pollev.com/kakiryan

COMP311: *COMPUTER ORGANIZATION!*

Lecture 22: Loading, Storing, Memory

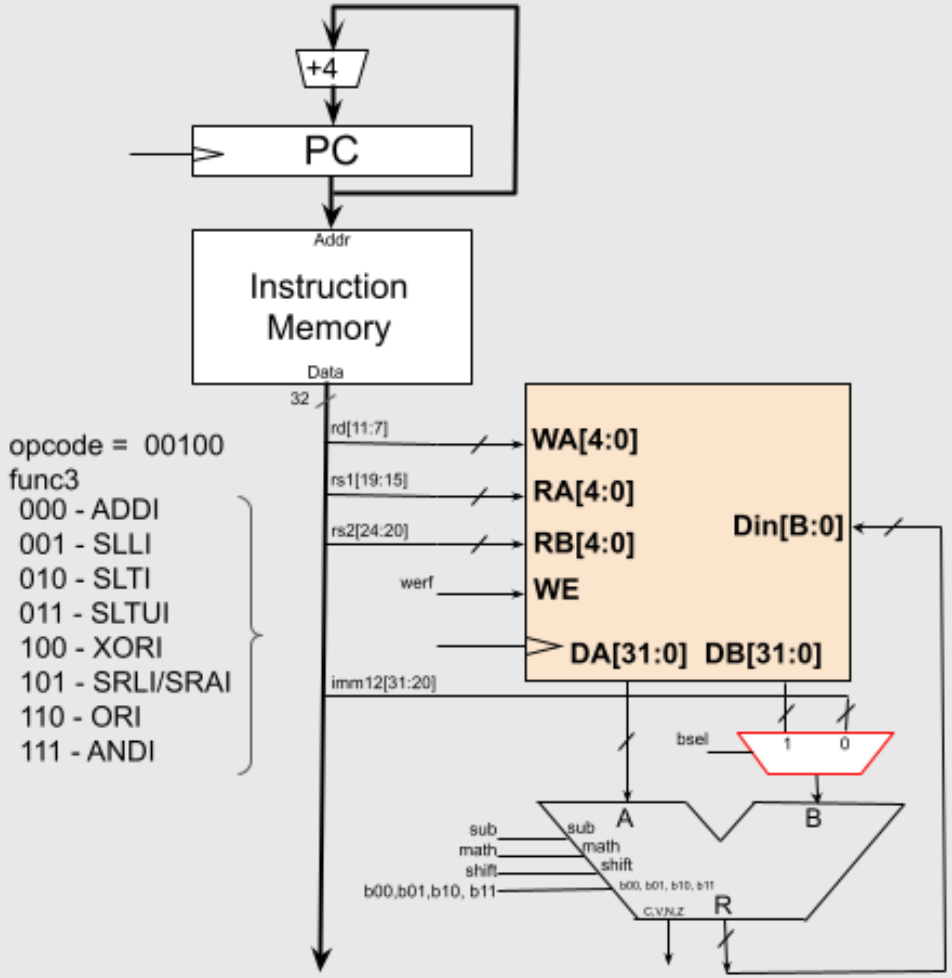
tinyurl.com/comp311-fa25

I-type Data Processing

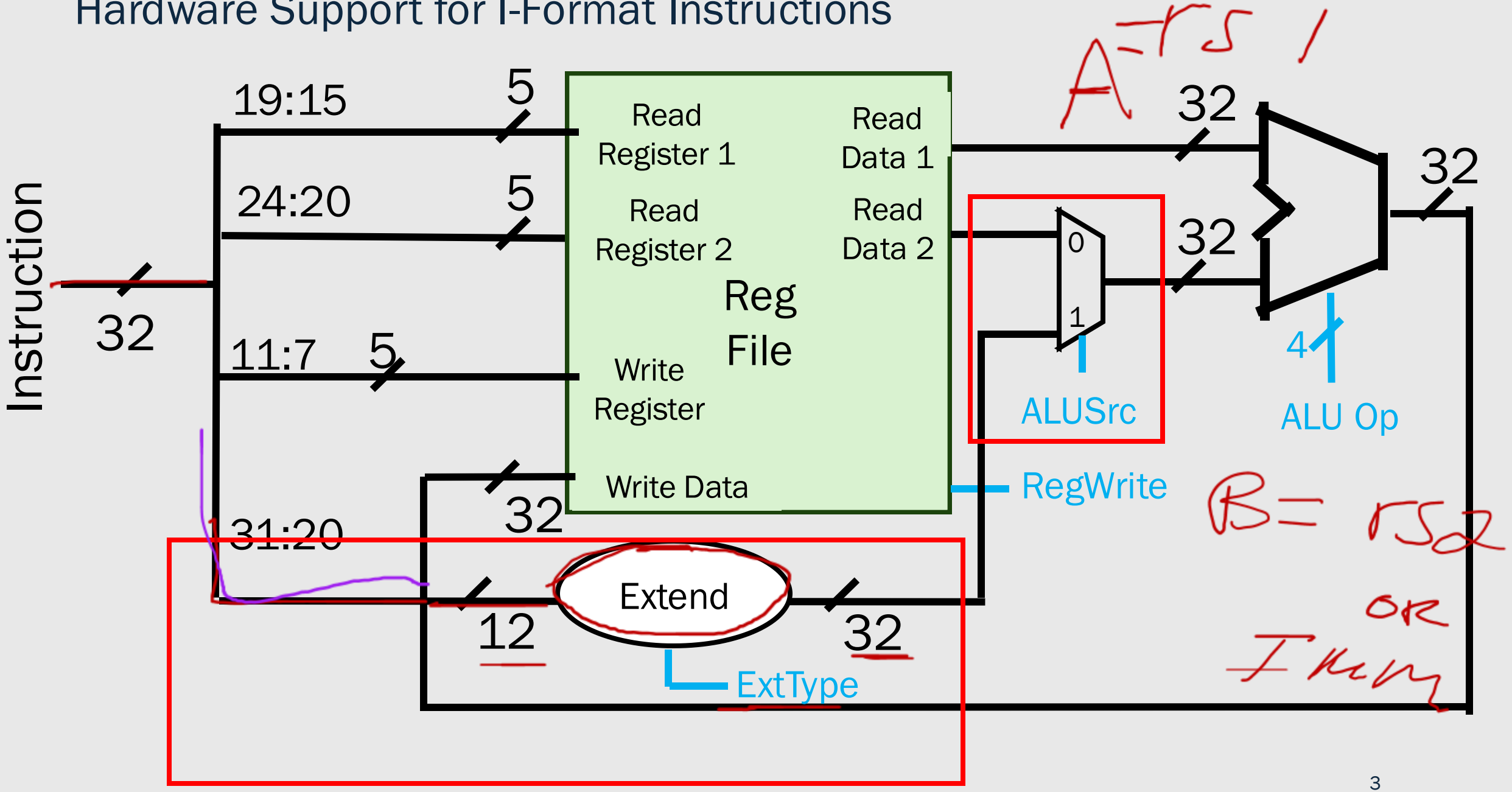
ALU instructions with a register and an immediate operand

Rd - register file write address
Rs1 - register source operand
Imm12 - 12-bit immediate operand

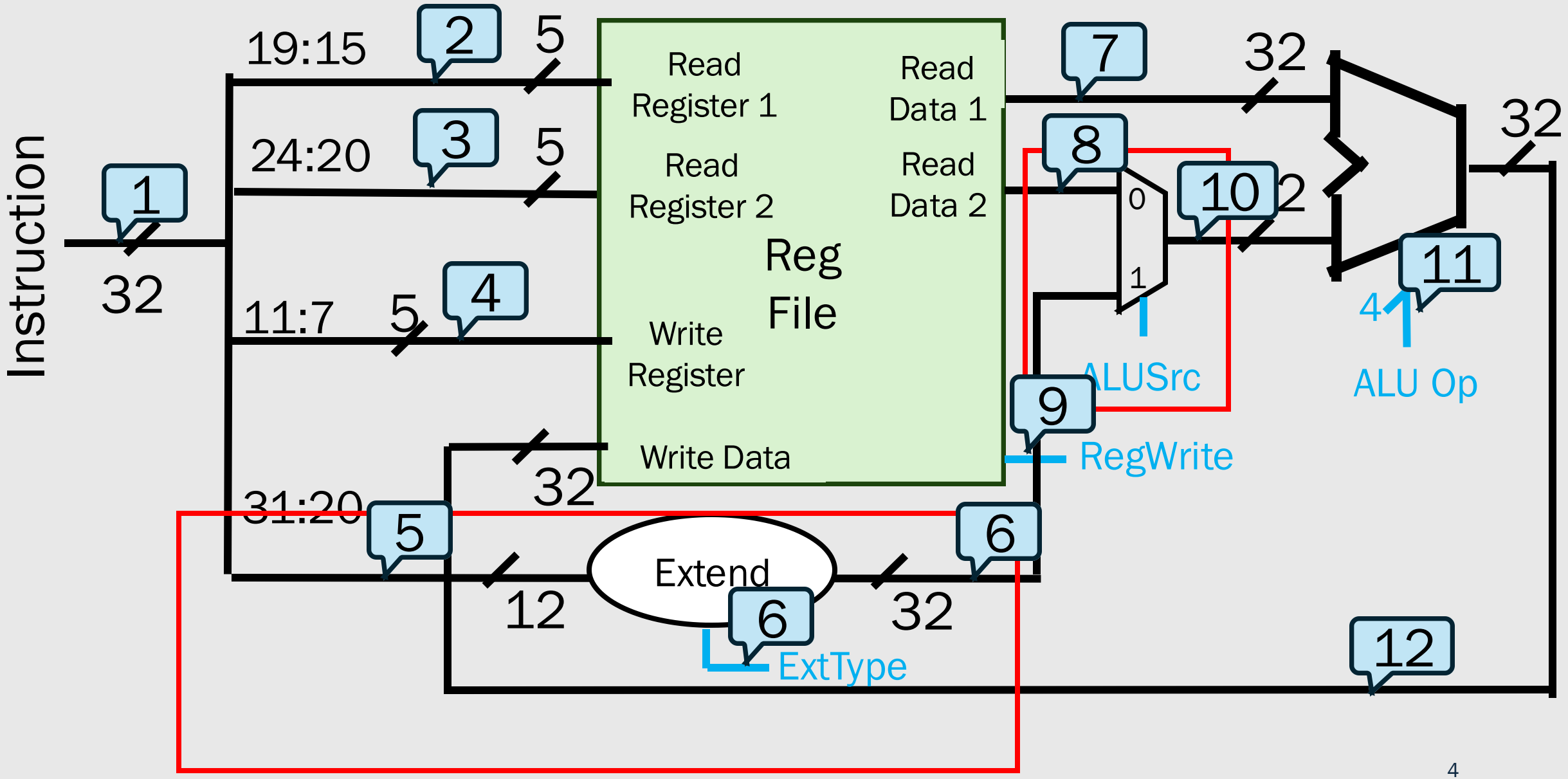
Adds a mux to the B input of the ALU
- 12-bit immediate value is sign-extended 20-bits



