pollev.com/kakiryan

COMP311: *COMPUTER ORGANIZATION!*

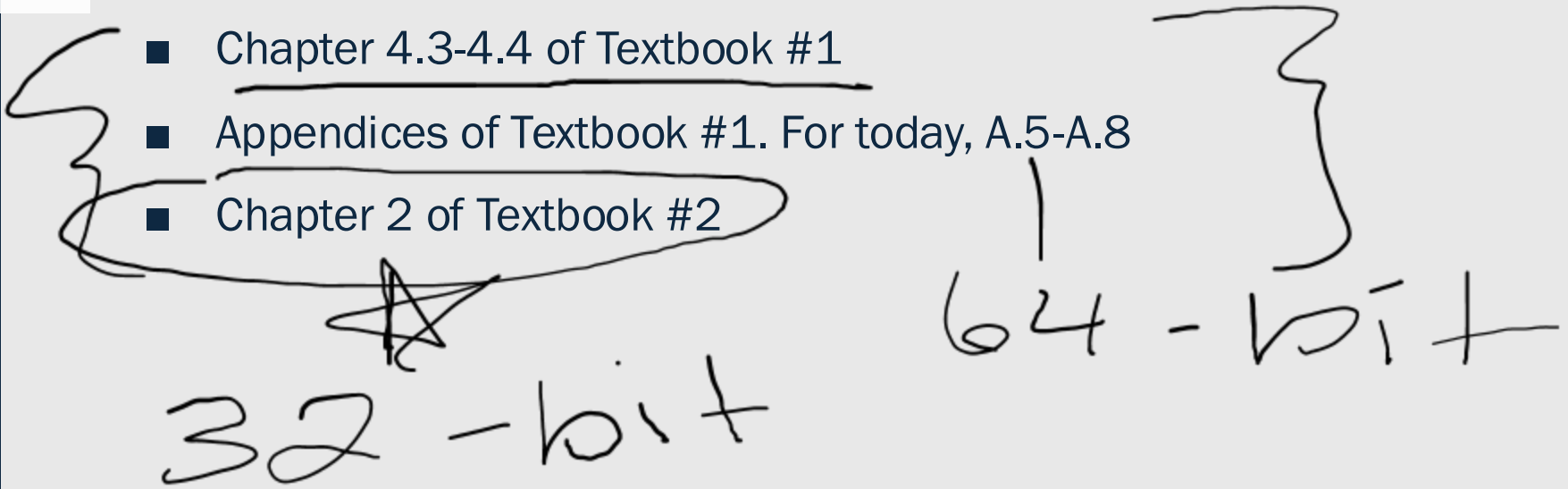
Lecture 19: RISC-V Assembly!

tinyurl.com/comp311-fa25

RISC-V ASSEMBLY! ☺ YAY YAY YAY FUN FUN

References/Additional Resources

- miniRISC-V Simulator
  - *Associated documentation is awesome! Thank you Leonard!*
- Chapter 4.3-4.4 of Textbook #1
- Appendices of Textbook #1. For today, A.5-A.8
- Chapter 2 of Textbook #2

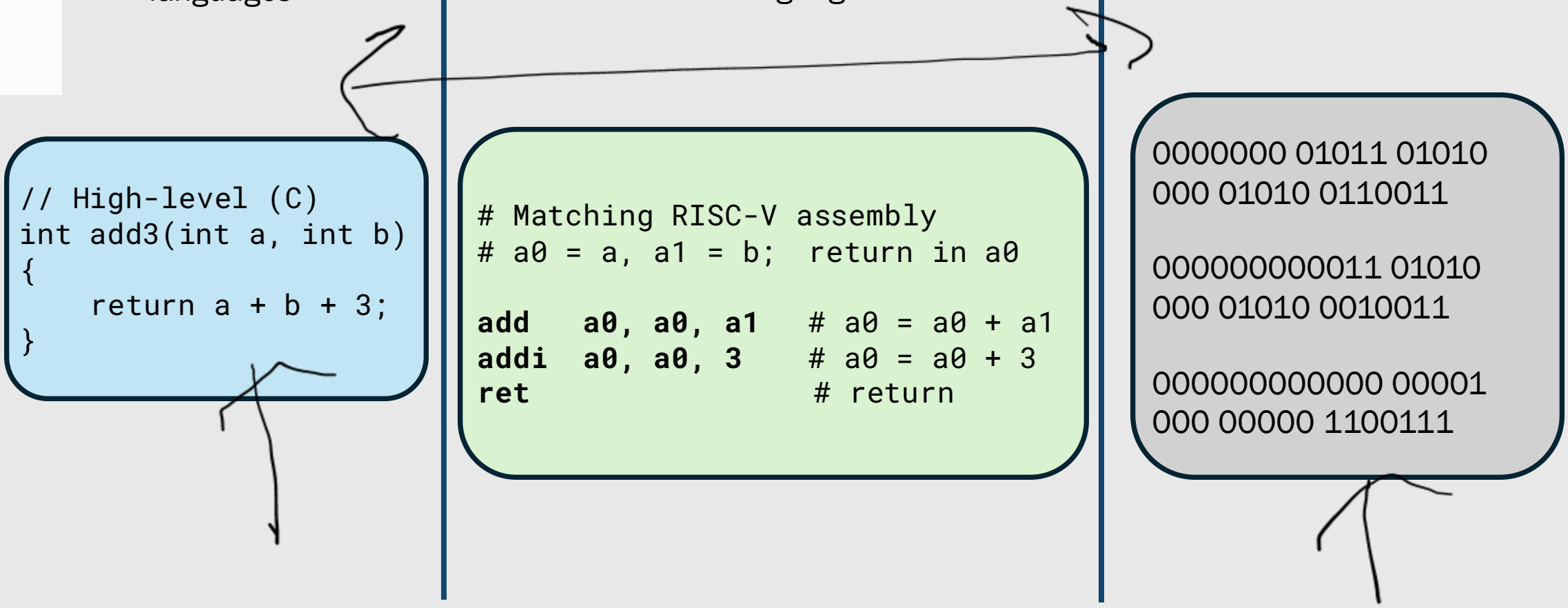


What is the course about?

High-level programming languages

Assembly
Low-level languages

Machine code



```
// High-level (C)
int add3(int a, int b)
{
    return a + b + 3;
}
```

```
# Matching RISC-V assembly
# a0 = a, a1 = b;  return in a0

add   a0, a0, a1   # a0 = a0 + a1
addi  a0, a0, 3    # a0 = a0 + 3
ret                # return
```

```
0000000 01011 01010
000 01010 0110011

000000000011 01010
000 01010 0010011

000000000000 00001
000 00000 1100111
```

Easy for humans to read/write

Human readable

Machine-readable

Square in different assembly languages

MIPS

x86

ARM

RISC-V

```
square(int):
    addi    sp, sp, -16      # make stack frame
    sw      ra, 12(sp)       # save return address
    sw      s0, 8(sp)        # save frame pointer
    addi    s0, sp, 16       # set up new frame pointer
    sw      a0, -4(s0)       # save argument x

    lw      t0, -4(s0)       # t0 = x
    mul     t0, t0, t0       # t0 = x * x
    mv      a0, t0           # move result to a0

    lw      ra, 12(sp)       # restore return address
    lw      s0, 8(sp)        # restore frame pointer
    addi    sp, sp, 16       # pop stack frame
    jr      ra               # return
```

```
square(int):
        push    {r7}
        sub     sp, sp, #12
        add     r7, sp, #0
        str     r0, [r7, #4]
        ldr     r3, [r7, #4]
        mul     r3, r3, r3
        mov     r0, r3
        adds    r7, r7, #12
        mov     sp, r7
        ldr     r7, [sp], #4
        bx      lr
```

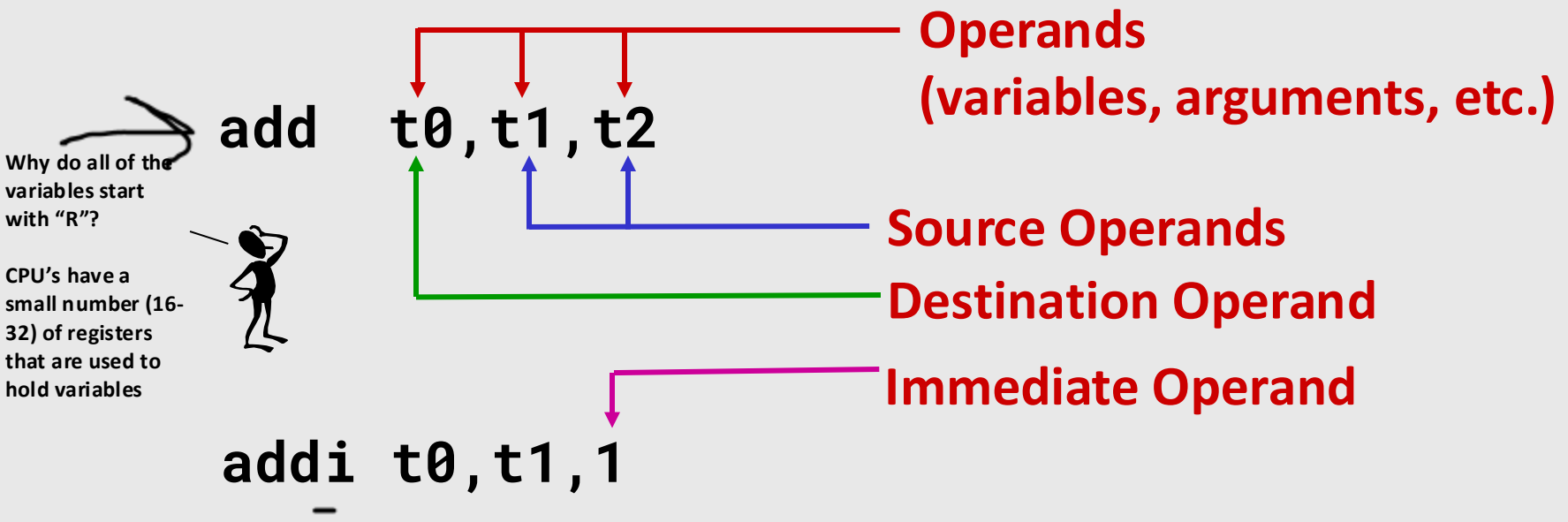
Instruction Set Architecture (ISA)

- A set of rules that define the interface between the hardware and software.

t0 ← t1 + t2

Anatomy of an Instruction

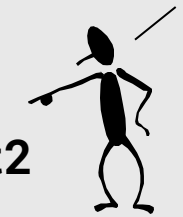
- Computers execute a set of primitive operations called instructions
- Instructions specify an operation and its operands (arguments of the operation)
- Types of operands: destination, source, and immediate



Meaning of an Instruction

- Operations are abbreviated into **opcodes** (1-4 letters)
- Instructions are specified with a very regular syntax
  - *Opcodes are followed by arguments*
  - *Usually the destination is next, then one or more source arguments (This is not strictly the case, but it is generally true)*
- Why this order?
  Analogy to high-level language like Java or C

```
add  t0,t1,t2
```



The instruction syntax provides operands in the same order as you would expect in a statement from a high level language.

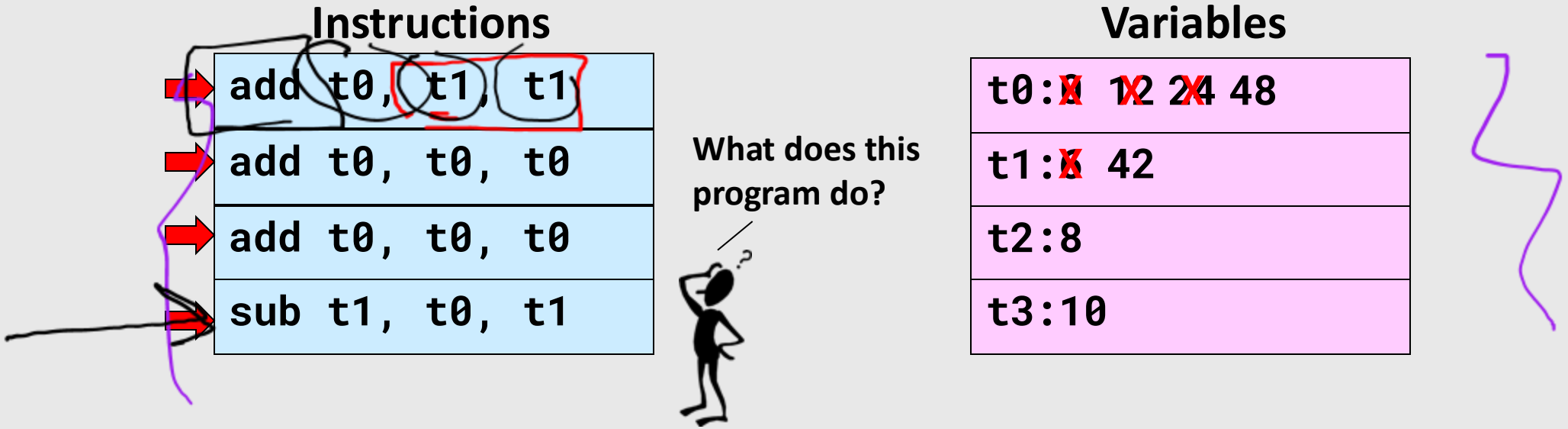
```
int r0, r1, r2;
r0 = r1 + r2;
```

Instead of:

```
r1 + r2 = r0;
```

A Series of Instructions

- Generally…
  - *Instructions are retrieved sequentially from memory*
  - *An instruction executes to completion before the next instruction is started*
  - *But, there are exceptions to these rules*



