5DV118
Computer Organization and Architecture
Umeå University
Department of Computing Science

Stephen J. Hegner

Topic 2: Instructions

Part B: Numbers and Shifting

These slides are mostly taken verbatim, or with minor changes, from those prepared by

Mary Jane Irwin (www.cse.psu.edu/~mji)

of The Pennsylvania State University

[Adapted from *Computer Organization and Design, 4th Edition*,

Patterson & Hennessy, © 2008, MK]

Key to the Slides

- The source of each slide is coded in the footer on the right side:
  - Irwin CSE331 PSU = slide by Mary Jane Irwin from the course CSE331 (Computer Organization and Design) at Pennsylvania State University.
  - Irwin CSE431 PSU = slide by Mary Jane Irwin from the course CSE431 (Computer Architecture) at Pennsylvania State University.
  - Hegner UU = slide by Stephen J. Hegner at Umeå University.

Review: MIPS Arithmetic Instructions

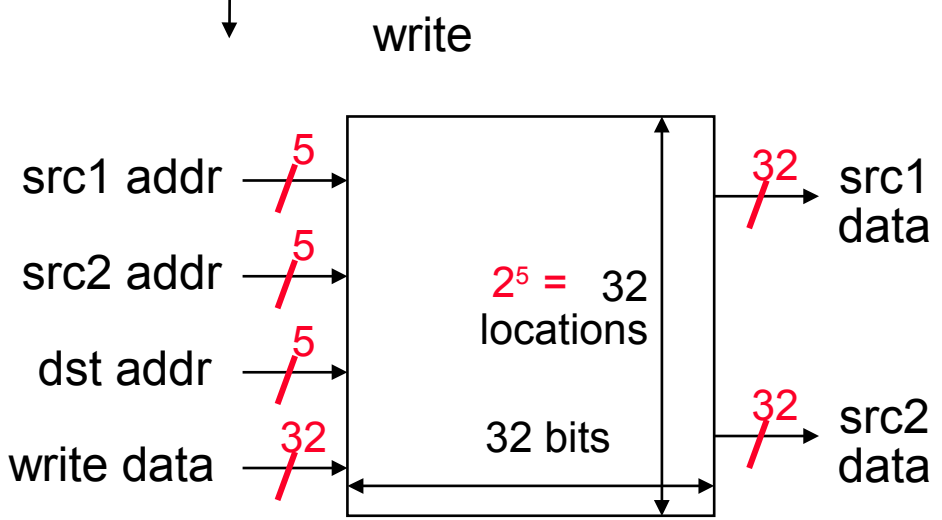
- MIPS assembly language arithmetic statements

dst ← src1 op src2

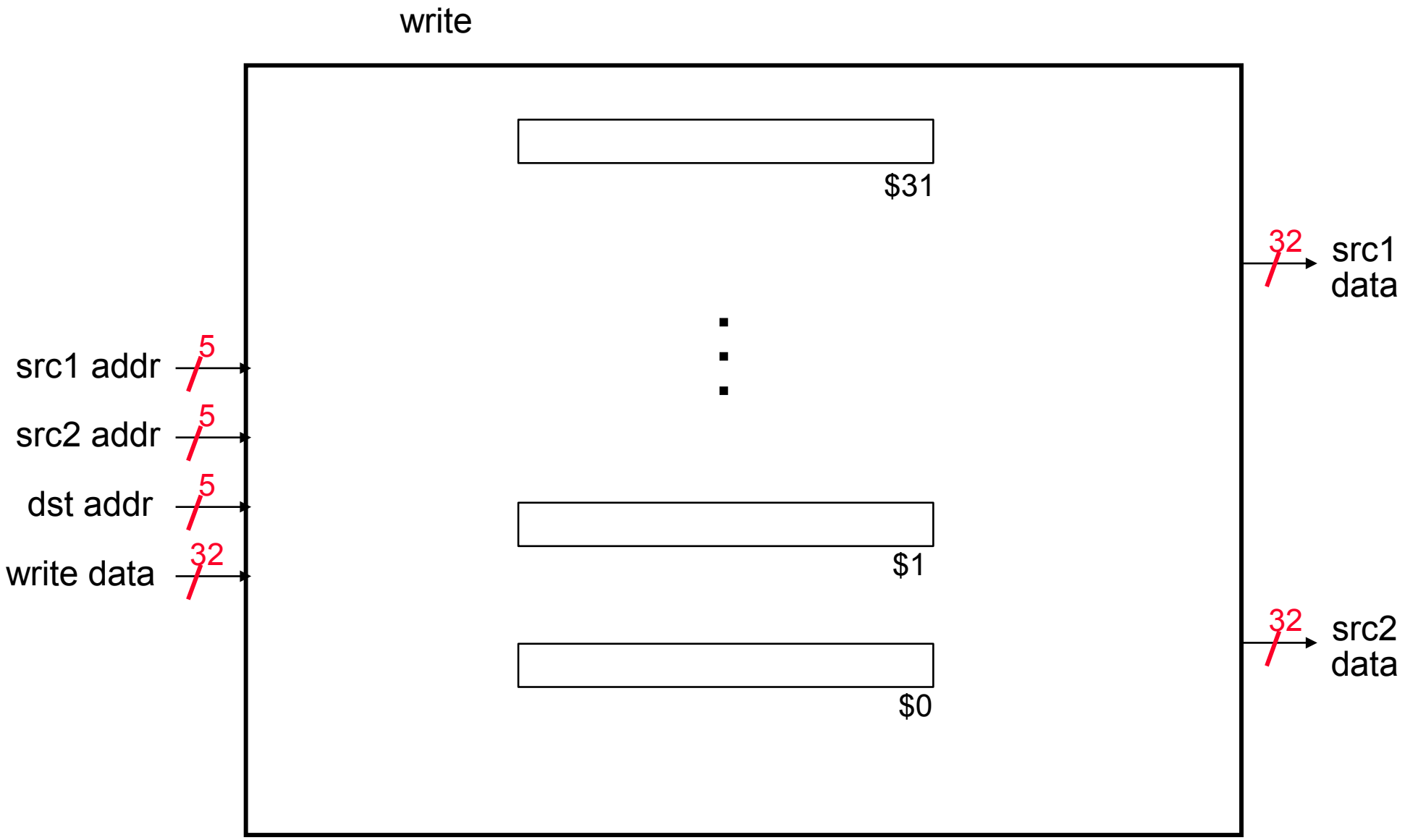
```
add   $t0, $s1, $s2
sub   $t0, $s1, $s2
```

- The operands (`$t0, $s1, $s2`) are contained in the datapath's register file which contains thirty-two 32-bit registers
  - Two read ports
  - One write port

