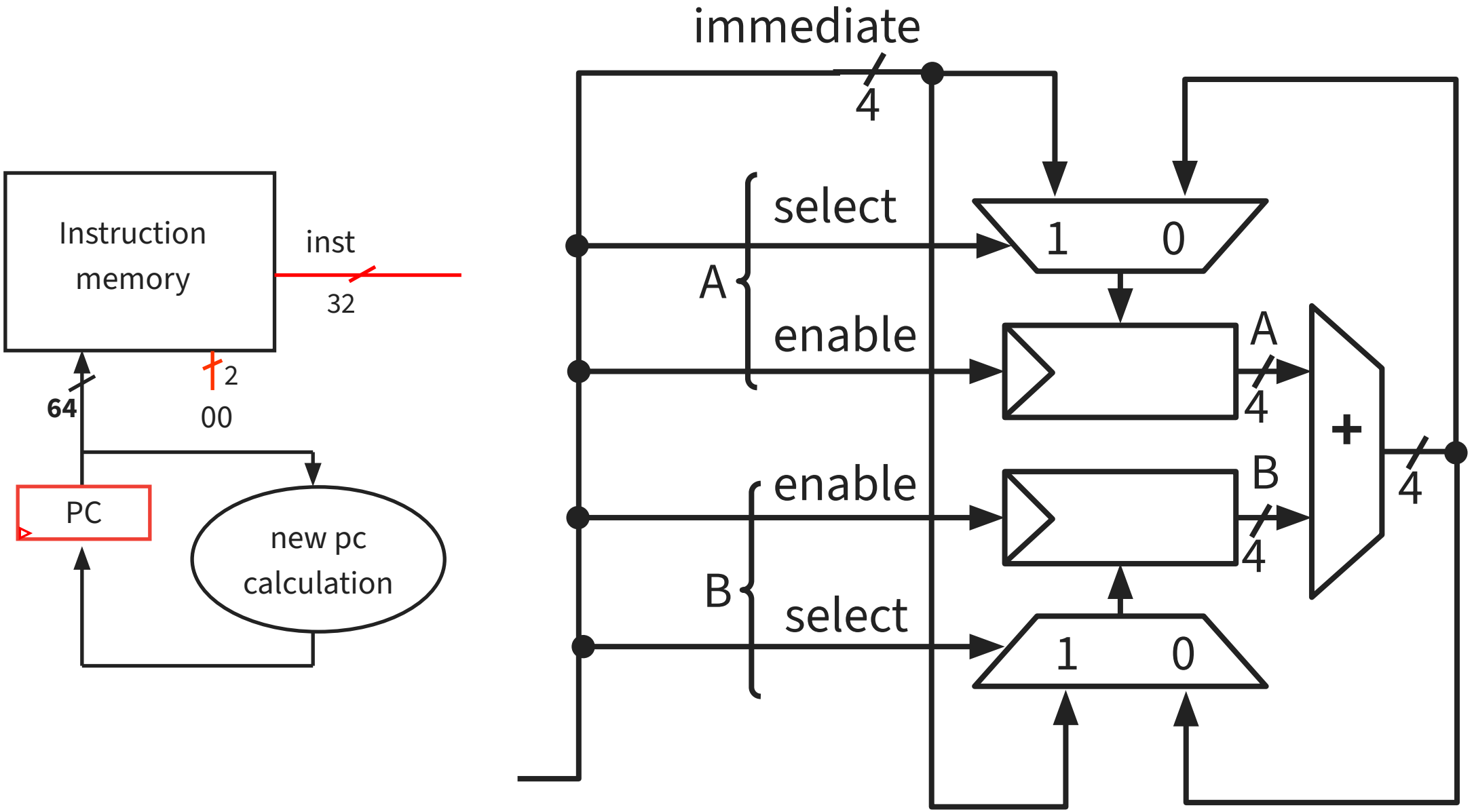
CS 3410: Computer System Organization and Programming