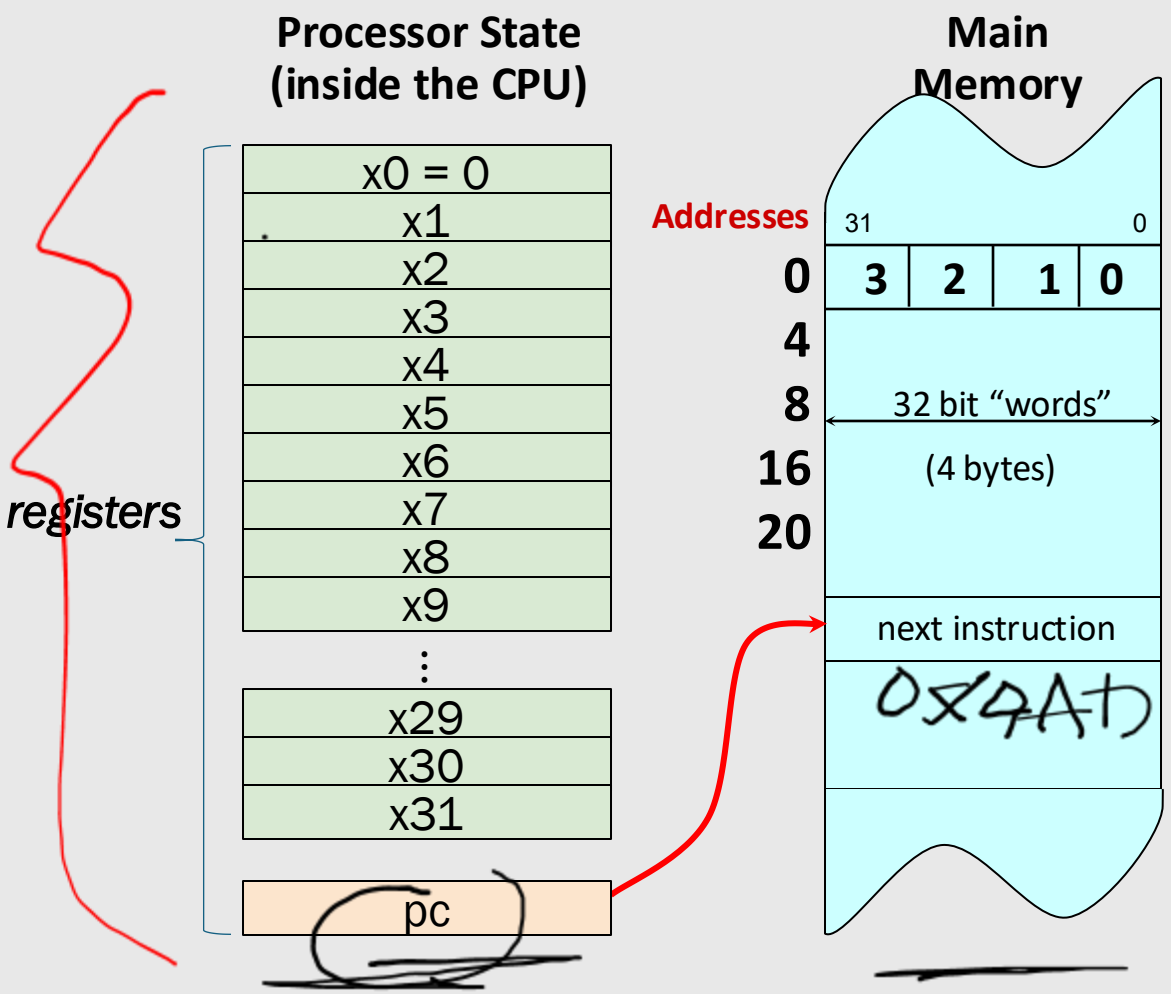
Instruction Set Architecture (ISA)

Encoding of instructions raises some interesting choices…

- Trade Offs: performance, compactness, programmability
- Uniformity. Should different instructions
  - *Be the same size (number of bits)?*
  - *Take the same amount of time to execute?*
  - *Trend: Uniformity. Affords simplicity, speed, pipelining.*
- Complexity. How many different instructions? What level operations?
  - *Level of support for particular software operations: array indexing, procedure calls, "polynomial evaluate", etc*
  - ***"Reduced Instruction Set Computer"*** *(RISC) philosophy: simple instructions, optimized for speed*
- Mix of Engineering & Art…

RISC-V Programming Model

A representative RISC machine



In Comp 311 we'll use a subset of the RISC-V core Instruction set as an example ISA (RV32I).

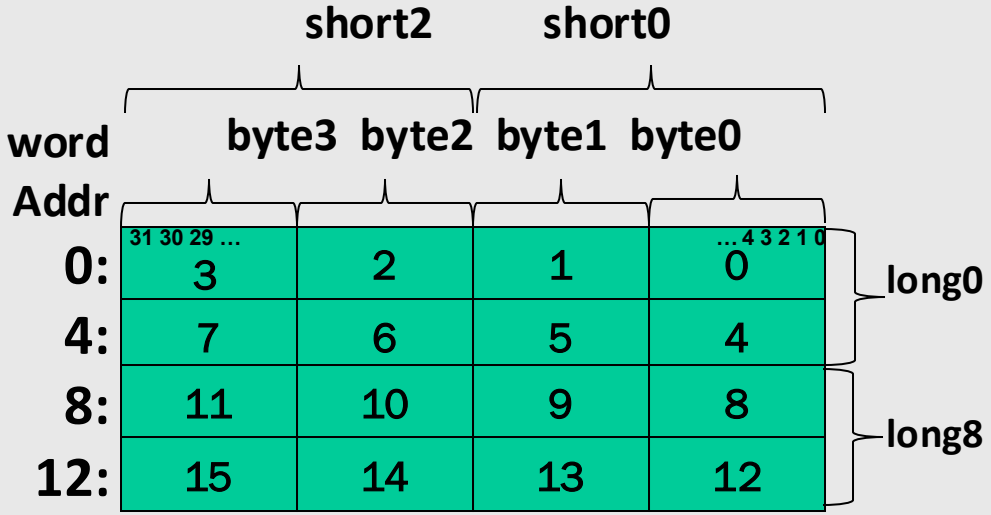
Fetch/Execute loop:

- **fetch Mem[PC]**
- **execute fetched instruction (may change PC!)**
- **PC = PC + 4**
- **repeat!**

RISC-V uses BYTE memory addresses. However, each instruction is 32-bits wide.. Each word contains four 8-bit bytes. Addresses of consecutive instructions (words) differ by 4.

RISC-V Memory Details

- Memory locations are addressable in different sized chunks
  - *8-bit chunks (bytes)*
  - *16-bit chunks (shorts)*
  - *32-bit chunks (words)*
  - *64-bit chunks (longs/doubles)*
- We also frequently need access to individual bits! (Instructions help with this)
- Every BYTE has a unique address (RISC-V is a byte-addressable machine)
- Most instructions are one word



RISC-V Register Details

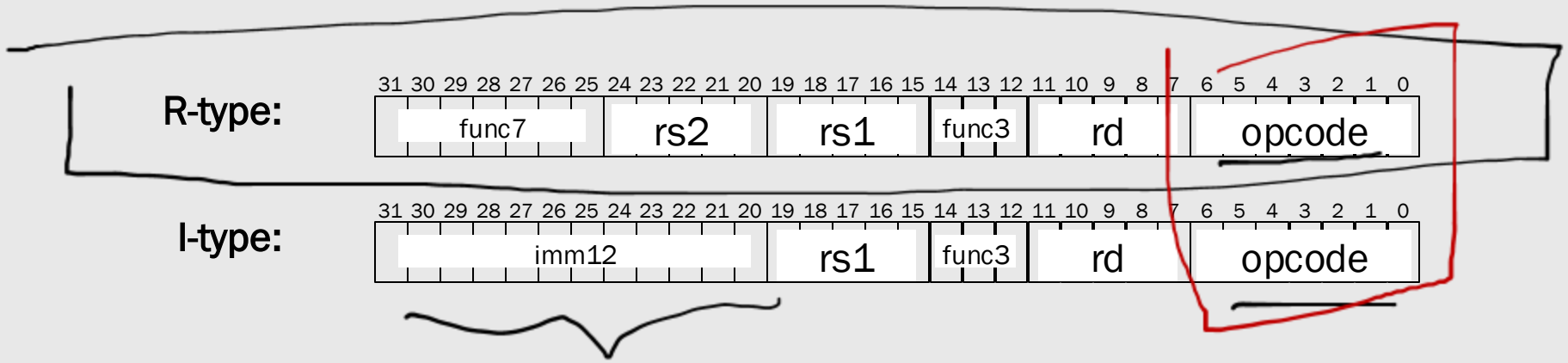
- There are 32 named registers [x0, x1, …. x31]
- x0 is special. It always contains "0" and, when used as a destination, the result is ignored
- The operands of most instructions are registers
- This means to operate on a variables in memory you must:
  - *Load the value/values from memory into a register*
  - *Perform the instruction*
  - *Store the result back into memory*
- Going to and from memory can be expensive (4x to 20x slower than operating on a register)
- Net effect: Keep variables in registers as much as possible!
- By convention, most registers are dedicated to specific tasks

A.K.A a "Load-Store Architecture"

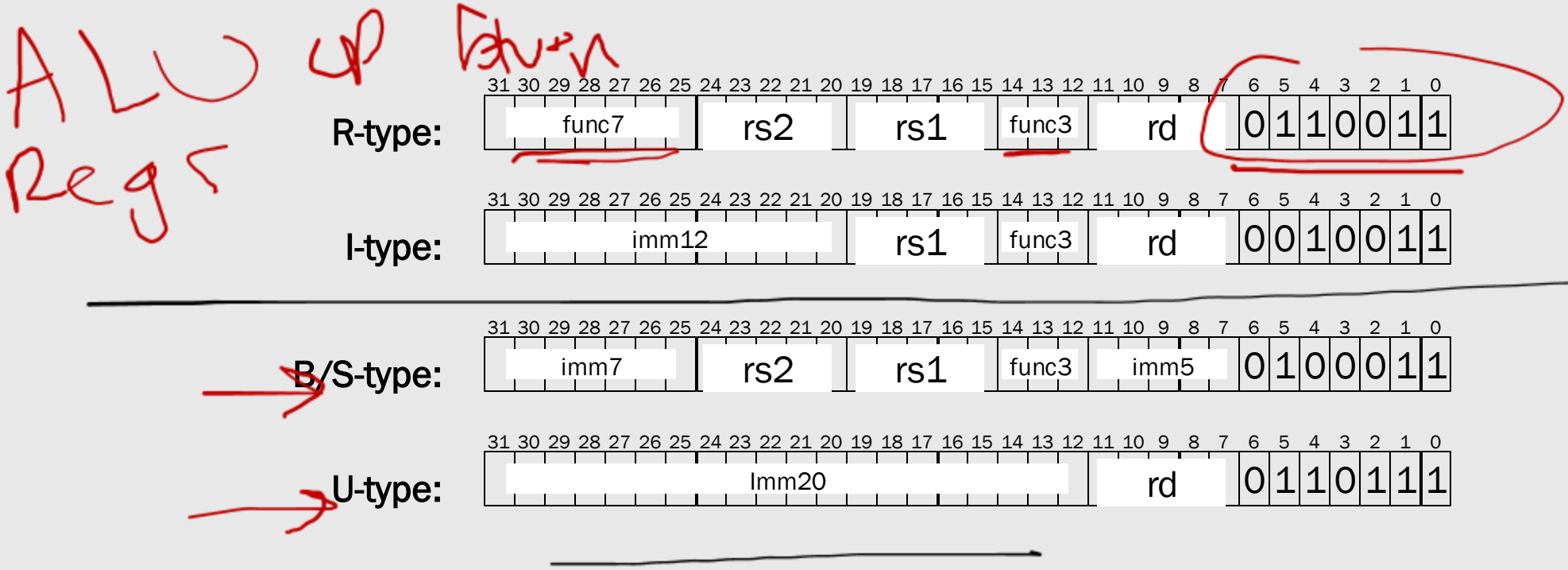
Basic RISC-V Instructions

- Instructions include various “fields” that encode combinations of OPCODES and arguments
- special fields enable extended functions
- several 5-bit OPERAND fields, for specifying the sources and destination of the operation, usually one of the 32 registers
- Embedded constants (“immediate” values) of various sizes,

The basic data-processing instruction formats:



The miniRISC-V ISA

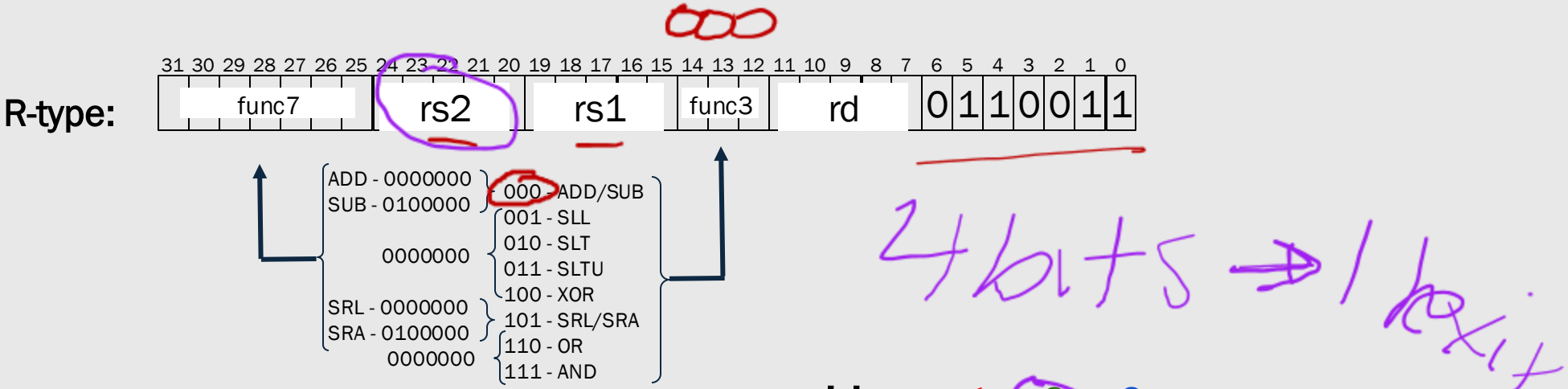


Four key instruction formats (each one has a corresponding opcode!):

