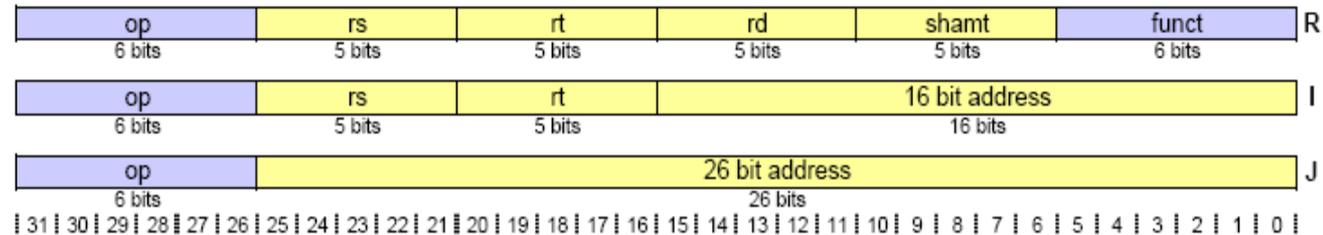


# Lecture 3: MIPS addressing modes

## Topics

Three instruction formats: R, I, J



- Traversing arrays – Further remarks
- Handling character strings in MIPS
- Handling constants in MIPS
- MIPS addressing modes
  - Addressing in branches and jumps
  - Decoding instruction

# Arrays: Element index [Text PH4, P157]

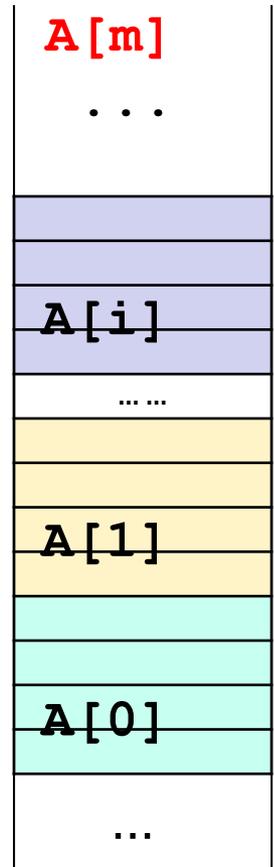
- to perform operations of a number of consecutive locations of memory we use arrays
  - to select an element of an array we use the **index**

```
int array[32];  
array[i]
```

- for example, to clear all elements of the array

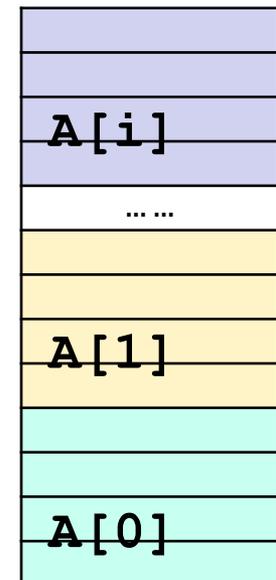
```
int i;  
for(i = 0; i < size; i = i + 1)  
    array[i] = 0;
```

- this involves calculation of the new index value and address of the next element for every iteration of the loop



# Arrays: Element pointer

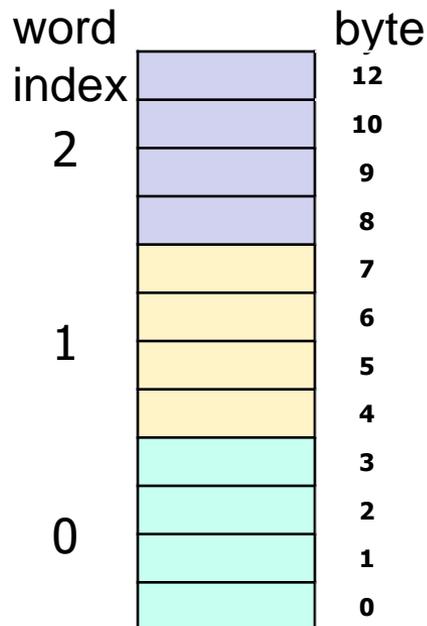
- we can operate on consecutive locations in memory by calculating the **address** directly
  - make direct use of memory abstraction (as an array of bytes)
- so instead of using the indices we use so called **pointers**
  - a pointer is an address of a memory location
  - Java does not use pointers directly (but references to objects), but C and C++ do
- However using pointers
  - the code is more cryptic
  - it is easier to make mistakes



# Array addressing and traversing in MIPS

- Memory ARRAY access: **A[k]**

Index	Offset	Base	Location	Content Addressing
k	k*4	A	k*4 + A	$k*4 (A)$
				$0 (A + k*4)$

