EECE 2322: Fundamentals of Digital Design and Computer Organization

Lecture 7_2: MIPS ISA

Xiaolin Xu
Department of ECE
Northeastern University

Review: Instruction Formats

R-Type

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

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

Backward: From Machine to Assembly Language

- How to translate the following machine language?
  - R, I, or J type?

```
0x2237FFF1
0x02F34022
```

Backward: From Machine to Assembly Language

- Step 1: we need to know what type of instruction
  - Interpret machine language

```
0x2237FFF1

0x02F34022
```

R-Type

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

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

J-Type

op	addr
6 bits	26 bits

Backward: From Machine to Assembly Language

- Step 1: we need to know what type of instruction
  - Interpret machine language
  - Determine the op first!

```
0x2237FFF1=________________________?
0x02F34022=________________________?
```

Backward: From Machine to Assembly Language

- Step 1: we need to know what type of instruction
  - Interpret machine language
  - Determine the op first!

```
0x2237FFF1=> op = 8 => ?
0x02F34022=> op = 0 => ?
```

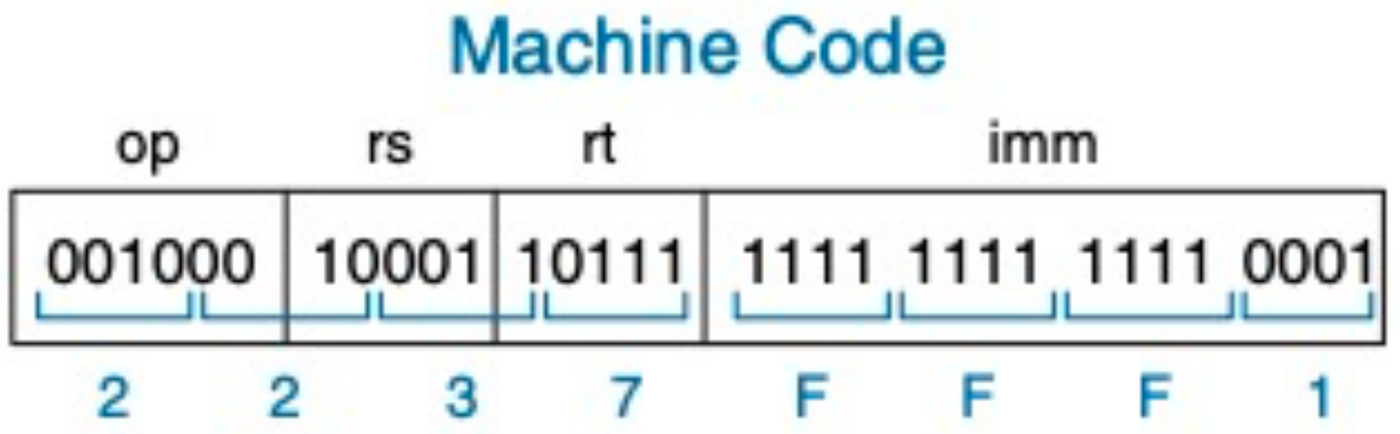
Backward: From Machine to Assembly Language