1) ALU with two register operands
2) ALU with a register and an immediate operand
3) Stores and Branches
4) Jumps and large constants (LUI, AUIPC)

R-type Data Processing

Instructions that process three-register arguments:



add x1, x2, x3

Later we'll introduce more R-type variants

R-type instructions have the following template:

OPfunc3 rd, rs1, rs2

Is encoded as:

0000 0000 0011 0001 0000 0000 1011 0011

0x003100b3

Arithmetic Instructions

```
add x5,x6,x7
```

x5 ← x6 + x7
Registers can contain either 32-bit unsigned values or 32-bit 2's-complement signed values.

```
sub x6,x7,x28
```

x6 ← x7 - x28
Once more, either 32-bit unsigned values or 32-bit 2's-complement signed values.

```
mul x28,x5,x6
```

x28 ← x5 * x6
Register contents are treated as signed-values and multiplied together. The lower 32-bits of the result are saved in the destination.

```
div x7,x28,x6
```

x7 ← x28 / x6
The first source register is divided by the second. The result is saved in the destination. A divisor of 0 sets the destination to all '1's

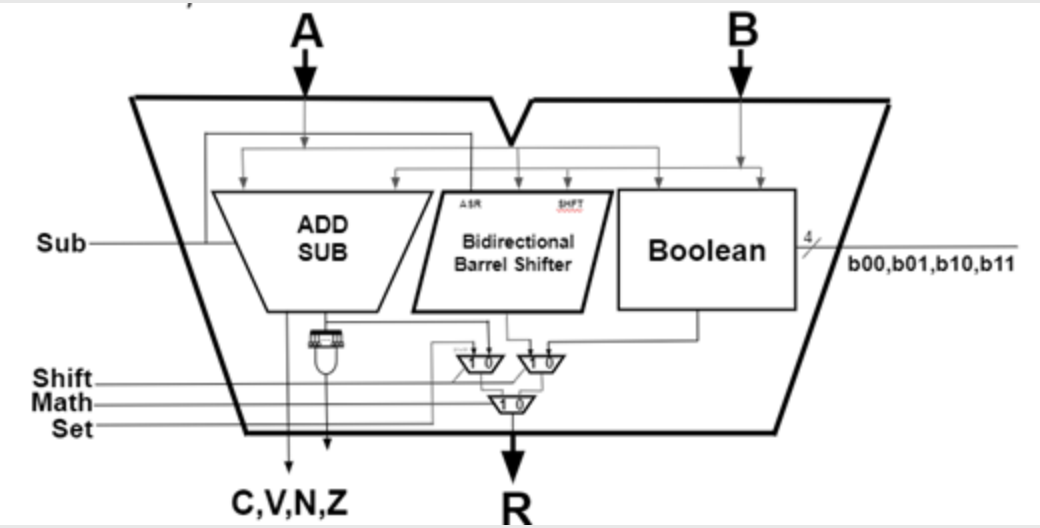
```
rem x6,x28,x6
```

x6 ← x28 % x6
The remainder left after dividing by the first operand by the second is stored in the destination. A divisor of 0 sets the result to the dividend.

Recall that the results of arithmetic operations can overflow, or in some cases aren't even possible, such as dividing by 0. These RISC-V instructions act exactly like the C-language operators. A user must write code that detects the overflow condition. Just as they need to do in C.

A modified RISC-V ALU

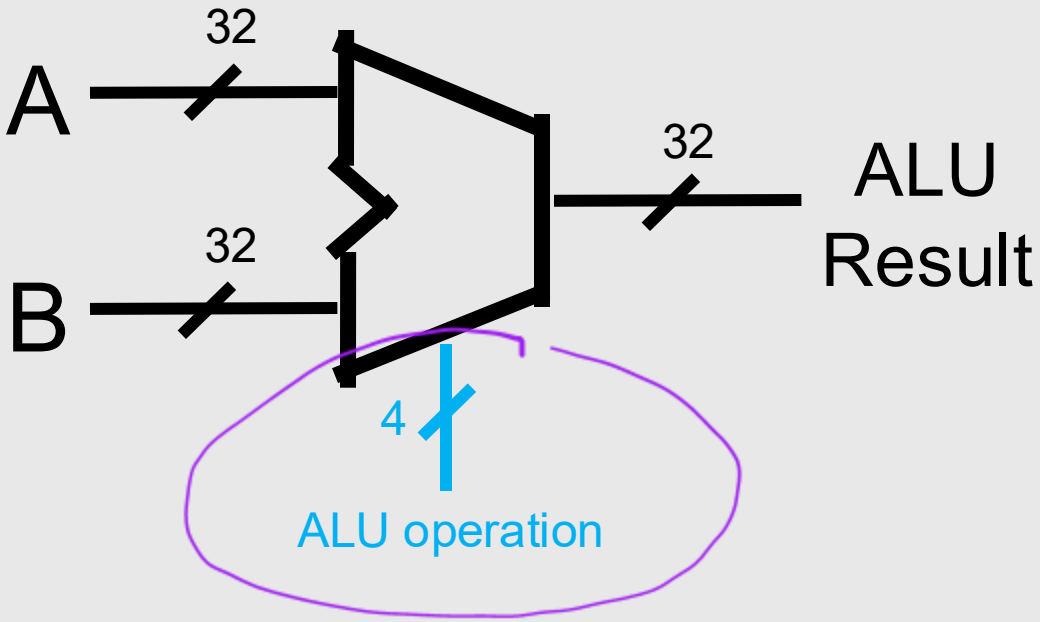
Special support to generate the constants "0" and "1" needed by the SLT, SLTU, SLTI, SLTUI, etc instructions



Math	Shift	Sub	Set	ALU Operation
0	0	X	X	f(A,B)
0	1	0	1	A << B
0	1	0	0	A >> B
0	1	1	0	A >>> B
1	0	0	X	A+B
1	0	1	X	A-B
1	1	X	0	0x0000
1	1	X	1	0x0001

See Appendix A.5 of the optional textbook #1!

ALU



Math	Shift	Sub	Set	ALU Operation
0	0	X	X	f(A,B)
0	1	0	1	A << B
0	1	0	0	A >> B
0	1	1	0	A >>> B
1	0	0	X	A+B
1	0	1	X	A-B
1	1	X	0	0x0000
1	1	X	1	0x0001

Control Signal Input

In RISC-V The ALUOp is generated by decoding funct7, funct3 and the opcode fields

Set on Less Than

- If A < B
  - *ALU result = 1*
- If A >= B
  - *ALU result = 0*

ALU

Operations to Perform

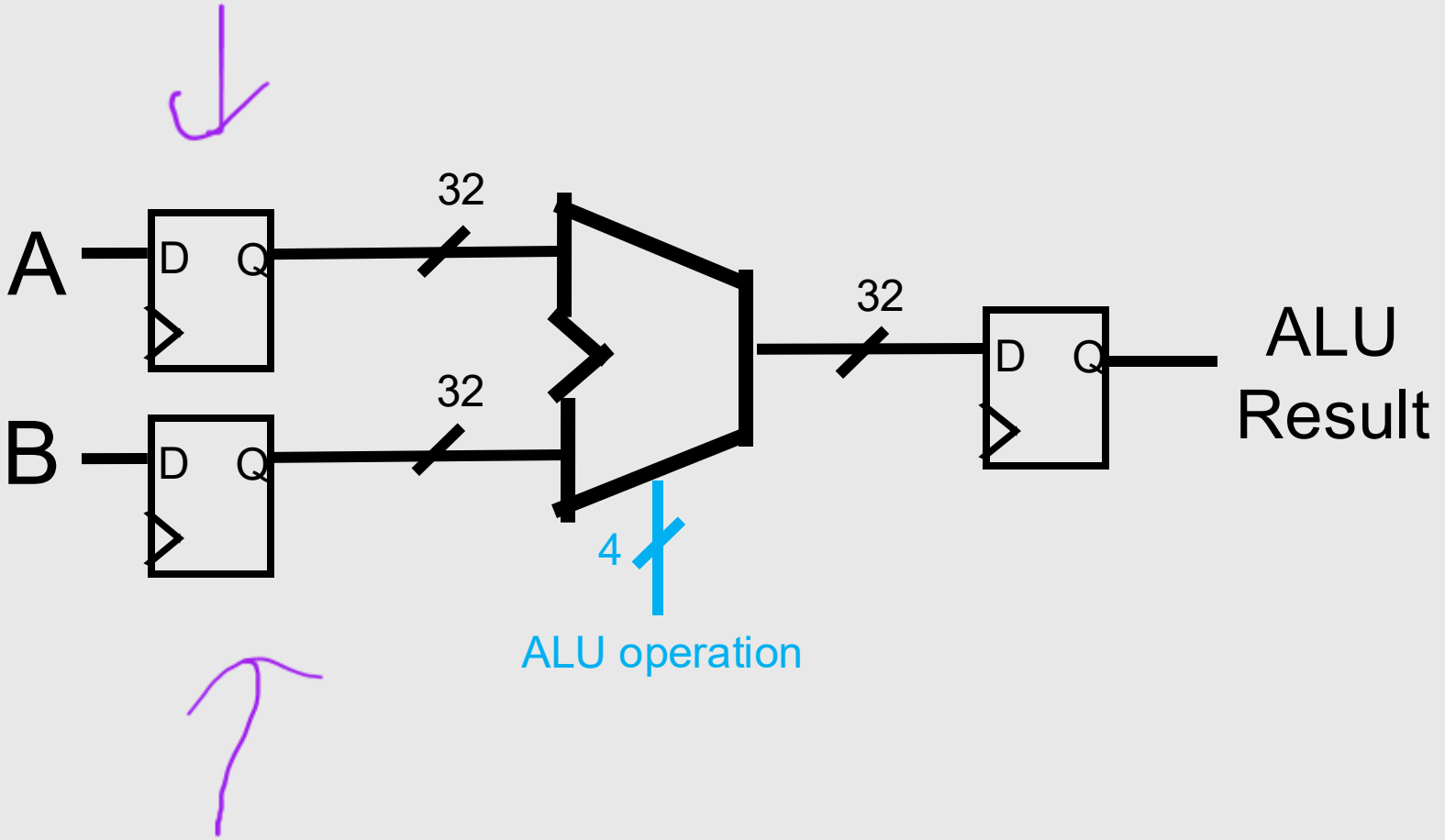
3 + 4

5 – 2

9 OR 4

6 AND 8

5 + 9



Assume all registers are connected to the same clock (not shown)

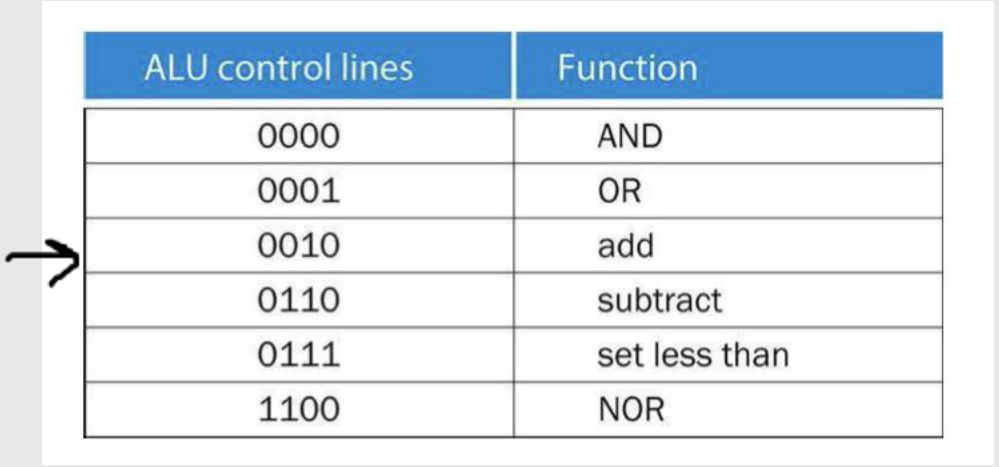
ALU

- The ALU will perform one operation per clock cycle
- To perform an operation, we have three inputs to set: A, B, and ALUOp
  - *(ALU Op (ALU Control line from the textbook) is also synchronized with a register, but that is abstracted away for now)*

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

	Operation	A	B	ALUOp
Cycle 0	3 + 4	3	4	0010
Cycle 1	5 - 2	5	2	0110
Cycle 2	9 OR 4	9	4	0001
Cycle 3	6 AND 8			
Cycle 4	5 + 9			