Fall 2025

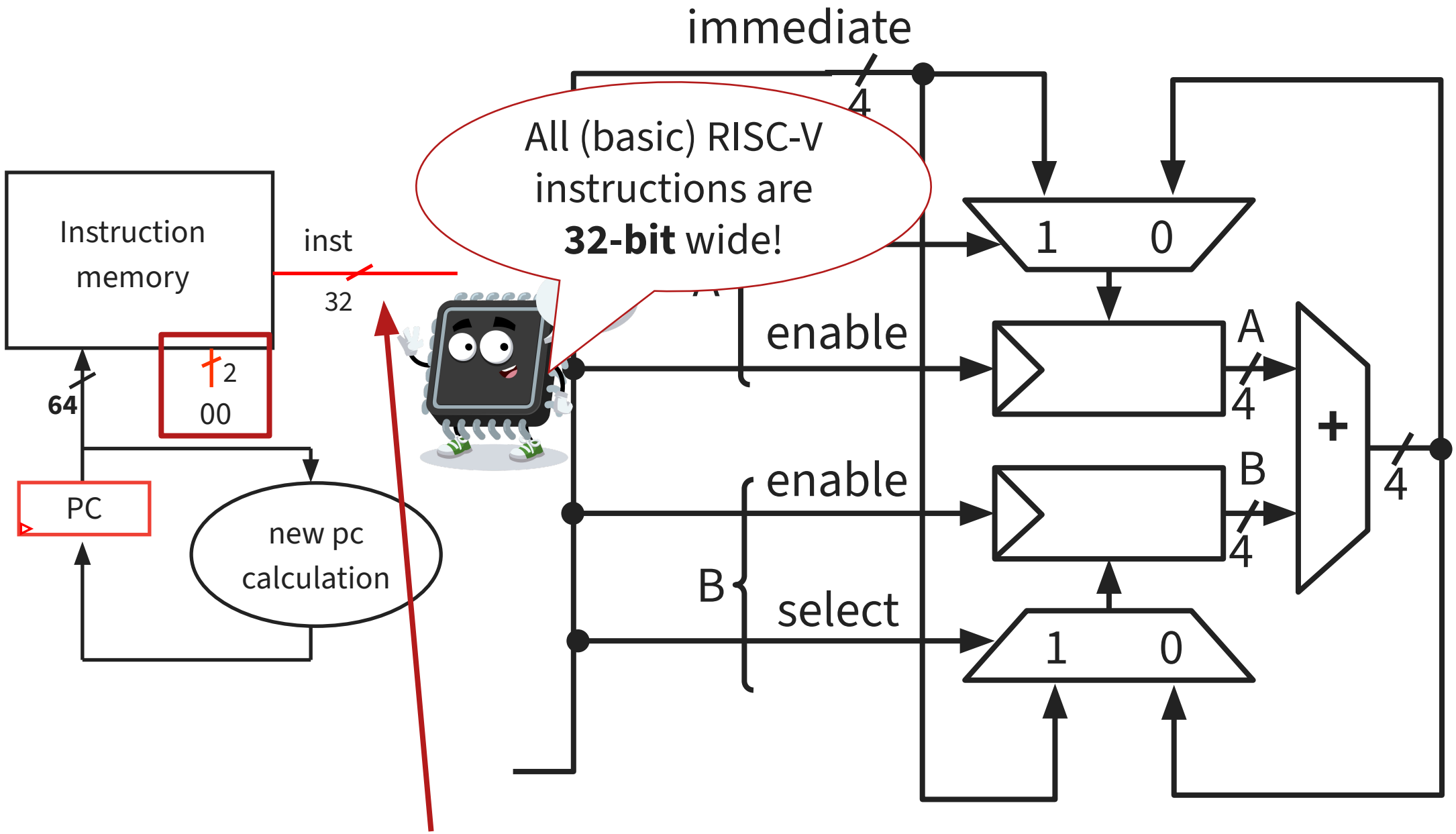
From FemtoProc to RISC-V

Instruction memory

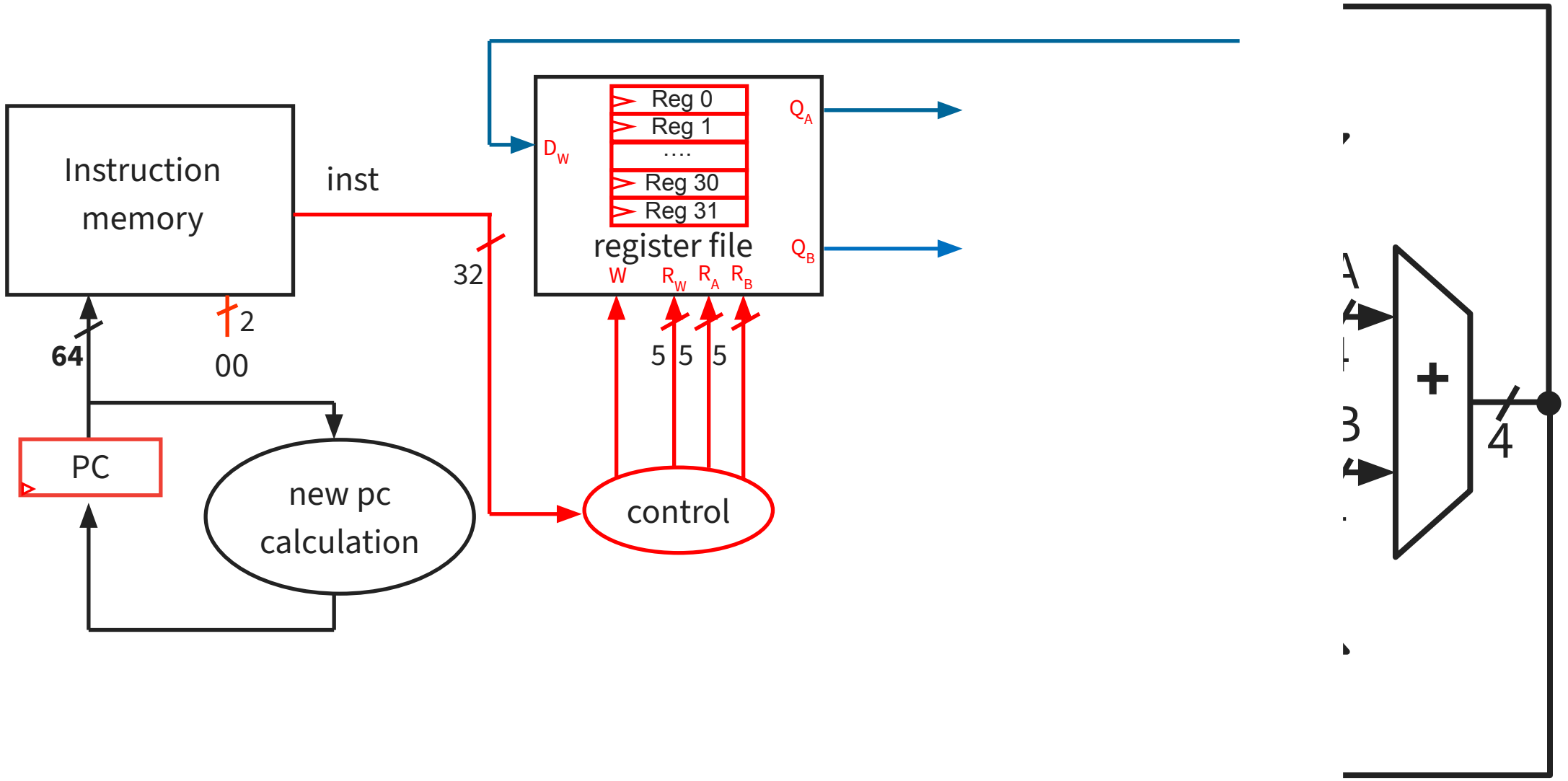


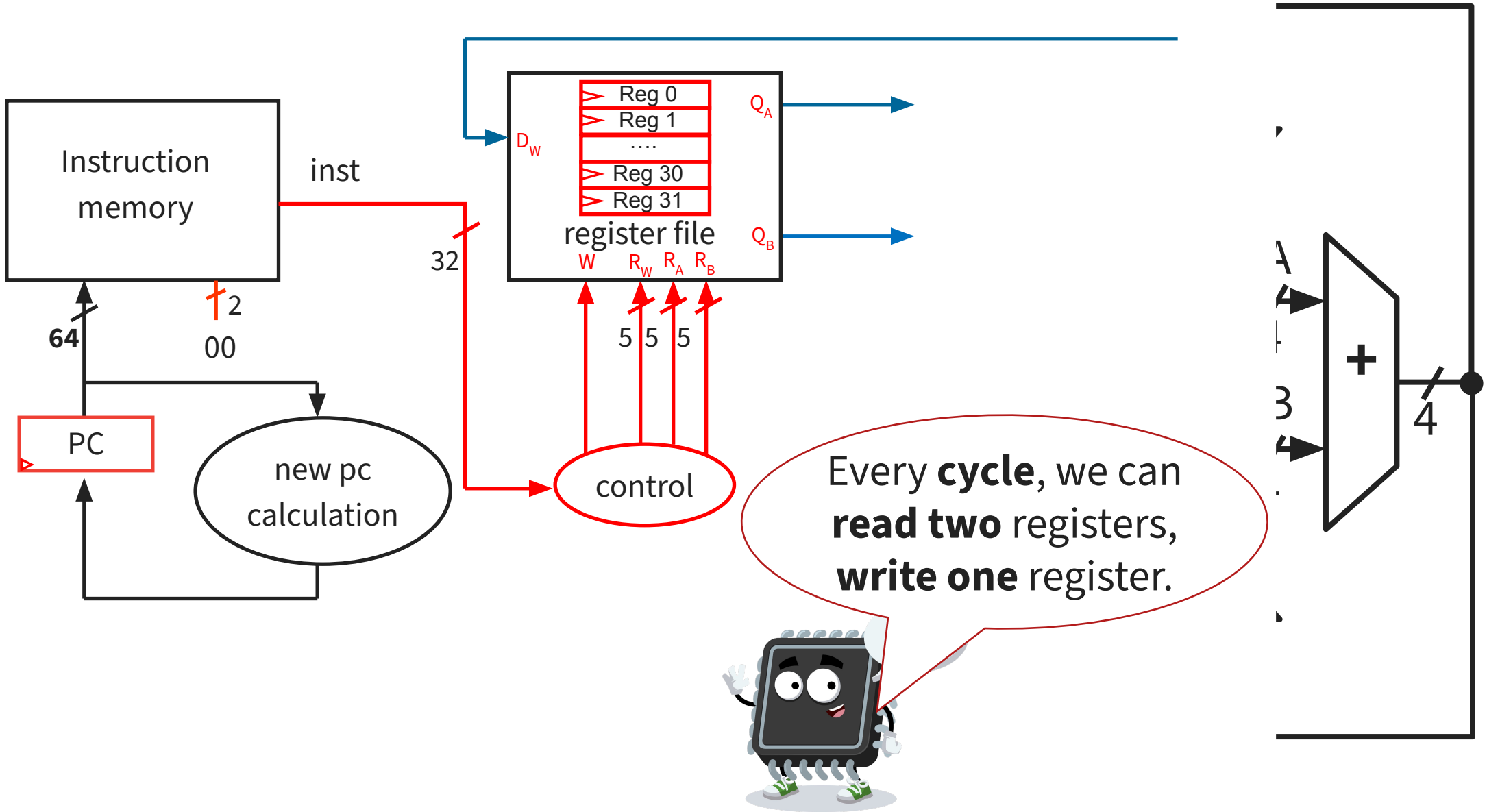


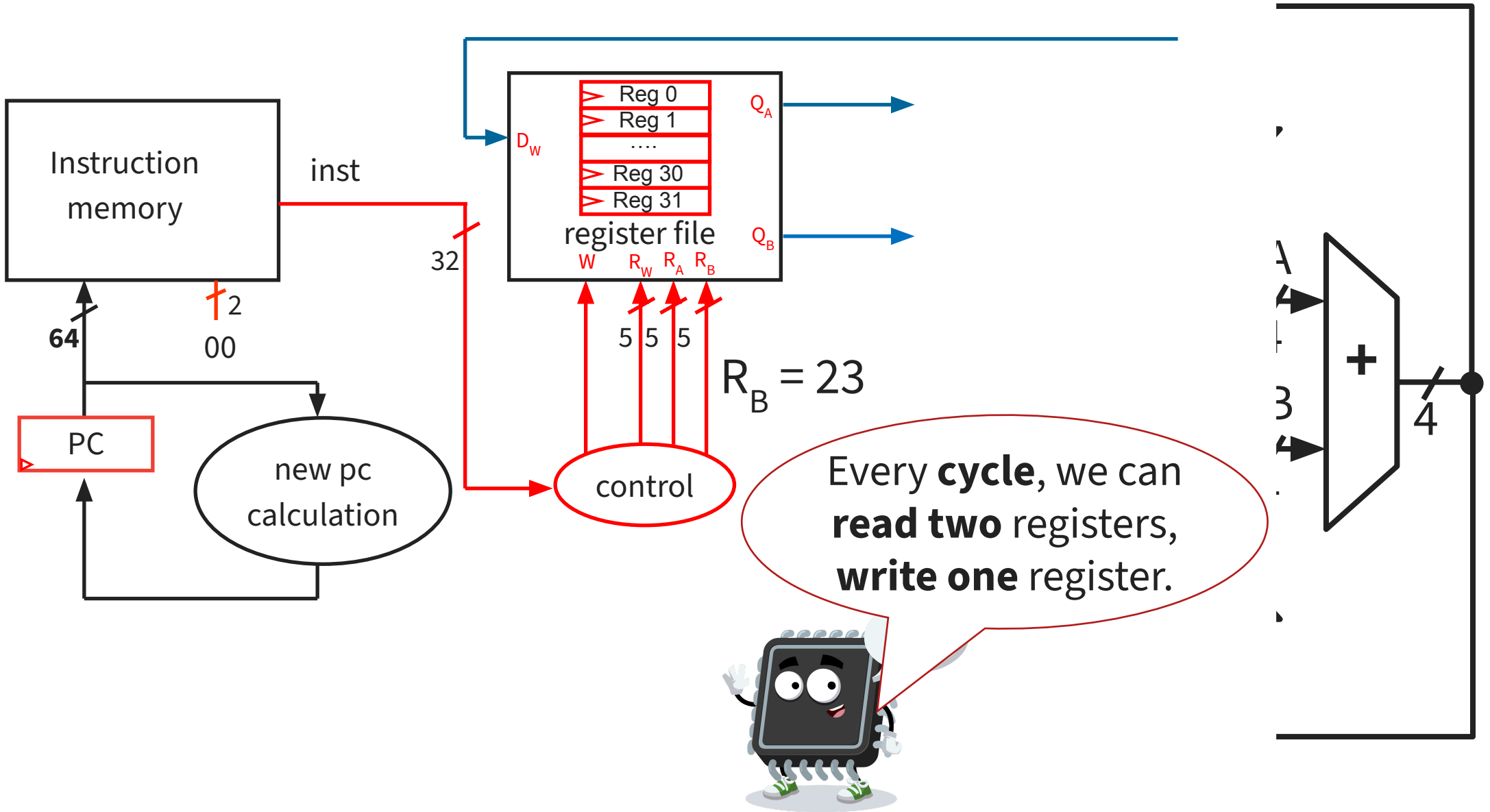
instruction word

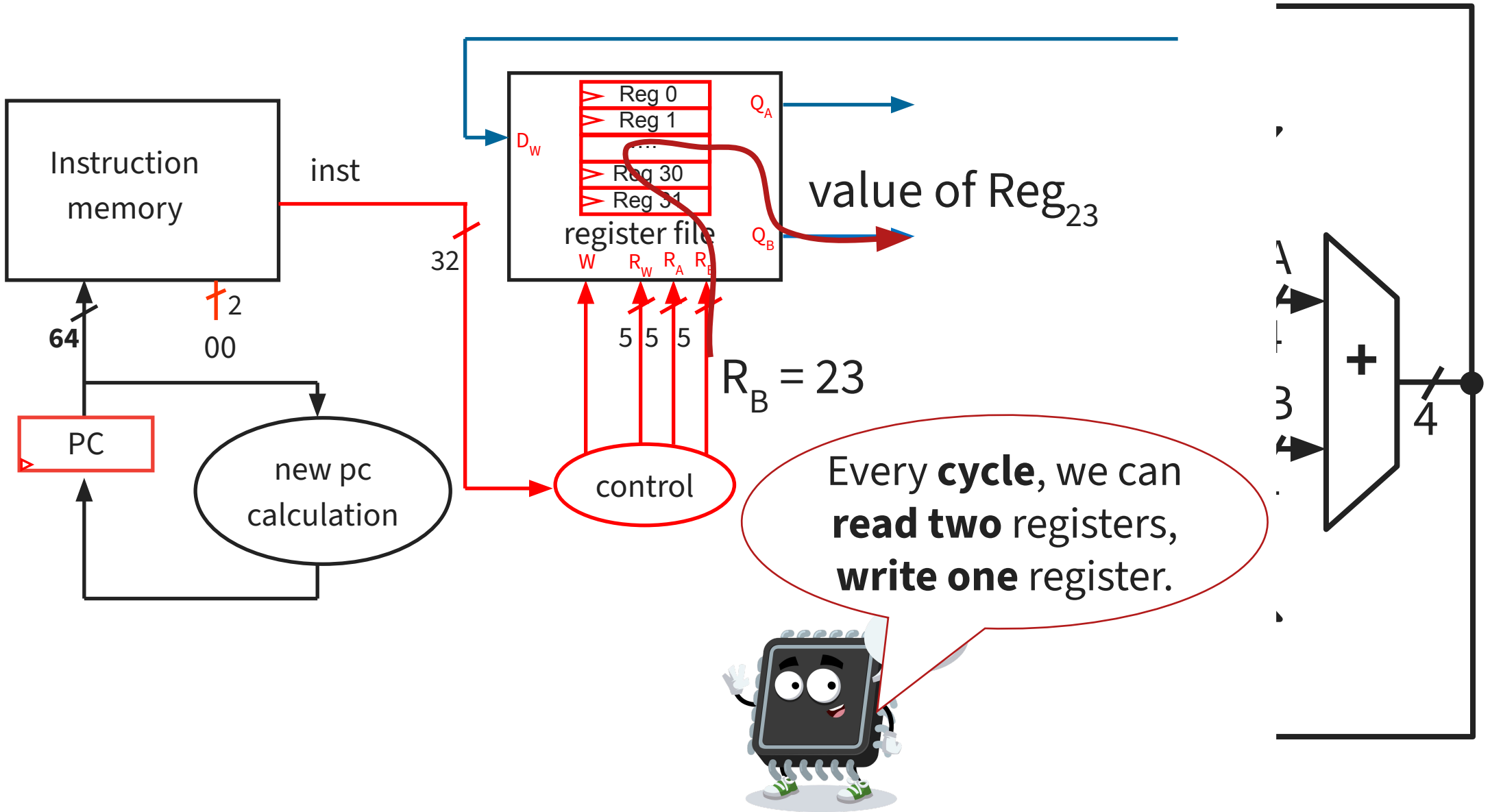


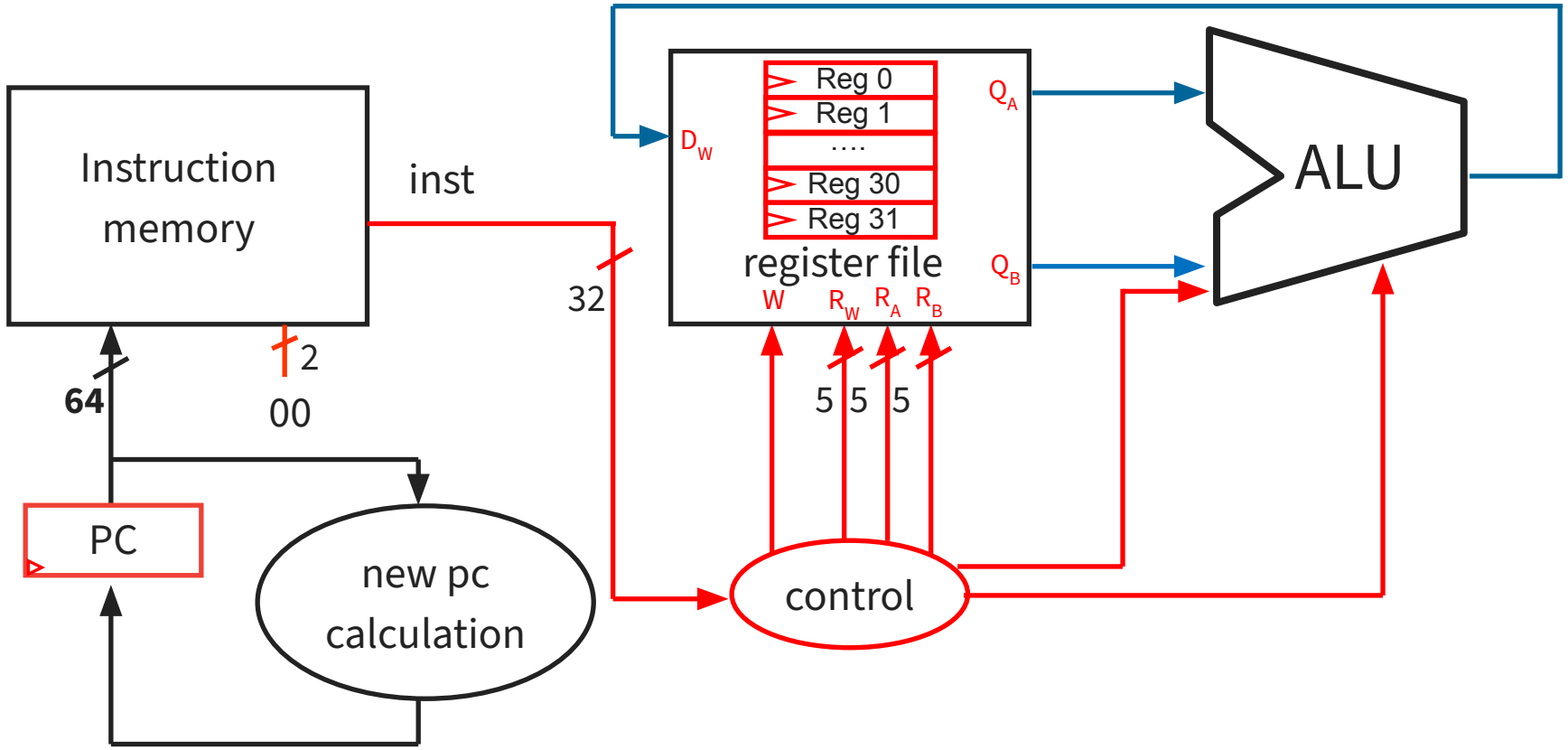
32-bit instruction word

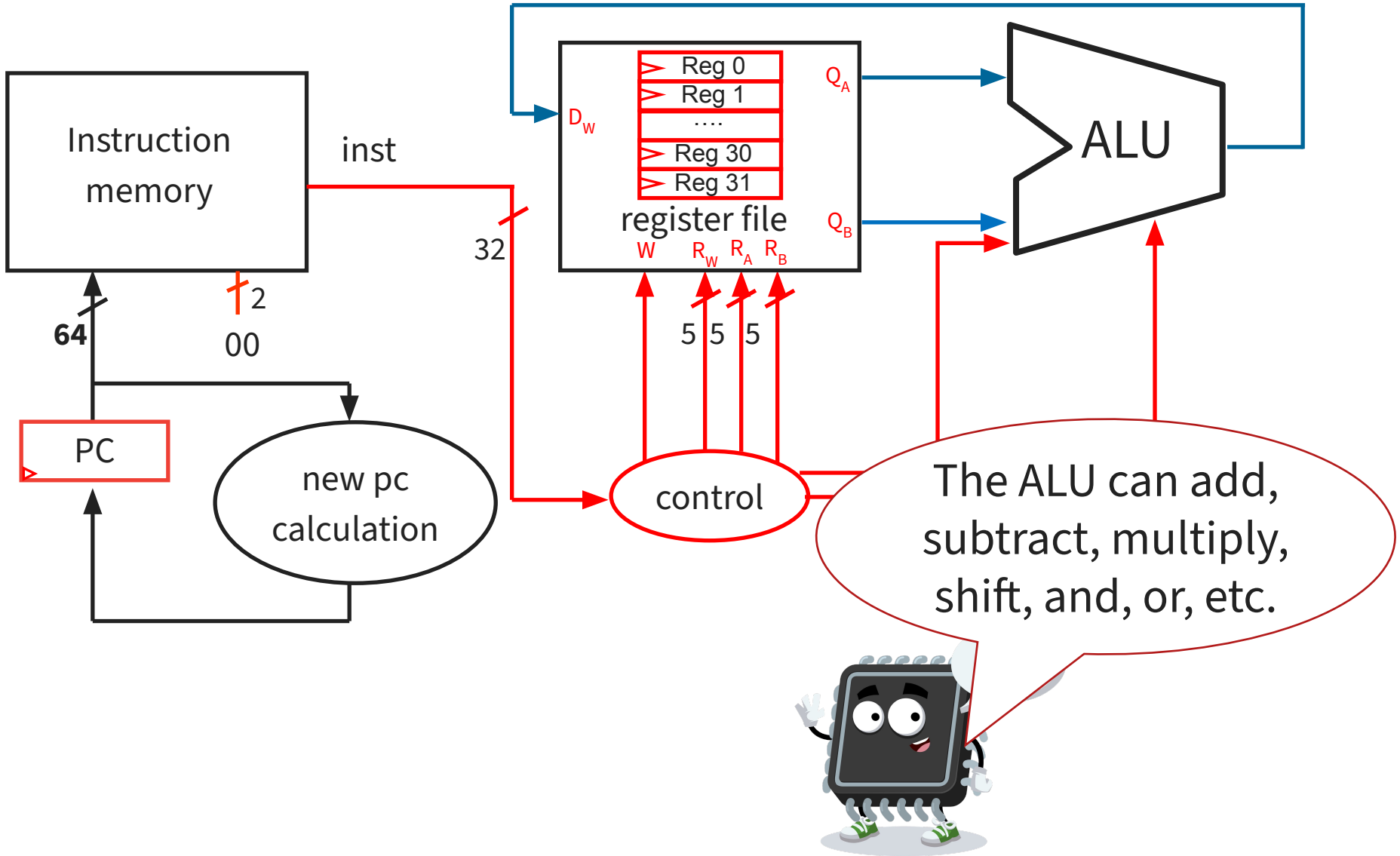

Instruction memory
inst
32
64
2
00
PC
new pc calculation
control
Reg 0
Reg 1
....
Reg 30
Reg 31
register file
D_W
Q_A
Q_B
W
R_W
R_A
R_B
5 5 5
+
4


Instruction memory
inst
32
64
2
00
PC
new pc calculation
Reg 0
Reg 1
....
Reg 30
Reg 31
register file
D_W
Q_A
Q_B
W
R_W
R_A
R_B
5 5 5
control
Every **cycle**, we can **read two** registers, **write one** register.
+
4


Instruction memory
inst
32
64
2
00
PC
new pc calculation
Reg 0
Reg 1
....
Reg 30
Reg 31
register file
D_W
Q_A
Q_B
W
R_W
R_A
R_B
5 5 5
$R_B = 23$
control
Every **cycle**, we can **read two** registers, **write one** register.
+
4


Instruction memory
inst
32
64
2
00
PC
new pc calculation
Reg 0
Reg 1
Reg 30
Reg 31
register file
D_W
Q_A
Q_B
W
R_W
R_A
R_B
5 5 5
control
value of Reg_{23}
$R_B = 23$
+
4
Every **cycle**, we can **read two** registers, **write one** register.


Instruction memory
inst
32
64
2
00
PC
new pc calculation
Reg 0
Reg 1
....
Reg 30
Reg 31
register file
D_W
Q_A
Q_B
W
R_W
R_A
R_B
5
5
5
ALU
control

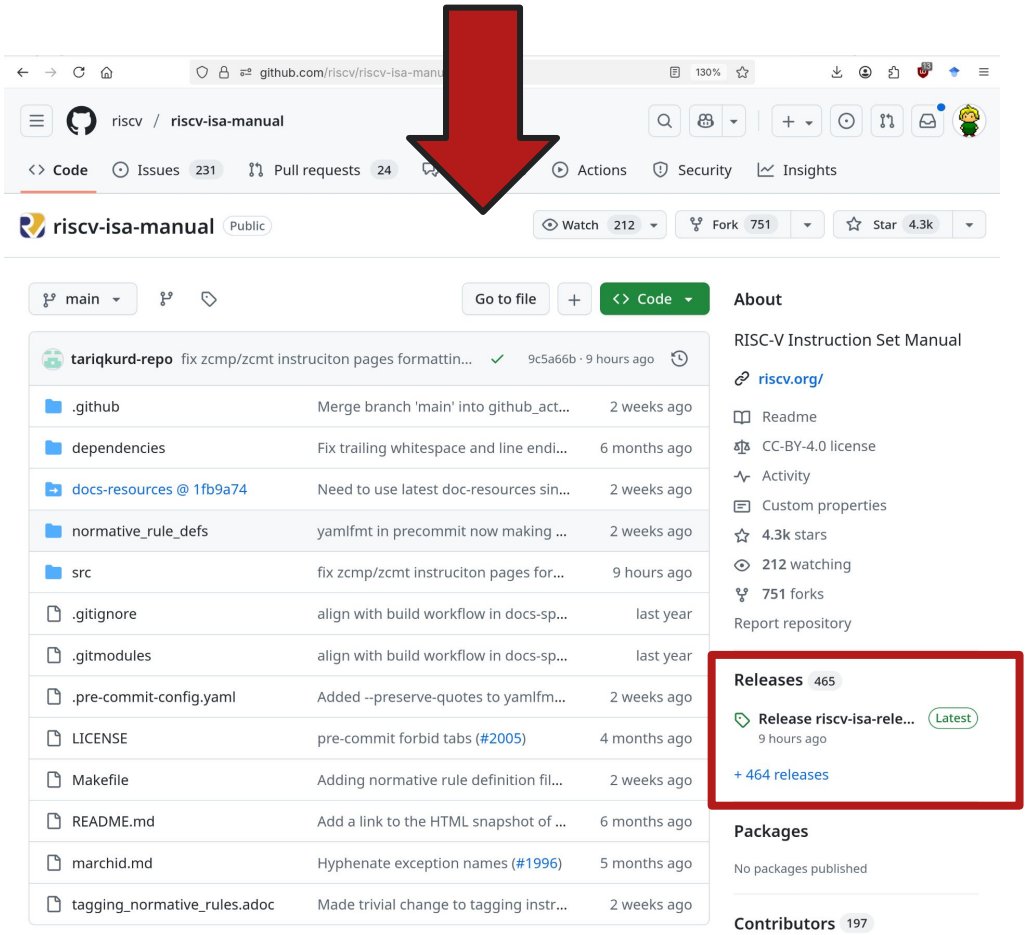

Instruction memory
inst
32
64
2
00
PC
new pc calculation
Reg 0
Reg 1
....
Reg 30
Reg 31
register file
D_W
Q_A
Q_B
W
R_W
R_A
R_B
5 5 5
control
ALU
The ALU can add, subtract, multiply, shift, and, or, etc.

The RISC-V Add Instruction

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

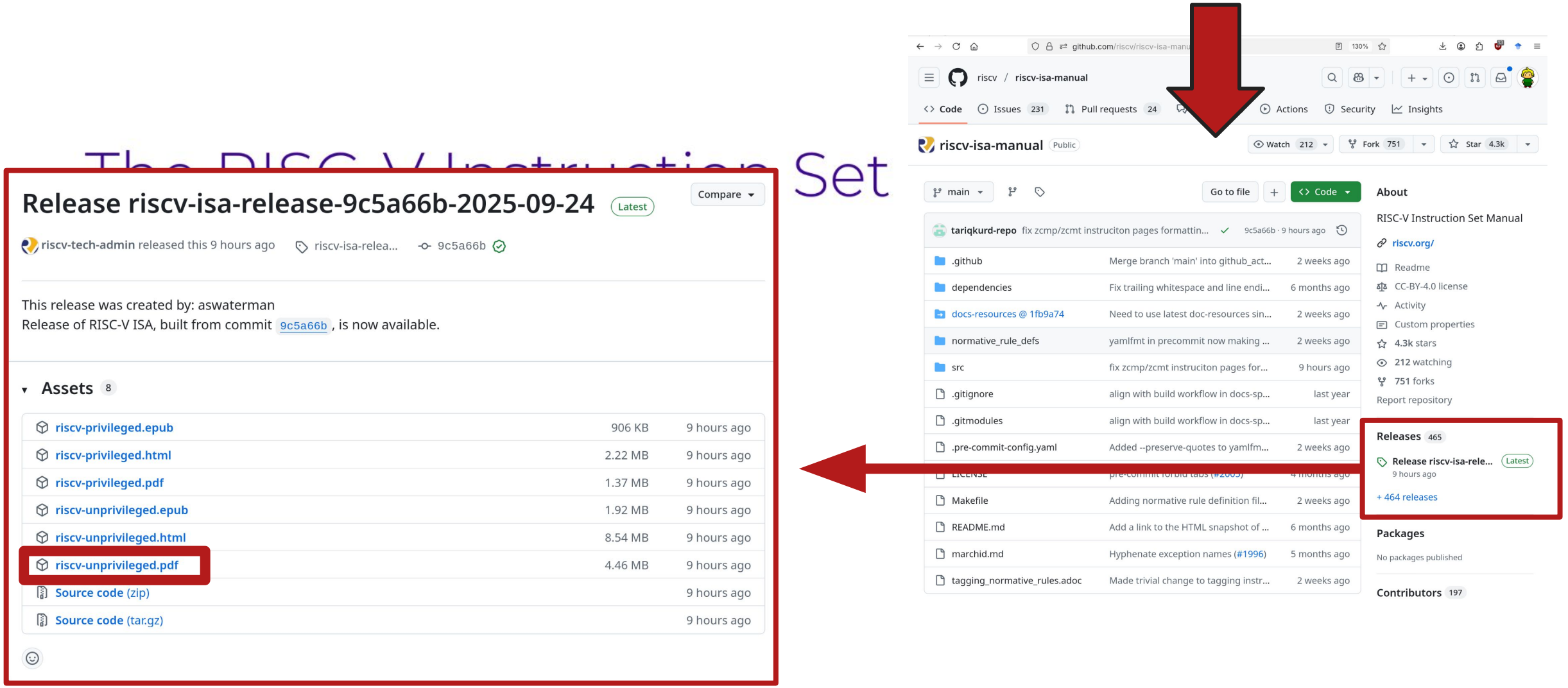
The RISC-V Add Instruction

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



The RISC-V Add Instruction

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

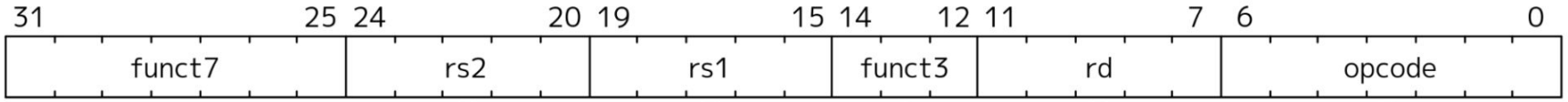


The RISC-V Add Instruction

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.



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

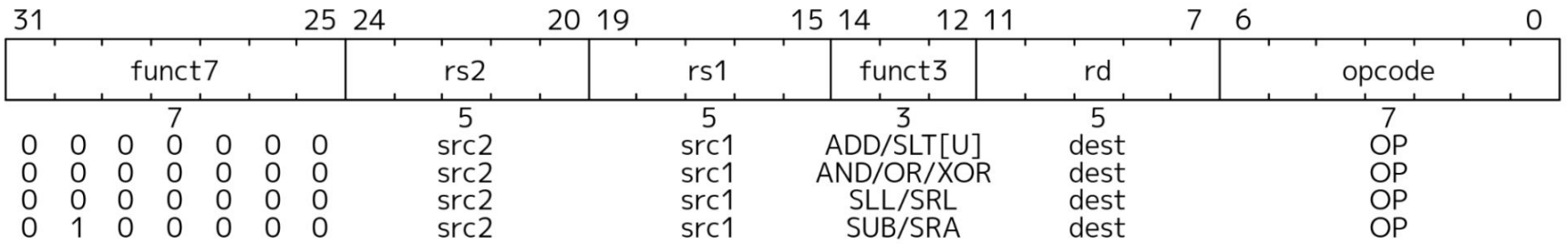
ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. 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

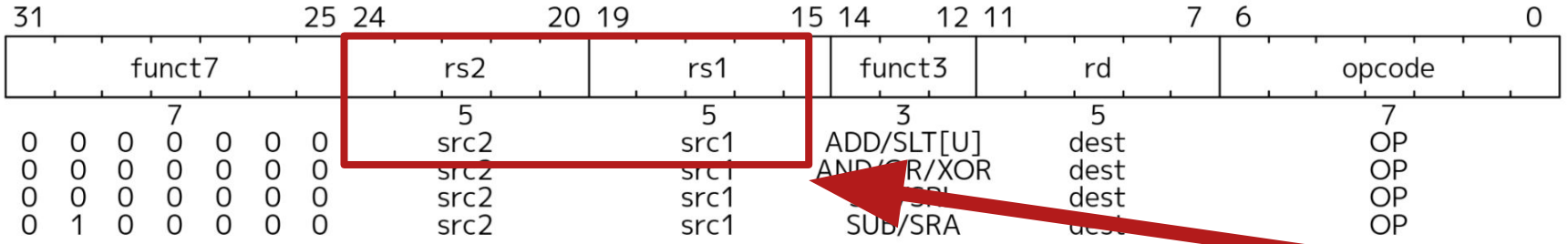
ADD performs the addition of *rs1* and *rs2*. SUB performs the subtraction of *rs2* from *rs1*. Overflows are ignored and the low XLEN bits of results are written to the destination *rd*. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to *rd* if *rs1* < *rs2*, 0 otherwise. 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.



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**

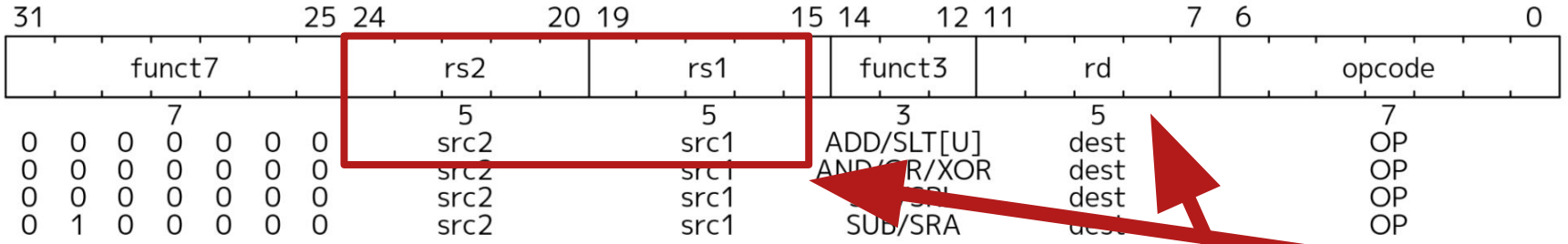
src2: 0..31

src1: 0..31

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

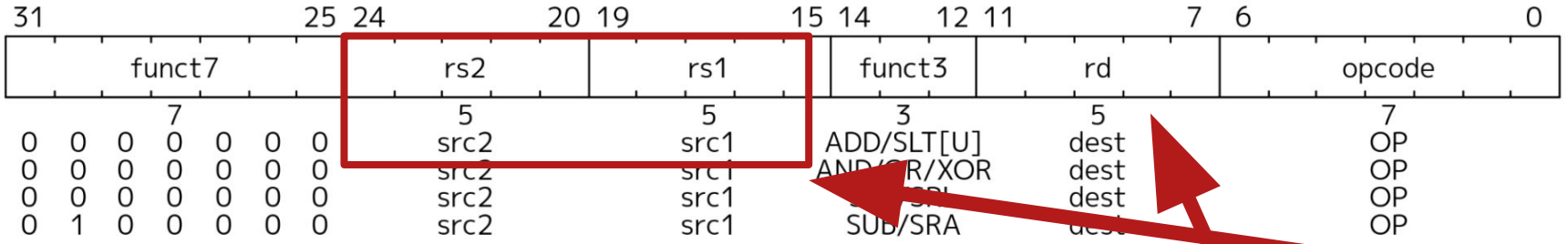
two **read registers**
src2: 0..31
src1: 0..31

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*.

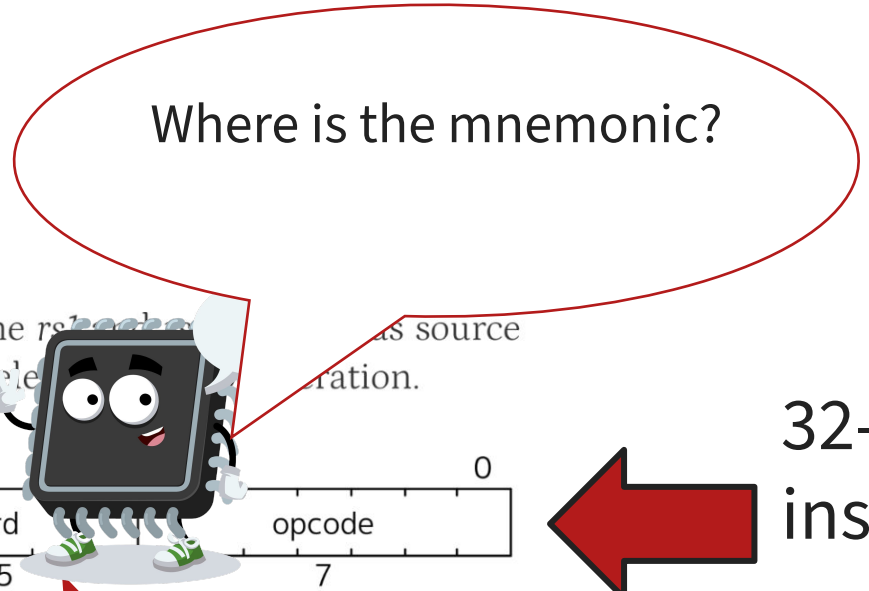
32-bit instruction word

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

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

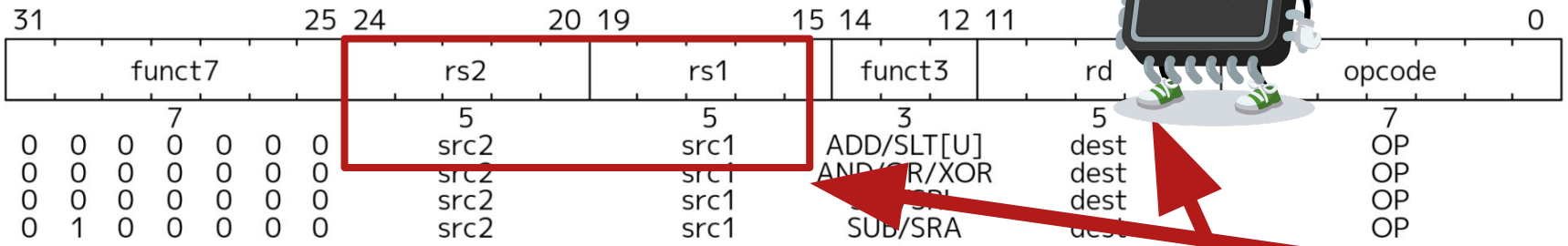
semantics (what the instruction actually does)

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* registers as 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*.