- Step 2: divide the machine language numbers accordingly

`0x2237FFF1=>` `op = 8, rs = 17, rt = 23, imm = ?`

I-Type

op	rs	rt	imm
6 bits	5 bits	5 bits	16 bits

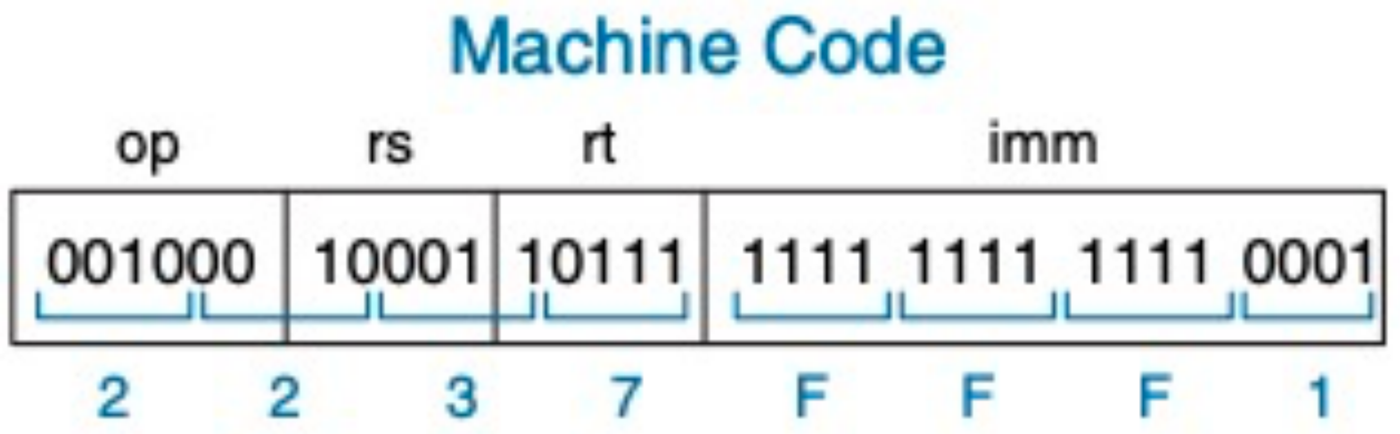


Backward: From Machine to Assembly Language

- Step 2: divide the machine language numbers accordingly

`0x2237FFF1=>` <u>`op = 8, rs = 17, rt = 23, imm = -15`</u>

`0x2237FFF1=>` <u>`op = 8`</u> `=>` <u>`addi $s7, $s1, -15`</u>



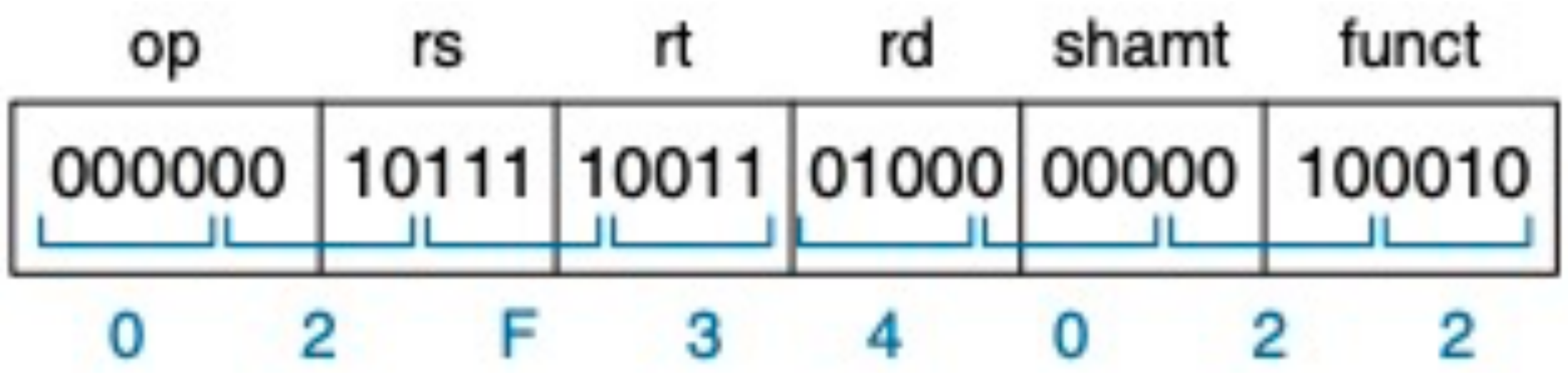
Backward: From Machine to Assembly Language

- Step 2: divide the machine language numbers accordingly

```
0x02F34022=> op = 0 => funct = 34 (sub),
rs = 23, rt = 19, rd = 8, shamt = 0
```

R-Type

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

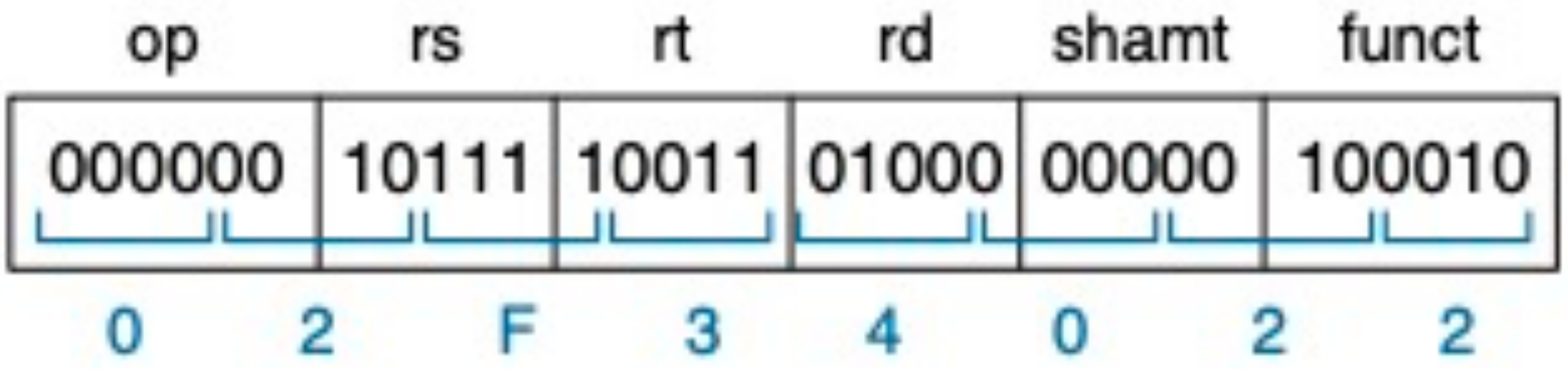


Backward: From Machine to Assembly Language

- Step 2: divide the machine language numbers accordingly

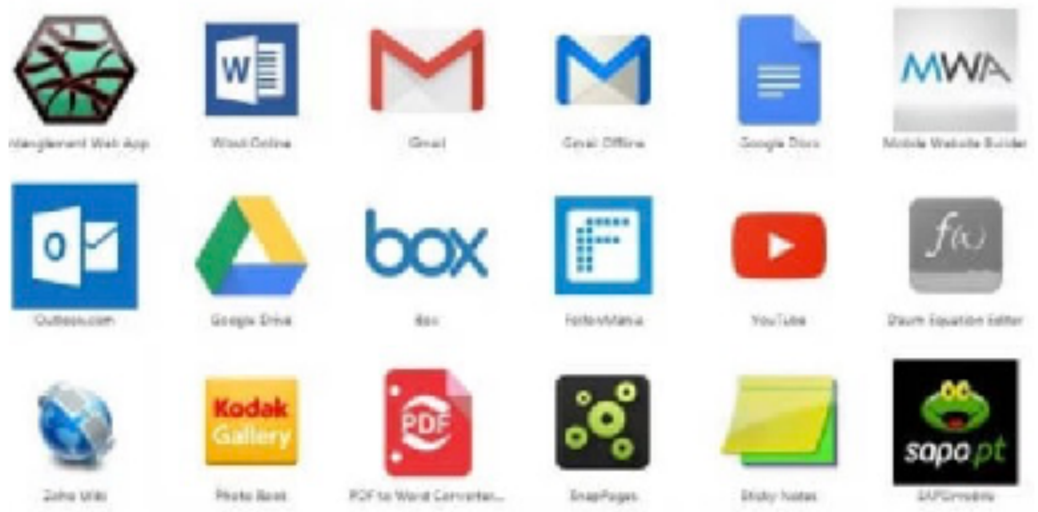
`0x02F34022=>` <u>`op = 0`</u> `=>` <u>`funct = 34 (sub), rs = 23, rt = 19, rd = 8, shamt = 0`</u>

