

Instruction Set Architecture

Michael C. Hackett
Assistant Professor, Computer Science

Community
College
of Philadelphia

Lecture Topics

- Instruction Set Architectures
- Instruction Formats
 - Register Format Instructions
 - Immediate Format Instructions
 - Jump Format Instructions
- Pseudo Operations
- Floating Point Instructions
 - Floating Point Register Format
 - Floating Point Immediate Format
- Addressing Modes
- Complex Instruction Set Computers (CISC)
- Reduced Instruction Set Computers (RISC)
- RISC-V

Instruction Set Architectures

- An **instruction set architecture (ISA)** the overall design of a CPU's instructions.
- An ISA must be selected (or created) prior to designing a CPU
 - We can't design a CPU until we decide how we will tell it to add, subtract, and perform dozens of kinds other operations.

Instruction Set Architectures

- A *zero-address architecture* uses a data structure called a stack.
 - The operands of any operation are the stack's top two values
- Using pseudocode (no particular programming language), we'll demonstrate the calculation of $5+(4-1)$ in a zero-address architecture

Instruction Set Architectures

Pushes 1 and 4 onto the stack

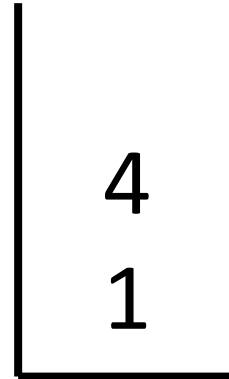
push 1

push 4

sub

push 5

add



Stack

Instruction Set Architectures

Takes the two values on top of the stack, subtracts, and puts the result in the stack

push 1

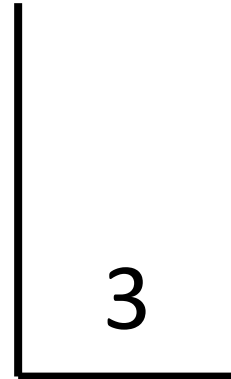
push 4

sub

$4 - 1 = 3$

push 5

add



Instruction Set Architectures

Pushes 5 onto the top of the stack

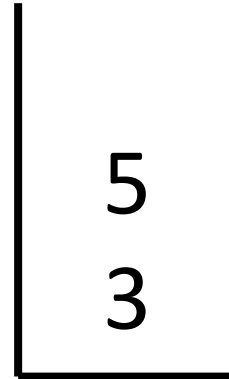
push 1

push 4

sub

push 5

add



Instruction Set Architectures

Takes the two values on top of the stack, adds, and puts the result in the stack

push 1

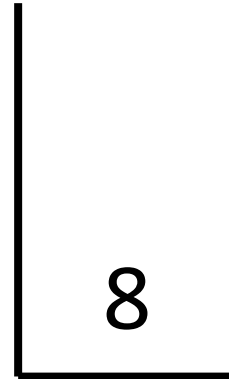
push 4

sub

push 5

add

$5 + 3 = 8$



Instruction Set Architectures

- A *one-address architecture* allows each instruction to use one memory address.
- The CPU uses a special register as an accumulator
 - This register is both an operation's left operand **and** result
- Using pseudocode (no particular programming language), we'll demonstrate the calculation of $5+(4-1)$ in a one-address architecture

Instruction Set Architectures

clear	#Accumulator = 0
add 1	#Accumulator = 0 + 1 = 1
negate	#Accumulator = -1
add 4	#Accumulator = -1 + 4 = 3
add 5	#Accumulator = 3 + 5 = 8

Instruction Set Architectures

- A *two-address architecture* allows each instruction to use two operands.
- The first operand is both the first operand and the register where the result is stored
- Using pseudocode (no particular programming language), we'll demonstrate the calculation of $5+(4-1)$ in a two-address architecture

Instruction Set Architectures

load r1, 5

load r2, 4

load r3, 1

sub r2, r3 #r2 = r2 - r3

add r1, r2 #r1 = r1 + r2

Instruction Set Architectures

- A *three-address architecture* allows each instruction to use three operands- the left and right operands and the where to store the result.
- MIPS is a three-address architecture
 - As is RISC-V, which will be discussed at the end of the lecture
- The calculation of $5+(4-1)$ in MIPS:

addi \$t1, \$0, 4	#\$t1 = 0 + 4 = 4
subi \$t1, \$t1, 1	#\$t1 = 4 - 1 = 3
addi \$t0, \$t1, 5	#\$t0 = 3 + 5 = 8

Instruction Formats

- An **instruction format** defines the arrangement and organization of the bits that form each instruction.
 - Each instruction is 32 bits in MIPS (same as its word size)
- This binary version of instructions is called the **machine language**
 - The assembler converts assembly instructions to machine language
 - The machine language is the actual instructions that are given to and understood by the CPU

Instruction Formats

- MIPS has three instruction formats:

- **Register Format (R Format)**

- Two operands and a target

add \$t2, \$t0, \$t1

- **Immediate Format (I Format)**

- Instructions with an immediate operand

addi \$t1, \$t0, 7

- **Jump Format (J Format)**

- Jump and Jump-and-Link instructions

j label

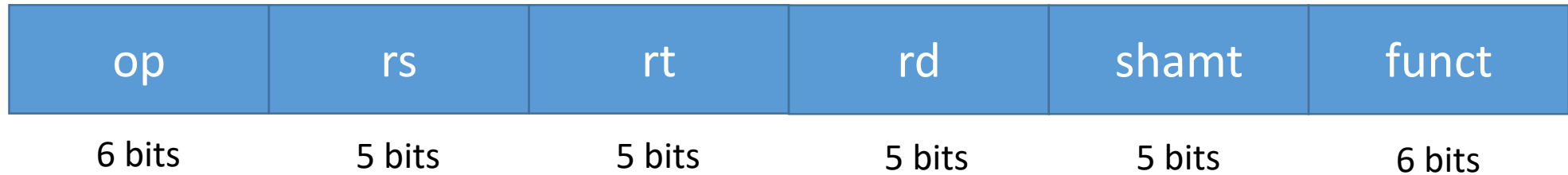
Instruction Formats

- See the MIPS Reference Card posted in Canvas as a reference for this lecture

MIPS Reference Data				ARITHMETIC CORE INSTRUCTION SET			
CORE INSTRUCTION SET			OPCODE / FUNCT (Hex)	ARITHMETIC CORE INSTRUCTION SET			OPCODE / FMT / FT / FUNCT (Hex)
NAME, MNEMONIC	FOR-MAT	OPERATION (in Verilog)		NAME, MNEMONIC	FOR-MAT	OPERATION	
Add	add	R R[rd] = R[rs] + R[rt]	(1) 0 / 20 _{hex}	Branch On FP True	bclt	FI if(FPcond)PC=PC+4+BranchAddr	(4) 11/8/1/--
Add Immediate	addi	I R[rt] = R[rs] + SignExtImm	(1,2) 8 _{hex}	Branch On FP False	bclt	FI if(!FPcond)PC=PC+4+BranchAddr	(4) 11/8/0/--
Add Imm. Unsigned	addiu	I R[rt] = R[rs] + SignExtImm	(2) 9 _{hex}	Divide	div	R Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	0/--/--/1a
Add Unsigned	addu	R R[rd] = R[rs] + R[rt]	0 / 21 _{hex}	Divide Unsigned	divu	R Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6) 0/--/--/1b
And	and	R R[rd] = R[rs] & R[rt]	0 / 24 _{hex}	FP Add Single	add.s	FR F[fd] = F[fs] + F[ft]	11/10/--/0
And Immediate	andi	I R[rt] = R[rs] & ZeroExtImm	(3) c _{hex}	FP Add Double	add.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}	11/11/--/0
Branch On Equal	beq	I if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4) 4 _{hex}	FP Compare Single	c.x.s*	FR FPcond = (F[fs] op F[ft]) ? 1 : 0	11/10/--/y
Branch On Not Equal	bne	I if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4) 5 _{hex}	FP Compare Double	c.x.d*	FR FPcond = ({F[fs],F[fs+1]} op {F[ft],F[ft+1]}) ? 1 : 0	11/11/--/y
Jump	j	J PC=JumpAddr	(5) 2 _{hex}	FP Divide Single	div.s	FR F[fd] = F[fs] / F[ft]	11/10/--/3
Jump And Link	jal	J R[31]=PC+8;PC=JumpAddr	(5) 3 _{hex}	FP Divide Double	div.d	FR {F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}	11/11/--/3
Jump Register	jr	R PC=R[rs]	0 / 08 _{hex}	FP Multiply Single	mul.s	FR F[fd] = F[fs] * F[ft]	11/10/--/2
Load Byte Unsigned	lbu	I R[rt] = {24'b0,M[R[rs]+SignExtImm](7:0)}	(2) 24 _{hex}	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/--/--/1

Register Format Instructions

- MIPS Register Format Instruction



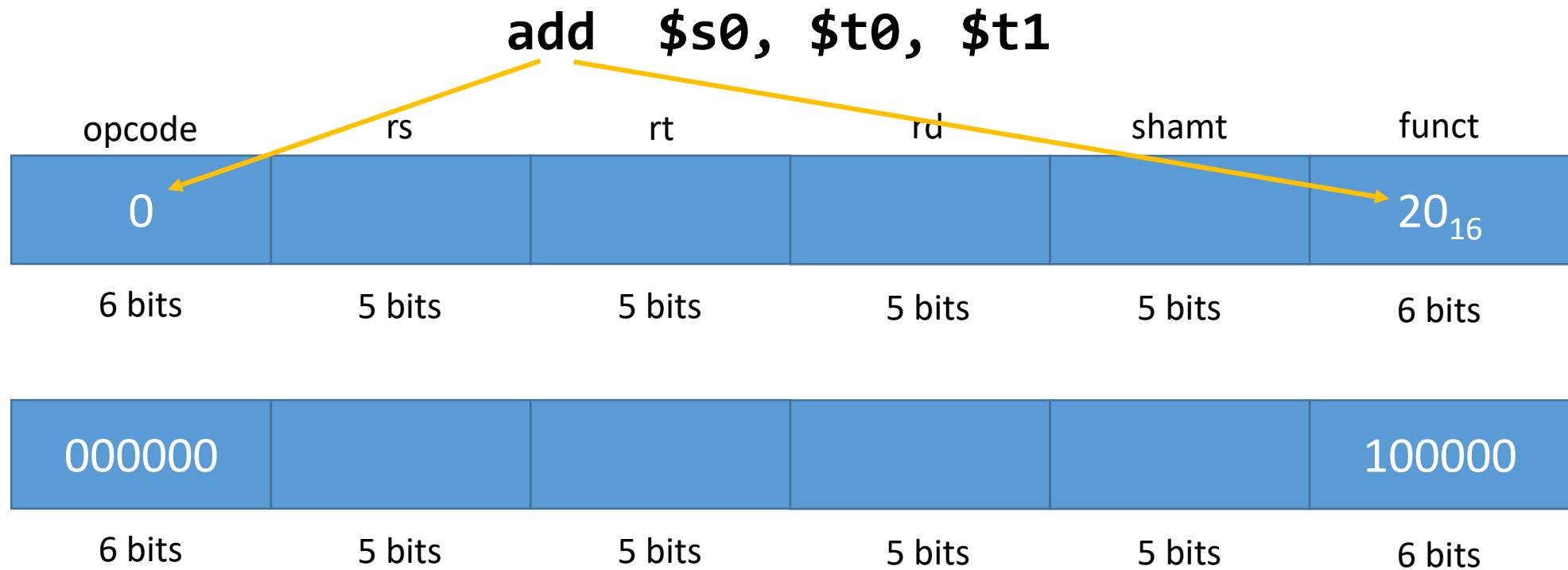
- op – Operation Code (**opcode**)
- rs – First source register (left operand)
- rt – Second source register (right operand)
- rd – Destination register
- shamt – Shift amount (used in shift operations)
- funct – Function code (specifies a variant of the opcode)

Register Format Instructions

- We see on the MIPS Reference Card, the **add** mnemonic has an opcode of 0 and a function code of 20_{16}

CORE INSTRUCTION SET			OPCODE / FUNCT (Hex)
NAME, MNEMONIC	FOR- MAT	OPERATION (in Verilog)	
Add	add	R $R[rd] = R[rs] + R[rt]$	(1) 0 / 20_{hex}

Register Format Instructions



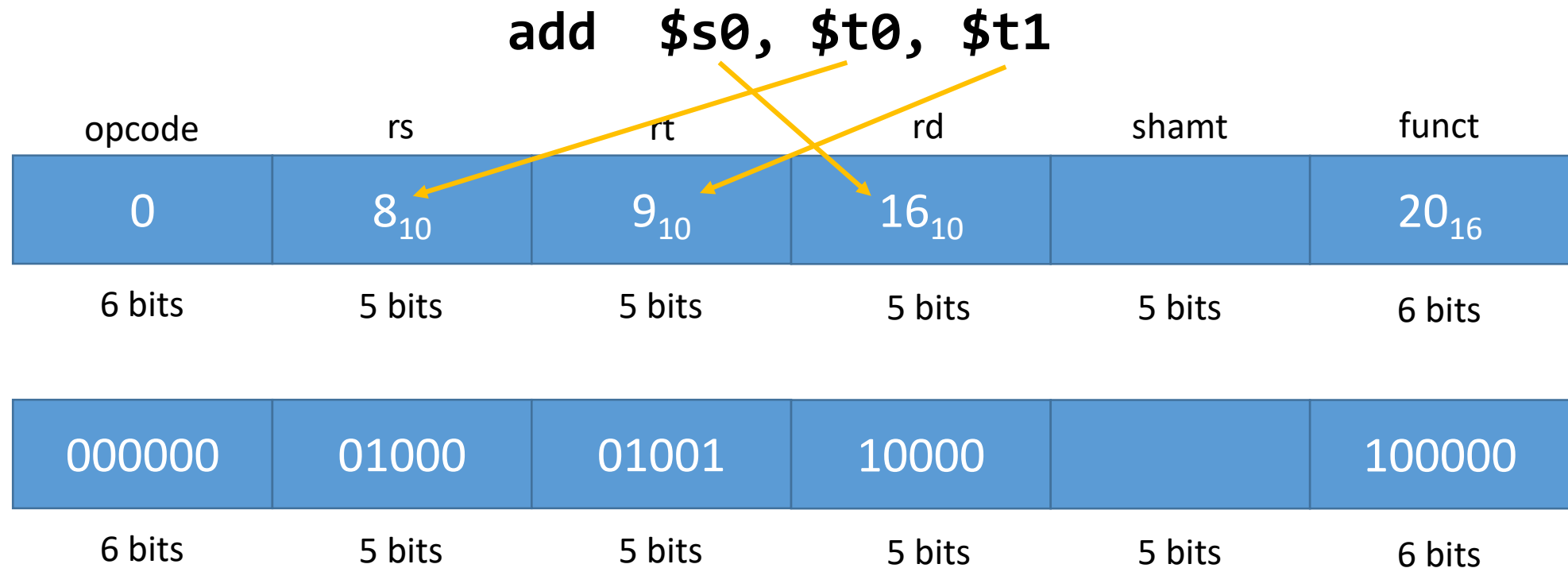
Register Format Instructions

- We can also see on the MIPS Reference Card, that each register corresponds to a number

REGISTER NAME, NUMBER, USE, CALL CONVENTION			
NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
\$zero	0	The Constant Value 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes

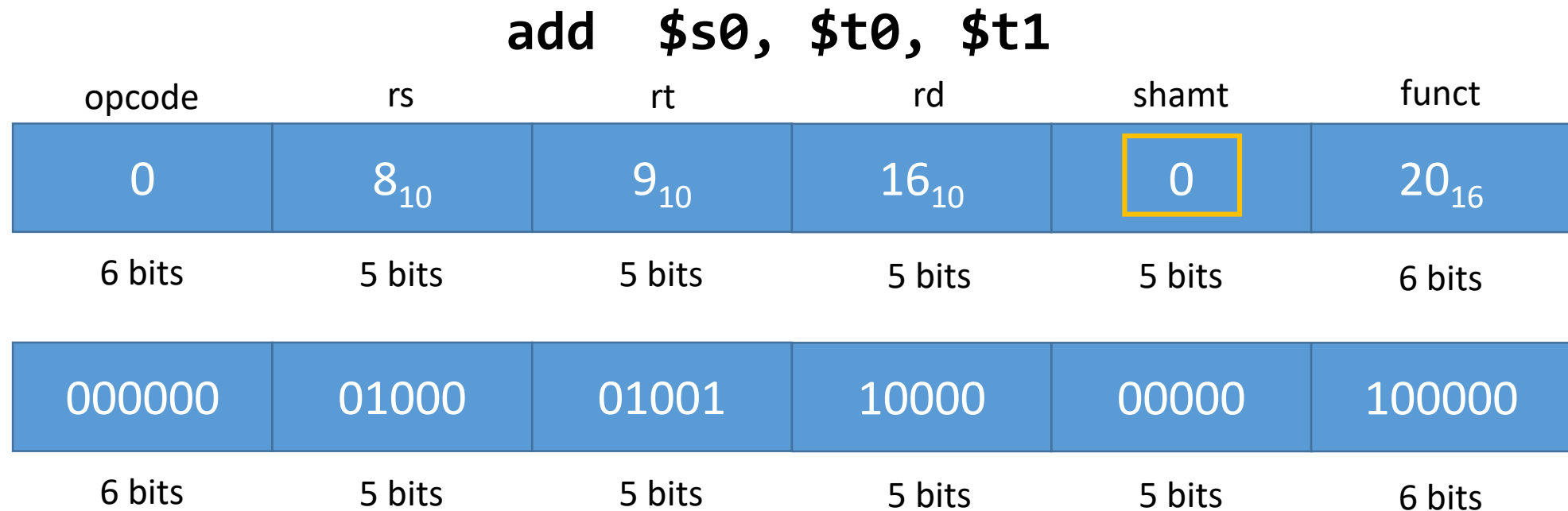
Register Format Instructions

\$t0-\$t7	8-15	Temporaries
\$s0-\$s7	16-23	Saved Temporaries

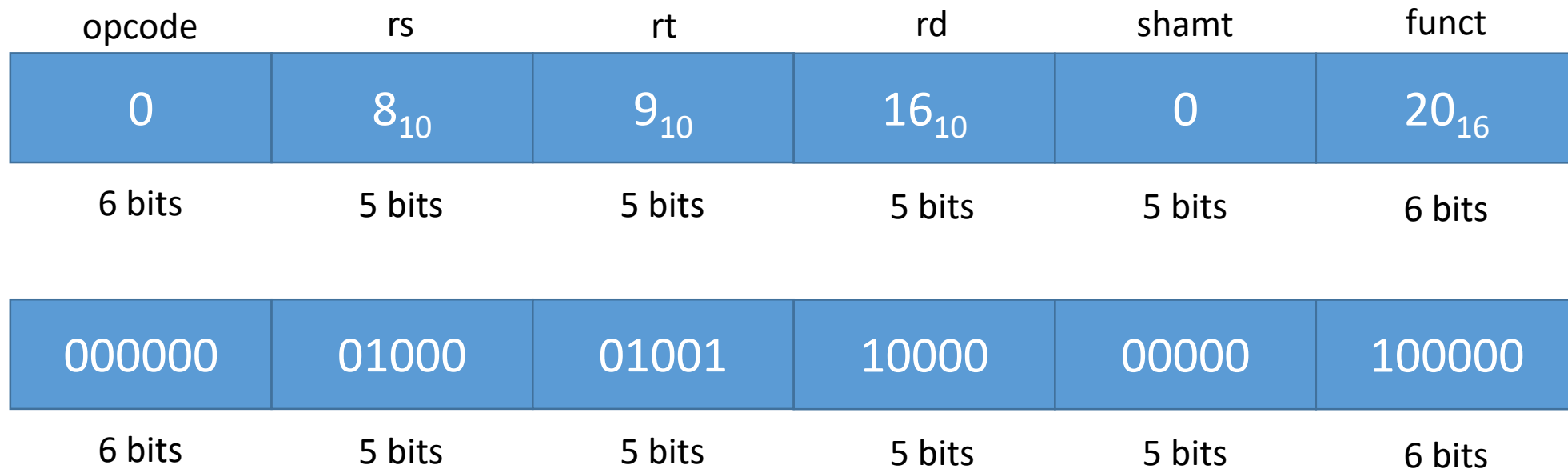


Register Format Instructions

- The **add** instruction does not perform a shift operation and therefore does not utilize the shamt field

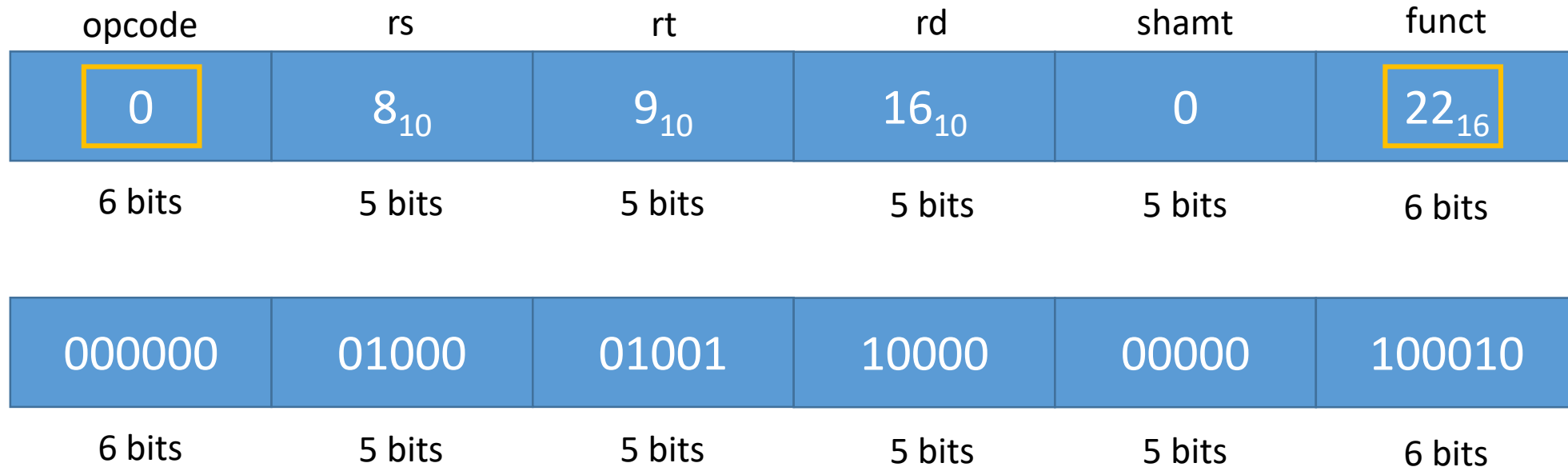


Register Format Instructions



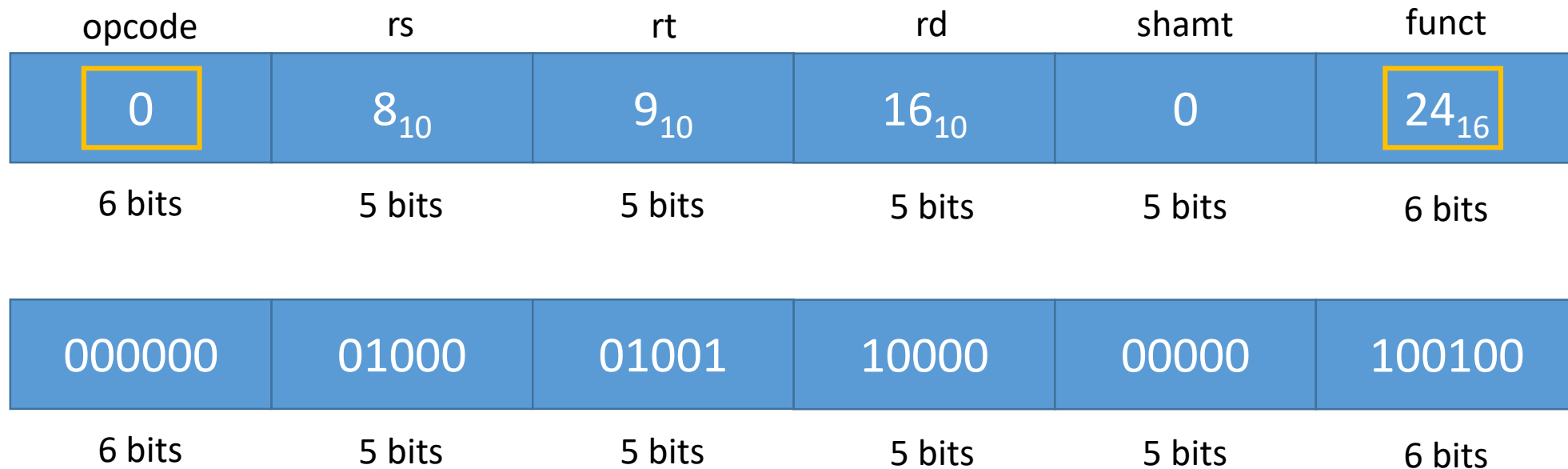
- Assembly Language: **add \$s0, \$t0, \$t1**
- Machine Language: **0000 0001 0000 1001 1000 0000 0010 0000**

Register Format Instructions



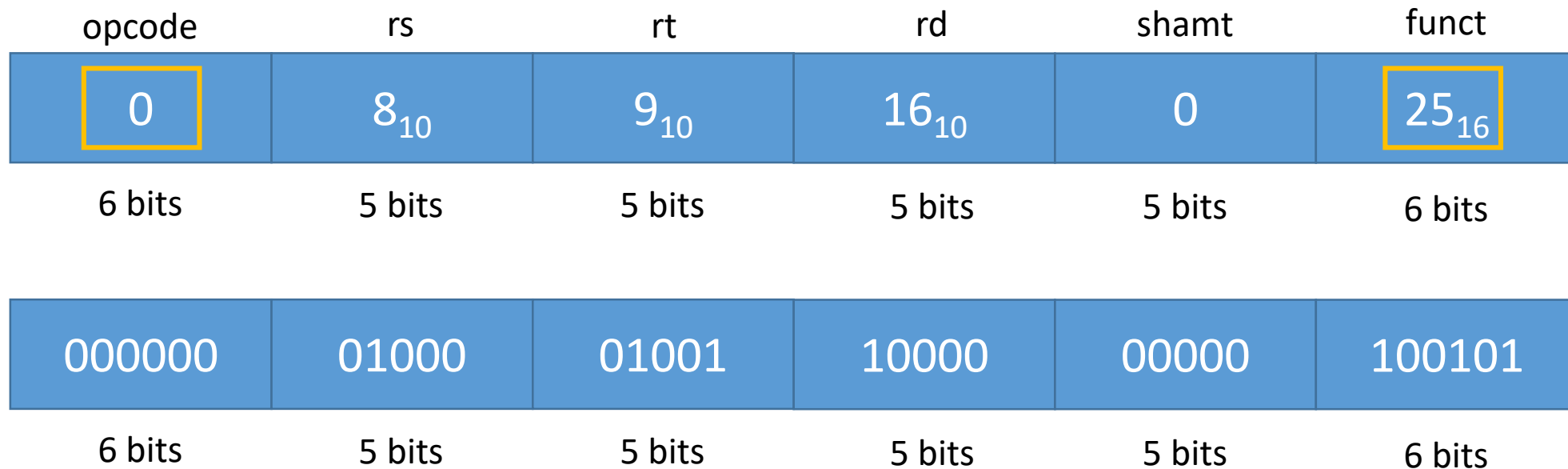
- Assembly Language: **sub** \$s0, \$t0, \$t1
- Machine Language: 0000 0001 0000 1001 1000 0000 0010 0010

Register Format Instructions



- Assembly Language: **and** \$s0, \$t0, \$t1
- Machine Language: 0000 0001 0000 1001 1000 0000 0010 0100

Register Format Instructions



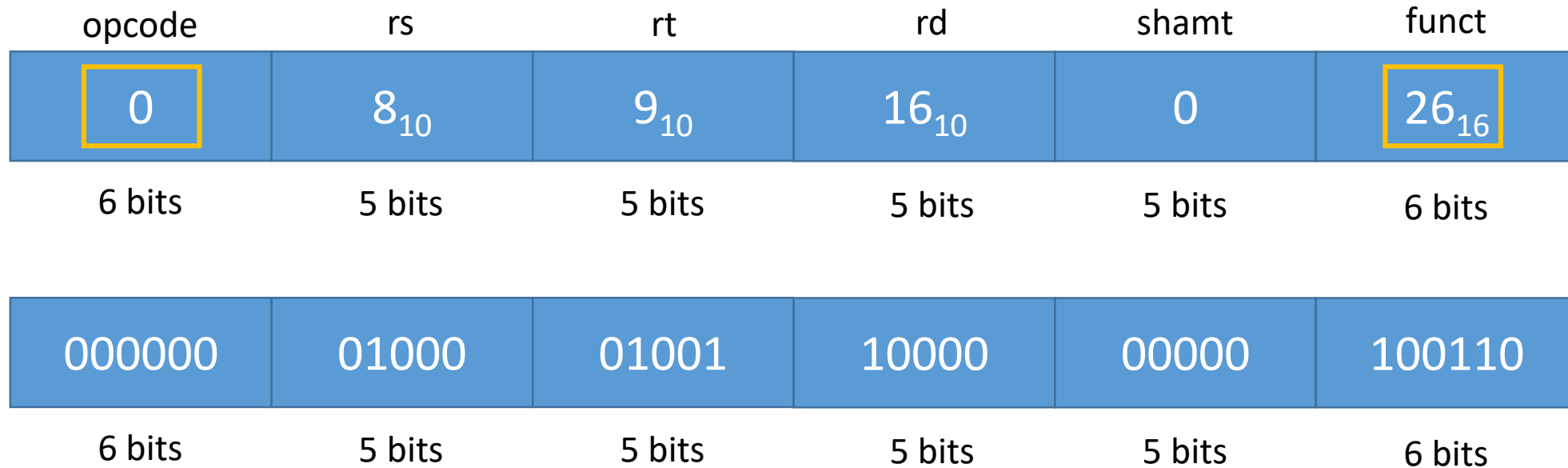
• Assembly Language:

or \$s0, \$t0, \$t1

• Machine Language:

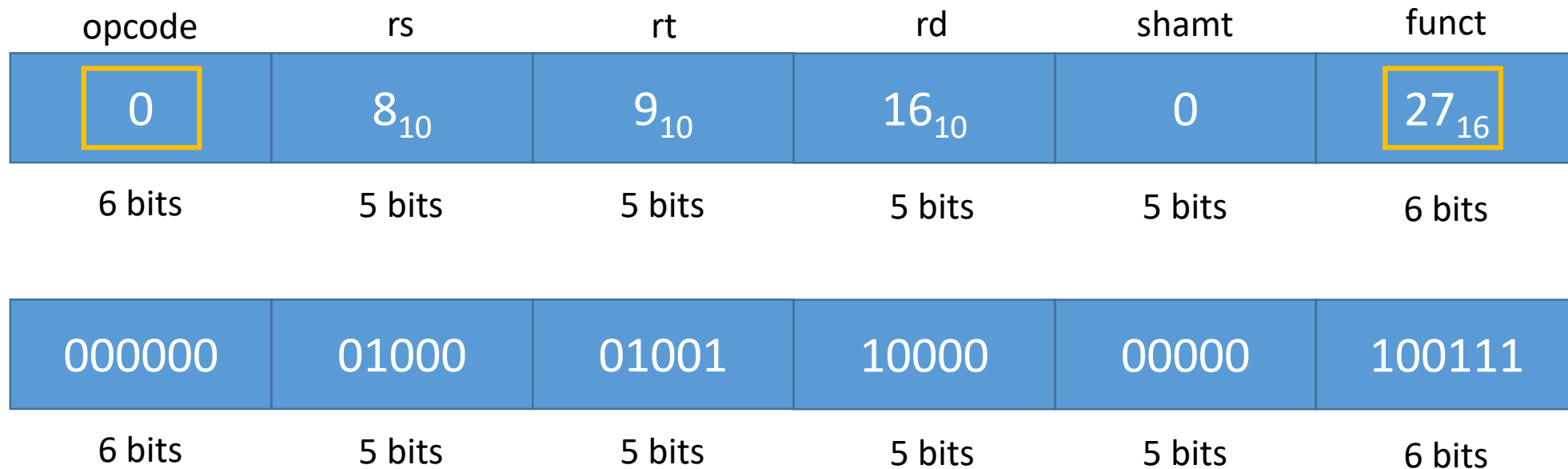
0000 0001 0000 1001 1000 0000 0010 0101

Register Format Instructions



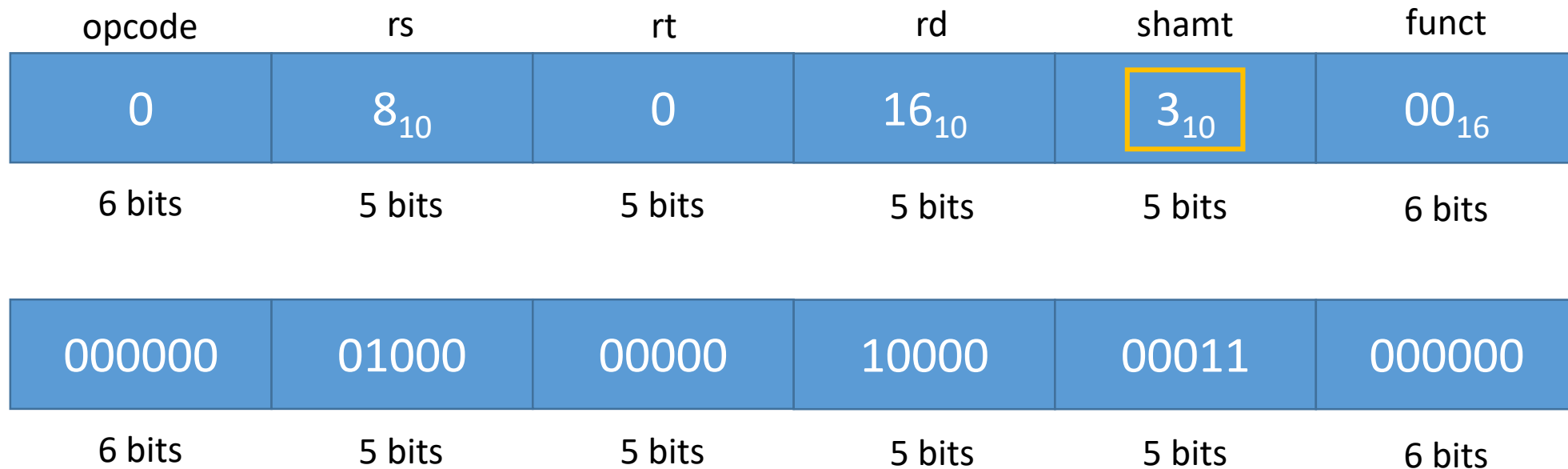
- Assembly Language: **xor** \$s0, \$t0, \$t1
- Machine Language: 0000 0001 0000 1001 1000 0000 0010 0110

Register Format Instructions



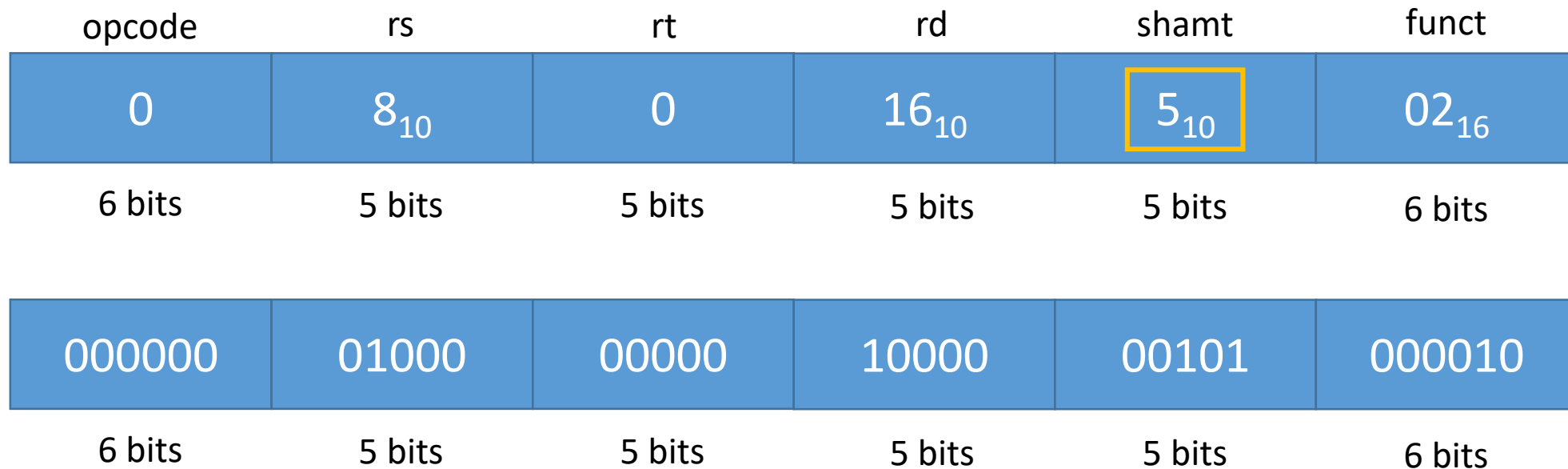
- Assembly Language: **nor** \$s0, \$t0, \$t1
- Machine Language: 0000 0001 0000 1001 1000 0000 0010 0111

Register Format Instructions



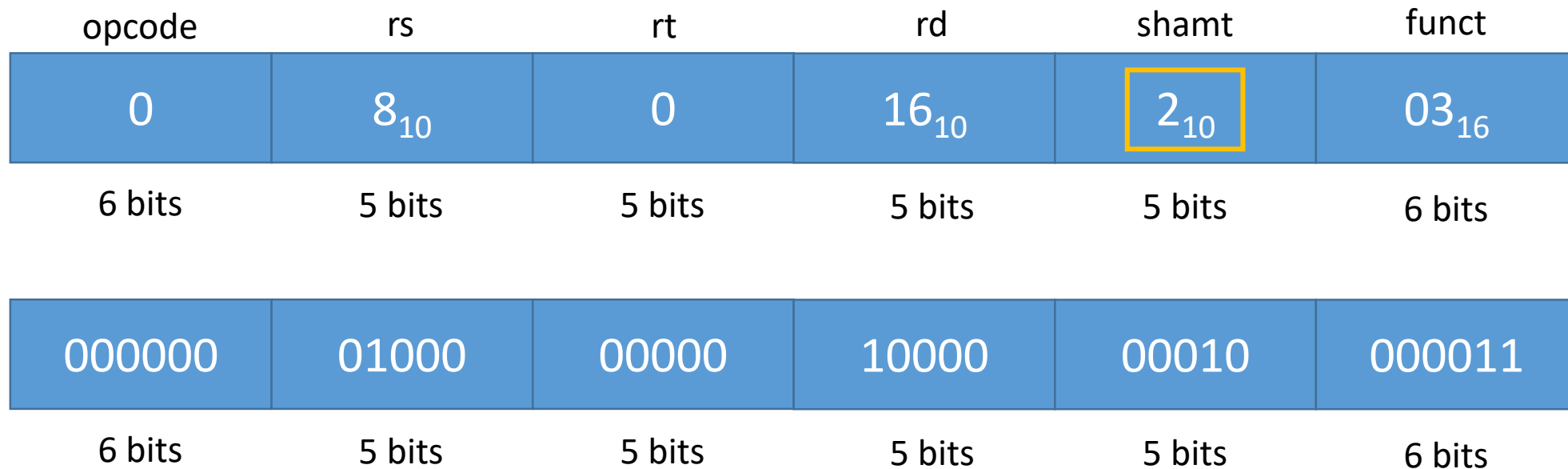
- Assembly Language: **sll \$s0, \$t0, 3**
- Machine Language: **0000 0001 0000 0000 1000 0000 1100 0000**

Register Format Instructions



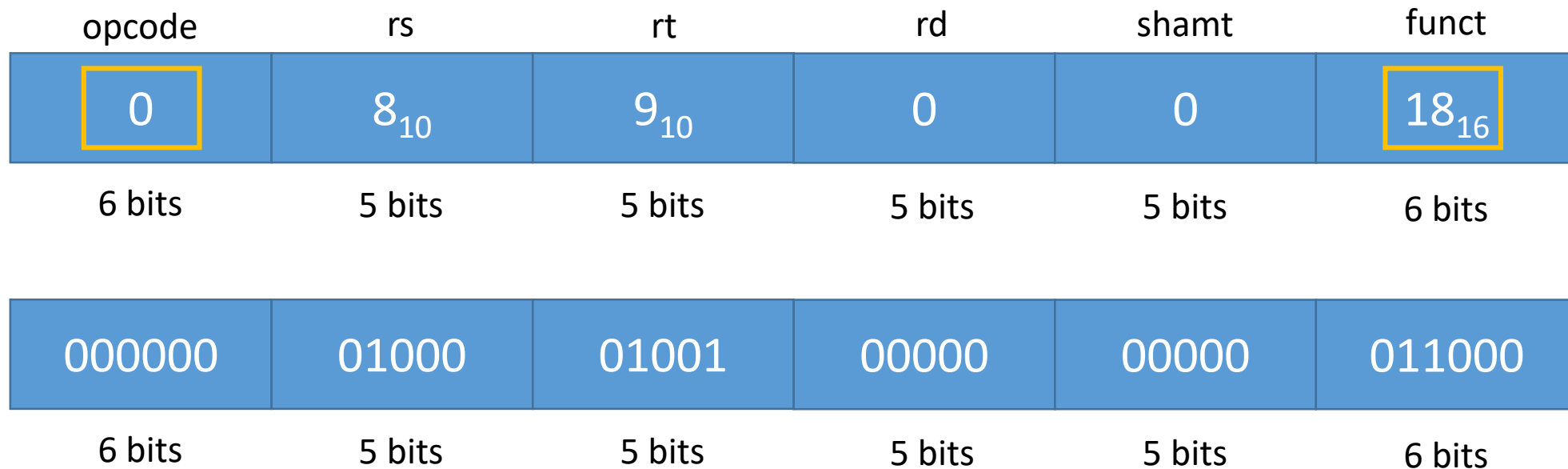
- Assembly Language: **sr1 \$s0, \$t0, 5**
- Machine Language: **0000 0001 0000 0000 1000 0000 1100 0010**

Register Format Instructions



- Assembly Language: **sra \$s0, \$t0, 2**
- Machine Language: **0000 0001 0000 0000 1000 0000 1100 0011**

Register Format Instructions



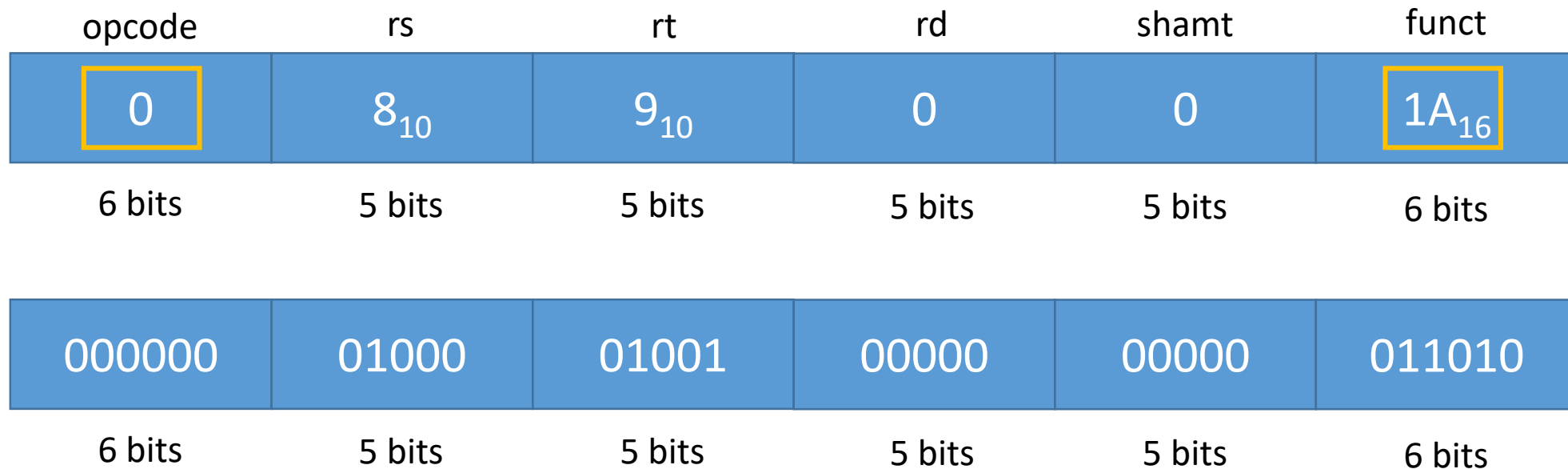
- Assembly Language:

mult \$t0, \$t1

- Machine Language:

0000 0001 0000 1001 0000 0000 0001 1000

Register Format Instructions



• Assembly Language:

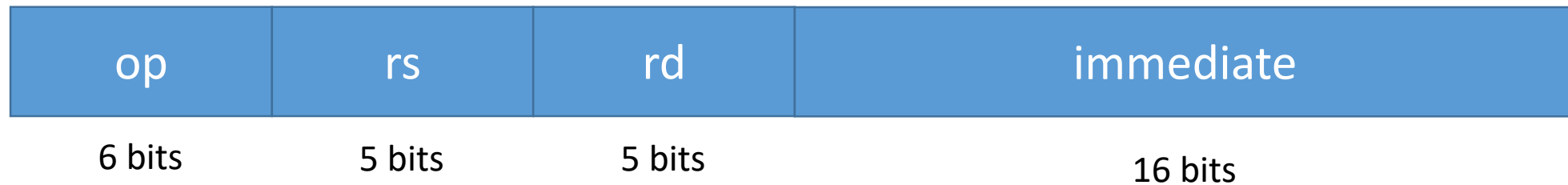
div \$t0, \$t1

• Machine Language:

0000 0001 0000 1001 0000 0000 0001 1010

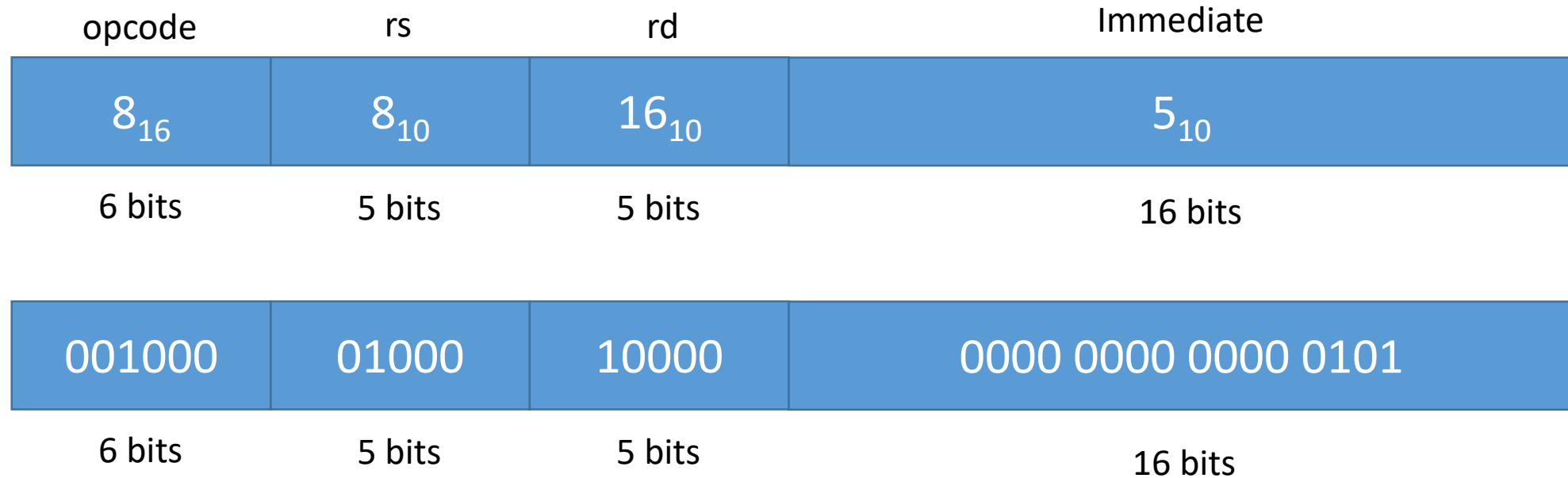
Immediate Format Instructions

- MIPS Immediate Format Instruction



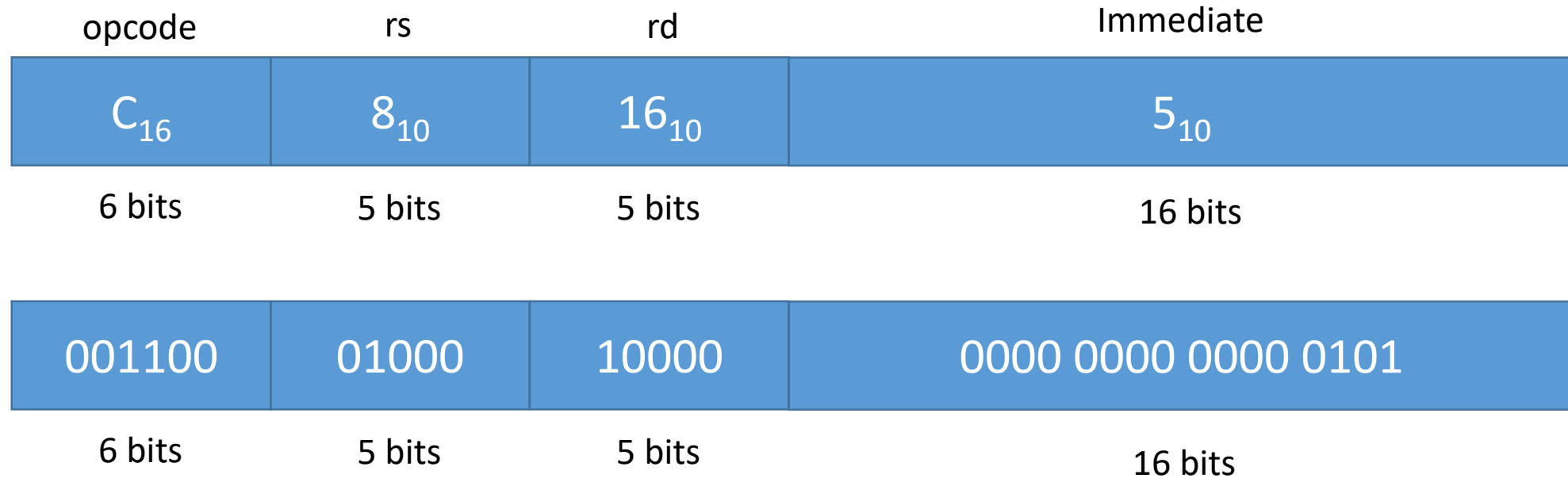
- op – Operation Code (opcode)
- rs – Source register
- rd – Second/Destination register

Immediate Format Instructions



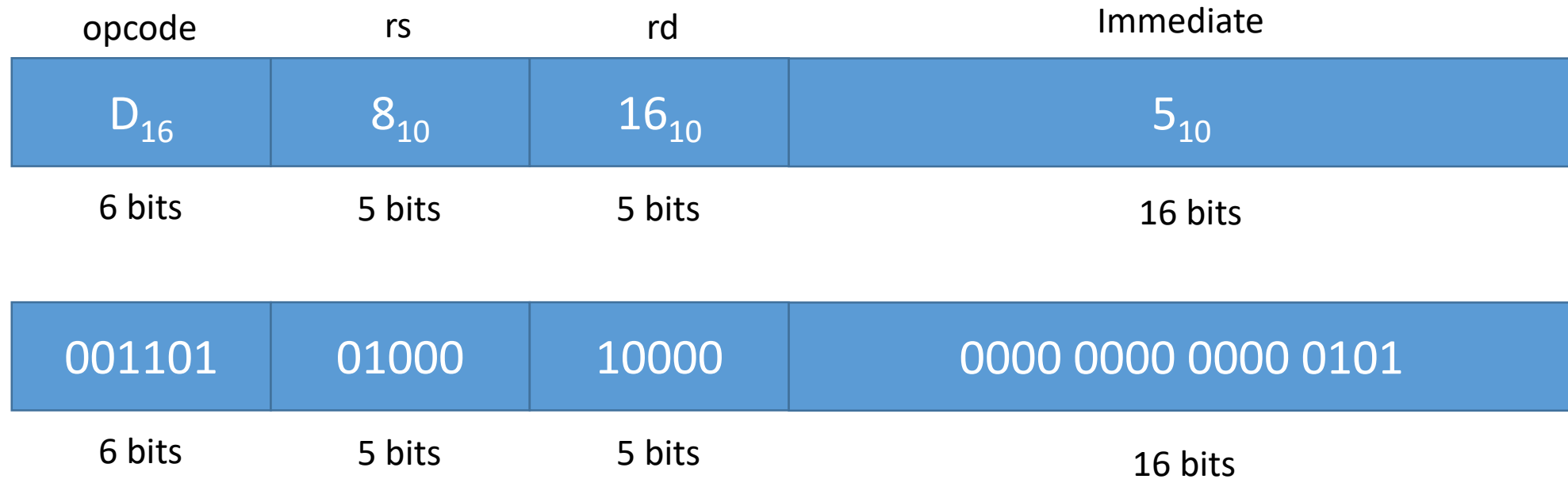
- Assembly Language: **addi \$s0, \$t0, 5**
- Machine Language: **0010 0001 0001 0000 0000 0000 0000 0101**

Immediate Format Instructions



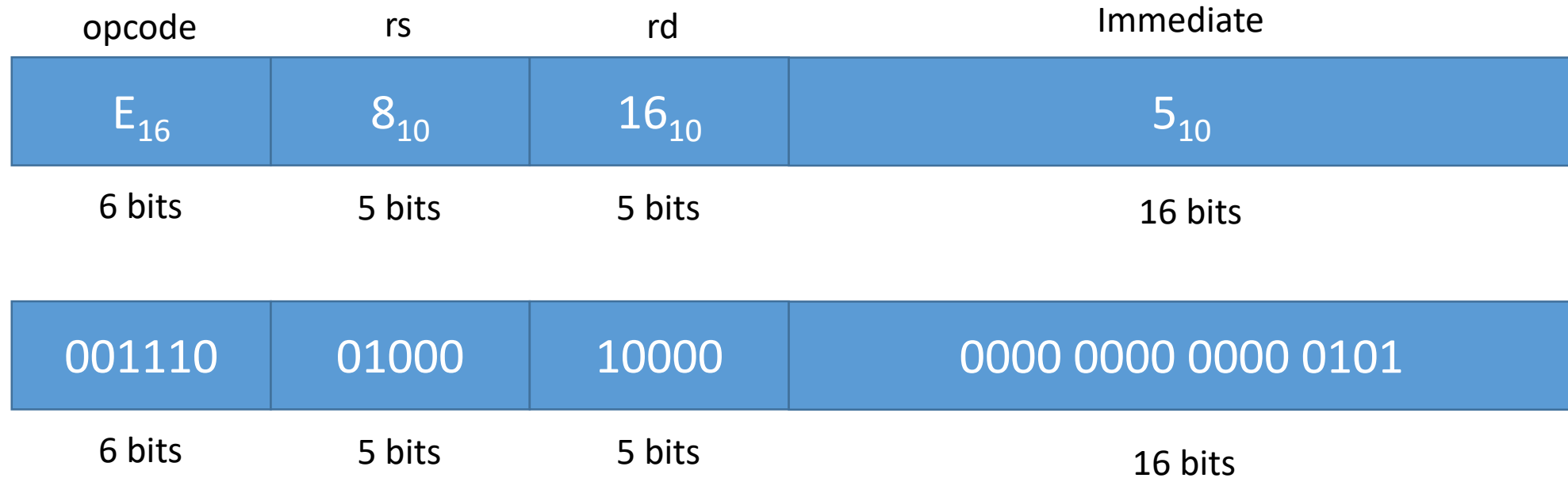
- Assembly Language: **andi \$s0, \$t0, 5**
- Machine Language: **0011 0001 0001 0000 0000 0000 0000 0101**

Immediate Format Instructions



- Assembly Language: **ori \$s0, \$t0, 5**
- Machine Language: **0011 0101 0001 0000 0000 0000 0000 0101**

Immediate Format Instructions

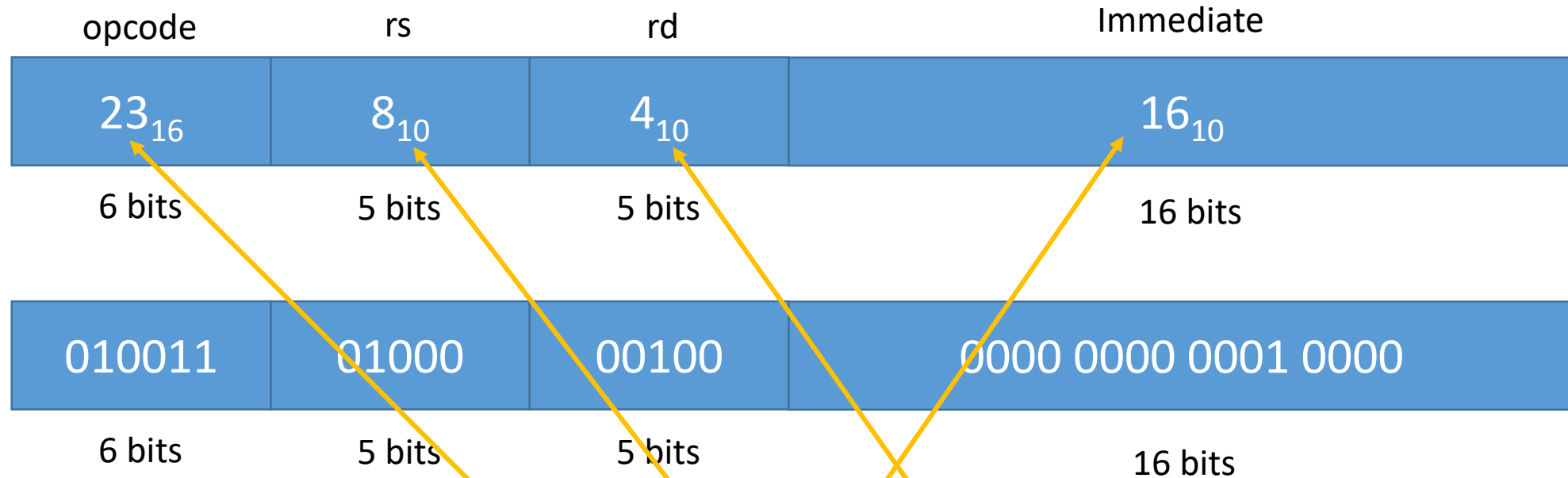


- Assembly Language: **xori \$s0, \$t0, 5**
- Machine Language: **0011 1101 0001 0000 0000 0000 0000 0101**

Immediate Format Instructions

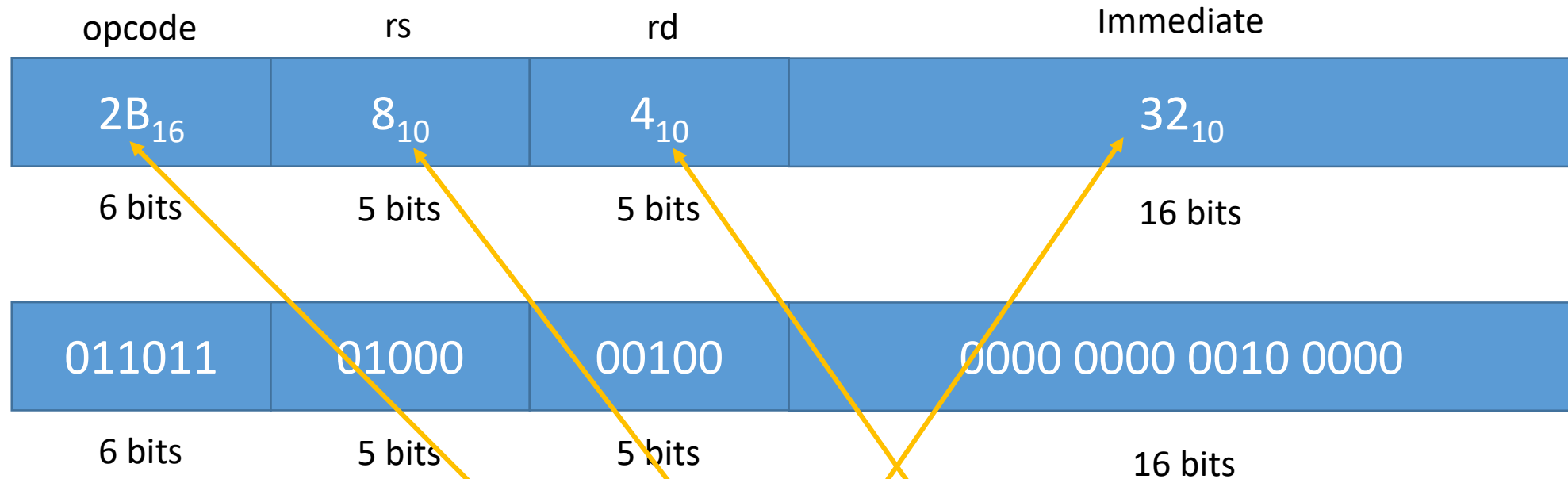
- Recall that memory reference instructions (which are I Format instructions) are used to move data to and from main memory.
- The instructions (**lw** and **sw**) were used for both non-symbolic and symbolic memory reference instructions.
 - Non-symbolic: **lw \$s0, 16(\$a0)**
 - Symbolic: **lw \$s0, mem_label**

Immediate Format Instructions



- Assembly Language: **lw \$t0, 16(\$a0)**
- Machine Language: **0100 1101 0001 0000 0000 0000 0001 0000**

Immediate Format Instructions



- Assembly Language: **sw \$t0, 32(\$a0)**
- Machine Language: **0110 1101 0001 0000 0000 0000 0001 0000**

Immediate Format Instructions

- When using a symbolic memory reference, a MIPS assembler will split a single **lw** or **sw** instruction into two instructions.
- Recall that the memory area in MIPS begins with the address 0x10010000
 - Each label in the data section is an offset from this starting address

Immediate Format Instructions

- We'll use the following data section of a hypothetical program as the example for the next two slides
 - Each word is 4 bytes (32-bits)

.data

value1	.word 55	#Starts at memory address 0x10010000 (offset 0)
value2	.word 106	#Starts at memory address 0x10010004 (offset 4)
value3	.word 99	#Starts at memory address 0x10010008 (offset 8)

Immediate Format Instructions

lw \$s0, value2

- First, we load the starting memory address to a register.
 - Here, we're using the \$at (assembler temporary) register

lui \$at, 0x1001

- This stores 0x10010000 to \$at

- Then, we load the data using the label's offset
 - value2 has an offset of 4

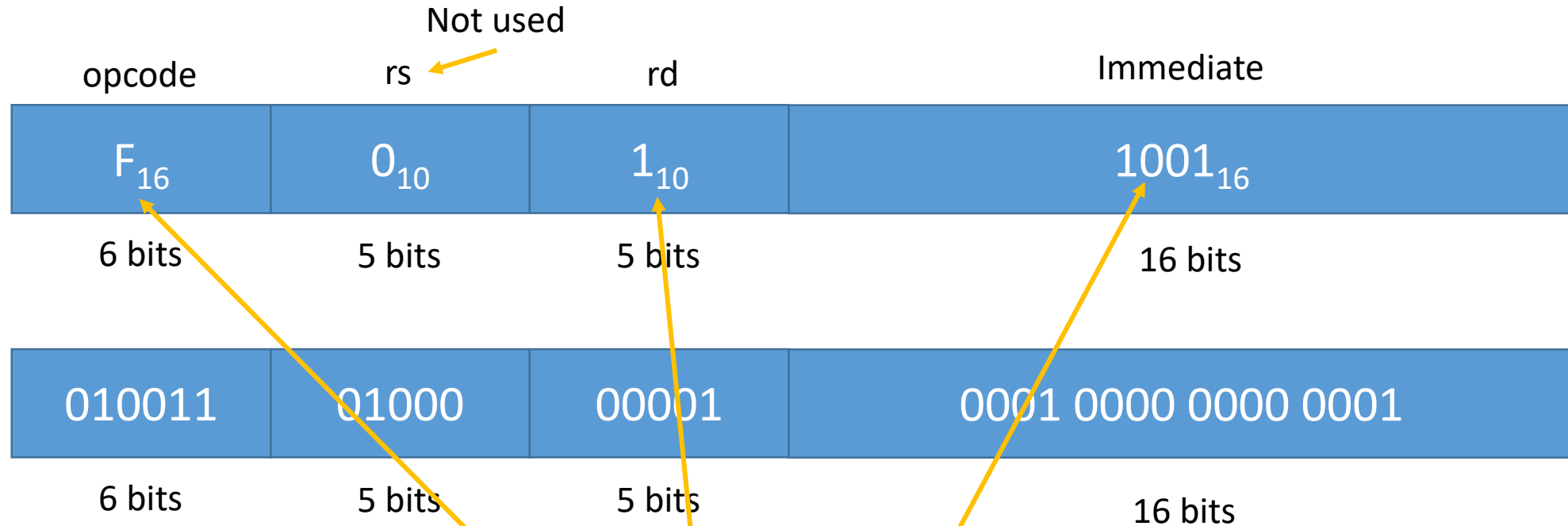
lw \$s0, 4(\$at)

lw \$s0, value2



lui \$at, 0x1001
lw \$s0, 4(\$at)

Immediate Format Instructions



- Assembly Language: **lui \$at, 0x1001**
- Machine Language: **0011 1100 0000 0001 0001 0000 0000 0001**

Immediate Format Instructions

- We previously saw six branching instructions (**beq**, **bgt**, etc.)
- Only **beq** and **bne** are actual MIPS instructions
 - The other four are pseudo-operations (discussed later in the lecture)
- In a branch instruction, we specified the label where the program should branch to.
 - In reality, we are moving X number of instructions forward or backward.

