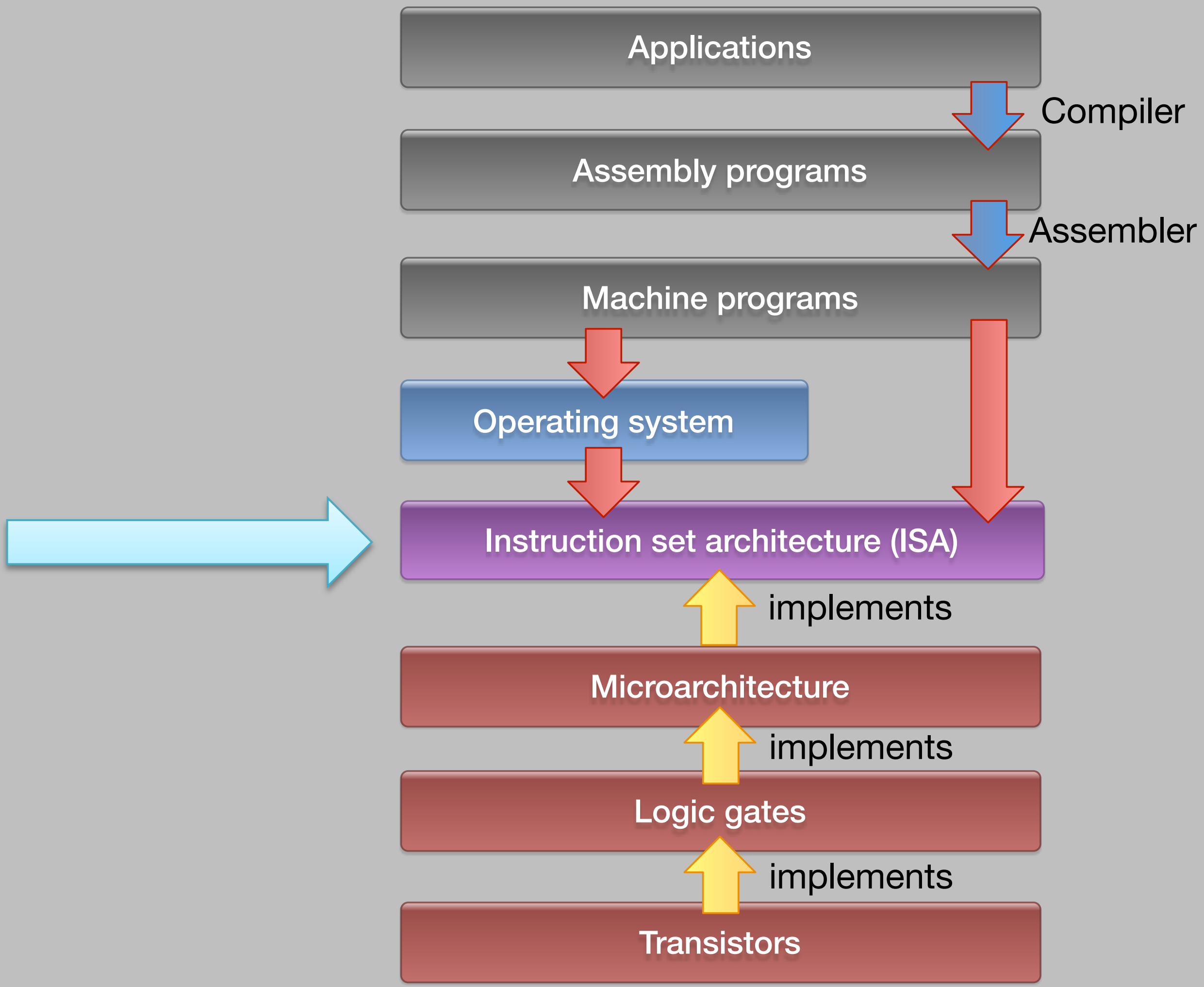
Instruction Set Architecture at the Example of MIPS

Becker/Molitor, Chapter 10 treats a similar but not identical system.
We follow the American book "Digital design and computer architecture" by Harris and Harris, 2013.
Here, Chapters 6 and 7 are particularly relevant.

