ECE 3504
Principles of Computer Architecture

MIPS Review (Test 1)

Readings: Ch. 2 (up to 2.13) and Appendix A

Test 1

- Study materials
  - Lecture slides
  - Read textbook (Chapter 2 + Appendix A)
  - Review the homework + sample tests
- Make sure you specify the radix used
  - Base 10 is assumed unless there is a leading 0x
  - Use hexadecimal instead of long binary strings
- OK to use pseudo-instructions given in the reference sheet or textbook or notes, *unless stated otherwise*
- Write comments even if you don't know the exact solution!

Test 1

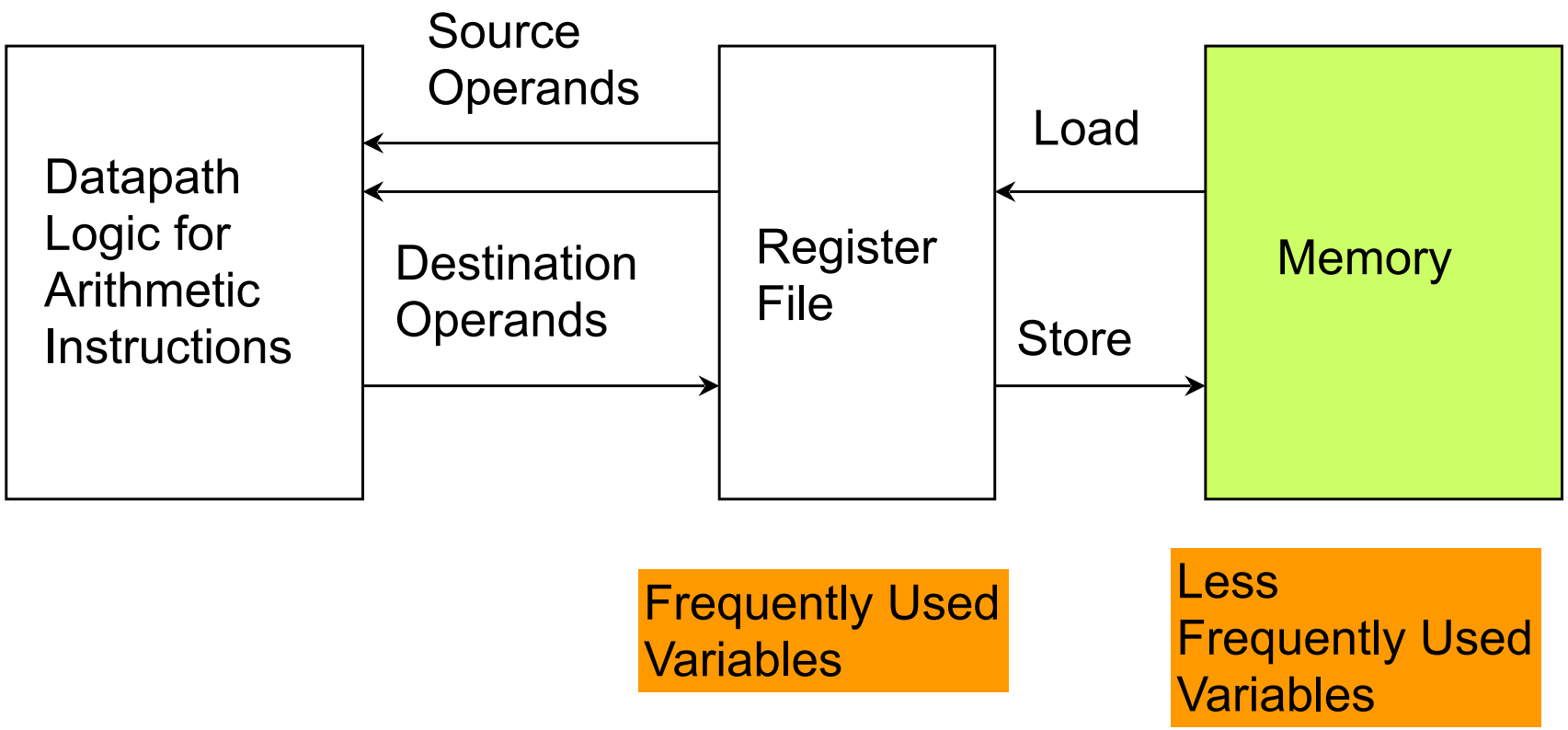
- Will be provided with the MIPS Green Card (both pages)
  - Funct field is really part of the opcode
- Focuses on lecture notes, sample code and homeworks
- Types of questions:
  - Convert from C or an English description of code requirements to MIPS assembly language
    - Arithmetic, if-then-else, array indexing and looping
  - Assemble MIPS assembly language into machine code
    - Determine the numeric opcode/funct, registers, shift amount, immediate / address fields
    - Include load/stores, arithmetic, conditional branches
    - Exclude pseudo-instructions
    - Primarily use page 1 of the MIPS Green Card
  - Disassemble binary instructions into MIPS assembly language
    - Decode the symbolic opcode, register names, immediate values, …
    - Primarily use page 2 of the MIPS Green Card
  - Add instructions to assembly language functions in order to preserve/restore registers on the stack
    - Including use of `$fp`
  - Know how branch and jump instructions alter the program counter
  - Know how to load 16-bit and 32-bit constant values into registers
  - Know the difference between pseudo-instructions and real instructions

Test 1 Logistics

- Given in class during our normal lecture time on Tuesday Feb 21st
  - Test quiz will be available on Canvas at 11:00am
    - Quiz format will be similar to the homeworks
    - Formatting and documentation coding rules apply
    - Open book
- **You must have an accommodation or permission from me to take the exam remotely**
- Quiz must be completed by 12:15pm unless you have an extra-time accommodation
  - People with an extra-time accommodation must take the exam remotely in a location of their choosing

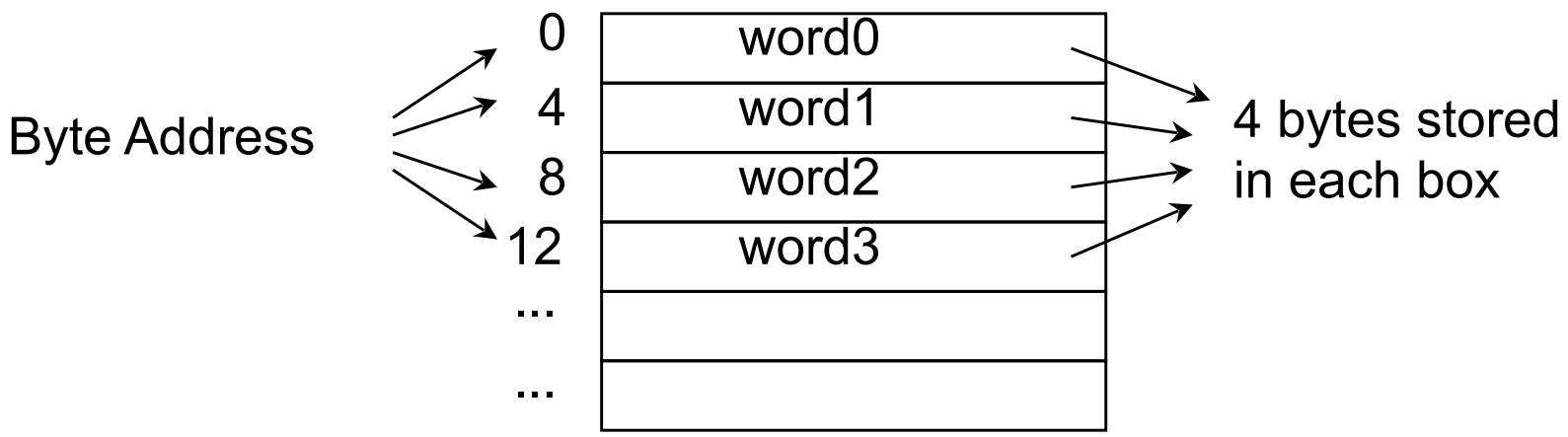
Memory Operands

- Compiler will associate variables with registers
- If more variables than registers, use memory to *spill* variables



