EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 6_3: MIPS ISA

Xiaolin Xu
Department of ECE
Northeastern University

Operands: Registers

- Registers:
  - `$` before name
  - Example: `$0`, "register zero", "dollar zero"
- Registers used for specific purposes:
  - `$0` always holds the constant value 0.
  - the *saved registers*, $s0-$s7, used to hold variables
  - the *temporary registers*, `$t0 - $t9,` used to hold intermediate values during a larger computation
  - Discuss others later

MIPS Register Set

Name	Register Number	Usage
$0	0	the constant value 0
$at	1	assembler temporary
$v0-$v1	2-3	Function return values
$a0-$a3	4-7	Function arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved variables
$t8-$t9	24-25	more temporaries
$k0-$k1	26-27	OS temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	Function return address

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c
add ___, ____, $s2
```

Instructions with Registers

- Revisit add instruction

C Code

```
a = b + c
```

MIPS assembly code

```
# $s0 = a, $s1 = b, $s2 = c
add $s0, $s1, $s2
```

From High-level Code to Assembly Language

C Code

```
a = b + c;
f= (g + h) - (i + j)
```

MIPS assembly code

```
# assuming that a-to-c are stored in $s0-
   to-$s2, f-to—j are stored in $s3-to-$s7

# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 =
   g, $s5 = h # $s6 = i, $s7 = j

How to convert the C code to assembly code?
```

```
_____________________ #a=b-c
______________________#$t0=g+h
______________________#$t1=i+j
______________________#f=(g+h)-(i+j)
```

From High-level Code to Assembly Language

C Code

```
a = b + c;
f= (g + h) - (i + j)
```

MIPS assembly code

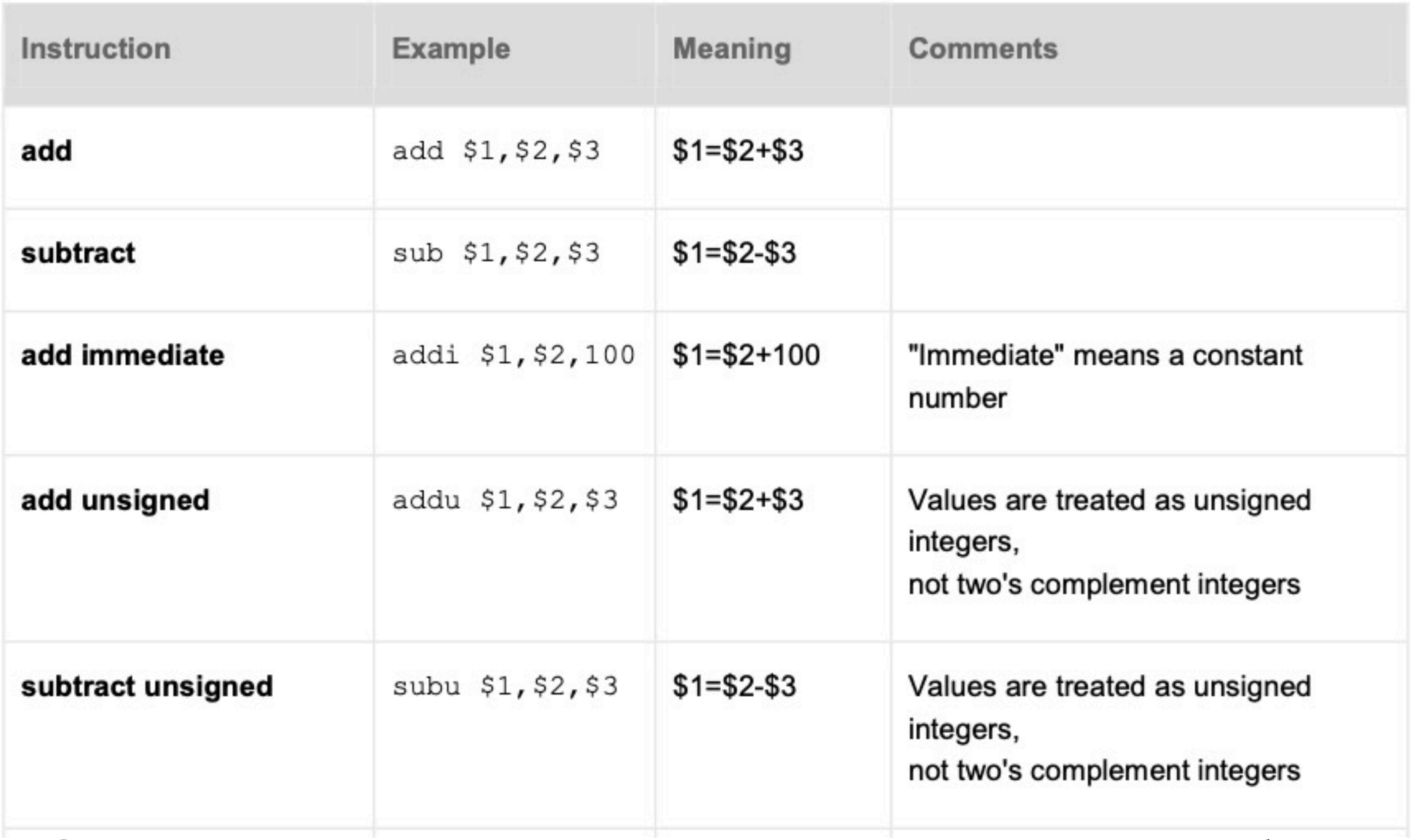
```
# assuming that a-to-c are stored in $s0-
   to-$s2, f-to–j are stored in $s3-to-$s7