Jan Reineke
Universität des Saarlandes

Abstraction layers in computer systems



Overview: Architecture vs Microarchitecture

- **Instruction set architecture** (also simply: architecture)
  = set of instructions, their **encoding** and **semantics**
  = „**What**" a computer computes
  For example: x86, ARM

- **Microarchitecture**
  = concrete implementation of an instruction set in hardware
  = „**How**" a computer works
  For example: Intel Skylake, AMD Zen 3 (both x86), Apple M1 (ARM)

Overview: Architecture vs Microarchitecture

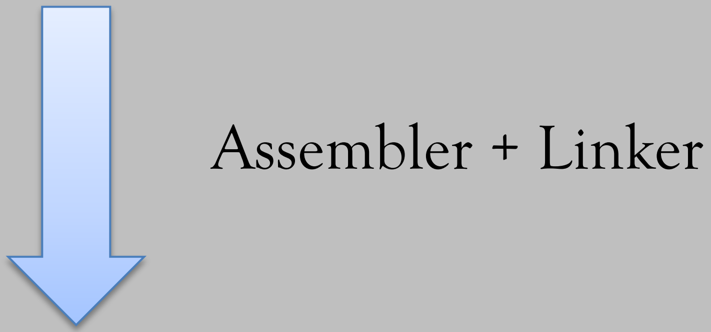
An **instruction set architecture** can be implemented by many different **microarchitectures**:

→ e.g. AMD and Intel processors implement x86 instruction sets

→ new microarchitectures do not require new compiler, nor a new operating system

Assembly and machine language

Assembly language = textual representation of instructions



Machine language = binary representation executed by a computer

MIPS instruction set

In the following:
A brief overview of the MIPS instruction set.

Logical state of MIPS instruction set

State is completely determined by:

- **Program counter (PC)**
- **Register file** consists of 32 registers, **(Reg)**
- **Memory (Mem)**

Logical execution of a machine program

```
PC := 0
while (true) {
   instruction = Mem[PC]
   (PC, Reg, Mem) :=
      execute(instruction, PC, Reg, Mem)
}
```

Three types of instructions:

1. Arithmetic and logic instructions
2. Memory instructions
3. Jump and branch instructions

Arithmetic and logic instructions (immediate)

Arithmetic instructions perform arithmetic and logic operations on registers

Immediate instructions employ **immediate addressing**.

Examples:

Comment

```
slti $t1, $s2, 100 # if ($s2 < 100) then $t1 = 1
                                 else $t1 = 0

addi $t1, $s2, 100 # $t1 = $s2 + 100
```

„immediate“

Register $t1 contains the value of $s2+100 in two‘s complement.

Arithmetic and logic instructions (register)

Arithmetic instructions perform arithmetic and logic operations on registers

Register instructions employ **register addressing**. All operands are register contents.

Examples:

```
add $s1, $s2, $s3   # $s1 = $s2 + $s3
slt $s1, $s2, $s3   # if ($s2 < $s3) then $s1 = 1
                     else $s1 = 0
and $s1, $s2, $s3   # $s1 = $s2 & $s3
```

Bitwise logical AND

MIPS registers

Name	Register number	Usage
`$0`	0	the constant value 0
`$at`	1	assembler temporary
`$v0-$v1`	2-3	Function return values
`$a0-$a3`	4-7	Function arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved variables
`$t8-$t9`	24-25	more temporaries
`$k0-$k1`	26-27	OS temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	Function return address

Memory instructions

Memory instructions move data between memory and registers. They employ **base addressing**.