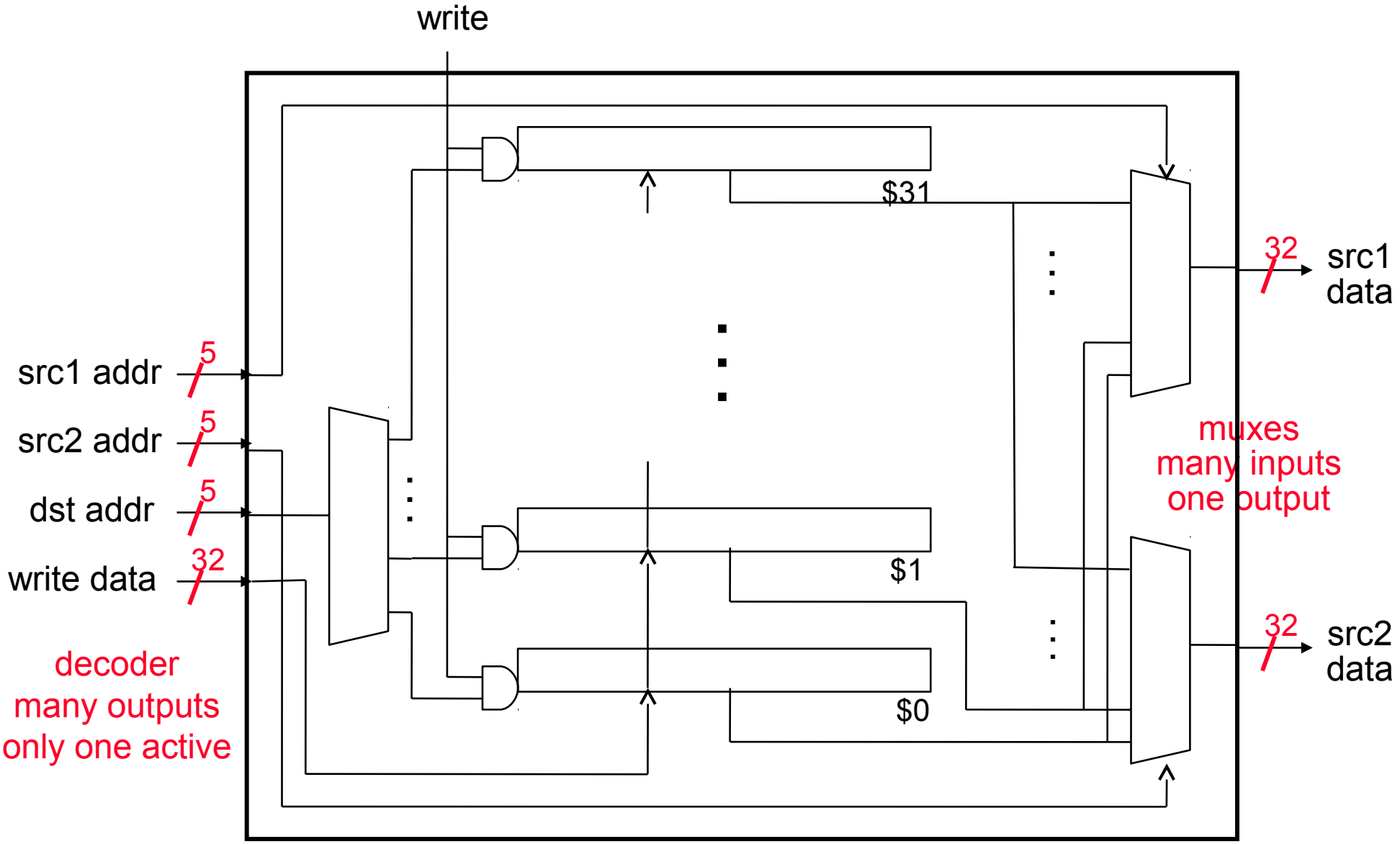
  which takes ~ ½ clock cycle to read from or write to



Inside the Register File



Inside the Register File



Binary Numbers

- Binary digit – bit – can be one of two values, 0 or 1
- To convert from a binary number to decimal just

$$\ldots d_3\ d_2\ d_1\ d_0 = \ldots + d_3 x 2^3 + d_2 x 2^2 + d_1 x 2^1 + d_0 x 2^0$$

- Convert $1\ 1\ 0\ 1\ 1_{two}$ to decimal

- Convert 52_{ten} to binary

Binary Numbers

- Binary digit – bit – can be one of two values, 0 or 1
- To convert from a binary number to decimal just

  $\ldots d_3\ d_2\ d_1\ d_0 = \ldots + d_3x2^3 + d_2x2^2 + d_1x2^1 + d_0x2^0$

- Convert $1\ 1\ 0\ 1\ 1_{two}$ to decimal

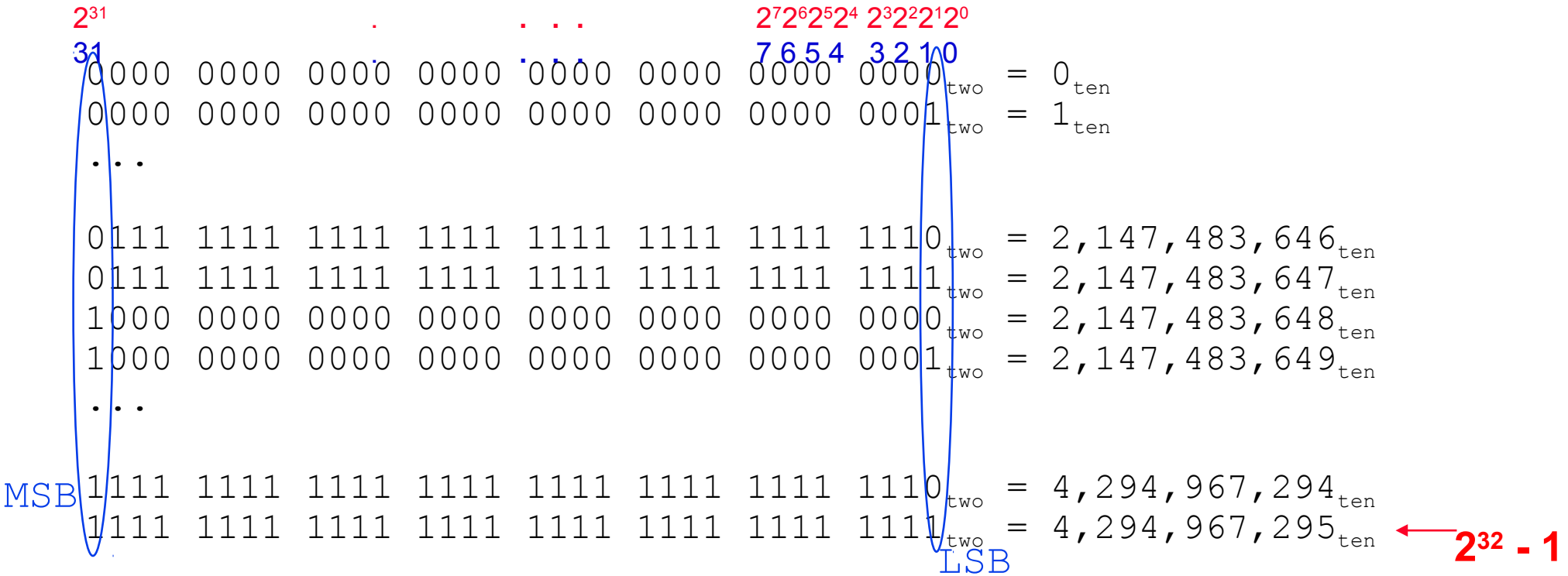
$$1x2^4 + 1x2^3 + 1x2^1 + 1x2^0 = 27_{ten}$$