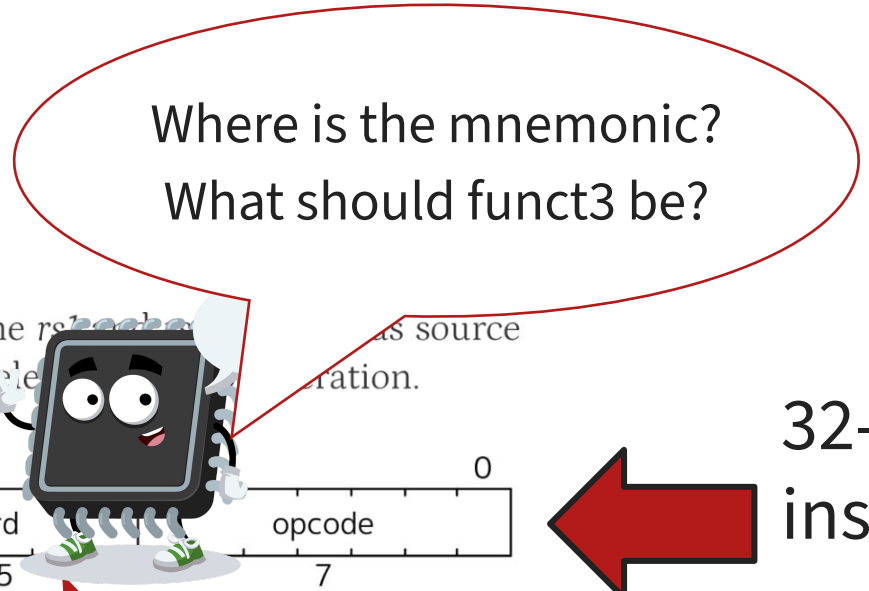
32-bit instruction word

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

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

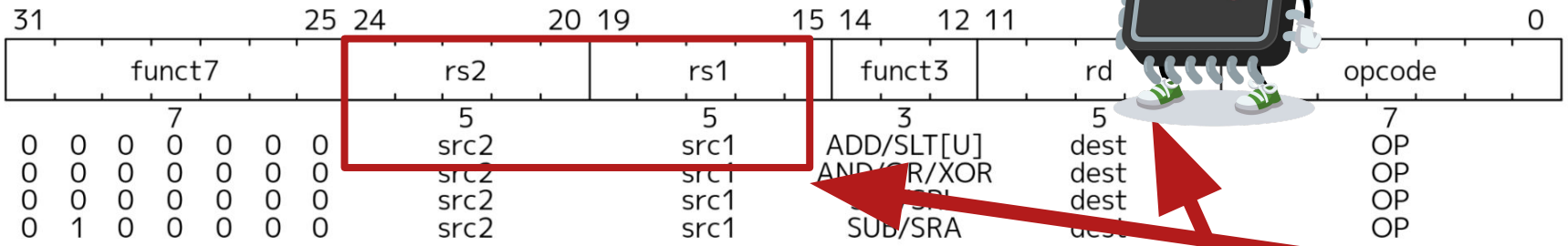
semantics (what the instruction actually does)

The RISC-V Add Instruction



2.4.2. Integer Register-Register Operations

RV32I defines several arithmetic R-type operations. All operations read the *rs1* [illegible] source operands and write the result into register *rd*. The *funct7* and *funct3* fields s[illegible]ration.



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

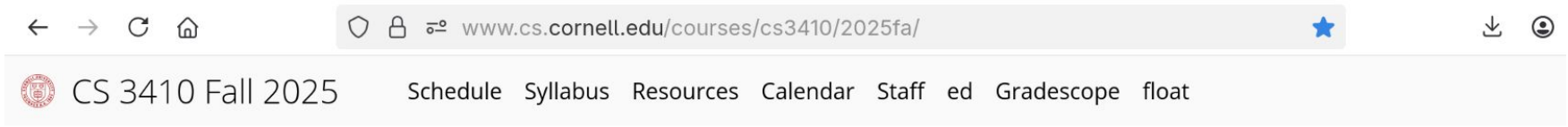
one **write** (destination) **registers**
dest: 0..31

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

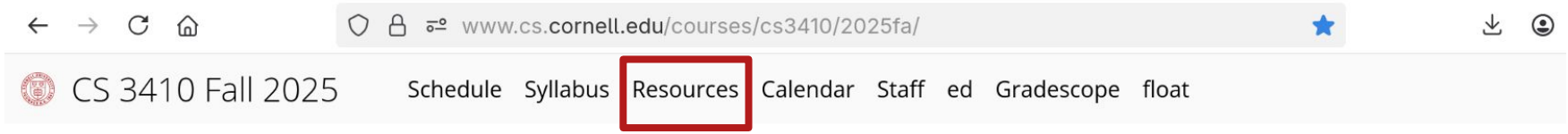
CS 3410 - Fall 2025 taught by Giulia Guidi and Kevin Laeufer

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

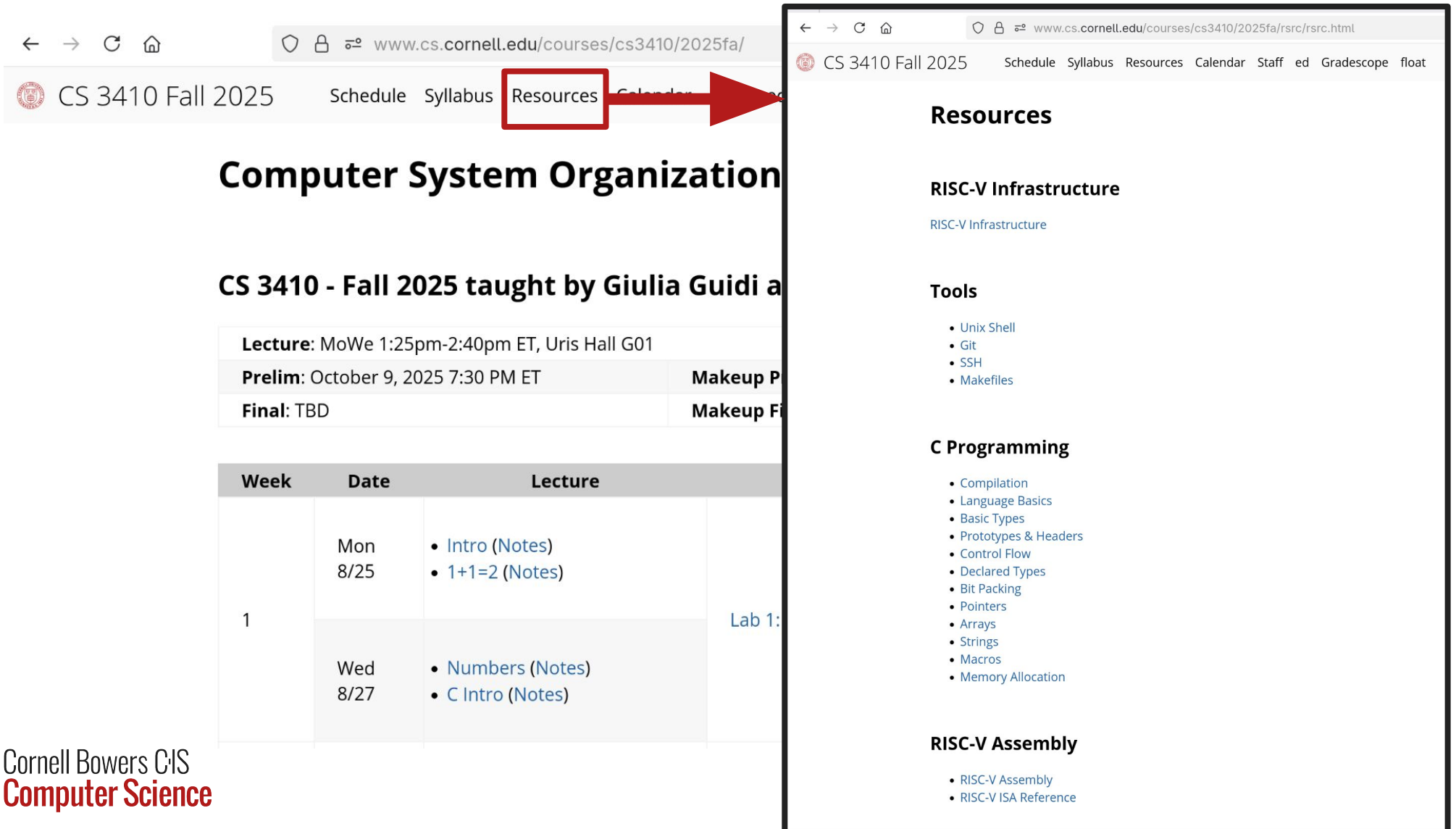
CS 3410 - Fall 2025 taught by Giulia Guidi and Kevin Laeufer

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)		

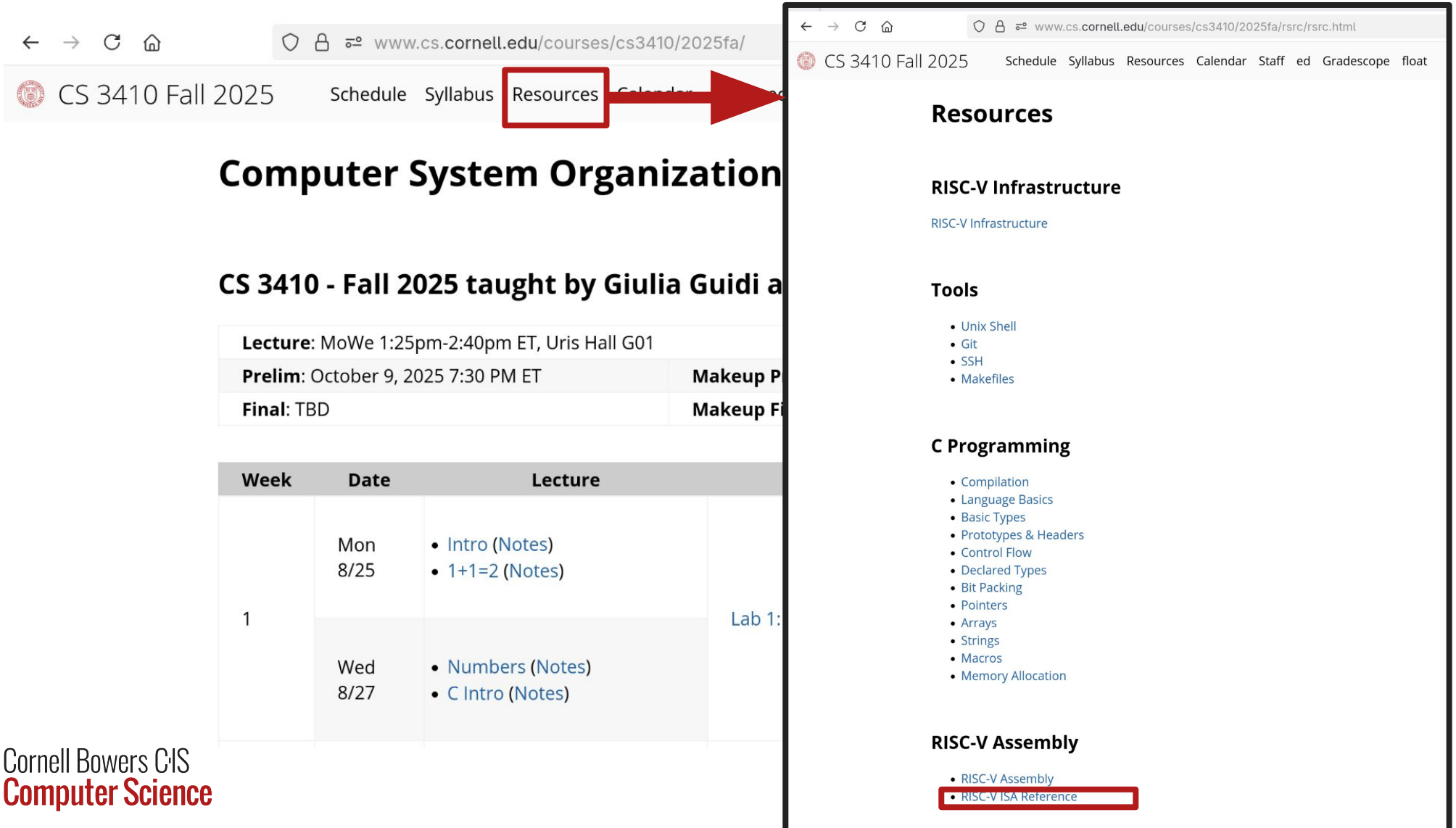
The RISC-V Add Instruction: 2nd Try

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



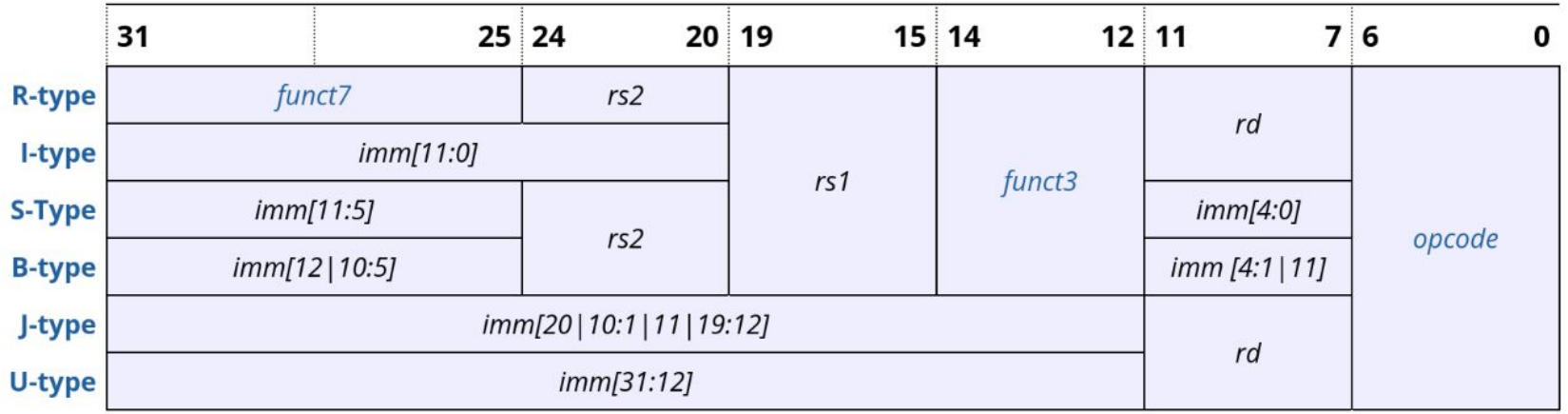
The RISC-V Add Instruction: 2nd Try

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



RISC-V ISA Reference

Instruction Encoding Templates



RV32I Base Integer Instruction Set

Arithmetic Instructions

Instruction	Name	Description	Opcode	Funct3	Funct7
`add rd rs1 rs2`	*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/

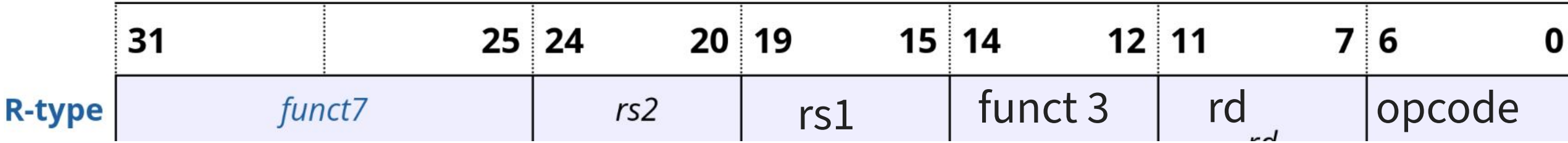
Instruction	Name	Description	Opcode	Funct 3	Funct7
`add rd rs1 rs2`	*ADD*	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	*SUBTRACT*	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`

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

The RISC-V Add Instruction: 2nd Try

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

Instruction	Name	Description	Opcode	Funct 3	Funct7
`add rd rs1 rs2`	*ADD*	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	*SUBTRACT*	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`

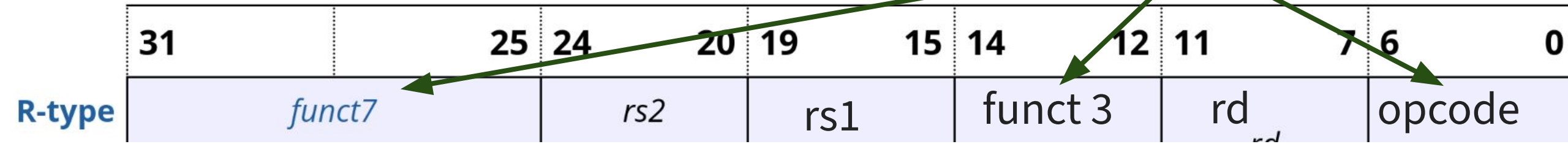


Assembling machine code

The RISC-V Add Instruction: 2nd Try

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

Instruction	Name	Description	Opcode	Funct 3	Funct7
`add rd rs1 rs2`	*ADD*	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	*SUBTRACT*	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`



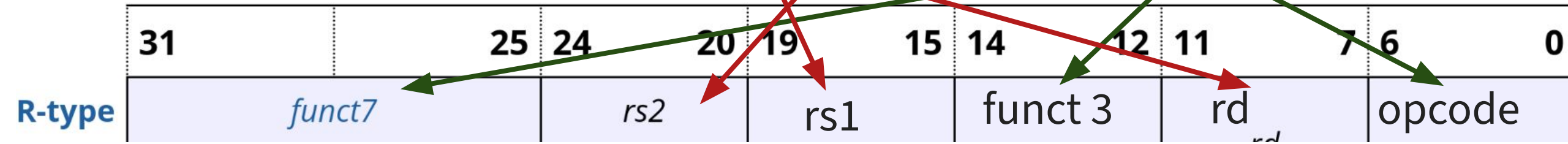
Assembling machine code

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
`add rd rs1 rs2`	*ADD*	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	*SUBTRACT*	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`



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
`add rd rs1 rs2`	*ADD*	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	*SUBTRACT*	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`

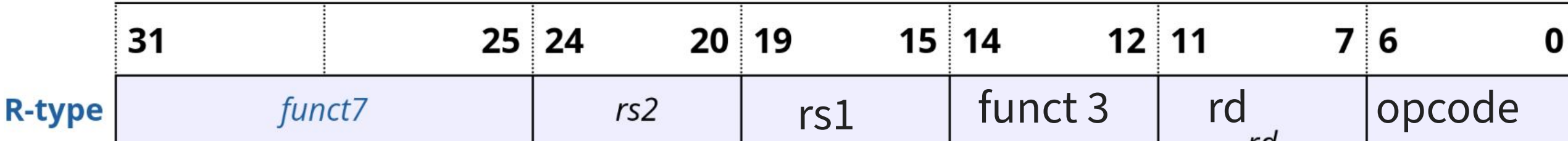
	31 25	24 20	19 15	14 12	11 7	6 0
R-type	*funct7*	*rs2*	rs1	funct 3	rd	opcode

Semantics: How do I execute the instruction?

The RISC-V Add Instruction: 2nd Try

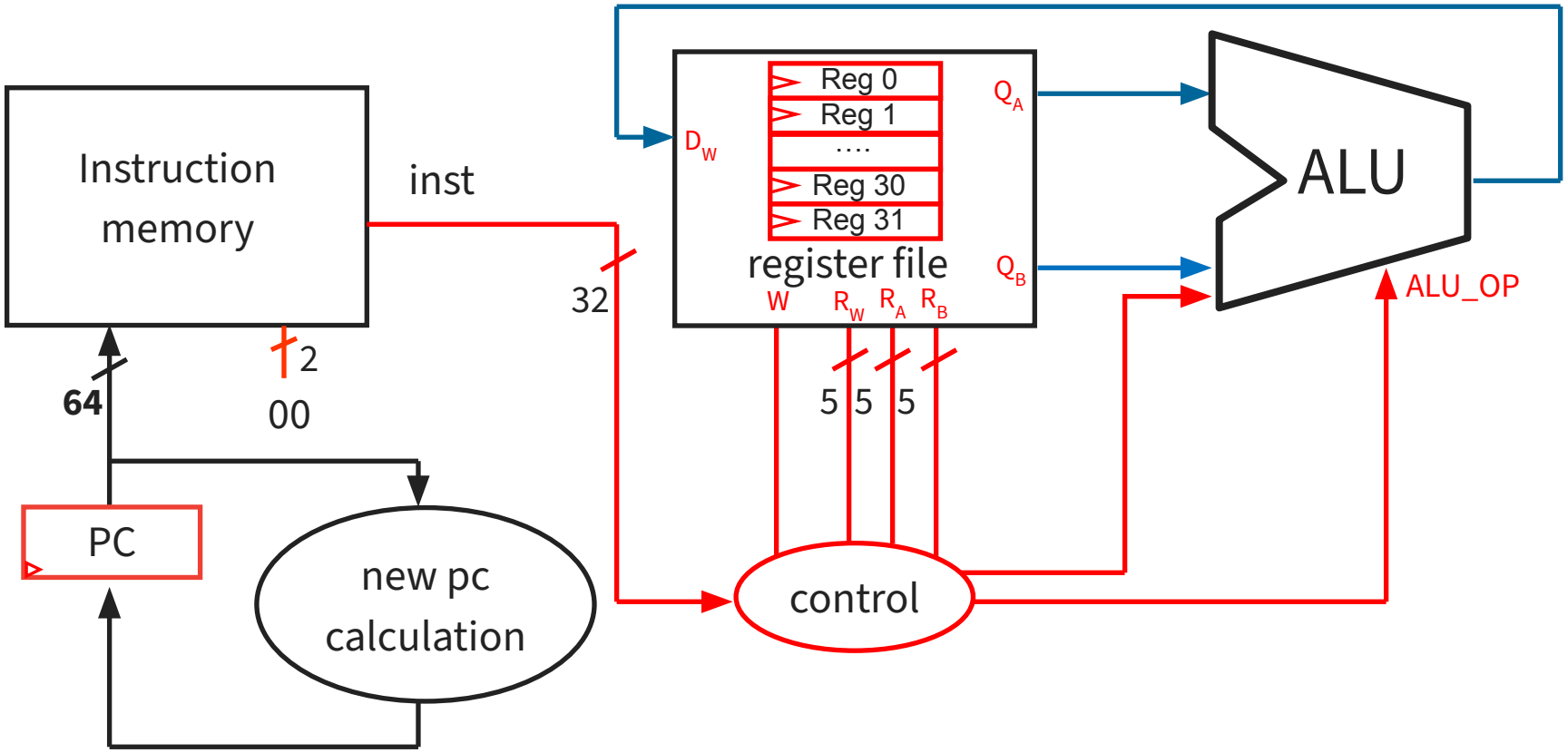
https://www.cs.cornell.edu/courses/cs3410/2025fa/

Instruction	Name	Description	Opcode	Funct 3	Funct7
`add rd rs1 rs2`	*ADD*	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	*SUBTRACT*	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`



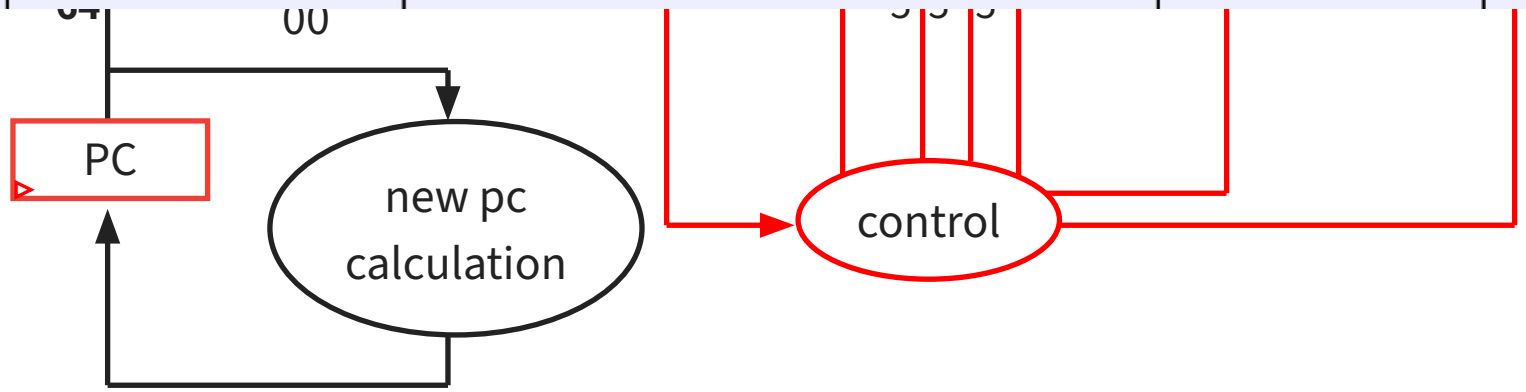
Semantics: How do I execute the instruction?