# $s0 = a, $s1 = b, $s2 = c, $s3 = f, $s4 =
   g, $s5 = h # $s6 = i, $s7 = j
```

How to convert the C code to assembly code?

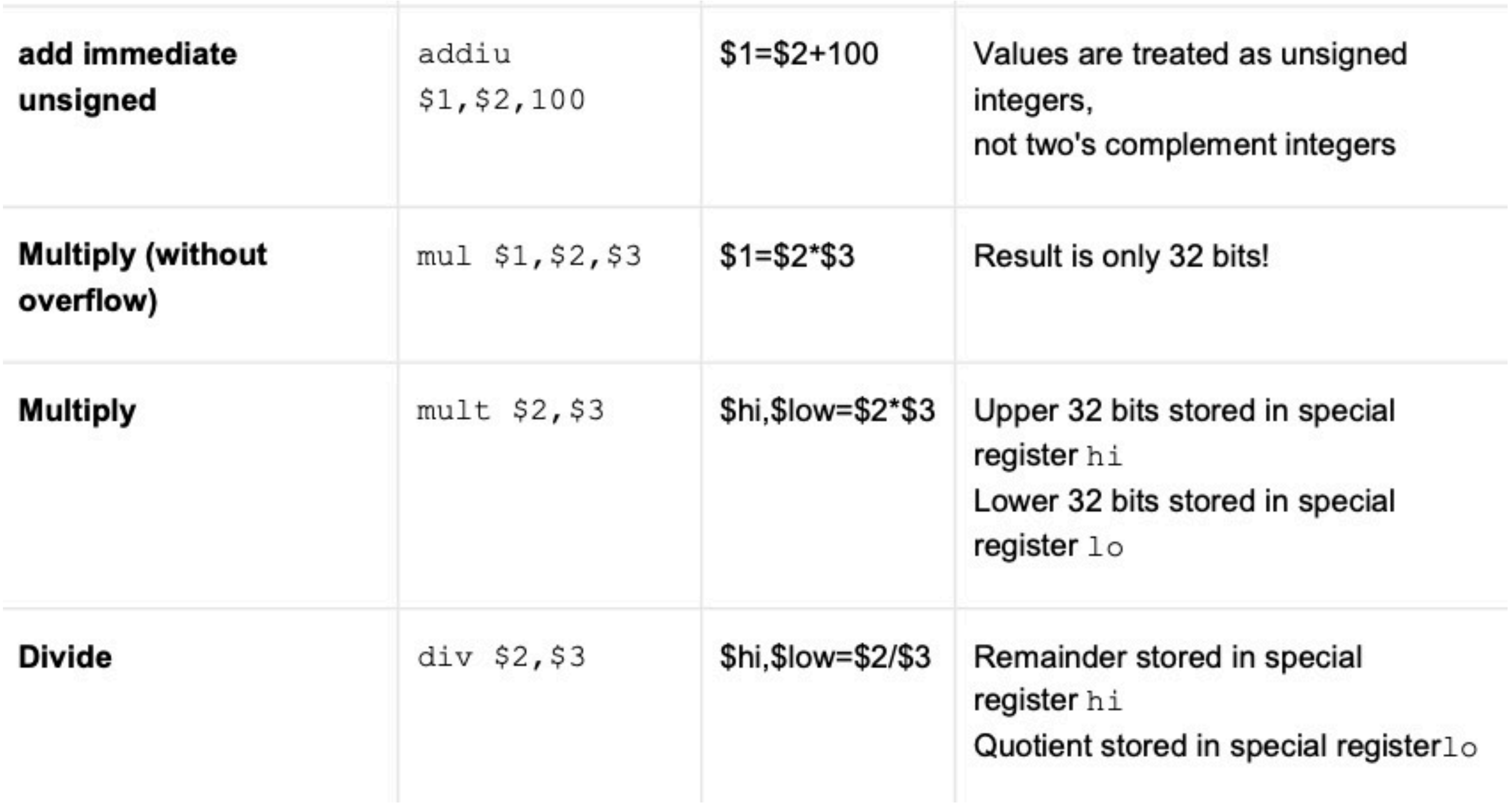
```
sub$s0,$s1,$s2 #a=b-c               ___________________ #a=b-c
add$t0,$s4,$s5 #$t0=g+h             ____________________#$t0=g+h
add$t1,$s6,$s7 #$t1=i+j             ____________________#$t1=i+j
sub$s3,$t0,$t1 #f=(g+h)-(i+j)       ____________________#f=(g+h)-(i+j)
```

MIPS Instructions: Arithmetic



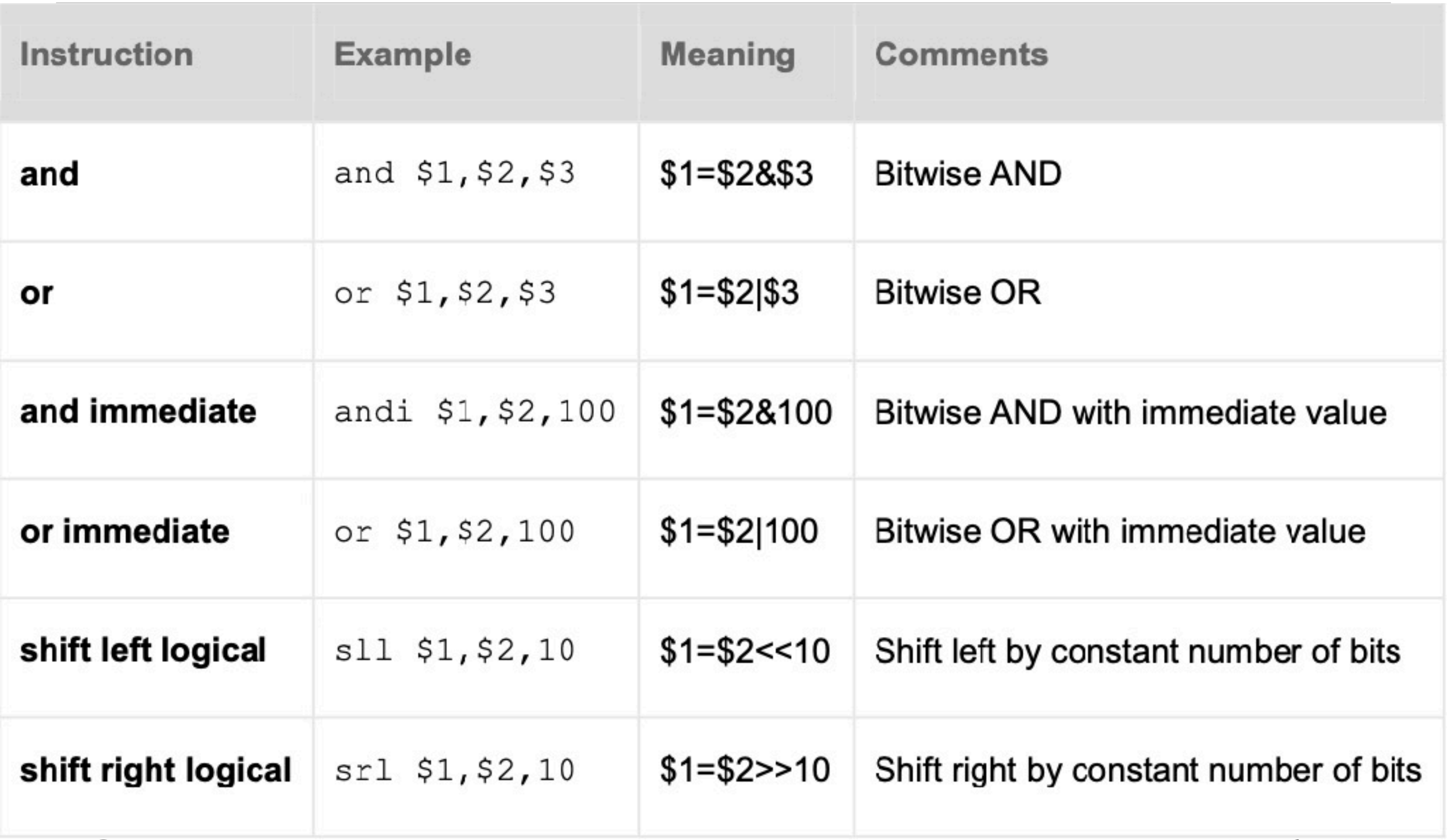
Instruction	Example	Meaning	Comments
add	`add $1,$2,$3`	$1=$2+$3	
subtract	`sub $1,$2,$3`	$1=$2-$3	
add immediate	`addi $1,$2,100`	$1=$2+100	"Immediate" means a constant number
add unsigned	`addu $1,$2,$3`	$1=$2+$3	Values are treated as unsigned integers, not two's complement integers
subtract unsigned	`subu $1,$2,$3`	$1=$2-$3	Values are treated as unsigned integers, not two's complement integers

MIPS Instructions: Arithmetic



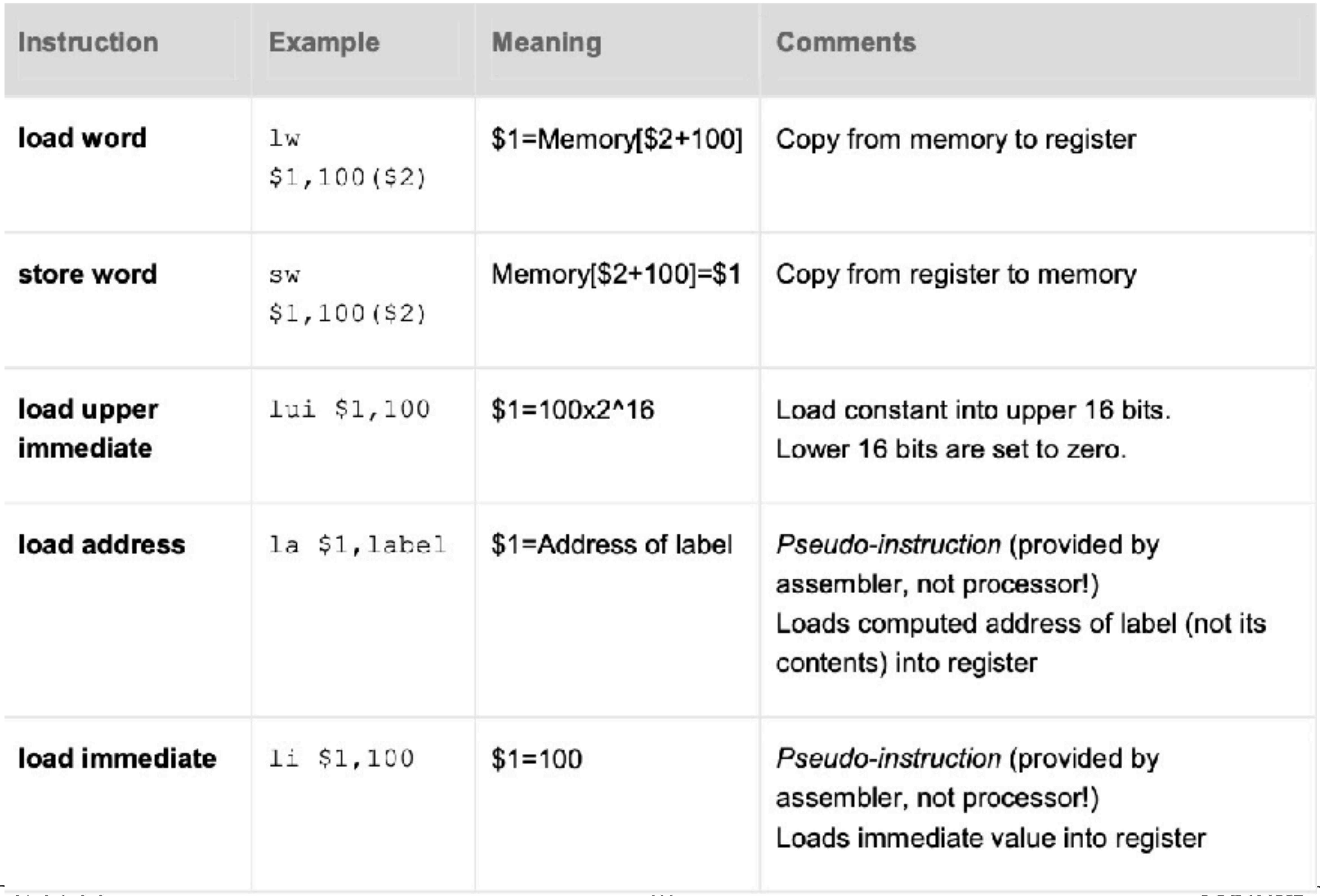
add immediate unsigned	`addiu $1,$2,100`	$1=$2+100	Values are treated as unsigned integers, not two's complement integers
Multiply (without overflow)	`mul $1,$2,$3`	$1=$2*$3	Result is only 32 bits!
Multiply	`mult $2,$3`	$hi,$low=$2*$3	Upper 32 bits stored in special register `hi` Lower 32 bits stored in special register `lo`
Divide	`div $2,$3`	$hi,$low=$2/$3	Remainder stored in special register `hi` Quotient stored in special register`lo`

MIPS Instructions: Logical



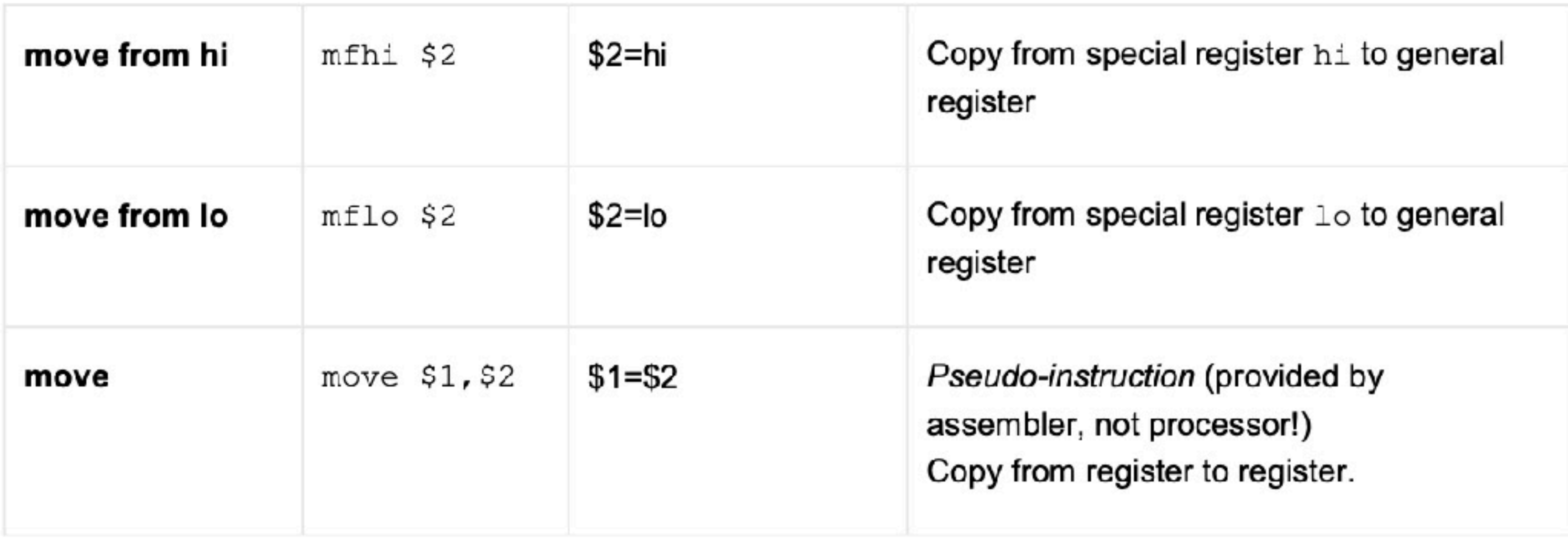
Instruction	Example	Meaning	Comments
and	`and $1,$2,$3`	$1=$2&$3	Bitwise AND
or	`or $1,$2,$3`	$1=$2|$3	Bitwise OR
and immediate	`andi $1,$2,100`	$1=$2&100	Bitwise AND with immediate value
or immediate	`or $1,$2,100`	$1=$2|100	Bitwise OR with immediate value