- Convert 52_{ten} to binary

$$52_{ten} = 1x2^5 + 1x2^4 + 1x2^2 = 1\ 1\ 0\ 1\ 0\ 0_{two}$$

MIPS Unsigned Number Representations

- With 32 bit words MIPS-32 can represent 2^{32} different numbers



- Bits are numbered from right to left in a word (starting with 0)
  - MSB – most significant bit (bit 31)
  - LSB – least significant bit (bit 0)

Representing both Positive and Negative #'s

- Sign magnitude – one sign bit (bit 31) and the rest are magnitude bits (bit 30 to bit 0)

$0\ 0\ldots0\ 1\ 0\ 1_{two} = +5_{ten}$ and $1\ 0\ldots0\ 1\ 0\ 1_{two} = -5_{ten}$

- Two's complement – bit 31 is both a sign *and* a magnitude bit

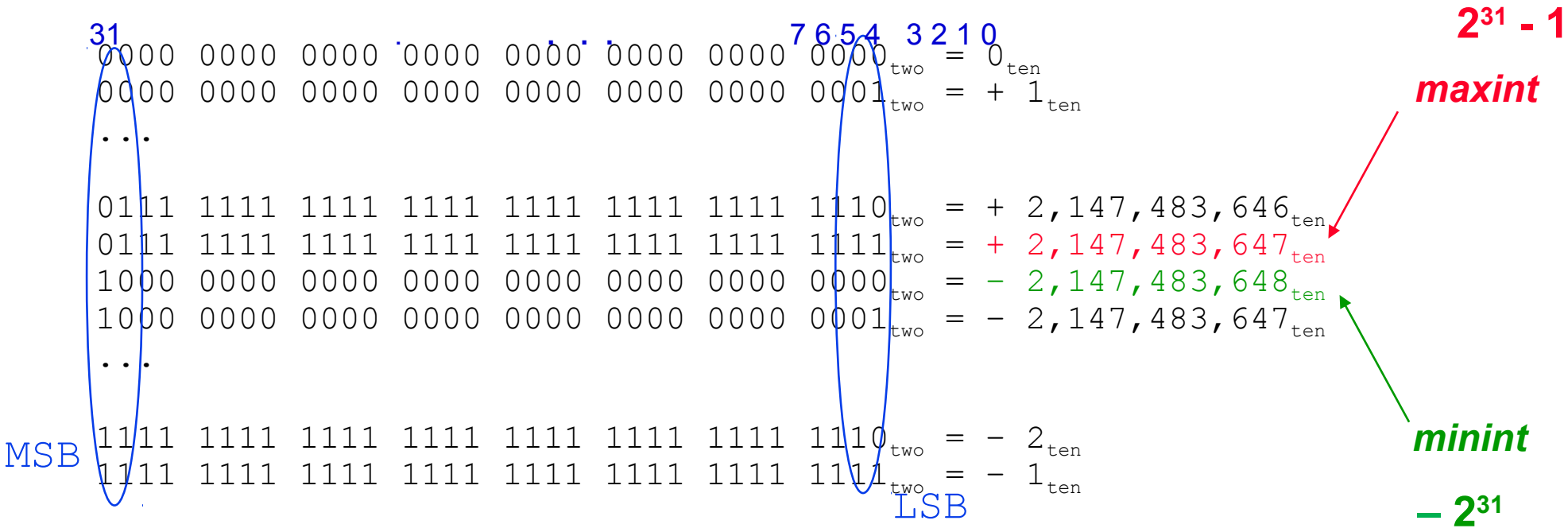
$$\# = -2^{31}d_{31} + \sum_{i=0}^{30} 2^i d_i$$

$0\ 0\ldots0\ 1\ 0\ 1_{two} = +5_{ten}$ and $1\ 1\ldots1\ 0\ 1\ 1_{two} = -5_{ten}$

