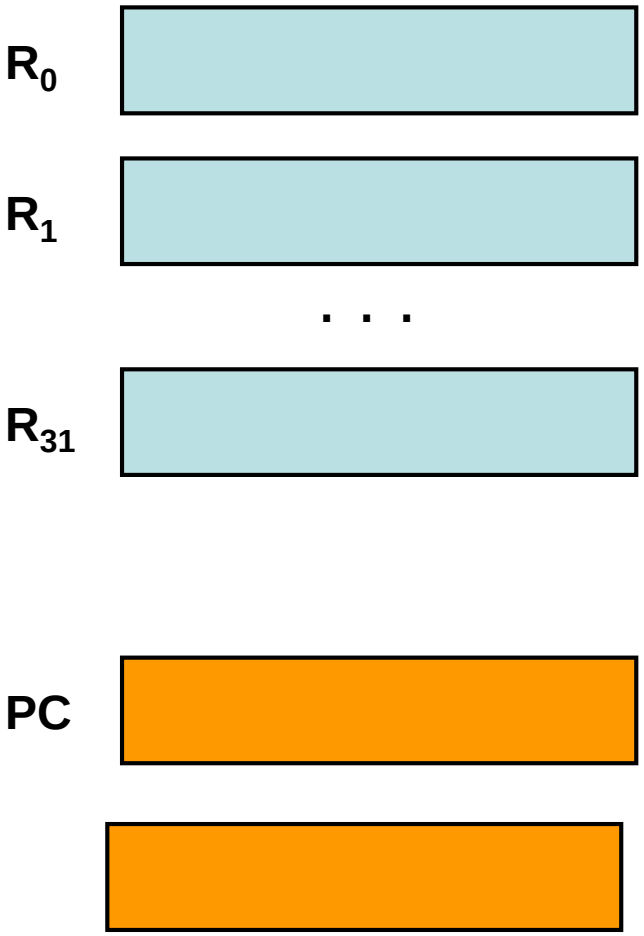
CPU Logical Structure

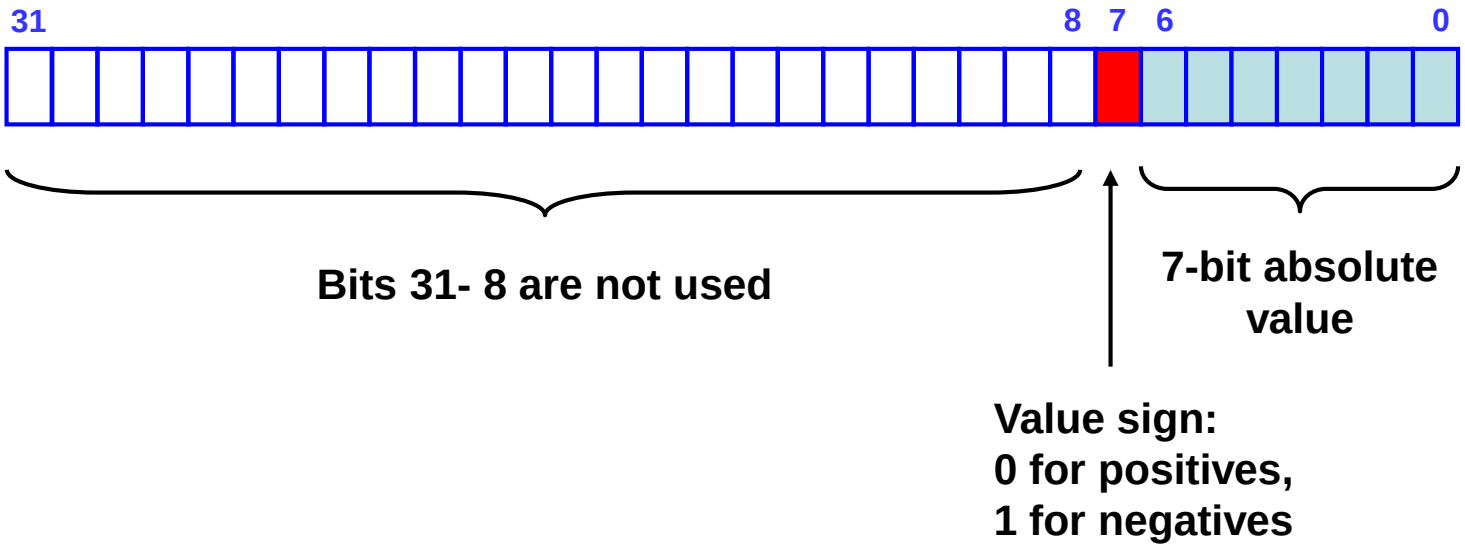


- **32 common registers** (R0…R31) each of which is of 32 bits. Every register can contain either a value or an address of a memory location.
  An address in a register can address any byte of memory.
- The CPU performs all actions on the operands taken from common registers. There are instructions for loading values from the memory to a register, and for storing a value from a register to the memory. Also there is a special instruction for loading short constants to registers.
- **PC register** (Program Counter) contains 32-bit address of the leftmost (high) byte of the instruction which is being currently executed. After the current instruction is completed, the address in PC is normally increased by 2 addressing the next instruction (because every instruction occupies 2 bytes, see later). The exception is CBR instruction which can alter this behavior setting the new address on the PC taking it from a common register. There are no other ways to modify the contents of the PC register.