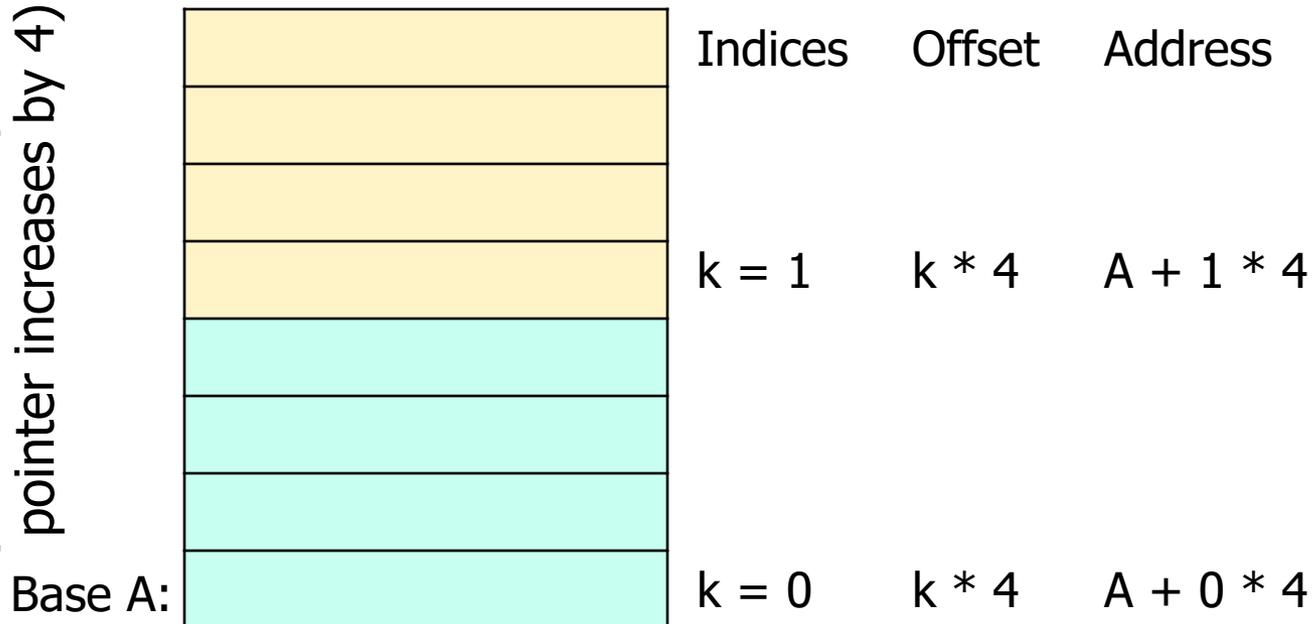


# Array addressing and traversing in MIPS

- Memory ARRAY access: **A[k]**

Traversing style	Counter	Address Algebra
<b>Indices</b> (relative, conceptual)	$k [0, m]$	$k * 4 + A$
<b>Pointers</b> (absolute, physical)	$P [A, A + m*4]$	$P += 4$

$\text{pointer} = \text{index} * 4 + \text{base}$   
 (index increases by 1 =>  
 pointer increases by 4)



# Array addressing and traversing in MIPS

TASK: clear a number of consecutive locations in memory in C:

```
int *p;
for(p = &array[0]; p <= &array[size-1]; p = p + 1)
    *p = 0;
```

## ■ Indexing version

```
# array base and size are in registers $a0 and $a1
# index i is allocated to register $t0
    move $t0, $zero          # i = 0
loop1: add $t1,$t0,$t0       # $t1 = 2i
    add $t1,$t1,$t1         # $t1 = 4i
    add $t2,$a0,$t1        # $t2 = array[i]
    sw $zero,0($t2)        # array[i] = 0
    addi $t0,$t0,1         # i = i + 1
    slt $t3,$t0,$a1       # $t3 = (i < size0)
    bne $t3,$zero,loop1
```

# Array addressing and traversing in MIPS

- Pointer version

# array address (address of the first array word) and array size (how many words the array has) are found in registers \$a0 and \$a1

# pointer p is allocated to register \$t0

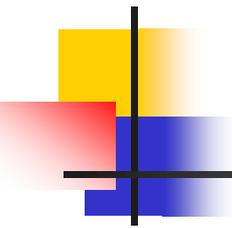
```
        move $t0, $a0           # p = address of array[0]
loop2:  sw $zero, 0($t0)        # memory[p] = 0
        addi $t0, $t0, 4        # p = p + 4
        add $t1, $a1, $a1       # $t1 = 2 x size
        add $t1, $t1, $t1       # $t1 = 4 x size
        add $t2, $a0, $t1       # $t2 = address of array[size]
        slt $t3, $t0, $t2       # $t3 = (p < &array[size])
        bne $t3, $zero, loop2
```

# Array addressing and traversing in MIPS

- Pointer version – some improvement

# the address of array[size] is calculated for every iteration of the loop, although it never changes  
# so we can move it outside the loop:

```
        move $t0, $a0           # p = address of array[0]
        add  $t1, $a1, $a1      # $t1 = 2 x size
        add  $t1, $t1, $t1      # $t1 = 4 x size
        add  $t2, $a0, $t1      # $t2 = &array[size]
loop2:  sw   $zero, 0($t0)      # memory[p] = 0
        addi $t0, $t0, 4        # p = p + 4
        slt  $t3, $t0, $t2      # $t3 = (p < &array[size])
        bne $t3, $zero, loop2
```