- Other possibilities – one's complement, biased

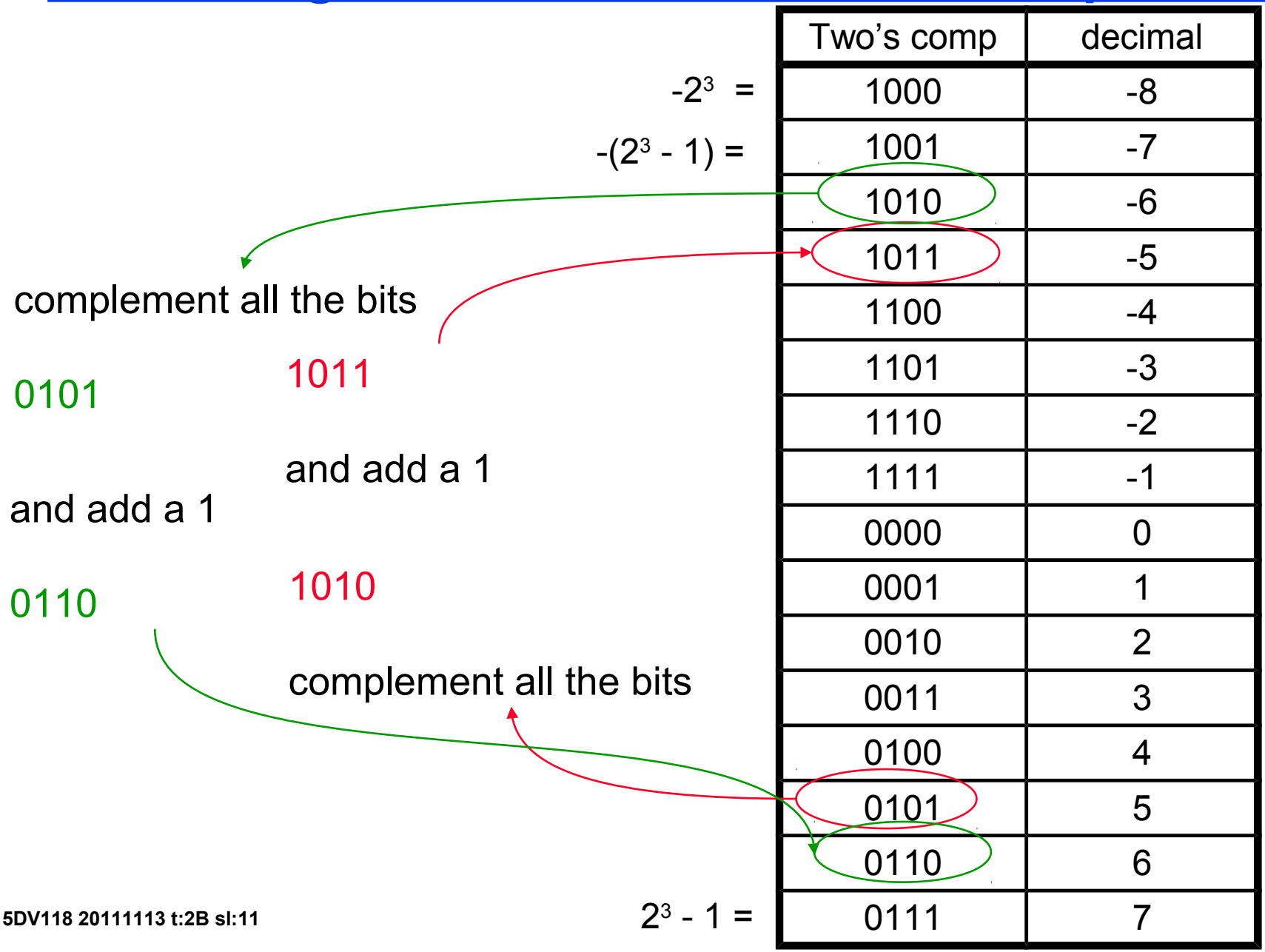
MIPS Number Representations

- MIPS-32 signed numbers (two's complement)



- The all zero bit string is the number 0_{ten}
- Have one (most) negative number that has no corresponding positive number
- All positive numbers have a 0 in the MSB and all negative numbers have a 1 in the MSB – the sign bit

Converting + to – and – to + Two's Complement



Overflow

- Overflow occurs if after conversion, addition, subtraction, or multiplication the number that is the correct result can**not** be represented in 32 bits.
- As simple examples consider 4-bit signed numbers

$$-(-8_{ten}) = 0\ 1\ 1\ 1_{two} + 1_{two} = 1\ 0\ 0\ 0_{two} = -8_{ten}\ ?$$

$$7_{ten} + 6_{ten} = 0\ 1\ 1\ 1_{two} + 0\ 1\ 1\ 0_{two} = 1\ 1\ 0\ 1_{two} = -3_{ten}\ ?$$

$$-8_{ten} + -5_{ten} = 1\ 0\ 0\ 0_{two} + 1\ 0\ 1\ 1_{two} = 0\ 0\ 1\ 1_{two} = +3_{ten}\ ?$$

- Something went wrong. What? Overflow (really need 5 bits to hold the *signed* result).
- Overflow occurs when the leftmost retained bit of the binary bit pattern is not the same as the infinite number of digits (bits) to the left – **the sign bit is incorrect** !

Sign Extension

- Have to do a “sign extension” when converting a < 32-bit value into its 32-bit equivalent
- When might this occur? When preparing to add a 16-bit immediate field value (e.g., in `lw`, `sw`, `addi`) to a 32-bit register value
- Want to retain the numerical *value* of the number, so copy the MSB (the sign bit) into the “empty” bits

```
+2 = 0010   -> 0000 0010 = +2
-6 = 1010   -> 1111 1010 = -6
```

- sign extend versus zero extend

```
+2 = 0010   -> 0000 0010 = +2
-6 = 1010   -> 0000 1010 = +10
```

Hex to Binary and Back

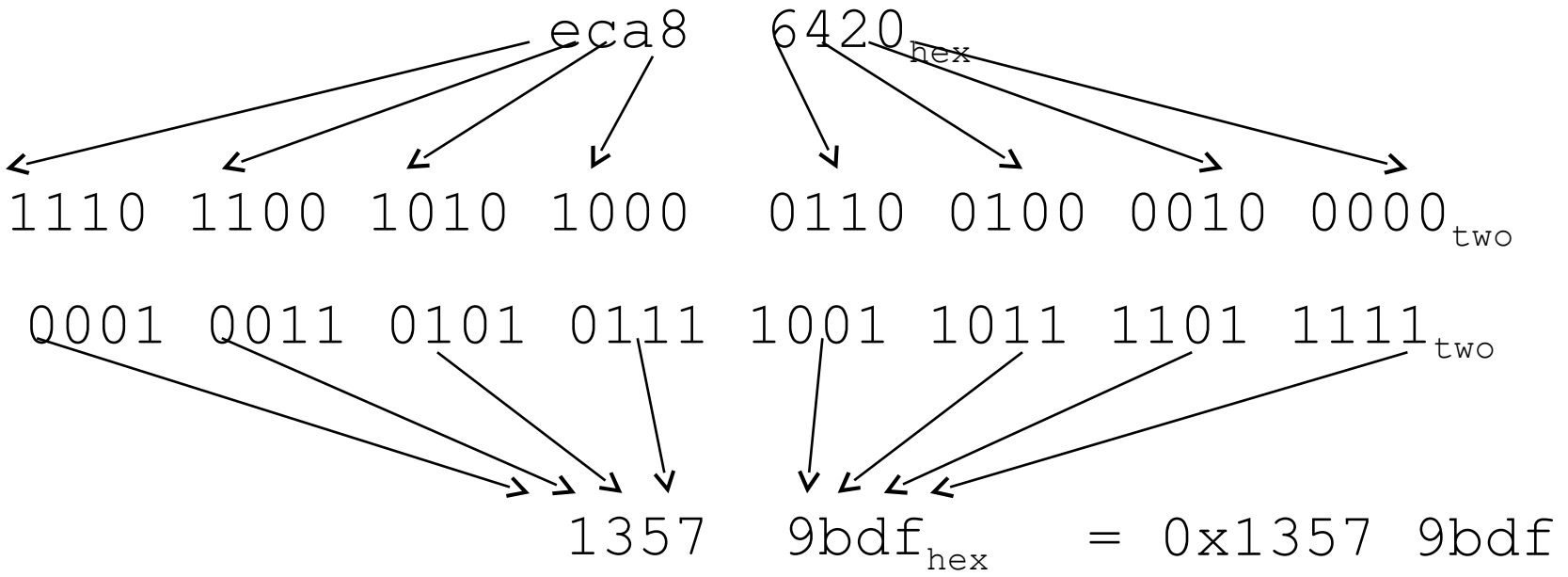
Hex	Binary	Hex	Binary	Hex	Binary	Hex	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