shift left logical	`sll $1,$2,10`	$1=$2<<10	Shift left by constant number of bits
shift right logical	`srl $1,$2,10`	$1=$2>>10	Shift right by constant number of bits

MIPS Instructions: Data Transfer



Instruction	Example	Meaning	Comments
load word	`lw $1,100($2)`	\$1=Memory[\$2+100]	Copy from memory to register
store word	`sw $1,100($2)`	Memory[\$2+100]=\$1	Copy from register to memory
load upper immediate	`lui $1,100`	\$1=100x2^16	Load constant into upper 16 bits. Lower 16 bits are set to zero.
load address	`la $1,label`	\$1=Address of label	*Pseudo-instruction* (provided by assembler, not processor!) Loads computed address of label (not its contents) into register
load immediate	`li $1,100`	\$1=100	*Pseudo-instruction* (provided by assembler, not processor!) Loads immediate value into register

MIPS Instructions: Data Transfer



move from hi	`mfhi $2`	$2=hi	Copy from special register `hi` to general register
move from lo	`mflo $2`	$2=lo	Copy from special register `lo` to general register
move	`move $1,$2`	$1=$2	*Pseudo-instruction* (provided by assembler, not processor!) Copy from register to register.

MIPS Instructions: Others

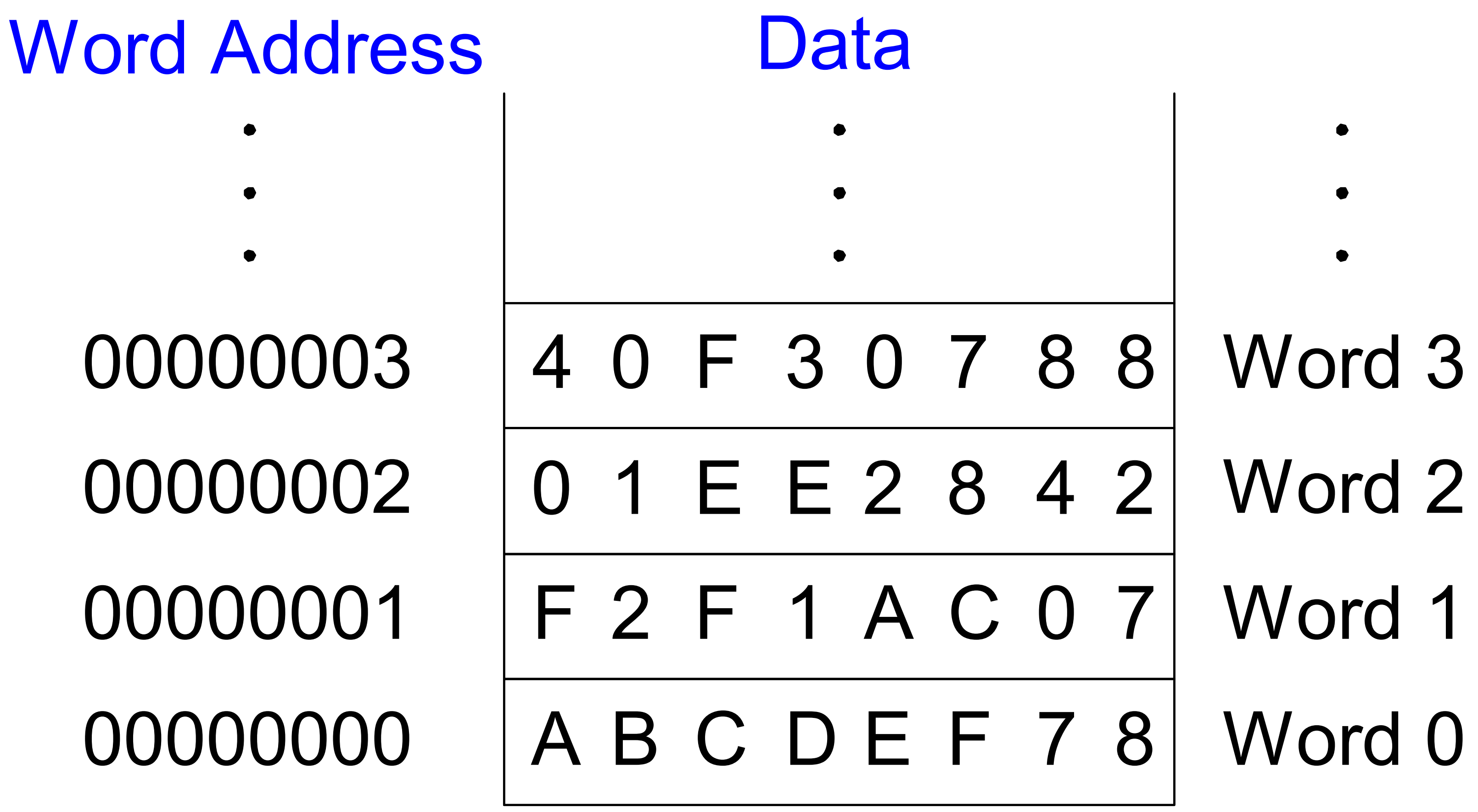
- See reference: https://www.dsi.unive.it/~gasparetto/materials/MIPS_Instruction_Set.pdf
- Un/Conditional Branch: enabling if-else, etc.
- System call: print/read/exit, etc.

Operands: Memory

- Too much data to fit in only 32 registers
- Store more data in memory
  - However, Memory is large, but slow
- Commonly used variables kept in registers

Word-Addressable Memory

- Each 32-bit data word has a unique address



Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Reading Word-Addressable Memory

- Memory read called ***load***
- **Mnemonic:** *load word* (`lw`)
- **Format:**