`0x02F34022=>` <u>`sub $t0, $s7, $s3`</u>



Power of the Stored Program

- In MIPS, any software program can be represented as
  - 32-bit instructions & data stored in memory
- Sequence of instructions: the only difference between two applications
- Easy to reconfigure



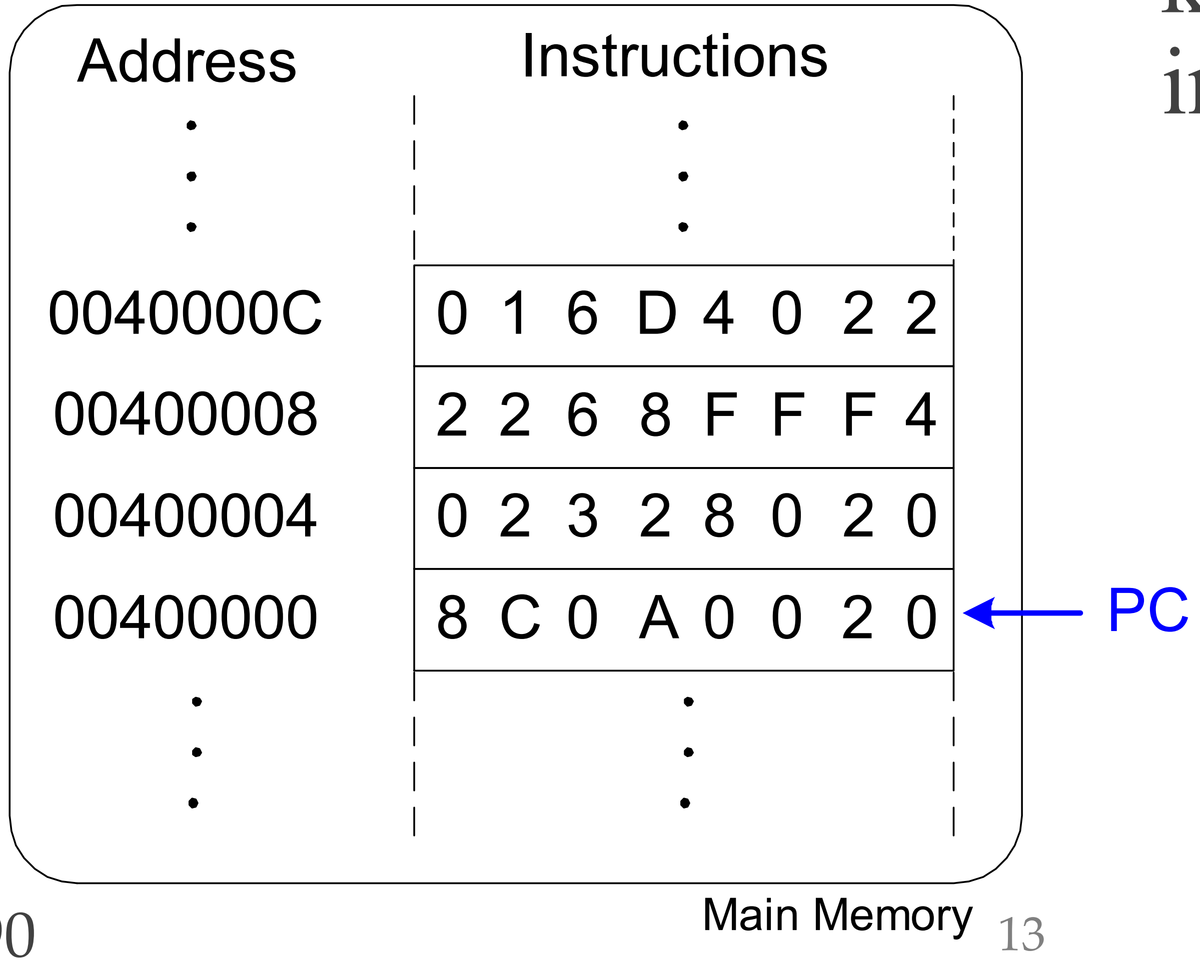
Power of the Stored Program

- To run a new program:
  - No rewiring required -> General purpose computing!
  - Simply store new program in memory
- Program Execution:
  - Processor *fetches* (reads) instructions from memory in sequence
  - Processor performs the specified operation

The Stored Program

Assembly Code	Machine Code
`lw   $t2, 32($0)`	`0x8C0A0020`
`add  $s0, $s1, $s2`	`0x02328020`
`addi $t0, $s3, -12`	`0x2268FFF4`
`sub  $t0, $t3, $t5`	`0x016D4022`