Executing `add x10, x10, x11`



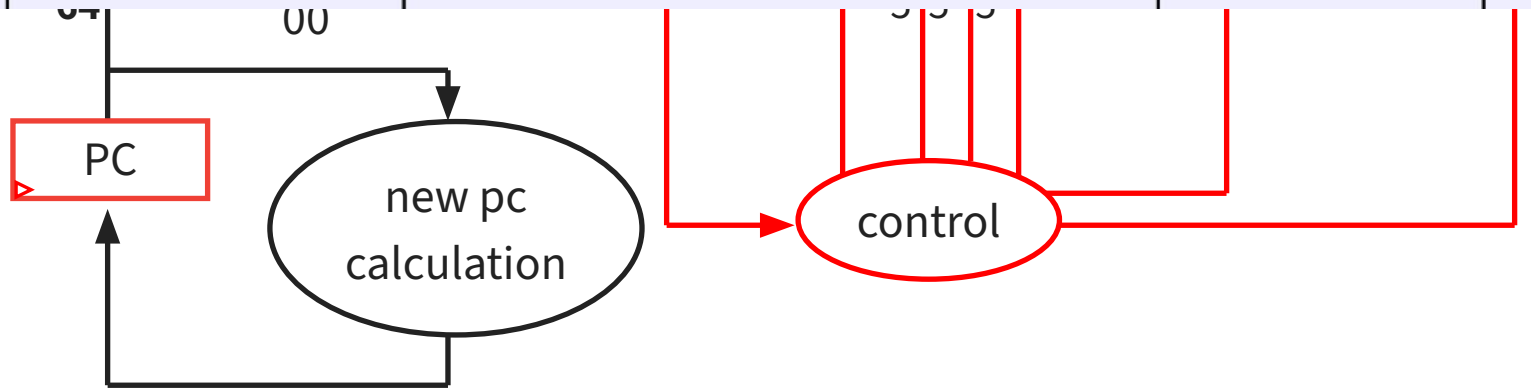
Executing `add x10, x10, x11`

Instruction	Name	Description	Opcode	Funct 3	Funct7
`add rd rs1 rs2`	*ADD*	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	*SUBTRACT*	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`



Executing `add x10, x10, x11`

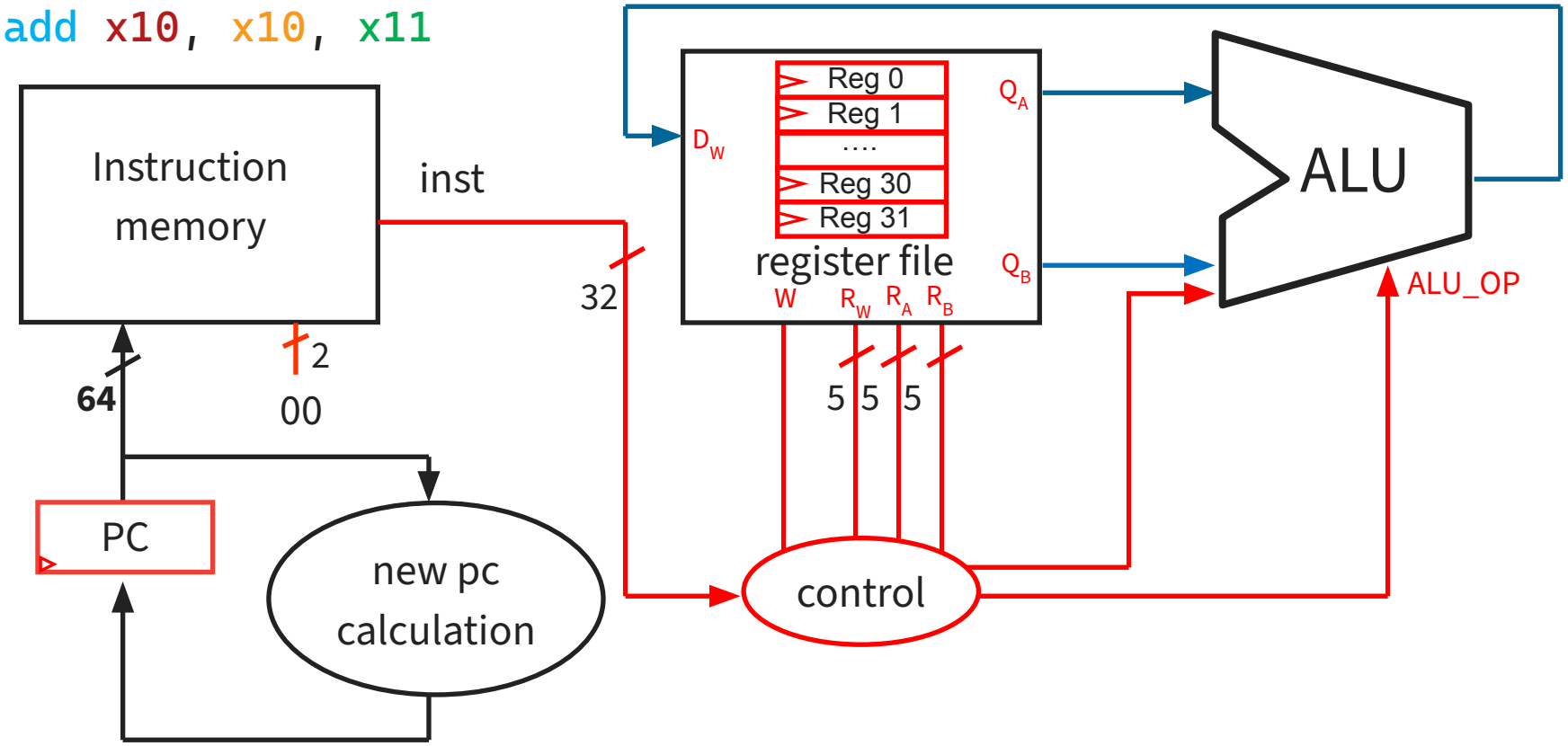
Instruction	Name	Description	Opcode	Funct 3	Funct7
`add rd rs1 rs2`	*ADD*	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
`sub rd rs1 rs2`	*SUBTRACT*	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`



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

Executing `add x10, x10, x11`

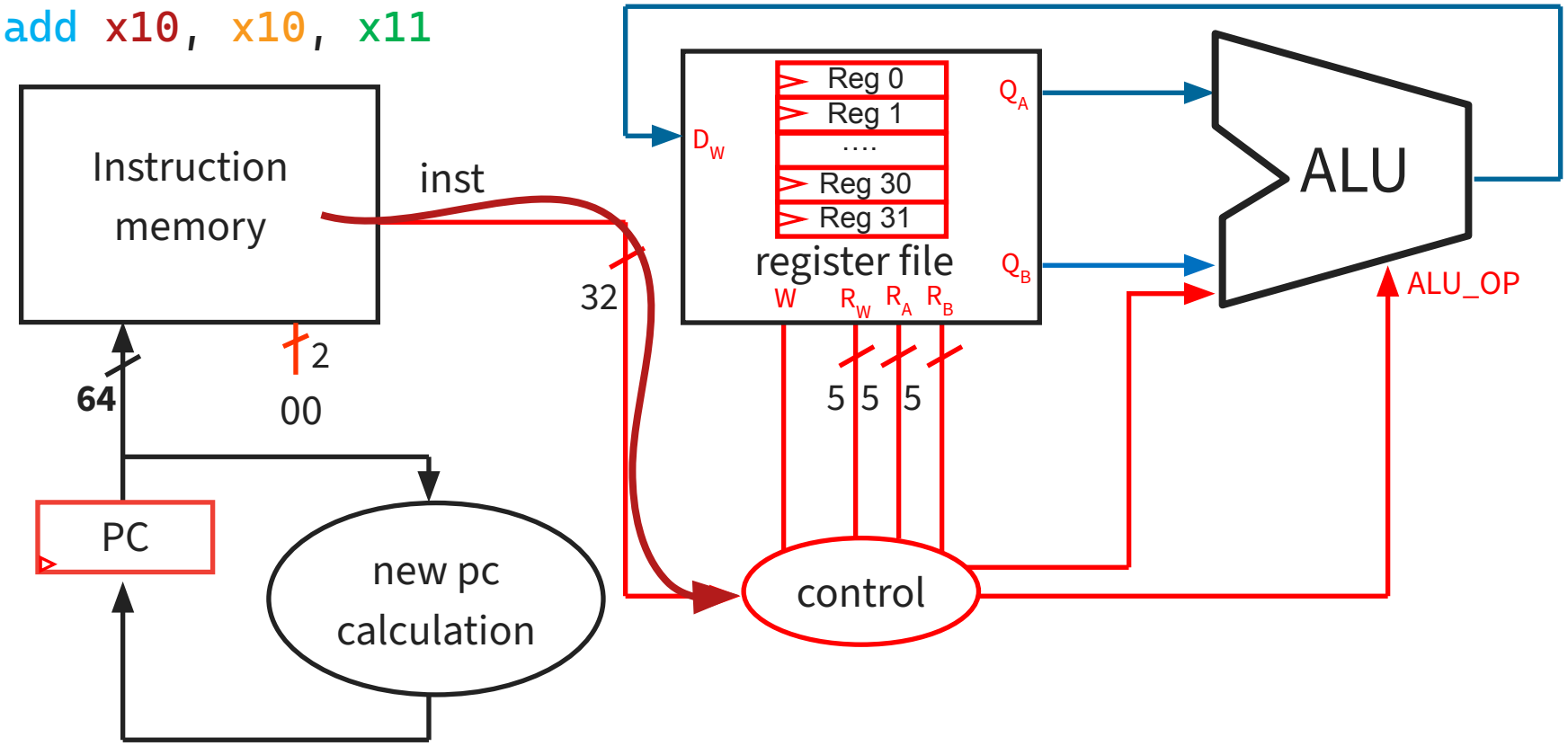
1. `add x10, x10, x11`



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

Executing `add x10, x10, x11`

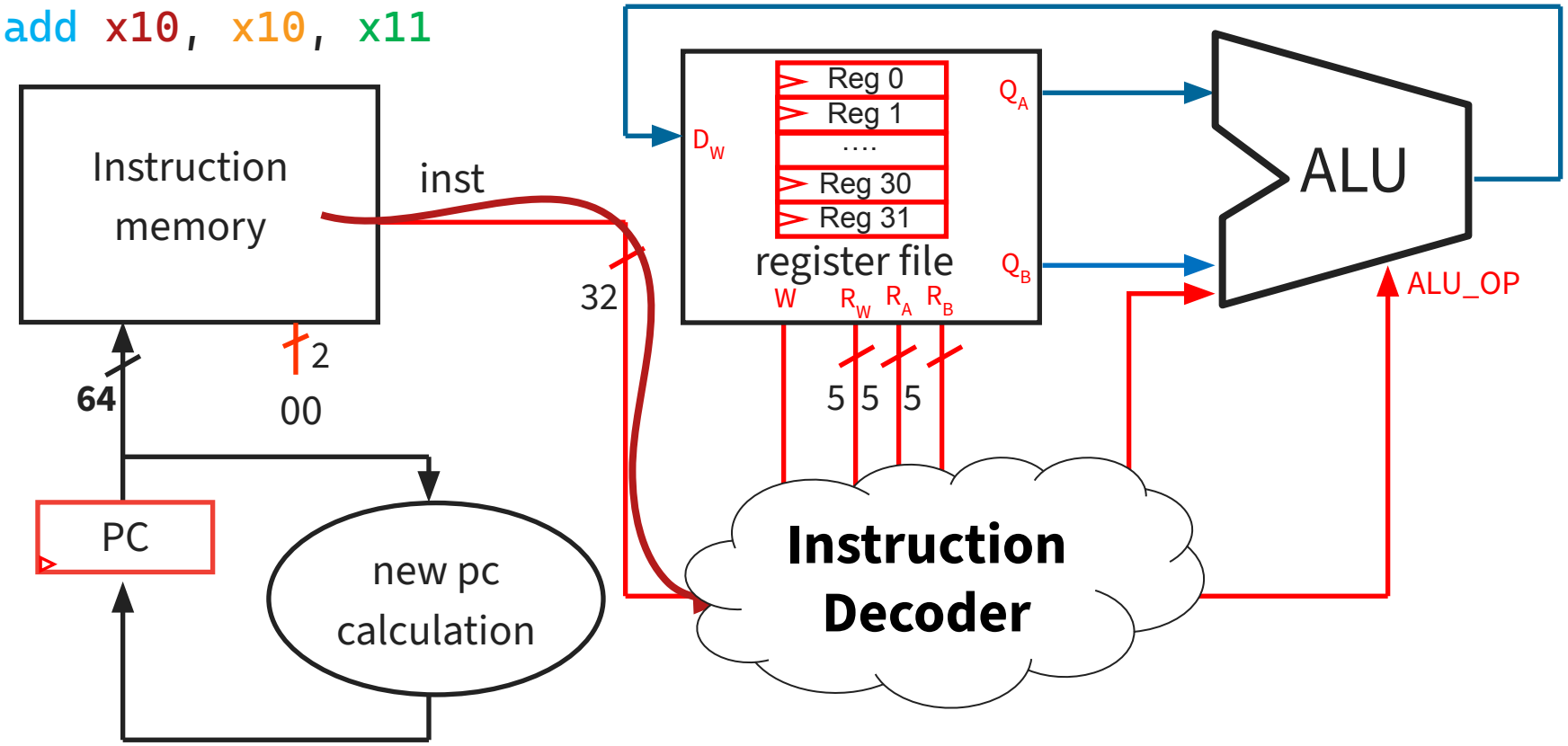
1. `add x10, x10, x11`



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

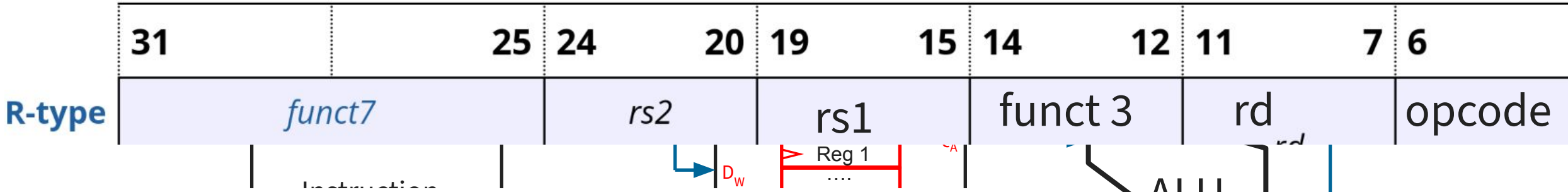
Executing `add x10, x10, x11`

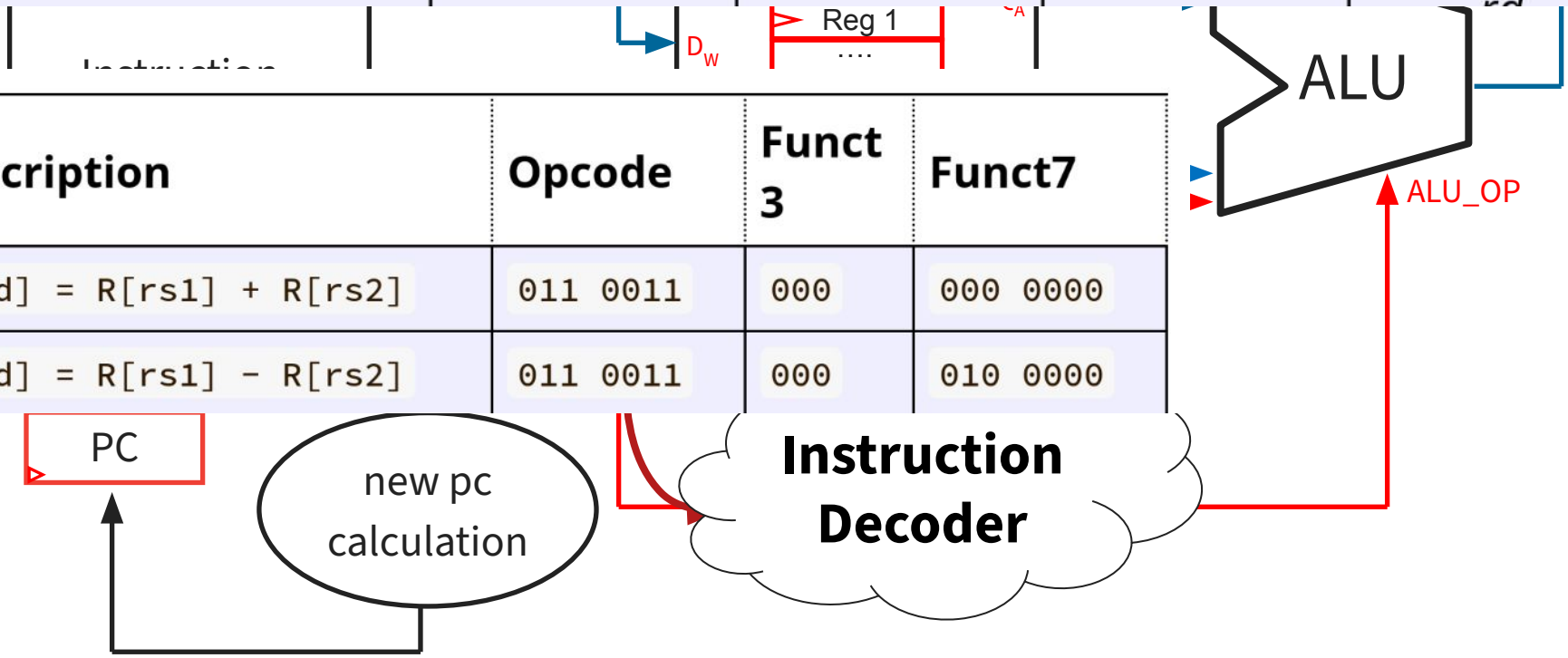
1. `add x10, x10, x11`



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

Executing `add x10, x10, x11`

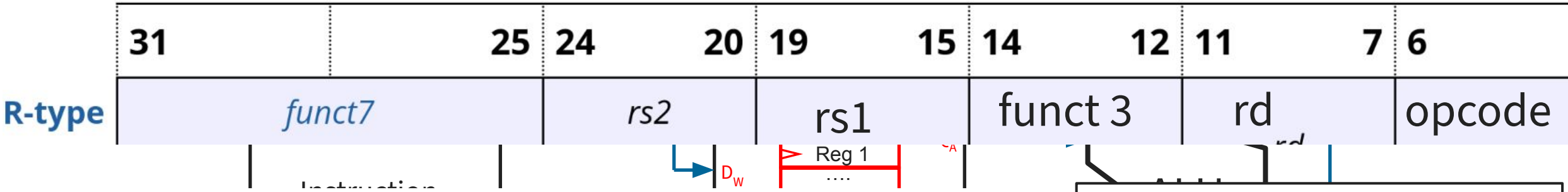




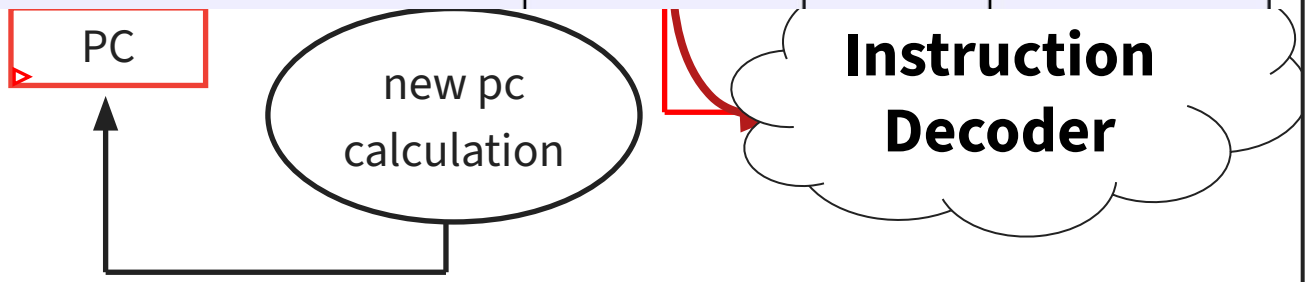
Name	Description	Opcode	Funct 3	Funct7
ADD	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
SUBTRACT	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`

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

Executing `add x10, x10, x11`



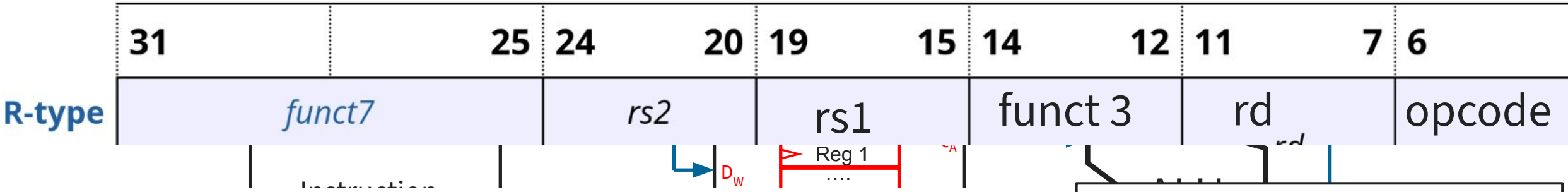
Name	Description	Opcode	Funct 3	Funct7
ADD	`R[rd] = R[rs1] + R[rs2]`	`011 0011`	`000`	`000 0000`
SUBTRACT	`R[rd] = R[rs1] - R[rs2]`	`011 0011`	`000`	`010 0000`