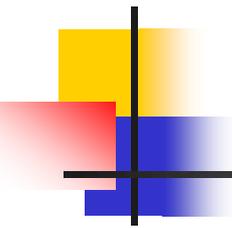


# Array addressing and traversing in MIPS

---

- Comparison

- the index version had to calculate new value of  $i$  for every iteration of the loop: 7 instructions per iteration
- the pointer version calculated size once only outside the loop: 4 instructions per iteration
- modern compilers have the ability to produce the more efficient pointer-like code for the array version



# While loop again (EXERCISE)

---

```
# actual start of the main program
```

```
# implements a while loop
```

```
    while (save[i] == k)
```

```
        i = i + j;
```

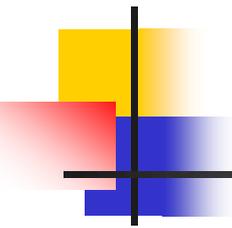
```
# alternative form:
```

```
    Loop:  if (save[i] != k) go to Exit
```

```
        i = i + j;
```

```
        go to Loop;
```

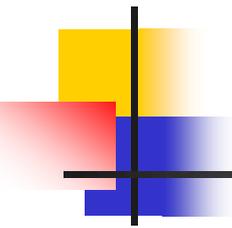
```
    Exit:
```



# While loop

---

```
# variables i, j and k are in registers $s3, $s4 and $s5
# address of save is in $s6
# Allocate 10-word array save
    .data
    .align 2                # aligning will be explained later
    .globl save
save: .word 0, 0, 0, 0, 0, 0, 0, 6, 3, 2
    .globl main
    .text
main:                        # main has to be a global label
    addu $s7, $0, $ra      # save the return address in $ra
# Initialize variables
    add $s3, $0, $0        # i=0 (initial value)
    addi $s4, $0, 1        # j=1
    add $s5, $0, $0        # k=0 (what if k=7 -> addi $s5,$0,7)
    la $s6, save           # $s6 = save[] (using la)
```



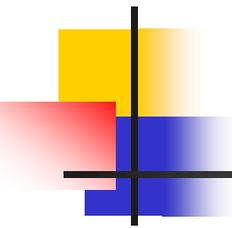
# While loop cont

---

Loop:

```
add $t1, $s3, $s3      # 2*i ($s3 has i)
add $t1, $t1, $t1      # 4*i ($t1 has 4*i -- the offset)
add $t1, $t1, $s6      # $s6 = save[]; $t1 = save[] + 4*i
                        #
lw $t0, 0($t1)         # gets save[i]
                        #
bne $t0, $s5, Exit     # $s5 has k
add $s3, $s3, $s4      # $s3 has i
j Loop                 #
```

Exit:



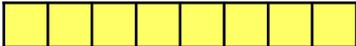
# While loop cont

---

```
# Output the value of i to see how far we got
.data
.globl message1
message1: .asciiz "\nThe value of i is: " #string to print
.text
li $v0, 4          #
la $a0, message1  # la used here
syscall
li $v0, 1          #
add $a0, $0, $s3   #
syscall

# Usual stuff at the end of the main
addu $ra, $0, $s7  # restore the return address
jr $ra             # return to the main program
add $0, $0, $0     # nop
```

# Processing text

- Computers can process any information represented as numbers
  - characters can be processed if they are represented as numbers
- ASCII (American Standard Code for Information Interchange – refer to the table in the last slide)
  - 8 bits (**one byte**) used to represent a character 
  - 256 possible combinations
- EBCIDC
  - another 8-bit code, introduced by IBM
- Unicode ...
  - 16 bits per character, Java

# Storing characters

- Storing an **8-bit** character in a **32-bit** word would be wasteful

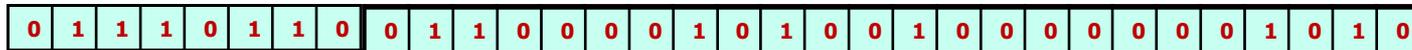
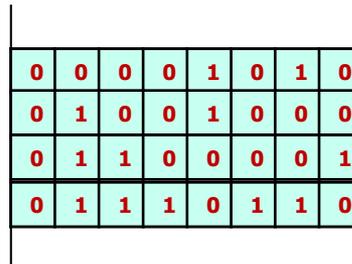
0	0	0	0	1	0	1	0
0	1	0	0	1	0	0	0
0	1	1	0	0	0	0	1
0	1	1	1	0	1	1	0

0	1	1	1	0	1	1	0	0	1	1	0	0	0	0	1	0	1	0	0	1	0	0	0	0	0	0	0	1	0	1	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

- We want to pack 4 characters into each word
  - 'Have a ... ..' [ $0A_{00001010}^{48}$   $01001000^{61}$   $01100001^{76}$   $01110110^{20}$  ...]
  - need a convention on how bytes are ordered in a word
    - this is called **endianness**
  - two ways are used
    - pack characters starting at the most significant bit (big end)
    - pack characters starting at the least significant bit (little end)

# Special instructions

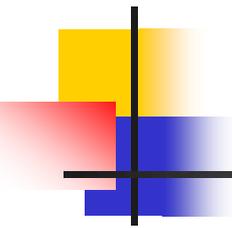
- we might use lw and sw to transfer characters bundled into words between memory and registers
  - we would need to extract single bytes to process text (masking?)