Stored Program



Program Counter (PC): keeps track of current instruction

Programming

- High-level languages:
  - e.g., C, Java, Python
  - Written at higher level of abstraction
- Common high-level software constructs:
  - if/else statements
  - for loops
  - while loops
  - arrays
  - function calls

Logical Instructions

- **`and, or, xor, nor`**
  - `and`: useful for **masking** bits
    - Masking all but the least significant byte of a value:
      0xF234012F AND 0x000000FF = 0x0000002F
  - `or:` useful for **combining** bit fields
    - Combine 0xF2340000 with 0x000012BC:
      0xF2340000 OR 0x000012BC = 0xF23412BC

  Recall that MIPS has $0 with all-0 inside
  - `nor:` useful for **inverting** bits:
    - A NOR $0 = NOT A
    - Thus, MIPS does not provide a NOT instruction

Logical Instructions

- **`andi`, `ori`, `xori`**
  - 16-bit immediate is zero-extended (*not* sign-extended)
    - `nori` not provided by MIPS, why?

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

$s3								
$s4								
$s5								
$s6								

Logical Instructions Example 1

Source Registers

$s1	1111	1111	1111	1111	0000	0000	0000	0000
$s2	0100	0110	1010	0001	1111	0000	1011	0111

Assembly Code

```
and $s3, $s1, $s2
or  $s4, $s1, $s2
xor $s5, $s1, $s2
nor $s6, $s1, $s2
```

Result

$s3	0100	0110	1010	0001	0000	0000	0000	0000
$s4	1111	1111	1111	1111	1111	0000	1011	0111
$s5	1011	1001	0101	1110	1111	0000	1011	0111
$s6	0000	0000	0000	0000	0000	1111	0100	1000

Logical Instructions Example 2

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

$\xleftrightarrow{\text{zero-extended}}$

Assembly Code

```
andi $s2, $s1, 0xFA34
ori  $s3, $s1, 0xFA34
xori $s4, $s1, 0xFA34
```

Result

$s2								
$s3								
$s4								

Logical Instructions Example 2

Source Values

$s1	0000	0000	0000	0000	0000	0000	1111	1111
imm	0000	0000	0000	0000	1111	1010	0011	0100

$\longleftarrow$ zero-extended $\longrightarrow$

Assembly Code

Result

`andi $s2, $s1, 0xFA34`	$s2	0000	0000	0000	0000	0000	0000	0011	0100
`ori  $s3, $s1, 0xFA34`	$s3	0000	0000	0000	0000	1111	1010	1111	1111
`xori $s4, $s1, 0xFA34`	$s4	0000	0000	0000	0000	1111	1010	1100	1011

Shift Instructions

- `sll:` shift left logical
  - **Example:** `sll $t0, $t1, 5  # $t0 <= $t1 << 5`
- `srl:` shift right logical
  - **Example:** `srl $t0, $t1, 5  # $t0 <= $t1 >> 5`
- `sra:` shift right arithmetic
  - **Example:** `sra $t0, $t1, 5  # $t0 <= $t1 >>> 5`

Shift Instructions: Practice

Assembly Code / Field Values

	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0				0
`srl $s2, $s1, 2`	0	0				2
`sra $s3, $s1, 2`	0	0				3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000				000000	(0x00114080)
000000	00000				000010	(0x00119082)
000000	00000				000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Shift Instructions: Practice

Assembly Code / **Field Values**

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000				000000	(0x00114080)
000000	00000				000010	(0x00119082)
000000	00000				000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Shift Instructions: Practice