$$eca8 \quad 6420_{hex}$$

$$0001\ 0011\ 0101\ 0111\ 1001\ 1011\ 1101\ 1111_{two}$$

Binary to Hex and Back

Hex	Binary	Hex	Binary	Hex	Binary	Hex	Binary
0_{hex}	0000_{two}	4_{hex}	0100_{two}	8_{hex}	1000_{two}	c_{hex}	1100_{two}
1_{hex}	0001_{two}	5_{hex}	0101_{two}	9_{hex}	1001_{two}	d_{hex}	1101_{two}
2_{hex}	0010_{two}	6_{hex}	0110_{two}	a_{hex}	1010_{two}	e_{hex}	1110_{two}
3_{hex}	0011_{two}	7_{hex}	0111_{two}	b_{hex}	1011_{two}	f_{hex}	1111_{two}

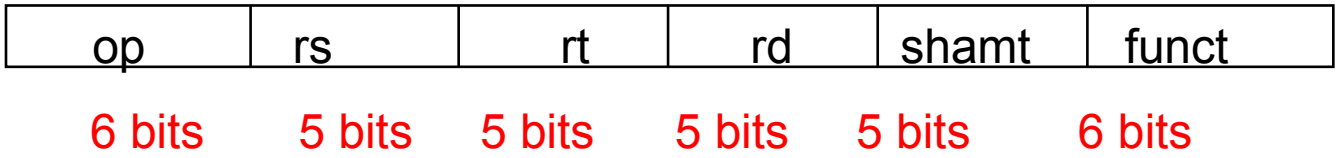


Review: MIPS Register Naming Convention

Nick Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$k0 - $k1	26-27	reserved for OS	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

Representing Instructions in the Machine

- Remember the MIPS-32 instruction fields, R format



- Arithmetic instructions

```
add $t0, $s1, $s2
```

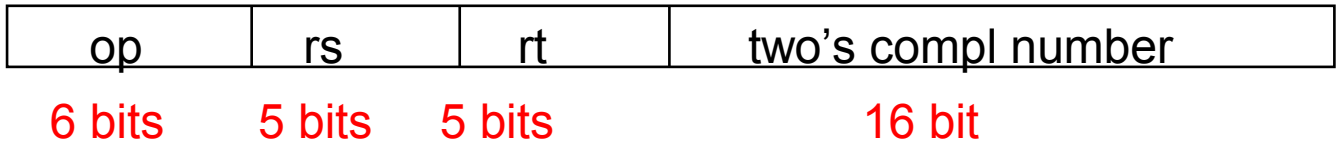
000000	10001	10010	01000	00000	100000
0x00	17	18	8	0	0x20

```
sub $t0, $s1, $s2
```

000000	10001	10010	01000	00000	100010
0x00	17	18	8	0	0x22

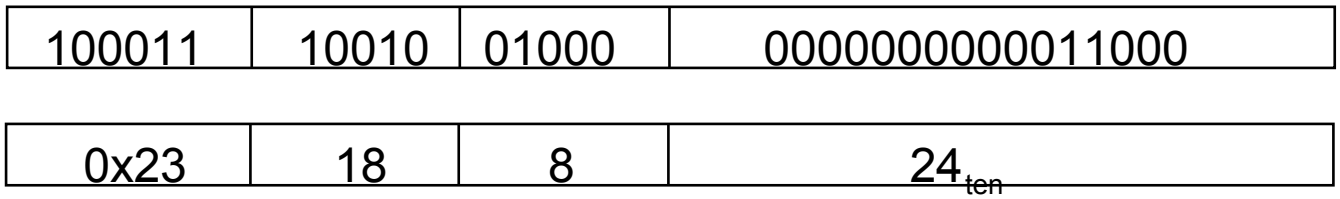
Representing Instructions in the Machine

- Remember the MIPS-32 instruction fields, I format

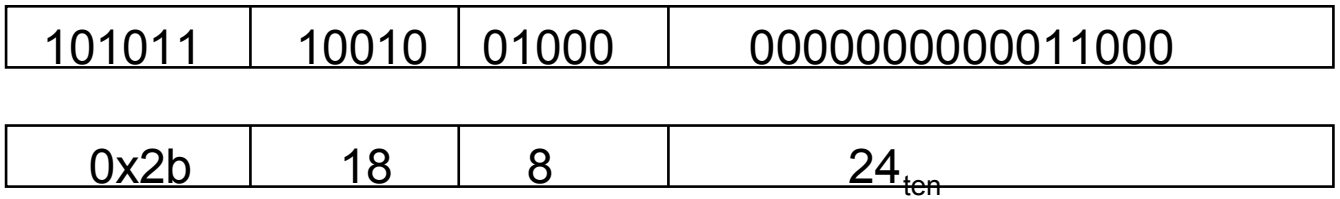


- Data transfer instructions

```
lw $t0, 24($s2)
```



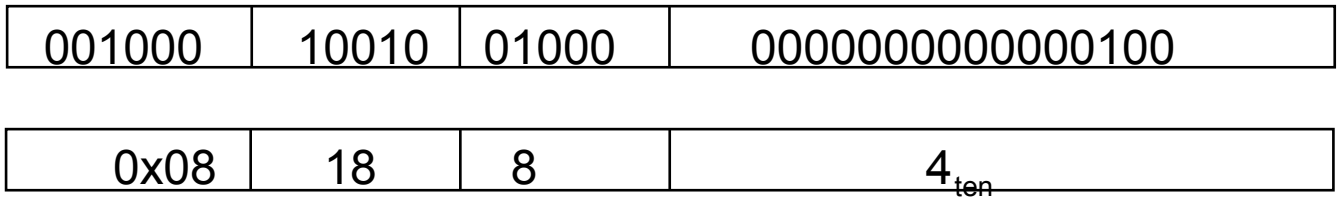

```
sw $t0, 24($s2)
```



Representing Instructions in the Machine

- Immediate instructions

```
addi $t0, $s2, 4
```



- A 16-bit offset means, for loads and stores, that access is limited to memory locations within a range of $+2^{13}-1$ to -2^{13} (~8,192) words ($+2^{15}-1$ to -2^{15} (~32,768) bytes) of the address in the base register
  - two's complement (1 sign bit + 15 magnitude bits)