- since character processing is so common (4th design principle), special instructions are provided

Byte #			
3	2	1	0

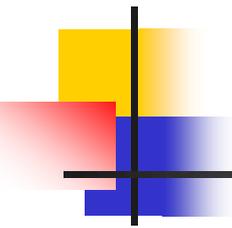
- lb register, address
  - loads a byte from memory into **rightmost** byte of the register
- sb register, address
  - stores the **rightmost** byte of the register in memory



# Character strings

---

- text consists of strings of characters
  - strings have variable length
- three choices
  1. reserve the first position of the string to store its length
  2. create a structure consisting of an integer to store length, and a variable length character string
  3. use last position of the string to mark its end by storing a terminating character
- 3rd choice is commonly used
  - C sets the last byte to zero
  - a byte whose value is zero is called Null in ASCII



# MIPS directives for strings

A-48 (PH3) or B-48 (PH4) explains directive `.asciiz`

- `.ascii` "sample character string"
  - stores the characters in memory packed in consecutive bytes
  - this string occupies 23 bytes of memory
  - strings are enclosed in double quotes
- `.asciiz` "sample character string"
  - adds a null byte at the end of the string
  - this string occupies 24 bytes of memory
- special characters follow C conventions
  - new line `\n`
  - tab `\t`
  - double quote `\"`

# Strings in MIPS [EXERCISE]

- C code to copy string *y* to string *x* using the Null byte as a terminating character

```
char x[], y[];
int i = 0; //counter
while(y[i] != 0) {
    x[i] = y[i];
    i = i + 1;
}
```

- MIPS assembly language code

# base address for x and y is in \$a0 and \$a1  
 # i is in \$s0

```

    add $s0,$zero,$zero # i = 0 (initialisation)
L1:  add $t1,$a1,$s0     # address of y[i]; not 4*i (word)
    ① lb $t2,0($t1)     # load character y[i] to $t2
    add $t3,$a0,$s0     # address of x[i]; not 4*i (word)
    ② sb $t2,0($t3)     # x[i] = y[i]
    addi $s0,$s0,1      # i = i + 1 (update; byte by byte)
    bne $t2,$zero,L1   # if y[i] != 0 (the NUL char)
```

18	0	0	0	0	1	0	1	0
19	0	1	0	0	1	0	0	0
1A	0	1	1	0	0	0	0	1
1B	0	1	1	1	0	1	1	0
1C								

lb / sb

# Constants

See Text: HP4, P86

- Small constants are used quite frequently

e.g., `a = a + 4`

- Solutions?

1. build hard-wired registers (like \$zero) for constants
2. define 'typical constants' in memory and load them
3. put constants in instructions themselves

```
addi $sp,$sp,4      # add immediate (add 4)
```

```
slti $t0,$s2,35     # common use of constants is comparison
```

- We call constant operands **immediate** operands

- this way of accessing data is called **immediate addressing**

4th principle of good design:  
Make the common case fast

# Constants continued

- I-type Instruction with **immediate** operands

op	rs	rt	constant
6 bits	5 bits	5 bits	16 bits

- Large** constant: I-type instructions limit to 16-bit constants, HOW do we load a 32-bit constant into a 32-bit register?

For example: 1110101010101011 1010101010101010

- New "load upper immediate" takes care of higher order 16 bits:

```
lui $t0, 1110101010101011
```

- Then we must get the lower order 16 bits right, i.e.,

```
ori $t0, $t0, 1010101010101010
```

	1110101010101011	0000000000000000	\$t0
ori	0000000000000000	1010101010101010	immediate
	1110101010101011	1010101010101010	

# Large constants cont.

- MIPS assembler provides a **pseudo** instruction

```
li register, constant # load immediate
```



replaced with

```
lui reg, const1 # const1 := constant >> 16
```

```
ori reg, reg, const2 # const2 := constant AND 0x0000ffff
```

```
1110101010101011 1010101010101010
```

- the constant must be broken at some stage either by the compiler, or the assembler (as in MIPS)
  - a temporary register **\$at** is used for this purpose
  - this is why it is reserved (NOT enforced by hardware!)

# Addresses in loads/stores

See Text: HP4, P128-P133

- **lw** and **sw** are I-type instructions
  - **lw** *rt*, **address** # e.g. **lw \$t0, 32(\$s3)**
  - **address**: constant [offset] + contents of a register *rs* [base]

<b>op</b>	<b>rs</b>	<b>rt</b>	<b>constant [offset]</b>
6 bits	5 bits	5 bits	16 bits

- how about:

```
.data
labelx: .word 235      # define number '235'
.text
lw $t0, labelx # I-type instruction
```

- labelx address is 32-bit long, BUT constant field is only 16 bits
  - these **lw/sw** are treated by the assembler as pseudoinstructions
  - substituted with **lui** and common **lw/sw**

# Load instructions

- so far we saw

```
li register, constant      # pseudoinstruction  
                           # translate into more  
                           # than 1 common instr.
```

```
lw register, address      # e.g. lw $t0, 32($s3)  
                           # or lw $t0, labelx
```

- another useful load instruction

```
la register, address      # pseudoinstruction
```