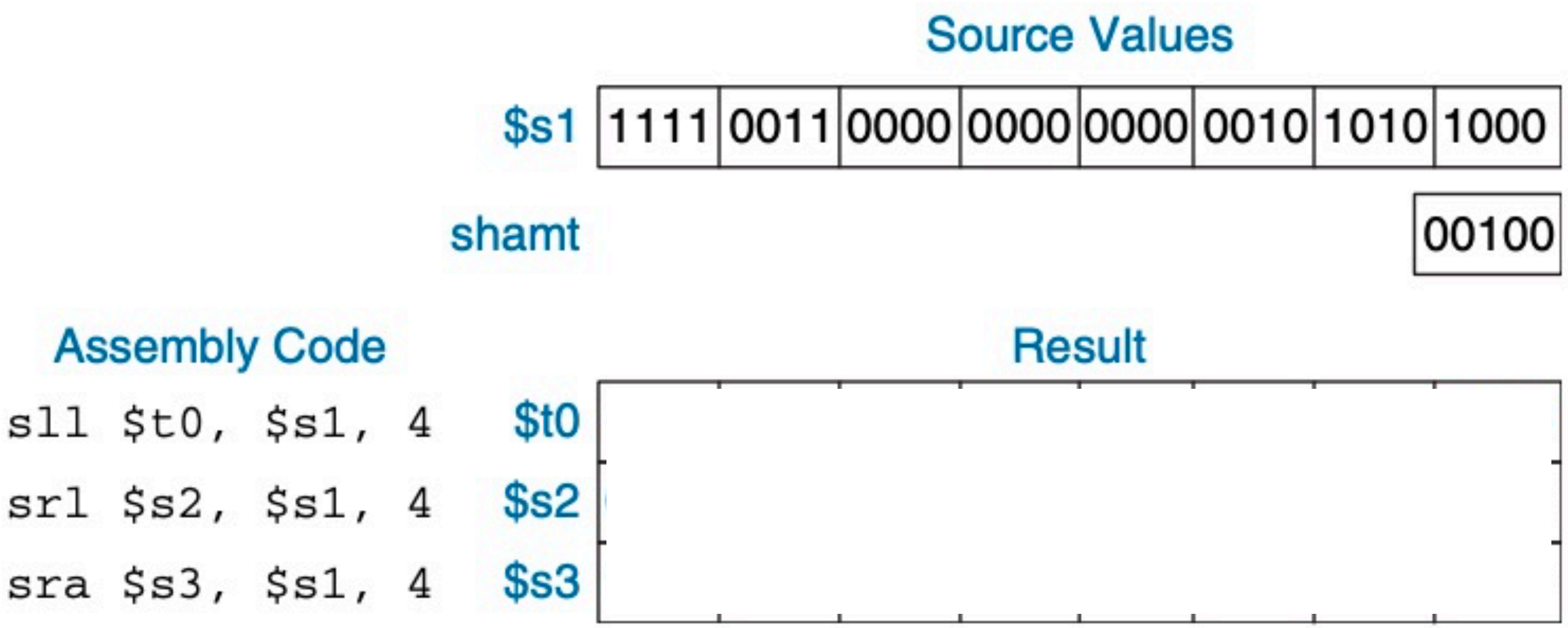
Assembly Code / Field Values

Assembly Code	op	rs	rt	rd	shamt	funct
`sll $t0, $s1, 2`	0	0	17	8	2	0
`srl $s2, $s1, 2`	0	0	17	18	2	2
`sra $s3, $s1, 2`	0	0	17	19	2	3
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	00000	10001	01000	00010	000000	(0x00114080)
000000	00000	10001	10010	00010	000010	(0x00119082)
000000	00000	10001	10011	00010	000011	(0x00119883)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Shift Instructions: Practice



Source Values

$s1	1111	0011	0000	0000	0000	0010	1010	1000
shamt								00100

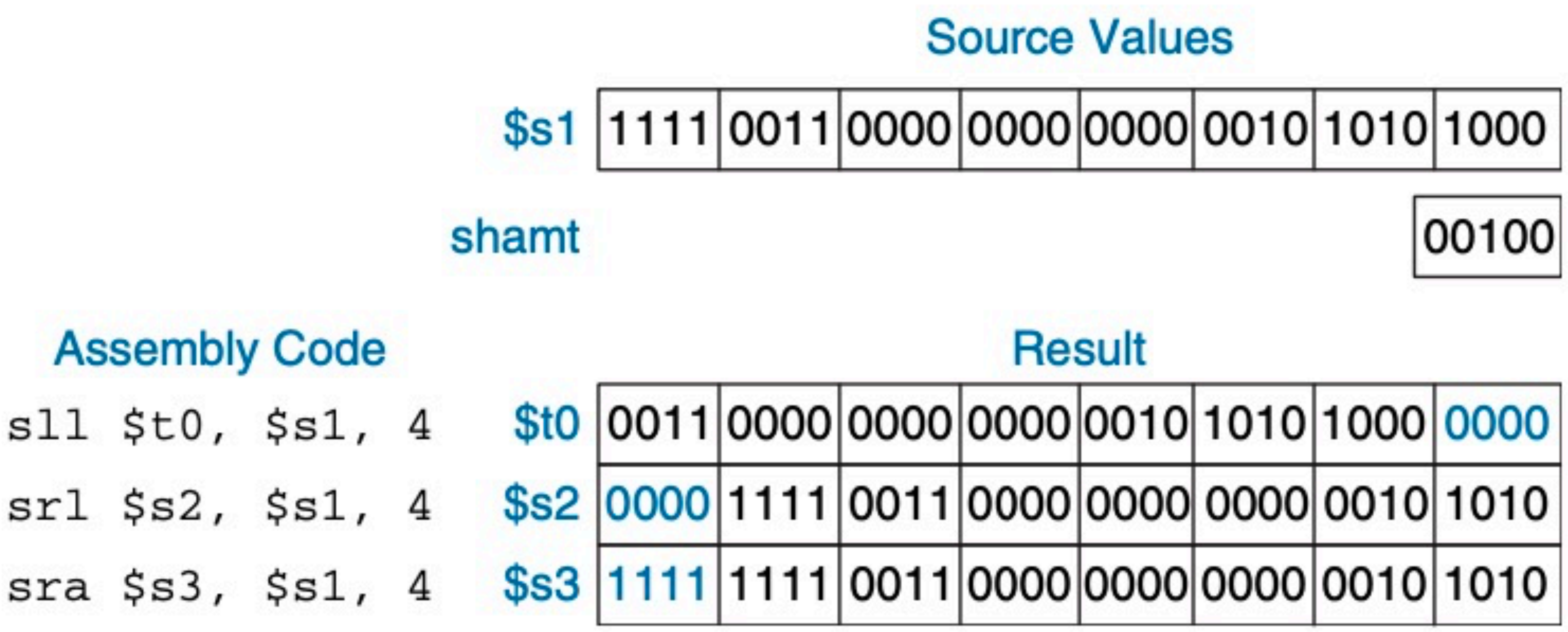
Assembly Code

```
sll $t0, $s1, 4
srl $s2, $s1, 4
sra $s3, $s1, 4
```