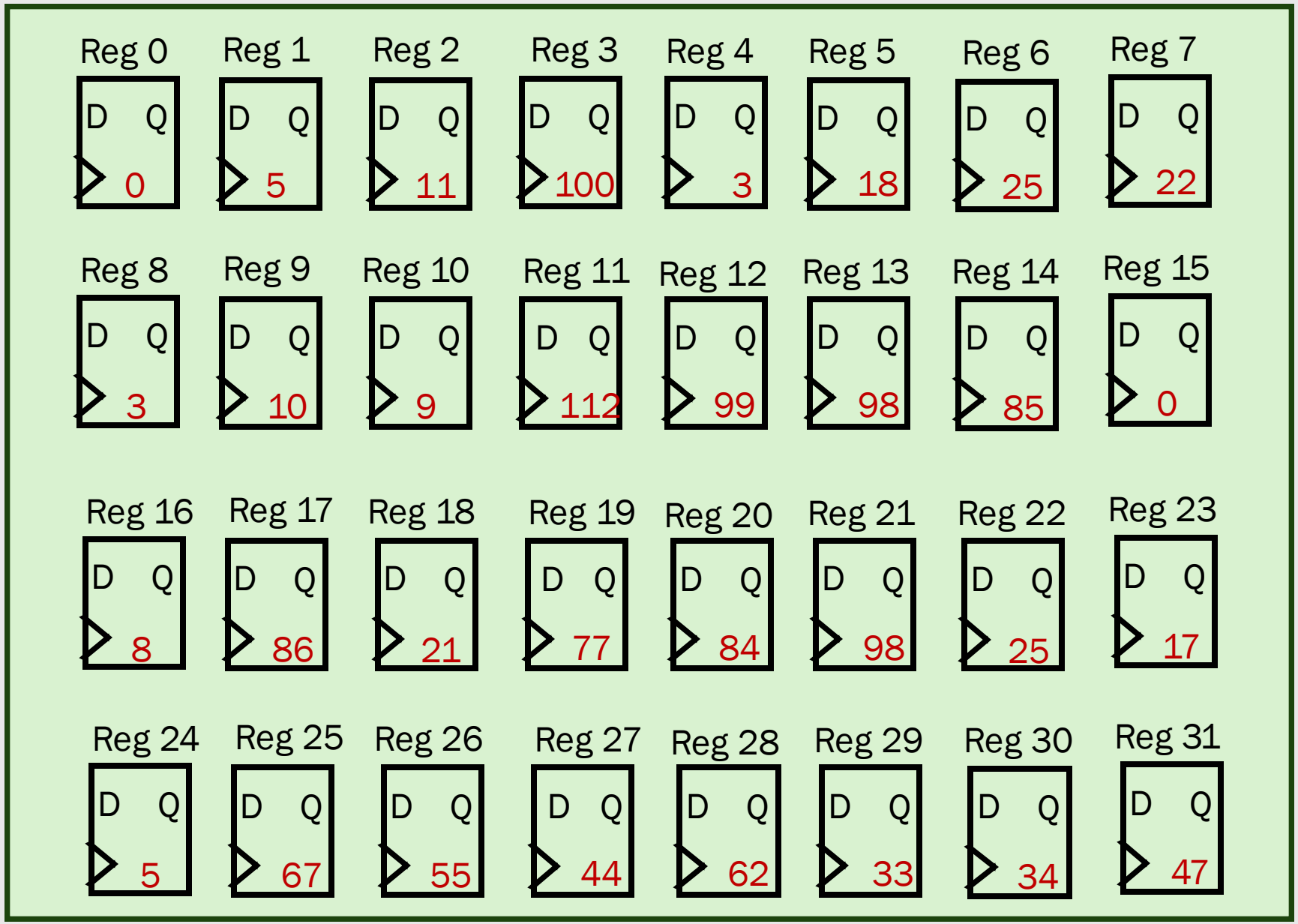
Hardware Support for I-Format Instructions



What is the value of each signal when executing `ori x4, x5 20`?



Contents of RegFile

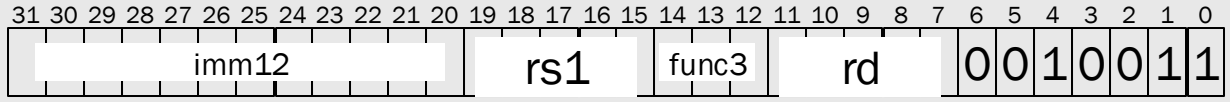


What is the value of each signal when executing `ori x4, x5, 20`

immediate	source register	funct3	destination register	opcode
0000 0001 0100	00101	110	00100	0010011
31 20	19 15	14 12	11 7	6 0

Why Built-in Constant operands?
(Immediates)

I-type:

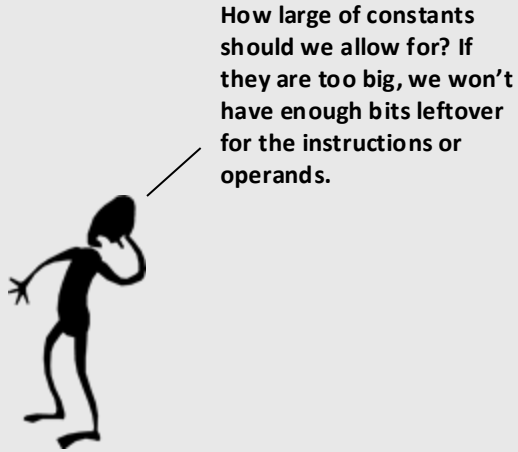


- Alternatives? Why not? Do we have a choice?
  - *put constants in memory (was common in older ISAs)*
- SMALL constants are used frequently (50% of operands)
  - *In a C compiler (gcc) 52% of ALU operations involve a constant*
  - *In a circuit simulator (spice) 69% involve constants*
  - *e.g.,* `B = B + 1; C = W & 0xff; A = B - 1;`
- ISA Design Principle:

  Make the common case easy

  Make the common case fast

Supporting immediates directly avoids a huge number of memory accesses!



Bigger Constants

If you use only addi and shifts, you can technically build any 32-bit constant.

We can load any 32-bit constant using a series of instructions

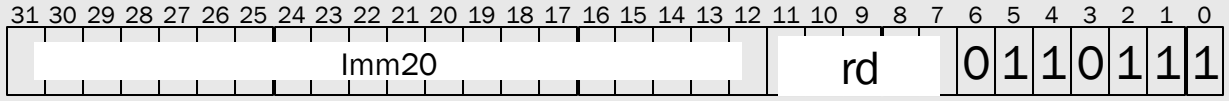
```
addi    x10,x0,0x123
slli    x10,x10,12
addi    x10,x10,0x456
slli    x10,x10,12
addi    x10,x10,0x78
```

Five instructions, is there a better way?

Load a small chunk, shift it up, add another chunk, shift again, add again…

But there there is a special instruction for constructing large constants., called "Load Upper Immediate" or **lui**. lui uses a new instruction format call the U-type. lui can be used as part of a two-instruction sequence to construct any 32-bit constant

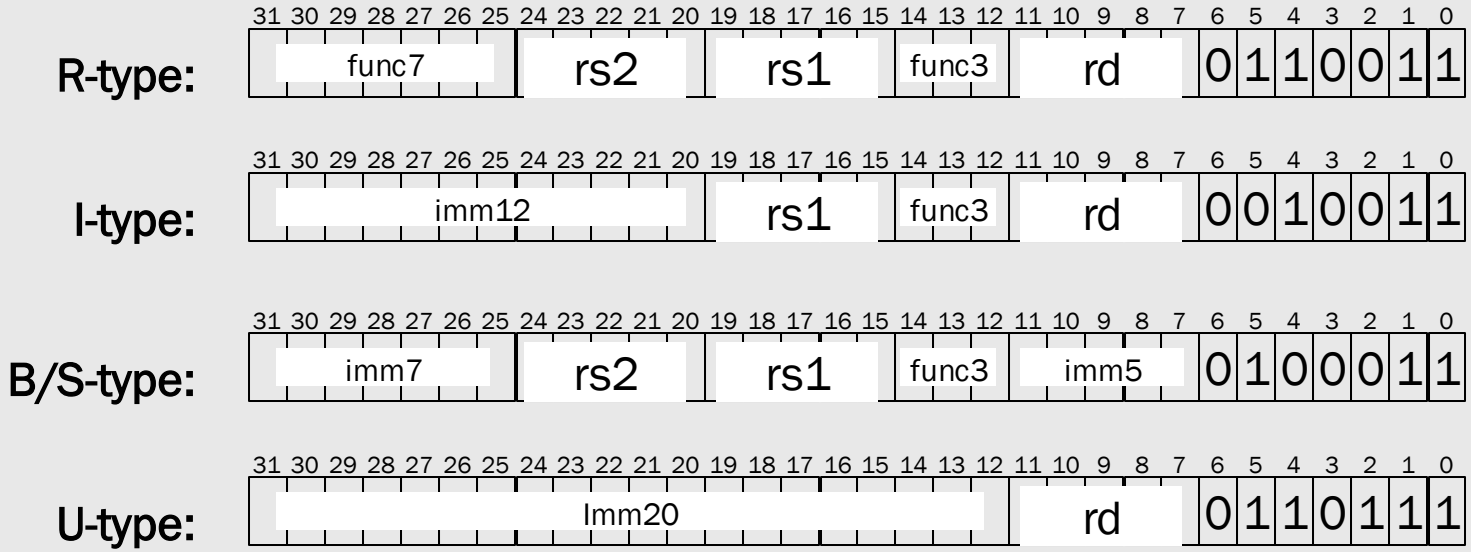
U-type:




```
lui    x10,0x12345   # encoded as 0x12345537
```

Key idea: ISA designers realized this pattern was common so it deserved HW support!

The miniRISC-V ISA

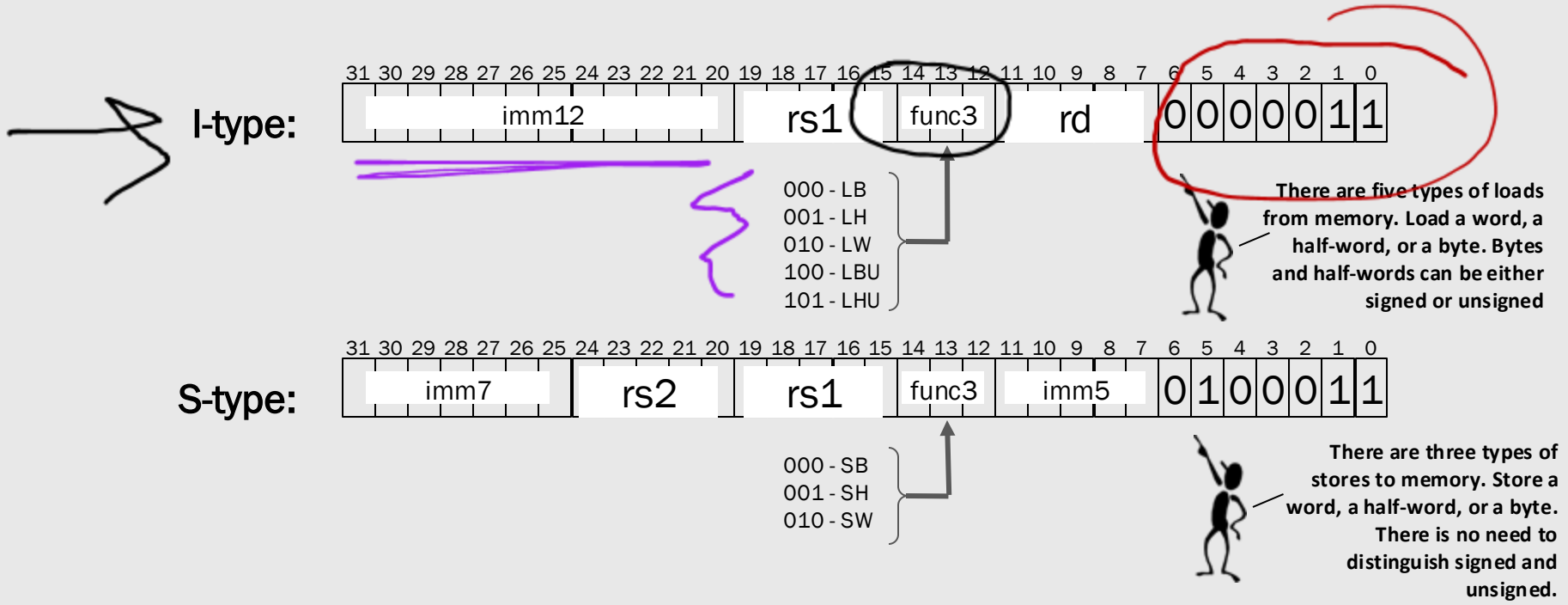


Four key instruction formats (each one has a corresponding opcode!):