```
lw $s0, 5($t1)
```

- **Address calculation:**
  - add *base address* (`$t1`) to the *offset* (5)
  - address = (`$t1` + 5)
- **Result:**
  - `$s0` holds the value at address (`$t1` + 5)

Any register may be used as base address

Reading Word-Addressable Memory

- **Example:** read a word of data at memory address 1 into `$s3`
  - address = (`$0` + 1) = 1
  - `$s3` = 0xF2F1AC07 after load

Assembly code

```
lw $s3, 1($0)   # read memory word 1 into $s3
```

Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

Writing Word-Addressable Memory

- Memory write are called ***store***
- **Mnemonic:** *store word* (`sw`)

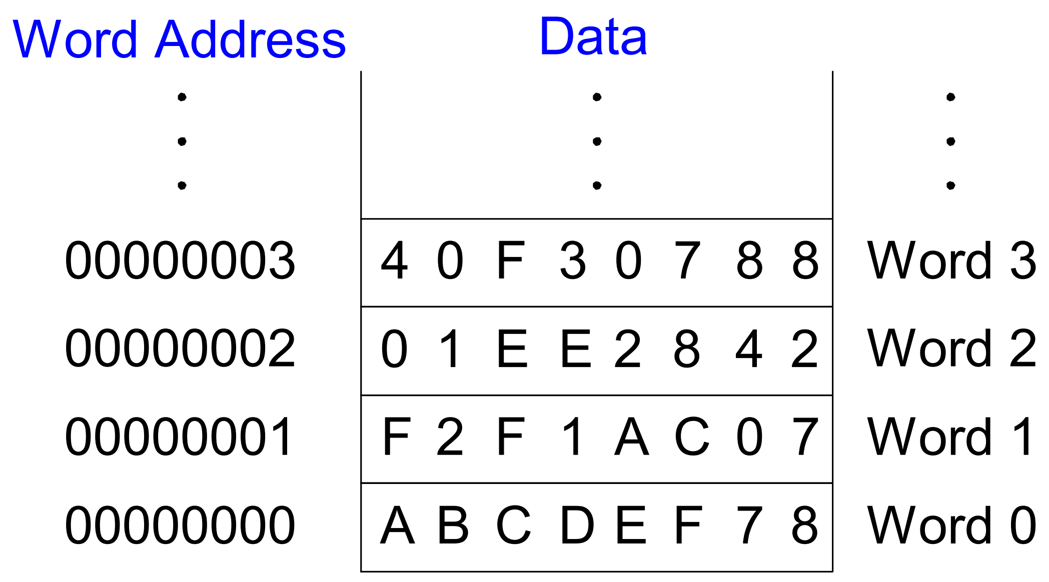
Writing Word-Addressable Memory

- **Example:** Write (store) the value in `$t4` into memory address 7
  - add the base address (`$0`) to the offset (0x7)
  - address: ($0 + 0x7) = 7

Offset can be written in decimal (default) or hexadecimal

Assembly code

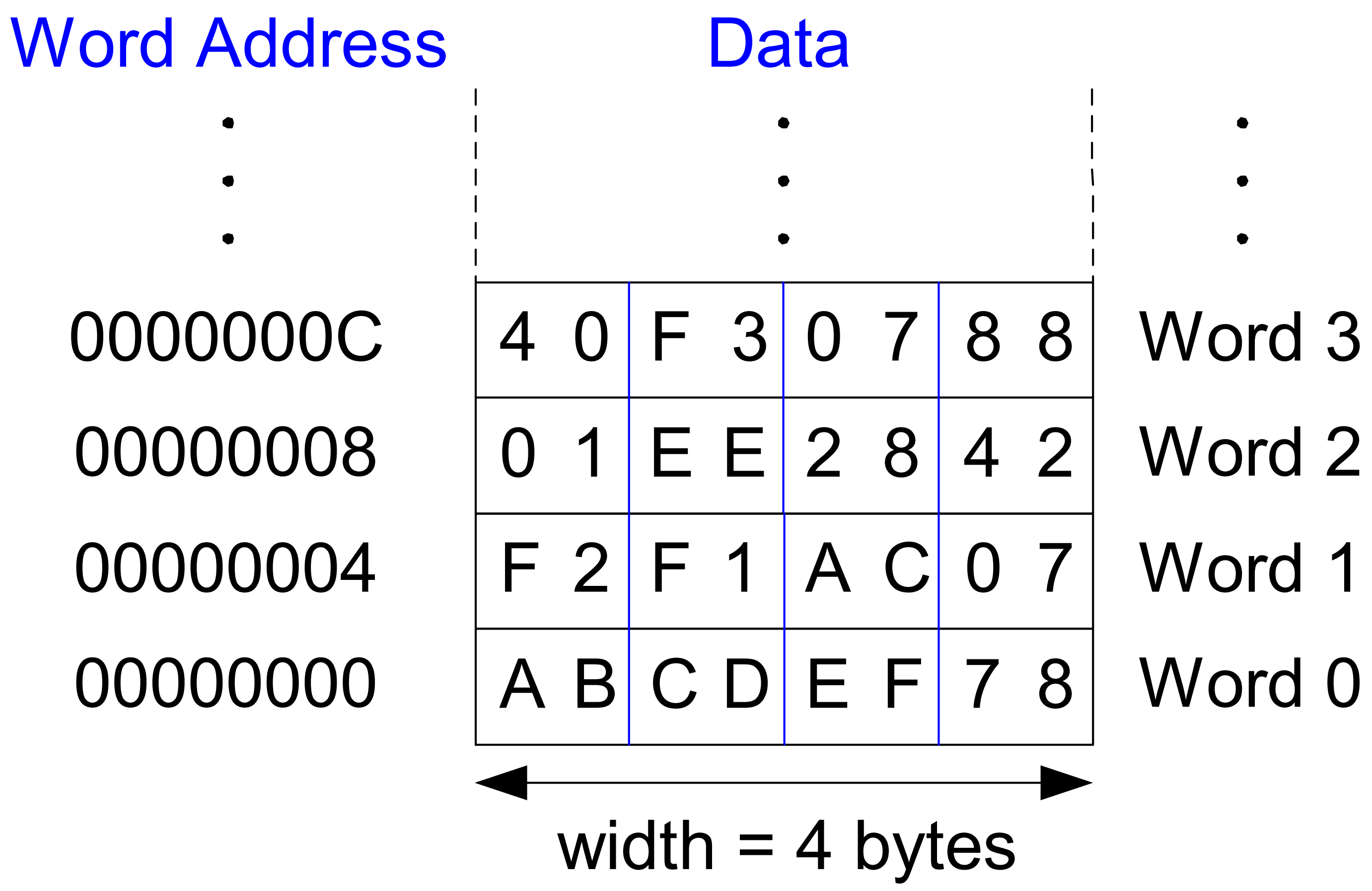
```
sw $t4, 0x7($0)   # write the value in $t4
                  # to memory word 7
```



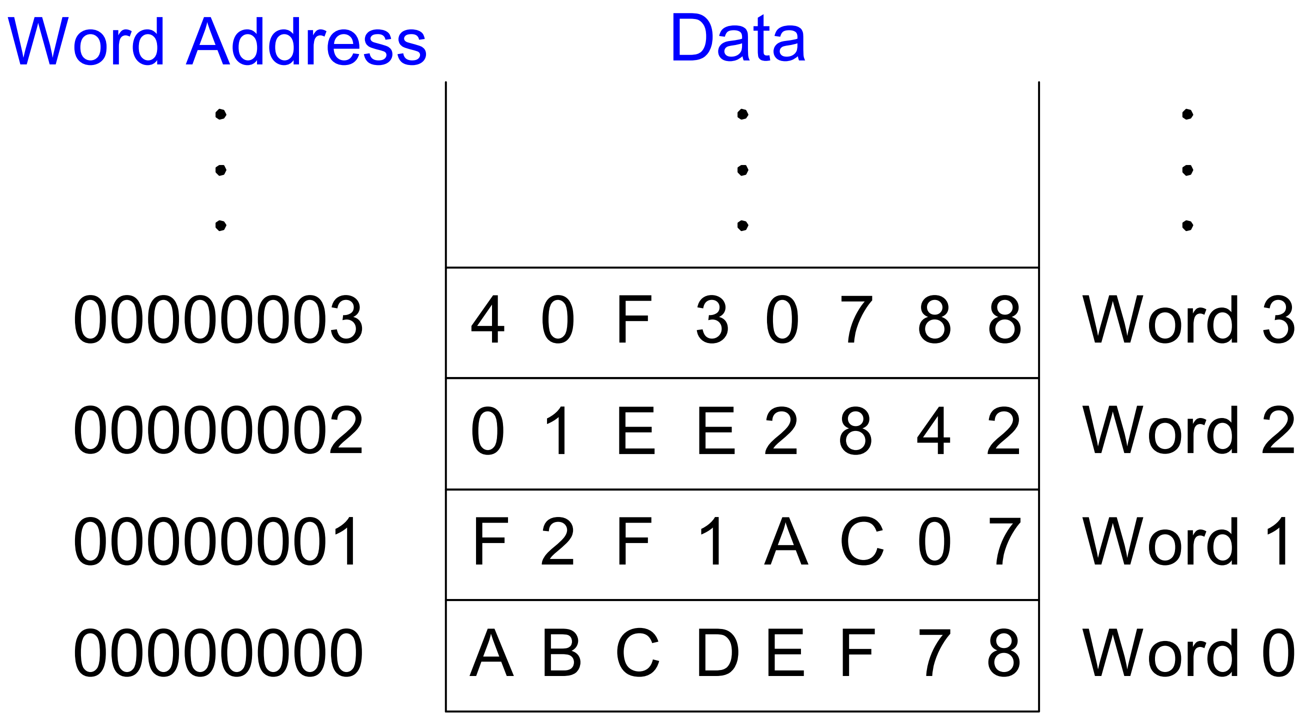
Word Address	Data	
⋮	⋮	⋮
00000003	4 0 F 3 0 7 8 8	Word 3
00000002	0 1 E E 2 8 4 2	Word 2
00000001	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

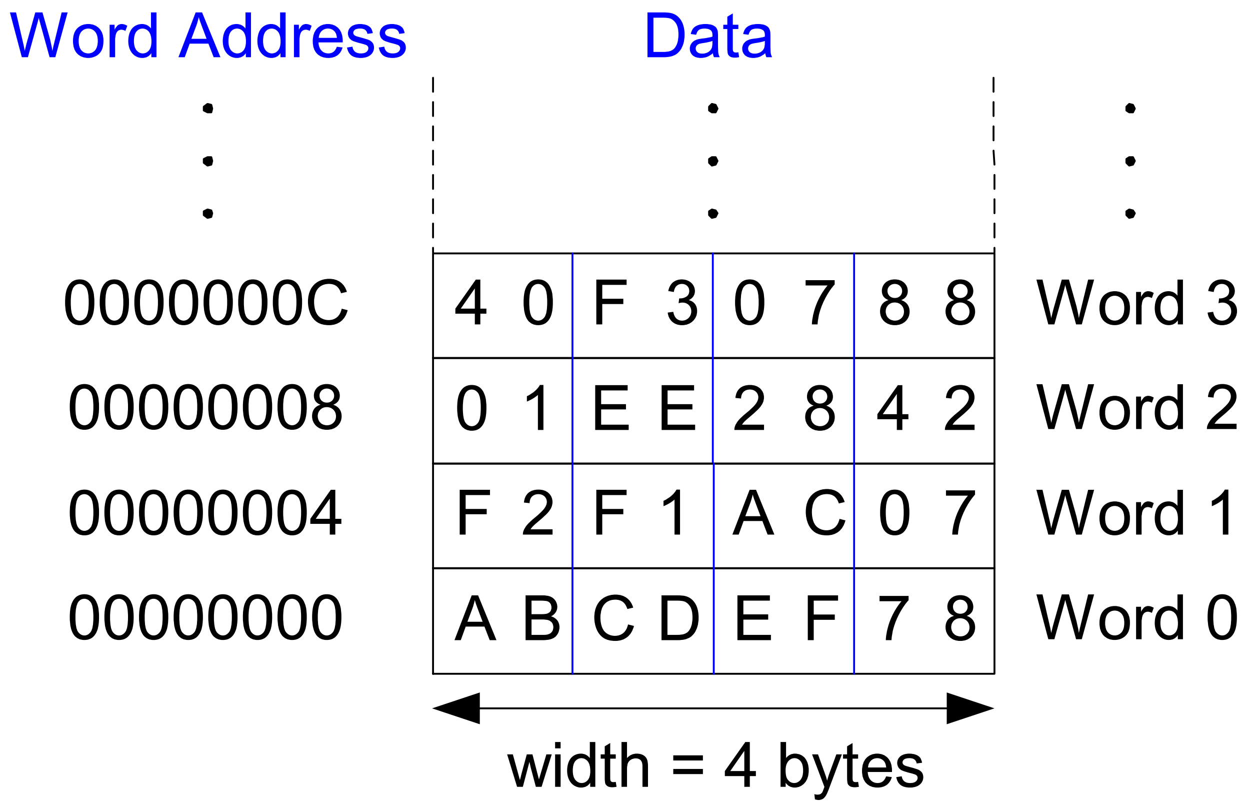
Byte-Addressable Memory

- Each data byte has unique address
- Load/store words or single bytes: load byte (`lb`) and store byte (`sb`)
- 32-bit word = 4 bytes, so word address increments by 4



Word / Byte-Addressable Memory





Reading Byte-Addressable Memory

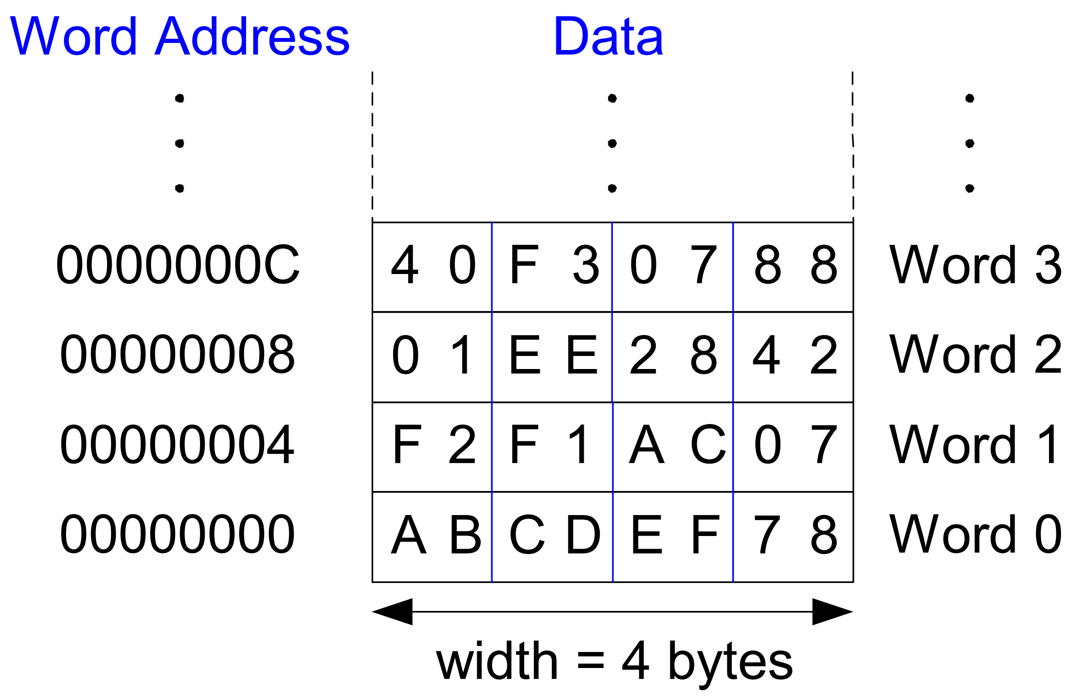
- The address of a memory word must now be multiplied by 4. For example,
  - the address of memory word 2 is $2 \times 4 = 8$
  - the address of memory word 10 is $10 \times 4 = 40$ (0x28)
- **MIPS is byte-addressed, not word-addressed**

Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value ___?___ after load

MIPS assembly code