- And limits immediate values to the range $+2^{15}-1$ to -2^{15}

Assemble these … (in hex)

```
sub  $t0, $s3, $s2
```

```
add  $t0, $s3, $zero
```

```
addi $t0, $s3, -16
```

```
lw $t0, 257($zero)
```

```
sw $t0, -24($s3)
```

Assemble these … (in hex)

```
sub  $t0, $s3, $s2
```

0x00	19	18	8	0	0x22

```
add  $t0, $s3, $zero
```

0x00	19	0	8	0	0x20

```
addi $t0, $s3, -16
```

0x08	19	8	0xfff0	

```
lw $t0, 257($zero)
```

0x23	0	8	0x0101	

```
sw $t0, -24($s3)
```

0x2b	19	8	0xffe8	

Operating on Fields of Bits in a Word

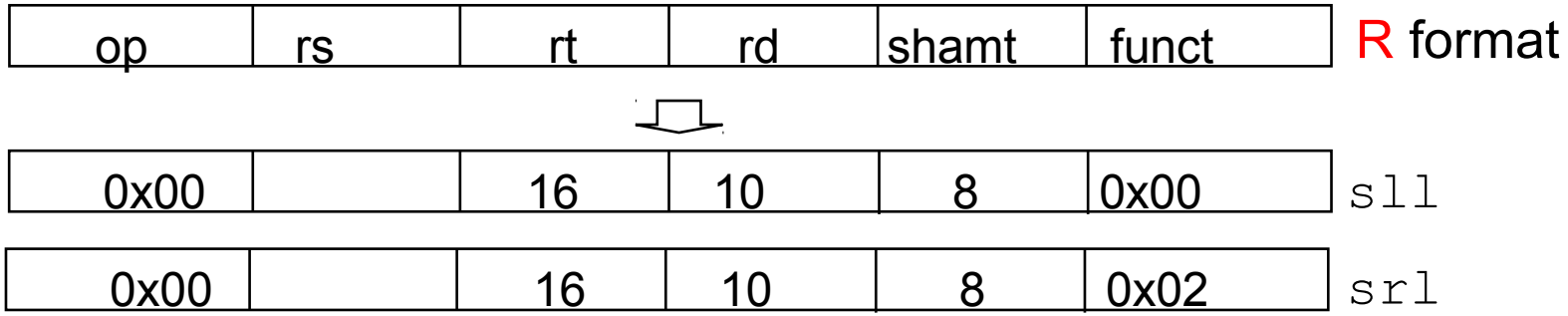
- It is useful to be able to operate on fields or bits within a word or even on individual bits
  - Is the word even or odd ?
  - What is the value of the second byte of the word ?
  - Counting the number of one's in a word
  - Checking to see if the ASCII character for CR (carriage return) exists within a word
- For this we need to have
  - Operations which can isolate a bit or set of bits within a word
    - e.g., zero out all of the bits except the LSB and then look to see if the resulting value is 0 (even) or 1 (odd)
  - Operations which can shift the bit(s) of interest to one end of the word (packing and unpacking)
    - e.g., zero out all of the bits except in the second byte then shift that byte to the far right of the word

MIPS-32 Shift Operations

- Shifts move all the bits in a word left or right by a specified amount

```
sll $t2, $s0, 8     #$t2 = $s0 << 8 bits
srl $t2, $s0, 8     #$t2 = $s0 >> 8 bits
```

- Instruction Format (R format)

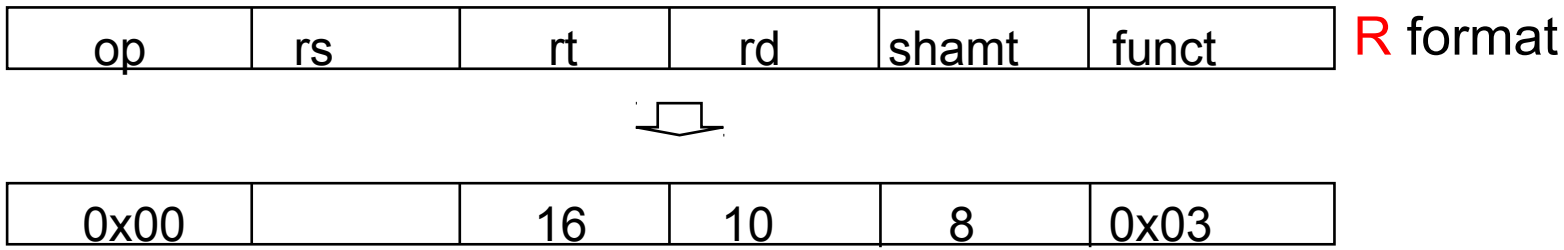


- Such shifts are called logical (notice the trailing `l` in the op mnemonic) because they fill with zeros
- The 5-bit shamt field is just large enough to specify a value which can shift a 32-bit value 31 bit positions

One More Shift Operation

- An arithmetic shift (`sra`) must maintain the arithmetic correctness of the shifted value (i.e., a number shifted right one bit should be ½ of its original value; a number shifted left one bit should be 2 times its original value)
  - `sra` copies the MSB bit (sign bit) as the bit shifted in
  - `srl` shifts in zeros to the MSB
  - `sll` shifts in zeros to the LSB so it also works for arithmetic left shifts for two's complement (so there is no need for a `sla`)

```
sra $t2, $s0, 8    #$t2 = $s0 >> 8 bits
```



Shift Examples and Decimal Equivalents

- Consider shifting the value 6 = 00 … 00110_{two}
  - One bit to the left (so `sll`)
  - One bit to the right, arithmetic (so `sra`)
  - One bit to the right, logical (so `srl`)
- Now consider shifting the value -6 = 11 …11010_{two}
  - One bit to the left (so `sll`)
  - One bit to the right, arithmetic (so `sra`)
  - One bit to the right, logical (so `srl`)

Shift Examples and Decimal Equivalents

- Consider shifting the value 6_{ten} = 00 … 0 0110_{two}
  - One bit to the left (so `sll`)

    $00 \ldots 0\ 1100_{two}$ $= 12_{ten}$
  - One bit to the right, arithmetic (so `sra`)