Examples:

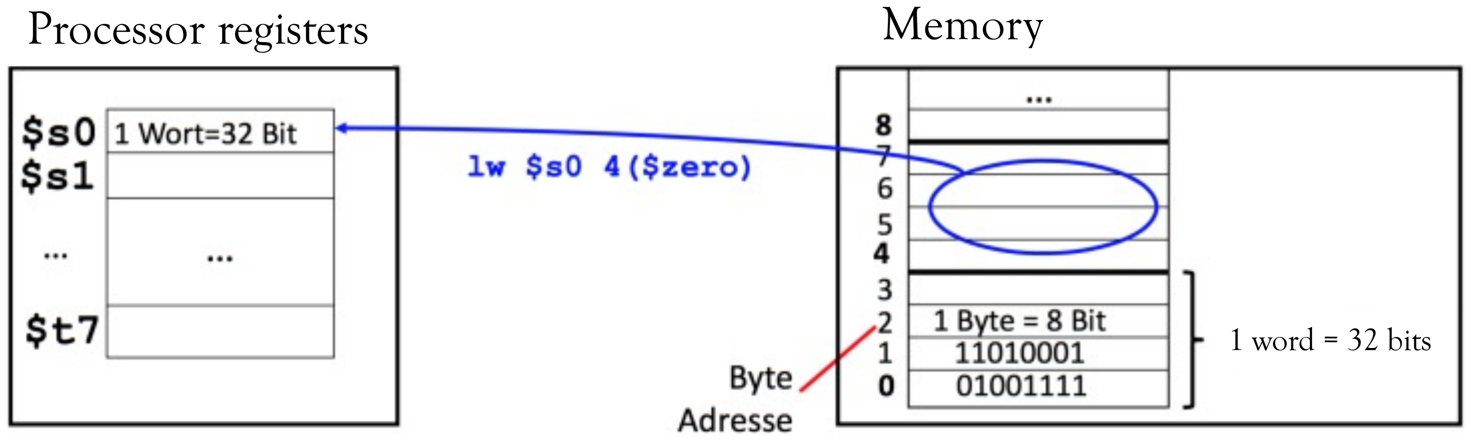
```
lw $t0, 100($s2)   # $t0 = Mem[100+$s2]
sw $t0, 100($s2)   # Mem[100+$s2] = $t0
```

Memory organization

Memory is **byte-addressable**, i.e., each byte can be addressed separately

Registers store **words**:

- 1 byte = 8 bit
- 1 word = 4 byte = 32 bit



Word access must be correctly aligned.

Jump and branch instructions

… alter the control flow of a program.

Two types:

- **Jump instructions**
- **Branch instructions**

Jump instructions

Jumps modify the program counter *in any case*.

Jump instructions employ either

- **pseudodirect addressing** (`j, jal`) or
- **register addressing** (`jr, jalr`).

Jump instructions

Examples:

```
jr $ra          # PC = $ra (next instruction
                          is at Mem[$ra])
j Label1        # go to Label1
jal Label2      # $ra = PC+4 ; go to Label2
```

LABEL1: add ...

The instructions modify the program counter.
As instructions in MIPS always occupy *an entire word*,
but memory is byte-addressed,
the two least significant bits of the register with the jump address
always have to be 0.

Branch instructions

Branch instructions modify the program counter *only if a condition is satisfied.*

Branch instructions (`beq`, `bne`) employ **PC-relative addressing**.

Branch instructions

Examples:

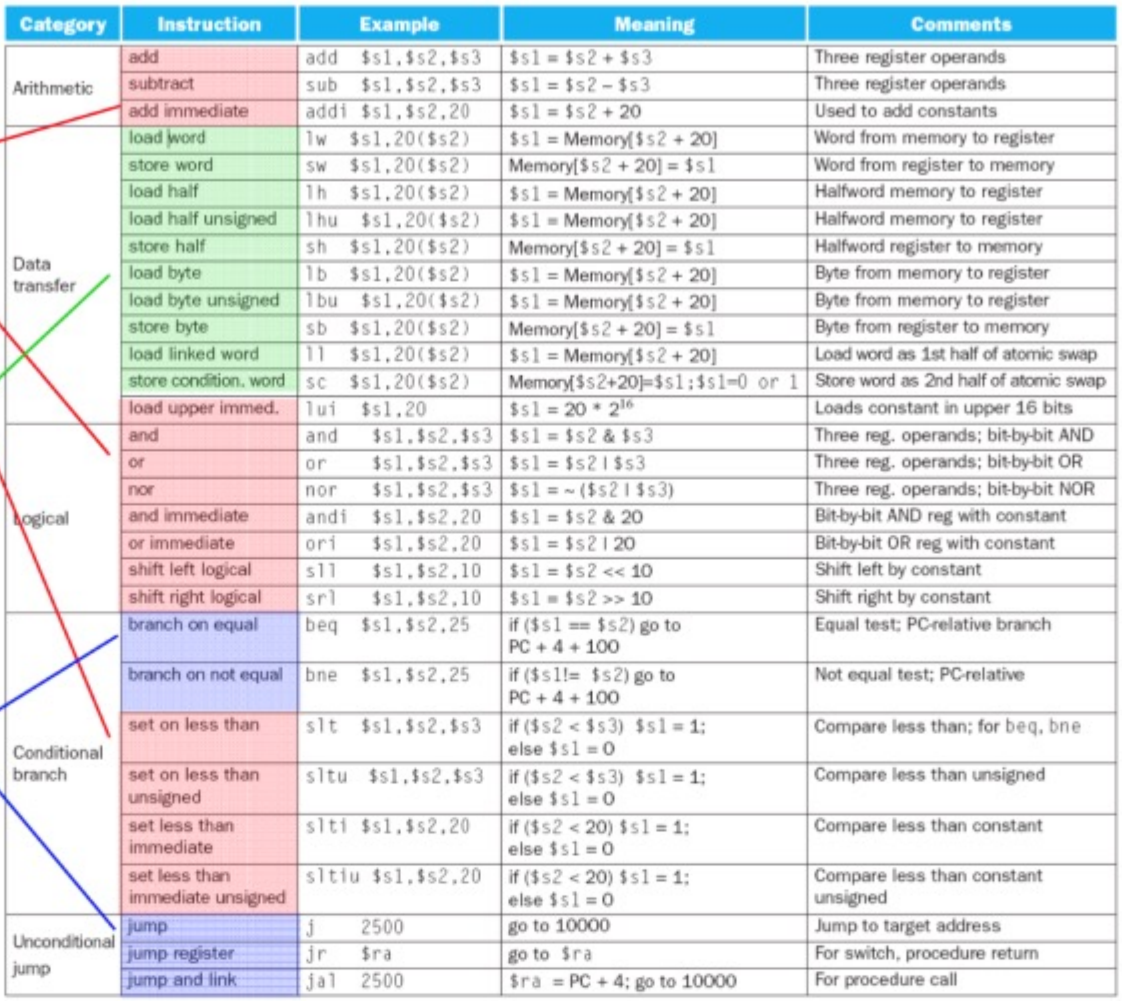
```
beq $s1, $s2, Label3 # if ($s1=$s2) then go to Label3
bne $s1, $s2, Label4 # if ($s1!=$s2) then go to Label4
```

