# MIPS I-type Instructions

Author: Jared Moore
Edited by Nathan Bowman

# Problem with R-Type

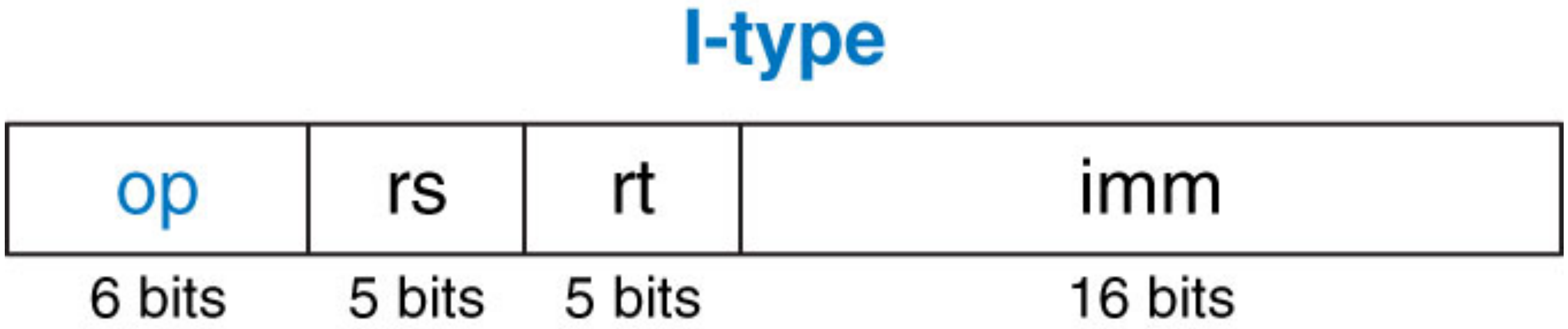
R-type instructions are convenient because they have very regular structure – every R-type instruction can be processed by same hardware we have constructed

However, both R-type source operands must be registers

How can we get value into register in the first place?

# I-Type Instructions

Two register operands and an immediate.



# Immediate

Immediate means hard-coded constant

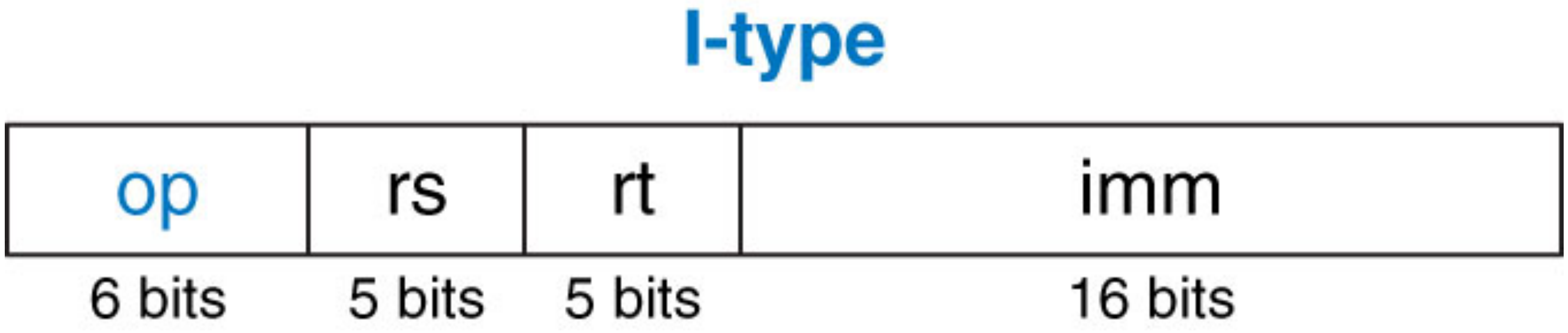
This is value chosen by programmer that is part of assembly program

In high-level instruction “b = a + 5”, 5 would correspond to immediate in assembly instruction

# I-Type Instructions

rs is source register, as before

rt can be second source register or destination register, depending on instruction (note that we have no rd)



# Examples

```
addi $t1, $t0, 5
slti $t2, $t1, 4
addi $t3, $t1, -10
```

No such thing as `subi`. Why?