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

Load and Store Instructions

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



Load Word (lw)

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

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

Store Word (sw)

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

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

Load Word vs Store Word

- Load Word
  - *Reads 4 bytes (one word) from memory*
- Store word
  - *Writes 4 bytes (one word) to memory*

Data Transfer

Loads and stores are the only instructions that touch memory in RISC-V (everything else in registers)

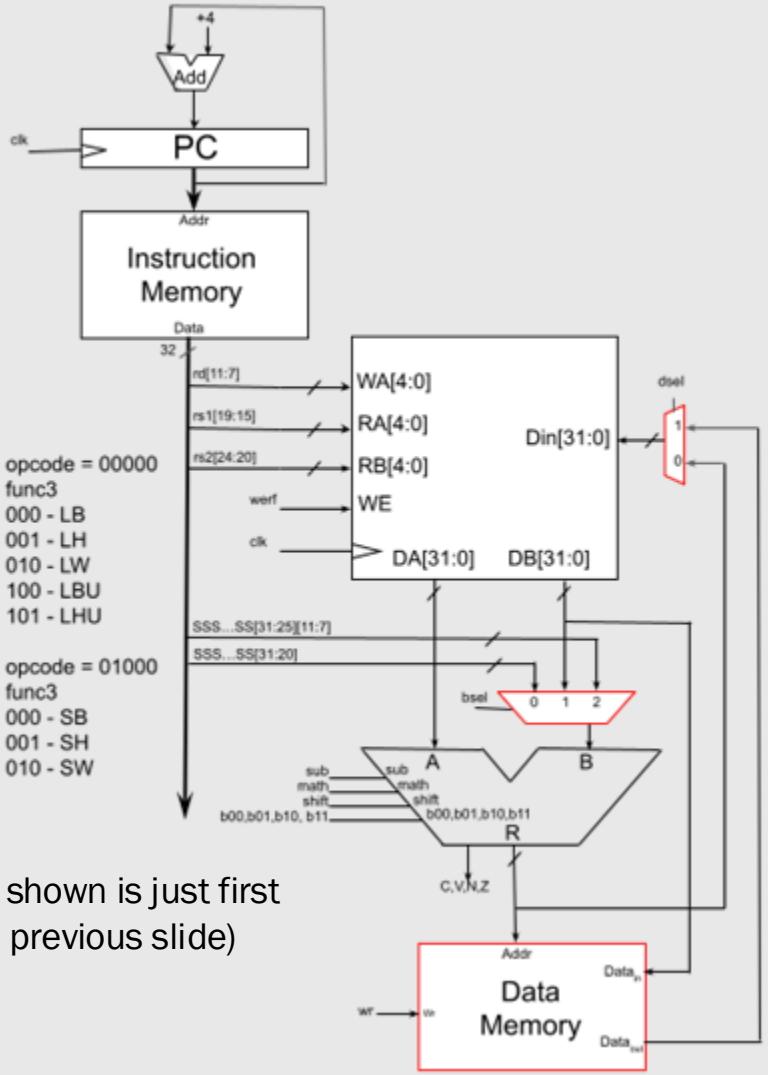
To access memory, we need two pieces of info: a base address coming from a reg & small offset encode in the instruction

Load/Store instructions using a base register and an immediate offset

- Rd - loaded register
- Rs1 - stored register
- Rs2 - base register
- Imm12 - 12-bit immediate load offset
- Imm7:Imm5 - 12-bit immediate store offset

Widens mux to the ALU's B input

- 12-bit immediate value is zero-extended 20-bits



Note: opcode shown is just first 5 bits (6-2 on previous slide)

Data Transfer

For a load, rs1 is the base register and rd is where the loaded value will go.

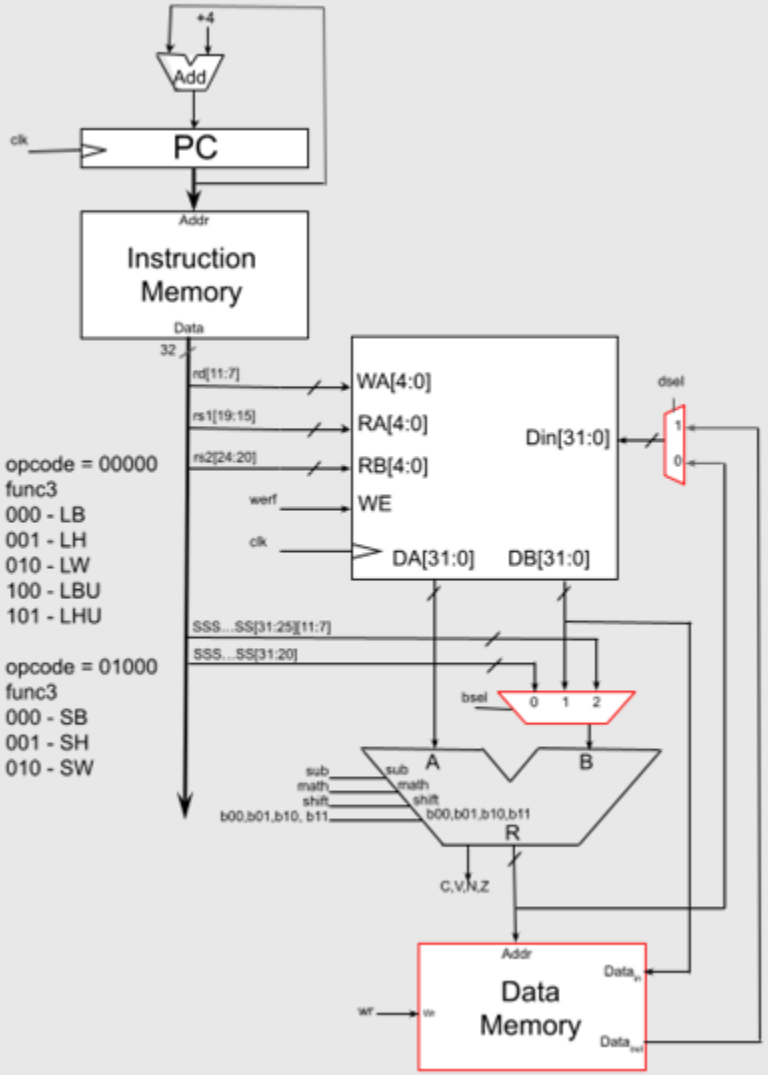
For a store, rs1 is still the base register, but now rs2 provides the value being written. Stores split their immediate across 2 fields

Load/Store instructions using a base register and an immediate offset

- Rd - loaded register
- Rs1 - stored register
- Rs2 - base register
- Imm12 - 12-bit immediate load offset
- Imm7:Imm5 - 12-bit immediate store offset

Widens mux to the ALU's B input

- 12-bit immediate value is zero-extended 20-bits



Data Transfer

The (sign-extended) immediate needs to be routed to the ALU, so the mux feeding the ALU's B input has to widen. Earlier, only reg values went in

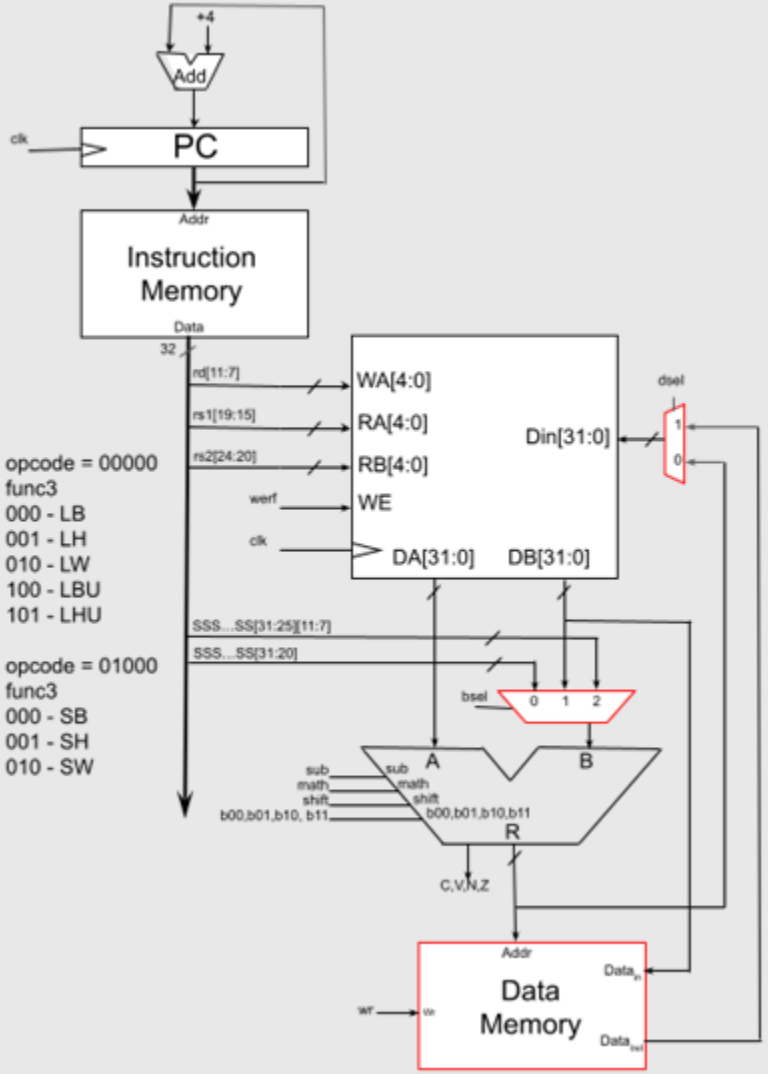
The ALU is now going to be adding the base register and the offset to compute the memory address

Load/Store instructions using a base register and an immediate offset

- Rd - loaded register
- Rs1 - stored register
- Rs2 - base register
- Imm12 - 12-bit immediate load offset
- Imm7:Imm5 - 12-bit immediate store offset

Widens mux to the ALU's B input

- 12-bit immediate value is zero-extended 20-bits



Data Transfer

For a load, the computed address from ALU goes to data memory, returns a 32 bit word.