Memory Operands

- Memory is byte addressed
  - Each address identifies an 8-bit byte
  - MIPS uses 32-bit addresses
- Words are aligned in memory
  - Address must be a multiple of 4



Instructions that access Memory

- MIPS has two basic data transfer instructions for accessing memory (assume `$s3` holds 24_{10})

  - `lw  $t0, 4($s3)  # load word from byte addr 24 + 4`
  - `sw  $t0, 8($s3)  # store word at byte addr 24 + 8`

- The memory address is formed by summing the constant portion of the instruction and the contents of the second register

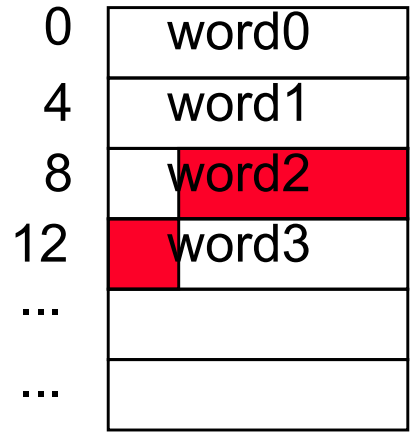
Word-aligned addresses

- If $s3 holds 5, what is the effect of the following instruction ?

```
lw    $t0, 4($s3)
```

4 + $s3 = 9

doesn't work !



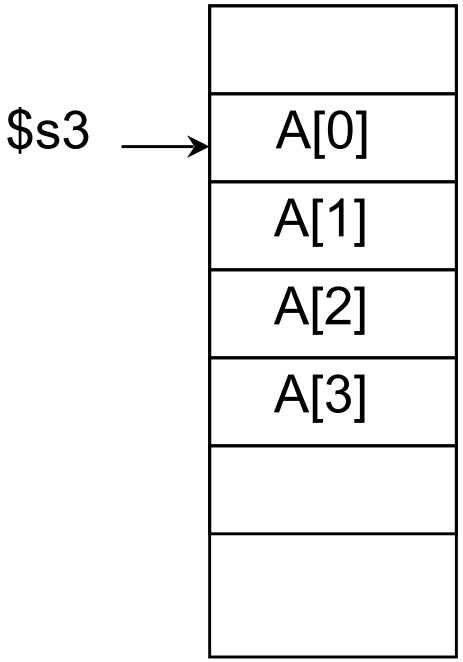
This is not a <u>word-aligned</u> address
(i.e. an address byte which is a multiple of 4)

This instruction will not complete
The MIPS processor will generate an *exception*
The physical memory must be written/read at word boundaries
(The MIPS simulator will complain as well)

Compiling with Loads and Stores

- Assuming variable b is stored in `$s2` and that the base address of array A is in `$s3`, what is the MIPS assembly code for the C statement

```
A[8] = A[2] - b
```




```
lw$     t0, 8($s3)
sub     $t0, $t0, $s2
sw      $t0, 32($s3)
```

An addition involving memory

- Find the sum of the words stored at byte address 0x100 and 0x104 and store the result in $s0

```
add   $reg, $reg, $reg
addi  $reg, $reg, constant
lw    $reg, offset($reg)
```

An addition involving memory

- Find the sum of the words stored at byte address 0x100 and 0x104 and store the result in $s0

```
add   $reg, $reg, $reg
addi  $reg, $reg, constant
lw    $reg, offset($reg)
```

```
addi   $s0, $0, 0x100        # load base address
lw     $s1, 4($s0)           # load word at 0x104
lw     $s0, 0($s0)           # load word at 0x100
add    $s0, $s1, $s0
```

create a base address only once and
form the actual addresses by adding an offset

Compiling with a Variable Array Index

- Assume:
  - variables `b`, `c`, and `i` are in `$s1`, `$s2`, and `$s3`
  - base address of array A is in register `$s4`

```
c = A[i] - b
```

1. Form address for `A[i]`
   `= base-addr of A[0]+ 4*i`
2. Load `A[i]`
3. Subtract `b` and store result in `c`

```
sll     $t1, $s3, 2
add     $t1, $s4, $t1
lw      $t0, 0($t1)
sub     $s2, $t0, $s1
```

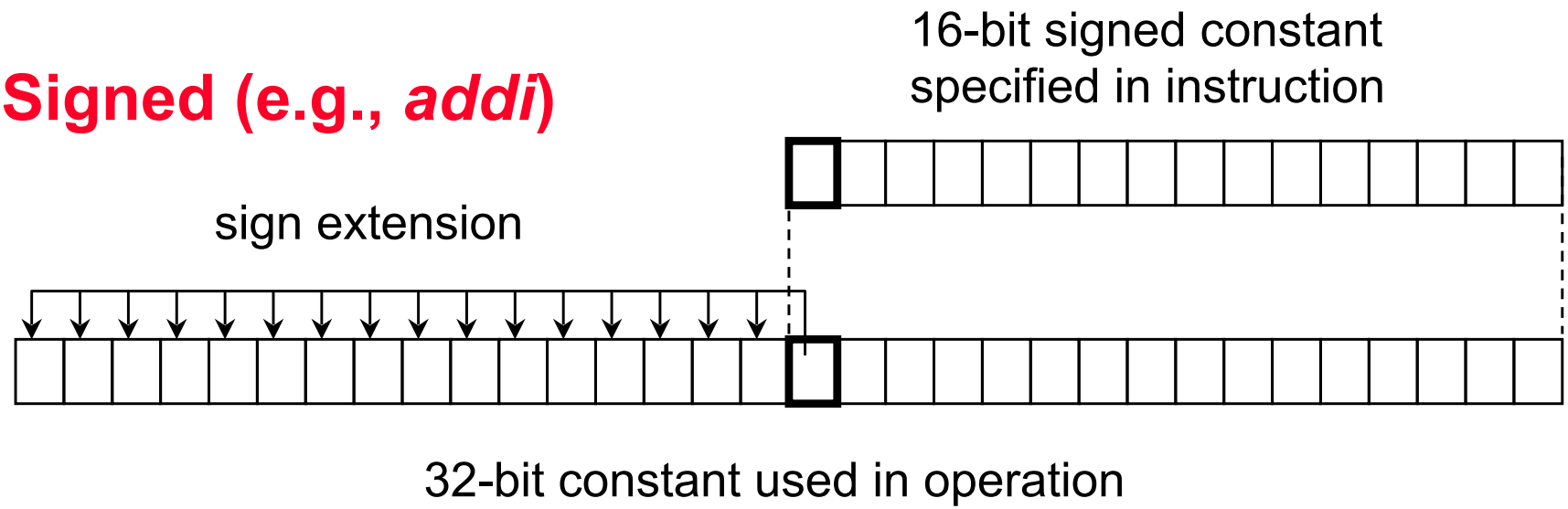
Dealing with Constants

- Exercise: Assume `$s0` is used to store variable `b`. compile into MIPS assembly.