    $00 \ldots 0\ 0011_{two}$ $= 3_{ten}$
  - One bit to the right, logical (so `srl`)

    $00 \ldots 0\ 0011_{two}$ $= 3_{ten}$
- Now consider shifting the value -6 = 11 …1 1010_{two}
  - One bit to the left (so `sll`)

    $11 \ldots 1\ 0100_{two}$ $= -12_{ten}$
  - One bit to the right, arithmetic (so `sra`)

    $11 \ldots 1\ 1101_{two}$ $= -3_{ten}$
  - One bit to the right, logical (so `srl`)

    $01 \ldots 1\ 1101_{two}$

MIPS Logical Operations

- There are also a number of bit-wise logical operations in the MIPS-32 ISA

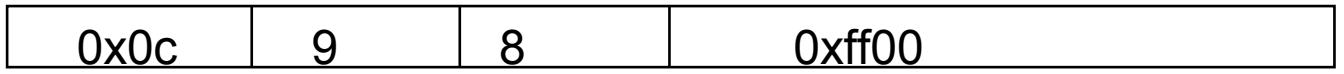
```
and $t0, $t1, $t2  #$t0 = $t1 & $t2
or  $t0, $t1, $t2  #$t0 = $t1 | $t2
nor $t0, $t1, $t2  #$t0 = ~($t1 | $t2)
```

- Instruction Format (R format)

and

```
andi $t0, $t1, 0xff00   #$t0 = $t1 & ff00
ori  $t0, $t1, 0xff00   #$t0 = $t1 | ff00
```

- Instruction Format (I format)



andi

Logical Operations in Action

- Logical operations operate on individual bits of the operand.

$t2 = 0…0 0000 1101 0000

$t1 = 0…0 0011 1100 0000

```
and   $t0, $t1, $t2    $t0 =

or    $t0, $t1 $t2     $t0 =

nor   $t0, $t1, $t2    $t0 =

xor   $t0, $t1, $t2    $t0 =
```

Logic Operations

- Logic operations operate on individual bits of the operand.

$t2 = 0…0 0000 1101 0000

$t1 = 0…0 0011 1100 0000

```
and   $t0, $t1, $t2
```
$t0 =

0…0 0000 1100 0000

```
or    $t0, $t1 $t2
```
$t0 =

0…0 0011 1101 0000

```
nor   $t0, $t1, $t2
```
$t0 =

1…1 1100 0010 1111

```
xor   $t0, $t1, $t2
```
$t0 =

0…0 0011 0001 0000

Uses of Logical Operations

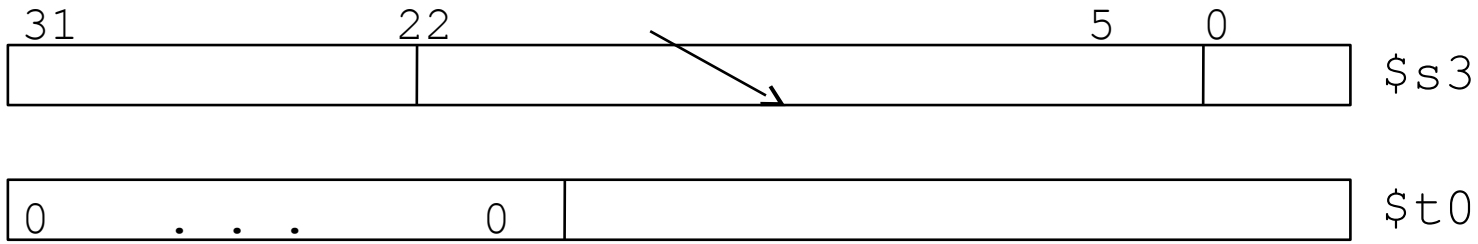
- `and` can apply a bit pattern to a set of bits to force zeros where there is a 0 in the bit pattern. The bit pattern is called a mask, since it "conceals" the bits it is zeroing out.
- `Nor` is often used to invert the bits of a single operand

```
nor $t1, $t1, $zero
```

- Along with `sll` and `slr`, and `and` and `or` are used to insert and extract sub-fields within a 32-bit word
- The full instruction set also include `xor`

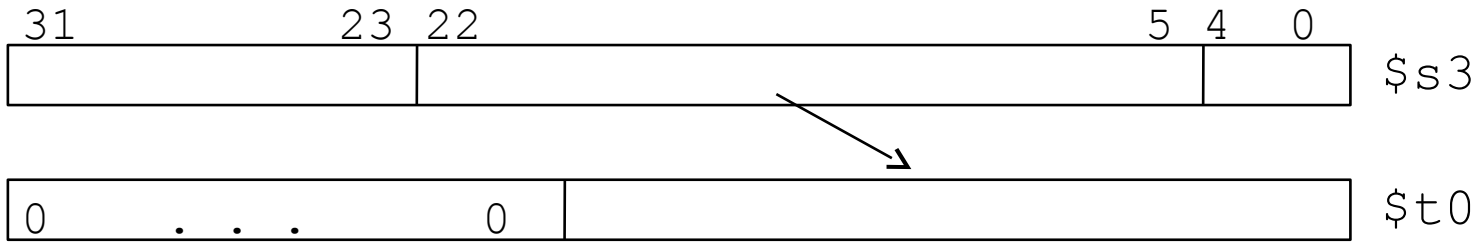
Coding Practice #1

- Give the shortest sequence of MIPS instructions that can extract the bits in $s3 - from bit location 5 to bit location 22 - and place them in register $t0



Coding Practice #1

- Give the shortest sequence of MIPS instructions that can extract the bits in $s3 - from bit location 5 to bit location 22 - and place them in register $t0




```
Snip:   sll  $t0, $s3, 9
        srl  $t0, $t0, 14
```

Coding Practice #2

- Write the MIPS code loop that counts the number of bits that are 1 in $s3 and leaves that count in $t1

Coding Practice #2

- Write the MIPS code loop that counts the number of bits that are 1 in $s3 and leaves the count in $t1

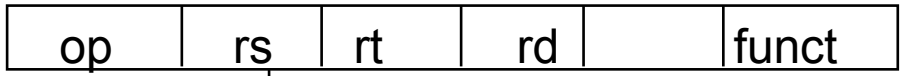
```
        addi $t3, $zero, 32
        add  $t2, $zero, $zero
        add  $t1, $zero, $zero
Loop:   add  $t0, $s3, $zero
        sll  $t0, $t0, 31
        beq  $t0, $zero, Even
        addi $t1, $t1, 1
Even:   addi $t2, $t2, 1
        beq  $t2, $t3, Exit
        srl  $s3, $s3, 1
        j    Loop
Exit:   . . .
```