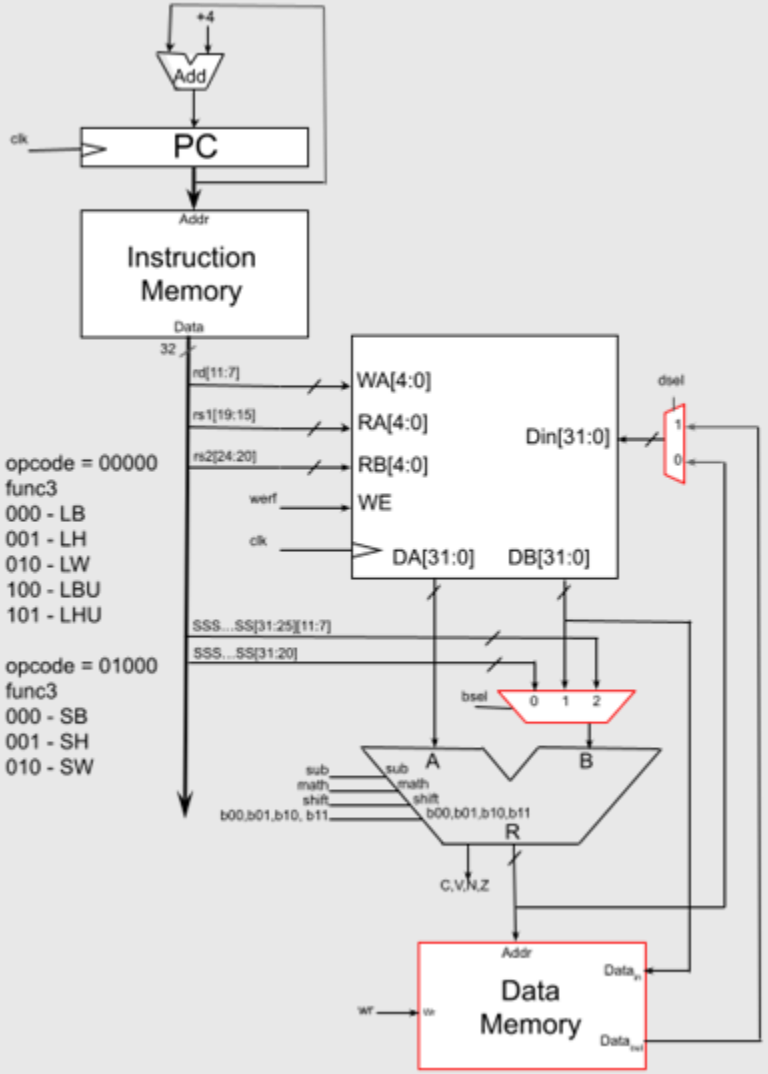
Write back mux selects it so it goes in Rd

Load/Store instructions using a base register and an immediate offset

- Rd - loaded register
- Rs1 - stored register
- Rs2 - base register
- Imm12 - 12-bit immediate load offset
- Imm7:Imm5 - 12-bit immediate store offset

Widens mux to the ALU's B input

- 12-bit immediate value is zero-extended 20-bits



Data Transfer

For a store, the address still comes from the ALU.

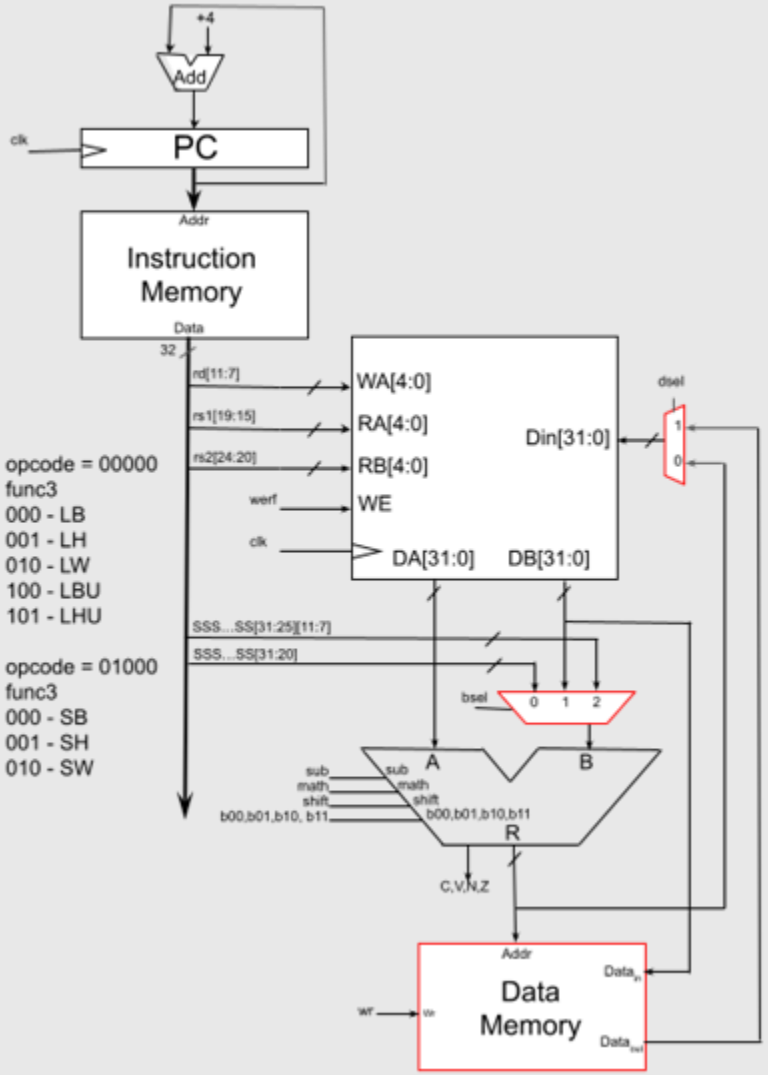
Instead of sending memory's output back to registers, we send rs2's value into memory at that address.

Load/Store instructions using a base register and an immediate offset

- Rd - loaded register
- Rs1 - stored register
- Rs2 - base register
- Imm12 - 12-bit immediate load offset
- Imm7:Imm5 - 12-bit immediate store offset

Widens mux to the ALU's B input

- 12-bit immediate value is zero-extended 20-bits



Data Transfer

Funct3 encodes the type of access: byte, halfword, word.

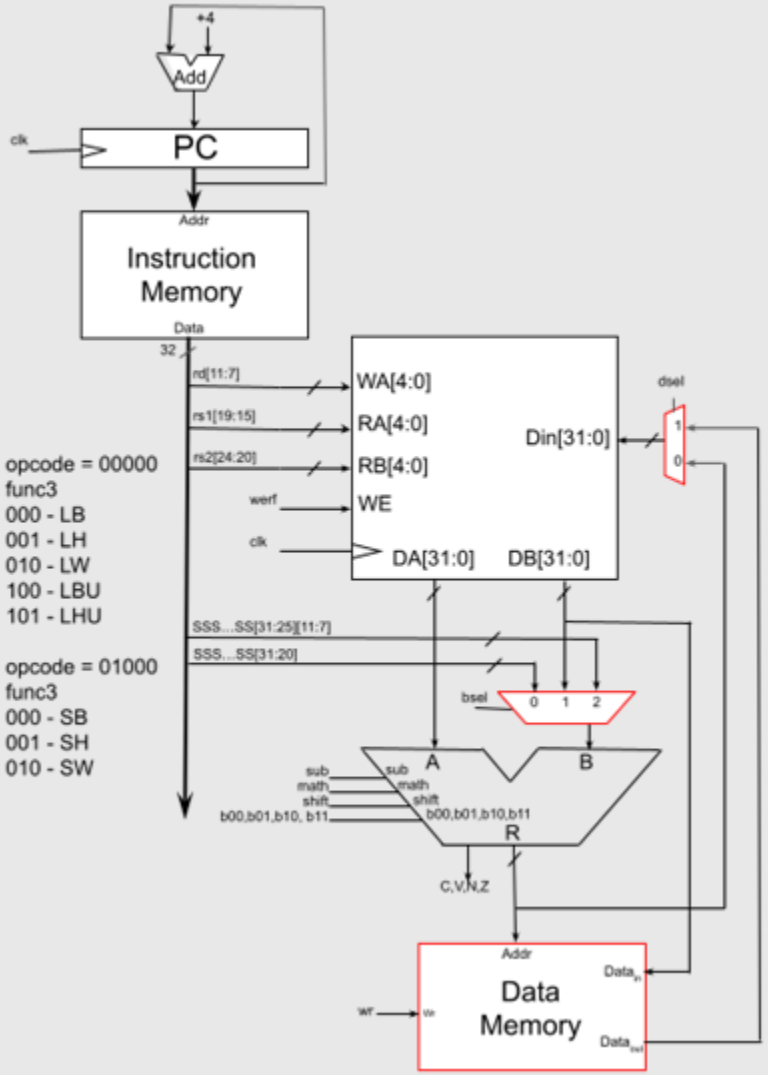
This datapath diagram doesn't show all masking and sign-extension logic for clarity of high-level flow

Load/Store instructions using a base register and an immediate offset

- Rd - loaded register
- Rs1 - stored register
- Rs2 - base register
- Imm12 - 12-bit immediate load offset
- Imm7:Imm5 - 12-bit immediate store offset

Widens mux to the ALU's B input

- 12-bit immediate value is zero-extended 20-bits



Load and Store Options

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

```
lw   rd,imm12(rs)
```

Rd ← Memory[Rs + imm12]
Rd is loaded with the contents of memory at the address found by adding the contents of the base register, Rs, to the supplied constant

```
lb   rd,-4(rs)
```

Rd ← sign_extend(Memory[Rs - 4])
Offsets can be either added or subtracted, as indicated by a negative sign. The byte is signed-extend to fill the 32 bits of

```
lw   rd,(rs)
```

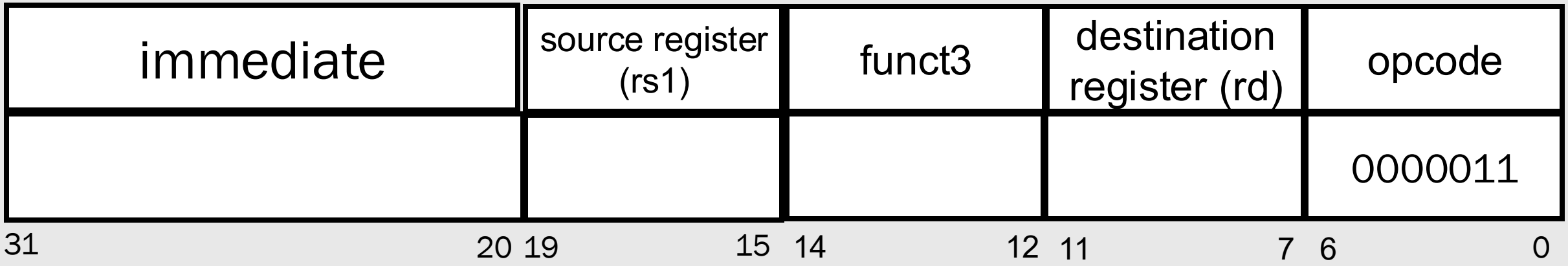
If no offset is specified it is assumed to be zero

```
sw   rd,12(rs)
sh   rd,(rs)
```