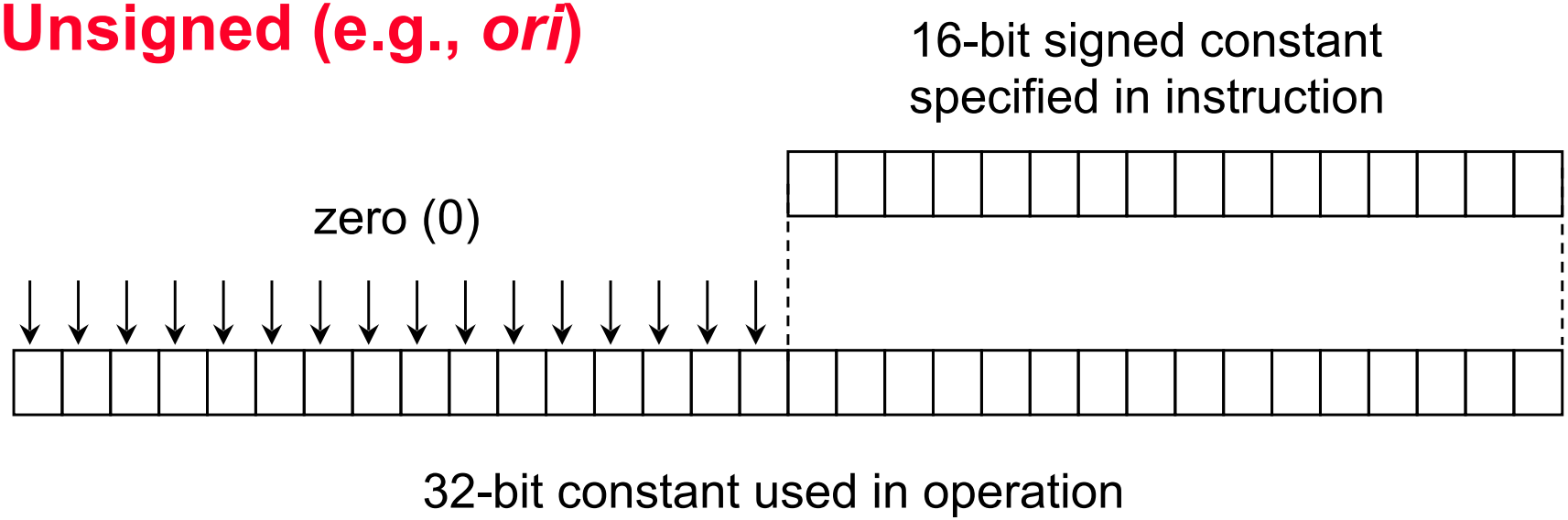
```
int b;

b = 20;
```

Signed and Unsigned Immediate Constants

- **Signed (e.g., *addi*)**

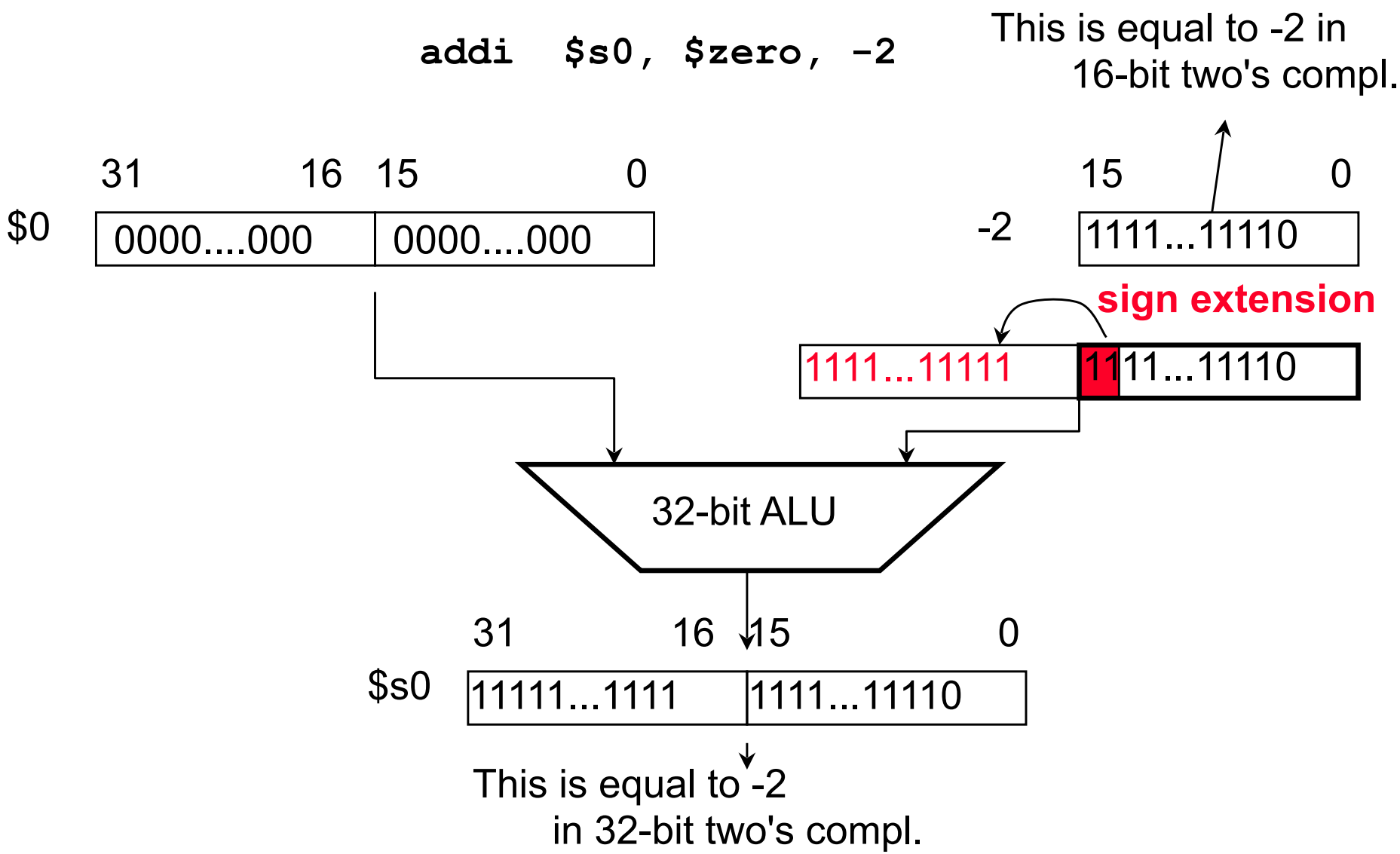


- **Unsigned (e.g., *ori*)**



A closer look at addi

- What is the bit pattern in $s0 after this instruction:



Loading 32-bit constants

- Initialize $s0 to 0x1FFFF using `ori`

```
ori  $reg, $reg, constant
lui  $reg, constant
```

```
lui  $s0, 0x1            # load upper part
ori  $s0, $s0, 0xffff    # load lower half
```

This is how a compiler will load 32-bit constants into registers.

Assembling instructions


```
lw   $s0, 40($s1)
```

Assembling instructions


```
lw    $s0, 40($s1)
```

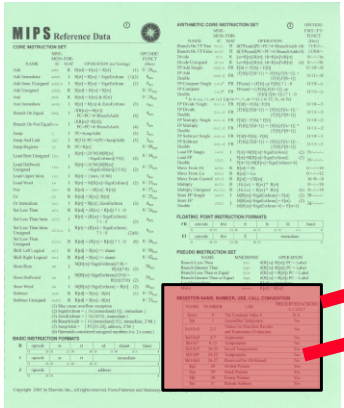
0

Find out about the instruction encoding type

Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}

OP	RS	RT	Immediate

Assembling instructions




```
lw   $s0, 40($s1)
```

1

Registers are encoded by their number (from the table on right bottom of card). Constants are directly encoded.

Register numbers

- $t0 – $t7 are reg's 8 – 15
- $t8 – $t9 are reg's 24 – 25
- $s0 – $s7 are reg's 16 – 23

17 16 40

OP	RS	RT	Immediate

Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}

Assembling instructions


```
lw    $s0, 40($s1)
```

Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}

2 Each instruction has an opcode

23hex 17 16 40

OP	RS	RT	Immediate

Assembling instructions

```
lw    $s0, 40($s1)
```

Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2) 23_{hex}

23hex 17 16 40

3

Arrange fields and extract hex

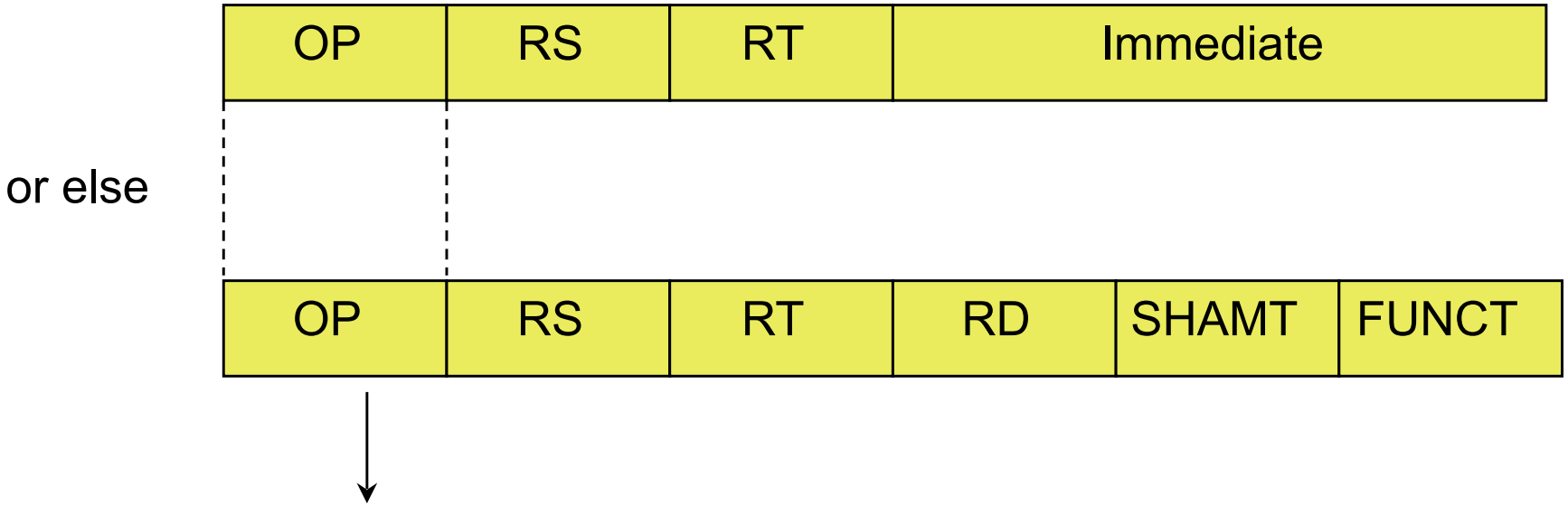
OP	RS	RT	Immediate
100011	10001	10000	0000000000101000

0x8e300028

Dis-assembling Instructions

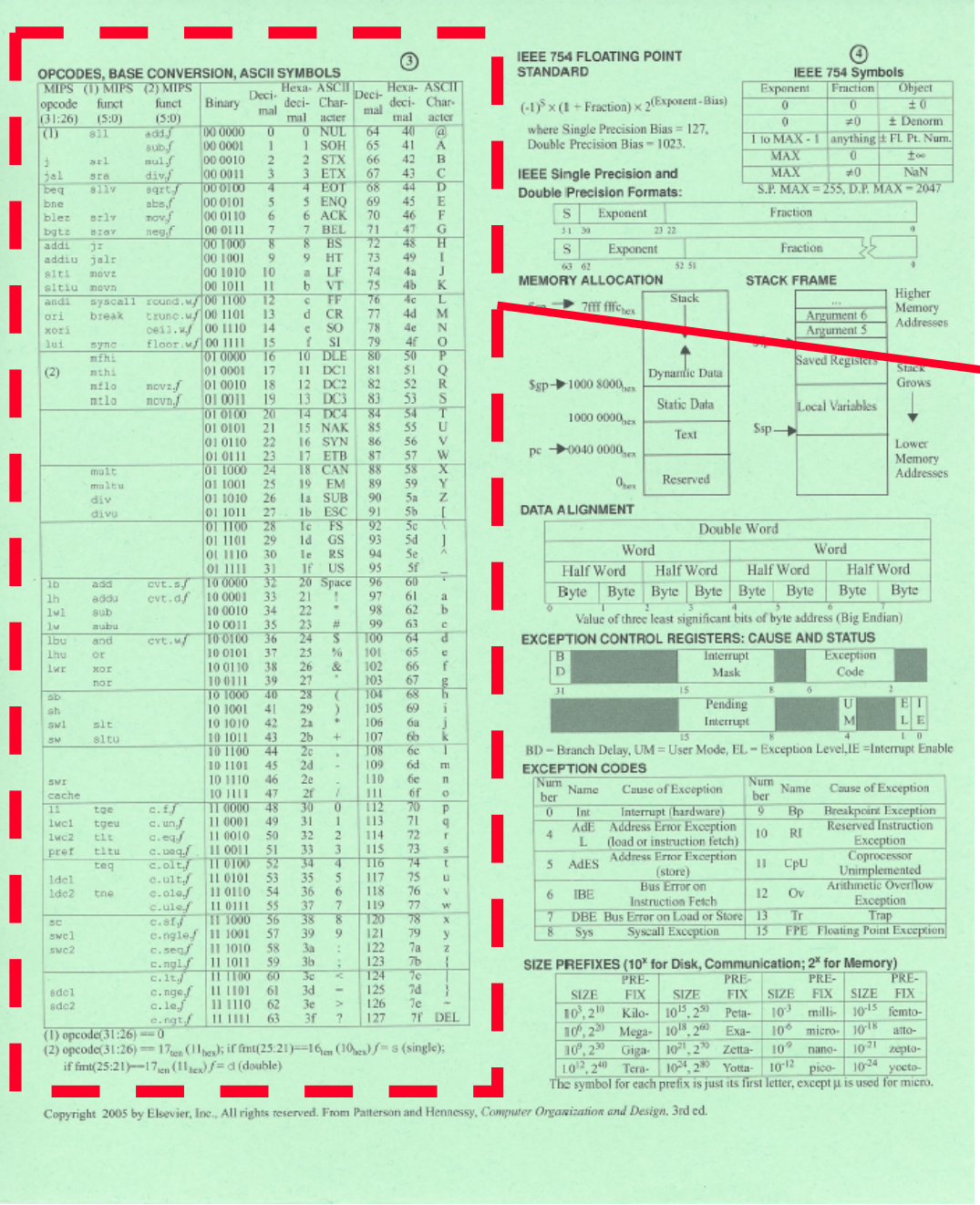
- Find the assembly code for 0x00c23021

What is the instruction format? It could be



The opcode makes the difference.
Start right here.

Use the back of the MIPS refcard



③

OPCODES, BASE CONVERSION, ASCII SYMBOLS

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Decimal	Hexadecimal	ASCII Character	Decimal	Hexadecimal	ASCII Character
(1)	sll	add.f	00 0000	0	0	NUL	64	40	@
		sub.f	00 0001	1	1	SOH	65	41	A
j	srl	mul.f	00 0010	2	2	STX	66	42	B
jal	sra	div.f	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.f	00 0100	4	4	EOT	68	44	D
bne		abs.f	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.f	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.f	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.w.f	00 1100	12	c	FF	76	4c	L
ori	break	trunc.w.f	00 1101	13	d	CR	77	4d	M
xori		ceil.w.f	00 1110	14	e	SO	78	4e	N
lui	sync	floor.w.f	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.f	01 0010	18	12	DC2	82	52	R
	mtlo	movn.f	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.f	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.f	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.f	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.f.f	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.f	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.f	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.f	11 0011	51	33	3	115	73	s
	teq	c.olt.f	11 0100	52	34	4	116	74	t
ldc1		c.ult.f	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.f	11 0110	54	36	6	118	76	v
		c.ule.f	11 0111	55	37	7	119	77	w
sc		c.sf.f	11 1000	56	38	8	120	78	x
swc1		c.ngle.f	11 1001	57	39	9	121	79	y
swc2		c.seq.f	11 1010	58	3a	:	122	7a	z
		c.ngl.f	11 1011	59	3b	;	123	7b	{
		c.lt.f	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.f	11 1101	61	3d	=	125	7d	}
sdc2		c.le.f	11 1110	62	3e	>	126	7e	~
		c.ngt.f	11 1111	63	3f	?	127	7f	DEL

(1) opcode(31:26) = = 0
(2) opcode(31:26) = = 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) f = s (single); if fmt(25:21)==17_{ten} (11_{hex}) f = d (double)

IEEE 754 FLOATING POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Single Precision Bias = 127, Double Precision Bias = 1023.

④

IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

IEEE Single Precision and Double Precision Formats:

S	Exponent	Fraction
31	30 … 23	22 … 0

S	Exponent	Fraction
63	62 … 52	51 … 0

MEMORY ALLOCATION

$sp → 7fff fffc$ hex
$gp → 1000 8000$ hex
$1000 0000$ hex
pc → $0040 0000$ hex
0 hex

Stack
Dynamic Data
Static Data
Text
Reserved

STACK FRAME

...
Argument 6
Argument 5
Saved Registers
Local Variables
$sp

Higher Memory Addresses
Stack Grows
Lower Memory Addresses

DATA ALIGNMENT

Double Word							
Word				Word			
Half Word		Half Word		Half Word		Half Word	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS

BD		Interrupt Mask		Exception Code	
31		15 … 8		6 … 2	

	Pending Interrupt		UM		EL	IE
	15 … 8		4		1	0

BD = Branch Delay, UM = User Mode, EL = Exception Level, IE = Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^x for Disk, Communication; 2^x for Memory)

SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX	SIZE	PREFIX
10^3, 2^{10}	Kilo-	10^{15}, 2^{50}	Peta-	10^{-3}	milli-	10^{-15}	femto-
10^6, 2^{20}	Mega-	10^{18}, 2^{60}	Exa-	10^{-6}	micro-	10^{-18}	atto-
10^9, 2^{30}	Giga-	10^{21}, 2^{70}	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
10^{12}, 2^{40}	Tera-	10^{24}, 2^{80}	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.

Disassembly Table

Use the back of the MIPS refcard

OPCODES, BASE CONVERSION, ASCII SYMB

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal
(1)	sll	add.*f*	00 0000	0	0
		sub.*f*	00 0001	1	1
j	srl	mul.*f*	00 0010	2	2
jal	sra	div.*f*	00 0011	3	3
beq	sllv	sqrt.*f*	00 0100	4	4
bne		abs.*f*	00 0101	5	5
blez	srlv	mov.*f*	00 0110	6	6
bgtz	srav	neg.*f*	00 0111	7	7
addi	jr		00 1000	8	8
addiu	jalr		00 1001	9	9
slti	movz		00 1010	10	a
sltiu	movn		00 1011	11	b
andi	syscall	round.w.*f*	00 1100	12	c
ori	break	trunc.w.*f*	00 1101	13	d
xori		ceil.w.*f*	00 1110	14	e
lui	sync	floor.w.*f*	00 1111	15	f

col3 = R-type function opcode = 17

col2 = R-type function code when opcode = 0

col1 = opcode (I-type and R-type)

Dis-assembling Instructions

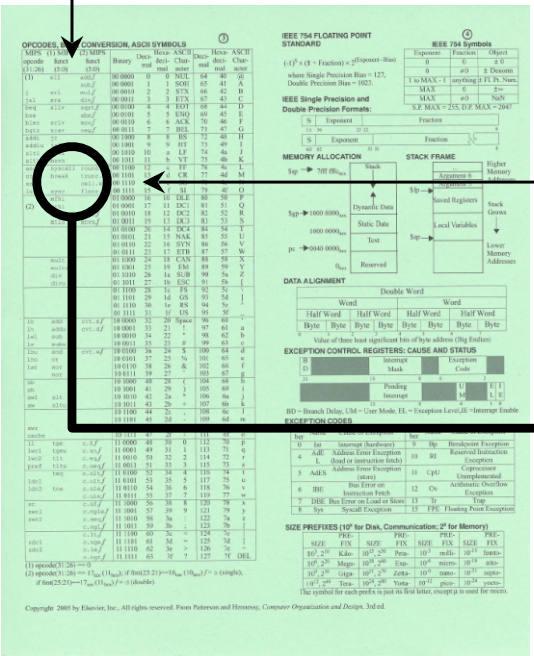
- Find the assembly code for 0x00c23021

The first 6 bits are zero (opcode).
The last 6 bits are 0x21 (function code).

0	RS	RT	RD	SHAMT	21

Look in column 2 (opcode = 0)

Look in row 0x21 (function)



We find .. addu !

The rest is easy ...

- Find the assembly code for 0x00c23021

in binary: 0000 0000 1100 0010 0011 0000 0010 0001
split in fields: 000000 00110 00010 00110 00000 100001

0	6	2	6	0	21
	RS	RT	RD	SHAMT	

```
addu  $6, $6, $2
```

or

0x00c23021 ⇨

```
addu  $a2, $a2, $v0
```

Recap: 3 Types of Branch Instructions

1. jump using 32-bit target specified in register (R-format)

```
jr  $s0
```

2. jump using 26-bit immediate target (J-format)

```
j  0x50401   OR   j  LABEL
```

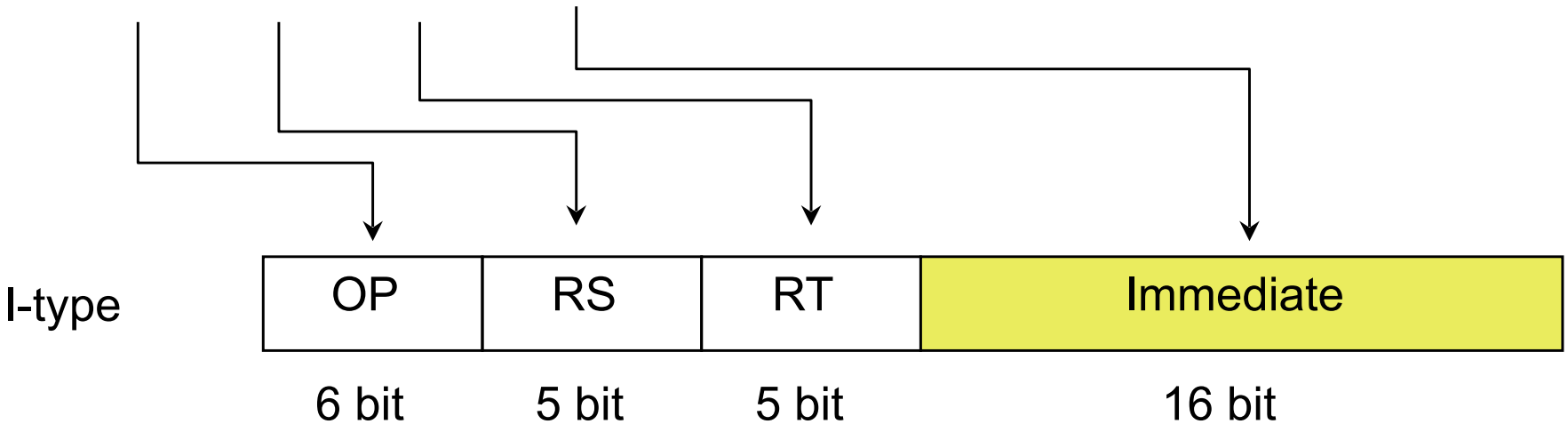
3. branch using 16-bit signed immediate target (I-format)

```
beq  $s0, $s1, LABEL
```

- Branch targets are evaluated relative to next-PC

Conditional branches have I-format

```
bne  $s0, $s1, Label
beq  $s0, $s1, Label
```



- Branch Target must be encoded in 16-bit constant
- Uses *signed offset* relative to next-PC

If-then-else statements

- Implement the if-then-else statement

```
if (i != j)
      h = i - j;
else
      h = i + j;
```

Assume:

i	$s0
j	$s1
h	$s2

```
        beq   $s0, $s1, Else
        sub   $s2, $s0, $s1
        beq   $0, $0, Endif
Else:   add   $s2, $s0, $s1
Endif:  ..
```

While loops

- Implement the while loop

```
while (i != j)
  i = i + 1;
```

Assume:

i	$s0
j	$s1

```
Whileloop:  beq   $s0, $s1, Endloop
            addi  $s0, $s0, 1
            beq   $s0, $s0, Whileloop
Endloop:    …
```

For loops

- A for-loop can be written as a while-loop

```
for (i = 0; i<10; i++) {
   .. bodyoftheloop
}
```

```
i = 0;
while (i<10) {
   .. bodyoftheloop
   i = i + 1;
}
```

Assembly

Comparison-set instructions

- slt is an arithmetic operation that sets a Boolean value.

```
slt    $rd, $rs, $rt
# if $rs <signed $rt, then $rd = 1, else $rd = 0

slti   $rt, $rs, imm
# if $rs <signed sign-extended(imm), then $rt = 1, else $rt = 0
```

Comparison-set instructions

- `slt rd, rs, rt`
  - if (rs < rt) rd = 1; else rd = 0;
- `slti rt, rs, constant`
  - if (rs < constant) rt = 1; else rt = 0;

- Use in combination with beq, bne

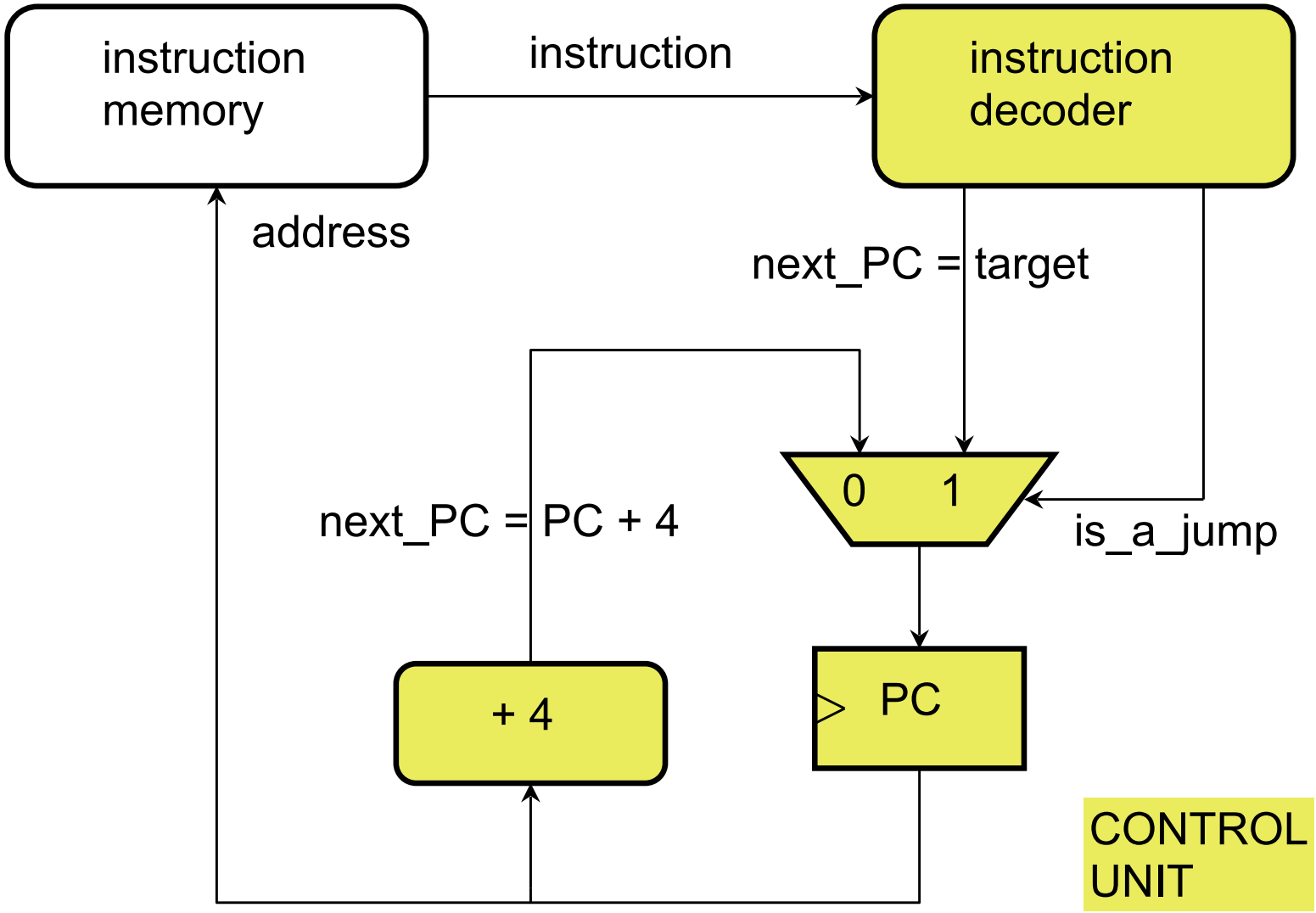
```
slt  $t0, $s1, $s2  # if ($s1 < $s2)
bne  $t0, $zero, L  #   branch to L
```

Stored-Program Concept

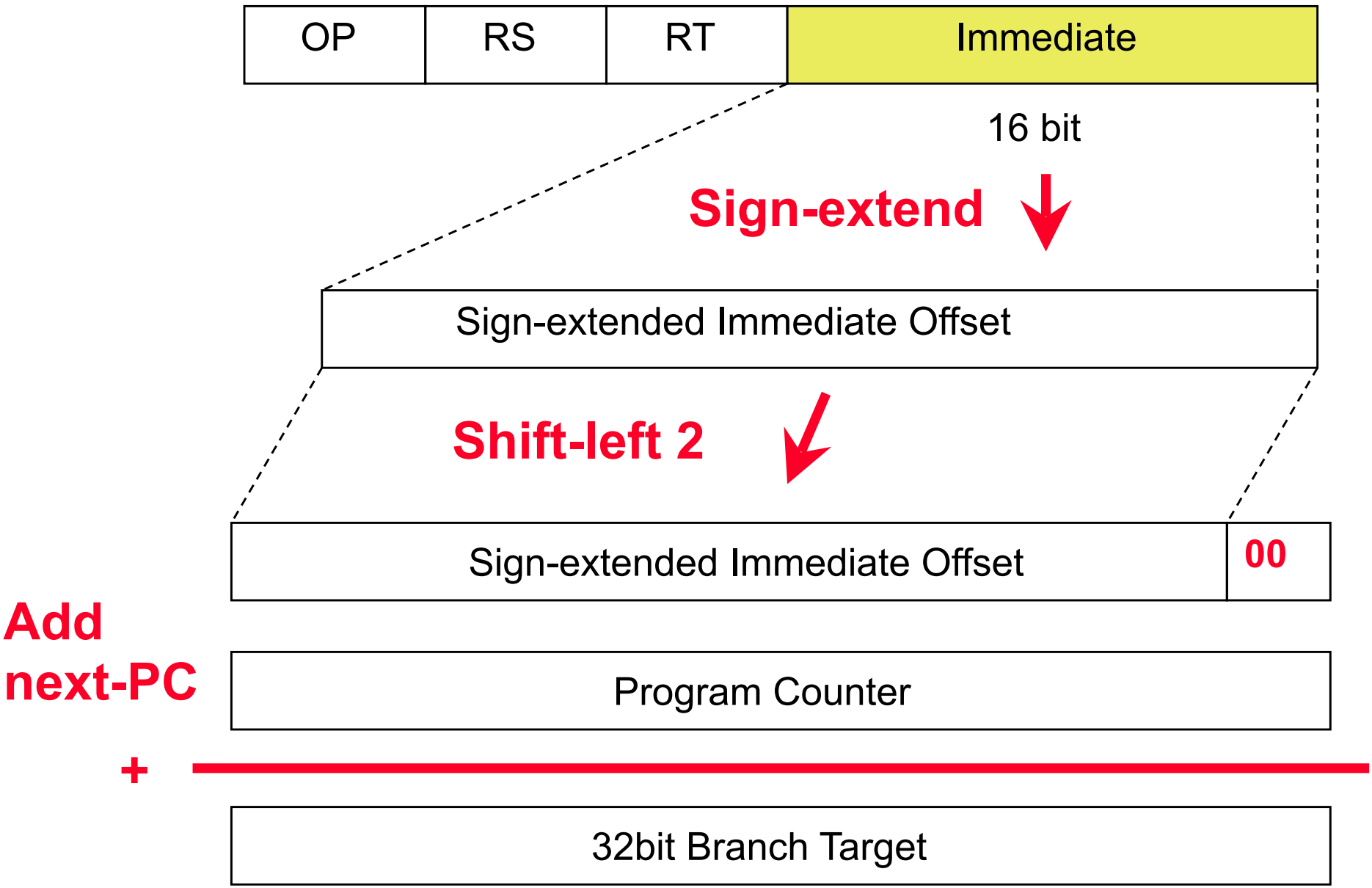
- The program (32-bit machine codes) is stored in the *instruction memory*
  - Later on, we will see that the instruction memory is the same as the data memory
  - For now, think of them as two distinct memories
- Just like data memory, we have 32-bit address for instruction memory
- The *Program Counter (PC)* stores the address of the next instruction that must be fetched and executed
- The assembler will convert the *Label* in your instruction into offset relative to PC for the conditional branch instructions

Jump mechanism

- Here is what happens inside the control unit of the processor



Determining the target of branches



- Remember that PC already points to the next instruction when executing the conditional branch instruction.

- Therefore, offset must be relative to next instruction.

- Also, offset will be later multiplied by 4 (unlike `lw/sw`)

Jump

- Changes PC to a new *target*

target

```
j   0x504
```

- Target = new value of PC after jump completes
- Target must be a word-aligned address

Jump Instruction Format

- Jump instruction includes target as constant

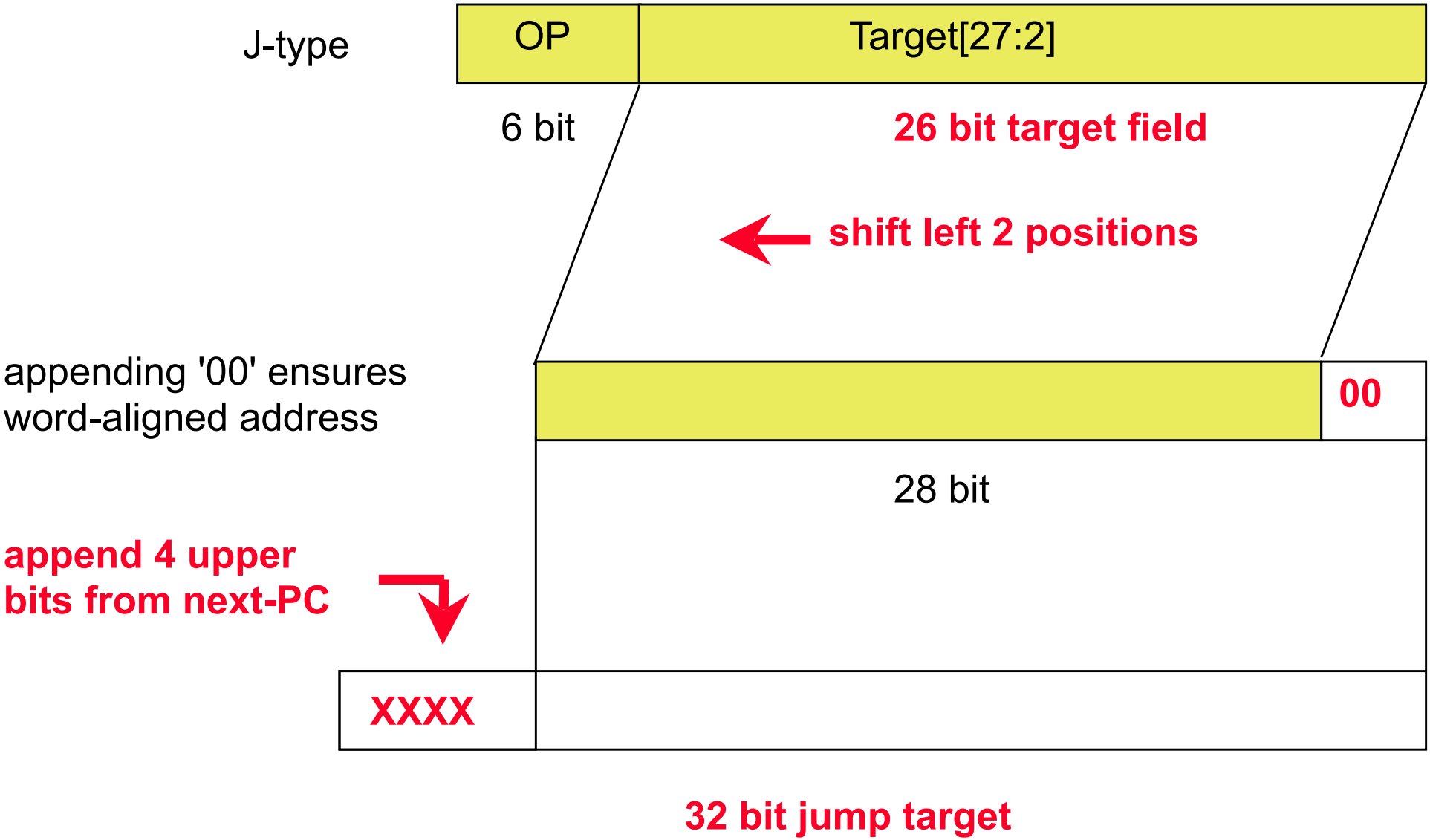
J-type	OP	Target[27:2]
	6 bit	26 bit

J-type instructions are different from R-type and I-type:

R-type	OP	RS	RT	RD	SHAMT	FUNCT
	6 bit	5 bit	5 bit	5 bit	5 bit	6 bit

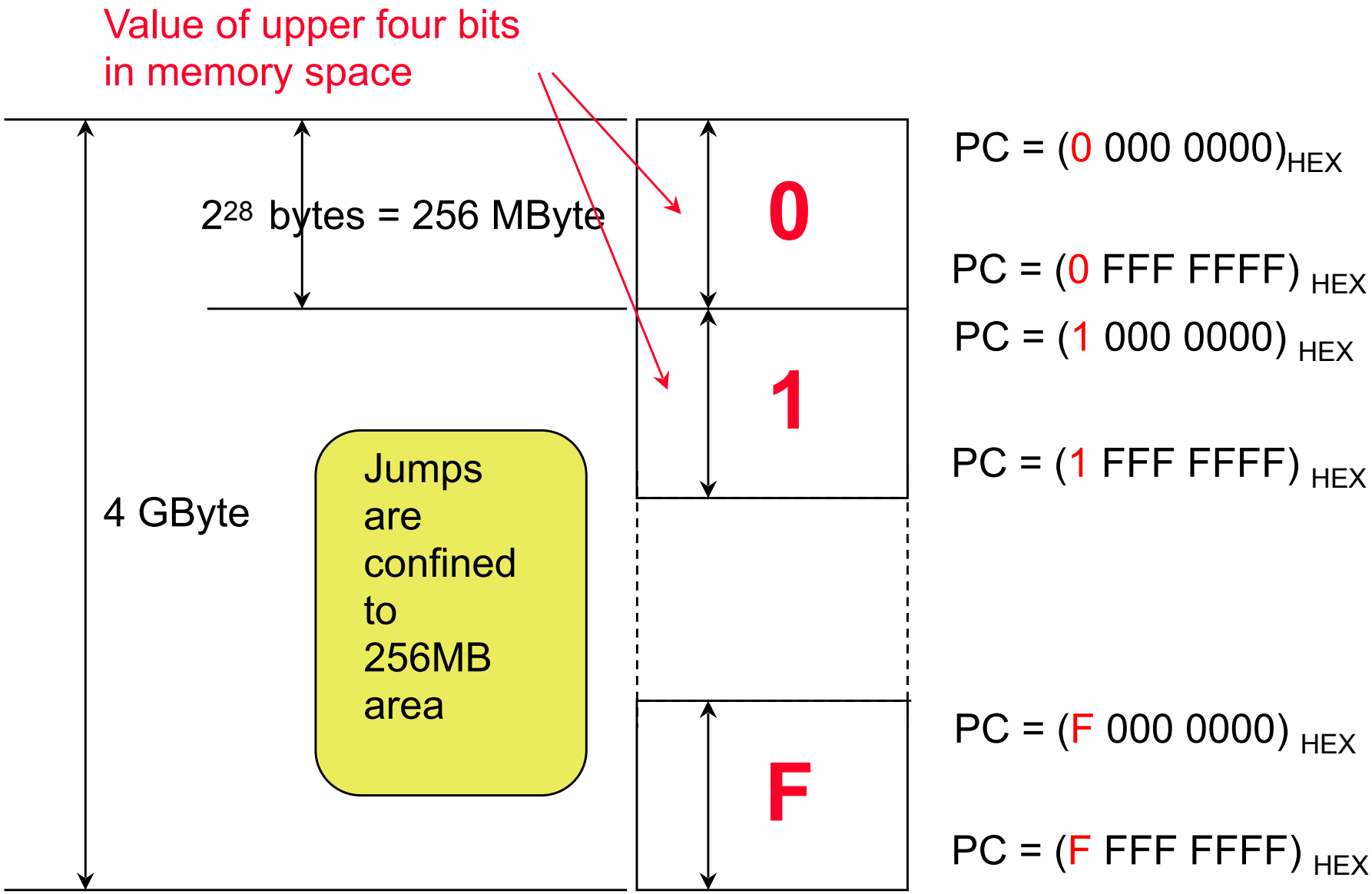
I-type	OP	RS	RT	Immediate
	6 bit	6 bit	6 bit	16 bit

How to encode a 32-bit target in J instruction?



Consequence of expansion to 32-bit target

- The upper four bits of an address cannot be specified



Jumps of 32 bits

```
jr   $s0
```

target is determined by value stored in `$s0`

- Write an absolute jump to `0xF8004500`

```
lui   $s0, 0xF800
ori   $s0, 0x4500
jr    $s0
```

Pseudo-instruction branch expansion

- Some branches, like `blt`, are *pseudo instructions*

```
blt    $s0, $s1, label
```

branch if $s0 < $s1

- Assembler expands the above to

```
#if (s0<signed s1) then $at=1 else $at=0
slt  $at, $s0, $s1
bne  $at, $0, label  # real branch instruction
```

- `$at` register is reserved by the assembler for pseudo instructions. **Why not just use `$t0-$t9`?**

Functions

- Simple functions: arguments in `$a0-$a3`, return in `$v0-$v1`, return address in `$ra`
- The stack is used to support 'complicated' functions:
  - Nested functions
  - Recursive functions
  - Callees that use registers of the caller
  - Callees with more than 4 arguments
- The stack pointer points to the top of the stack, and grows downward as the stack expands
- A *stack frame* is the standard layout of stack storage for a function call implemented with a compiler.

Stack operations

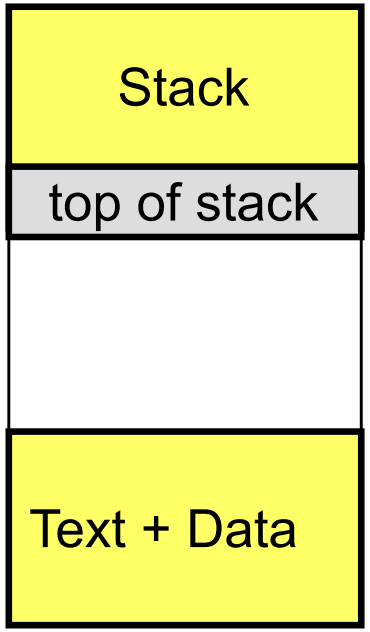
- PUSH = adding elements onto the stack

```
addi  $sp, $sp, -4
sw    $t1, 0($sp)
```

- POP = retrieving elements onto the stack

```
lw    $t1, 0($sp)
addi  $sp, $sp, 4
```

Typical Layout
MIPS Memory



Procedure calls that save registers

caller

1. Preserve `$t0-$t7` before call
2. call callee
3. Restore `$t0-$t7` after call

callee

1. Preserve `$s0-$s7` before executing
2. execute
3. Restore `$s0-$s7` after call

Note that you would only preserve $ti that you actually will be using after the call

Note that you would only preserve $si that you actually will be using in the callee

3. The stack can implement nested functions

X()

1. Push $t0-$t7
2. jal Y
3. Pop $t0-$t7

Y()

1. Push $s0-$s7
2. **Push $ra**
3. jal Z
4. **Pop $ra**
5. Pop $s0-$s7

Z()

1. Push $s0-$s7
2. execute
3. Pop $s0-$s7

Function Prolog and Epilog

- Assume a function *proc* that uses $s0
- Compiler has a standard way of creating code:

function prolog: create stack frame

```
proc:
```

```
        .
        .
    function body
        .
        .
```

function epilog: destroy stack frame

Function Prolog and Epilog

- Assume a function *proc* that uses $s0
- Compiler has a standard way of creating code:

function prolog: create stack frame

```
proc:
• Save old $fp on the stack
• Update $fp with the new frame pointer
• Save $ra on the stack
• Save $s0 on the stack
        .
        .
    function body
        .
        .
```

function epilog: destroy stack frame

Function Prolog and Epilog

- Assume a function *proc* that uses `$s0`
- Compiler has a standard way of creating code:

function prolog: create stack frame

```
proc: addi $sp,$sp,-12   # create 3 spaces on top of stack
      sw   $fp,8($sp)    # save the old frame pointer
      addi $fp,$sp,12    # copy old $sp to $fp
      sw   $ra,-8($fp)   # save ($ra) in 2nd stack element
      sw   $s0,-12($fp)  # save ($s0) in top stack element
         .
         .
      function body
         .
         .
```

function epilog: destroy stack frame

Function Prolog and Epilog

- Assume a function *proc* that uses $s0
- Compiler has a standard way of creating code:

function prolog: create stack frame

```
proc: addi $sp,$sp,-12 # create 3 spaces on top of stack
      sw   $fp,8($sp)  # save the old frame pointer
      addi $fp,$sp,12  # copy old $sp to $fp
      sw   $ra,-8($fp) # save ($ra) in 2nd stack element
      sw   $s0,-12($fp)# save ($s0) in top stack element
         .
         .
      function body
         .
         .
```

function epilog: destroy stack frame

```
      lw   $s0,-12($fp)# put top stack element in $s0
      lw   $ra,-8($fp) # put 2nd stack element in $ra
      lw   $fp,-4($fp) # restore $fp to original state
      addi $sp,$sp, 12 # restore $sp to original state
      jr   $ra         # return from procedure
```