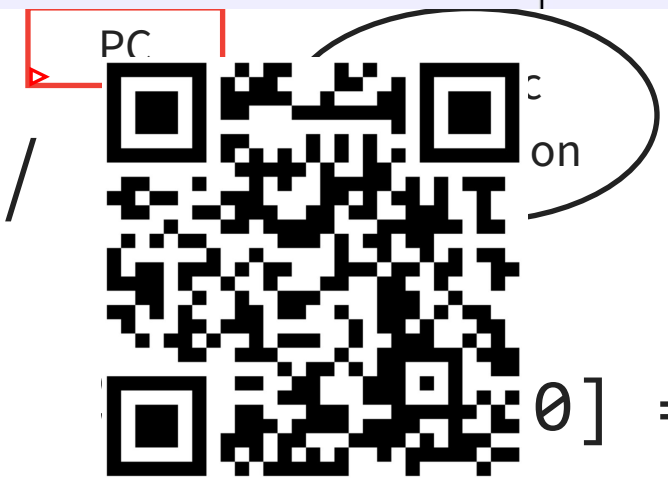
Recognize add

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`



Instruction Decoder

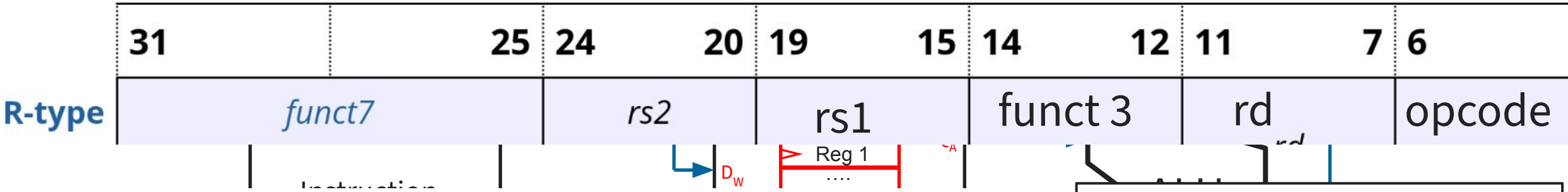
`0] = R[10] + R[11]`

PollEv.com/cs3410

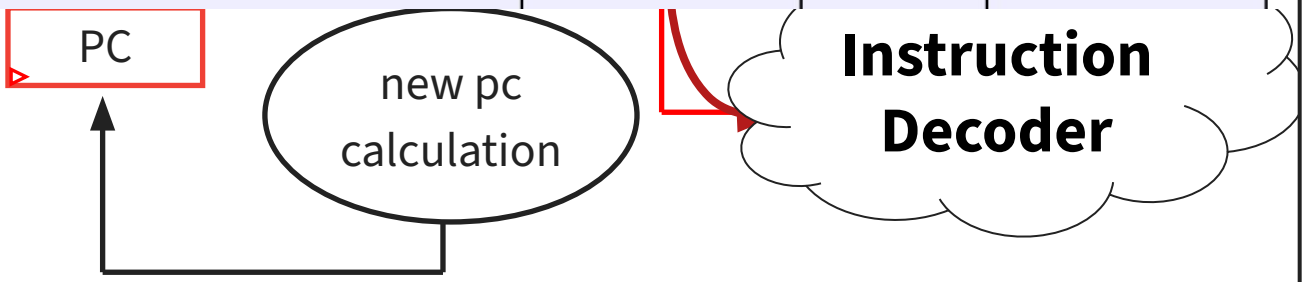
Recognize add

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

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`

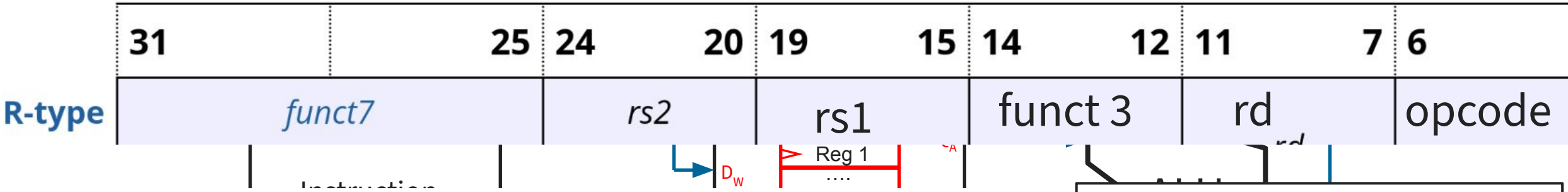


Recognize add

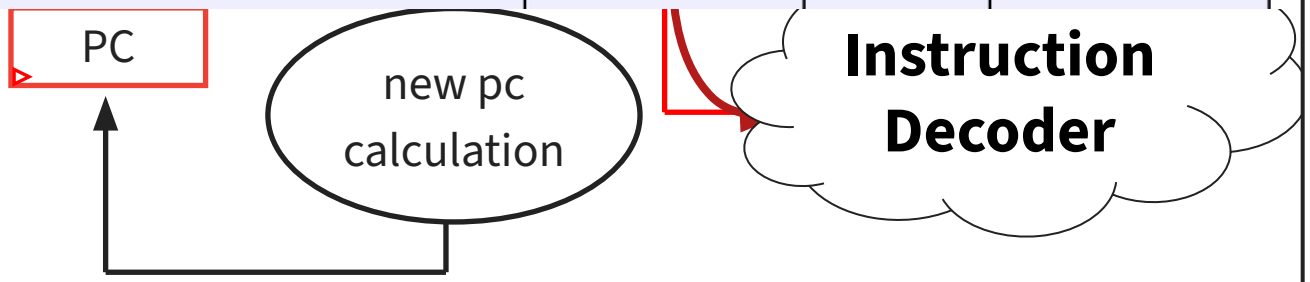
- inst[31:25] = funct7 = 0

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`

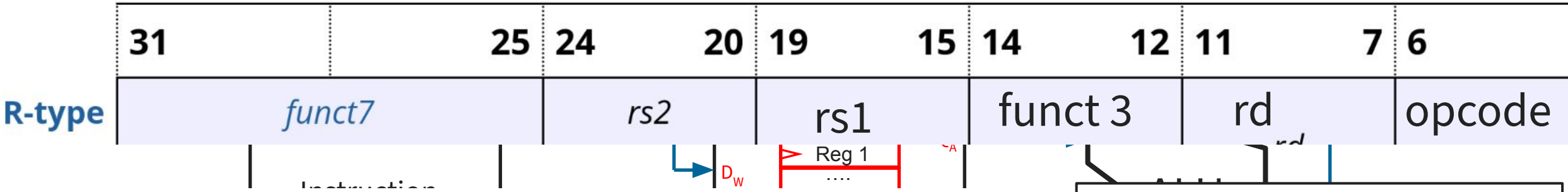


Recognize add

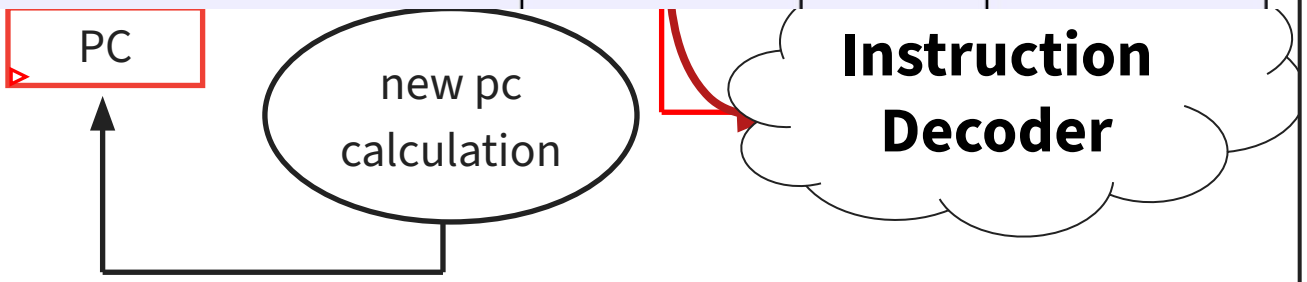
- inst[31:25] = funct7 = 0
- inst[14:12] = funct3 = 0