ALU

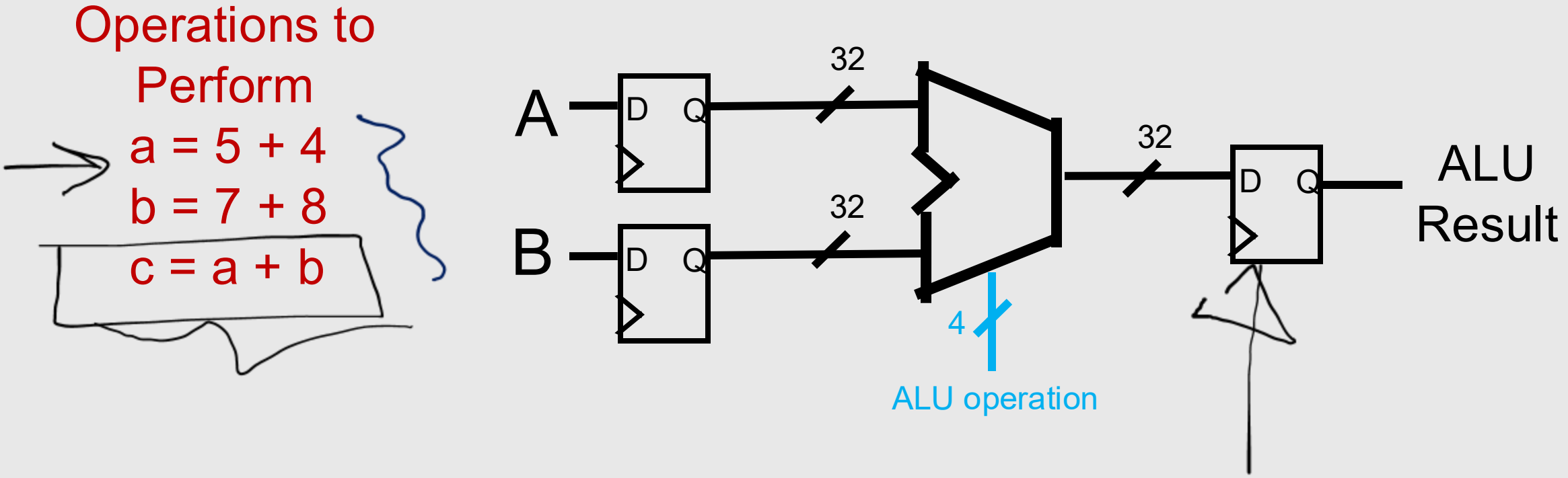


ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set less than
1100	NOR

- The ALU will perform one operation per clock cycle
- To perform an operation, we have three inputs to set: A, B, and ALUOp
  - *(ALU Op is also synchronized with a register, but that is abstracted away for now)*

	Operation	A	B	ALUOp
Cycle 0	3 + 4	3	4	0b0010
Cycle 1	5 - 2	5	2	0b0110
Cycle 2	9 OR 4	9	4	0b0001
Cycle 3	6 AND 8	6	8	0b0000
Cycle 4	5 + 9	5	9	0b0010

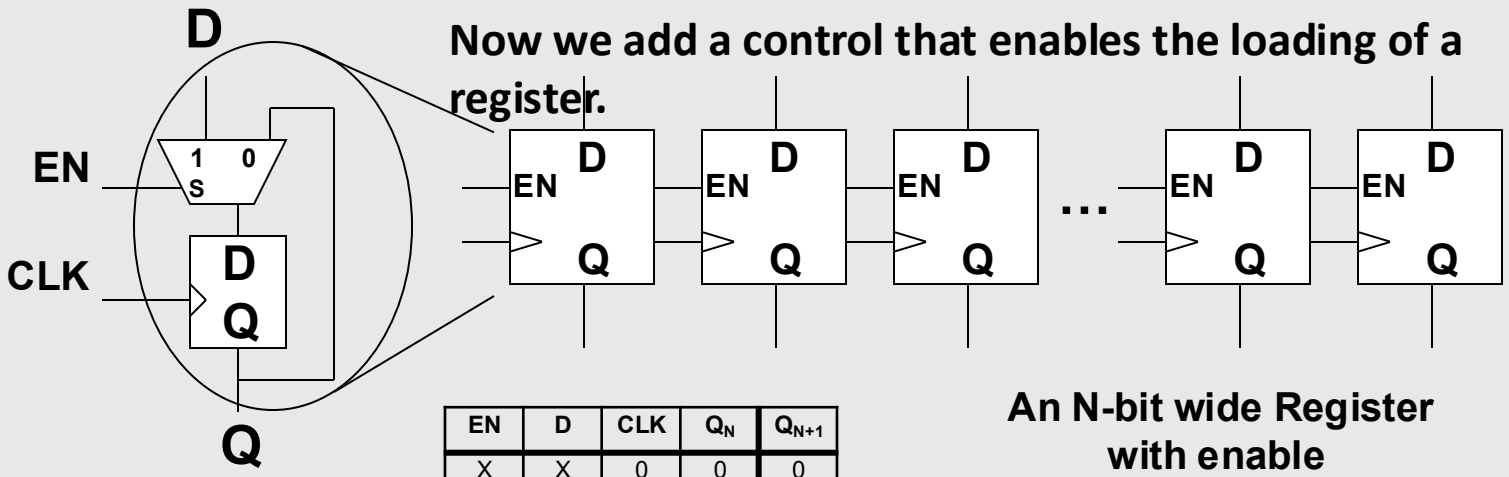
Dependent Operations



Assume all registers are connected to the same clock (not shown)

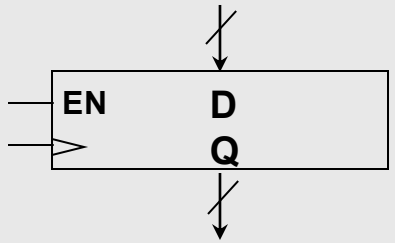
One More Functional Unit

Thus far, our building blocks units have focused on logical and arithmetic functions. We'll also need functional units for storing intermediate results. By now, we are used to the notion of building wide registers.



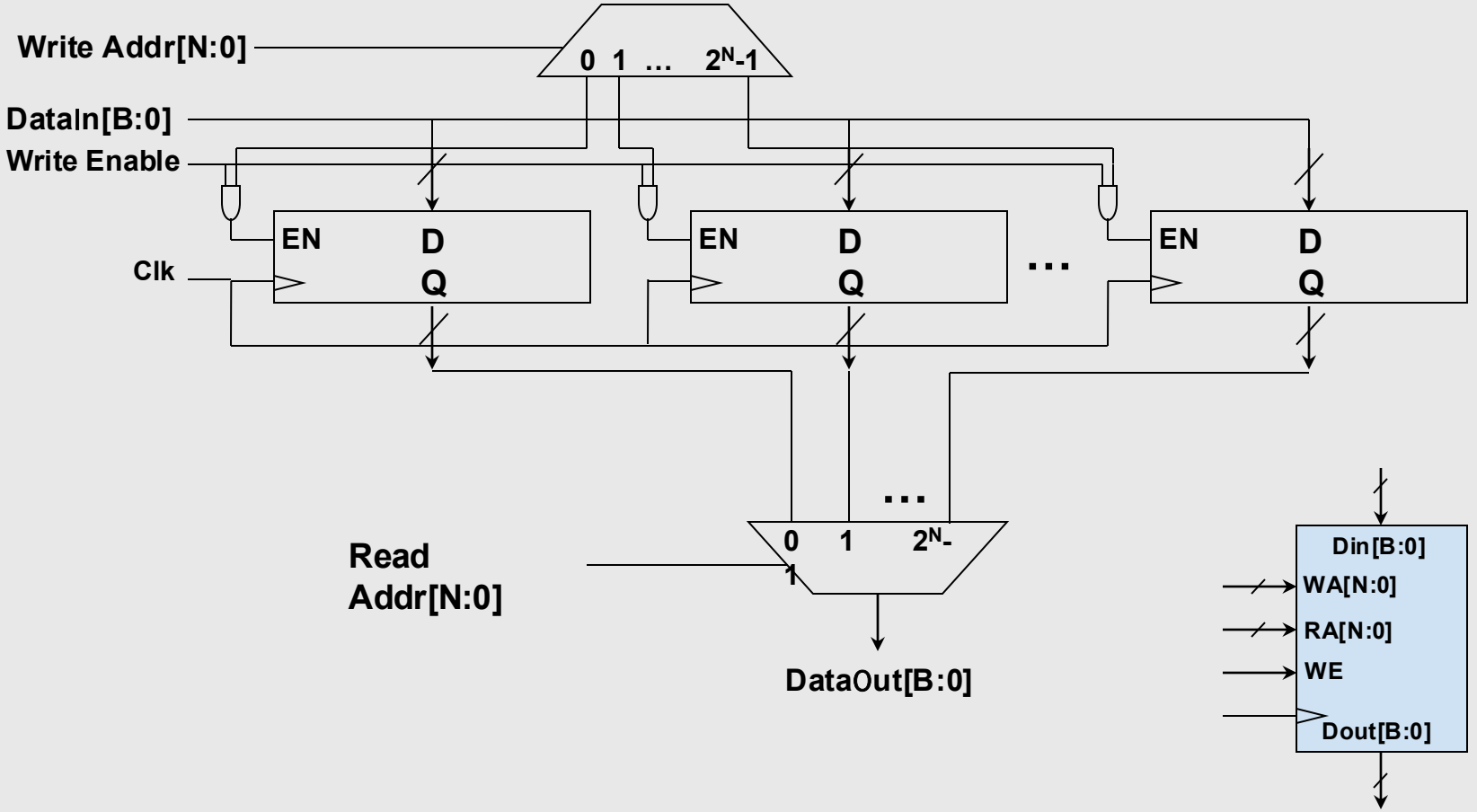
EN	D	CLK	Q_N	Q_{N+1}
X	X	0	0	0
X	X	0	1	1
X	X	1	0	0
X	X	1	1	1
0	X	↑	0	0
0	X	↑	1	1
1	0	↑	X	0
1	1	↑	X	1