```
lw $s3, 4($0)   # read word at address 4 into $s3
```

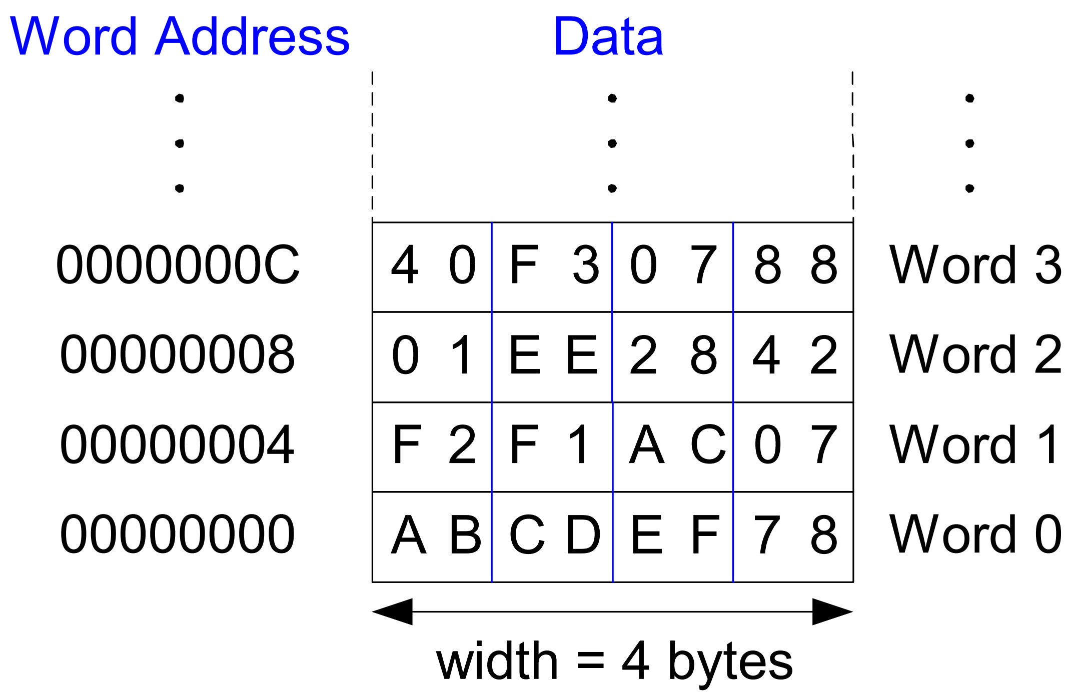


Reading Byte-Addressable Memory

- **Example:** Load a word of data at memory address 4 into `$s3`.
- `$s3` holds the value 0xF2F1AC07 after load

MIPS assembly code