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

Executing `add x10, x10, x11`



Name	Description	Opcode	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`

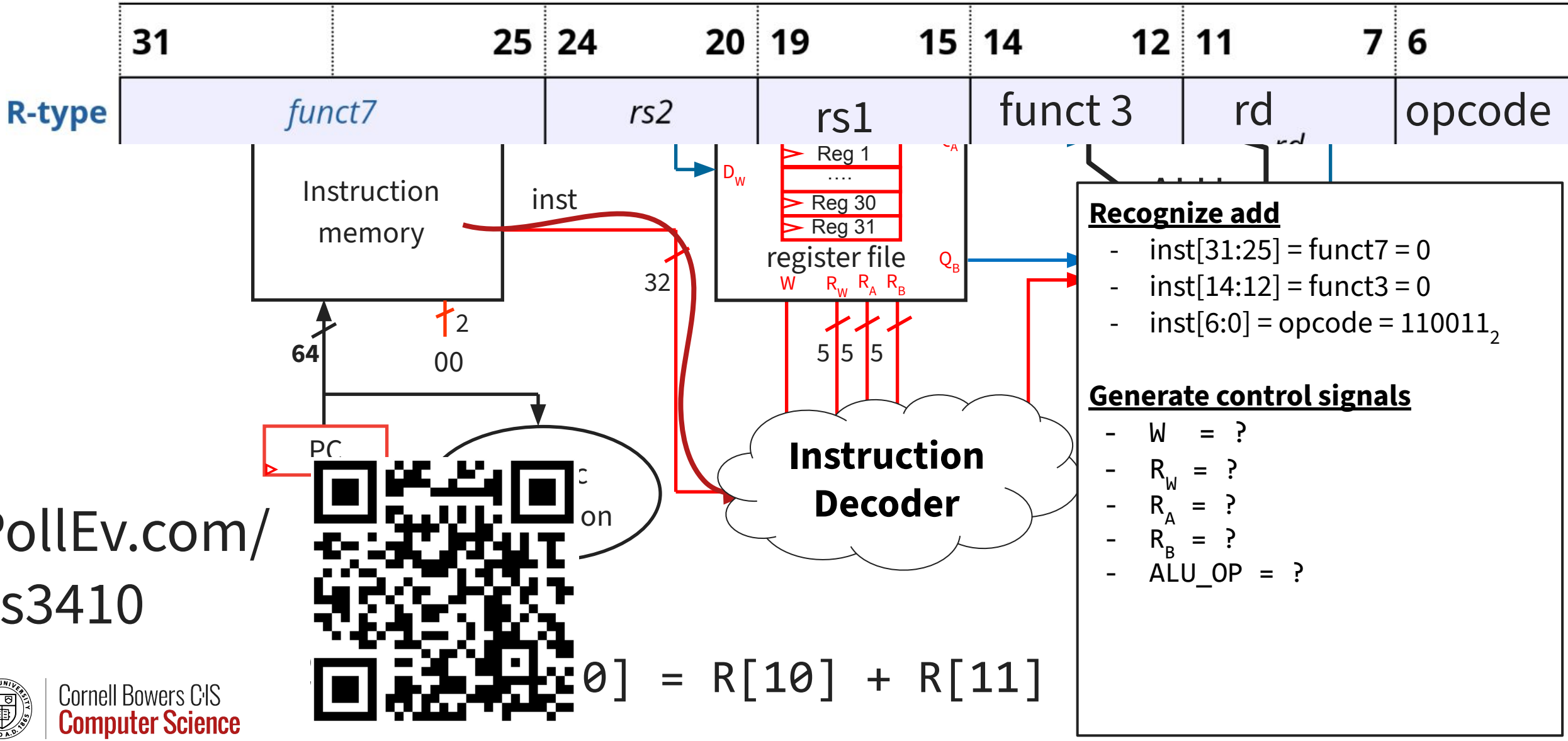


Recognize add

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

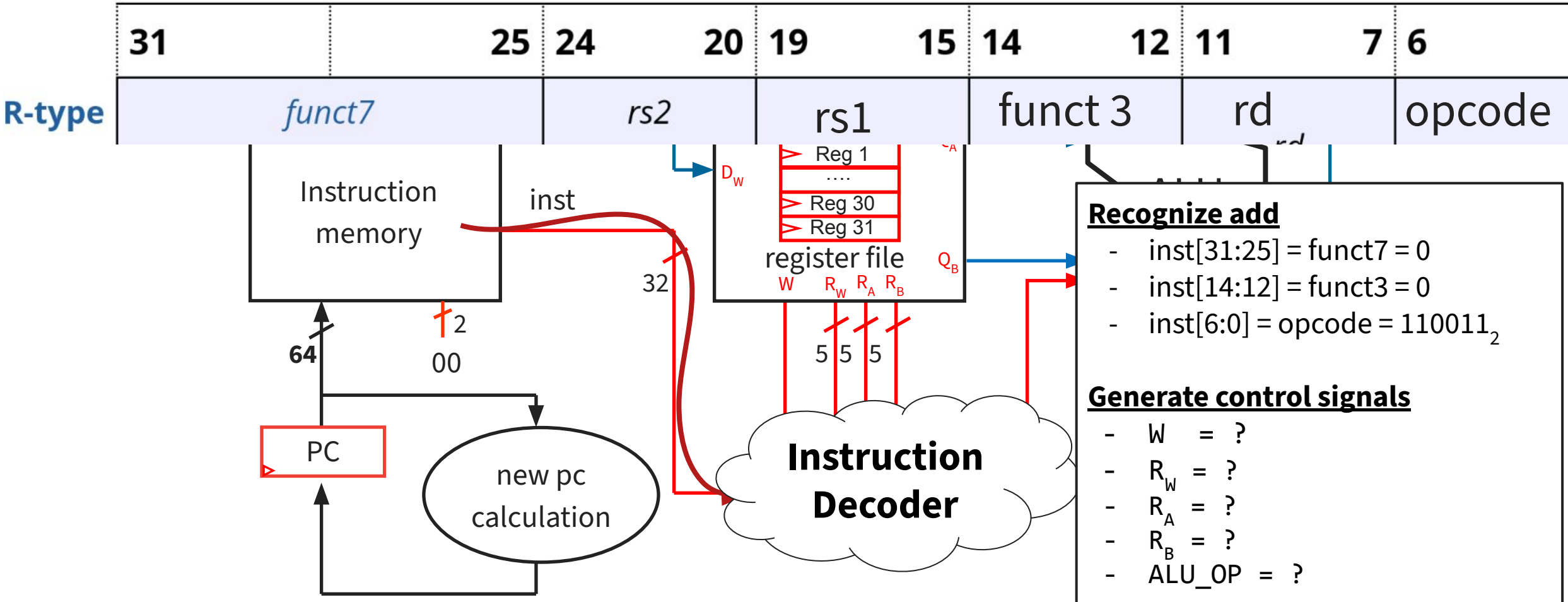
Semantics: `R[10] = R[10] + R[11]`

Executing `add x10, x10, x11`



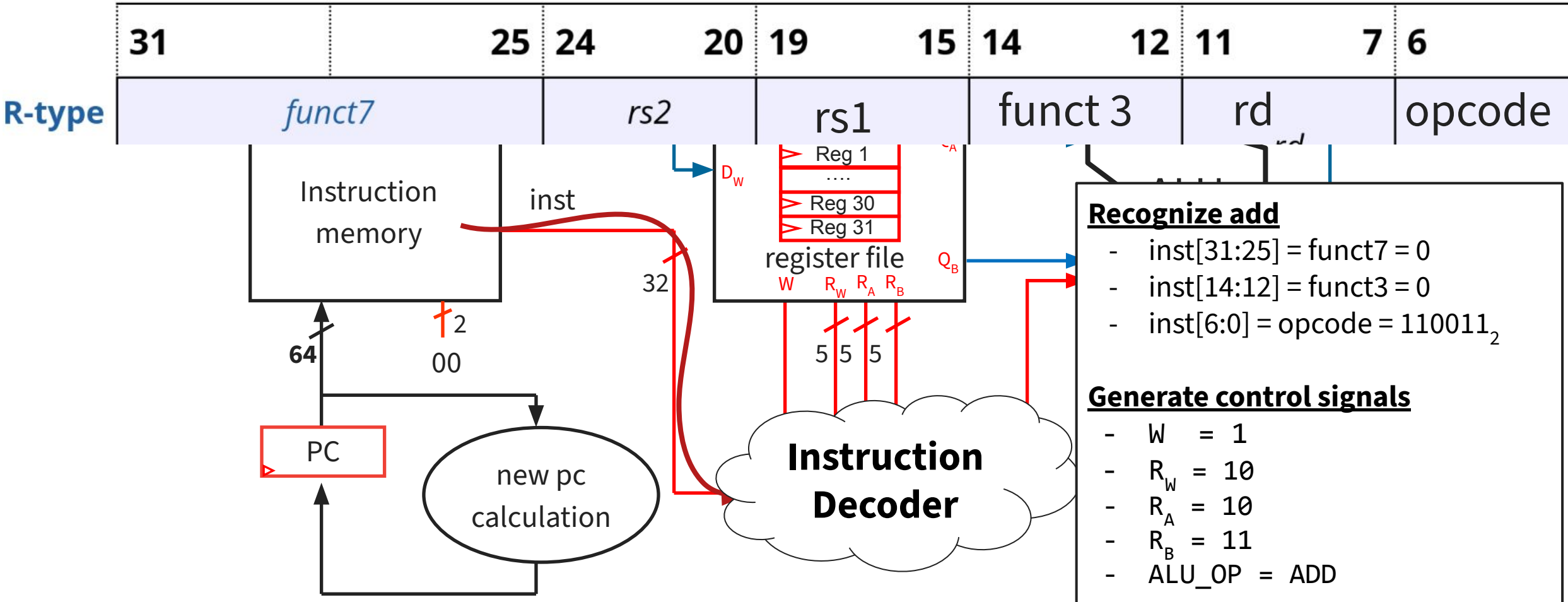
PollEv.com/cs3410

Executing `add x10, x10, x11`



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

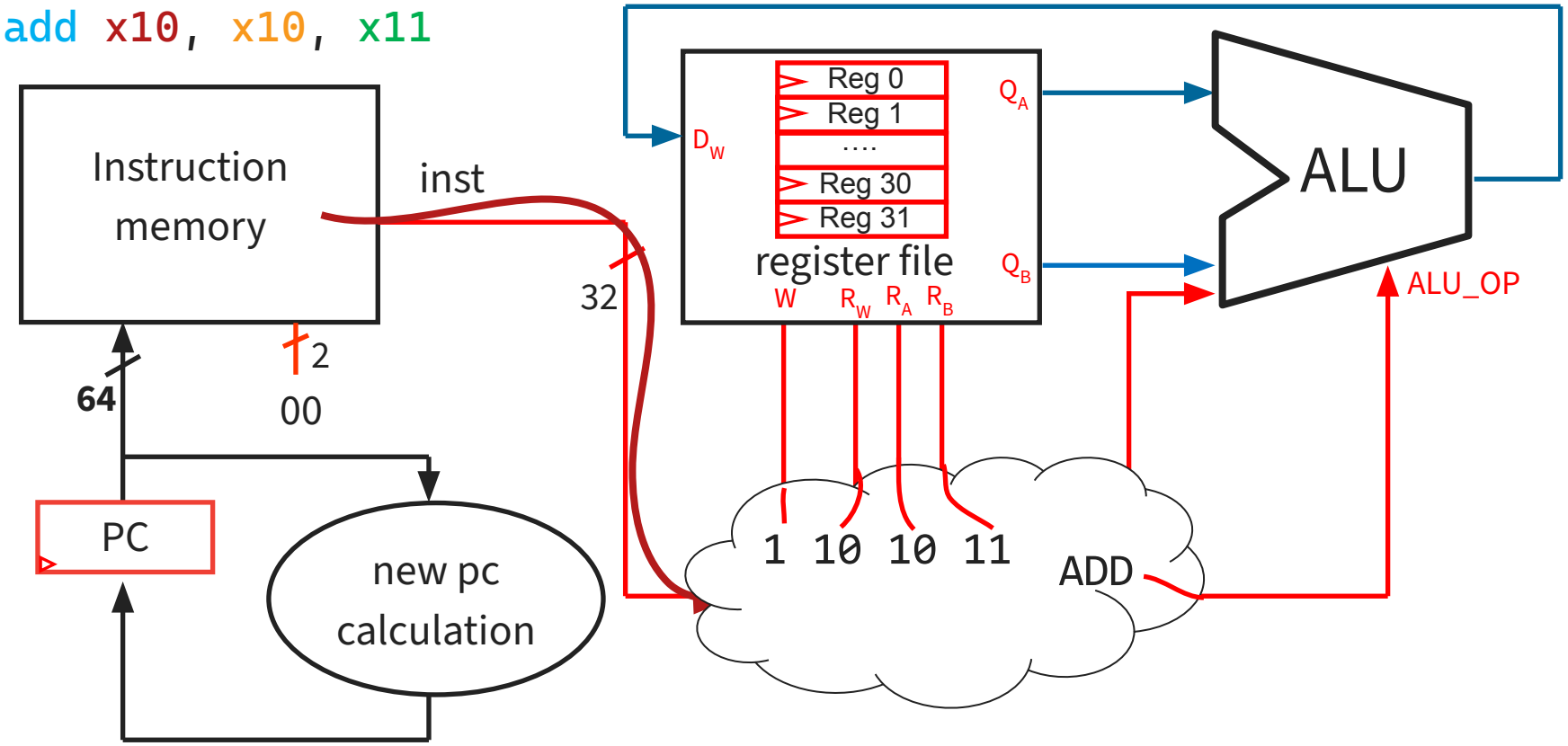
Executing `add x10, x10, x11`



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

Executing `add x10, x10, x11`

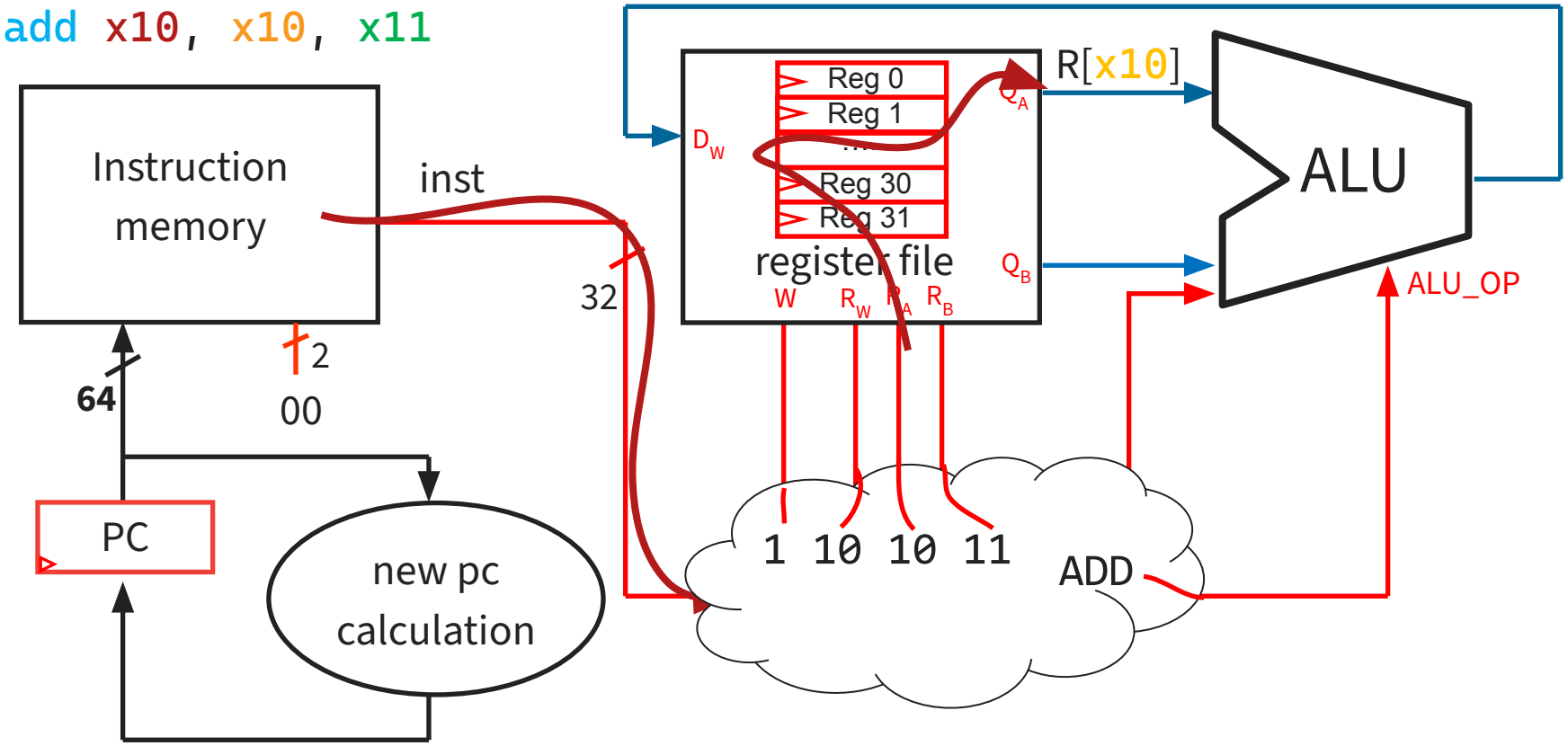
1. `add x10, x10, x11`



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

Executing `add x10, x10, x11`

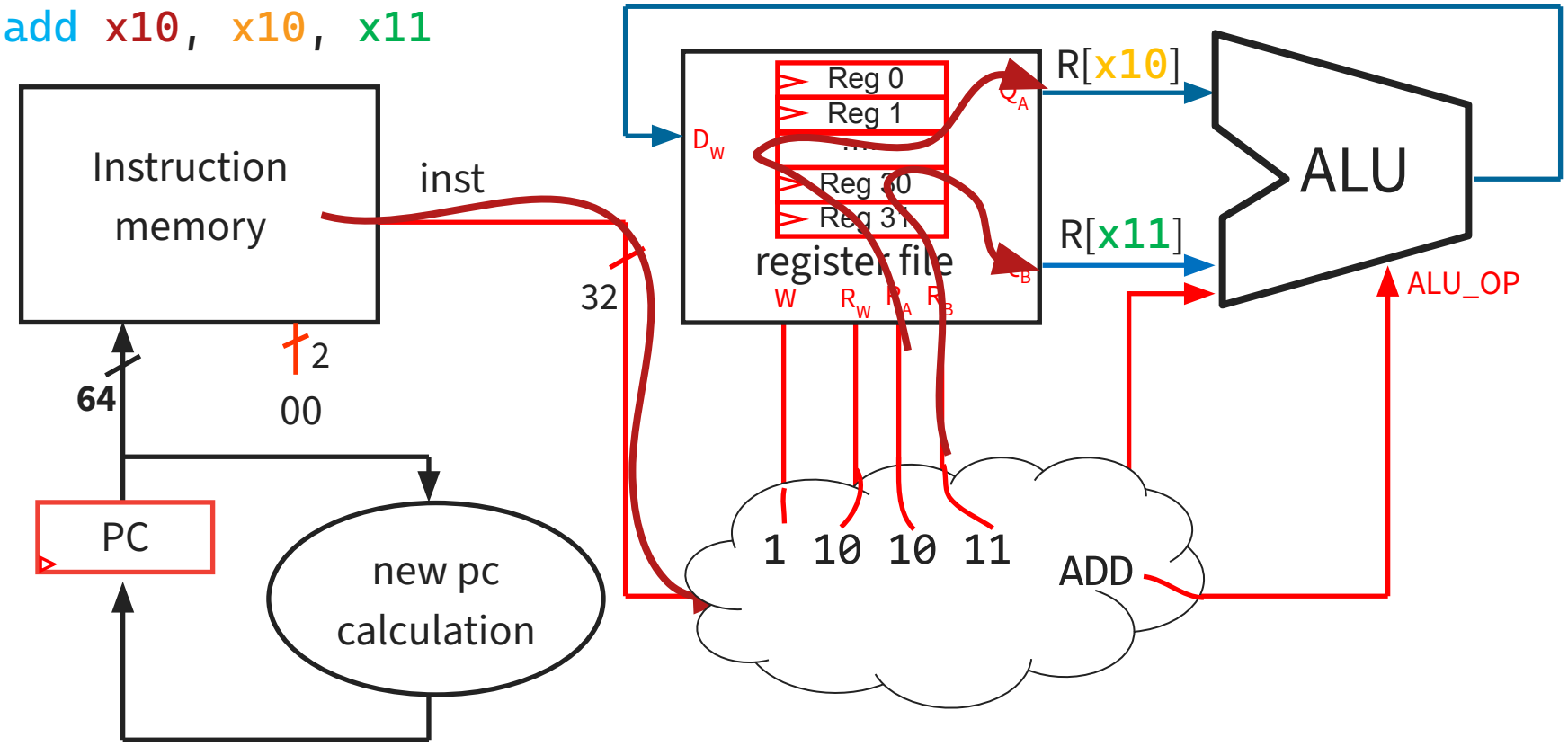
1. `add x10, x10, x11`



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

Executing `add x10, x10, x11`

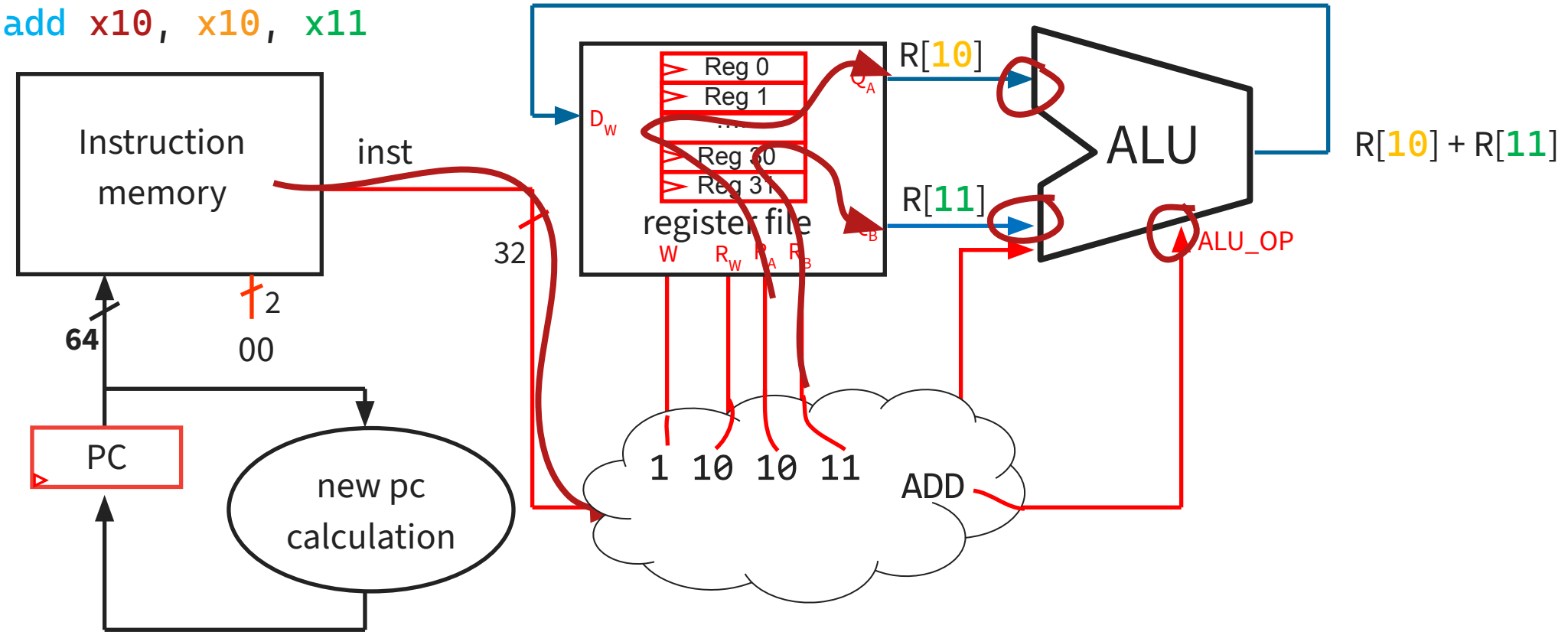
1. `add x10, x10, x11`



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

Executing `add x10, x10, x11`

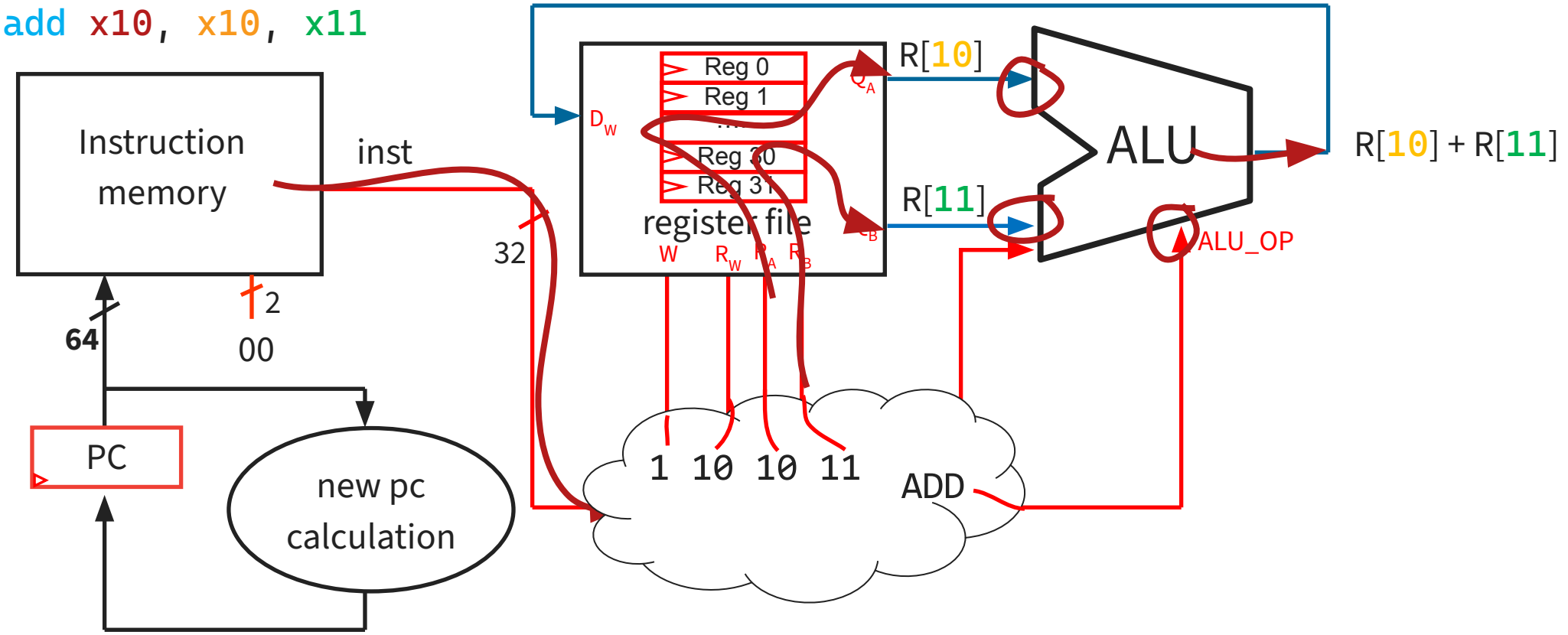
1. `add x10, x10, x11`



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

Executing `add x10, x10, x11`

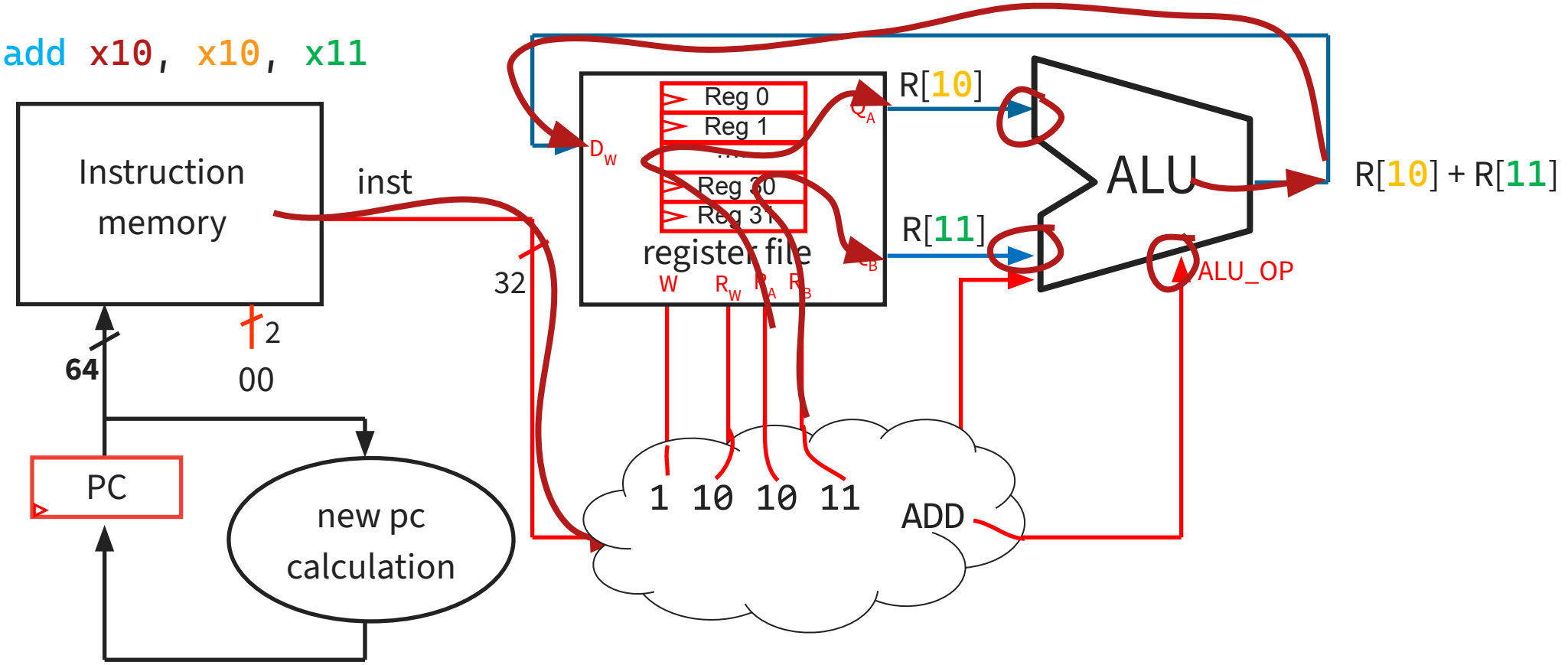
1. `add x10, x10, x11`



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

Executing `add x10, x10, x11`

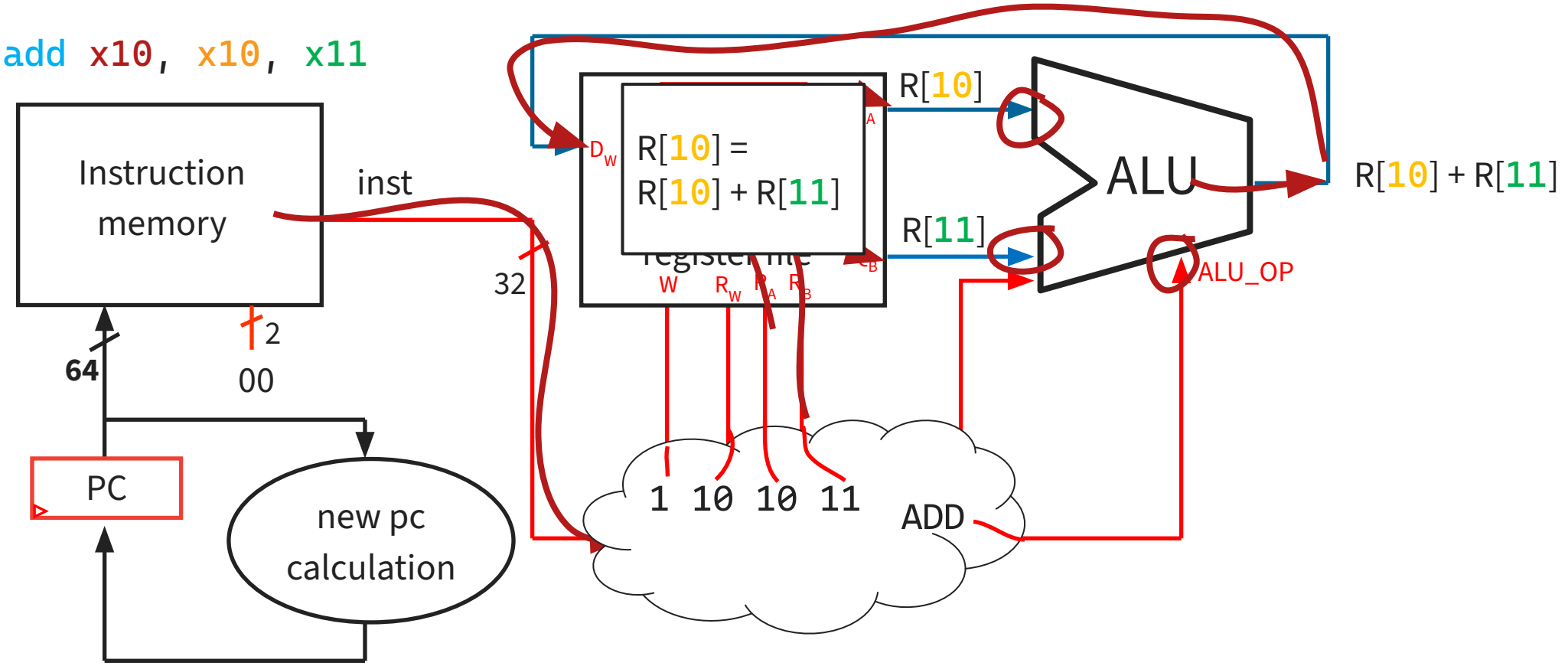
1. `add x10, x10, x11`



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

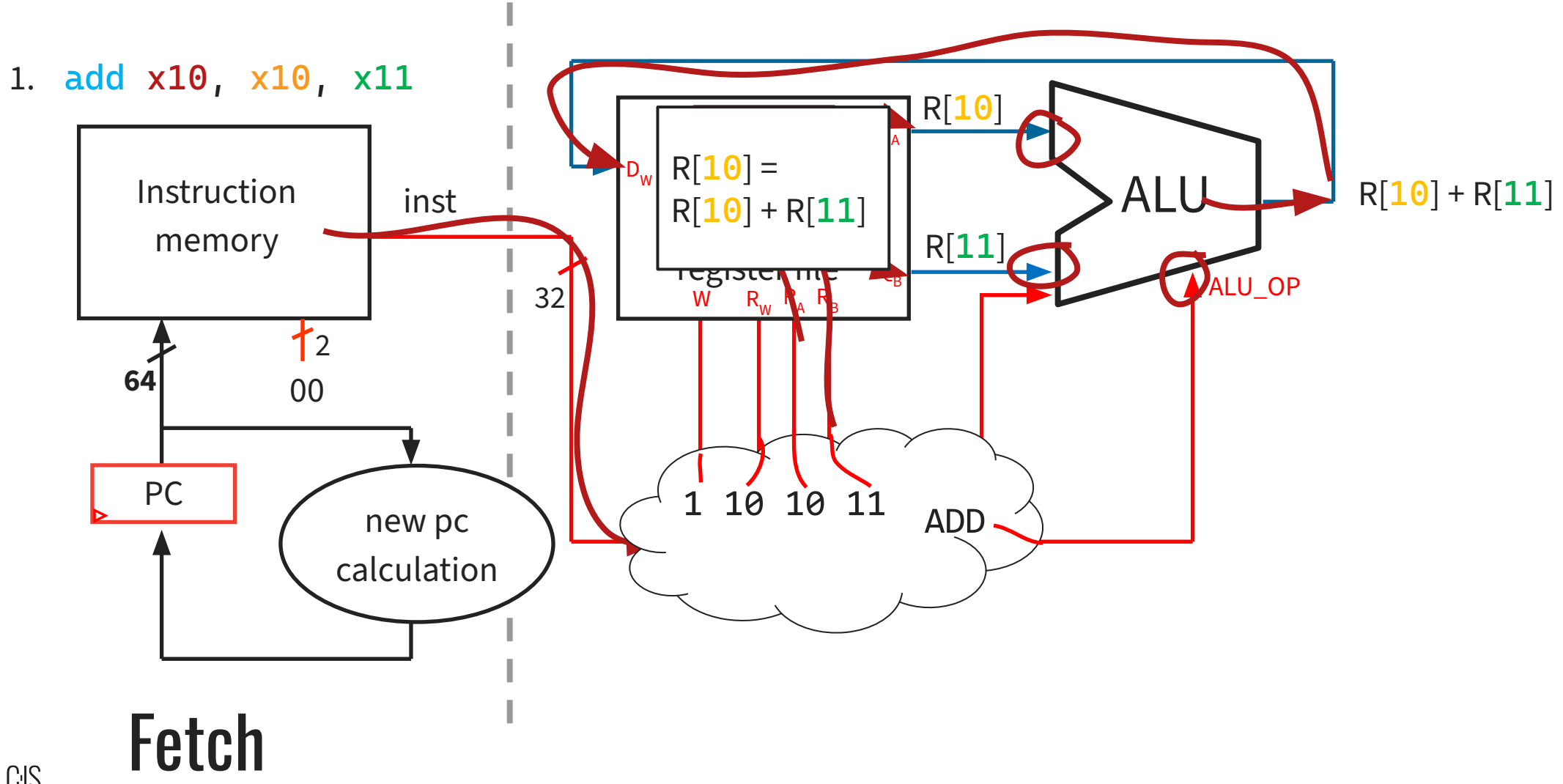
Executing `add x10, x10, x11`

1. `add x10, x10, x11`

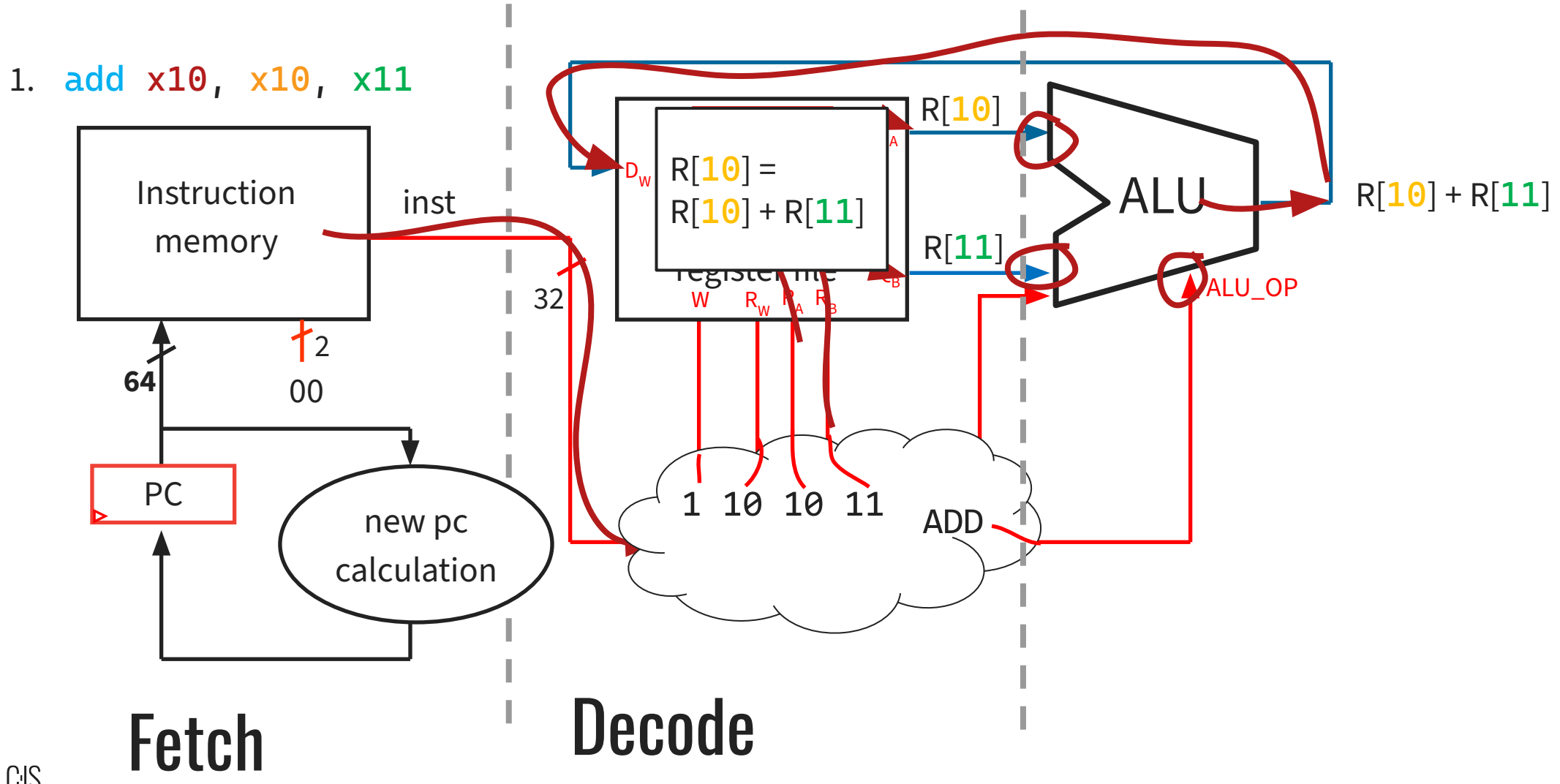


Semantics: `R[10] = R[10] + R[11]`

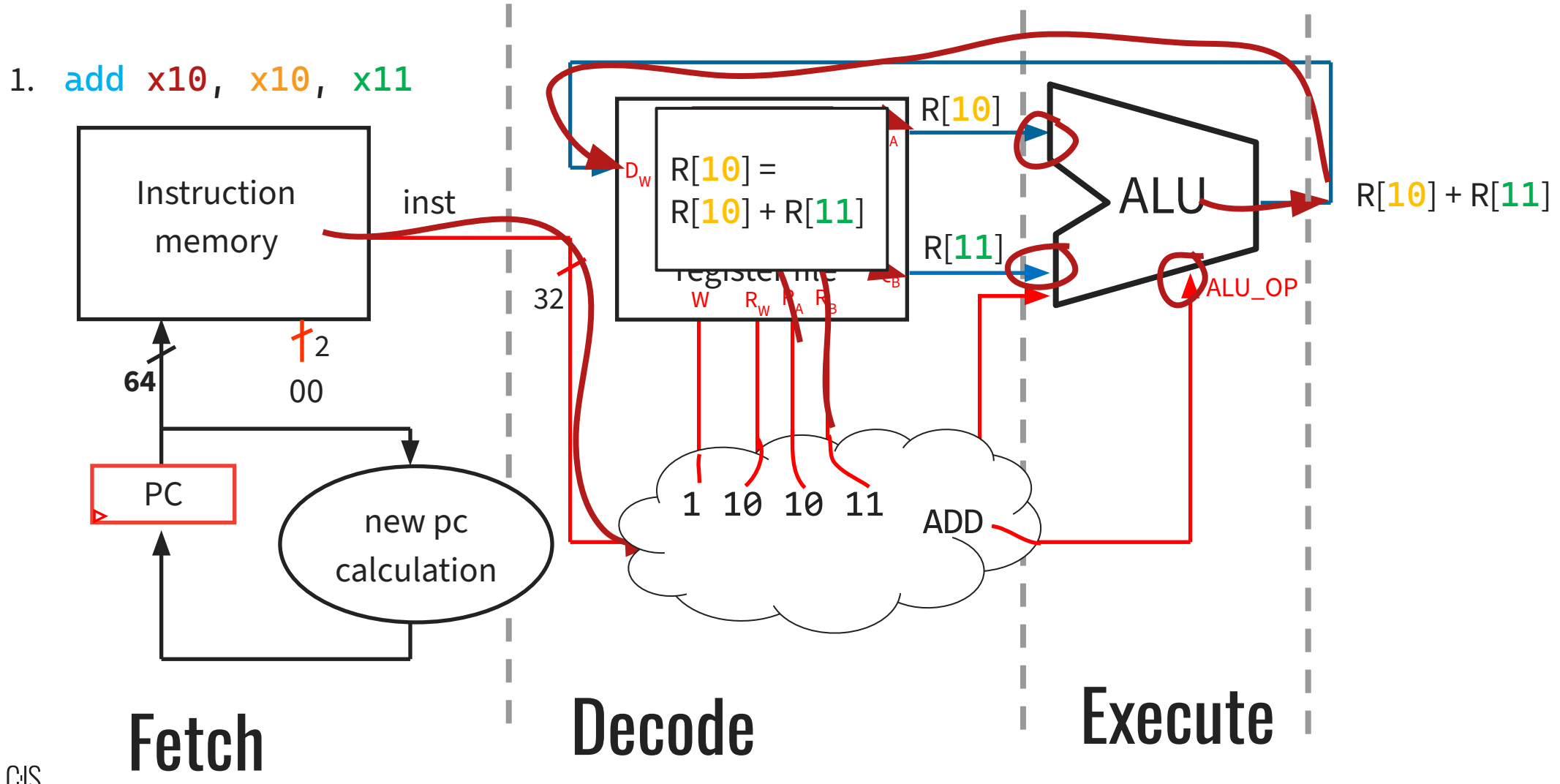
Executing `add x10, x10, x11`



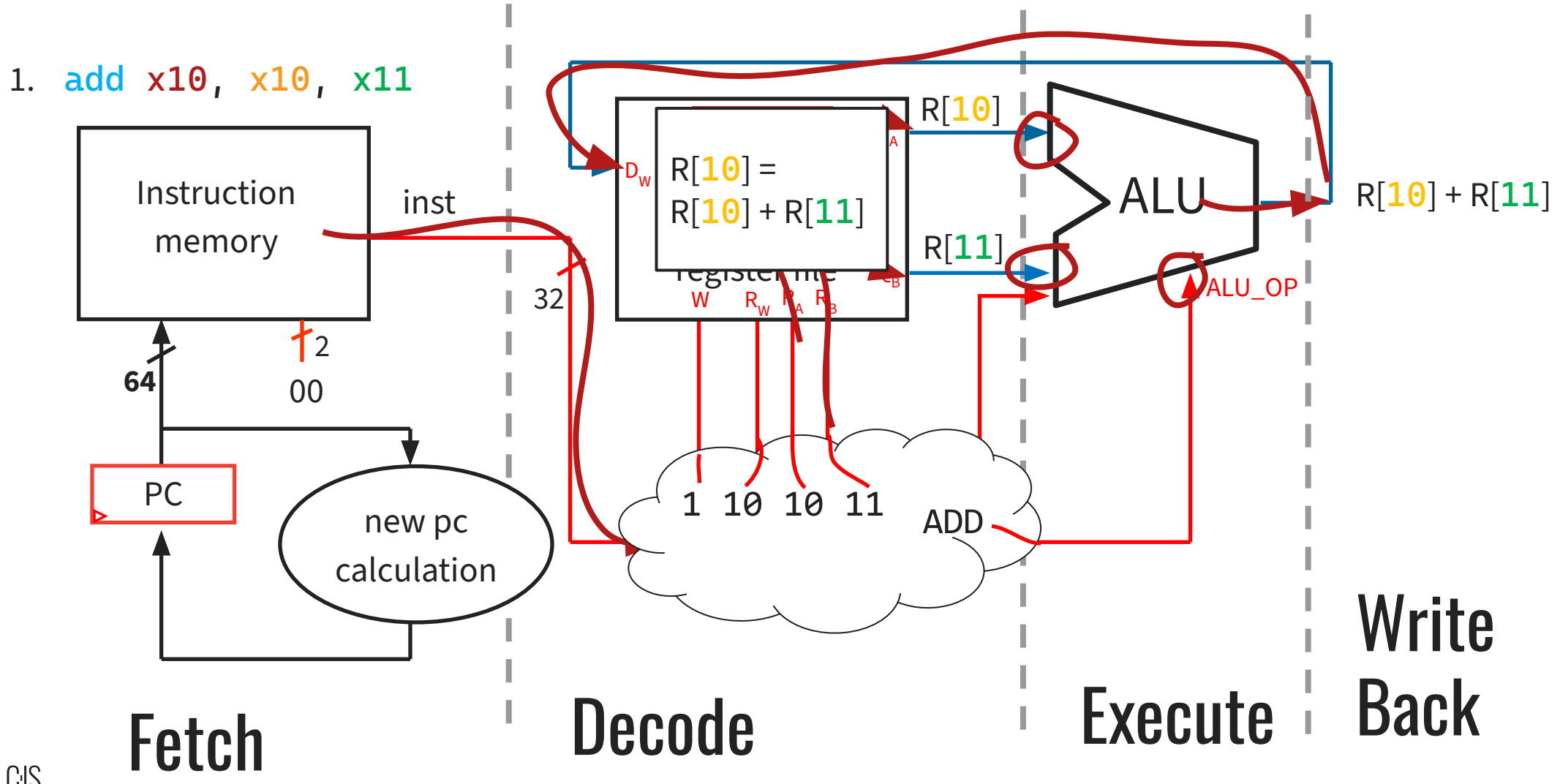
Executing `add x10, x10, x11`

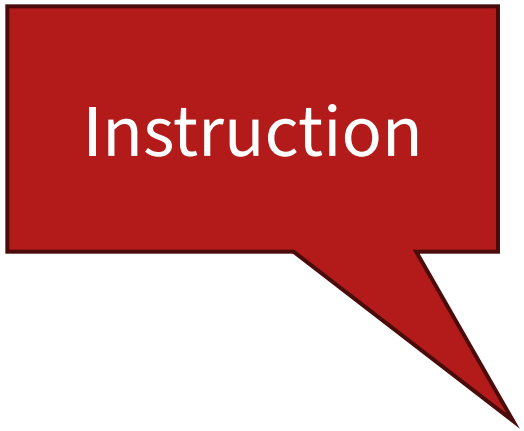


Executing `add x10, x10, x11`

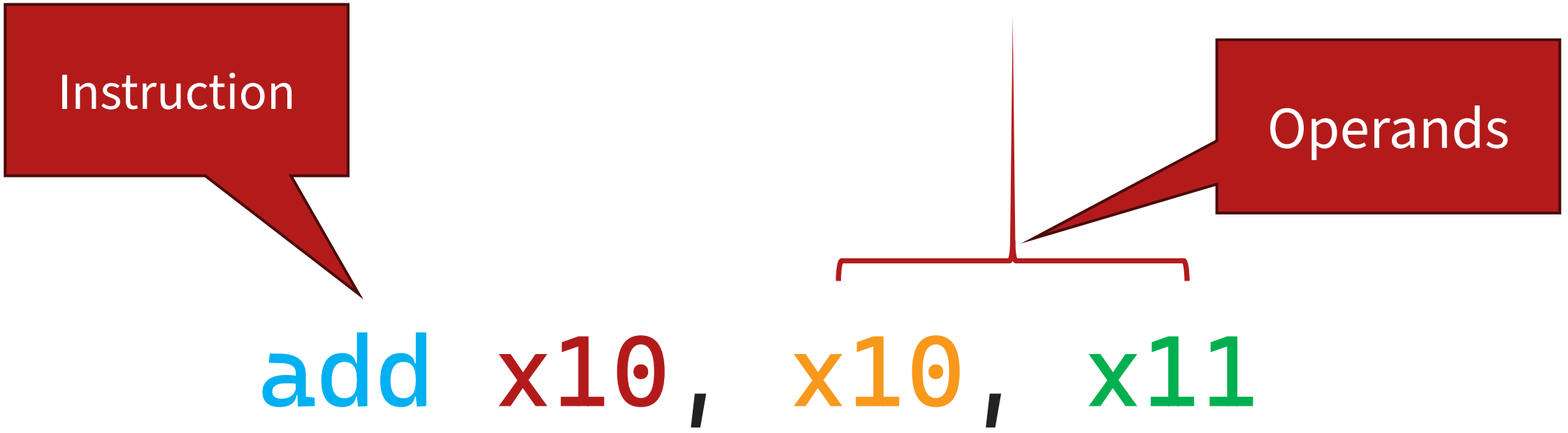


Executing `add x10, x10, x11`

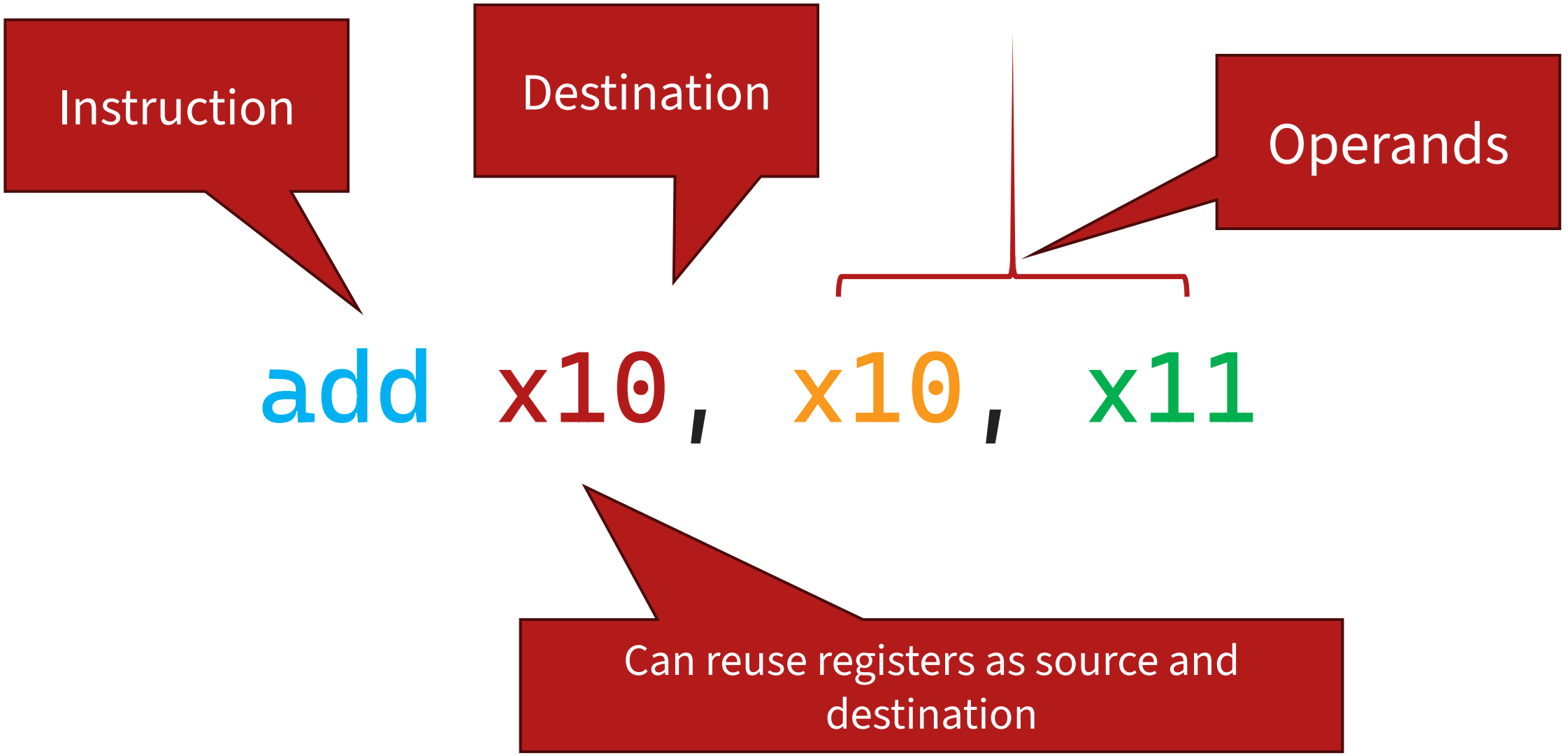





```
add x10, x10, x11
```


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!

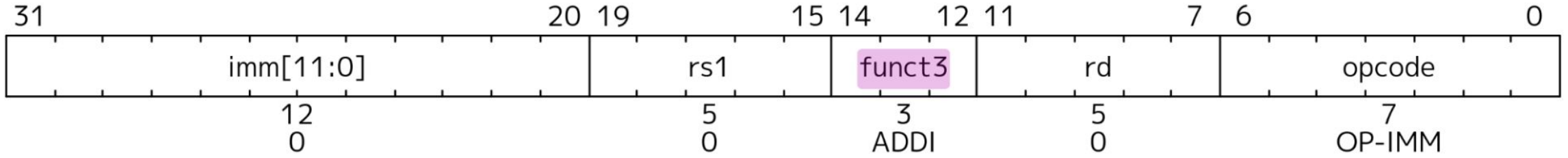
Register name	Symbolic name	Description
32 integer registers		