- load computed address, NOT the contents of the location
- this is another case when we need a 32-bit immediate value

```
A: .word 11,12,13,14,15,16,17 #Array definition A[7], see Lab 4  
la $s3, A  
lw $s4, 8($s3)
```

# Addresses in Jumps, part 1

- the simplest addressing is in jump instruction

```
j Label # Next instruction is at Label
```

- such instruction use another format: J-type

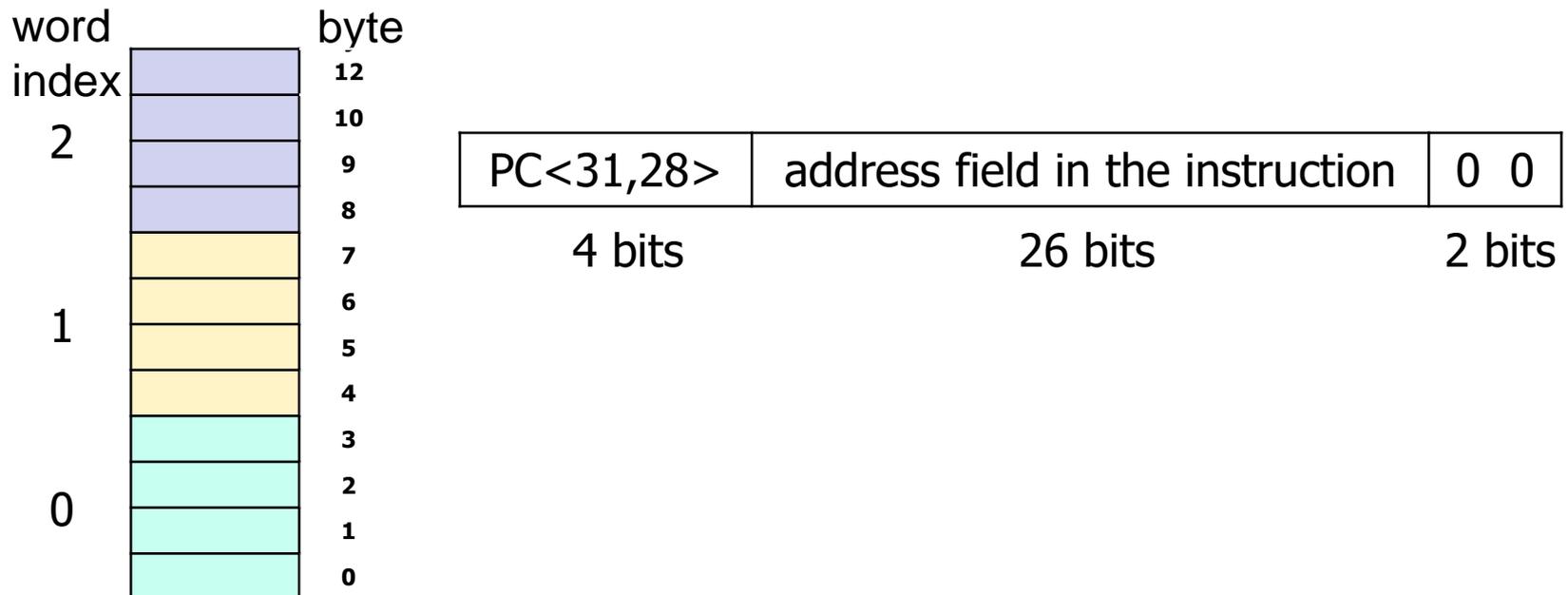
op	address
6 bits	26 bits

- ...but address field is still only 26 bits, not 32 bits required by the address (e.g. Loop address) , what to do?

```
Loop: add $t1, $s3, $s3 #
      ... ..
      lw $t0, 0($t1) #
      bne $t0, $s5, Exit #
      add $s3, $s3, $s4 #
      j Loop #
```

# Addresses in Jumps, part 2

- as each instruction is 4 byte long, we only need to address words (not individual bytes – cells), so:
  - 26-bit address field can represent 28-bit byte address
- the 4 missing bits are provided by leaving the upper 4 bits of the PC (Program Counter) unchanged



# Addresses in Branches

- Instructions:

```
bne $t4,$t5,Label    #Next instruction is at Label if $t4!=$t5
beq $t4,$t5,Label    #Next instruction is at Label if $t4==$t5
```

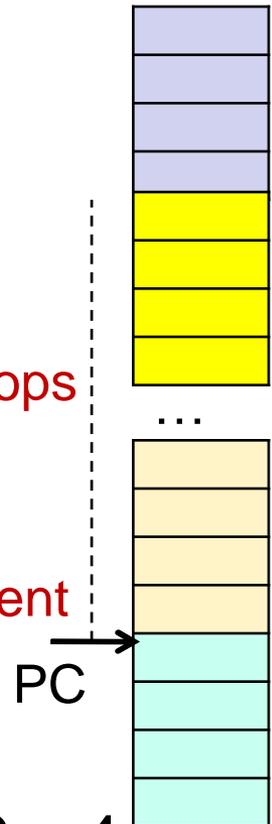
- Format (I-Type):