```
lw $s3, 4($0)  # read word at address 4 into $s3
```

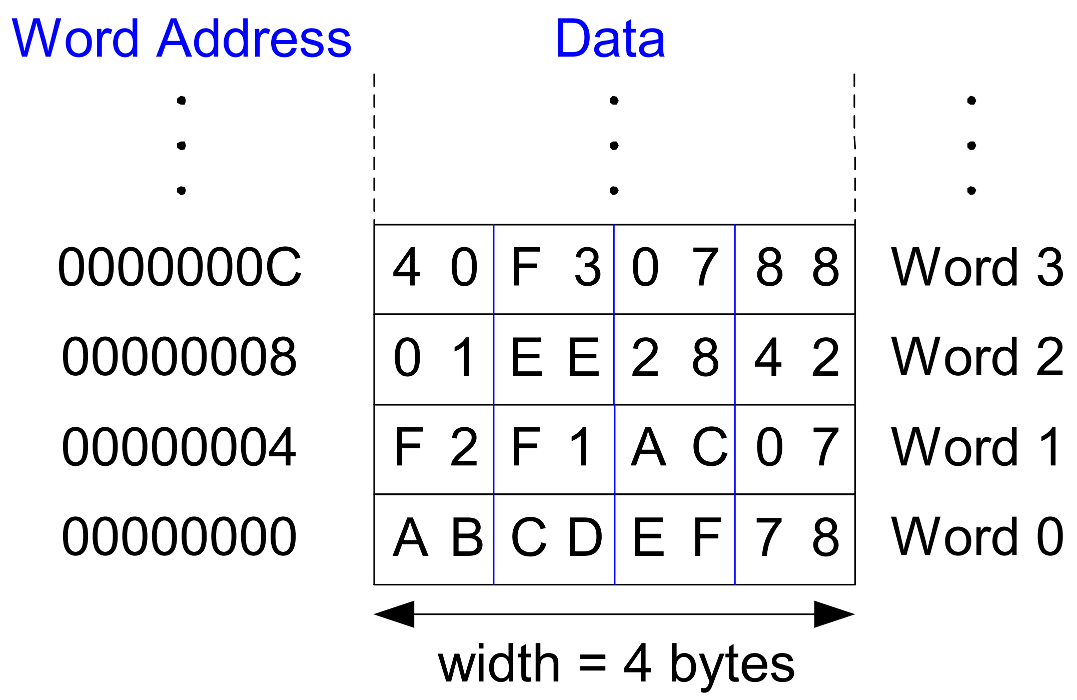


Writing Byte-Addressable Memory

- **Example:** stores the value held in `$t7` into memory address 0x2C (44)

MIPS assembly code