Supported Values: 8-bit integers

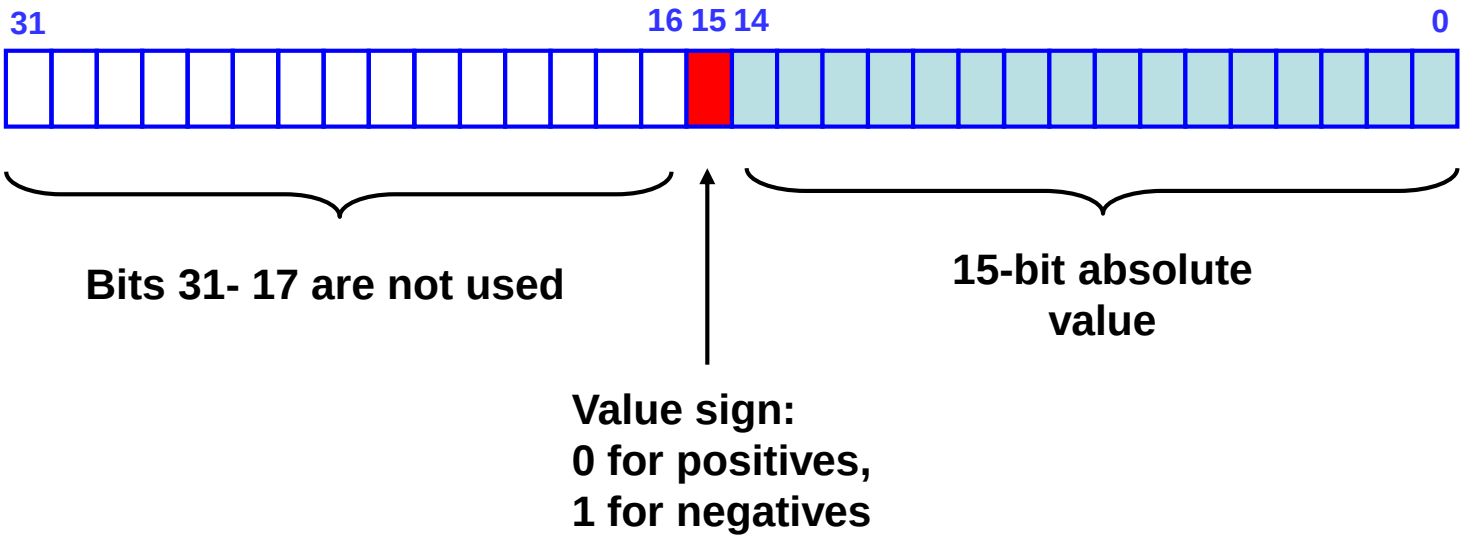
- Alignment?

- The CPU operates on 8-, 16-, and 32-bit values.
- The values of all formats are considered either as **signed integers** (arithmetic ADD and SUB instructions, and arithmetic shift ASL, ASR instructions) or as **bit scales** (logical shift LSL, LSR instructions and logical AND, OR, and XOR instructions).
- Positive integer values are represented **in the direct code** (with 0 in the sign bit). Negative integers are represented **in the two's complement code**.
  See ISO-XXXX for details.
- The range of possible 8-bit integer values is [-128..127].
- The format of 8-bit signed integers is shown below:



Supported Values: 16-bit integers

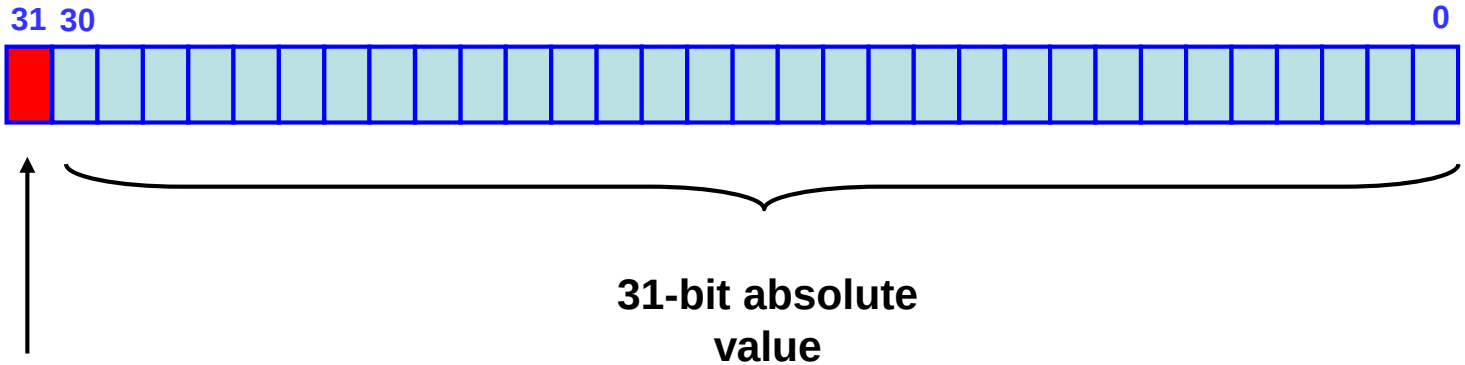
- The range of possible 16-bit integer values is [-32768..32767].
- The format of 16-bit signed integers is shown below:



- Alignment?