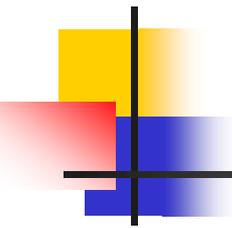
<b>op</b>	<b>rs</b>	<b>rt</b>	<b>constant</b>
6 bits	5 bits	5 bits	16 bit address

- we only have 16 bits for the address
- this limits the size of the program to  $2^{16}$ , not an acceptable option

# Addresses in Branches: PC-relative addressing

- we could specify a register (like in lw and sw, e.g. `lw $t0, 32($s3)`) and add it to address
  - so next PC = register + branch\_address
- which register to use
  - most branches are local (principle of locality), used in loops and if statements
  - use **Program Counter (PC)**
  - we can branch within  $2^{15}$  words either way from the current instruction (not  $2^{16}$ , leaving one bit for direction)
- this is called PC-relative addressing: **const(PC)**
- MIPS uses the address of the next instruction, PC + 4
  - by the time when address is calculated PC has been already incremented





# Branching far away

---

- if a branch is to a far-away location (beyond the 16 bit limit)
  - the assembler replaces the branch with a pair of instructions
  - for example:

```
    beq $s0,$s1,L1
    add . . .
```

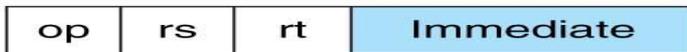
- becomes

```
                bne $s0,$s1,L2
                j  L1
L2:            add . . .
```

# Addressing mode summary

PH4 Section 2.10, P132

## 1. Immediate addressing



I-type instructions

## 2. Register addressing

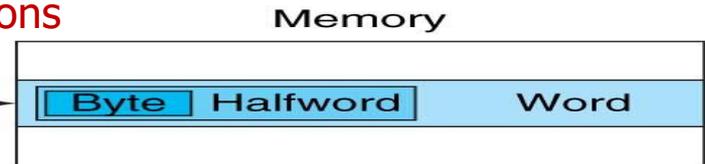


R-type instructions

## 3. Base addressing



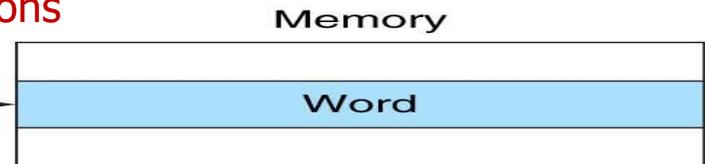
I-type instructions



## 4. PC-relative addressing



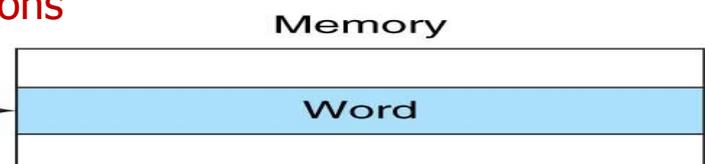
I-type instructions



## 5. Pseudodirect addressing

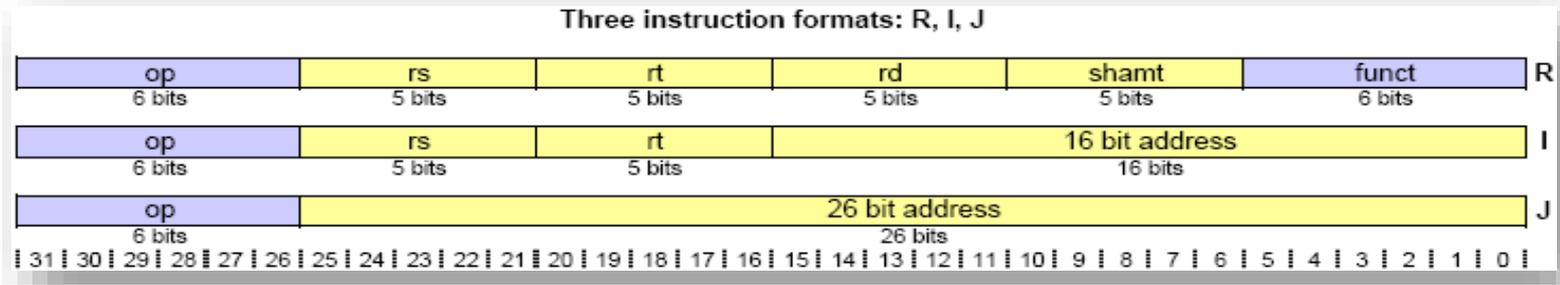


J-type instructions



# Instruction Formats

- Three instruction format: R, I, J



- Instruction assembling that converts mnemonic format to machine code

R-type instruction: **add**  $\$t0$ ,  $\$s1$ ,  $\$s2$  [mnemonic]  
**add** 8 17 18 [assembled]

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
	<b>op</b>	<b>rs</b>	<b>rt</b>	<b>rd</b>	<b>shamt</b>	<b>funct</b>
dec	0	17	18	8	0	32
bin	000000	10001	10010	01000	00000	100000
0x	<b>0</b>	<b>2</b>	<b>3</b>	<b>2</b>	<b>4</b>	<b>0</b>