An N-bit wide Register with enable



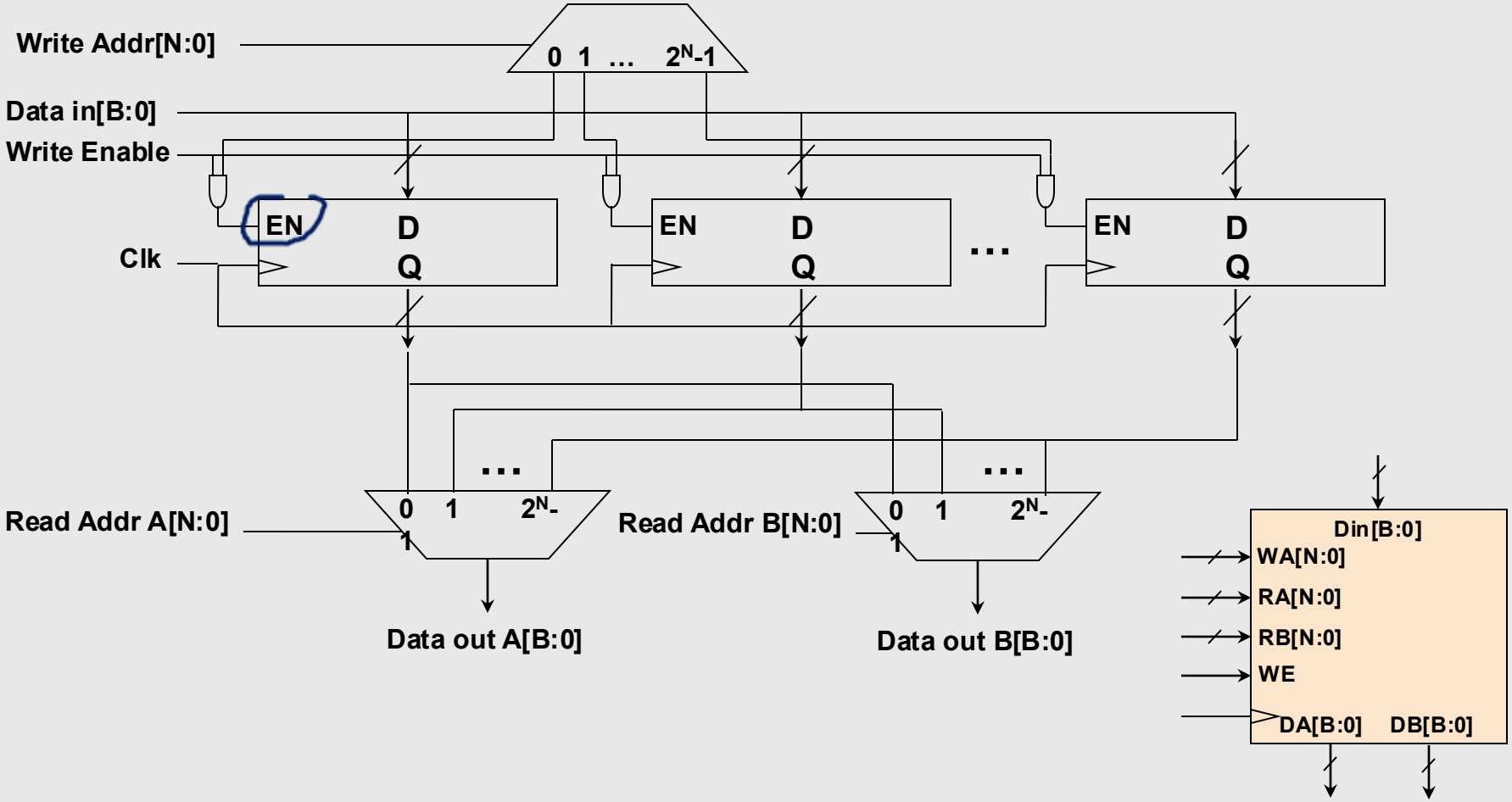
A Register File

We can also construct an addressable array of registers

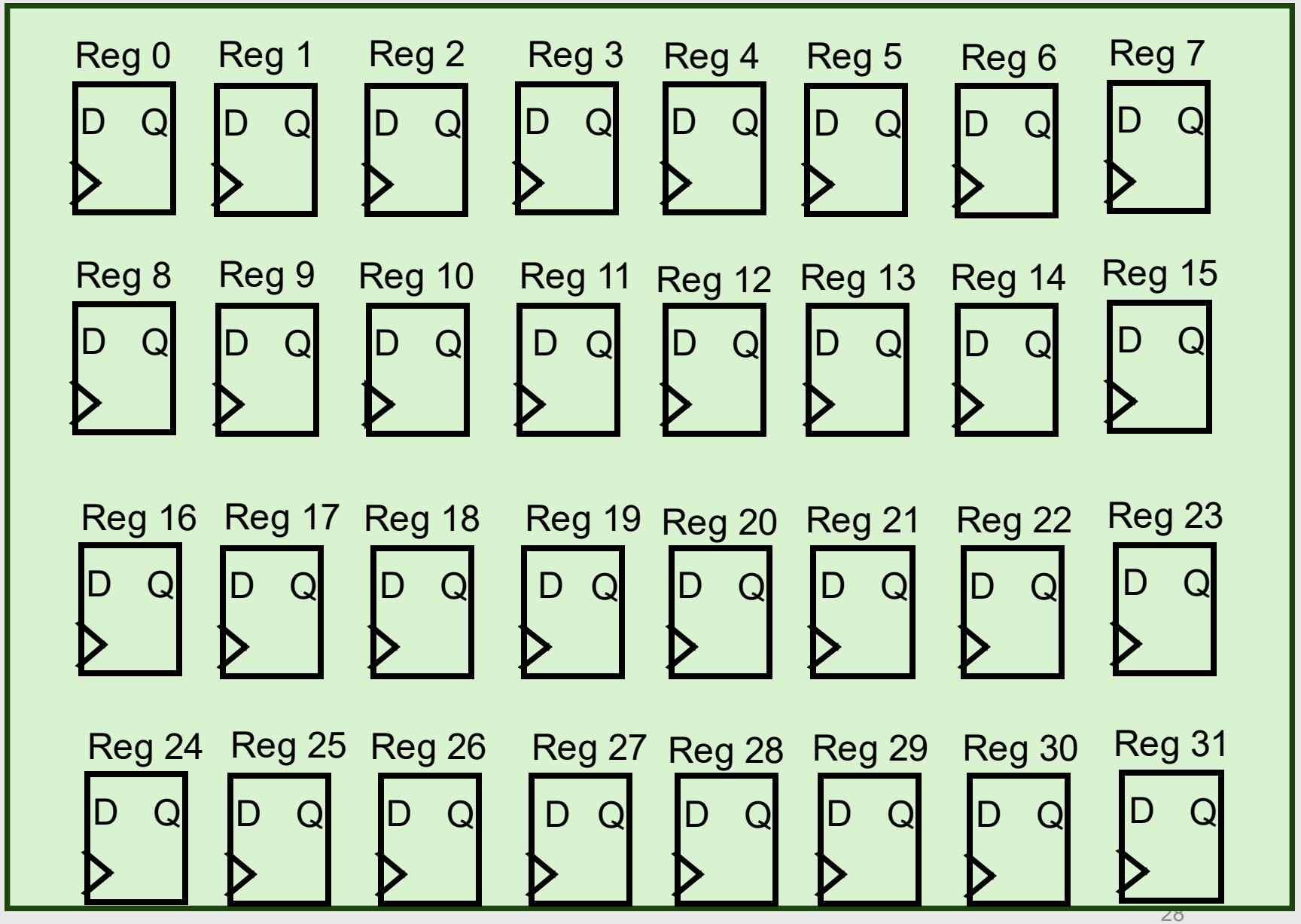


A Multi-Ported Register File

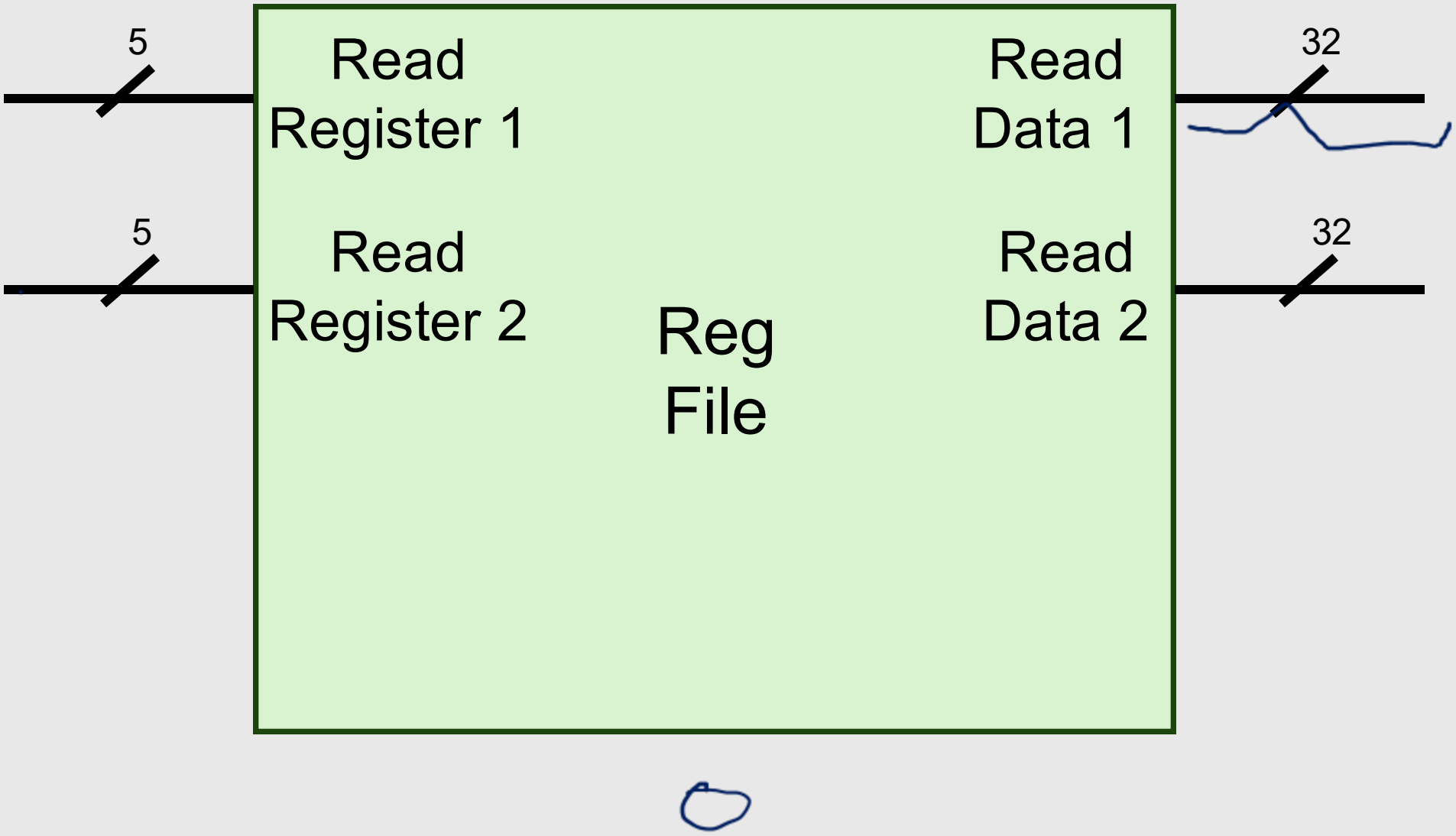
Multiple read ports by simply adding more output MUXs



Register File



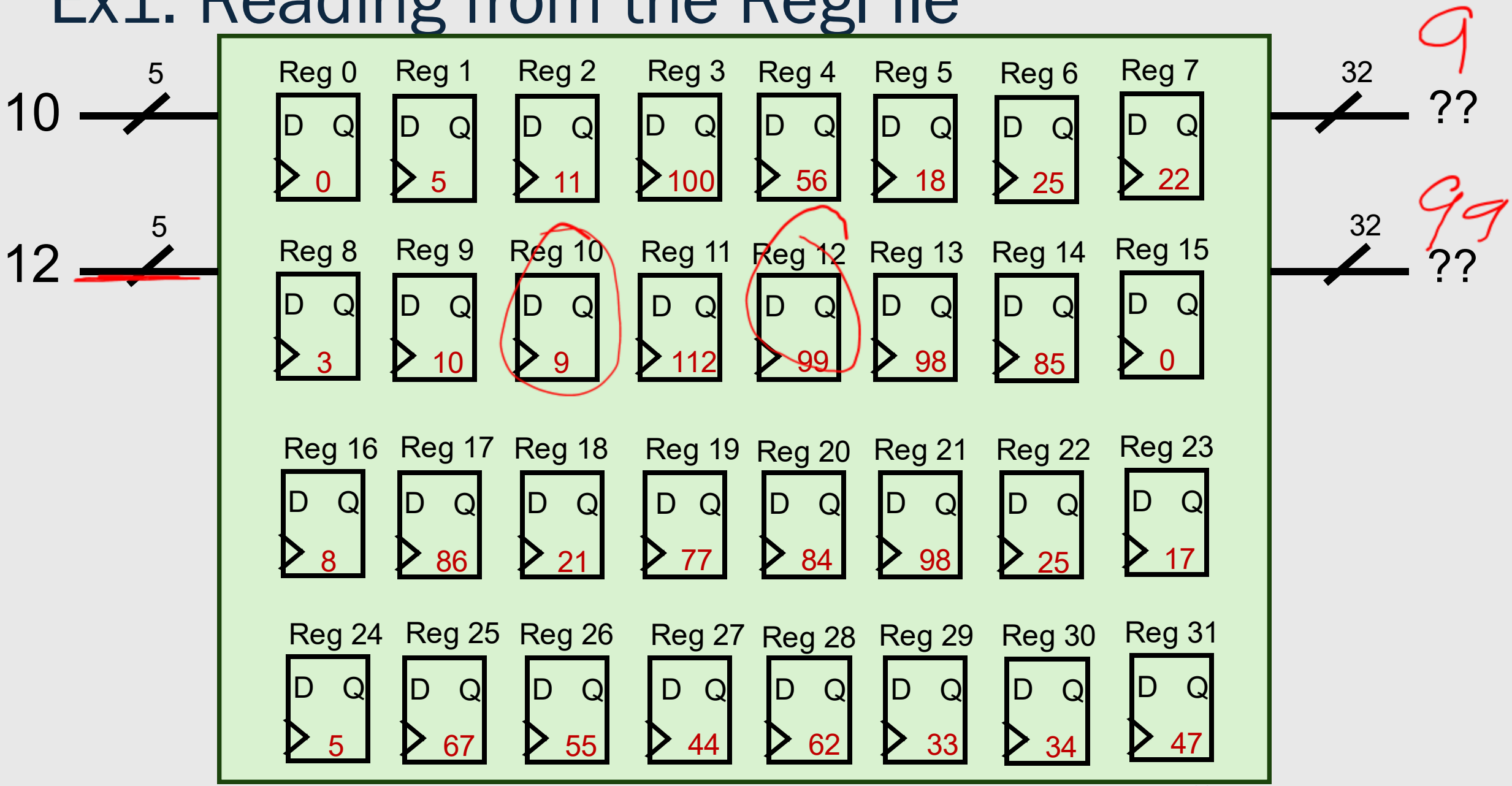
Reading from the Register File



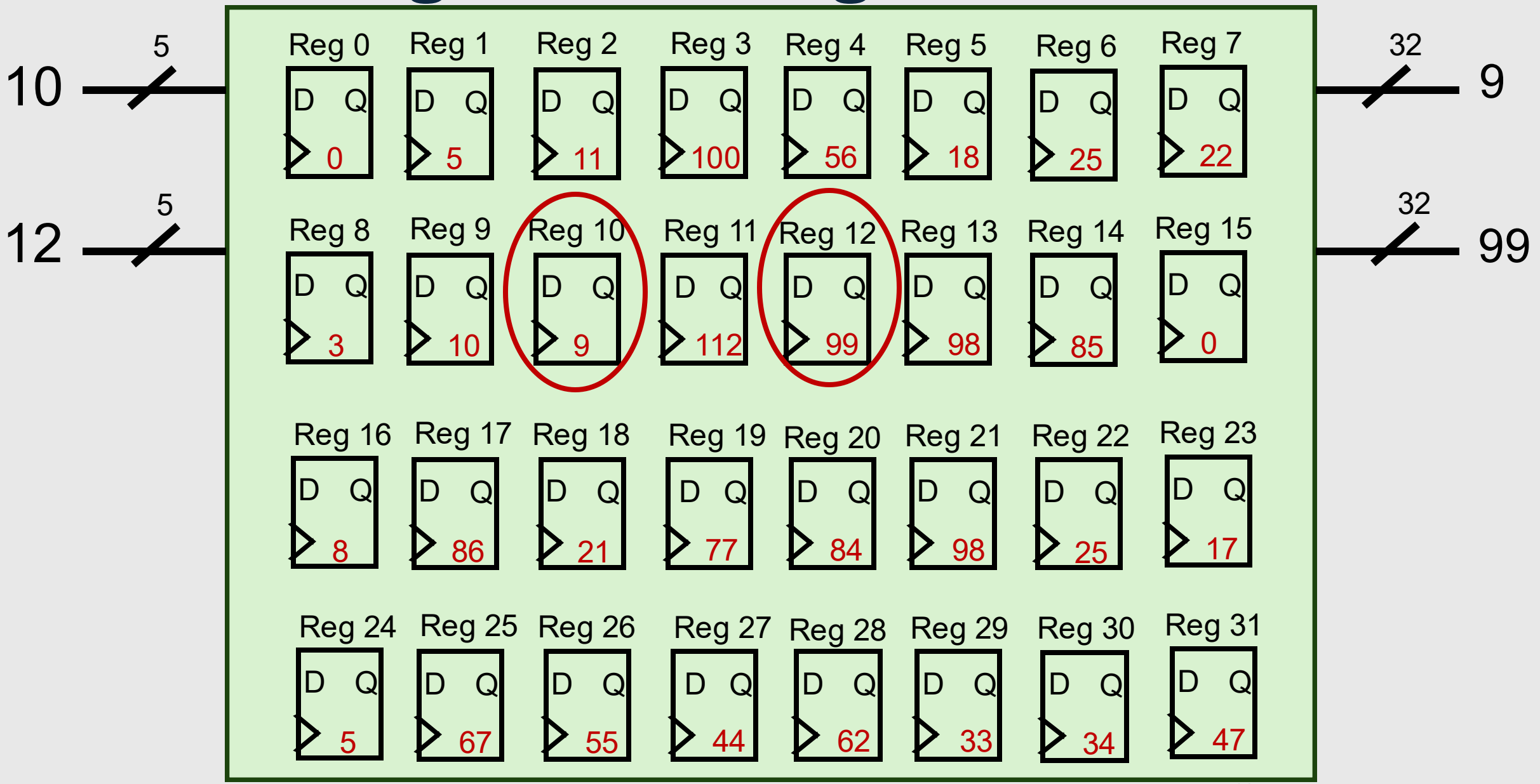
Reading from the Register File

- To read a value from the register file, set one of the read register inputs to the number of the register you want to read. The value will appear on the corresponding output.
- For example, if you want to read the value stored inside $10, you can set Read Register 1 to 10. The value of $10 will appear on the Read Data 1 output.