Summary: MIPS assembler

Arithmetic and logic instructions

Memory instructions

Jump and branch instructions

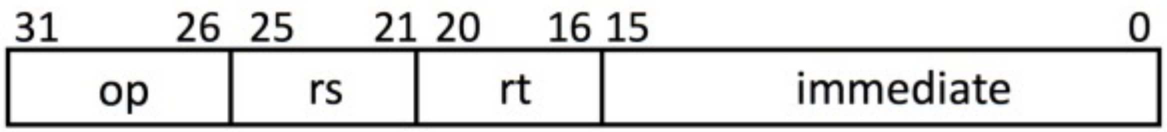


Category	Instruction	Example	Meaning	Comments
Arithmetic	add	add $s1,$s2,$s3	$s1 = $s2 + $s3	Three register operands
	subtract	sub $s1,$s2,$s3	$s1 = $s2 – $s3	Three register operands
	add immediate	addi $s1,$s2,20	$s1 = $s2 + 20	Used to add constants
Data transfer	load word	lw $s1,20($s2)	$s1 = Memory[$s2 + 20]	Word from memory to register
	store word	sw $s1,20($s2)	Memory[$s2 + 20] = $s1	Word from register to memory
	load half	lh $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	load half unsigned	lhu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Halfword memory to register
	store half	sh $s1,20($s2)	Memory[$s2 + 20] = $s1	Halfword register to memory
	load byte	lb $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	load byte unsigned	lbu $s1,20($s2)	$s1 = Memory[$s2 + 20]	Byte from memory to register
	store byte	sb $s1,20($s2)	Memory[$s2 + 20] = $s1	Byte from register to memory
	load linked word	ll $s1,20($s2)	$s1 = Memory[$s2 + 20]	Load word as 1st half of atomic swap
	store condition. word	sc $s1,20($s2)	Memory[$s2+20]=$s1;$s1=0 or 1	Store word as 2nd half of atomic swap
	load upper immed.	lui $s1,20	$s1 = 20 * 2^16	Loads constant in upper 16 bits
Logical	and	and $s1,$s2,$s3	$s1 = $s2 & $s3	Three reg. operands; bit-by-bit AND
	or	or $s1,$s2,$s3	$s1 = $s2 | $s3	Three reg. operands; bit-by-bit OR
	nor	nor $s1,$s2,$s3	$s1 = ~ ($s2 | $s3)	Three reg. operands; bit-by-bit NOR
	and immediate	andi $s1,$s2,20	$s1 = $s2 & 20	Bit-by-bit AND reg with constant
	or immediate	ori $s1,$s2,20	$s1 = $s2 | 20	Bit-by-bit OR reg with constant
	shift left logical	sll $s1,$s2,10	$s1 = $s2 << 10	Shift left by constant
	shift right logical	srl $s1,$s2,10	$s1 = $s2 >> 10	Shift right by constant
Conditional branch	branch on equal	beq $s1,$s2,25	if ($s1 == $s2) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	bne $s1,$s2,25	if ($s1!= $s2) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	slt $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than; for beq, bne
	set on less than unsigned	sltu $s1,$s2,$s3	if ($s2 < $s3) $s1 = 1; else $s1 = 0	Compare less than unsigned
	set less than immediate	slti $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant
	set less than immediate unsigned	sltiu $s1,$s2,20	if ($s2 < 20) $s1 = 1; else $s1 = 0	Compare less than constant unsigned
Unconditional jump	jump	j 2500	go to 10000	Jump to target address
	jump register	jr $ra	go to $ra	For switch, procedure return
	jump and link	jal 2500	$ra = PC + 4; go to 10000	For procedure call

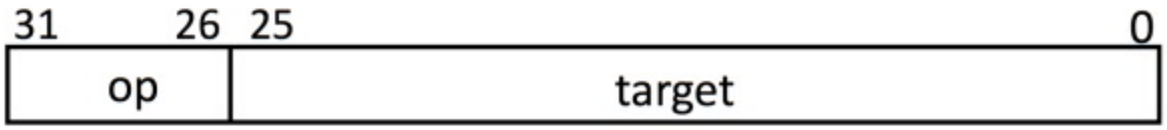
Instruction encoding

- **Instruction encoding** refers to the encoding of instructions in machine words.
- MIPS uses a fixed-length 32-bit encoding of all instructions (this is unlike, e.g. x86)
- We distinguish three types I, J and R:

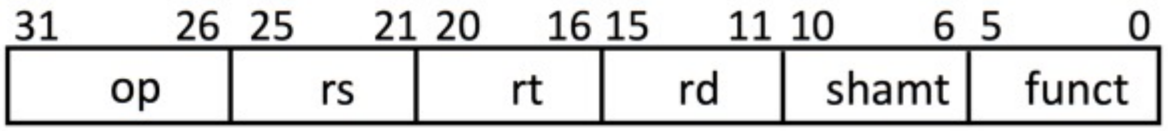
"immediate" **Type I**



"jump" **Type J**



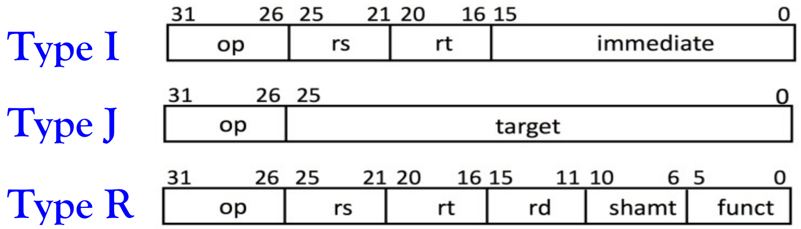
"register" **Type R**



Instruction encoding

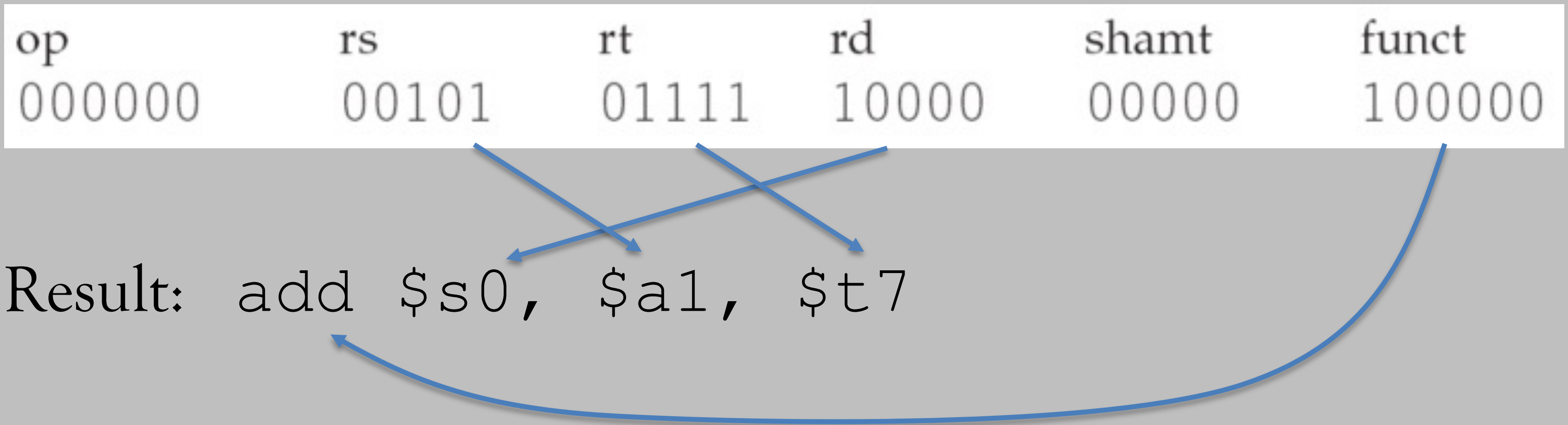
Abbreviation	Meaning
I	immediate
J	jump
R	register
op	6-bit encoding of the operation
rs	5-bit encoding of a source register
rt	5-bit encoding of a source or target register
immediate	16-bit immediate value
target	26-bit jump target
rd	5-bit encoding of the target register
shamt	5-bit encoding of “shift amount”
funct	6-bit encoding of the function

Example of instruction encoding



Converting `00af8020`$_{\texttt{hex}}$ into an assembly instruction:

- Binary represent.: 0000 0000 1010 1111 1000 0000 0010 0000
- Decoding the instruction type: Type R, because *op*=000000
- Division of the binary representation