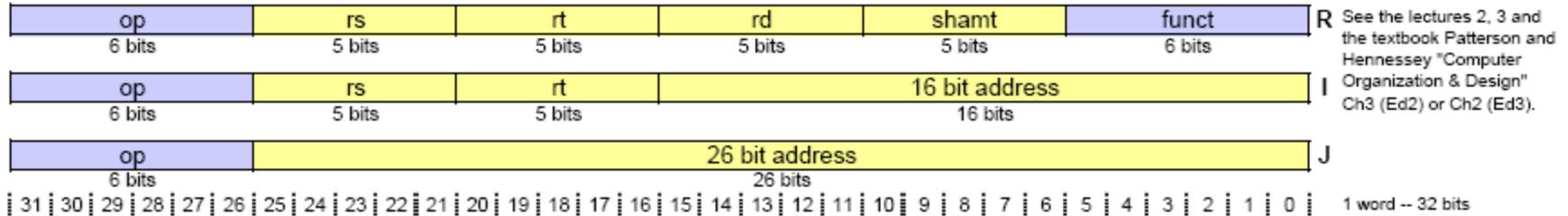
# Home EXERCISE

## Examples of translating machine instructions

See Text: HP4, P134; *instruction decoding.pdf* on vUWS

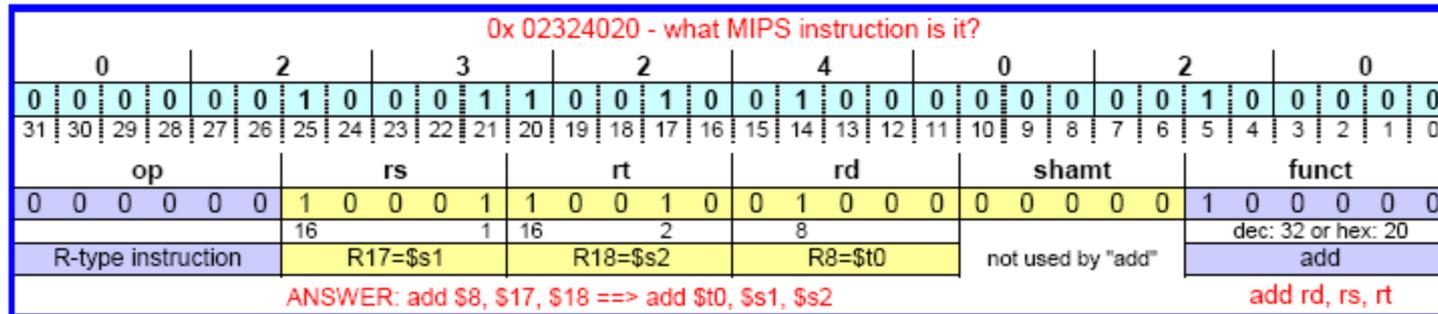
Three instruction formats: R, I, J

NOTES:

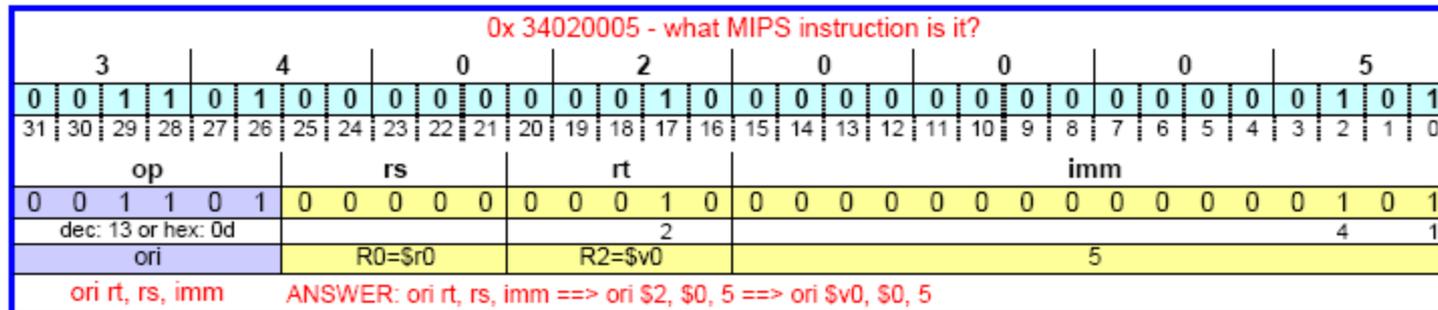


See the lectures 2, 3 and the textbook Patterson and Hennessey "Computer Organization & Design" Ch3 (Ed2) or Ch2 (Ed3).

Two examples of translating a machine instruction into a MIPS assembly instruction



op (operation) field tells us that this is R-type instruction (see the textbook from p.117, Fig. 3.18 or Fig. A.19). R-type instructions fields are allocated in groups: 6bits-5bits-5bits-5bits-5bits-6bits. Funct field tells us that the instruction is 'add', format: add rd, rs, rt (p. A-55).



op field tells us that this instruction is 'ori' (see the textbook from p.117, Fig. 3.18 or Fig. A.19). ori is I-type instructions, so fields are allocated in groups: 6bits-5bits-5bits-18bits. The instruction format is: ori rt, rs, imm (p. A-57).