Immediate Format Instructions

.text

a:

li **\$t0, 5**

li **\$t1, 6**

bne **\$t0, \$t1, b**

li **\$t2, 7** 0

li **\$t3, 8** 1

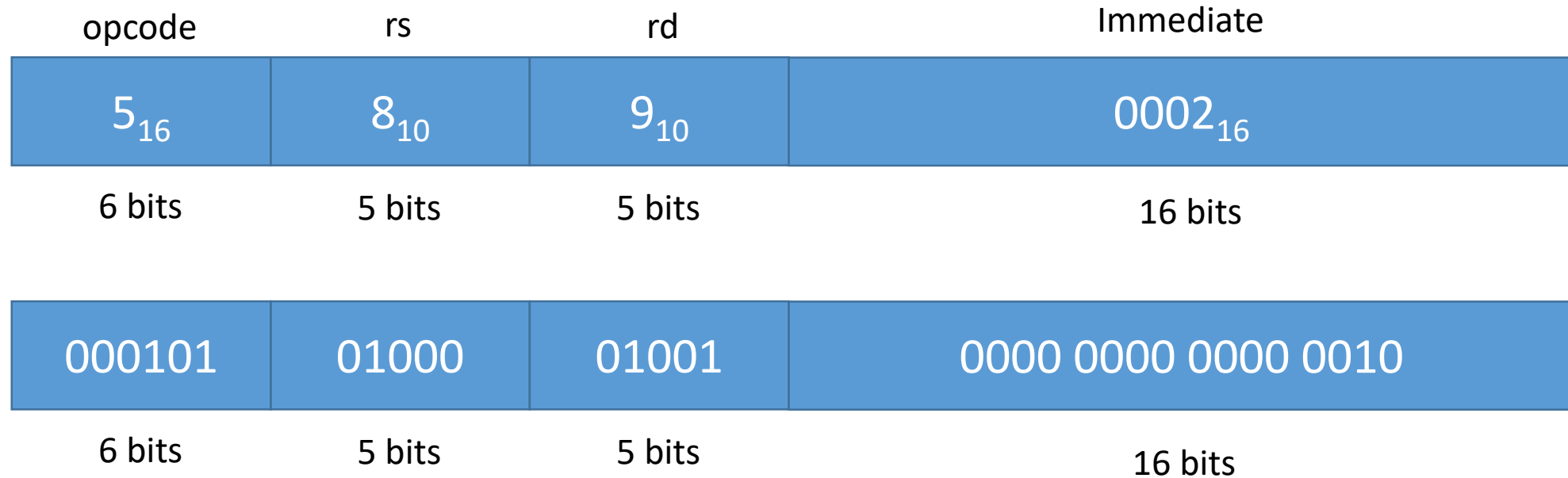
b:

li **\$t4, 9** 2

li **\$t5, 10**

- This branch's condition is true, so the program advances to label b.
- In this program, we advanced two instructions.
 - This begins *after* the instruction following the branch
 - It helps to simply start counting from zero

Immediate Format Instructions



- Assembly Language: **bne \$t0, \$t1, *Label/offset***
- Machine Language: **0001 0101 0000 1001 0000 0000 0000 0010**

Immediate Format Instructions

.text

a:

li \$t0, 5 4

li \$t1, 6 3

b:

li \$t4, 9 2

li \$t5, 9 1

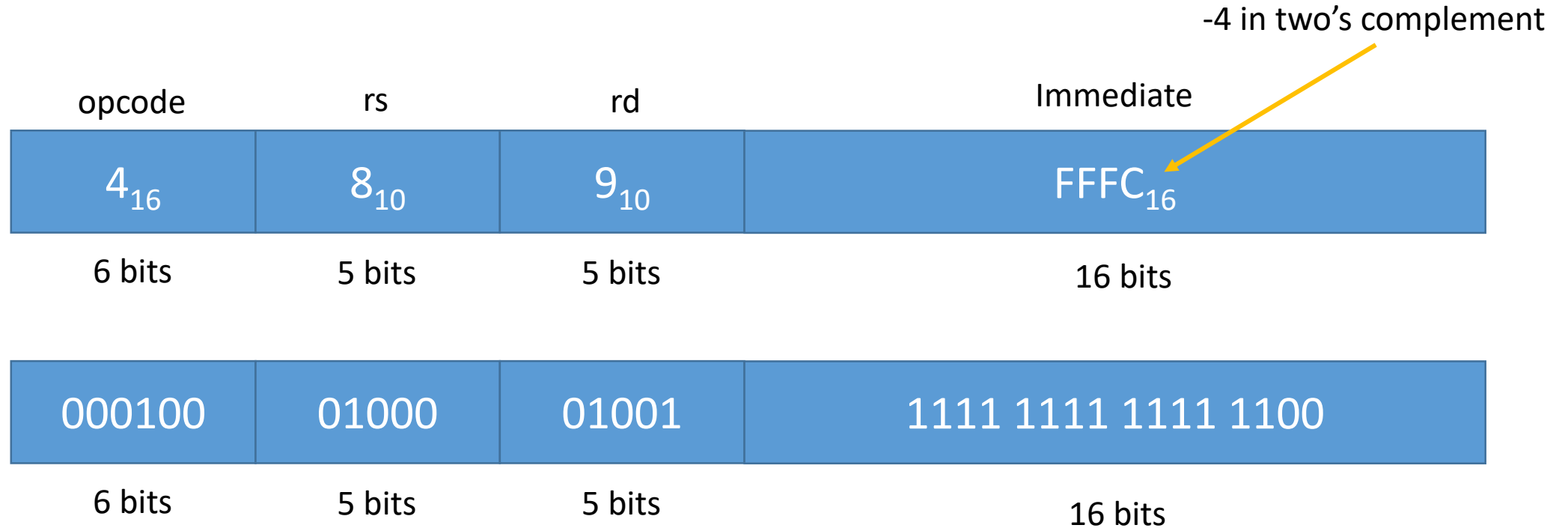
beq \$t0, \$t1, a 0

li \$t2, 7

li \$t3, 8

- This branch's condition is true, so the program goes back to label a.
- In this program, we went back four instructions.
 - This begins *starting with* the instruction preceding the branch instruction
- The offset is -4

Immediate Format Instructions



- Assembly Language: **beq \$t0, \$t1, *Label/offset***
- Machine Language: **0001 0001 0000 1001 1111 1111 1111 1100**

Jump Format Instructions

- MIPS Jump Format Instruction



- op – Operation Code (opcode)
- MIPS only has two Jump Format Instructions: **j** and **jal**

Jump Format Instructions

.text

a:

li \$t0, 5

li \$t1, 6

j b

li \$t2, 7

li \$t3, 8

b:

li \$t4, 9

li \$t5, 10

- Unlike the relative addresses (offsets) used by branch instructions, jump instructions use absolute addresses.
- We'll say the address of the instruction labeled by b is 0x0040001C
- This is a 32-bit address, which won't fit in the 26-bit jump address portion of a J Format Instruction

Jump Format Instructions

- An instruction address will always end with 00_{16} , so we can omit those eight bits.
 - Memory is byte addressable and every instruction is 1 word/4 bytes/32-bits

0x0040001C

0000 0000 0100 0000 0000 0000 0001 1100

0000 0000 0100 0000 0000 0000 0001 1100

0000 0000 0100 0000 0000 0000 0001 1100

00 0001 0000 0000 0000 0000 0111

0x0100007

Jump Format Instructions

- Can be achieved one of two ways:
 1. Divide the address by 4 and take the lower 26 bits of the result
 - $0x0040001C / 4_{10} = 0x00100007 = \underline{0x010007}$
 2. Right shift the address by two bits, take the lower 26 bits of the result
 - Similar to what we did on the last slide

0x0040001C

0000 0000 0100 0000 0000 0000 0001 1100

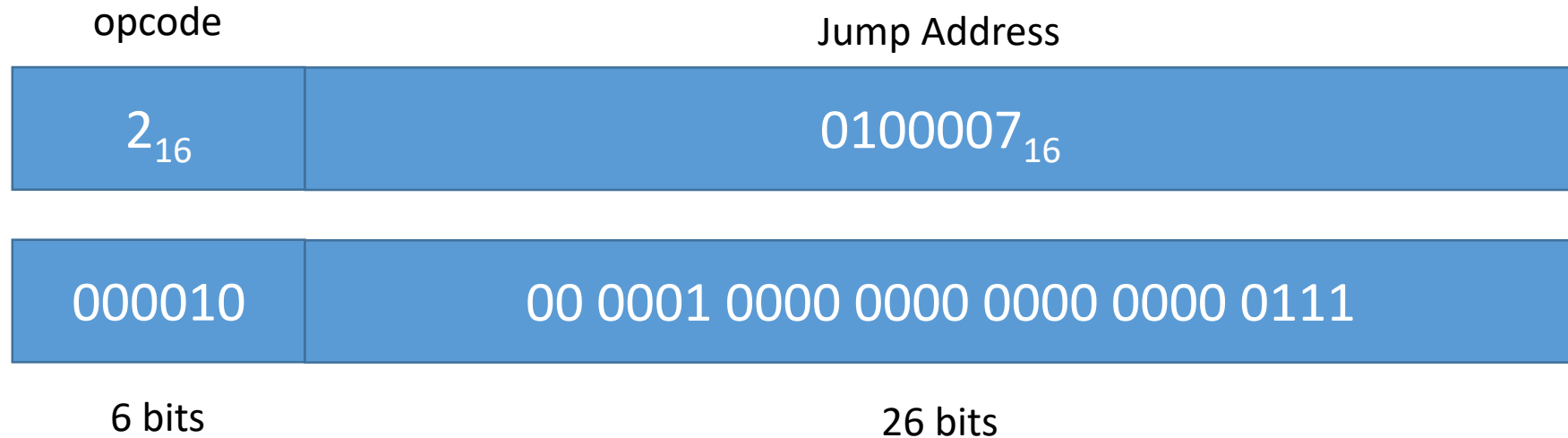
0000 0000 0001 0000 0000 0000 0000 0111

0000 0000 0001 0000 0000 0000 0000 0111

00 0001 0000 0000 0000 0000 0000 0111

0x0100007

Jump Format Instructions



- Assembly Language: **j b #address 0x0040001C**
- Machine Language: **0000 1000 0001 0000 0000 0000 0000 0111**

Pseudo Operations

- An **pseudo operation** (or **pseudo op**) is an instruction/operation that does not correspond to any machine language instruction.
 - Pseudo ops exist for convenience, not out of necessity
- The assembler translates the pseudo op into one or more real instructions.
 - Each assembler has its own pseudo operations
 - The pseudo ops shown in the next few slides are specific to the MARS assembler, but can be found in other MIPS assemblers as well

Pseudo Operations

- The load immediate (**li**) instruction is a pseudo op.

li \$t0, 5

- The MARS assembler converts this to an **addi** instruction

addi \$t0, \$0, 5

- Adds the constant with 0 and assigns the result to the specified register

li \$t0, 5



addi \$t0, \$0, 5

Pseudo Operations

- The load address (**la**) instruction is a pseudo op.

la \$t0, 8(\$t1)

- The MARS assembler converts this to an **addi** instruction

addi \$t0, \$t1, 8

- Adds the constant to the address in \$t1 and assigns the result to \$t0

la \$t0, 8(\$t1)



addi \$t0, \$t1, 8

Pseudo Operations

- The **move** instruction is a pseudo op.

move \$t0, \$t1

- The MARS assembler converts this to an **add** instruction

add \$t0, \$0, \$t1

- Adds the value in \$t1 with 0 and assigns the result to \$t0

move \$t0, \$t1



add \$t0, \$0, \$t1

Pseudo Operations

- The **not** instruction is a pseudo op.

not \$t0, \$t1

- The MARS assembler converts this to a **nor** instruction

nor \$t0, \$t1, \$0

- nor's the value in \$t1 with 0 and assigns the result to \$t0

not \$t0, \$t1



nor \$t0, \$t1, \$0

Pseudo Operations

- The branch if less than (**blt**) instruction is a pseudo op.

blt \$t0, \$t1, label

- The MARS assembler converts this to two instructions:

slt \$at, \$t0, \$t1

bne \$at, \$0, label

- \$at is set to 1 if \$t1 is less than \$t0
- If \$at is not equal to 0, then \$t1 was less than \$t0 and the branch occurs

blt \$t0, \$t1, label



slt \$at, \$t0, \$t1
bne \$at, \$0, label

Pseudo Operations

- The branch if less than or equal (**ble**) instruction is a pseudo op.

ble \$t0, \$t1, label

- The MARS assembler converts this to two instructions:

slt \$at, \$t1, \$t0

beq \$at, \$0, label

- \$at is set to 1 if \$t0 is less than \$t1
- If \$at is equal to 0, then \$t1 was less than or equal to \$t0 and the branch occurs

ble \$t0, \$t1, label



slt \$at, \$t1, \$t0
beq \$at, \$0, label

Pseudo Operations

- The branch if greater than (**bgt**) instruction is a pseudo op.

bgt \$t0, \$t1, label

- The MARS assembler converts this to two instructions:

slt \$at, \$t1, \$t0

bne \$at, \$0, label

- \$at is set to 1 if \$t0 is less than \$t1
- If \$at is not equal to 0, then \$t1 was greater than \$t0 and the branch occurs

bgt \$t0, \$t1, label



slt \$at, \$t1, \$t0
bne \$at, \$0, label

Pseudo Operations

- The branch if greater than or equal (**bge**) instruction is a pseudo op.

bge \$t0, \$t1, label

- The MARS assembler converts this to two instructions:

slt \$at, \$t0, \$t1

beq \$at, \$0, label

- \$at is set to 1 if \$t1 is less than \$t0
- If \$at is equal to 0, then \$t1 was greater than or equal to \$t0 and the branch occurs

bge \$t0, \$t1, label



slt \$at, \$t0, \$t1

beq \$at, \$0, label

Floating Point Instruction Formats

- MIPS has two floating point instruction formats:

- **Floating Point Register Format (FR Format)**

add.s \$f2, \$f0, \$f1

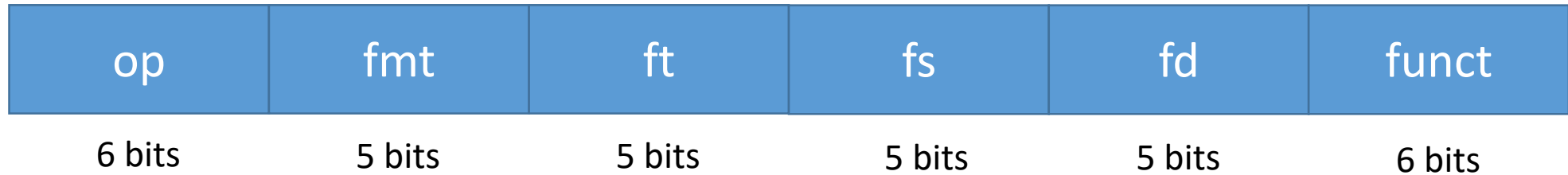
- **Floating Point Immediate Format (FI Format)**

- Only used for conditional branch instructions

bc1t label

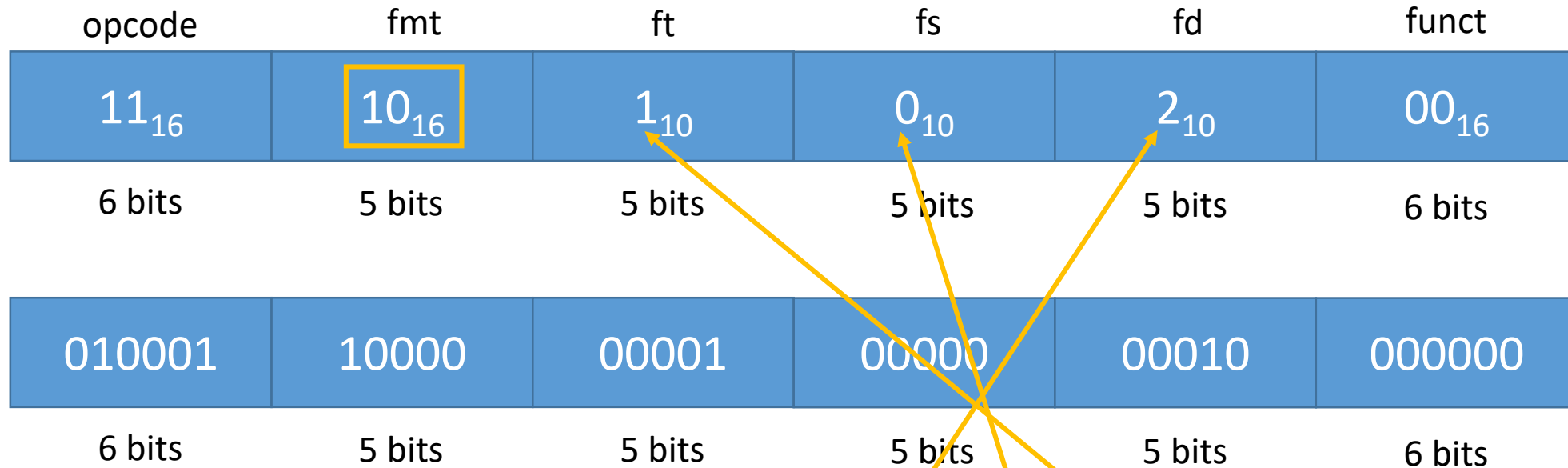
Floating Point Register Format Instructions

- MIPS Floating Point Register Format Instruction



- op – Opcode
- fmt – Format (indicates single or double precision)
- ft – Second source register (right operand)
- fs – First source register (left operand)
- fd – Destination register
- funct – Function code (specifies a variant of the opcode)

Floating Point Register Format Instructions



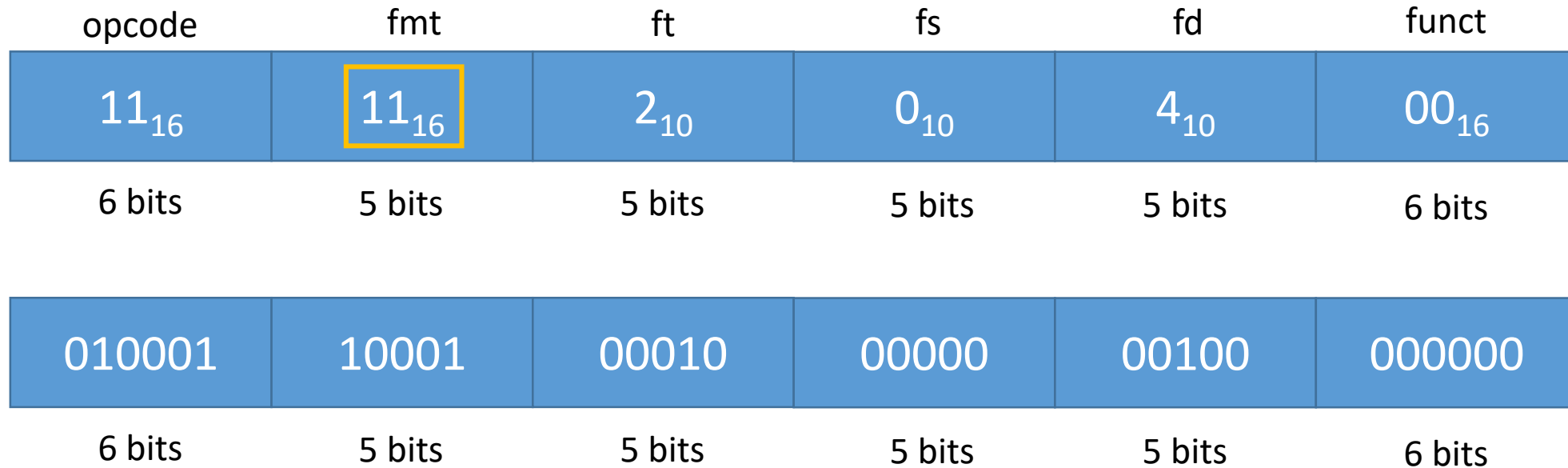
- Assembly Language:

add.s \$f2, \$f0, \$f1

- Machine Language:

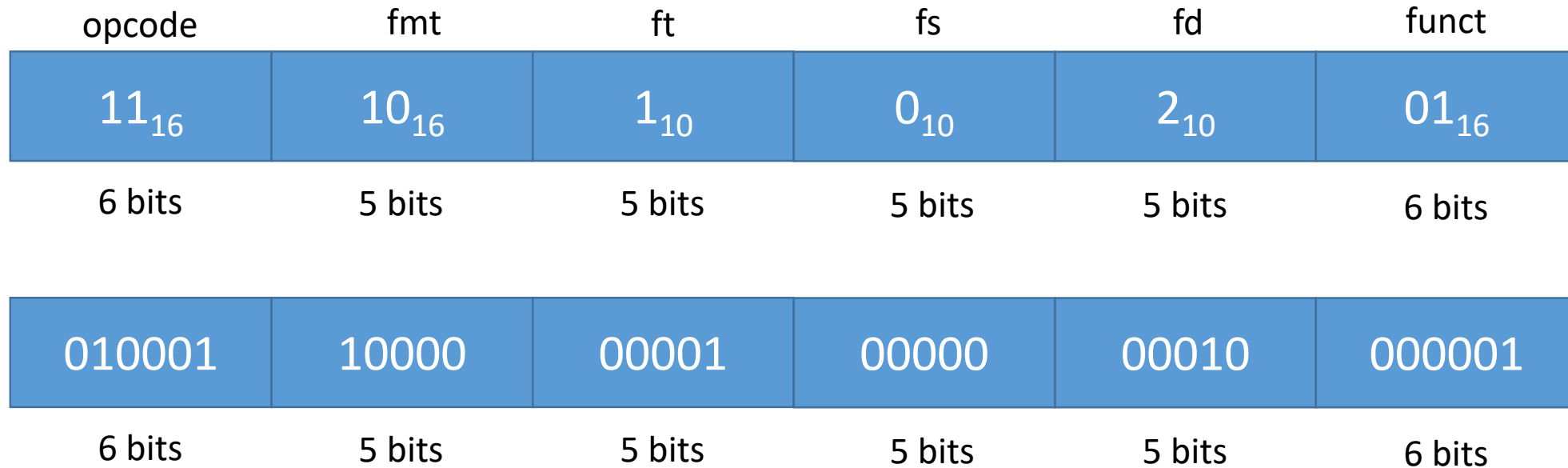
0100 0110 0000 0001 0000 0000 1000 0000

Floating Point Register Format Instructions



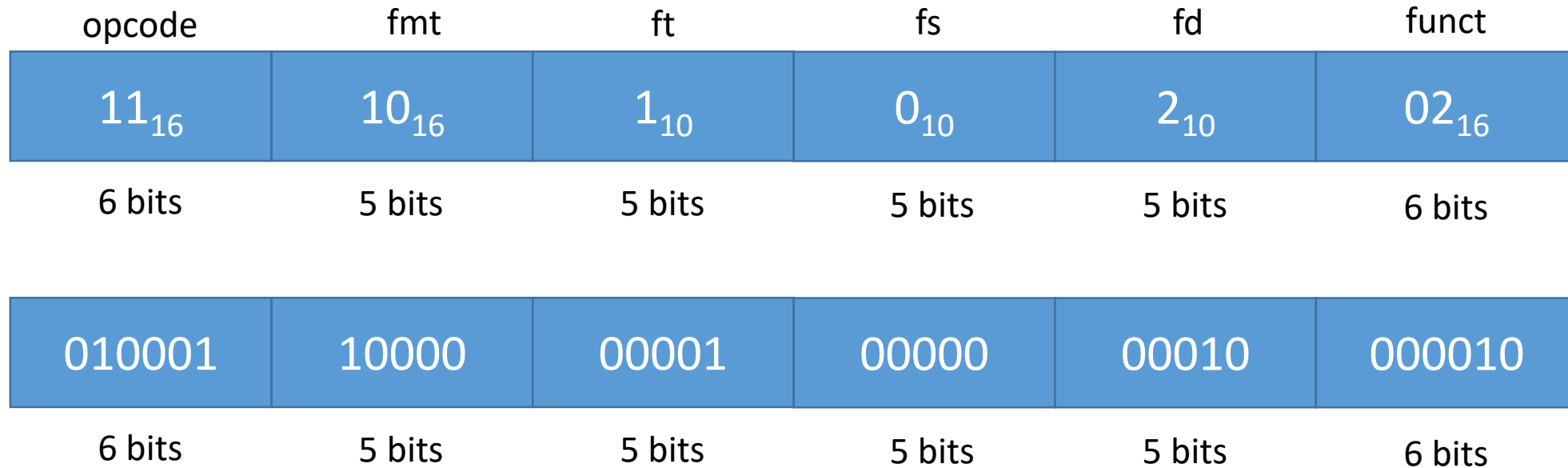
- Assembly Language: `add.d $f4, $f0, $f2`
- Machine Language: `0100 0110 0010 0010 0000 0001 0000 0000`

Floating Point Register Format Instructions



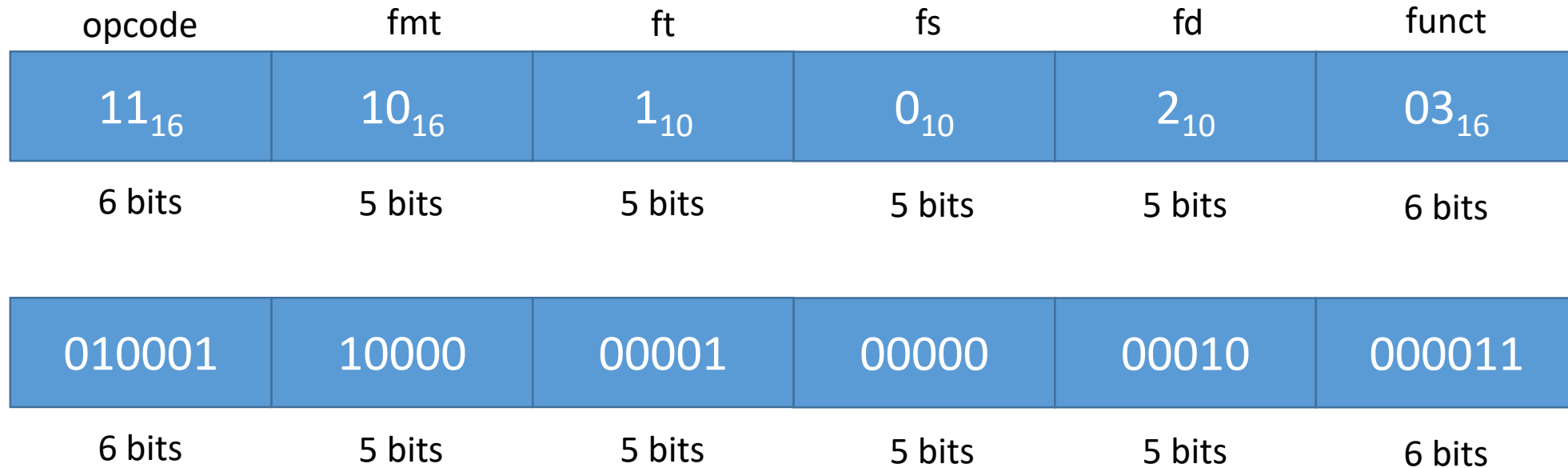
- Assembly Language: **sub.s \$f2, \$f0, \$f1**
- Machine Language: **0100 0110 0000 0001 0000 0000 1000 0001**

Floating Point Register Format Instructions



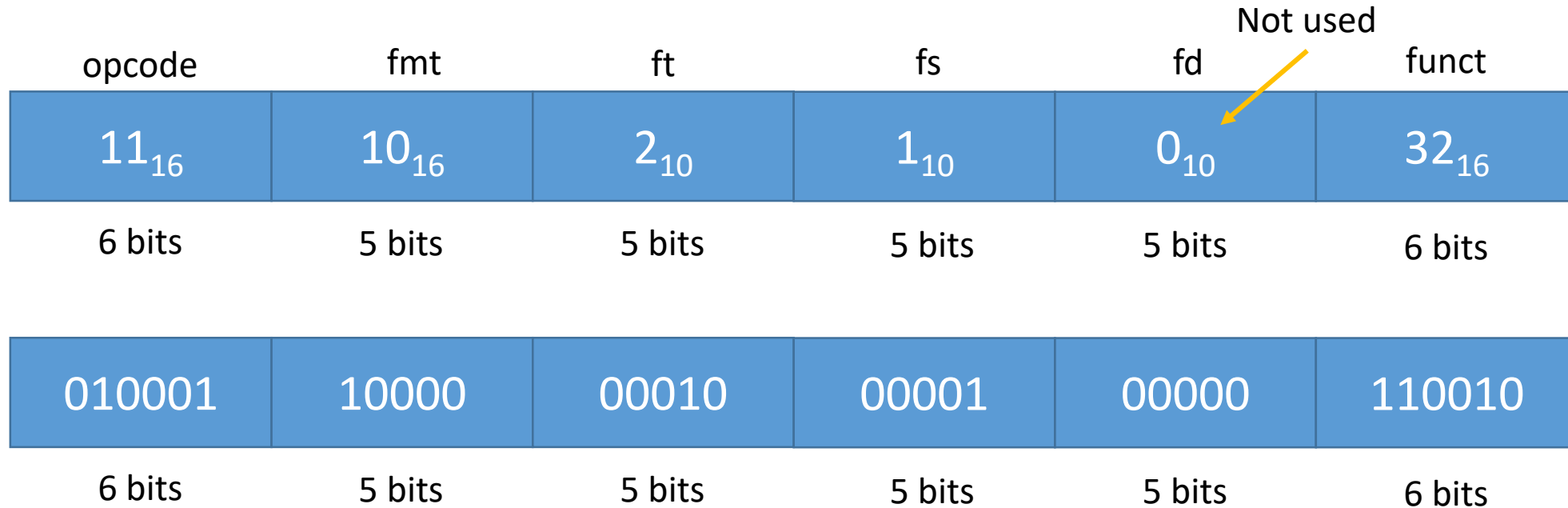
- Assembly Language: **mul.s \$f2, \$f0, \$f1**
- Machine Language: **0100 0110 0000 0001 0000 0000 1000 0010**