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

Load Word Encoding

I-Type, *but slightly different opcode!*



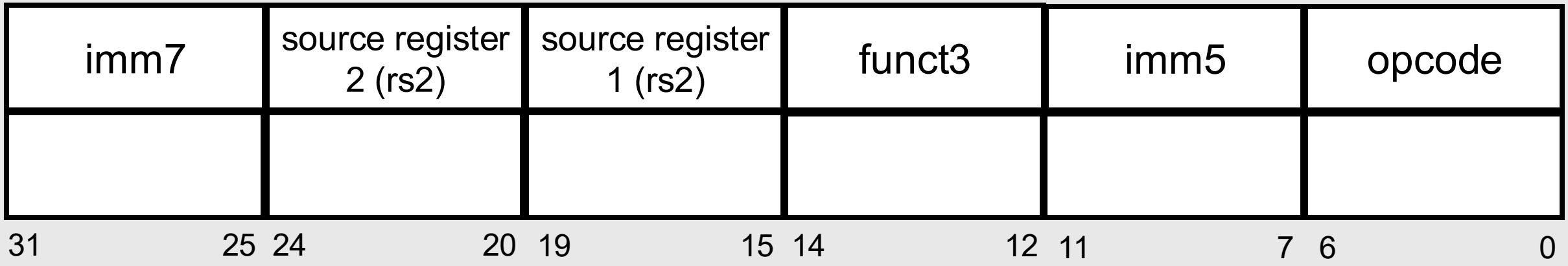
1. Compute address
2. Load data from memory
3. Store data into register file

lw rd, imm12(rs1)

R[rd] ← M[R[rs1] + SignExt(imm12)

Store Word Encoding

S-type



sw rs2, imm(rs1)

M[R[rs1] + SignExt(Imm)] = R[rs2]

opcode = 0b0100011

Translate the following instructions into machine code

lw x5, 0x20(x7)

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

sw x20, -12(x9)

imm7	source register 2 (rs2)	source register 1 (rs2)	funct3	imm5	opcode
31 25	24 20	19 15	14 12	11 7	6 0

Translate the following instructions into machine code

lw x5, 0x20(x7) 0x0203A283

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

sw x20, -12(x9) 0xFF44AA23

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

ACCESSING MEMORY

Memory

- The register file only contains 32 32-bit registers
- We can’t store all of our data in the register file. So where does it all go?
- ***In the main memory***

RISC-V Memory Model

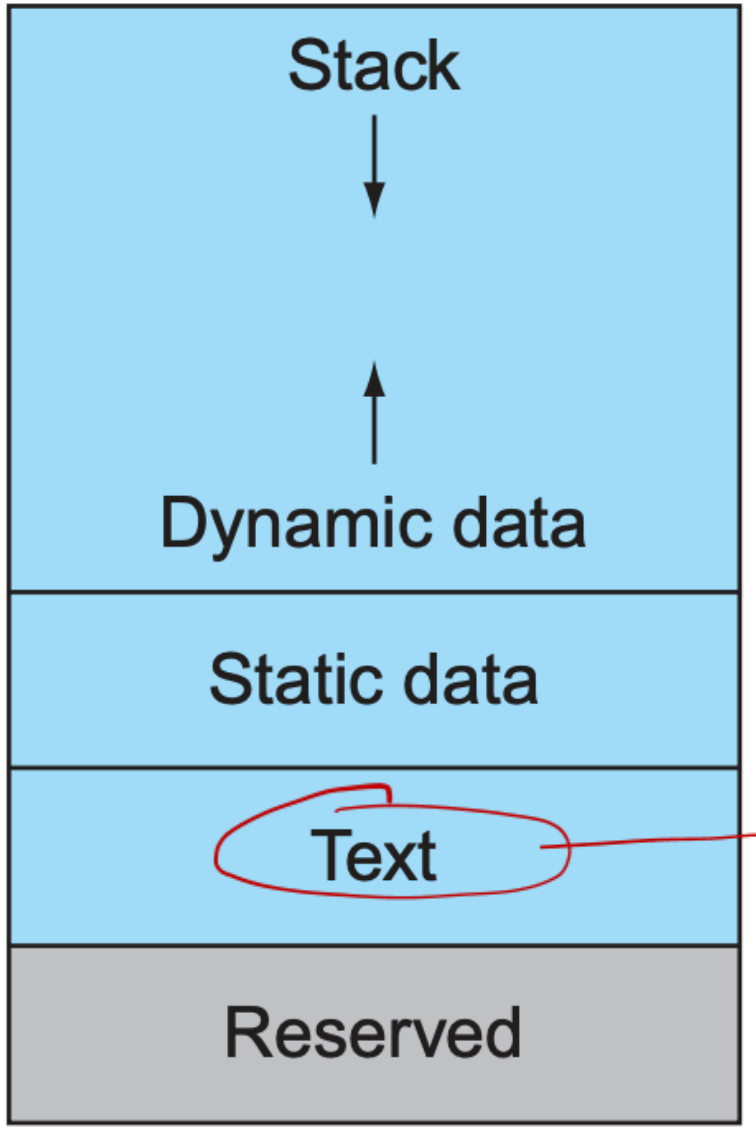
Used to manage function calls and local variables

Used for dynamic memory allocation

Variables allocated at compile-time

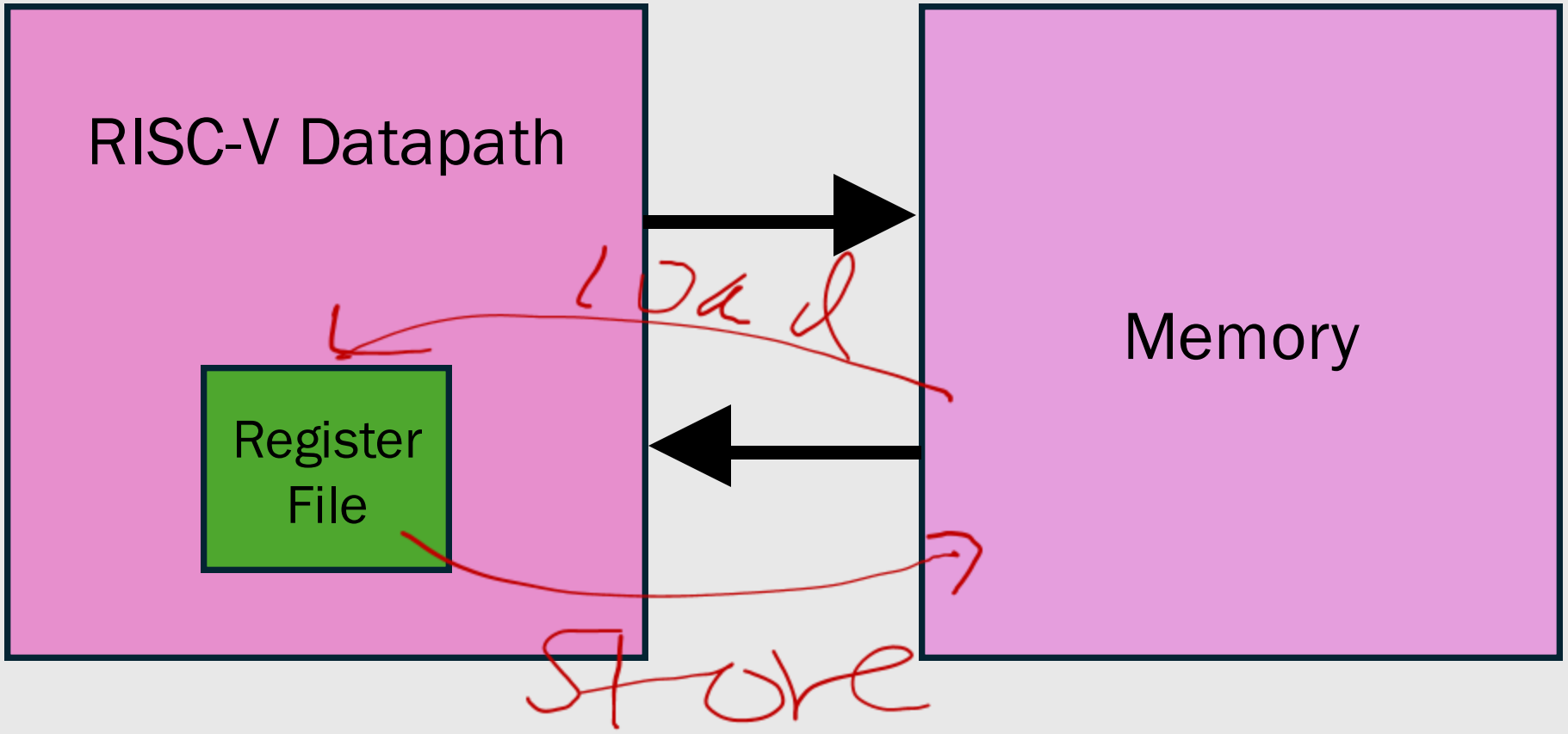
Programs

Reserved for the OS



code!

Only reaches out to memory when a **load** or a **store** happens!

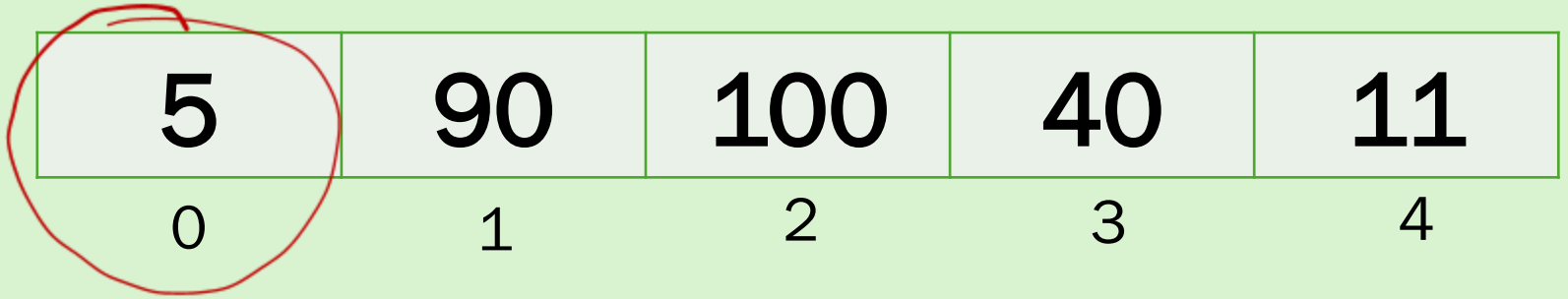


Memory

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

Memory vs. Register File

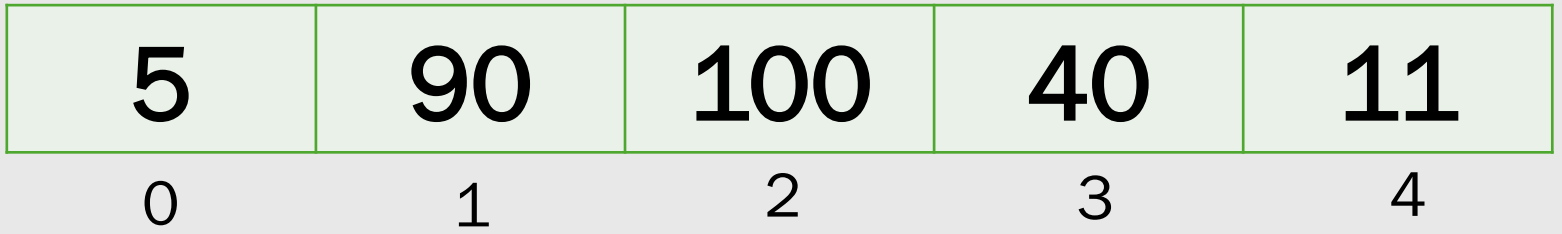
- Memory
  - *large*
  - *takes longer to access*
  - *not part of the CPU*
- Register File
  - *small*
  - *quick access*
  - *part of the CPU*