x0	zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporaries
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	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

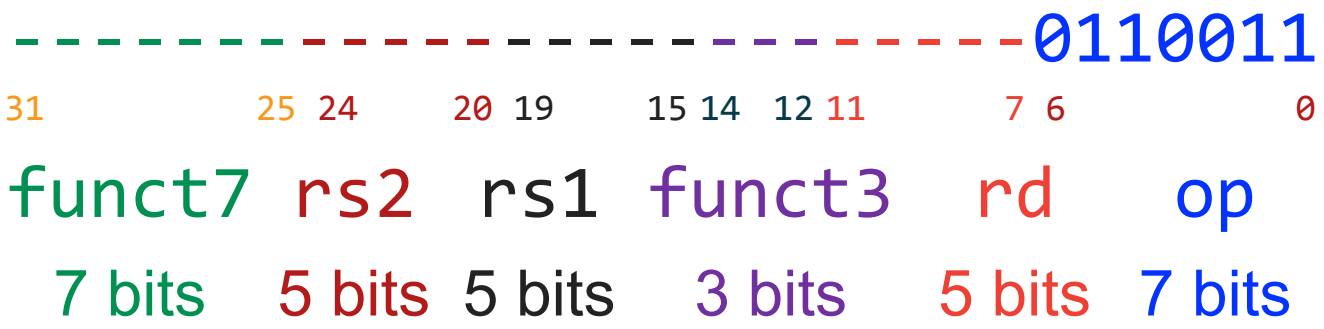
Memory Access

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

Control flow

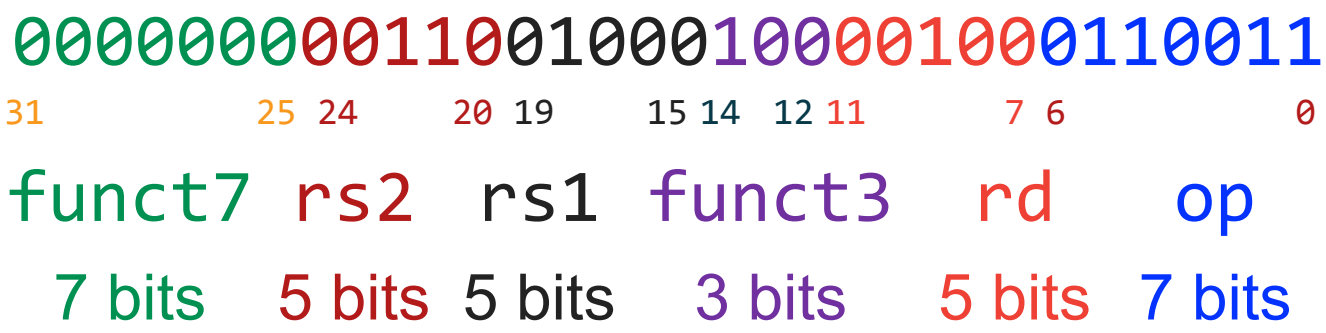
- **S-type:** conditional branches: pc-relative addresses
- **U-type:** jump-and-link
- **I-type:** jump-and-link register

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] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

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] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

R-Type (1): Arithmetic and Logic

```
0000000 00110 01000 100 01000 110011
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] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

R-Type (1): Arithmetic and Logic

```
0000000 00110 01000 100 01000 110011
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] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

#XOR

R-Type (1): Arithmetic and Logic

```
0000000 00110 01000 100 01000 0110011
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] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

```
#XOR  x4
      rd
```

R-Type (1): Arithmetic and Logic

```
0000000 00110 01000 100 01000 0110011
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] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

```
#XOR  x4,   x8
      rd,  rs1
```

R-Type (1): Arithmetic and Logic

```
0000000 00110 01000 100 01000 0110011
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] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ R[rs2]

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

R-Type (1): Arithmetic and Logic

```
0000000 00110 01000 100 01000 110011
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] | R[rs2]
0000000	100	XOR rd, rs1, rs2	R[rd] = R[rs1] ⊕ 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 ≠ `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) = !(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

-(-1) = !(1111 1111) + 1
      = 0 + 1
      = 1
```

Aside: Sign-Extension

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

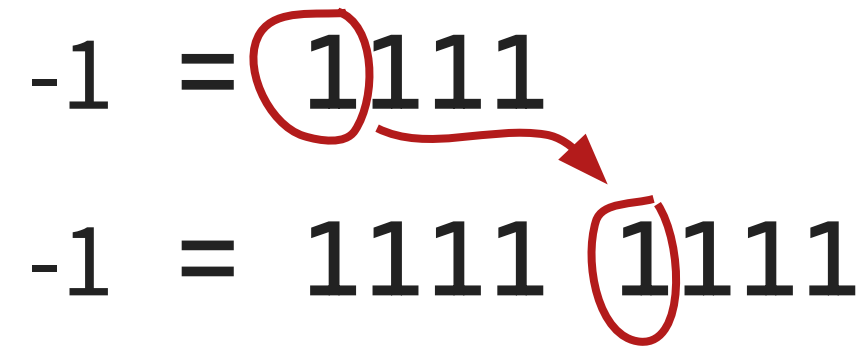
```
-1  = 1111