```
sw $t7, 44($0)   # write $t7 into address 44
```



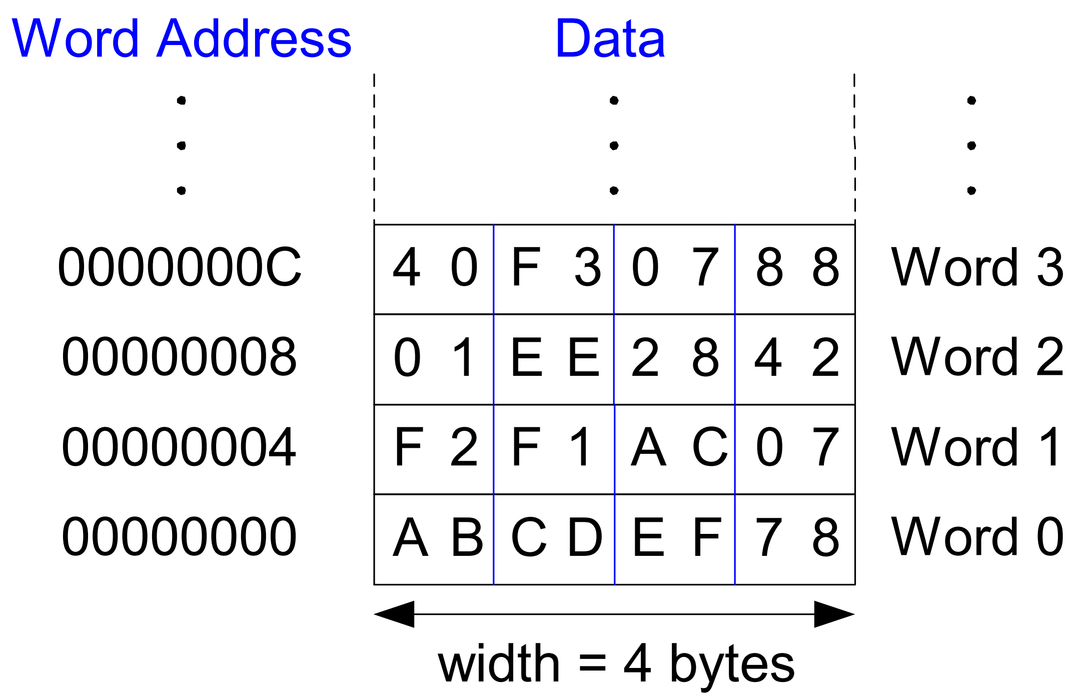
Writing Byte-Addressable Memory

- **Example:** stores the value held in `$t7` into memory address 0x2C (44)

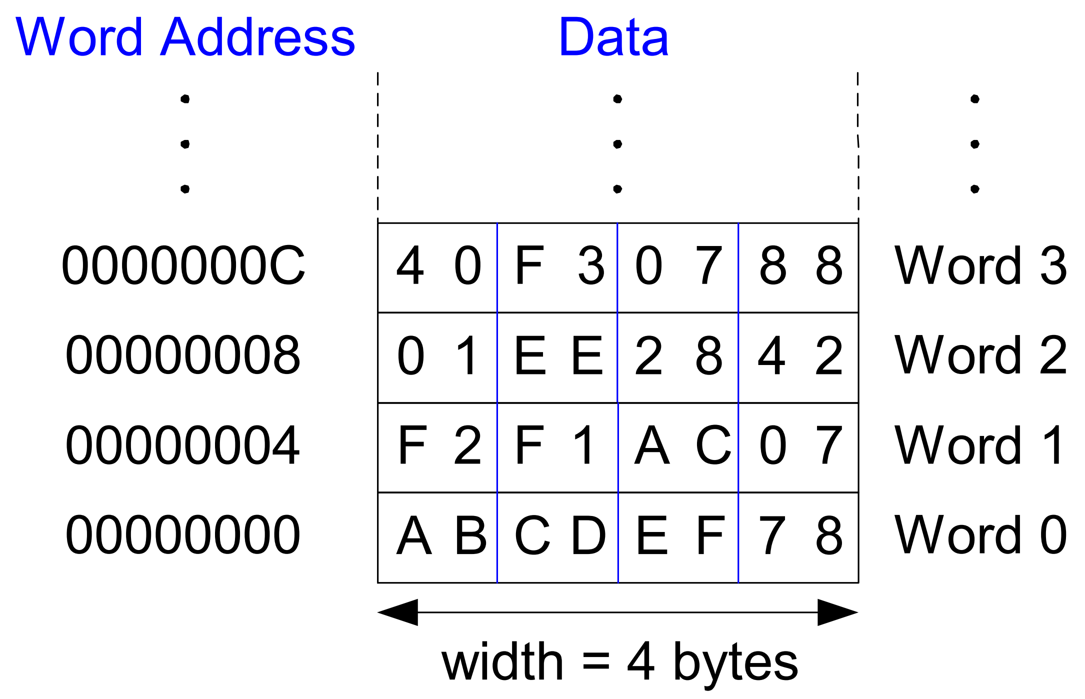
MIPS assembly code