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

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

5



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

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

5

5	90	100	40	11
0	1	2	3	4

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

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

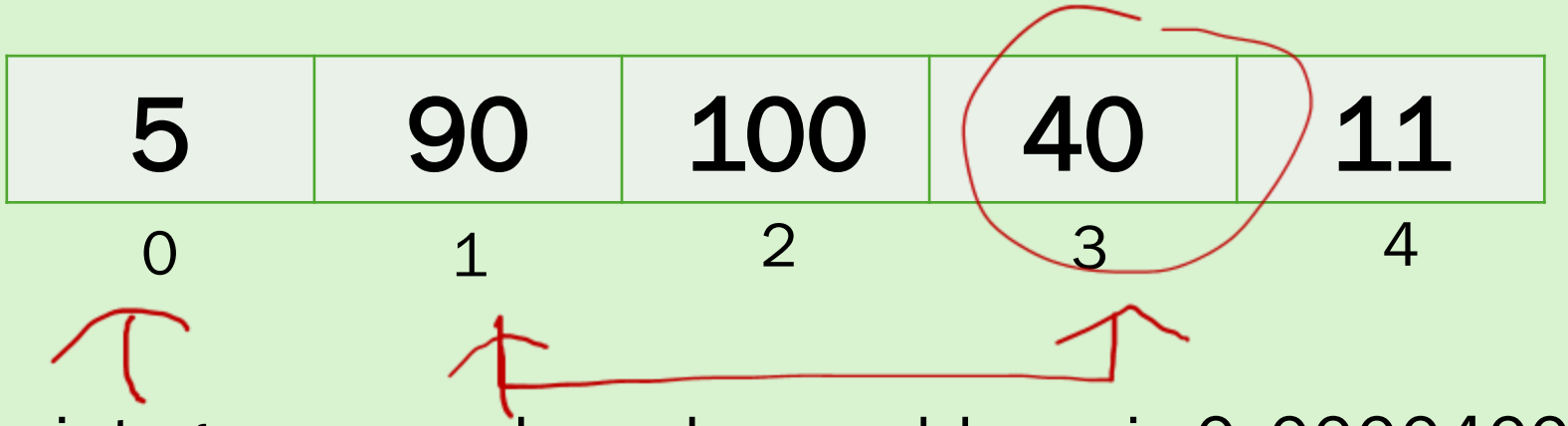
0x00004000 + 4 ⟹ 90

5	90	100	40	11
0	1	2	3	4

- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000.
- What is the value of $12 after executing the following instruction?

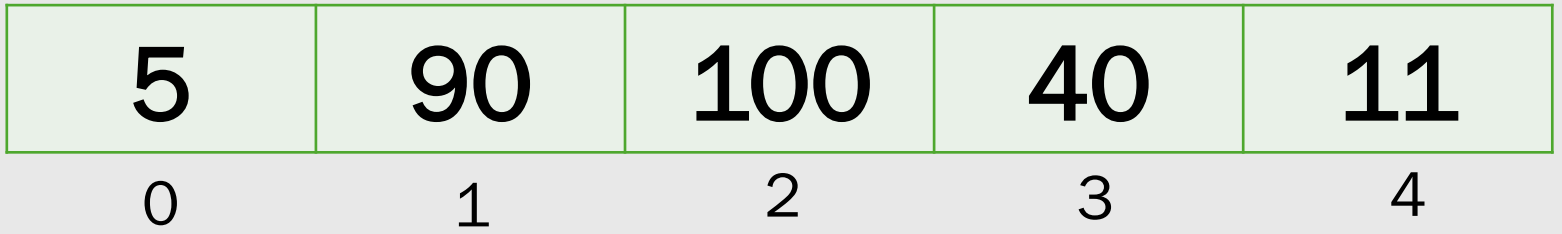
```
lw $12 4(x10)
```

90



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

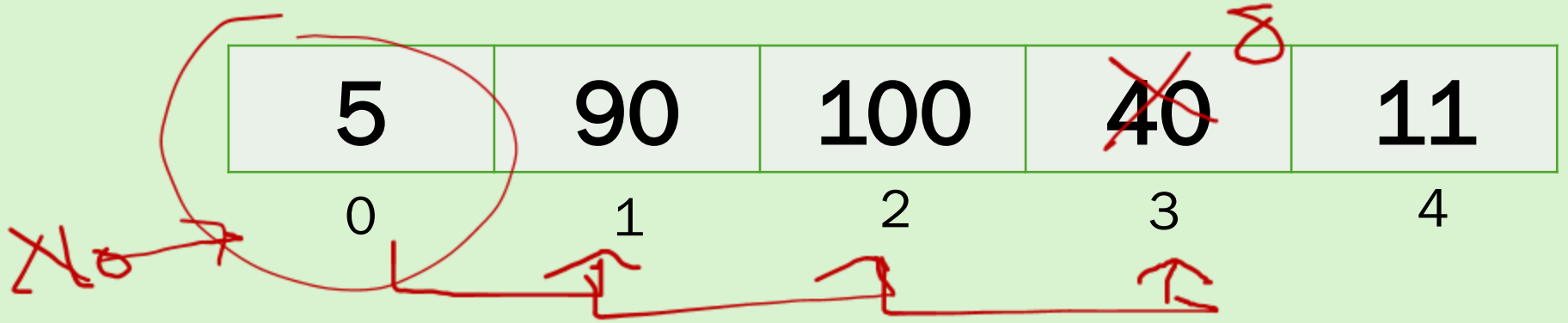
```
addi x10, x10, 4
lw x12, 8(x10)
```



- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume $10 = 0x00004000.
- What is the value of $12 after executing the following instructions?

```
addi 10 $10 4
lw x12 8(x10)
```

40

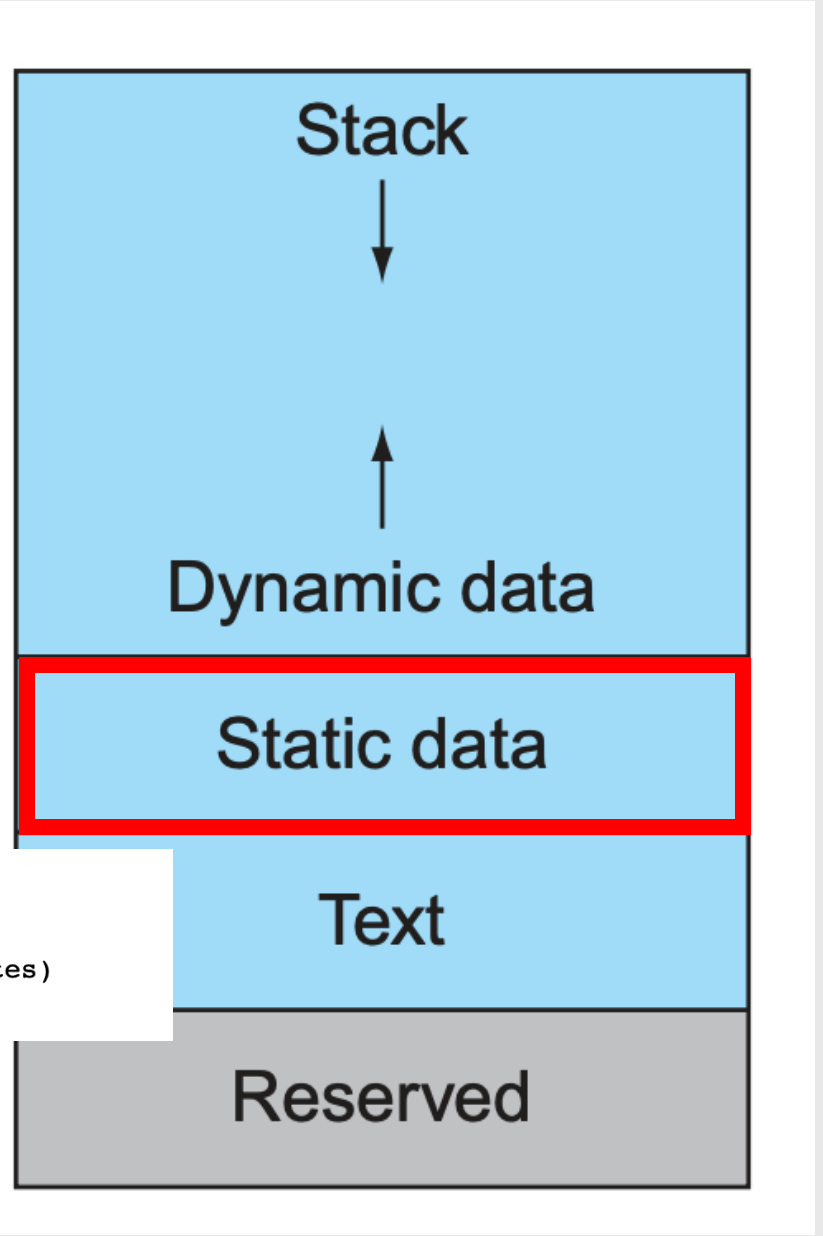


- Above is an integer array whose base address is 0x00004000.
- The size of an int is 4 bytes.
- Assume x10 = 0x00004000 and x7 = 8.
- What are the contents of the array after executing the following instruction?

```
sw x7, 12(x10)
```

Storing Arrays in Memory

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




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

Load Address

- How do you get the address of the array into a register?
  - *with an instruction called load address (la)*
- la rd Label
  - *places the address of **Label** in rd*
- This is actually a ***pseudoinstruction***

Loads the address of the label into *Reg[rd]*. It is typically implemented using the two instruction sequence:

```
auipc rd, label
addi rd, rd, label
```

A pseudoinstruction is a “fake” assembly instruction that doesn’t exist in HW, but assembler expands into one or more real RISC-V instructions!

Load Address

- How do you get the address of the array into a register?
  - *with an instruction called load address (la)*
- la rd Label
  - *places the address of **Label** in rd*
- This is actually a ***pseudoinstruction***

AUIPC is an instruction we will talk about more in a bit… it adds an immediate to the PC. Let's review what the PC is / related to program execution.