© Derek Bem, 2005

# Revision: Memory access

- Given `var: .word 32`

Which of the following is to load the address of `var` to register `$s1`?

1) `la $s1, var` 2) `lw $s1, var` 3) `li $s1, var`

- Given `arr: .word 0, 0, 0, 0, 0, 0, 0, 6, 3, 2`

Which of the following is the correct algebra to calculate the address of `arr[i]`?

1) `arr+4*i` 2) `arr+4+i` 3) `(arr+i)*4`

- Given `arr: .word 0, 0, 0, 0, 0, 0, 0, 6, 3, 2` and assume array base and index are in registers `$a0` and `$t0`.

Is the following code legal in syntax?

```
add $t1, $t0, $t0
```

```
add $t1, $t1, $t1
```

```
lw $t2, $t1($a0)
```

# Recommended readings

<b>General Data</b>	<a href="#">UnitOutline</a>   <a href="#">LearningGuide</a>   <a href="#">Teaching Schedule</a>   <a href="#">Aligning Assessments</a> 
<b>Extra Materials</b>	<a href="#">ascii_chart.pdf</a>   <a href="#">bias_representation.pdf</a>   <a href="#">HP_AppA.pdf</a>   <a href="#">instruction_decoding.pdf</a>   <a href="#">masking_help.pdf</a>   <a href="#">PCSpim.pdf</a>   <a href="#">PCSpim Portable Version</a>   <a href="#">Library materials</a>

PH6, §2.3, P72: immediate operands; making the common case fast  
PH5, §2.3, P72: immediate operands; making the common case fast  
PH4, §2.3, P86: immediate operands; 3rd Principle of hardware design

PH6, §2.10, P118: Addressing mode in MIPS  
PH5, §2.10, P111: Addressing mode in MIPS  
PH4, §2.10, P128: Addressing mode in MIPS

PH6, §2.10, P125-P127: Instruction decoding and Instruction Formats  
PH5, §2.10, P118-P120: Instruction decoding and Instruction Formats  
PH4, §2.10, P134-P136: Instruction decoding and Instruction Formats  
Also refer to "instruction\_decoding.pdf" on vUWS

PH6, §2.14, P147: Traversing arrays – index vs pointer  
PH5, §2.14, P141: Traversing arrays – index vs pointer  
PH4, §2.14, P157: Traversing arrays – index vs pointer

HP\_AppA.pdf -> A-43 (PH6, PH5) or P-43 (PH4) pack characters  
Also refer to "ascii\_chart.pdf" on vUWS

HP\_AppA.pdf -> A-48 (PH6, PH5) or P-48 (PH4) explains directive .asciiiz

Text readings are listed in Teaching Schedule and Learning Guide

PH6 (PH5 & PH4 also suitable): check whether eBook available on library site

PH6: companion materials (e.g. online sections for further readings)

<https://www.elsevier.com/books-and-journals/book-companion/9780128201091>

PH5: companion materials (e.g. online sections for further readings)

<http://booksite.elsevier.com/9780124077263/?ISBN=9780124077263>

# ASCII TABLE

See *ascii\_chart.pdf* on vUWS

DEC	HEX	CHAR		DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	00	^@	NUL	32	20	SPC	64	40	@	96	60	`
1	01	^A	SOH	33	21	!	65	41	A	97	61	a
2	02	^B	STX	34	22	"	66	42	B	98	62	b
3	03	^C	ETX	35	23	#	67	43	C	99	63	c
4	04	^D	EOT	36	24	\$	68	44	D	100	64	d
5	05	^E	ENQ	37	25	%	69	45	E	101	65	e
6	06	^F	ACK	38	26	&	70	46	F	102	66	f
7	07	^G	BEL	39	27	'	71	47	G	103	67	g
8	08	^H	BS	40	28	(	72	48	H	104	68	h
9	09	^I	HT	41	29	)	73	49	I	105	69	i
10	0A	^J	LF	42	2A	*	74	4A	J	106	6A	j
11	0B	^K	VT	43	2B	+	75	4B	K	107	6B	k
12	0C	^L	FF	44	2C	,	76	4C	L	108	6C	l
13	0D	^M	CR	45	2D	-	77	4D	M	109	6D	m
14	0E	^N	SO	46	2E	.	78	4E	N	110	6E	n
15	0F	^O	SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P	DLE	48	30	0	80	50	P	112	70	p
17	11	^Q	DC1	49	31	1	81	51	Q	113	71	q
18	12	^R	DC2	50	32	2	82	52	R	114	72	r
19	13	^S	DC3	51	33	3	83	53	S	115	73	s
20	14	^T	DC4	52	34	4	84	54	T	116	74	t
21	15	^U	NAK	53	35	5	85	55	U	117	75	u
22	16	^V	SYN	54	36	6	86	56	V	118	76	v
23	17	^W	ETB	55	37	7	87	57	W	119	77	w
24	18	^X	CAN	56	38	8	88	58	X	120	78	x
25	19	^Y	EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z	SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[	ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^\	FS	60	3C	<	92	5C	\	124	7C	
29	1D	^]	GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^	RS	62	3E	>	94	5E	^	126	7E	~
31	1F	^_	US	63	3F	?	95	5F	_	127	7F	DEL