Floating Point Register Format Instructions



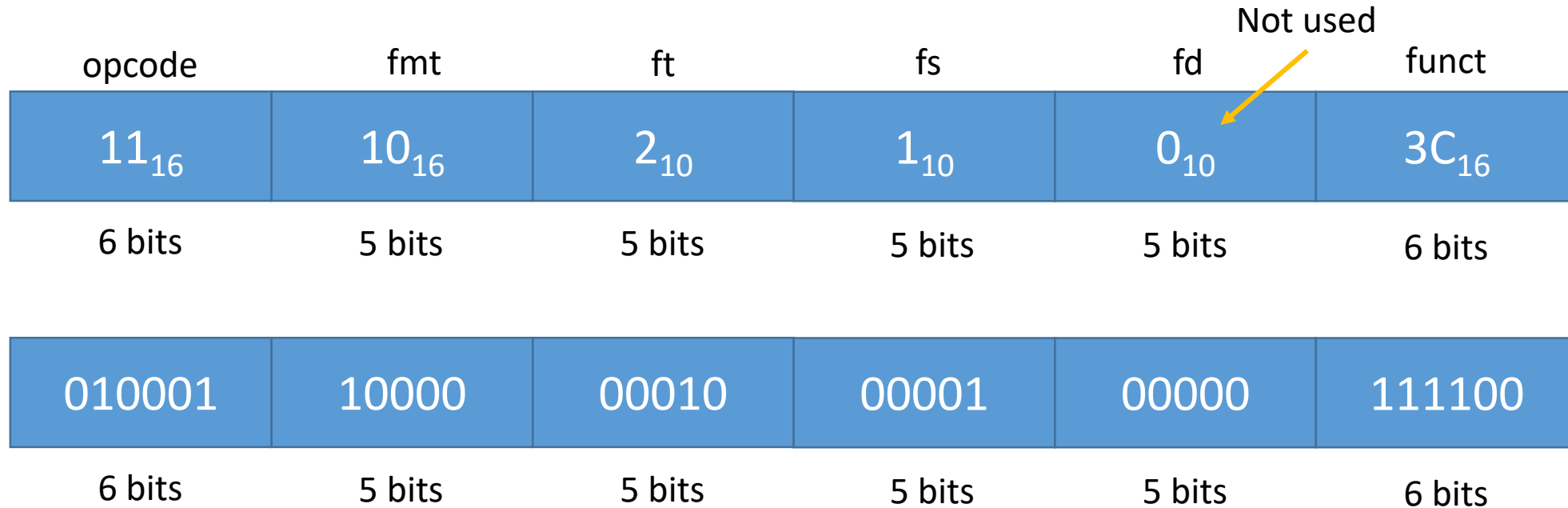
- Assembly Language: **div.s \$f2, \$f0, \$f1**
- Machine Language: **0100 0110 0000 0001 0000 0000 1000 0011**

Floating Point Register Format Instructions



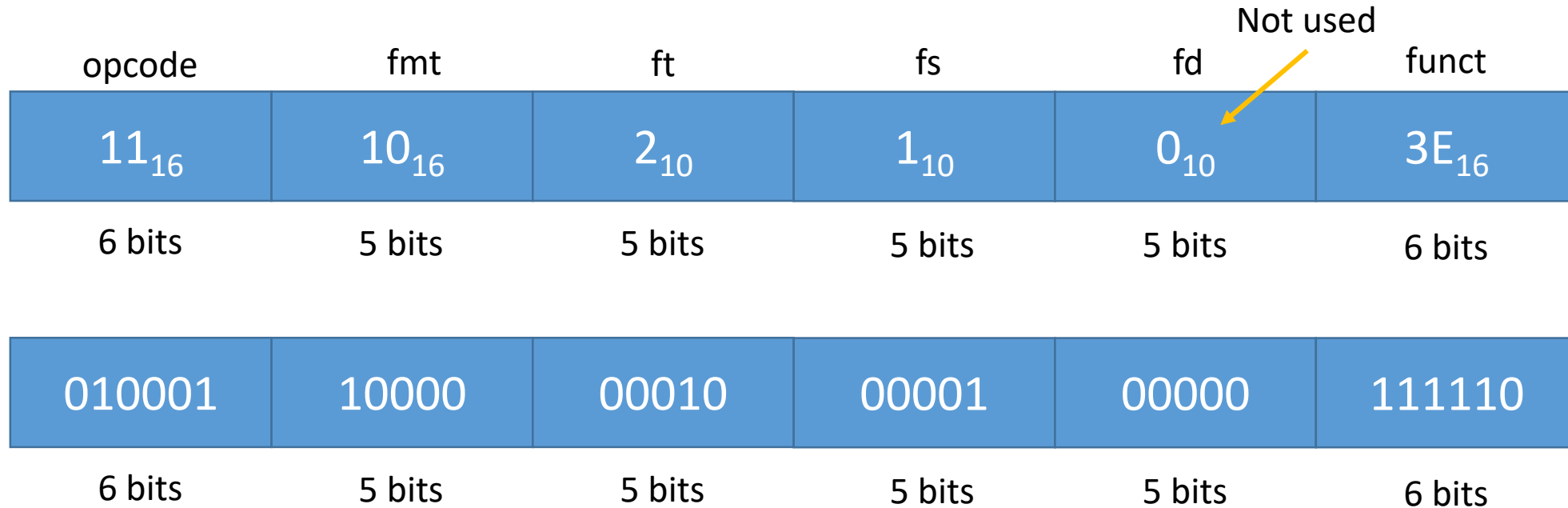
- Assembly Language: **c.eq.s \$f1, \$f2**
- Machine Language: **0100 0110 0000 0010 0000 1000 0011 0010**

Floating Point Register Format Instructions



- Assembly Language: **c.lt.s \$f1, \$f2**
- Machine Language: **0100 0110 0000 0010 0000 1000 0011 1100**

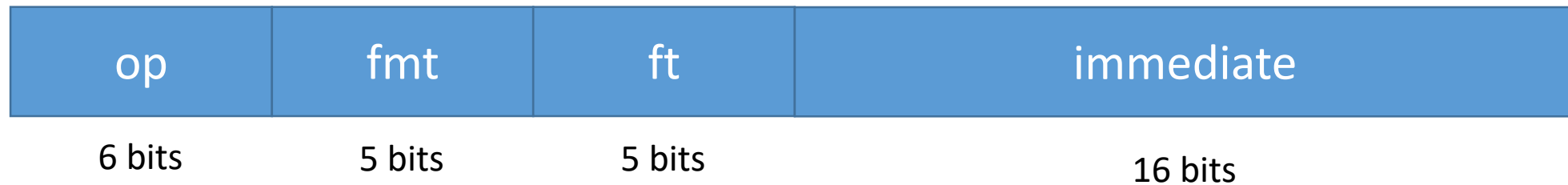
Floating Point Register Format Instructions



- Assembly Language: **c.le.s \$f1, \$f2**
- Machine Language: **0100 0110 0000 0010 0000 1000 0011 1110**

Floating Point Immediate Format Instructions

- MIPS Floating Point Immediate Format Instruction

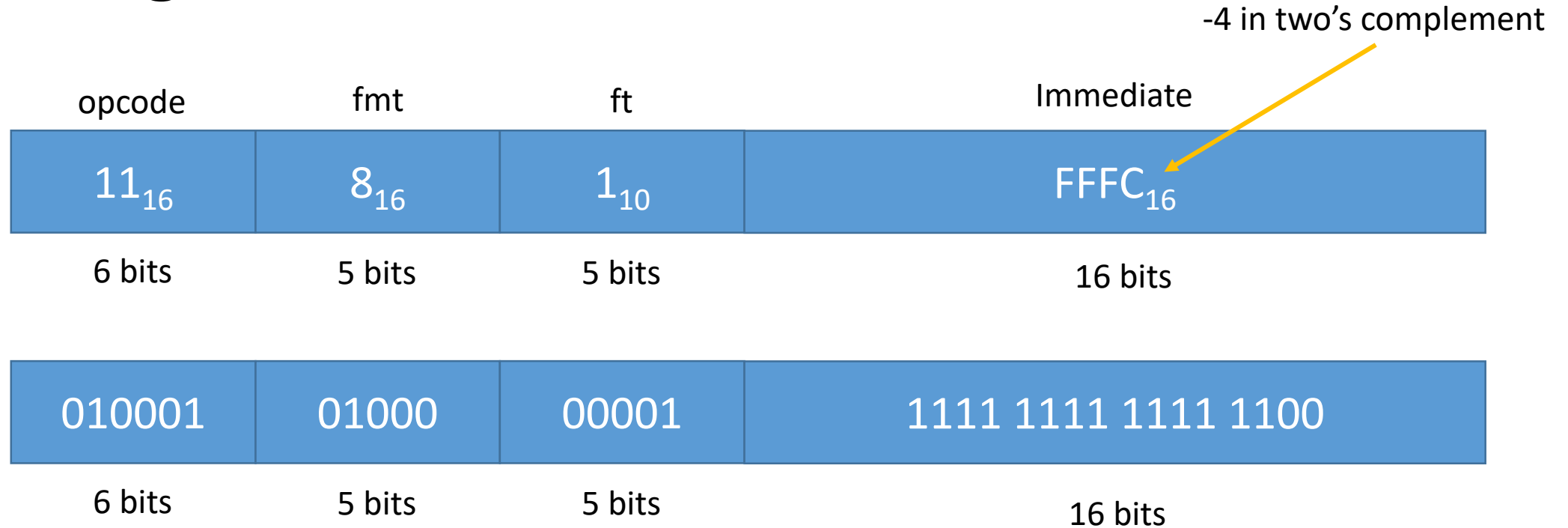


- op – Opcode
- fmt – Format
- ft – Source register
- immediate – Second/Destination register

Floating Point Immediate Format Instructions

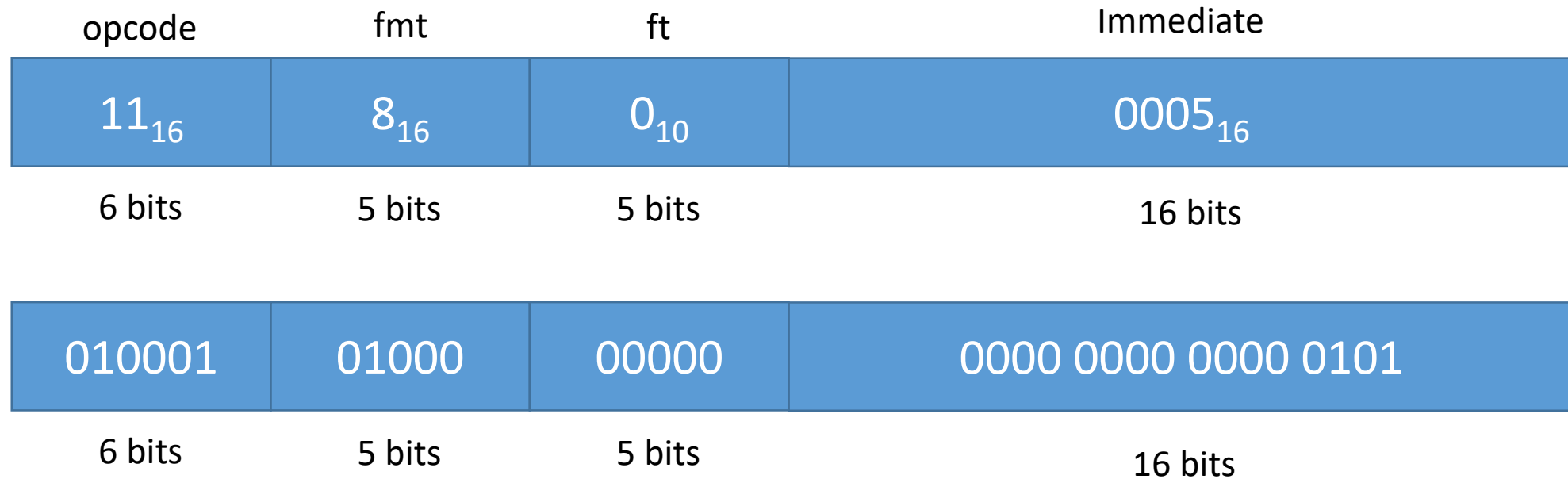
- We previously saw the two branching instructions for comparisons of floating point numbers
 - **bc1f** and **bc1t**
 - Relies on the value of the condition code, set by a conditional instruction like **c.le.s** or **c.eq.d**
- Branching to a label works in a similar fashion to the previous examples of advancing or backing up X number of instructions.

Floating Point Immediate Format Instructions



- Assembly Language: **bc1t** *Label/offset*
- Machine Language: **0100 0101 0000 0001 1111 1111 1111 1100**

Floating Point Immediate Format Instructions

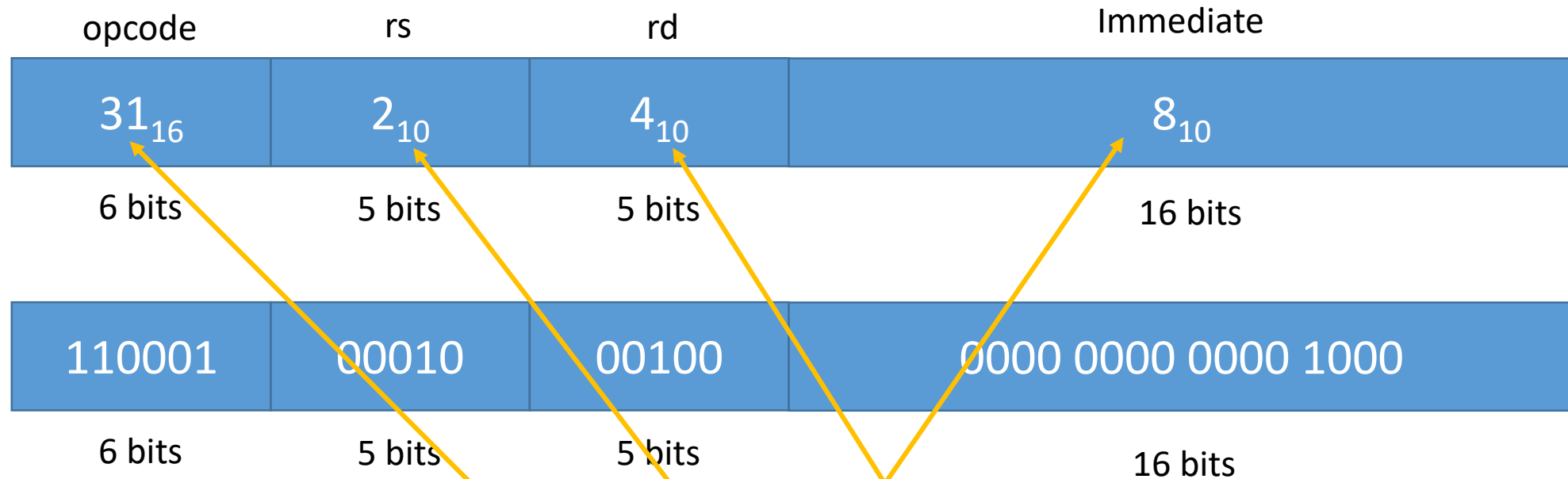


- Assembly Language: **bc1f** *Label/offset*
- Machine Language: **0100 0101 0000 0000 0000 0000 0000 0101**