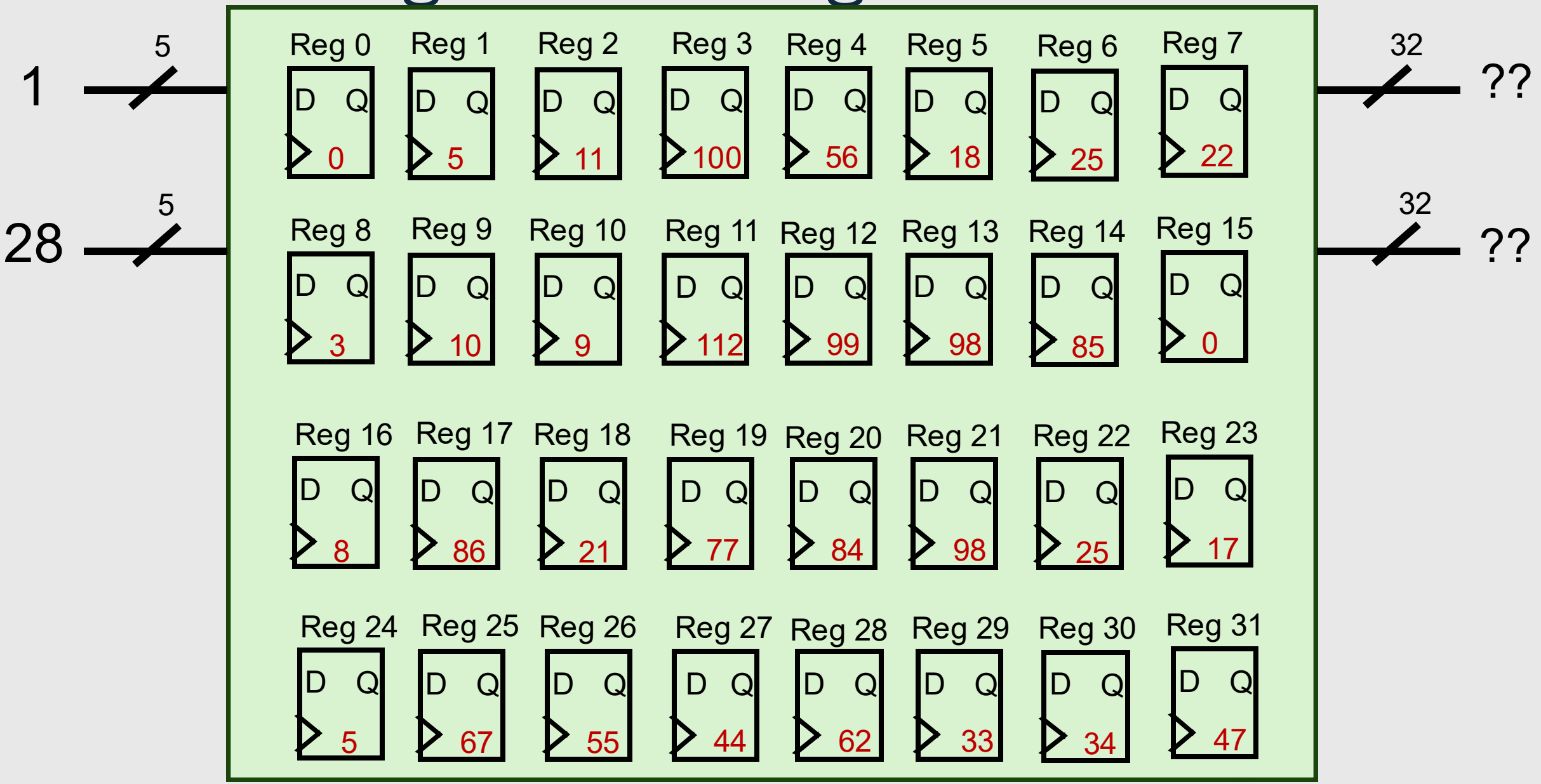
Ex1: Reading from the RegFile



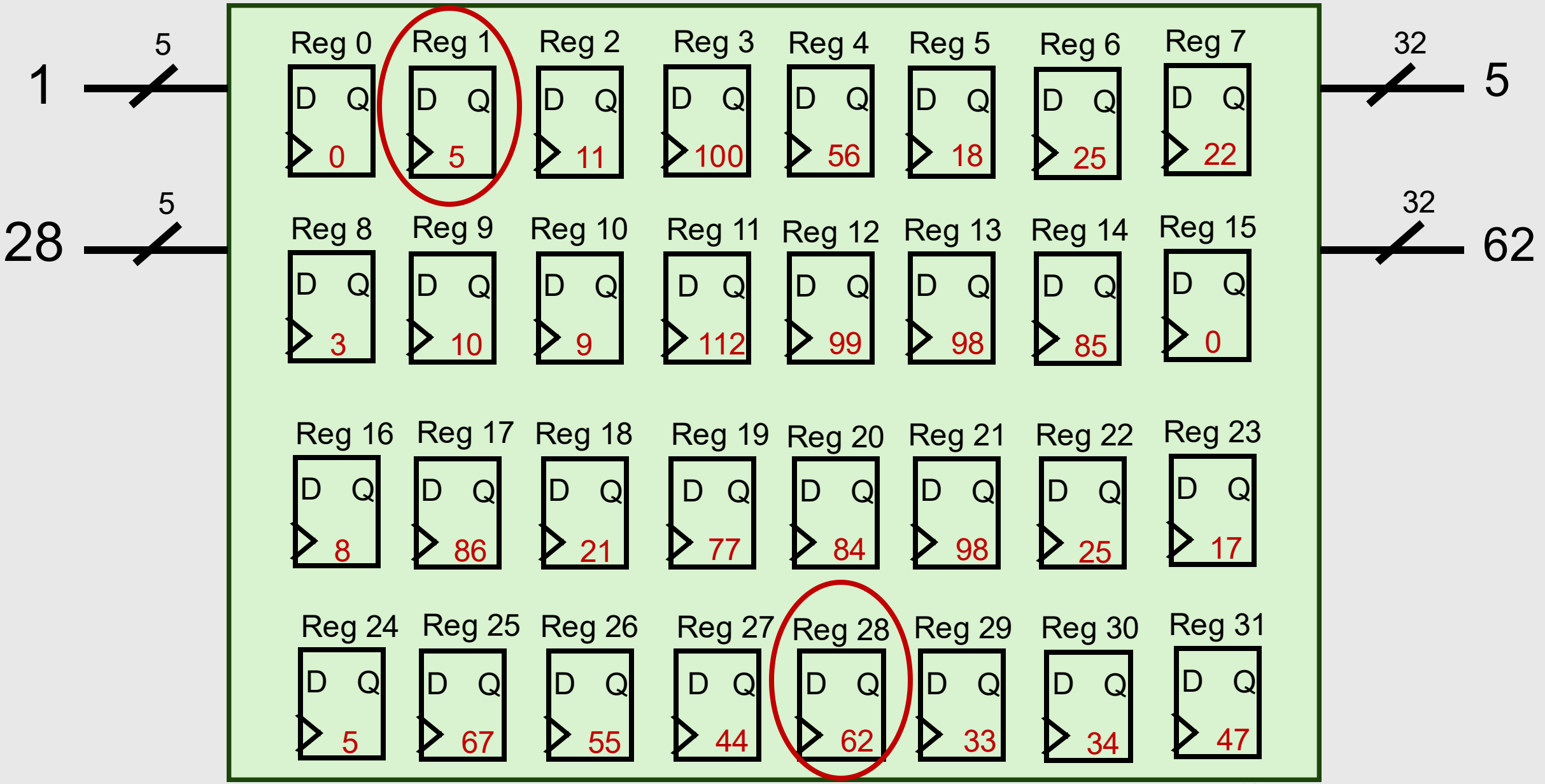
Ex1: Reading from the RegFile



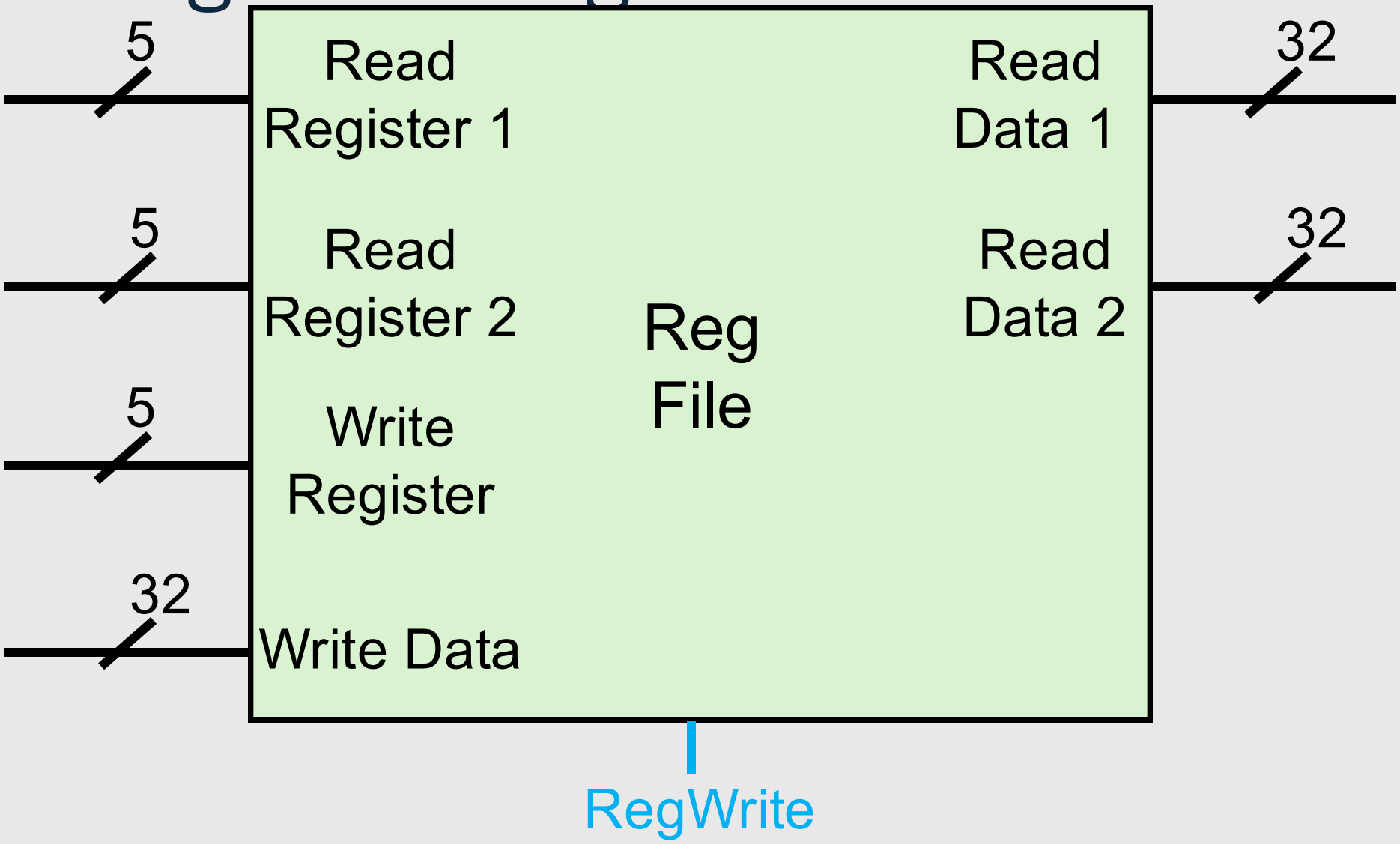
Ex2: Reading from the RegFile



Ex2: Reading from the RegFile



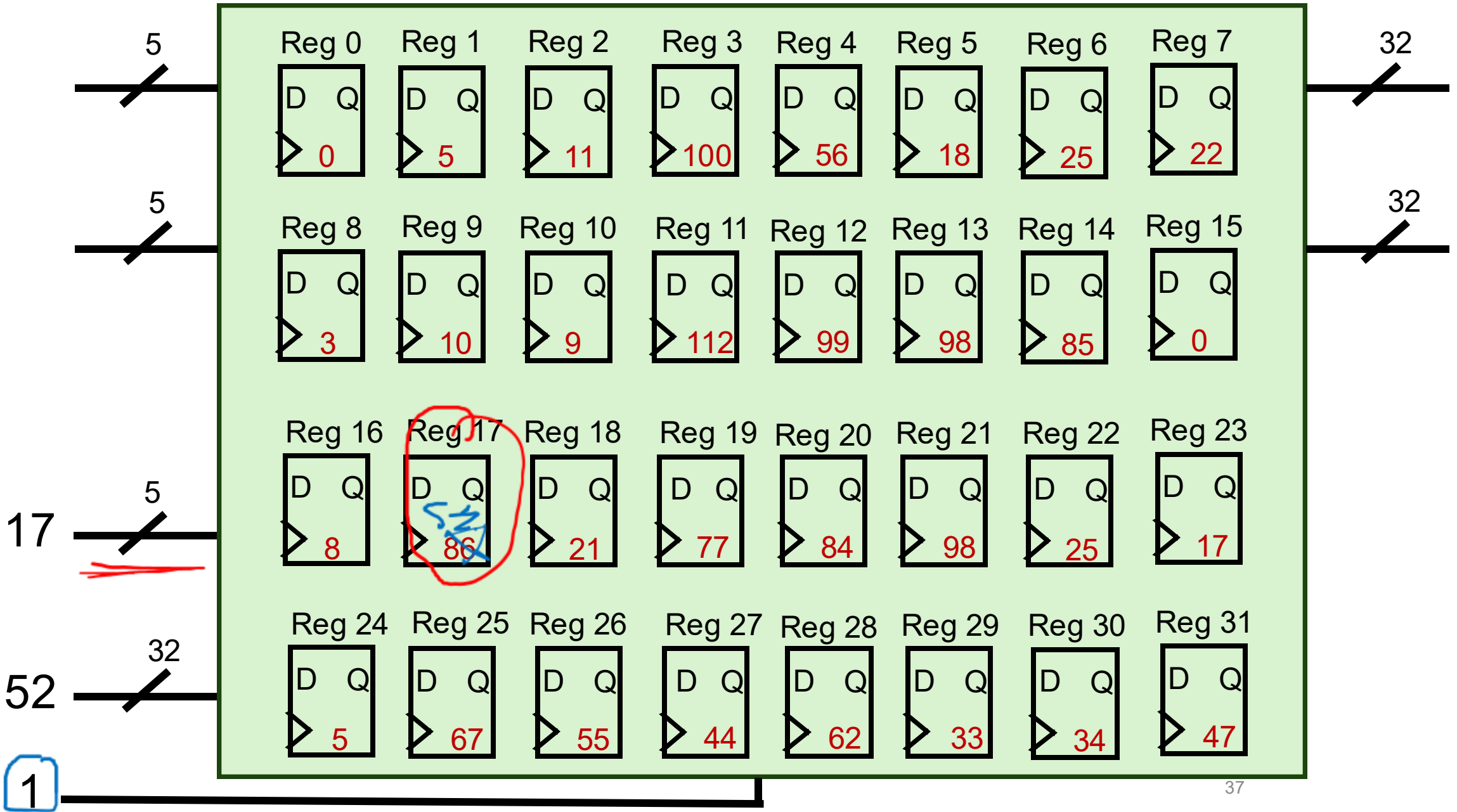
Writing to the Register File



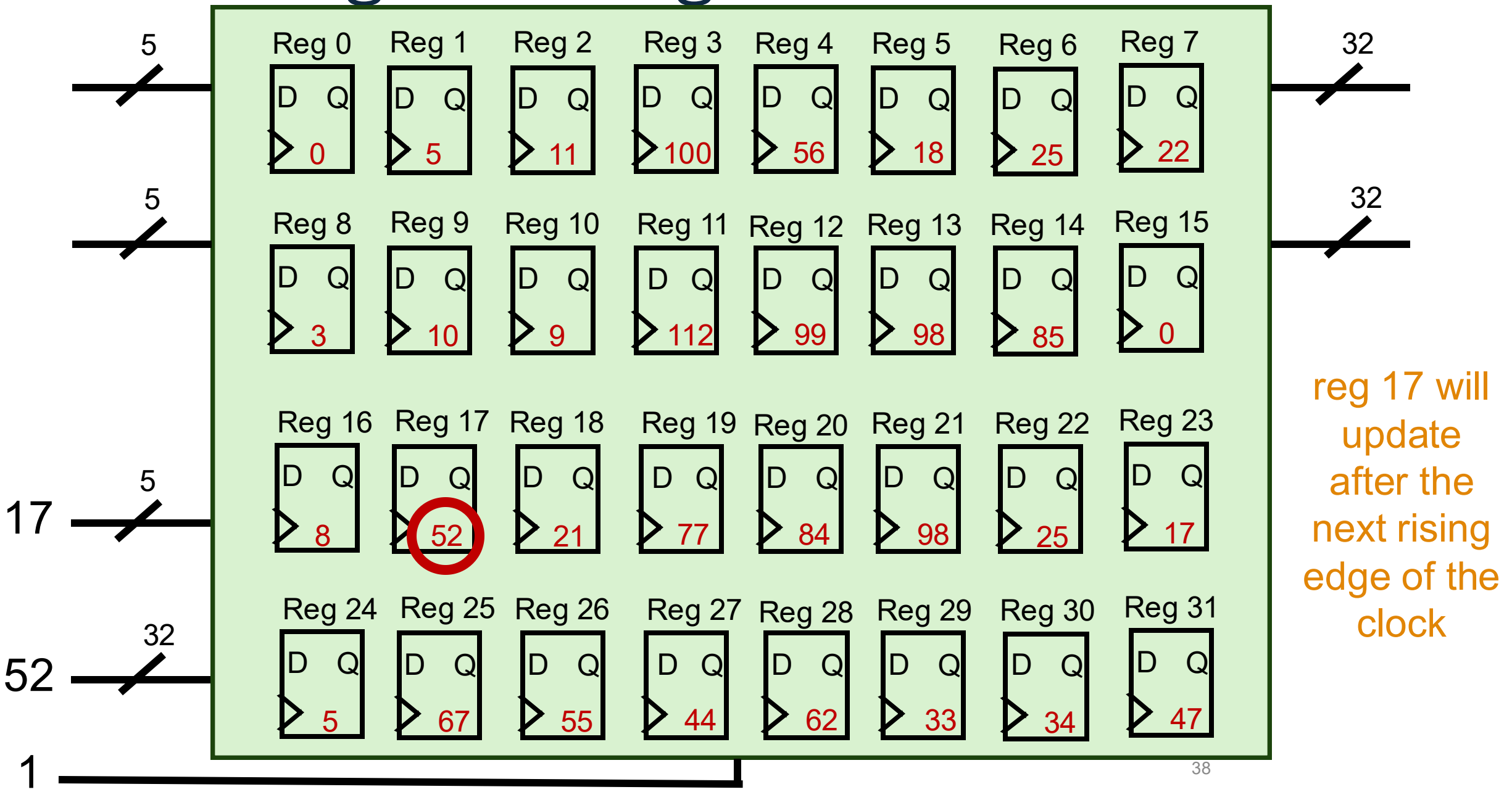
Writing to the Register File

- To write to a register inside of the register file, set the following signals
  - *Write register: The number of the register you are writing to*
  - *Write data: The data you want to store inside the register*