Result

$t0								
$s2								
$s3								

Shift Instructions: Practice



Source Values

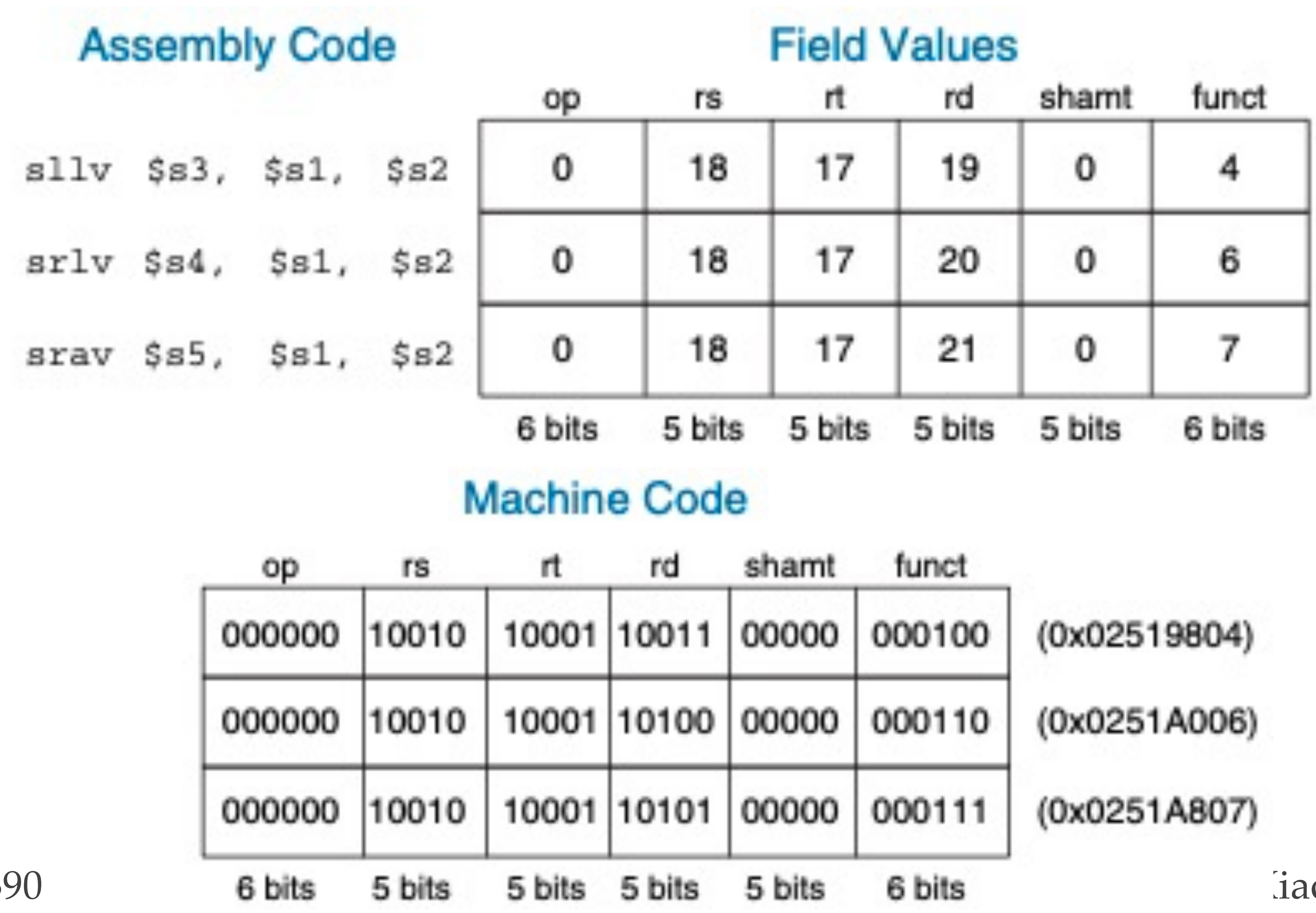
$s1	1111	0011	0000	0000	0000	0010	1010	1000
shamt								00100

Assembly Code	Result								
`sll $t0, $s1, 4`	$t0	0011	0000	0000	0000	0010	1010	1000	0000
`srl $s2, $s1, 4`	$s2	0000	1111	0011	0000	0000	0000	0010	1010
`sra $s3, $s1, 4`	$s3	1111	1111	0011	0000	0000	0000	0010	1010

Variable Shift Instructions

- `sllv:` shift left logical variable
  - **Example:** `sllv $t0, $t1, $t2 # $t0 <= $t1 << $t2`
- `srlv:` shift right logical variable
  - **Example:** `srlv $t0, $t1, $t2 # $t0 <= $t1 >> $t2`
- `srav:` shift right arithmetic variable
  - **Example:** `srav $t0, $t1, $t2 # $t0 <= $t1 >>> $t2`