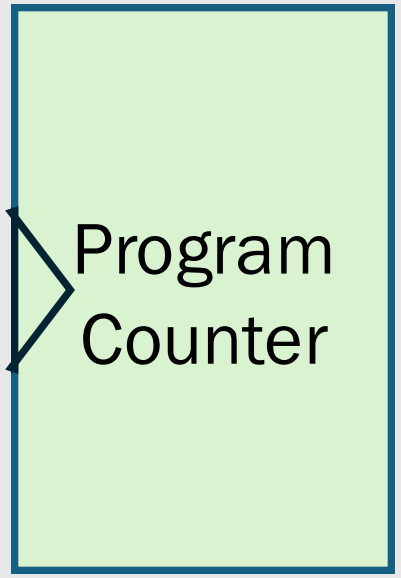
Loads the address of the label into *Reg[rd]*. It is typically implemented using the two instruction sequence:

```
auipc rd, label
addi rd, rd, label
```

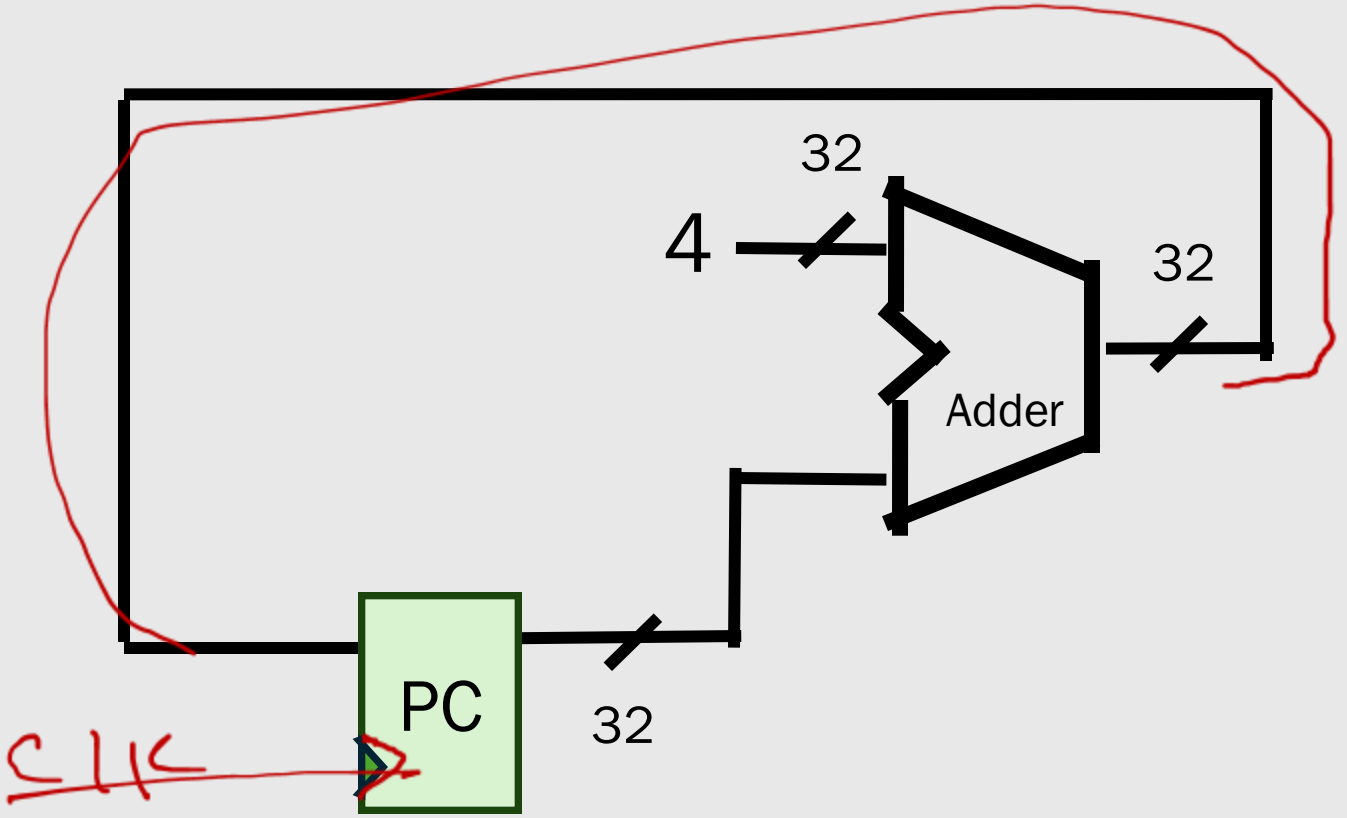
A pseudoinstruction is a "fake" assembly instruction that doesn't exist in HW, but assembler expands into one or more real RISC-V instructions!

Program Counter (PC)

- The Program Counter (PC) is a 32-bit register
- The PC is not inside of the register file!!!
- The datapath executes one instruction per clock cycle
- On every clock cycle, PC is incremented by 4 to fetch the next instruction



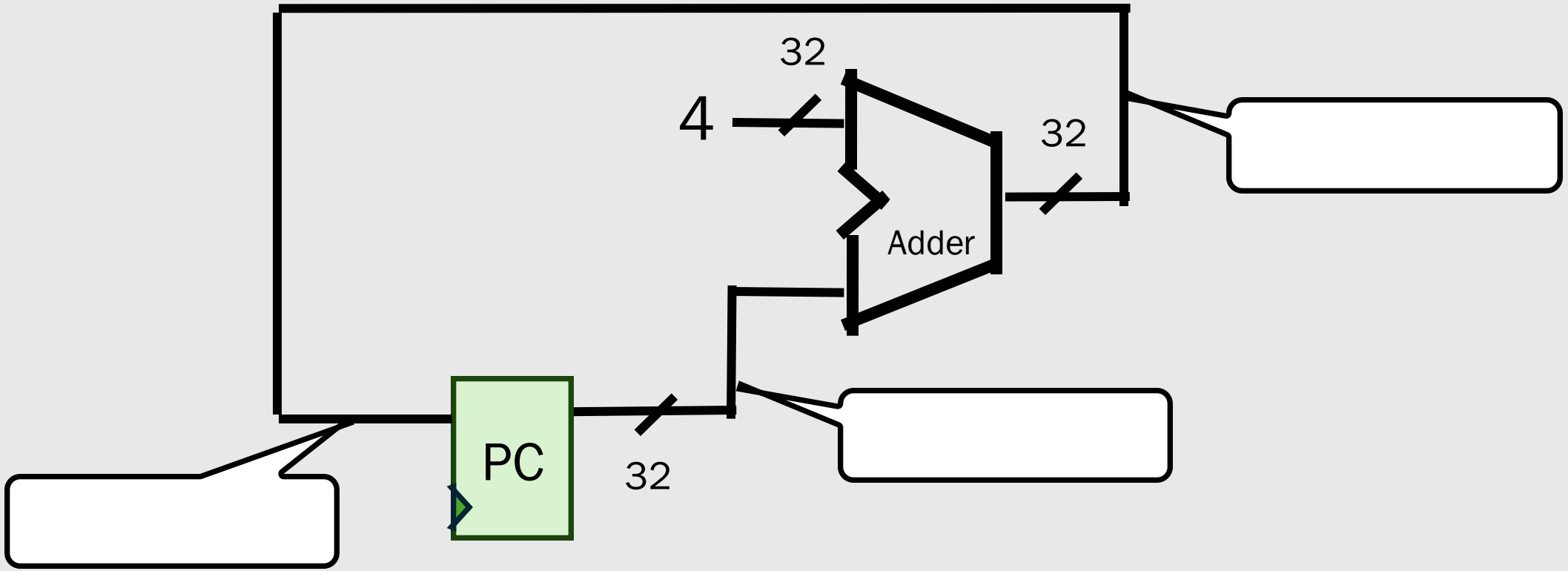
Program Counter (PC)



Increment the PC by 4 on every clock cycle

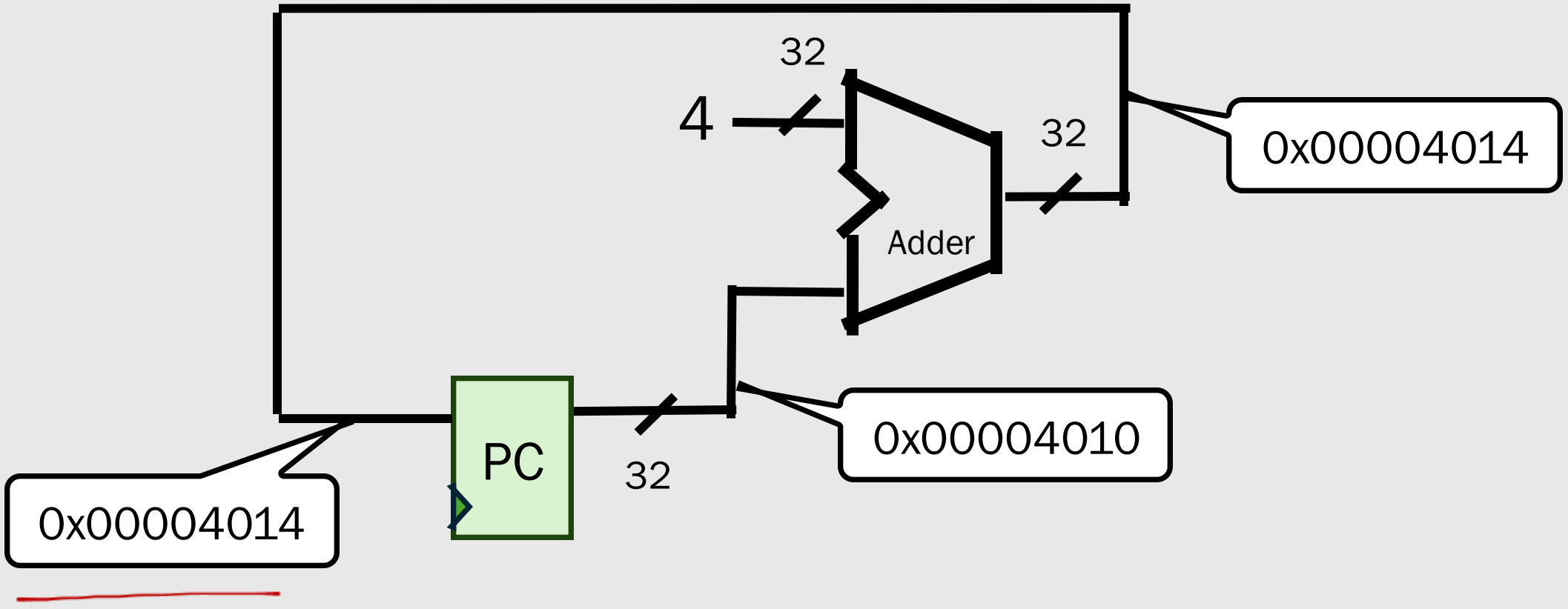
Program Counter (PC)