op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

- Result: `add $s0, $a1, $t7`

Addressing modes

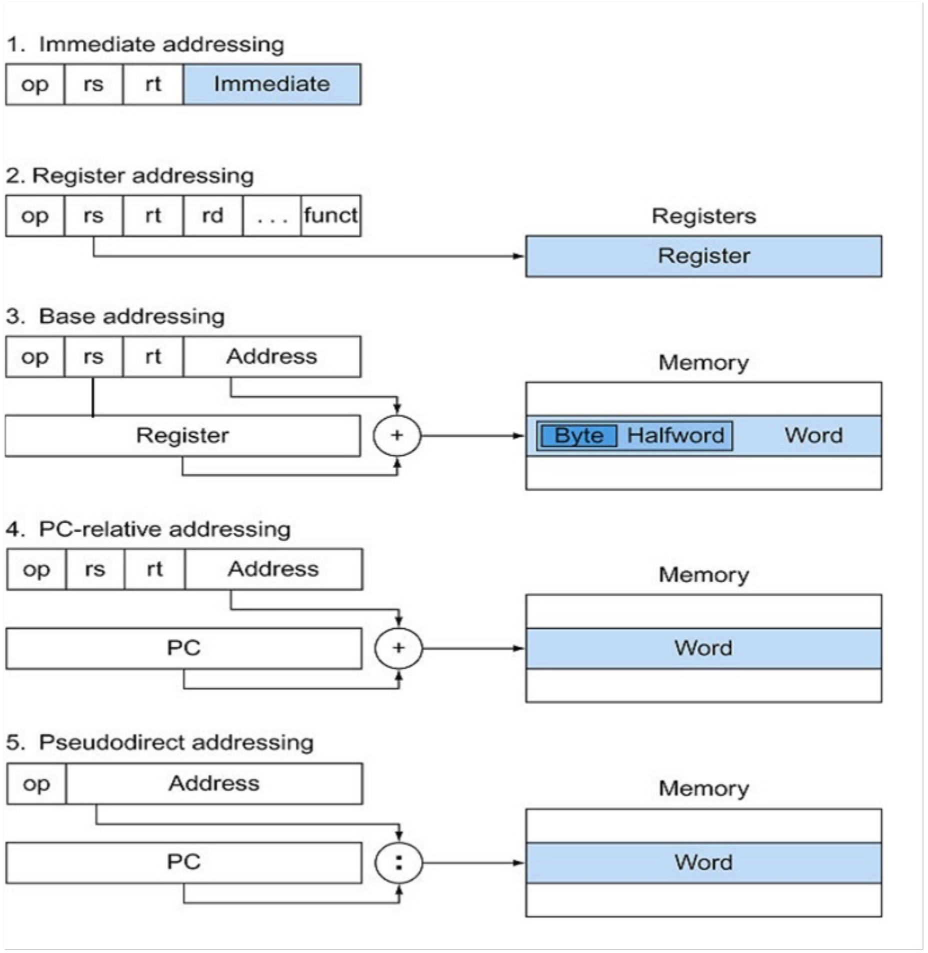
Addressing modes are used to determine required data in an instruction:

- as operands of, e.g., arithmetic operations,
- as jump or branch targets.

MIPS addressing modes

- **Immediate addressing:** The operand is a constant in the instruction.
- **Register addressing:** The operand is the value of a register.
- **Base addressing:** The operand is a memory value whose address is determined by the sum of a register value and a constant in the instruction.
- **PC-relative addressing:** The new program counter is the sum of the program counter (PC+4) and a constant in the instruction.
- **Pseudodirect addressing:** The new program counter is determined from a constant (26 bit) and the 4 most-significant bits of the old program counter (PC+4).

Addressing modes



1. In which **order** are words stored in the bytes of memory?
→ **"little endian"** vs **"big endian"**

2. Binary numbers are interpreted as ***unsigned*** (e.g. multu) or ***signed*** (e.g. mult).

3. Binary numbers of different bit widths are added up
→ need *sign extension*

Data formats

The following **data formats** are defined:

- Byte (8 bit)
- Half-word (16 bit)
- Word (32 bit)

MIPS allows to switch between little and big endian.

MIPS uses both "**little endian**" and "**big endian**".

Following "**big endian**" we have:

The **most significant** byte of a word is at its smallest address.

A word is addressed with the byte address of its **smallest address** (i.e. with its most significant byte).

Data formats

The following **data formats** are defined:
- Byte (8 bit)
- Half-word (16 bit)
- Word (32 bit)

x86 uses little-endian

MIPS uses both "**little endian**" and "**big endian**".

Following "**little endian**" we have:
The **least significant** byte of a word is at its smallest address.
A word is addressed with the byte address of its **smallest address** byte (i.e. with its least significant byte).