  - *RegWrite: This is a control signal whose value needs to be 1 to write to a register*
    - When we are not writing to a register, this value should be 0

Ex1: Writing to the RegFile

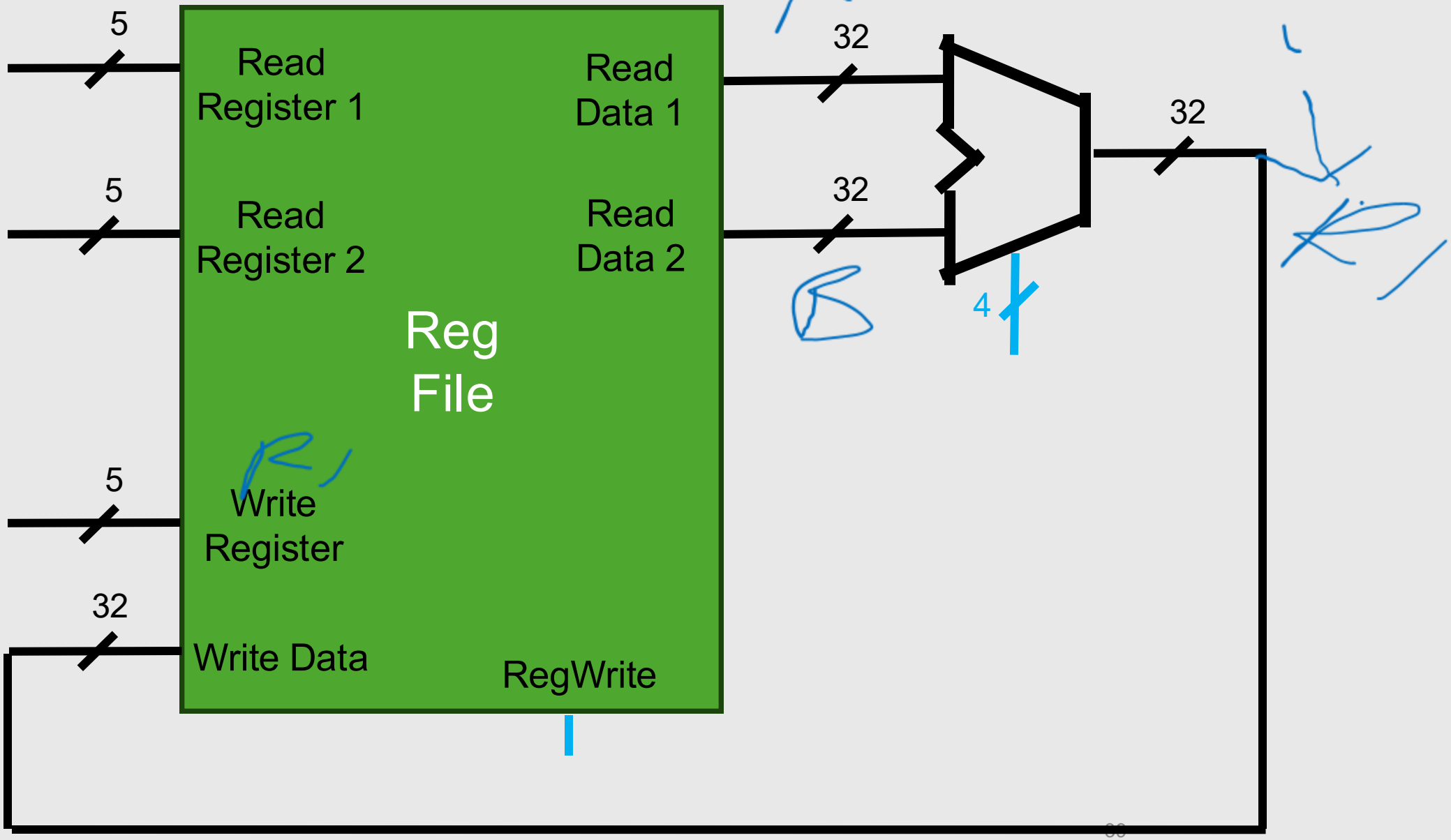


Ex1: Writing to the RegFile

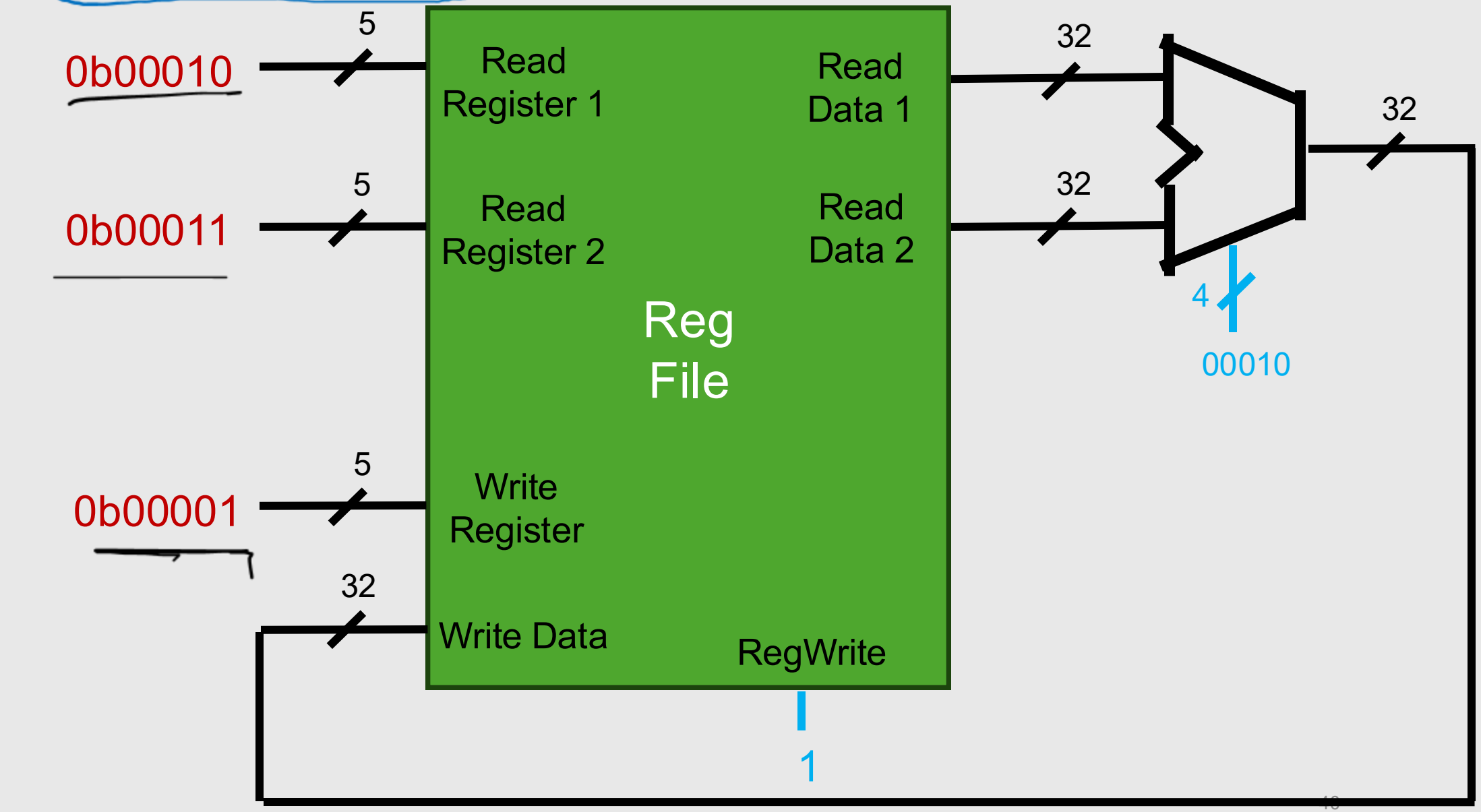


reg 17 will update after the next rising edge of the clock

r1 = r2 + r3


5
Read Register 1
5
Read Register 2
5
Write Register
32
Write Data
Reg File
Read Data 1
Read Data 2
RegWrite
32
32
32
4

r1 = r2 + r3


0b00010
5
Read Register 1
Read Data 1
32
0b00011
5
Read Register 2
Read Data 2
32
32
4
00010
Reg File
0b00001
5
Write Register
32
Write Data
RegWrite
1