# Examples

```
addi $t1, $t0, 5
slti $t2, $t1, 4
addi $t3, $t1, -10
```

No such thing as `subi`. Subtraction is just adding negative number. Use negative immediate instead. Subtract immediate would be redundant

# Interpreting immediate

Since we are allowed to use negative, must be storing immediate as (16-bit) two's complement number

This is true for *arithmetic* expressions – logical expressions behave differently, as we will see shortly

Note that value in register is 32 bits, whereas immediate just 16 bits

To convert signed 16-bit number to equivalent 32-bit number, we *sign extend*, as discussed in previous lecture

# Interpreting immediate

MIPS also has logical I-type function, such as

```
andi $t1, $t0, 5
ori $t2, $t1, 4
xori $t3, $t1, 10
```

# Interpreting immediate

```
andi $t1, $t0, 5
```

Two things to note:

- Operation is *bitwise*
- Immediate *cannot* be negative for logical expressions

Immediate treated as unsigned number and zero-extended (add 16 zeros to the left to make it 32-bit number)

# Interpreting immediate

Consider 8-bit example with 4-bit immediate (not actual MIPS)

```
addi $t1, $0, 7   # t1 = 7 (= 00000111b)
andi $t2, $t1, 5  # 5 = 0101b
```

```
    00000111
and     0101
```

becomes

```
    00000111
and 00000101
=   00000101
```

# Interpreting immediate

Important to bear in mind whether you are working with

- arithmetic instruction (with sign-extension), or
- logical instruction (with zero-extension)

# Zero register

Wait, but we said one reason for all this was to get values into registers!

I-type instructions still take one register as arguments, so we've pushed off the problem, but not solved it

Recall the special zero register: `$0` (a.k.a `$zero`)

Value in zero register is *always* 0

# Zero register

You can try

`addi $0 $t1 5`

for example, and value in register does not change from 0

May seem limiting to use register this way, but 0 is needed so often it is worthwhile

How would you set the value in `$t1` equal to 10?

# Zero register

```
addi $t1 $0 10    # sets value in $t1 to 10
```

Get used to this pattern – you will see it *a lot*

Sometimes see shorthand instruction

```
li $t1 10
```

This is same instruction – `li` is not a “real” instruction – but the assembler will convert it to `addi` for you

# I-type instruction

In reality, I-type not just used because we need some way to get initial values into registers

Whenever programmer wants to include constant in program, such as adding 1 in loop, immediate is used

# Opcode

**R-type**

| op | rs | rt | rd | shamt | funct |
|---|---|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 5 bits | 5 bits | 6 bits |

**I-type**

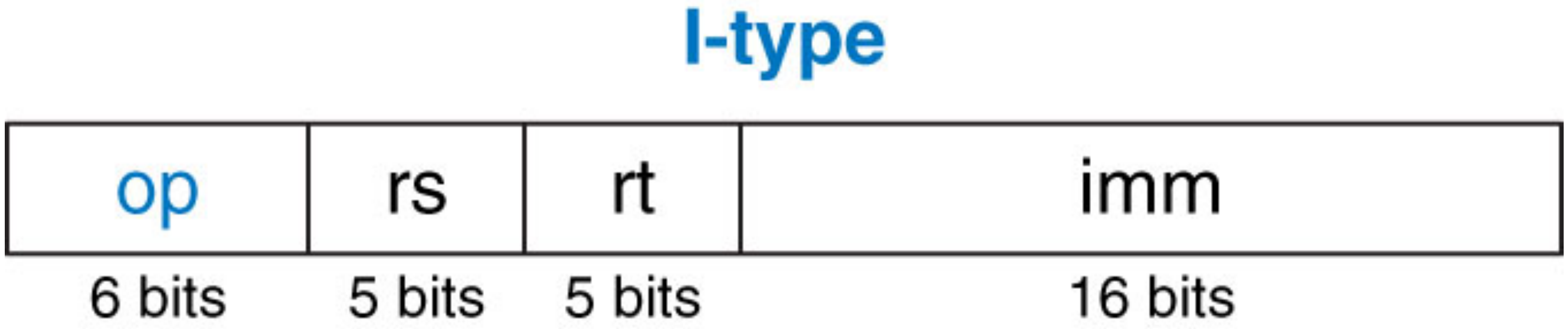
| op | rs | rt | imm |
|---|---|---|---|
| 6 bits | 5 bits | 5 bits | 16 bits |

# Opcode

Every R-type instruction has same opcode: `000000`

R-type instructions use `funct` field to distinguish between different operations, such as `add` or `sub`

I-type instructions have no `funct` field



# Opcode

Opcode for every I-type instruction different

Knowing opcode is enough to uniquely specify particular I-type instruction