Floating Point Immediate Format Instructions

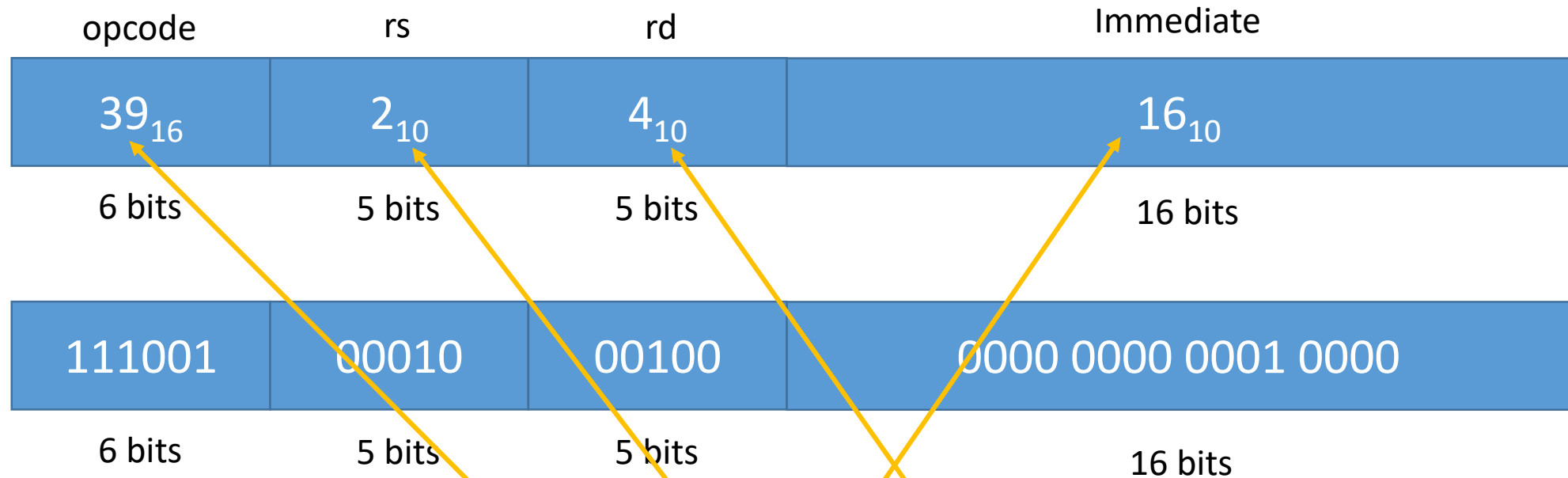
- Memory reference instructions for floating points use I Format instructions, *not* FI Format.
 - **swc1** and **lwc1**
- The machine language will be practically identical to the **lw** and **sw** instructions

Floating Point Immediate Format Instructions



- Assembly Language: **lwc1 \$f2, 8(\$a0)**
- Machine Language: **1100 0100 0100 0100 0000 0000 0000 1000**

Floating Point Immediate Format Instructions



- Assembly Language: **swc1 \$f2, 16(\$a0)**
- Machine Language: **1110 0100 0100 0100 0000 0000 0001 0000**

Addressing Modes

- An **addressing mode** specifies the way in which data is used via memory addresses.
 - An ISA may allow the use of multiple addressing modes in the design of its instructions.
 - **Direct addressing** is when an instruction uses the absolute address of an operand.
 - Unconditional jumps in MIPS use direct addressing (though the 26 bits are padded with zeroes to form a 32-bit address)
- j **label**

Addressing Modes

- **Register addressing** (a form of direct addressing) uses the data stored in registers as operands; no memory addresses needed
 - Instructions like **add** use register addressing
add \$t0, \$t1, \$t2
- **Immediate addressing** (also a form of direct addressing) is when an operand is part of the instruction
 - Immediate instructions in MIPS, like **addi**, use immediate addressing
addi \$t0, \$t1, 6

Addressing Modes

- **Indirect addressing** is when an instruction accesses an address of memory, which contains the address of the instruction's operand
- **Base addressing** (a form of indirect addressing) is when we add a constant (offset) to an address stored in a register
 - Instructions like **lw** use base addressing
`lw $t0, 16($a0)`
- **PC-relative addressing** (also a form of indirect addressing) is when an address is formed by adding the program counter with a constant
 - Branch instructions in MIPS, like **bne**, use PC-relative addressing
`bne $t0, $t1, label/offset`

Addressing Modes

- An instruction set with instructions that use any supported addressing mode is called an **orthogonal** instruction set.
 - For example, an **add** instruction for direct addressing, register addressing, base addressing, etc.
 - Adds much more complexity to the design of the processor
- **Complex Instruction Set Computers (CISC, “sisk”)**, are processors with a large number of instructions that are more like the instructions of a high-level programming language.
 - The main goal of a CISC architecture is to limit the number of lines of assembly code.
 - Goes hand-in-hand with an orthogonal instruction set

Complex Instruction Set Computers

- By reducing the number of lines of assembly code this achieves faster compile time of a high-level language to machine language
- This makes it easier to write a compiler for the processor, thus easier to support more high-level languages
 - It also requires less RAM for storing instructions

Complex Instruction Set Computers

- However, all of the complex instructions must be built into the hardware
 - Most instructions are executed in multiple CPU clock cycles
 - We'll discuss the CPU clock and clock cycles in a later lecture
- Instructions primarily operate directly on memory instead of using registers
 - Register access is faster than main memory access
- CISC architecture and orthogonal instruction sets were superseded by Reduced Instruction Set Computers.
 - CISC got its name retroactively
 - Intel's x86 architecture is basically the only CISC architecture still used in modern processors

Reduced Instruction Set Computers

- **Reduced Instruction Set Computers (RISC, “*risk*”)** are processors designed with a limited, yet optimized instruction set.
 - As opposed to the broad and specialized instructions found in a CISC architecture
- Processors with a RISC architecture have:
 - Simple, fixed length instructions that are executed in one CPU clock cycle
 - Few, simple data types
 - Simple addressing modes
 - Large number of general purpose registers

Reduced Instruction Set Computers

- MIPS is a RISC architecture, as is ARM (Advanced RISC Machine)
- RISC also has the benefit of *pipelining*, which processes instructions more efficiently by executing instructions simultaneously
 - We'll discuss pipelining in a later lecture
 - Pipelining gives RISC the performance advantage over CISC

RISC-V

- RISC-V (“*risk five*”) is an open standard ISA based on the philosophies of RISC architecture
 - Created by the University of California, Berkeley
- RISC-V is a free and open-source ISA
 - Most ISAs (like ARM and Intel’s x86 and x64) are patented and copyrighted
 - MIPS was proprietary but is now open source
 - Chips designers must pay royalties to use proprietary ISAs in their chip designs

RISC-V

- See the RISC-V Reference Card posted in Canvas as a reference for this lecture

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I					RV Privileged Instructions		
Category	Name	Fmt	RV32I Base	+RV{64,128}	Category	Name	RV mnemonic
Loads	Load Byte	I	LB rd,rs1,imm		CSR Access	Atomic R/W	CSRRW rd,csr,rs1
	Load Halfword	I	LH rd,rs1,imm			Atomic Read & Set Bit	CSRRS rd,csr,rs1
	Load Word	I	LW rd,rs1,imm	L{D Q} rd,rs1,imm		Atomic Read & Clear Bit	CSRRC rd,csr,rs1
	Load Byte Unsigned	I	LBU rd,rs1,imm			Atomic R/W Imm	CSRRWI rd,csr,imm
	Load Half Unsigned	I	LHU rd,rs1,imm	L{W D}U rd,rs1,imm		Atomic Read & Set Bit Imm	CSRRSI rd,csr,imm
Stores	Store Byte	S	SB rs1,rs2,imm		Atomic Read & Clear Bit Imm		
	Store Halfword	S	SH rs1,rs2,imm		CSRRCI rd,csr,imm		
	Store Word	S	SW rs1,rs2,imm	S{D Q} rs1,rs2,imm	Change Level Env. Call		
Shifts	Shift Left	R	SLL rd,rs1,rs2	SLL{W D} rd,rs1,rs2	Environment Breakpoint		
	Shift Left Immediate	I	SLLI rd,rs1,shamt	SLLI{W D} rd,rs1,shamt	Environment Return		
	Shift Right	R	SRL rd,rs1,rs2	SRL{W D} rd,rs1,rs2	Trap Redirect to Supervisor		
	Shift Right Immediate	I	SRLI rd,rs1,shamt	SRLI{W D} rd,rs1,shamt	Redirect Trap to Hypervisor		
	Shift Right Arithmetic	R	SRA rd,rs1,rs2	SRA{W D} rd,rs1,rs2	Hypervisor Trap to Supervisor		
	Shift Right Arith Imm	I	SRAI rd,rs1,shamt	SRAI{W D} rd,rs1,shamt	Interrupt Wait for Interrupt		
Arithmetic	ADD	R	ADD rd,rs1,rs2	ADD{W D} rd,rs1,rs2	MMU Supervisor FENCE		
	ADD Immediate	I	ADDI rd,rs1,imm	ADDI{W D} rd,rs1,imm	SFENCE.VM rs1		

RISC-V

- Many conventions you've seen in MIPS are also found in RISC-V
 - You'll notice the similarities between the registers in RISC-V and MIPS

<i>RISC-V Calling Convention</i>			
Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

REGISTER NAME, NUMBER, USE, CALL CONVENTION			
NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
\$zero	0	The Constant Value 0	N.A.
\$at	1	Assembler Temporary	No
\$v0-\$v1	2-3	Values for Function Results and Expression Evaluation	No
\$a0-\$a3	4-7	Arguments	No
\$t0-\$t7	8-15	Temporaries	No
\$s0-\$s7	16-23	Saved Temporaries	Yes
\$t8-\$t9	24-25	Temporaries	No
\$k0-\$k1	26-27	Reserved for OS Kernel	No
\$gp	28	Global Pointer	Yes
\$sp	29	Stack Pointer	Yes
\$fp	30	Frame Pointer	Yes
\$ra	31	Return Address	Yes

RISC-V

- You'll recognize a number of MIPS instructions that also appear in RISC-V

Category	Name	Fmt		RV32I Base
Loads	Load Byte	I	LB	rd,rs1,imm
	Load Halfword	I	LH	rd,rs1,imm
	Load Word	I	LW	rd,rs1,imm
	Load Byte Unsigned	I	LBU	rd,rs1,imm
	Load Half Unsigned	I	LHU	rd,rs1,imm
Stores	Store Byte	S	SB	rs1,rs2,imm
	Store Halfword	S	SH	rs1,rs2,imm
	Store Word	S	SW	rs1,rs2,imm
Shifts	Shift Left	R	SLL	rd,rs1,rs2
	Shift Left Immediate	I	SLLI	rd,rs1,shamt
	Shift Right	R	SRL	rd,rs1,rs2
	Shift Right Immediate	I	SRLI	rd,rs1,shamt
	Shift Right Arithmetic	R	SRA	rd,rs1,rs2
	Shift Right Arith Imm	I	SRAI	rd,rs1,shamt
Arithmetic	ADD	R	ADD	rd,rs1,rs2
	ADD Immediate	I	ADDI	rd,rs1,imm
	SUBtract	R	SUB	rd,rs1,rs2
	Load Upper Imm	U	LUI	rd,imm
	Add Upper Imm to PC	U	AUIPC	rd,imm
Logical	XOR	R	XOR	rd,rs1,rs2
	XOR Immediate	I	XORI	rd,rs1,imm
	OR	R	OR	rd,rs1,rs2
	OR Immediate	I	ORI	rd,rs1,imm
	AND	R	AND	rd,rs1,rs2
	AND Immediate	I	ANDI	rd,rs1,imm
Compare	Set <	R	SLT	rd,rs1,rs2
	Set < Immediate	I	SLTI	rd,rs1,imm
	Set < Unsigned	R	SLTU	rd,rs1,rs2
	Set < Imm Unsigned	I	SLTIU	rd,rs1,imm
Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm

RISC-V

- RISC-V has more instruction formats than MIPS

32-bit Instruction Formats

Register Format	→	R		31	30	25	24	21	20	19	15	14	12	11	8	7	6	0				
Immediate Format	→	I		funct7					rs2					rs1		funct3		rd		opcode		
Storage Format	→	S		imm[11:0]											rs1		funct3		rd		opcode	
Branch Format	→	SB		imm[11:5]					rs2					rs1		funct3		imm[4:0]		opcode		
Upper-Immediate Format	→	U		imm[12]	imm[10:5]				rs2					rs1		funct3		imm[4:1]	imm[11]	opcode		
Jump Format	→	UJ		imm[31:12]																rd		opcode
				imm[20]	imm[10:1]				imm[11]		imm[19:12]									rd		opcode

RISC-V


- Boston University's online RISC-V simulator (BRISC-V) is an easy way to practice with the basics of RISC-V
 - A link to the simulator is posted in Canvas
- You can also upload C source code and it will convert the program to RISC-V assembly.
 - This can give you a good understanding of the compile process and see how high-level instructions become low-level instructions.

RISC-V

BRISC-V Home



BRISC-V Simulator

Manual & Examples








BRISC-V Simulator
Adaptive and Secure Computing Systems Lab • Boston University

C source



```
1 int fib(int n) {
2     if (n <= 1) {
3         return n;
4     } else {
5         return fib(n-1)+fib(n-2);
6     }
7 }
8
9 int return_function (int result) {
10     return result;
11 }
12
13 int main(){
14     int n = 9;
15     int result = return_function (fib(n));
16     return result;
17 }
```

RISC-V Assembly



```
0      addi    zero,zero,0
kernel:
1      addi    sp,zero,1536
2      call    main
3      addi    zero,zero,0
4      mv      s1,a0
5      addi    zero,zero,0
6      addi    zero,zero,0
7      auipc   ra,0x0
8      jalr    ra,0(ra)
9      addi    zero,zero,0
10     addi    zero,zero,0

.file "source.c"
.option nopic
.text
.align 2
.globl fib
.type fib, @function

fib:
11     addi    sp,sp,-32
12     sw      ra,28(sp)
13     sw      s0,24(sp)
14     sw      s1,20(sp)
15     addi    s0,sp,32
16     sw      a0,-20(s0)
17     lw      a4,-20(s0)
18     li      a5,1
19     bgt     a4,a5,.L2
20     lw      a5,-20(s0)
```

Registers

Memory

Register		Value	Register		Value
zero	[0]	0	ra	[1]	0
sp	[2]	0	gp	[3]	0
tp	[4]	0	t0	[5]	0
t1	[6]	0	t2	[7]	0
s0/fp	[8]	0	s1	[9]	0
a0	[10]	0	a1	[11]	0
a2	[12]	0	a3	[13]	0
a4	[14]	0	a5	[15]	0
a6	[16]	0	a7	[17]	0
s2	[18]	0	s3	[19]	0
s4	[20]	0	s5	[21]	0
s6	[22]	0	s7	[23]	0
s8	[24]	0	s9	[25]	0
s10	[26]	0	s11	[27]	0
t3	[28]	0	t4	[29]	0
t5	[30]	0	t6	[31]	0

Console

Successfully compiled.
***** Parser Output *****
Parsing successful!

Instruction breakdown

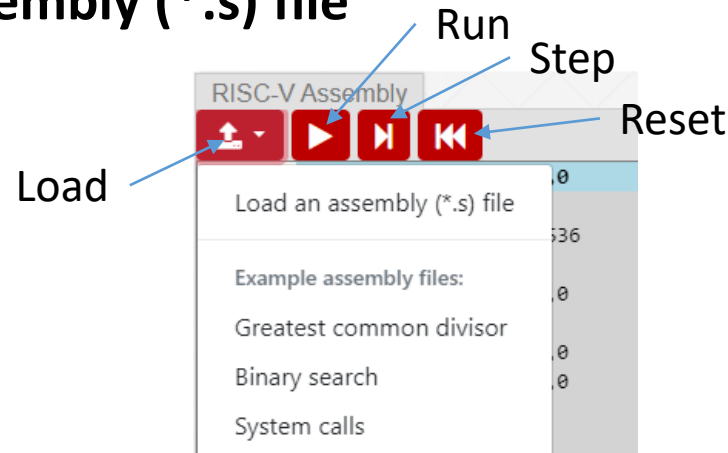
31	20	19	15	14	12	11	7	6	0
imm		rs1		funct3		rd		opcode	
000000000000		00000		000		00000		0010011	

RISC-V

- The Module Download contains sample RISC-V assembly code.
 - RISC-V source code files end with the .s extension
- You are encouraged to use these sample programs for exploring RISC-V further.

RISC-V

- To load the RISC-V assembly code, click the Upload button.
 - Select **Load an assembly (*.s) file**



- The Run button runs the program
- The Step button executes the program instruction by instruction
- The Reset button resets the program

RISC-V

- You cannot edit the uploaded assembly file.
 - Make any changes to the assembly file in a text editor, save the changes, and re-upload the modified assembly file.
- The simulator is limited in its capabilities.
 - Some common RISC-V pseudo-ops are not supported
 - Floating point numbers are not supported