```
sw $t7, 44($0)   # write $t7 into address 44
```

where is address 44?



Practice: Reading/Writing Byte-addressable Memory



Word Address	Data	
⋮	⋮	⋮
0000000C	4 0 F 3 0 7 8 8	Word 3
00000008	0 1 E E 2 8 4 2	Word 2
00000004	F 2 F 1 A C 0 7	Word 1
00000000	A B C D E F 7 8	Word 0

width = 4 bytes

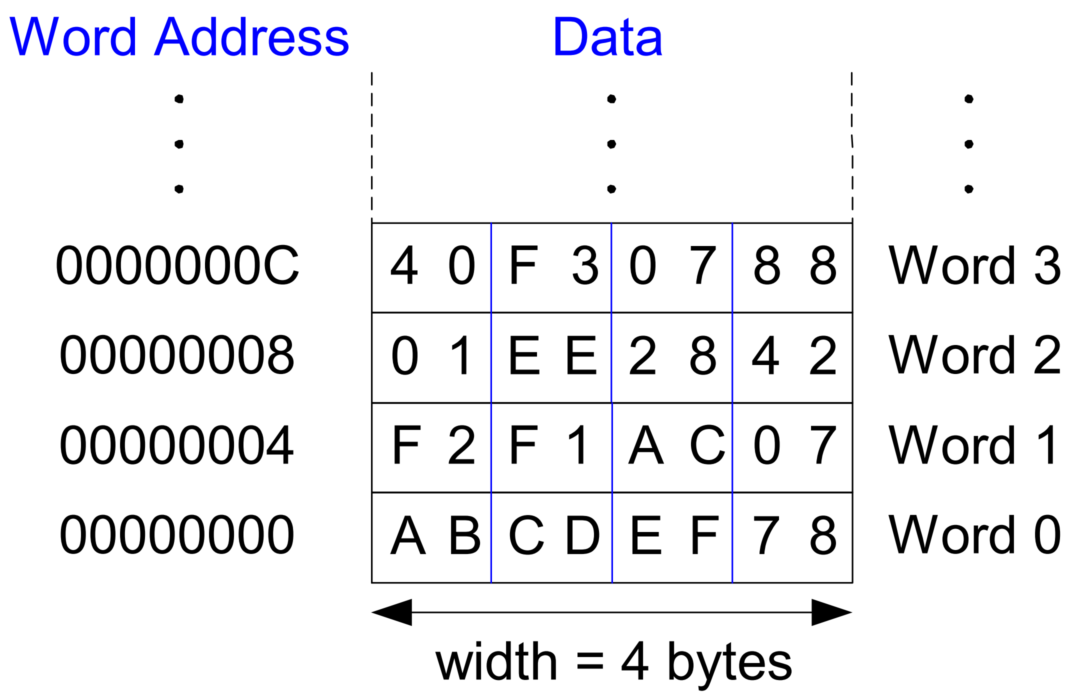
MIPS assembly code

```
lw $s0, 0($0)
lw $s1, 8($0)
lw $s2, OxC($0)
sw $s3, 4($0)
sw $s4, 0x20($0)
sw $s5, 400($0)
```

MIPS assembly code

```
# read data word _______________ into $s0
# read data word _______________ into $s1
# read data word _______________ into $s2
# write $s3 to data word ________?________
# write $s4 to data word ________?
# write $s5 to data word ________?
```

Practice: Reading/Writing Byte-addressable Memory



MIPS assembly code

MIPS assembly code

```
lw $s0, 0($0)       # read data word 0 (0xABCDEF78) into $s0
lw $s1, 8($0)       # read data word 2 (0x01EE2842) into $s1
lw $s2, OxC($0)     # read data word 3 (0x40F30788) into $s2
sw $s3, 4($0)       # write $s3 to data word 1
sw $s4, 0x20($0)    # write $s4 to data word 8
sw $s5, 400($0)     # write $s5 to data word 100
```

Problematic/Illegal Usage

- Why?

MIPS assembly code

```
lw $s0, 7($0)
```

Problematic/Illegal Usage

- Why?

MIPS assembly code

```
lw $s0, 7($0)
```

In the MIPS architecture, word addresses for lw and sw must be word aligned. That is, the address must be divisible by 4