Example: Current PC = 0x00004010

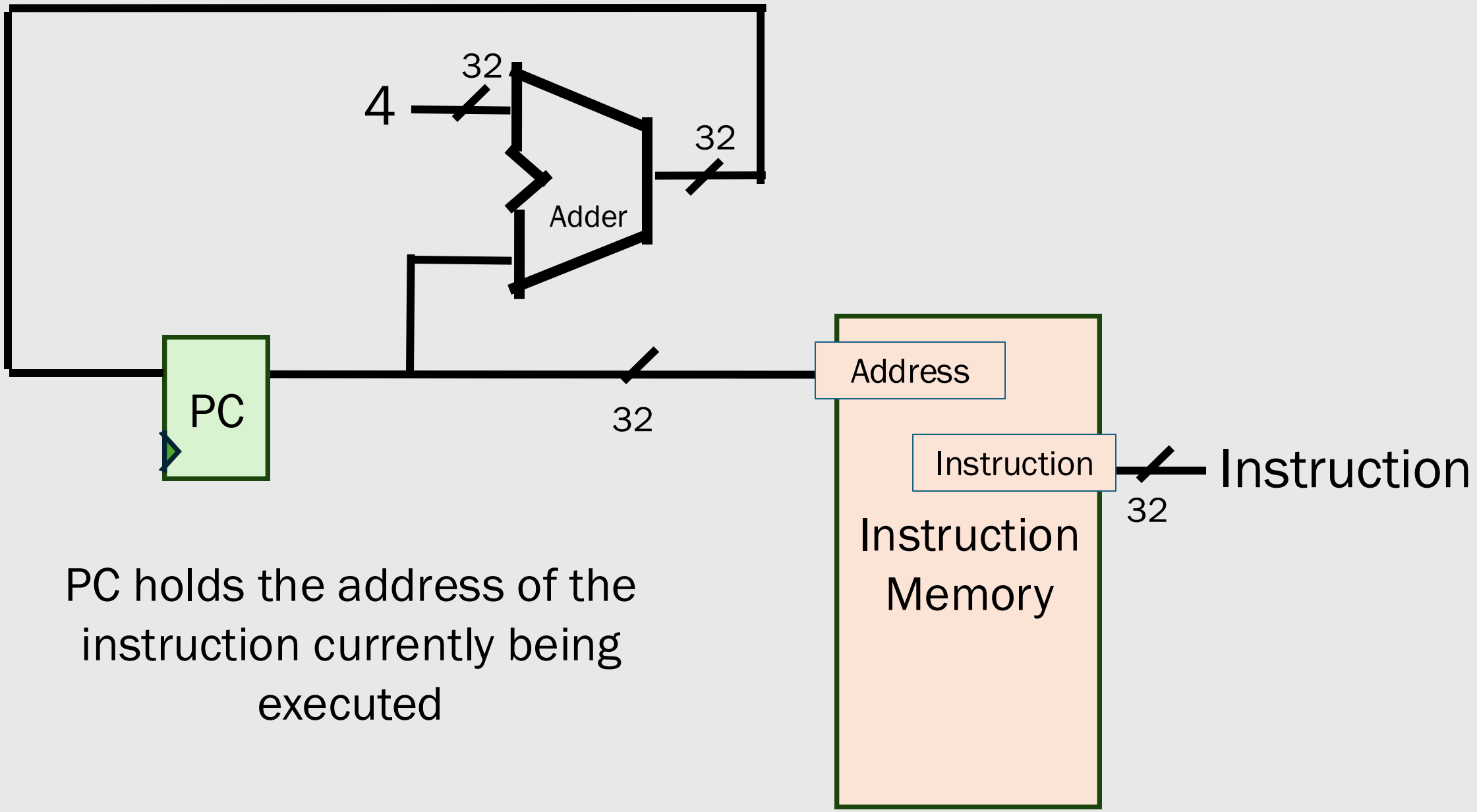


Program Counter (PC)

Example: Current PC = 0x00004010



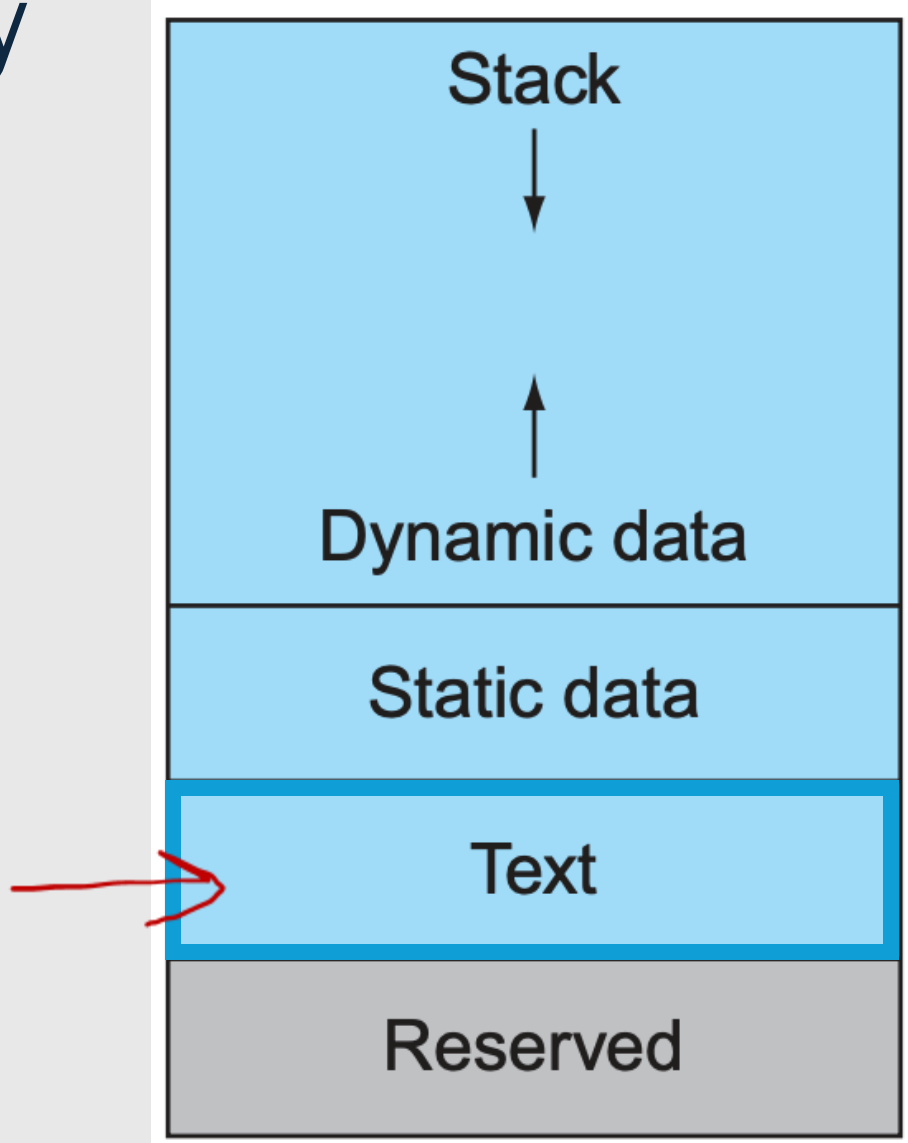
On the next rising edge of the clock, PC will become 0x00004014



PC holds the address of the instruction currently being executed

Storing Code in Memory

- Code is stored in the text segment
- To store code in the text segment, we use the .text directive



Straight-Line Code

At this point all of our standard R-type, I-type, and S-type instructions are implemented.

The resulting machine executes only straight in-line code

Can't get much interesting done this way.

No function calls, for-loops, while-loops, or if-then-else logic

Incorporating A Stack

```
int sqr(int x) {
  if (x > 1)
    x = sqr(x-1)+x+x-1;
  return x;
}


main()
{
  sqr(10);
}
```

```
sqr:    addi    sp,sp,-8
        sw      ra,4(sp)
        sw      s0,0(sp)
        slti    t0,a0,2
        bne     t0,x0,return
        add     s0,x0,a0
        addi    a0,a0,-1
        jal     ra,sqr
        add     a0,a0,s0
        add     a0,a0,s0
        addi    a0,a0,-1
return: lw      s0,0(sp)
        lw      ra,4(sp)
        addi    sp,sp,8
        jalr    x0,ra

main:   addi    sp,sp,-4
        sw      ra,(sp)
        addi    a0,x0,10
        jal     ra,sqr
        lw      ra,(sp)
        addi    sp,sp,4
        jalr    x0,ra
```

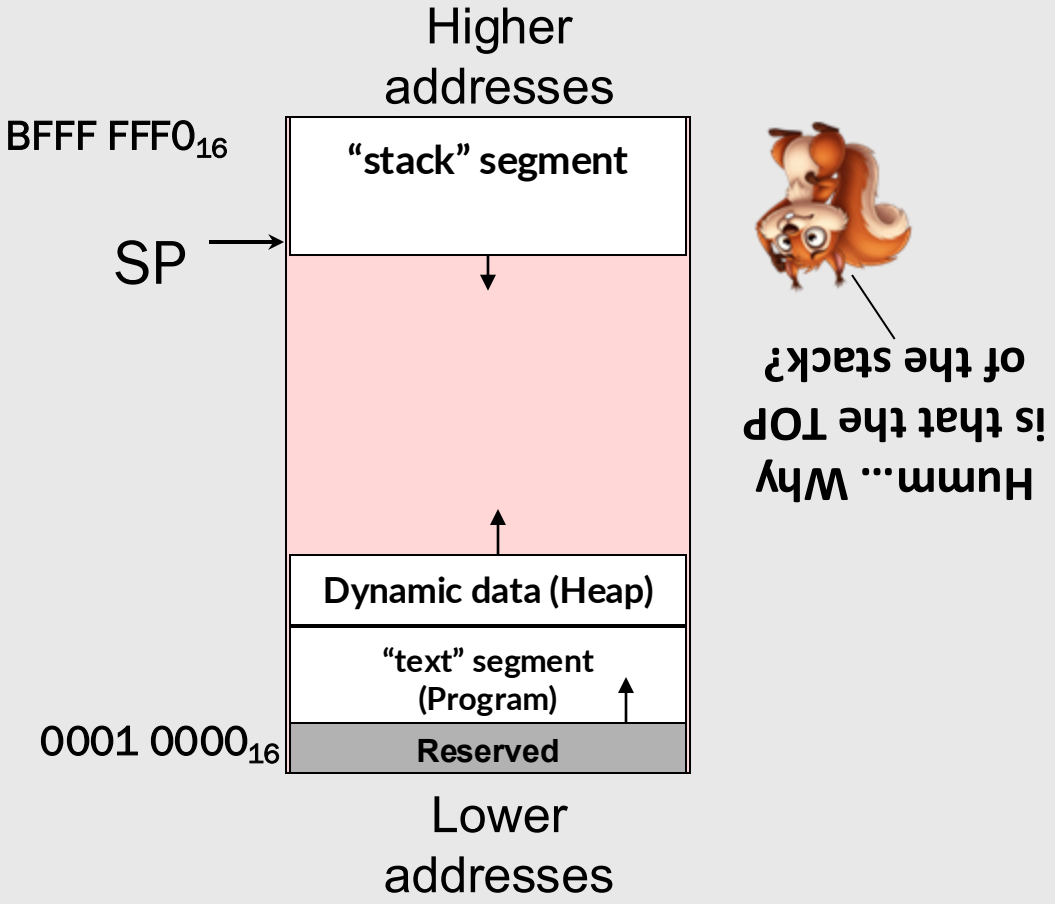
function prologue

function epilogue

RISC-V Stack Convention

CONVENTIONS:

- **Assign a register as the Stack Pointer (sp = x2).**
- **Stack grows DOWN (towards lower addresses) on *pushes* and *allocates***
- ***sp* points to the last or TOP *used* location.**
- **Stack is placed far away from the program and its data.**



Stack Management Policies

ALLOCATE k: reserve k WORDS of stack
SP = SP - 4*k

```
addi    sp,sp,-4*k
```

DEALLOCATE k: release k WORDS of stack
SP = SP + 4*k

```
addi   sp,sp,4*k
```

PUSH x: push Reg[x] onto stack
Mem[SP - 4] = Rx
SP = SP - 4

```
addi   sp,sp,-4
sw     rx,(sp)
```

POP x: pop the top of the stack into Reg[x]
Rx = Mem[SP]
SP = SP + 4

```
lw     rx,(sp)
addi   sp,sp,4
```