Little vs big endian in daily life

Data:

10. Juni 2021 → little endian

Time of day:

8:30 o'clock → big endian

Numbers:

235 (read from left to right) → big endian

235 (read from right to left, in Arabic) → little endian

zweihundertfünfunddreißig → mixed endian

2 … 5 … 3

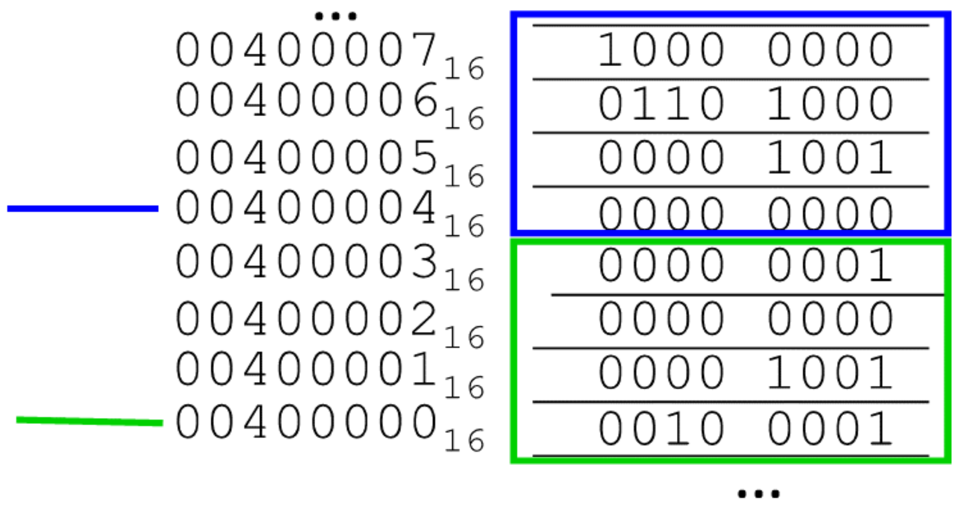
Data formats

Example “**big endian**”:

- Program:

```
addi $t1, $t0, 1          #    21090001hex
sll $t5, $t1, 2           #    00096880hex
```

- Program memory:



Address of the instruction
`sll $t5 $t1 2`

Address of the instruction
`addi $t1 $t0 1`

…	
00400007_{16}	1000 0000
00400006_{16}	0110 1000
00400005_{16}	0000 1001
00400004_{16}	0000 0000
00400003_{16}	0000 0001
00400002_{16}	0000 0000
00400001_{16}	0000 1001
00400000_{16}	0010 0001
	…

Data formats: Sign extension

Numbers in MIPS are either interpreted as **unsigned** or **signed** in two's complement:

- An n-bit *unsigned* binary number has the value:

$$B = \langle d_{n-1} \ldots d_0 \rangle = \sum_{i=0}^{n-1} d_i \cdot 2^i$$

- The extension into an *unsigned* binary is achieved by padding with zeroes:

$$\langle 0 \ldots 0 d_{n-1} \ldots d_0 \rangle = \sum_{i=0}^{m-1} d_i \cdot 2^i = \sum_{i=n}^{m-1} 0 \cdot 2^i + \sum_{i=0}^{n-1} d_i \cdot 2^i = B$$

Data formats: Sign extension

An *signed* n-bit binary number is represented in two's complement and has the following value:

$$B = [d_{n-1} \ldots d_0]_2 = -d_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} d_i \cdot 2^i$$

Brainstorming:
How can a signed n-bit binary number be extended into an m-bit binary number?

Data formats: Sign extension

An *signed* n-bit binary number is represented in two's complement and has the following value:

$$B = \left[d_{n-1} \ldots d_0\right]_2 = -d_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} d_i \cdot 2^i$$

The extension into an m-bit binary number is achieved by padding with the most-significant bit:

$$\left[d_{n-1} \ldots d_{n-1} d_{n-1} \ldots d_0\right]_2 = -d_{n-1} \cdot 2^{m-1} + \sum_{i=n-1}^{m-2} d_{n-1} \cdot 2^i + \sum_{i=0}^{n-2} d_i \cdot 2^i = B$$

Data formats

Example: register contents:

```
r1 = 0..0001, r2 = 0..0010, r3 = 1..1111
```

Assembler instructions:

```
slt r4,r2,r1    # if (r2<r1) then r4=1 else r4=0
slt r5,r3,r1    # if (r3<r1) then r5=1 else r5=0
sltu r6,r2,r1   # if (r2<r1) then r6=1 else r6=0
sltu r7,r3,r1   # if (r3<r1) then r7=1 else r7=0
```

"set less than"

"unsigned"

Quiz:
Which values do `r4`, `r5`, `r6`, and `r7` take?