Recall: Machine Code

High-level programming languages

Assembly
Low-level languages

Machine code

```
// High-level (C)
int add3(int a, int b)
{
    return a + b + 3;
}
```

```
# Matching RISC-V assembly
# a0 = a, a1 = b;  return in a0

add   a0, a0, a1   # a0 = a0 + a1
addi  a0, a0, 3    # a0 = a0 + 3
ret                # return
```

```
0000000 01011 01010
000 01010 0110011

000000000011 01010
000 01010 0010011

000000000000 00001
000 00000 1100111
```

Easy for humans to read/write

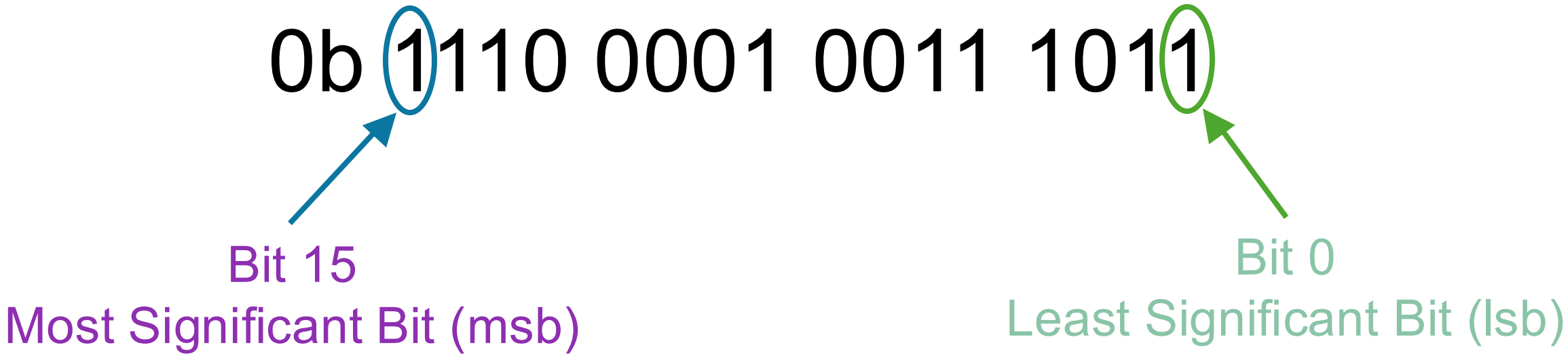
Human readable

Machine-readable

Reading Binary

0b 1110 0001 0011 1011

Reading Binary



rd rs1 rs2

r1 = r2 + r3

R-Type

funct7	source register 2	source register 1	funct3	destination register	opcode
0000000	00011	00010	000	00001	01101
31 25	24 20	19 15	14 12	11 7	6 0

inst → bin / bin → inst

encoding / decoding

r1 = r2 + r3

funct7	source register 2	source register 1	funct3	destination register	opcode
0000000	00011	00010	000	00001	0110011
31 25	24 20	19 15	14 12	11 7	6 0

0000000000110001000000010110011

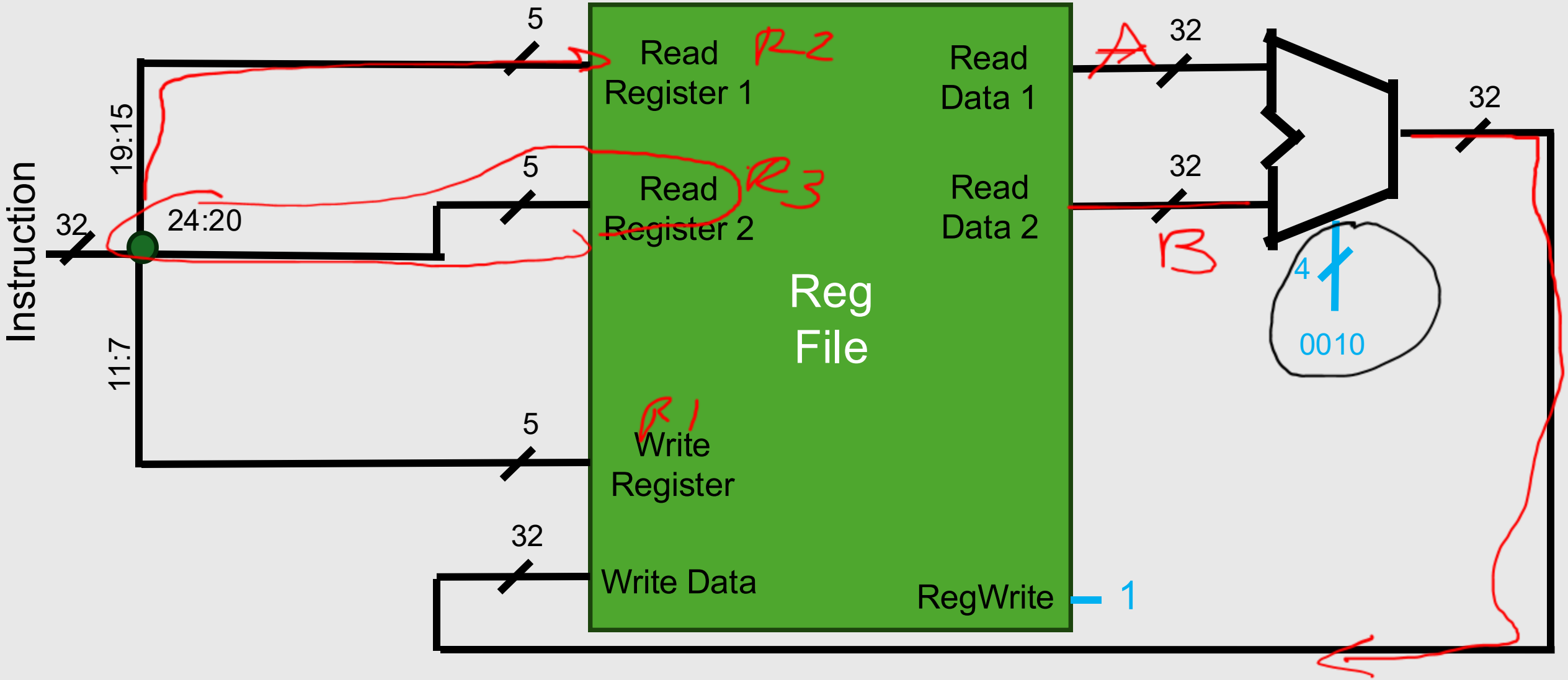
Opcode is always 0110011 for integer R-type ALU instructions

funct3 is always the primary operation code group (e.g. 000 = add/sub. 100 = xor, 111 = and)

funct7 is used to further specify the operation (e.g. distinguishes add vs. sub. srl vs. sra)

funct7	rs2	rs1	funct3	rd	opcode
0000000	00011	00010	000	00001	0110011

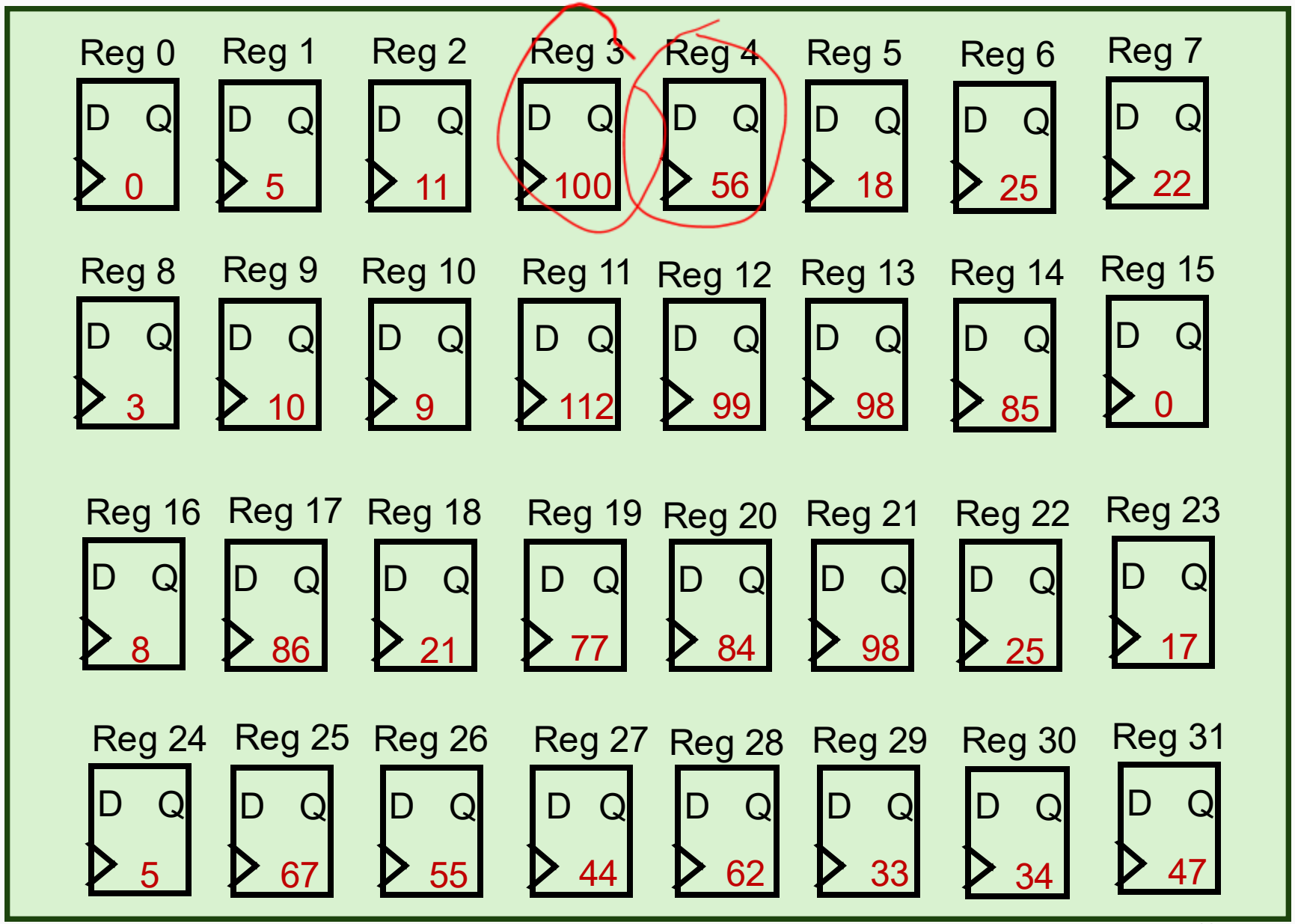
31 25 24 20 19 15 14 12 11 7 6 0



What are the contents of the register file after the following instructions are executed?

```
0x003202b3
0x00528533
0x00EA6020
```

Contents of RegFile



1 hex – 4 bits

What are the contents of the register file after the following instructions are executed?

0x003202b3

B ⇒ 1011

funct7	source register 2	source register 1	funct3	destination register	opcode
0000000	00011	00100	000	00101	
31 – 25	24 – 20	19 – 15	14 – 12	11 – 7	6 – 0

0000 0000 0011 0010 0000 0010 1011 0011

What are the contents of the register file after the following instructions are executed?

`0x003202b3`

`0b00000000001100100000000101010110011`

funct7	source register 2	source register 1	funct3	destination register	opcode
0000000	00011	00100	000	00101	0110011
31 25	24 20	19 15	14 12	11 7	6 0

x3 x4 x5

$5 = $4 + $3

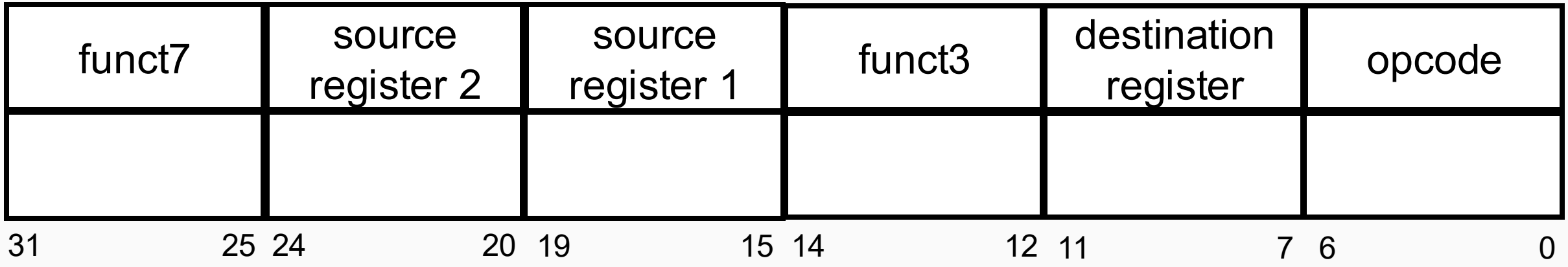
$5 = 56 + 100

$5 = 156

add x5, x4, x3

What are the contents of the register file after the following instructions are executed?

`0x00528533`



What are the contents of the register file after the following instructions are executed?

`0x00a38633`