Variable Shift Instructions: Practice



Assembly Code / **Field Values**

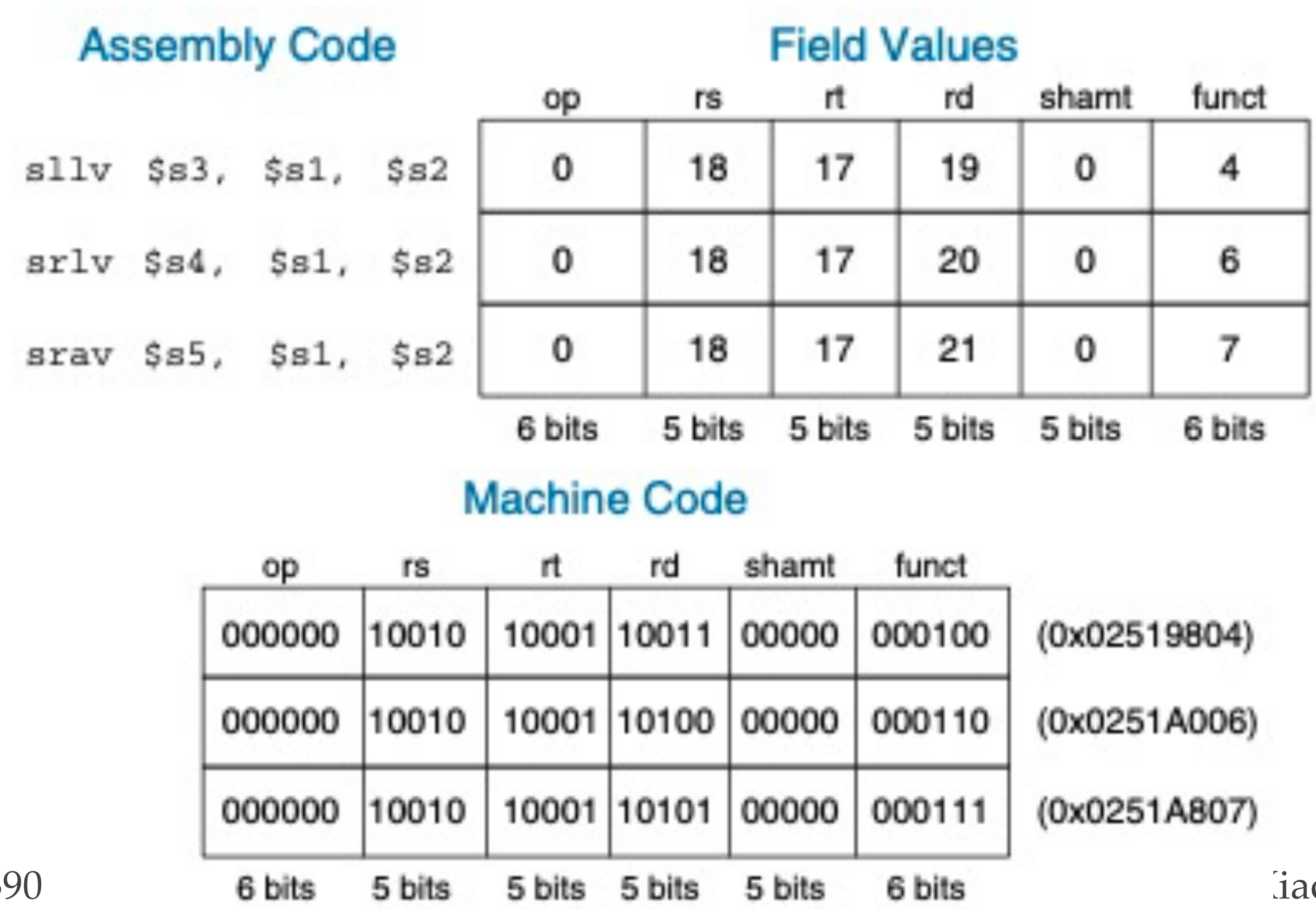
Assembly Code	op	rs	rt	rd	shamt	funct
`sllv $s3, $s1, $s2`	0	18	17	19	0	4
`srlv $s4, $s1, $s2`	0	18	17	20	0	6
`srav $s5, $s1, $s2`	0	18	17	21	0	7
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10010	10001	10011	00000	000100	(0x02519804)
000000	10010	10001	10100	00000	000110	(0x0251A006)
000000	10010	10001	10101	00000	000111	(0x0251A807)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Variable Shift Instructions: Practice