-1  = 1111 1111

-(-1) = !(1111 1111) + 1
      = 0 + 1
      = 1
```

Aside: Sign-Extension

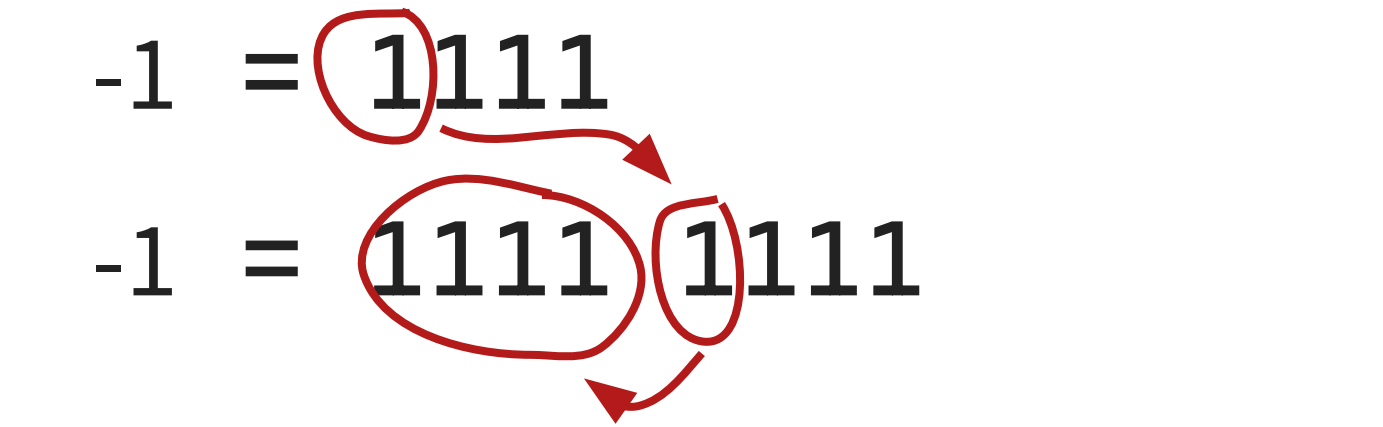
- Suppose we want to convert a 4-bit *negative* number into an 8-bit number




```
-(-1) = !(1111 1111) + 1
      = 0 + 1
      = 1
```

Aside: Sign-Extension

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




```
-(-1) = !(1111 1111) + 1
      = 0 + 1
      = 1
```

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

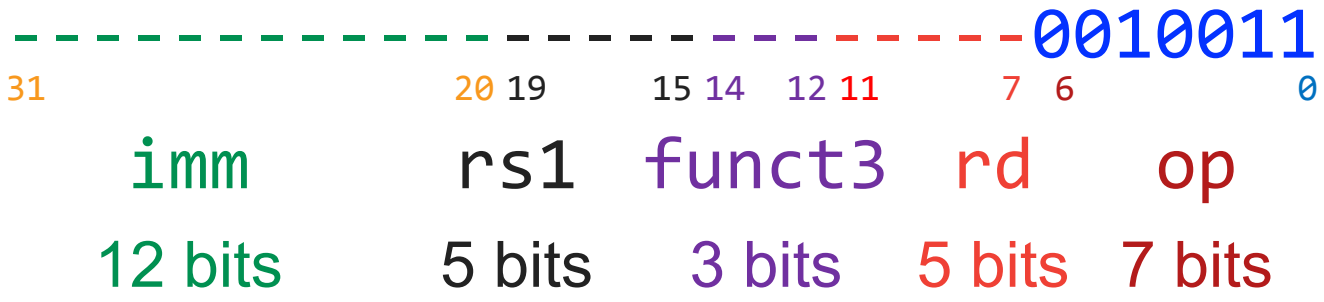
Aside: Truncation & Extension

- **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

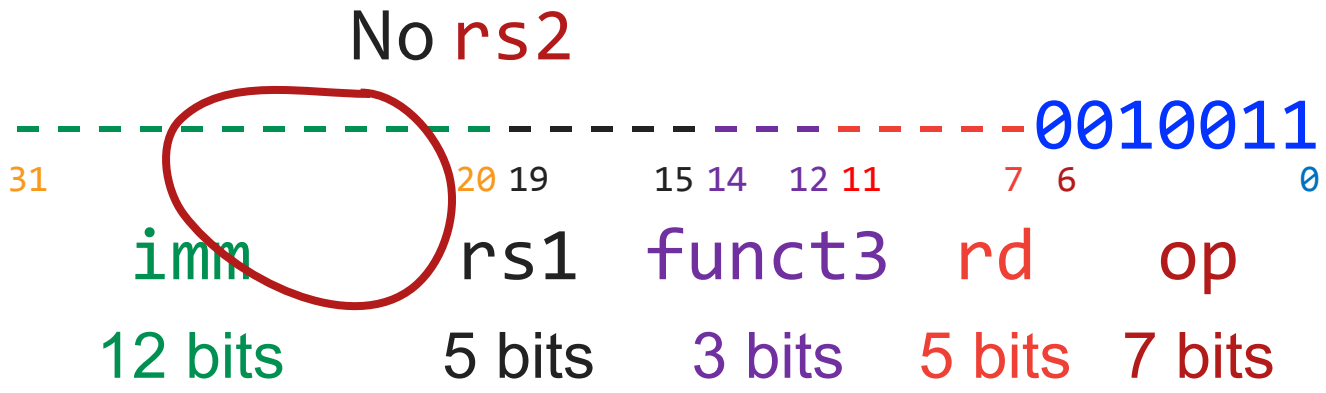
- **Truncation** *decreases* the size of a value
- **Extension** *increases* the size of a value
  - **Zero-extension** fills upper bits with 0
    - Used to extend *unsigned* numbers
  - **Sign-extension** fills upper bits with *copies of the most-significant bit*
    - Used to extend *signed* numbers

I-Type (1): Arithmetic w/ immediates



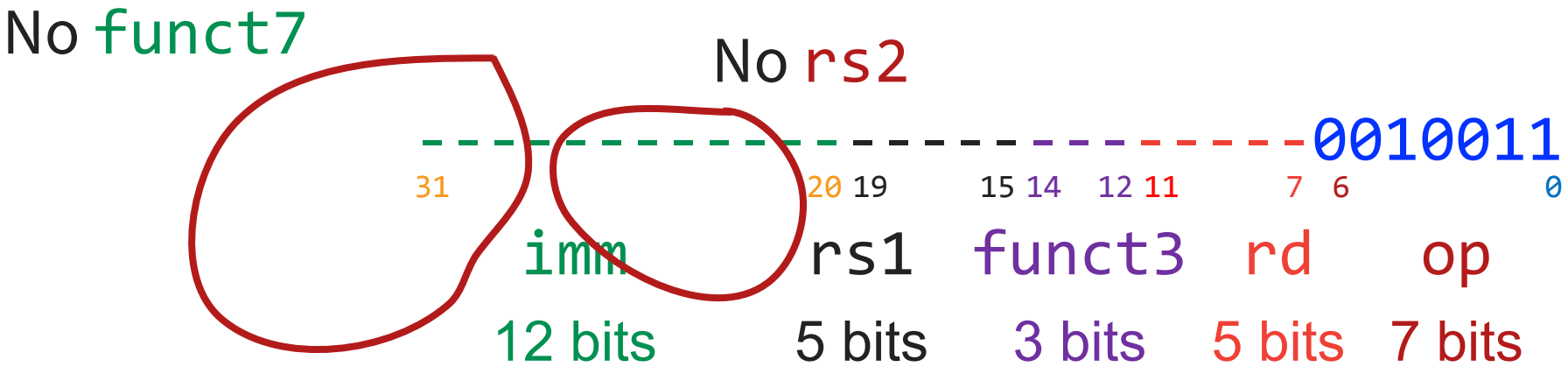
funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + sign_extend(imm)
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

I-Type (1): Arithmetic w/ immediates



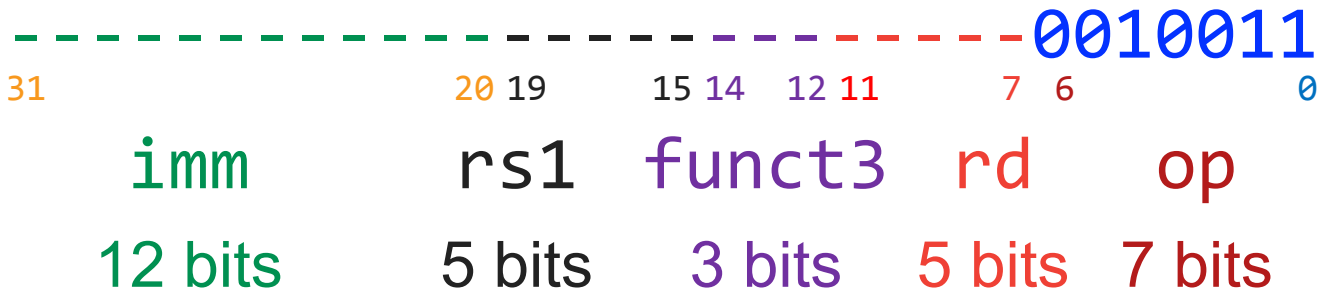
funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + sign_extend(imm)
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

I-Type (1): Arithmetic w/ immediates



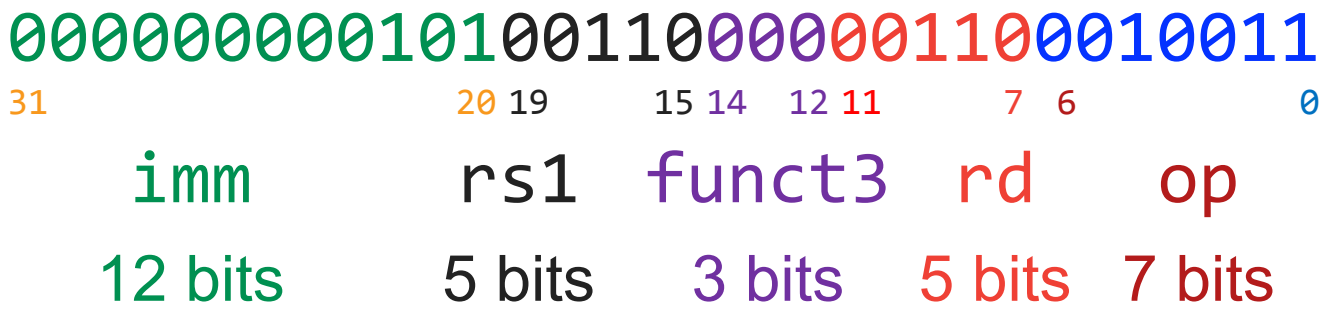
funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + sign_extend(imm)
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + sign_extend(imm)
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + imm
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

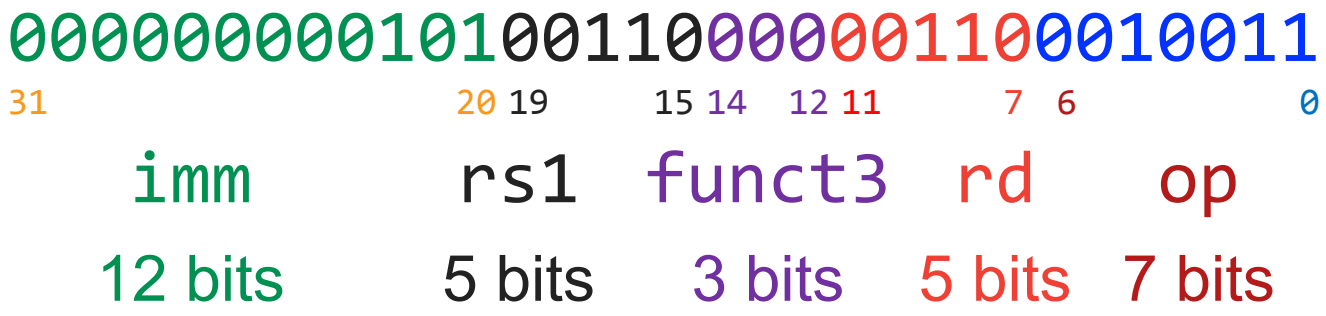
I-Type (1): Arithmetic w/ immediates

```
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] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

```
# ADDI
```

I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + imm
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

```
# ADDI  x6,
        rd,
```

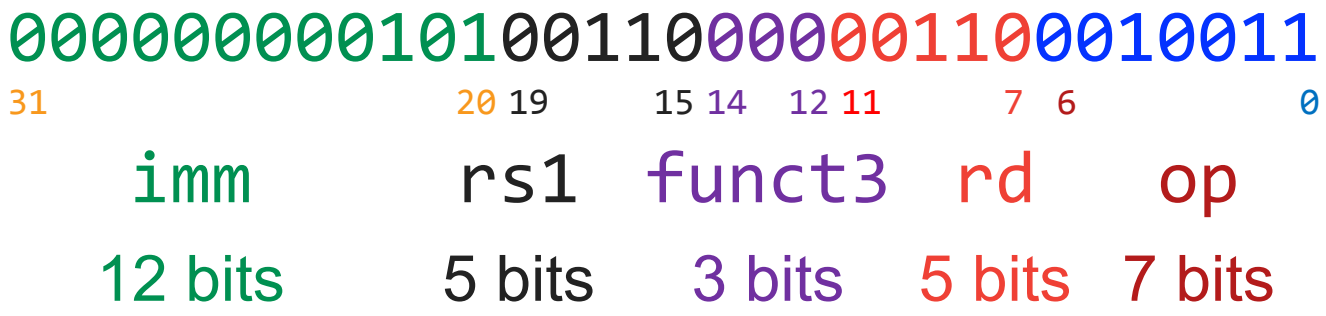
I-Type (1): Arithmetic w/ immediates

```
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] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

```
# ADDI   x6,   x6
         rd,  rs1
```

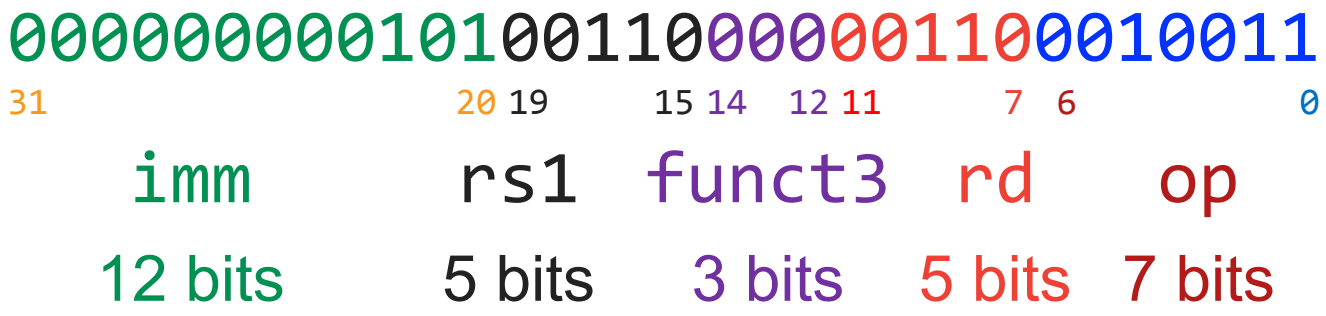
I-Type (1): Arithmetic w/ immediates



funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + imm
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

```
# ADDI   x6,   x6,    5
       rd, rs1, imm
```

I-Type (1): Arithmetic w/ immediates

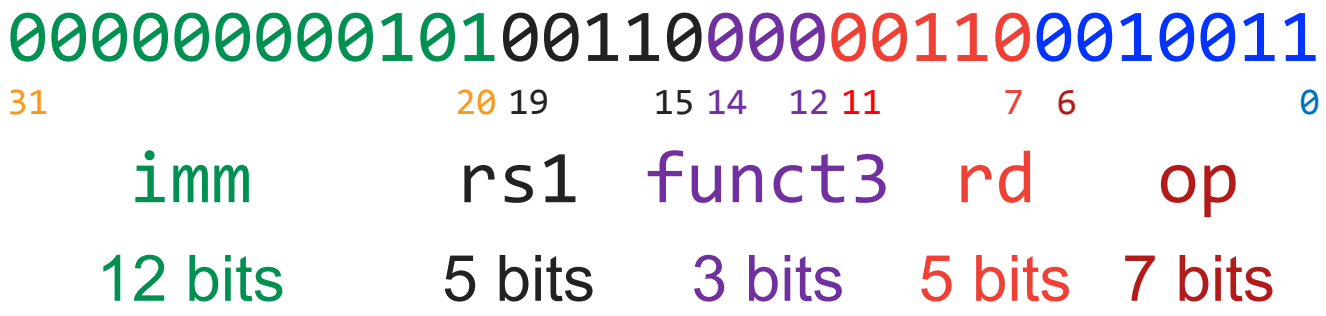


funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + imm
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

```
x6 = x6 + 5
```

```
# ADDI   x6,   x6,    5
         rd, rs1, imm
```

I-Type (1): Arithmetic w/ immediates

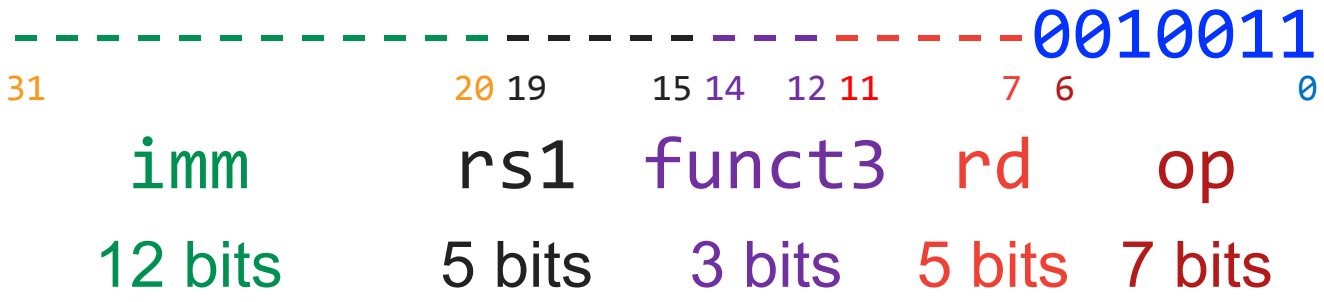


funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + imm
111	ANDI rd, rs1, imm	R[rd] = R[rs1] & sign_extend(imm)
110	ORI rd, rs1, imm	R[rd] = R[rs1] | sign_extend(imm)

```
x6 = x6 + 5
x6 += 5
```

```
# ADDI   x6,  x6,   5
         rd, rs1, imm
```

I-Type (1): Arithmetic w/ immediates



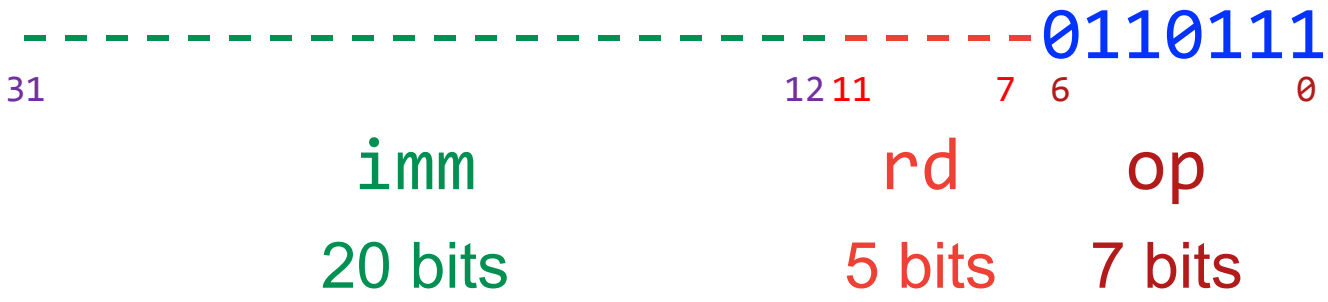
funct3	mnemonic	description
000	ADDI rd, rs1, imm	R[rd] = R[rs1] + sign_extend(imm)

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[rd] = imm << 12

U-Type (1): Load Upper Immediate

mnemonic	description
LUI rd, imm	R[rd] = imm << 12

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

U-Type (1): Load Upper Immediate

mnemonic	description
LUI rd, imm	R[rd] = imm << 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[rd] = imm << 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
```

Assembly Programming

Always zero!

Pseudocode

```
a0 = 34
a1 = a0 - 13
a2 = a1 * 2
```

Assembly

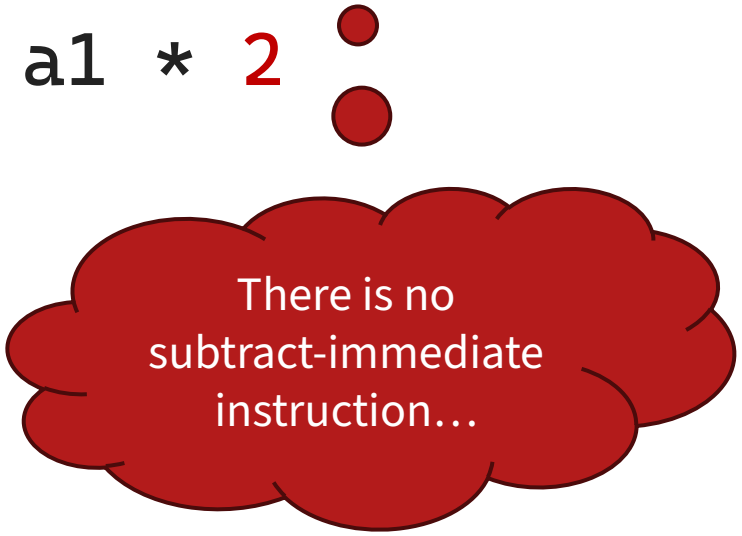
```
addi a0, x0, 34
```

How do we put 34 into register a0?

Assembly Programming

Pseudocode

```
a0 = 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
```

Assembly Programming

Pseudocode

```
a0 = 0 + 34
a1 = a0 - 13
a2 = a1 * 2
```

Assembly

```
addi a0, x0, 34
addi a1, a0, -13
```

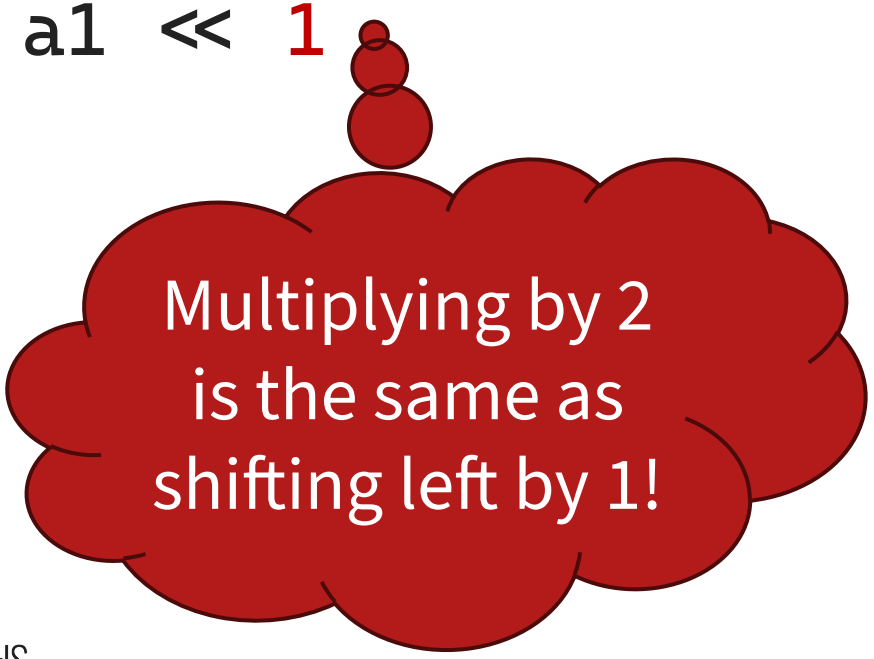
Assembly Programming

Pseudocode

```
a0 = 0 + 34
a1 = a0 - 13
a2 = a1 << 1
```

Assembly

```
addi a0, x0, 34
addi a1, a0, -13
slli a2, a1, 1
```



Next Monday

- RISC-V Control Flow
- RISC-V Memory