Supported Values: 32-bit integers

- The range of possible 32-bit integer values is [-2147483648..2147483647]
- The format of 32-bit signed integers is shown below:

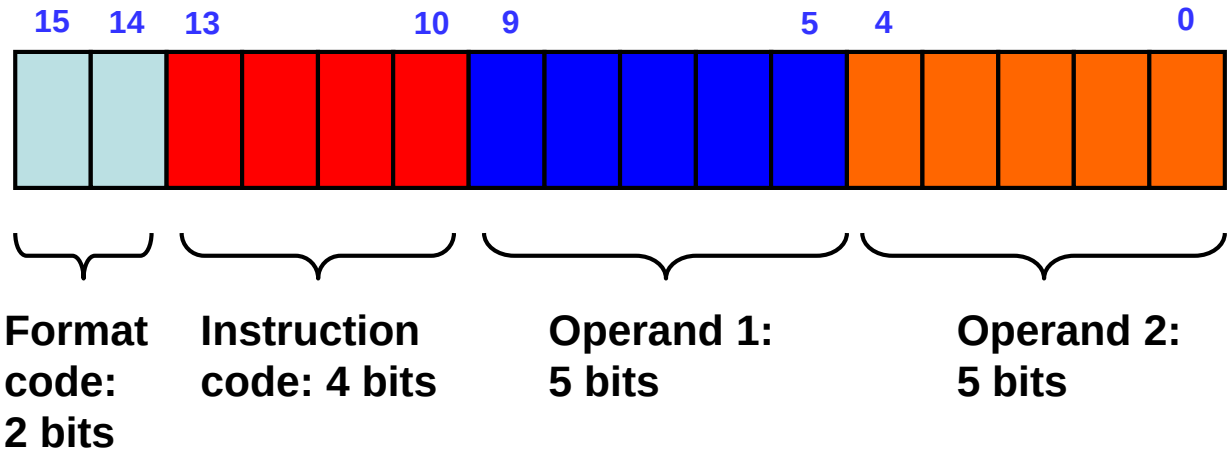


- Alignment?

Non-supported Values

- Long (64-bit) integer and floating point values are not directly supported by the CPU. If necessary, the support can be provided programmatically, by a (standard) library.
- The floating-point types should be conceptually associated with the 32-bit single-precision and 64-bit double-precision IEEE 754 values and operations as specified in *IEEE Standard for Binary Floating-Point Arithmetic*, ANSI/IEEE Standard 754-1985 (IEEE, New York).

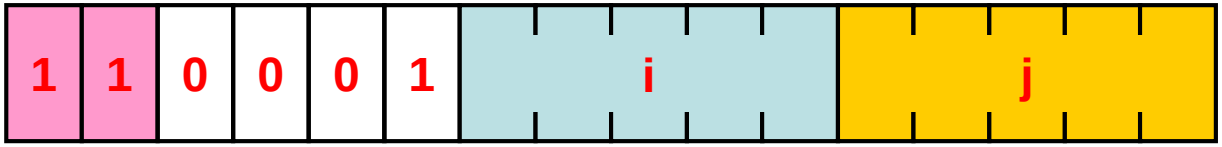
Common Instruction Format



- Every instruction occupies 16 bits (two bytes).
- **Format code** codes format of operands:
  00 is for 8 bit (lowest 8 bits of operands participate in the operation),
  01 is for 16 bit (lowest 16 bits of operands participate in the operation),
  10 is reserved,
  11 is for 32 bits (entire 32 bits of operands participate in the operation).
- **Instruction code** codes the operation kind of the instruction. There are 16 main kinds of instructions coded by 0x0, 0x1, … 0xF.
- **Operands** always (except the first operand of the LDC instruction) contain register codes (numbers within the range of 0..31).
- Alignment?

LD i j

- **The LD instruction copies the value of a 32-bit memory word pointed to by Ri register, to the Rj register**.
- The instruction format is as follows:

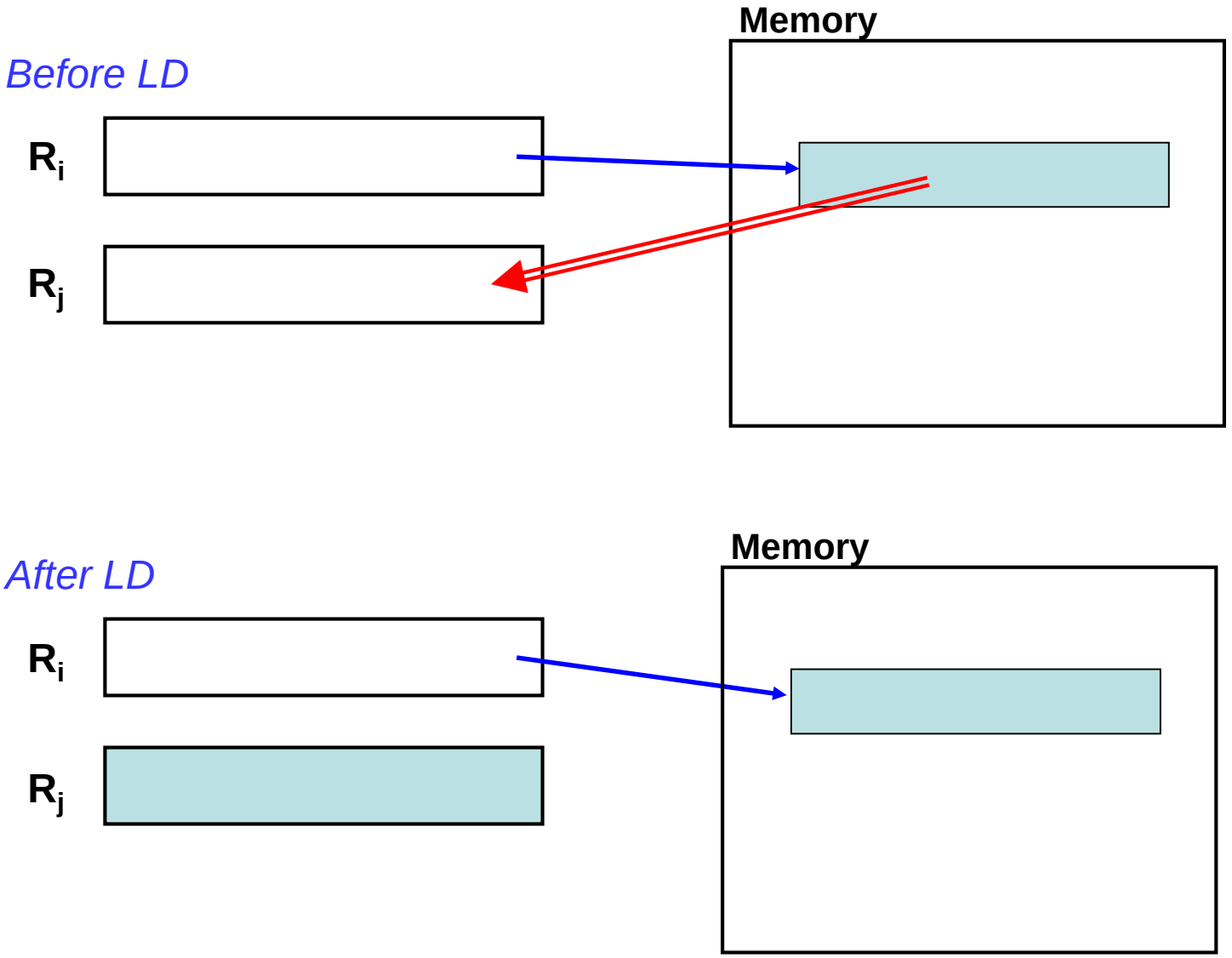


- The contents of the register Ri is considered as a 32-bit address of a 32-bit memory word.
- Instruction format is always **32**, i.e., the entire 32-bit word is copied from the memory to the register.
- When the instruction is completed, the original contents of the register Rj is lost. The contents of the Ri register (i.e., the address) does not change.

Suggested assembly statement for the LD instruction:

```
Rj := *Ri;
```

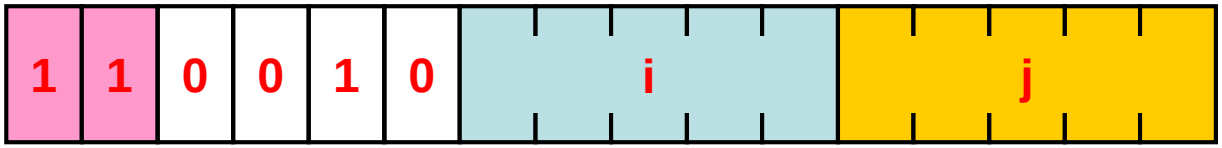
The effect of the LD instruction is shown below



LDA i j

The LDA instruction takes the value from the next 32-bit word, then adds it with the current value of the Ri register, and stores the result to the Ri register.