- `shamt not used here!`



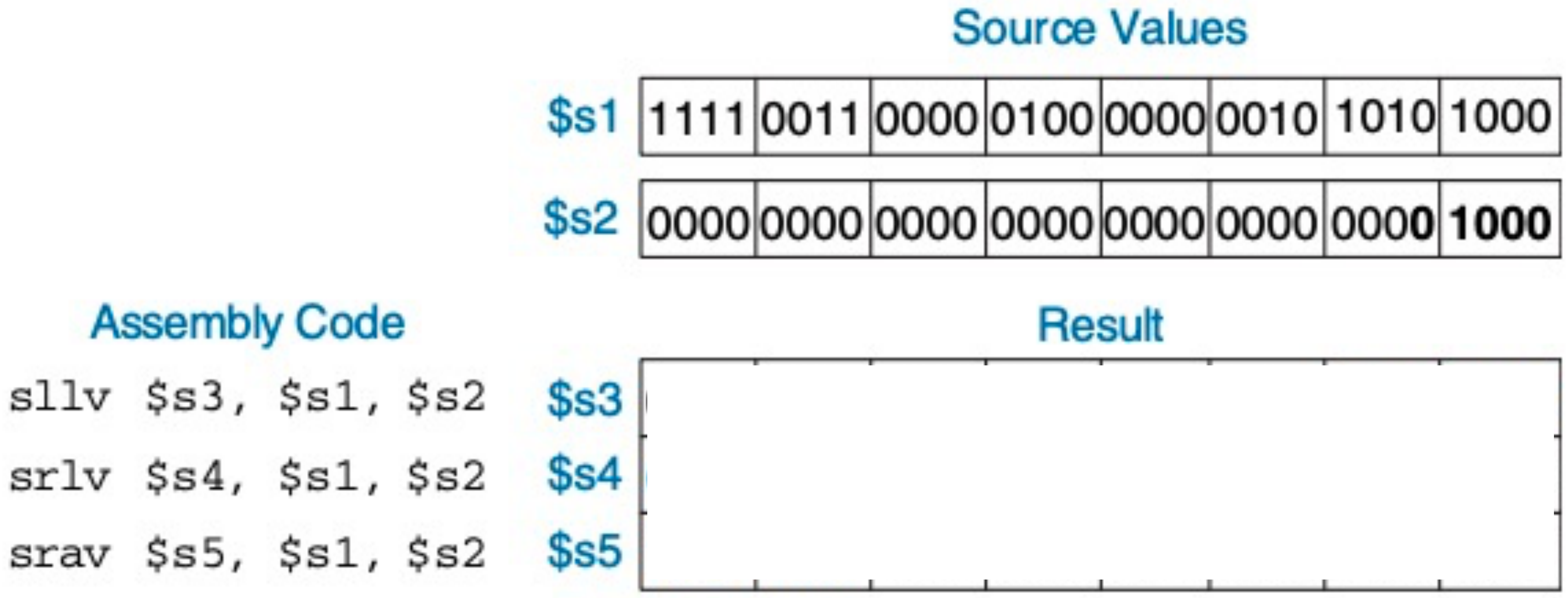
Assembly Code / **Field Values**

Assembly Code	op	rs	rt	rd	shamt	funct
`sllv $s3, $s1, $s2`	0	18	17	19	0	4
`srlv $s4, $s1, $s2`	0	18	17	20	0	6
`srav $s5, $s1, $s2`	0	18	17	21	0	7
	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Machine Code

op	rs	rt	rd	shamt	funct	
000000	10010	10001	10011	00000	000100	(0x02519804)
000000	10010	10001	10100	00000	000110	(0x0251A006)
000000	10010	10001	10101	00000	000111	(0x0251A807)
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	

Variable Shift Instructions: Practice



Source Values

$s1	1111	0011	0000	0100	0000	0010	1010	1000
$s2	0000	0000	0000	0000	0000	0000	000**0**	**1000**

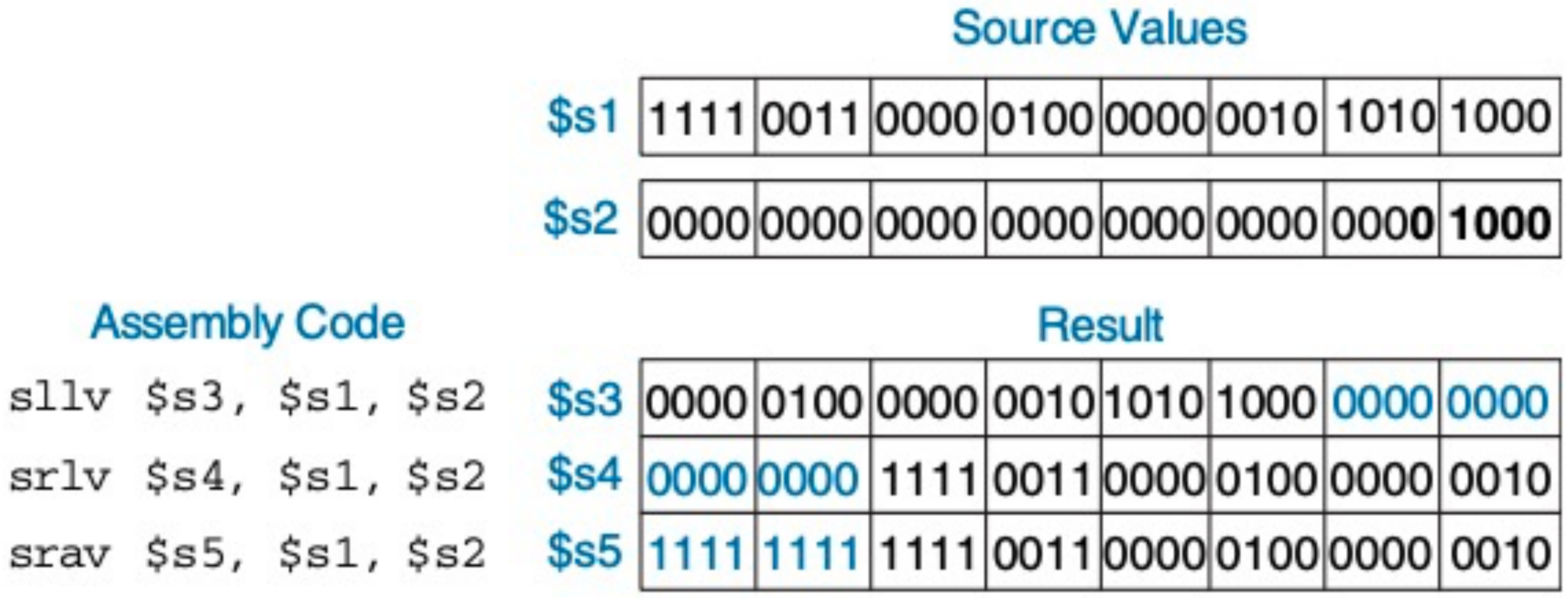
Assembly Code

```
sllv $s3, $s1, $s2
srlv $s4, $s1, $s2
srav $s5, $s1, $s2
```

Result

$s3								
$s4								
$s5								

Variable Shift Instructions: Practice



Source Values

$s1	1111	0011	0000	0100	0000	0010	1010	1000
$s2	0000	0000	0000	0000	0000	0000	0000	**1000**

Assembly Code	Result								
`sllv $s3, $s1, $s2`	$s3	0000	0100	0000	0010	1010	1000	0000	0000
`srlv $s4, $s1, $s2`	$s4	0000	0000	1111	0011	0000	0100	0000	0010
`srav $s5, $s1, $s2`	$s5	1111	1111	1111	0011	0000	0100	0000	0010