Review: MIPS Instructions, so far

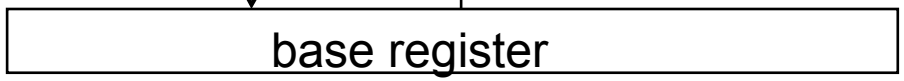
Category	Instr	OpC	Example	Meaning
Arithmetic (R & I format)	add	0 & 20	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 & 22	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	08	addi $s1, $s2, 4	$s1 = $s2 + 4
	shift left logical	0 & 00	sll $s1, $s2, 4	$s1 = $s2 << 4 (fill with 0's)
	shift right logical	0 & 02	srl $s1, $s2, 4	$s1 = $s2 >> 4 (fill with 0's)
	shift right arithmetic	0 & 03	sra $s1, $s2, 4	$s1 = $s2 >> 4 (fill with sign bit)
	and	0 & 24	and $s1, $s2, $s3	$s1 = $s2 & $s3
	or	0 & 25	or $s1, $s2, $s3	$s1 = $s2 | $s3
	nor	0 & 27	nor $s1, $s2, $s3	$s1 = not ($s2 | $s3)
	and immediate	0c	and $s1, $s2, ff00	$s1 = $s2 & 0xff00
	or immediate	0d	or $s1, $s2, ff00	$s1 = $s2 | 0xff00
Data transfer (I format)	load word	23	lw $s1, 100($s2)	$s1 = Memory($s2+100)
	store word	2b	sw $s1, 100($s2)	Memory($s2+100) = $s1

Review: Addressing Modes, so far

1. Register addressing



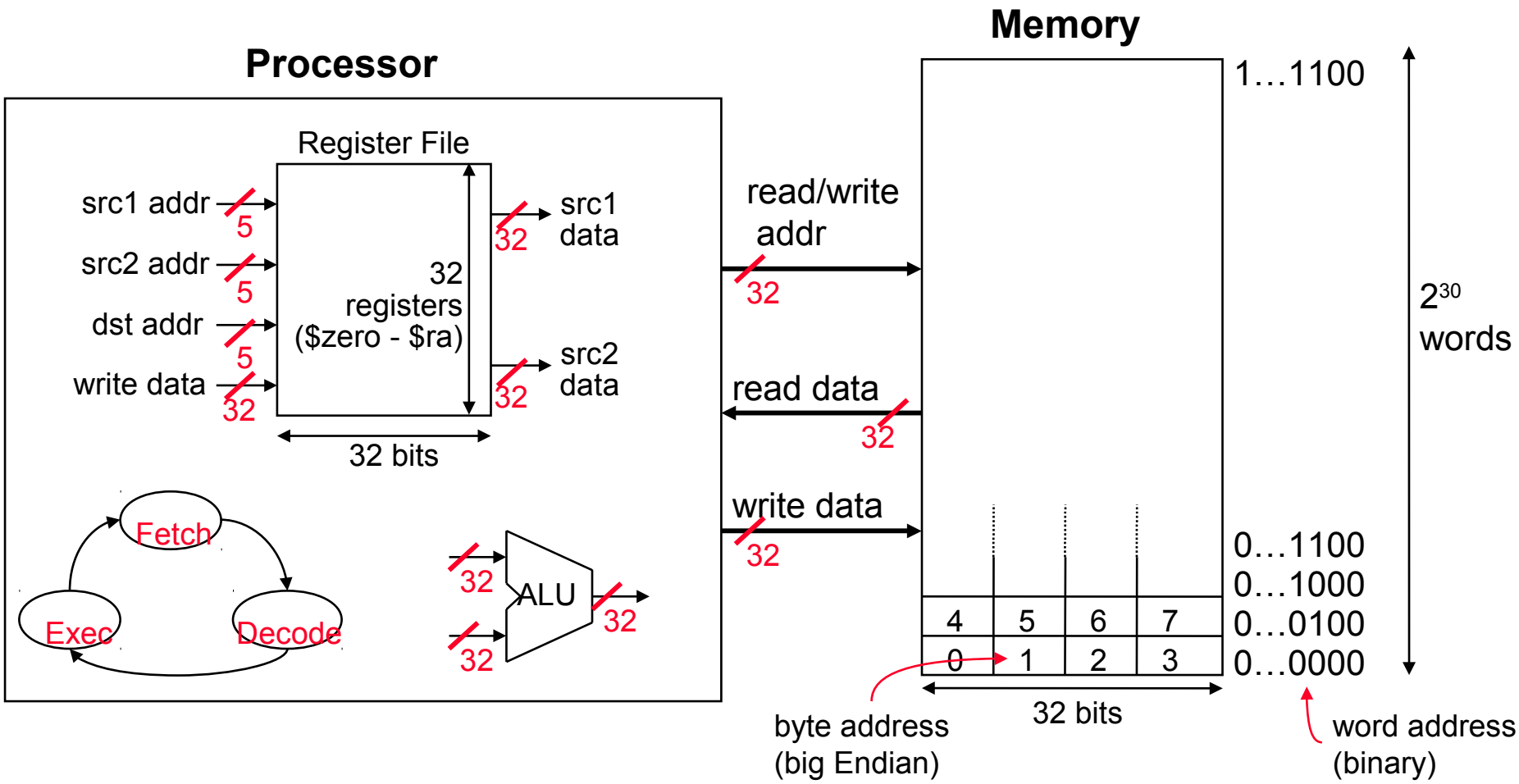
2. Base (displacement) addressing



3. Immediate addressing

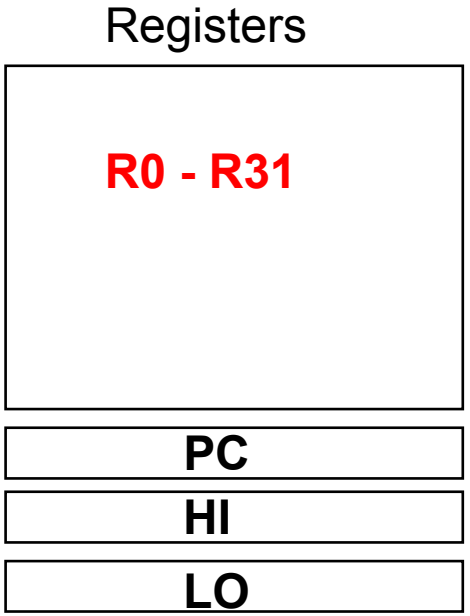
Review: MIPS Organization

- Arithmetic instructions – to/from the register file
- Load/store instructions – from/to memory



MIPS-32 ISA

- Instruction Categories
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide