Assembly Language and Computer Architecture Lab

Nguyen Thi Thanh Nga
Dept. Computer engineering
School of Information and Communication Technology

Week 4

- Logical (or bit-wise Boolean) operations in MIPS assembly language.
- Using logical operators
- Shifting data in registers, and the different types of shifts in MIPS.
- Translating assembly language into machine code

Logical operators

- Logical operators

Input		Output				
A	B	AND	OR	NAND	NOR	XOR
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	1	0	1
1	1	1	1	0	0	0

- Bitwise operators (MIPS implements and be called logical operators): AND/OR/NOT/XOR

and operator format 1

- The only real format
- Does a bit-wise **AND** of $\mathbf{R_s}$ and $\mathbf{R_t}$ and stores the result to $\mathbf{R_d}$ register
- The format and meaning are:

  format: **and $\mathbf{R_d}$, $\mathbf{R_s}$, $\mathbf{R_t}$**

  meaning: $\mathbf{R_d \leftarrow R_s}$ **AND** $\mathbf{R_t}$

and operator format 2

- Pseudo instruction
- The 3rd operator is an immediate value, and so this is just a short hand for implementing the **andi** operator.
- Does a bit-wise **AND** of $\mathbf{R_s}$ and an immediate value and stores the result to $\mathbf{R_t}$ register
- The format, meaning, and translation are:

  format: **and $\mathbf{R_t}$, $\mathbf{R_s}$, Immediate**

  meaning: **$\mathbf{R_t} \leftarrow \mathbf{R_s}$ AND Immediate**

  translation: **andi $\mathbf{R_t}$, $\mathbf{R_s}$, Immediate**

and operator format 3

- Pseudo instruction
- Does a bit-wise **AND** of **R_s** and an immediate value and stores the result to **R_s** register
- The format, meaning, and translation are:

format: **and R_s, Immediate**

meaning: **$R_s \leftarrow R_s$ AND Immediate**

translation: **andi R_s, R_s, Immediate**

andi operator

- Real instruction
- Does a bit-wise **AND** of $\mathbf{R_s}$ and an immediate value and stores the result to $\mathbf{R_t}$ register
- The format, meaning, and translation are:

  format: **andi $\mathbf{R_t}$, $\mathbf{R_s}$, Immediate**

  meaning: $\mathbf{R_t \leftarrow R_s}$ **AND Immediate**

andi operator

- The shorthand with a single register also applies to the **andi** instruction.

- The format, meaning, and translation are:

format: **andi R_s, Immediate**

meaning: **$R_s \leftarrow R_s$ AND Immediate**

translation: **andi R_s, R_s, Immediate**

or operator format 1

- The only real format
- Does a bit-wise **OR** of $\mathbf{R_s}$ and $\mathbf{R_t}$ and stores the result to $\mathbf{R_d}$ register
- The format and meaning are:

  format: **or $\mathbf{R_d}$, $\mathbf{R_s}$, $\mathbf{R_t}$**

  meaning: $\mathbf{R_d \leftarrow R_s}$ **OR** $\mathbf{R_t}$

or operator format 2

- Pseudo instruction
- The 3rd operator is an immediate value, and so this is just a short hand for implementing the **ori** operator.
- Does a bit-wise **AND** of $\mathbf{R_s}$ and an immediate value and stores the result to $\mathbf{R_t}$ register
- The format, meaning, and translation are:

  format: **or $\mathbf{R_t}$, $\mathbf{R_s}$, Immediate**

  meaning: **$\mathbf{R_t} \leftarrow \mathbf{R_s}$ OR Immediate**

  translation: **ori $\mathbf{R_t}$, $\mathbf{R_s}$, Immediate**

or operator format 3

- Pseudo instruction
- Does a bit-wise **OR** of **R_s** and an immediate value and stores the result to **R_s** register
- The format, meaning, and translation are:

format: **or R_s, Immediate**

meaning: **$R_s \leftarrow R_s$ OR Immediate**

translation: **ori R_s, R_s, Immediate**

ori operator

- Does a bit-wise **OR** of $\mathbf{R_s}$ and an immediate value and stores the result to $\mathbf{R_t}$ register

- The format, meaning, and translation are:

format: **ori R_t, R_s, Immediate**

meaning: **$R_t \leftarrow R_s$ OR Immediate**

ori operator

- The shorthand with a single register also applies to the **ori** instruction.

- The format, meaning, and translation are:

format: **ori R_s, Immediate**

meaning: **$R_s \leftarrow R_s$ OR Immediate**

translation: **ori R_s, R_s, Immediate**

xor operator format 1

- The only real format
- Does a bit-wise **XOR** of $\mathbf{R_s}$ and $\mathbf{R_t}$ and stores the result to $\mathbf{R_d}$ register
- The format and meaning are:

  format: **xor $\mathbf{R_d}$, $\mathbf{R_s}$, $\mathbf{R_t}$**

  meaning: $\mathbf{R_d} \leftarrow \mathbf{R_s}$ **XOR** $\mathbf{R_t}$

xor operator format 2

- Pseudo instruction
- The 3rd operator is an immediate value, and so this is just a short hand for implementing the **xori** operator.
- Does a bit-wise **XOR** of $\mathbf{R_s}$ and an immediate value and stores the result to $\mathbf{R_t}$ register
- The format, meaning, and translation are:

  format: **xor $\mathbf{R_t}$, $\mathbf{R_s}$, Immediate**

  meaning: **$\mathbf{R_t} \leftarrow \mathbf{R_s}$ XOR Immediate**

  translation: **xori $\mathbf{R_t}$, $\mathbf{R_s}$, Immediate**

xor operator format 3

- Pseudo instruction
- Does a bit-wise **XOR** of **R_s** and an immediate value and stores the result to **R_s** register
- The format, meaning, and translation are:

format: **xor R_s, Immediate**

meaning: **$R_s \leftarrow R_s$ XOR Immediate**

translation: **xori R_s, R_s, Immediate**

xori operator

- Does a bit-wise **OR** of $\mathbf{R_s}$ and an immediate value and stores the result to $\mathbf{R_t}$ register

- The format, meaning, and translation are:

format: **xori R_t, R_s, Immediate**

meaning: **$R_t \leftarrow R_s$ XOR Immediate**

xori operator

- The shorthand with a single register also applies to the **xori** instruction.

- The format, meaning, and translation are:

format: **xori R_s, Immediate**

meaning: **$R_s \leftarrow R_s$ XOR Immediate**

translation: **xori R_s, R_s, Immediate**

not operator

- Does a bit-wise **NOT** (bit inversion) of **R_t**, and stores the result **R_s** register.

- The format, meaning and translation are:

format: **not R_s, R_t**

meaning: **$R_s \leftarrow$ NOT(R_t)**

translation: **nor R_s, R_t, $zero**

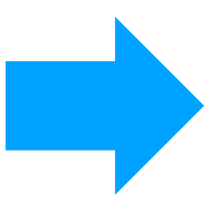
Using logical operators

- Storing immediate values in registers
- Converting a character from upper case to lower case
- Reversible operations with XOR

Storing immediate values in registers

- The **li** instruction is translated into:

addui R_d, $zero, Immediate



The addition can take a relatively long time for the propagation of a carry-bit

- If the OR is faster than addition, the **li** instruction can be implemented as:

ori R_d, $zero, Immediate

Converting a character from upper case to lower case

- In ASCII, the difference between upper case and lower case letters is 0x20.

- To convert uppercase letters to lowercase, you can add 0x20 to an uppercase.

- If the letter is already lowercase, what will happen?

Decimal - Binary - Octal - Hex – ASCII Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	“	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	‘	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

Program 4.1 Converting a character from upper case to lower case

```
7   .data
8   output1: .asciiz "\nPlease input a character: "
9   output2: .asciiz "\nUppercase to lowercase: result when anding with 0x20: "
10  output3: .asciiz "\nUppercase to lowercase: result when oring with 0x20: "
11
12  .text
13  .globl main
14  main:
15          ori $v0,$zero,4
16          la $a0,output1
17          syscall
18
19          ori $v0,$zero,12
20          syscall
21          move $s0,$v0
22
23          ori $v0, $zero,4
24          la $a0,output2
25          syscall
26
27          or $t0,$s0,$zero
28          addi $a0,$t0,0x20
29          ori $v0,$zero,11
30          syscall
31
32          ori $v0,$zero,4
33          la $a0,output3
34          syscall
35
36          ori $a0,$s0,0x20
37          ori $v0,$zero,11
38          syscall
39
40          ori $v0,$zero,10
41          syscall
```

- Input the program
- Assemble and run the program
- Explain the result
- Add comment to explain the purpose of each code block

Bitwise XOR operation

- Bitwise XOR operation can be used to invert the a value and translate it back to the original value.

A	1	1	0	0	0	1	1	1
B	1	0	1	0	1	0	1	0
A XOR B	0	1	1	0	1	1	0	1
(A XOR B) XOR B	1	1	0	0	0	1	1	1

Program 4.2
Reversible operations with XOR

```
6   .data
7   output1: .asciiz "\nPlease input a value: "
8   output2: .asciiz "\nInput value: "
9   output3: .asciiz "\nResult after first XOR: "
10  output4: .asciiz "\nResult after the second XOR: "
11
12  .text
13  .globl main
14  main:
15          ori $v0,$zero,4
16          la $a0,output1
17          syscall
18
19          ori $v0,$zero,5
20          syscall
21          move $s0,$v0
22
23          ori $v0,$zero,4
24          la $a0,output2
25          syscall
26
27          ori $v0,$zero,34
28          move $a0,$s0
29          syscall
30
```

```
31          ori $v0,$zero,4
32          la $a0,output3
33          syscall
34
35          xori $s0,$s0,0xffffffff
36          move $a0,$s0
37
38          ori $v0,$zero,34
39          syscall
40
41          ori $v0,$zero,4
42          la $a0,output4
43          syscall
44
45          xori $s0,$s0,0xffffffff
46          move $a0,$s0
47
48          ori $v0,$zero,34
49          syscall
50
51          ori $v0,$zero,10
52          syscall
```

- Input the program
- Assemble and run the program
- Explain the result
- Add comment to explain the purpose of each code block

Excercise

- Please write a program that:
  - Ask the user to input 2 integers a and b
  - Print on the console 2 integers in the form of hexa
  - Do a XOR b and print out the result
  - XOR the result again with b and print out the result

Shift Operations

- Shift allow bits to be moved around inside of a register.
- There are 2 directions of shifts, a right shift and a left shift.
- The **right shift** moves all bits in a register some specified number of spaces to the right
- The **left shift** moves all bits in a register some specified number of spaces to the left.

Shift Operations

- Left shift:
  - Logic shift: uses zeros (0) to replace the spaces.
- Right shift:
  - Logic shift: uses zeros (0) to replace the spaces
  - Arithmetic shift: replaces the spaces with the high order (left most) bit
- Circular shift (called a rotate in MIPS): shifts in the bit that was shifted out on the opposite side of the value.

sll (shift left logical) operator

- Shifts the value in $\mathbf{R_t}$ shift amount (**shamt**) bits to the left, replacing the shifted bits with 0's, and storing the results in $\mathbf{R_d}$.

- The numeric value in this instruction is not an immediate value, but a shift amount.

- Shift values are limited to the range 0…31

- The format and meaning are:

  format: **sll $\mathbf{R_d}$, $\mathbf{R_t}$, shamt**

  meaning: $\mathbf{R_d \leftarrow R_t << shamt}$

sllv (shift left logical variable) operator

- Shifts the value in **R_t** bits to the left by the number in **R_s**, replacing the shifted bits with 0's and stores the result in **R_d** register.

- The value in **R_s** should be limited to the range 0…31, but the instruction will run with any value.

- The format and meaning are:

  format: **sllv R_d, R_t, R_s**

  meaning: **$R_d \leftarrow R_t << R_s$**

srl (shift right logical) operator

- Shifts the value in **R_t** shift amount (**shamt**) bits to the right, replacing the shifted bits with 0's, and storing the results in **R_d**.

- The format and meaning are:

  format: **srl R_d, R_t, shamt**

  meaning: **$R_d \leftarrow R_t >> shamt$**

srlv (shift right logical variable) operator

- Shifts the value in **R_t** bits to the right by the number in **R_s**, replacing the shifted bits with 0's.

- The value in **R_s** should be limited to the range 0…31, but the instruction will run with any value.

- The format and meaning are:

  format: **srlv R_d, R_t, R_s**

  meaning: **$R_d \leftarrow R_t >> R_s$**

sra (shift right arithmetic) operator

- Shifts the value in $\mathbf{R_t}$ shift amount (**shamt**) bits to the right, replacing the shifted bits with sign bit for the number, and storing the results in $\mathbf{R_d}$.

- The format and meaning are:

  format: **sra R_d, R_t, shamt**

  meaning: $\mathbf{R_d \leftarrow R_t >> shamt}$

srav (shift right arithmetic variable) operator

- Shifts the value in $\mathbf{R_t}$ bits to the right by the number in $\mathbf{R_s}$, replacing the shifted bits the sign bit for the number.

- The value in $\mathbf{R_s}$ should be limited to the range 0…31, but the instruction will run with any value.

- The format and meaning are:

  format: **srav R_d, R_t, R_s**

  meaning: $\mathbf{R_d \leftarrow R_t >> R_s}$

rol (rotate left) pseudo operator

- Shifts the value in **R_t** shift amount (**shamt**) bits to the right, replacing the shifted bits with the bits that were shifted out, and storing the results in **R_d**.

- The format, meaning and translation are:

format: **rol R_d, R_t, shamt**

meaning: **R_d[shamt…0] ← R_t[31…31-shamt+1]**

R_d[31…shamt] ← R_t[31-shamt…0]

translation: **sll $at, $$R_t$, shamt**

srl $$R_d$, $$R_t$, 32-shamt

or $$R_d$, $$R_d$, $at

rol (rotate left) example

For example: **R_t: 00011010**

rol R_d, R_t, 3

	R_t	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**
sll $at, $$R_t$, 3	**$at**	**1**	**0**	**0**	**1**	**0**	**0**	**0**	**0**
srl $$R_d$, $$R_t$, 8-3	**R_d**	**0**	**0**	**0**	**0**	**0**	**1**	**1**	**0**
or $$R_d$, $$R_d$, $at	**R_d**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**

ror (rotate right) pseudo operator

- Shifts the value in **R_t** shift amount (**shamt**) bits to the right, replacing the shifted bits with the bits that were shifted out, and storing the results in **R_d**.

- The format, meaning and translation are:

format: **ror R_d, R_t, shamt**

meaning: **R_d[31-shamt..shamt] ← R_t[31..shamt]**

R_d[31..31-shamt+1] ← R_t[shamt-1..0]

translation: **srl $at, $$R_t$, shamt**

sll $$R_d$, $$R_t$, 32-shamt

or $$R_d$, $$R_d$, $at

ror (rotate right) example

For example: **R_t: 00011010**

ror R_d, R_t, 3

	R_t	**1**	**1**	**0**	**1**	**0**	**0**	**1**	**0**
srl $at, $$R_t$, 3	**$at**	**0**	**0**	**0**	**1**	**1**	**0**	**1**	**0**
sll $$R_d$, $$R_t$, 8-3	**R_d**	**0**	**1**	**0**	**0**	**0**	**0**	**0**	**0**
or $$R_d$, $$R_d$, $at	**R_d**	**0**	**1**	**0**	**1**	**1**	**0**	**1**	**0**

Exercise 4.1

- Write a program which has the following result:

```
Please input an integer value: -12345

Input value in binary:      11111111111111111100111111000111
Result after sll 4 bits:    11111111111111001111110001110000
Result after sllv 4 bits:   11111111111111001111110001110000
Input value in binary:      11111111111111111100111111000111
Result after srl 4 bits:    00001111111111111111110011111100
Result after srlv 4 bits:   00001111111111111111110011111100
Input value in binary:      11111111111111111100111111000111
Result after sra 4 bits:    11111111111111111111110011111100
Result after srav 4 bits:   11111111111111111111110011111100
Input value in binary:      11111111111111111100111111000111
Result after rol 4 bits:    11111111111111001111110001111111
Result after ror 4 bits:    01111111111111111111110011111100
```

Translating Assembly Language into Machine Code

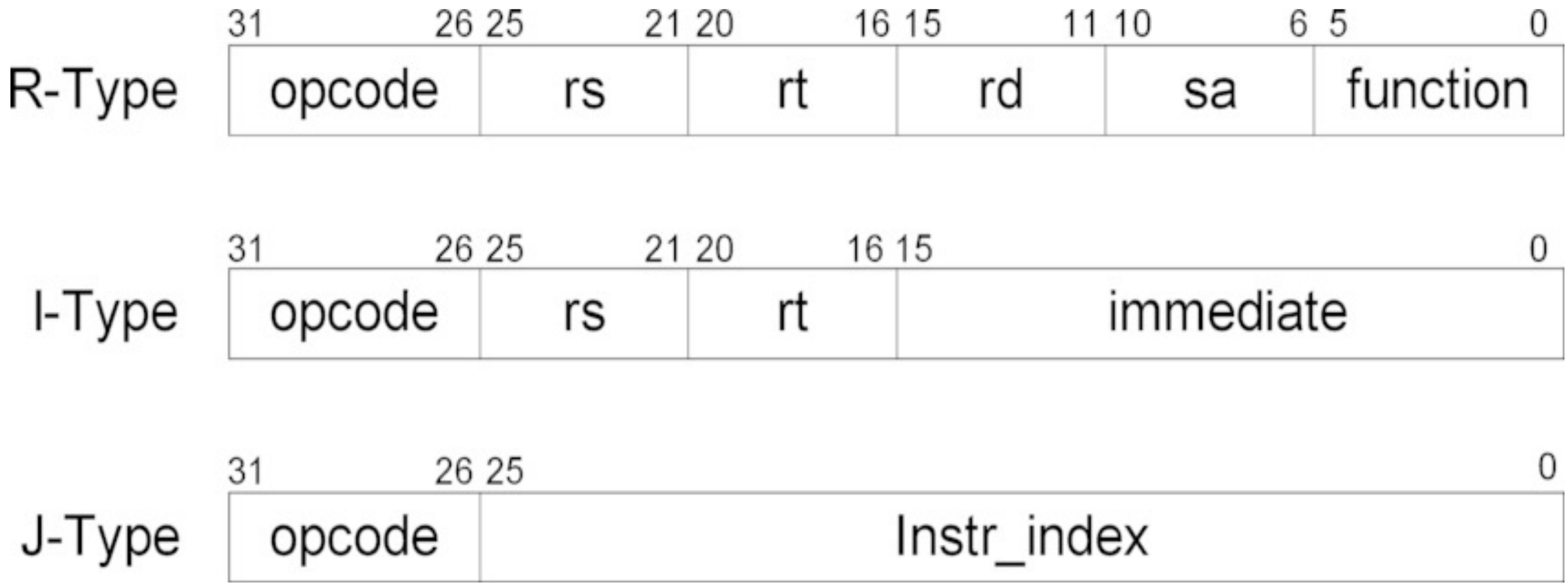
- There are only 3 ways to format instructions in MIPS:

  - The R-format (register): is used when the input values to the ALU come from two registers.

  - The I- format (immediate): is used when the input values to the ALU come from one register and an immediate value

  - The J-format (jump)

Instruction formats



- opcode: a 6 bit code which specifies the operation

http://www-inst.eecs.berkeley.edu/~cs61c/resources/MIPS_Green_Sheet.pdf

Instruction formats

High-level language statement:

```
a = b + c
```

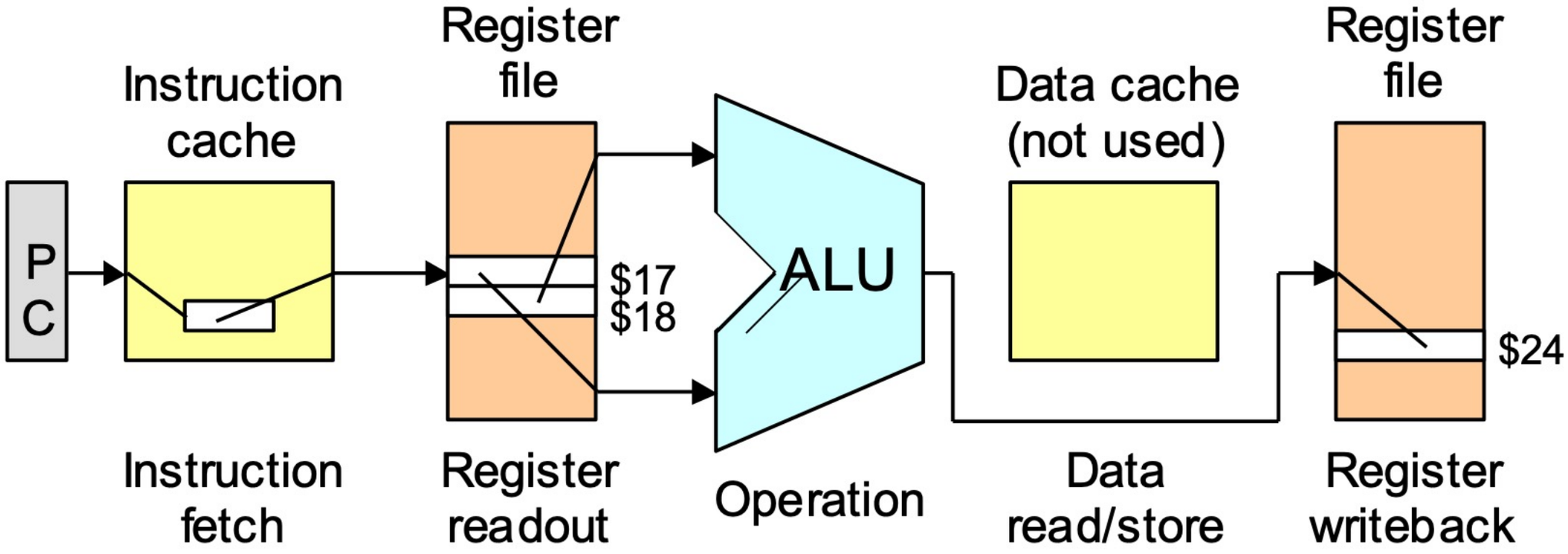
Assembly language instruction:

```
add $t8, $s2, $s1
```

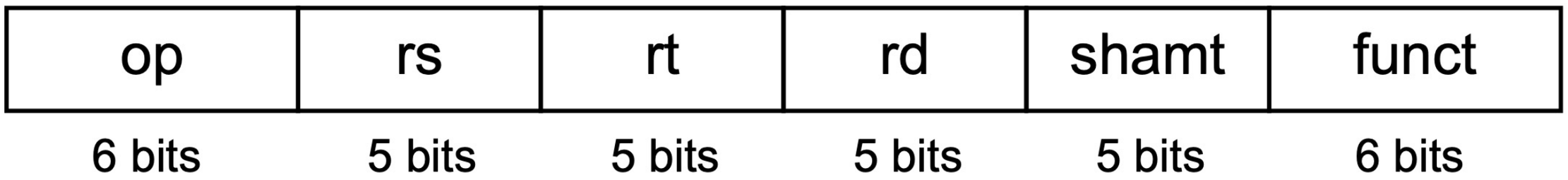
Machine language instruction:

```
000000 10010 10001 11000 00000 100000
```

ALU-type instruction	Register 18	Register 17	Register 24	Unused	Addition opcode



R instruction



- Instruction fields:

  - **op** (operation code - opcode): 000000

  - **rs**: the 1st register in the instruction, and is always an input to the ALU

  - **rt**: the 2nd register in the instruction and the 2nd input to the ALU.

  - **rd**: always is the destination register.

  - **shamt**: The number of bits to shift if the operation is a shift operation, otherwise it is 0. This field is 5 bits wide, which allows for shifting of all 32 bits in the register.

  - **funct**: 6 bits wide, specifies the operation for the ALU

R instruction

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

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

0	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

(0x02324020)

```
sub $s0, $t3, $t5
```

0	$t3	$t5	$s0	0	sub
0	11	13	16	0	34
000000	01011	01101	10000	00000	100010

(0x016D8022)

I instruction

- Used for arithmetic/logical instruction with immediate operators and **load/store** instruction.

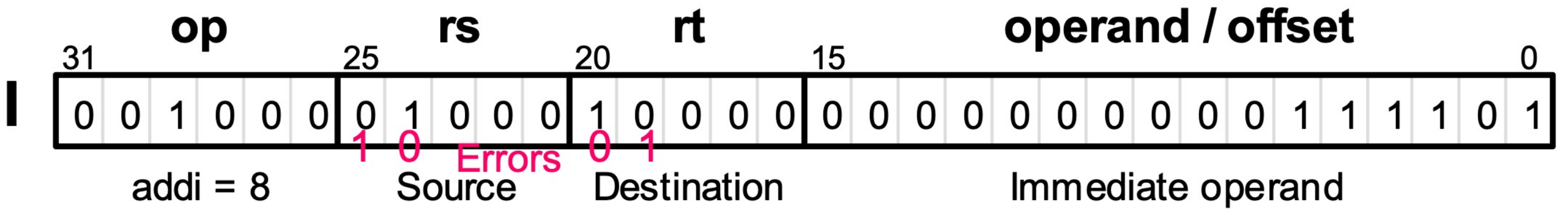
- Instruction fields:
  - **op** (operation code - opcode): specifies the operation
  - **rs**: the 1st register in the instruction, and is always an input to the ALU
  - **rt**: the 2nd register in the instruction and the destination register.
  - **immediate**: value for instructions

Arithmetic/logical instruction with immediate value

- Operations in the range [-32 768, 32 767] or [0x0000, 0xffff] can be set in the immediate field.

```
addi   $t0,$s0,61      # set $t0 to ($s0)+61
andi   $t0,$s0,61      # set $t0 to ($s0)∧61
ori    $t0,$s0,61      # set $t0 to ($s0)v61
xori   $t0,$s0,0x00ff  # set $t0 to ($s0)⊕ 0x00ff
```

For arithmetic instructions, the immediate operand is sign-extended



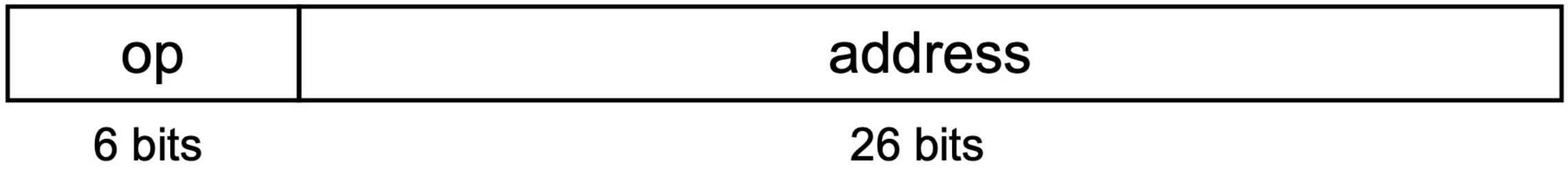
Arithmetic/logical instruction with immediate value

- For **addi**, **lw**, **sw** operators, the register content needs to be added to the immediate value:
  - Register is 32 bits wide
  - Immediate value is 16 bits wide, needs to be expanded to a sign-extended 32-bit type
- For example:

+5 =	0000 0000 0000 0101	16-bit
+5 =	0000 0000 0000 0000 0000 0000 0000 0101	32-bit
-12 =	1111 1111 1111 0100	16-bit
-12 =	1111 1111 1111 1111 1111 1111 1111 0100	32-bit

J instruction

- 26-bit address operator
- Used for jump instructions:
  - **j** (jump): op=000010
  - **jal** (jump and link): op=000011



Translate assembly into machine code

2. Fold bottom side (columns 3 and 4) together

perforation to separate card

MIPS Reference Data

①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	`add`	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	`addi`	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	`addiu`	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	`addu`	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	`and`	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	`andi`	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	`beq`	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	`bne`	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	`j`	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	`jal`	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	`jr`	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	`lbu`	I	R[rt]={24'b0,M[R[rs] +SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	`lhu`	I	R[rt]={16'b0,M[R[rs] +SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	`ll`	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	`lui`	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	`lw`	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	`nor`	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	`or`	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	`ori`	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	`slt`	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$

ARITHMETIC CORE INSTRUCTION SET

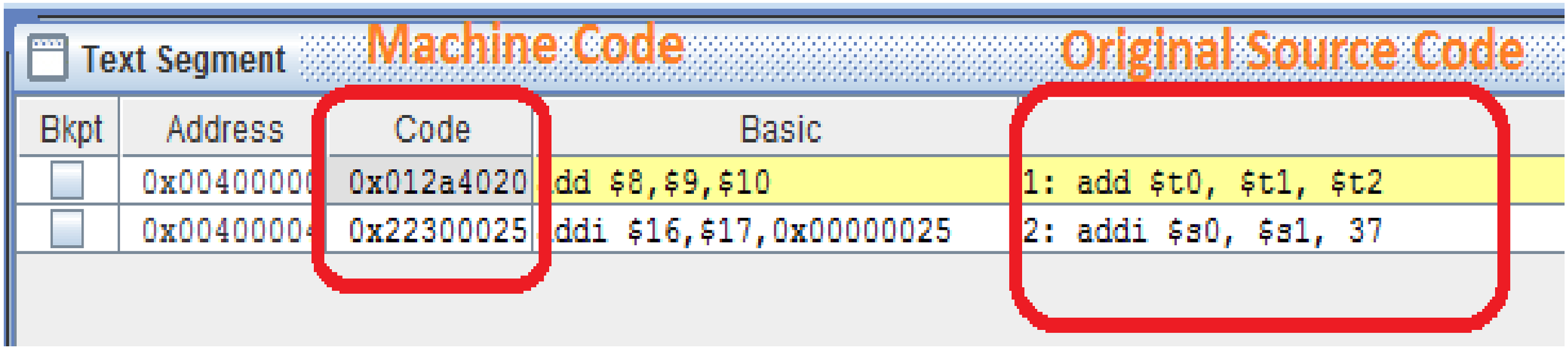
②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	`bc1t`	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	`bc1f`	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	`div`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	`divu`	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	`add.s`	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	`add.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	`c.x.s`*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	`c.x.d`*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
FP Divide Single	`div.s`	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	`div.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	`mul.s`	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	`mul.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	`sub.s`	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	`sub.d`	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	`lwc1`	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	`ldc1`	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	`mfhi`	R	R[rd] = Hi		0 /--/--/10
Move From Lo	`mflo`	R	R[rd] = Lo		0 /--/--/12
Move From Control	`mfc0`	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	`mult`	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	`multu`	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	`sra`	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	`swc1`	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	`sdc1`	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

* (*x* is `eq`, `lt`, or `le`) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)

Translate assembly into machine code

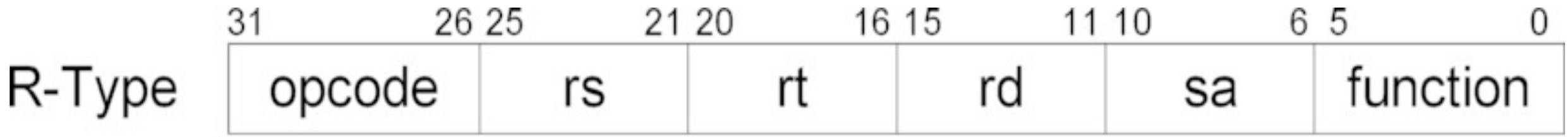
- Assembly to machine code



Machine code with **add** instruction

- Translate the following instruction to machine code:

add $t0, $t1, $t2



- R instruction format
- Opcode/function: 0/20

  → 00 0000/10 0000

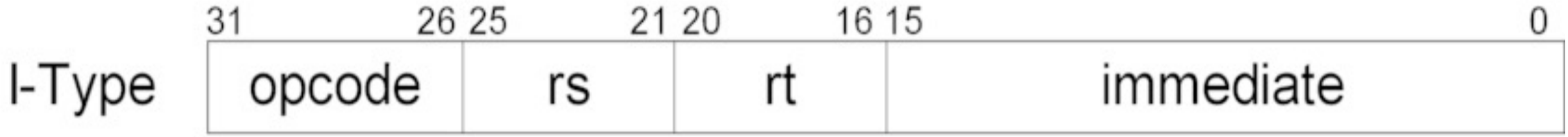
- R_d: $t0 is $8 or 01000
- R_s: $t1 is $9 or 01001
- R_t: $t2 is $10 or 01010
- sa (shamt) is 00000

0000 0010 0011 0010 1000 0000 0010 0010$_2$ or **0x02328022**

Machine code with **addi** instruction

- Translate the following instruction into machine code:

addi $s2, $t8, 37



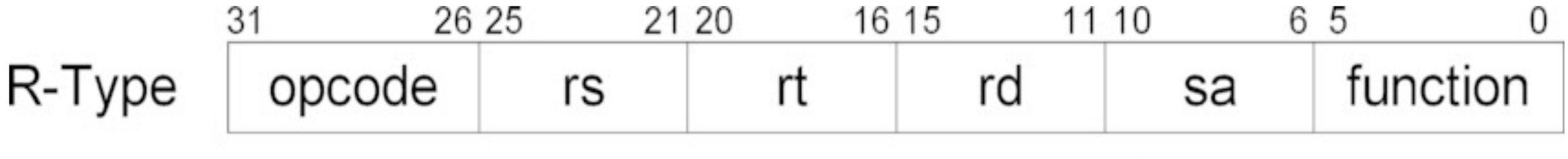
- I instruction format
- Opcode: 8 → 00 1000
- R_t: $s2 is $18 or 10010
- R_s: $t8 is $24 or 11000
- Immediate value is 37, or 0x0025

$0010\ 0011\ 0001\ 0010\ 0000\ 0000\ 0010\ 0101_2$ or **0x23120025**

Machine code with sll instruction

- Translate the following instruction into machine code:

sll $t0, $t1, 10



- R instruction format
- Opcode/function: 0/00

  → 00 0000/00 0000

- R_d: $t0 is $8 or 01000
- R_s unused
- R_t: $t1 is $9 or 01001
- sa (shamt) is 10 or 0x01010

0000 0000 0000 1001 0100 0010 1000 0000_2 or **0x00094280**

Exercise 4.2

- Translate the following instruction into machine code:

```
.text
.globl main
main:
        ori $t0, $zero, 15
        ori $t1, $zero, 3
        add $t1, $zero $t1
        sub $t2, $t0, $t1
        sra $t2, $t2, 2
        mult $t0, $t1
        mflo $a0
        ori $v0, $zero, 1
        syscall
        addi $v0, $zero, 10
        syscall
.data
        result: .asciiz "15 * 3 is "
```

Exercise 4.3

- Translate the following machine code into MIPS instruction:

```
0x2010000a
0x34110005
0x012ac022
0x00184082
0x030f9024
```

Assignment 4.1

- The sum of two 32-bit integers may not be representable in 32 bits. In this case, we say that an overflow has occurred. Overflow is possible only with operands of the same sign.

- For two nonnegative (negative) operands, if the sum obtained is less (greater) than eitheir operand, overflow has occurred.

- The following program dectects overflow based on this rule. Two operands are stored in register **$s1** and **$s2**, the sum is stored in register **$s3**. If overflow occur, **$t0** register is set to 1 and clear to 0 in otherwise.

Assignment 4.1

```
#Laboratory Exercise 4, Home Assignment 1
.text
start:
      li    $t0,0                #No Overflow is default status
      addu  $s3,$s1,$s2          # s3 = s1 + s2
      xor   $t1,$s1,$s2          #Test if $s1 and $s2 have the same sign

      bltz  $t1,EXIT             #If not, exit
      slt   $t2,$s3,$s1
      bltz  $s1,NEGATIVE#Test if $s1 and $s2 is negative?
      beq   $t2,$zero,EXIT     #s1 and $s2 are positive
           # if $s3 > $s1 then the result is not overflow
      j     OVERFLOW
NEGATIVE:
      bne   $t2,$zero,EXIT     #s1 and $s2 are negative
           # if $s3 < $s1 then the result is not overflow
OVERFLOW:
      li    $t0,1                #the result is overflow
EXIT:
```

Assignment 4.2

- The following program demonstrates how to use logical instructions to extract information from one register.
- We can extract one bit or more according to the mask we use. Read this example carefully and explain each lines of code.

```
#Laboratory Exercise 4, Home Assignment 2
.text
      li    $s0, 0x0563       #load test value for these function
      andi  $t0, $s0, 0xff    #Extract the LSB of $s0
      andi  $t1, $s0, 0x0400  #Extract bit 10 of $s0
```

Assigment 4.3

- Enter the following two instructions in MARS, and assemble them.

addiu $t0, $zero, 60000

ori $t0, $zero, 60000

- What differences do you notice? Are the results the same? Will one run faster than the other? Explain what you observe.

Assigment 4.4

- Implement a program to do a bitwise complement (NOT) of an integer number entered by the user.

- You should use the XOR operator to implement the NOT, do not use the NOT operator.

- Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

Assigment 4.5

- Implement a program to calculate the 2's complement of a number entered by the user.
- The program should only user the XOR and ADD operators.
- Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

Assigment 4.6

- Implement a simple program to do a bitwise NAND in MARS.
- Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

Assigment 4.7

- Implement a program which multiplies a user input by 10 using only bit shift operations and addition.

- Check to see if your program is correct by using the **mult** and **mflo** operators.

- Your program should include a proper and useful prompt for input, and print the results in a meaningful manner.

End of week 4