- The instruction format is as follows:



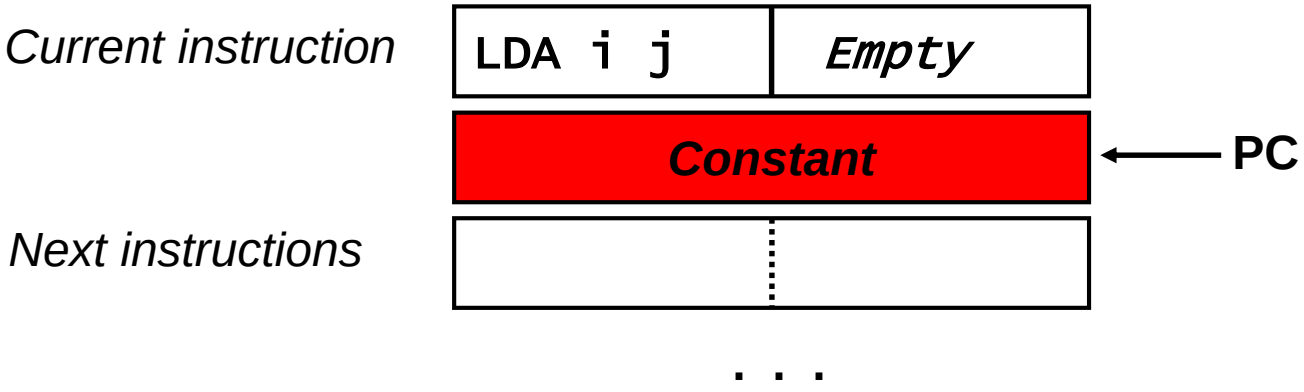
- The contents of the register Ri is considered as a 32-bit address of a 32-bit memory word.
- The next 32-bit word (pointed to by the PC register) is considered as an offset relatively to the “base” from the Ri register.
- Instruction format is always **32**, i.e., the entire 32-bit word is copied from the memory to the register.
- When the instruction is completed, the original contents of the register Rj is lost. The contents of the Ri register (i.e., the address) does not change.

Suggested assembly statement for the LD instruction:

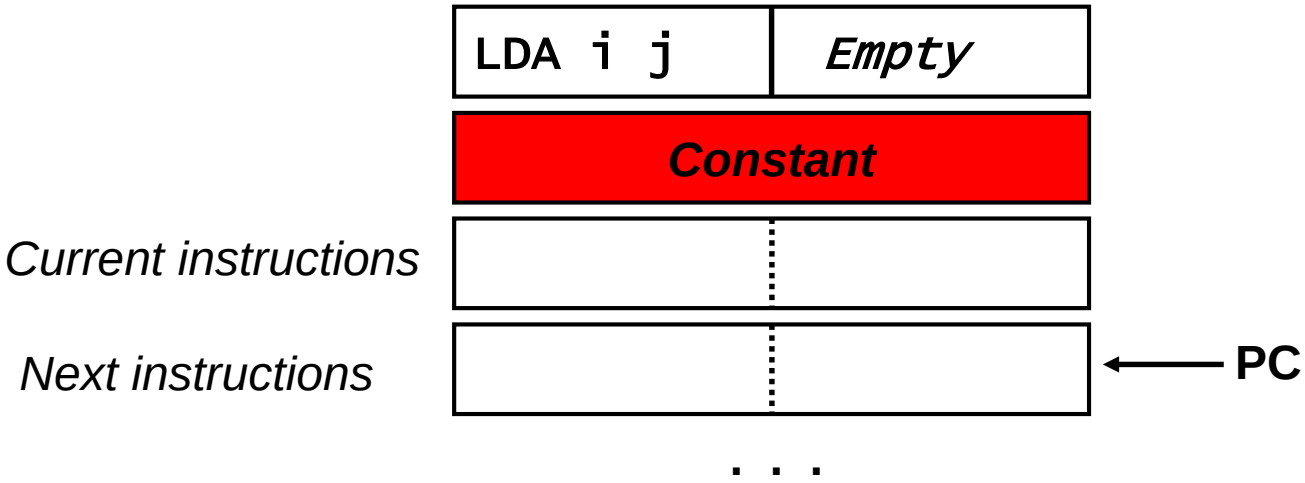
```
Rj := Ri + constant;
```

The scheme of how code is being processed is shown below

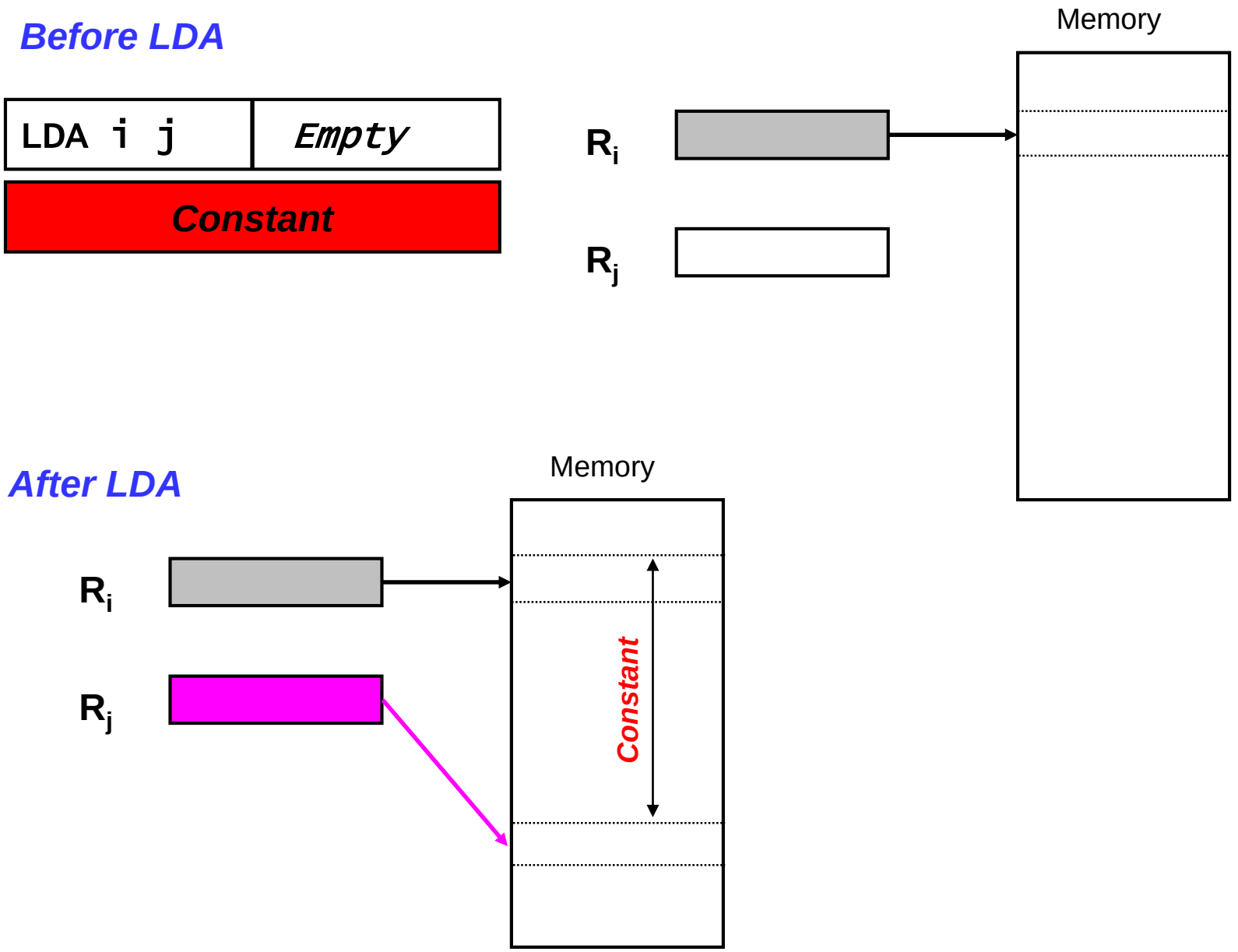
Before LDA



After LDA

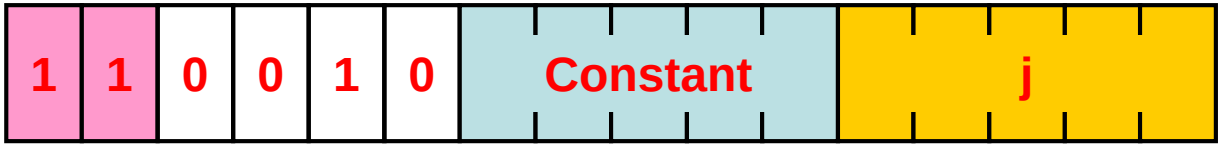


The effect of the LDA instruction is shown below



LDC c j

- **The LDC instruction assigns the value from its first operand to the Rj register**.
- The instruction format is as follows:



- The first operand is considered as a 5-bit unsigned constant (0-31).
- Instruction format is always **32**, i.e., the entire Rj register is updated. The bits 31-5 of the register Rj are nullified (set to 0s).
- Memory state is not considered in the instruction, and the memory state does not change.

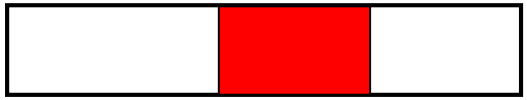
Suggested assembly statement for the LDC instruction:

```
Rj := Constant;
```

The effect of the LDC instruction is shown below

Before LDC

LDC instruction



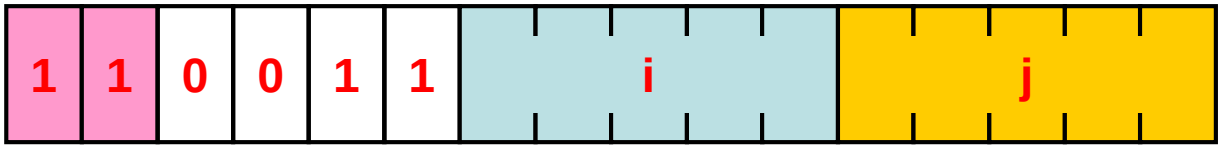
R_j

After LDC

R_j

ST i j

- **The ST instruction copies the value of Ri to the memory by address taken from the register Rj**.
- The instruction format is as follows:

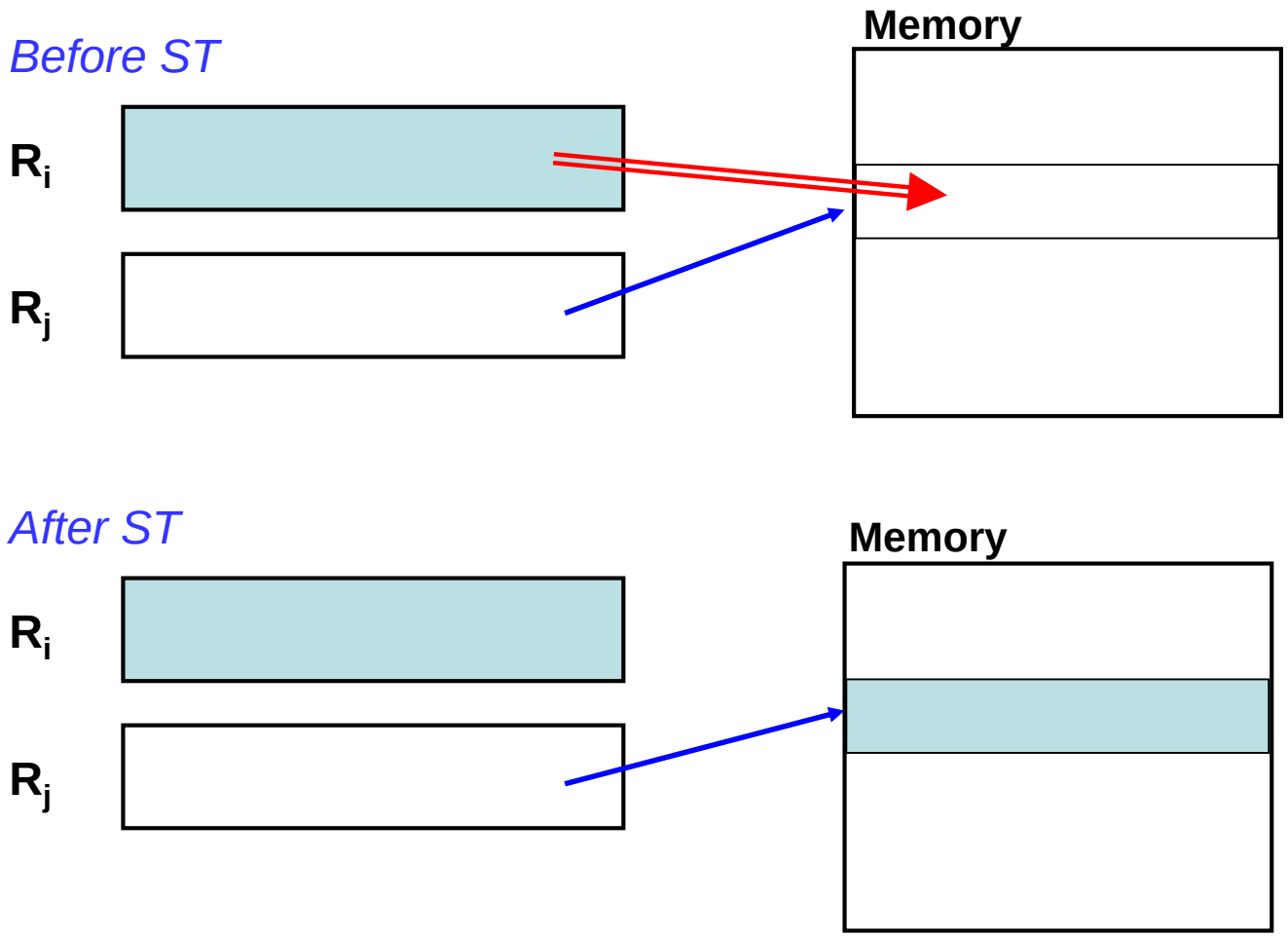


- The contents of the register Ri is treated as an arbitrary value. The contents of the register Rj is considered as a 32-bit address of a 32-bit memory word.
- Instruction format is always **32**, i.e., the entire 32-bit register is copied to the memory.
- Memory state is not considered in the instruction, and the memory state does not change.
- The contents of Ri and Rj registers do not change.

Suggested assembly statement for the LDC instruction:

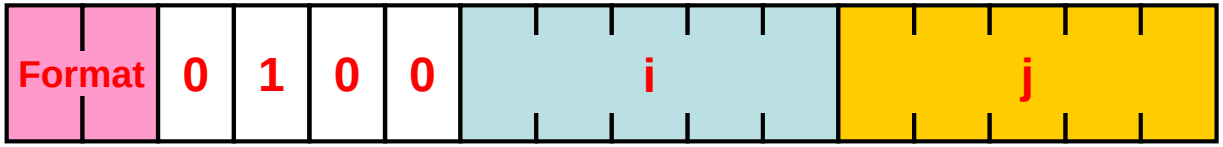
```
*Rj := Ri;
```

The effect of the ST instruction is shown below



MOV i j

- **The MOV instruction copies the value from register Ri to the register Rj**.
- Memory state is not considered in the instruction, and the memory state does not change.
- The instruction format is as follows:



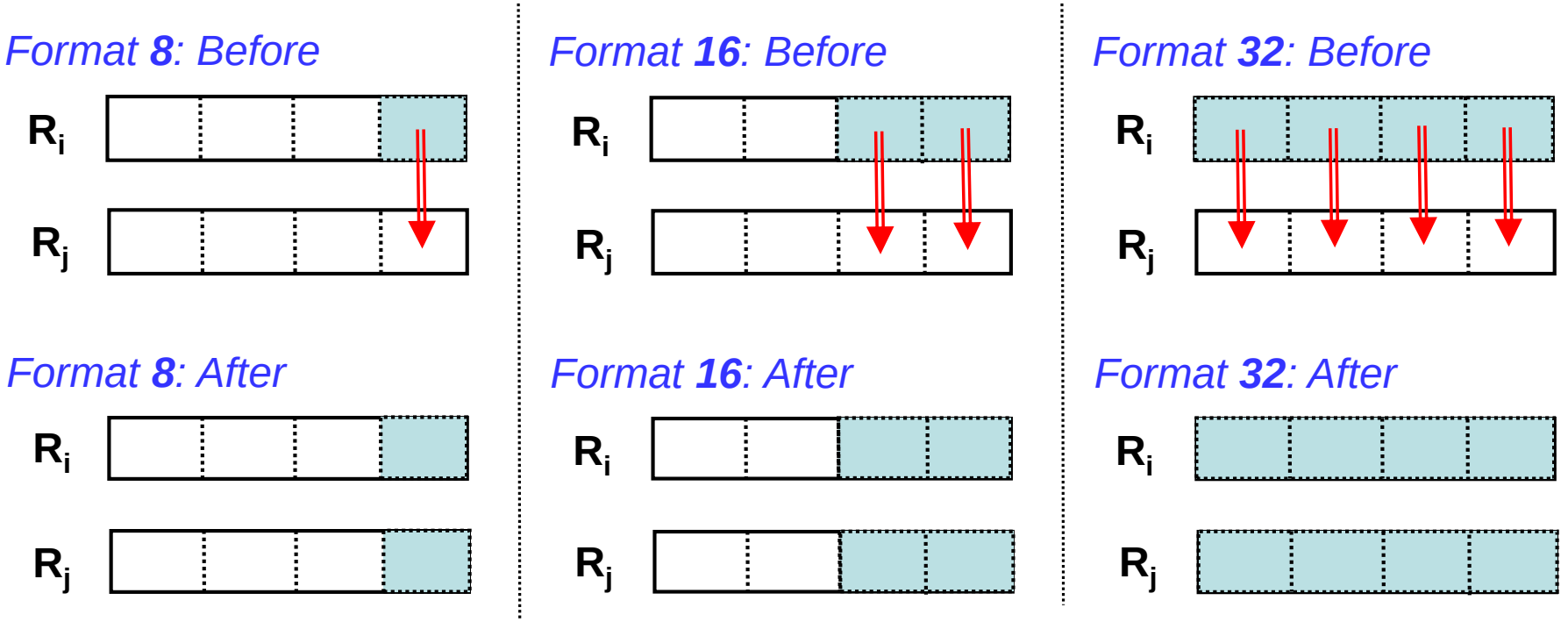
Suggested assembly statement for the MOV instruction:

```
Rj := Ri;
```

Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

The effect of the MOV instruction is shown below



- Instruction format 8: the lowest byte is copied; three highest bytes of Rj remain the same. The original value of Rj's lowest byte is lost.

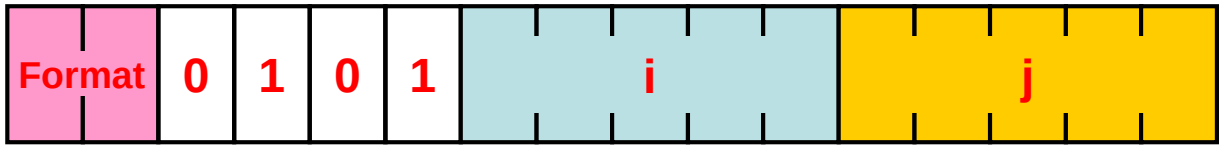
  Instruction format 16: two lowest bytes are copied; two highest bytes of Rj remain the same. The original value of Rj's two lowest bytes is lost.

  Instruction format 32: the entire 32-bit register is copied. The original value of Rj is lost.

- Memory state is not considered in the instruction, and the memory state does not change.

ADD i j

- **The ADD instruction denotes the two's complement arithmetic addition. The contents of registers Ri and Rj are arithmetically added, and the result is put into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- If the addition gives a result which cannot be put into the format specified in the instruction, then **overflow** happens: ?????

Suggested assembly statement for the ADD instruction:

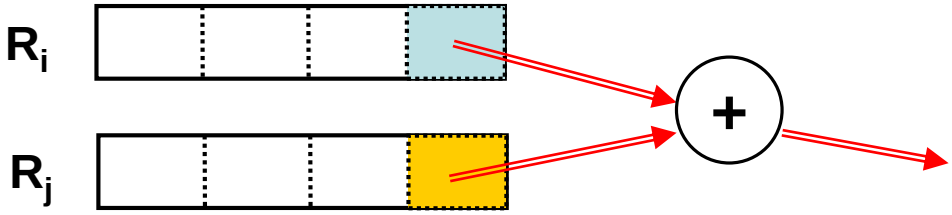
```
Rj += Ri;
```

Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

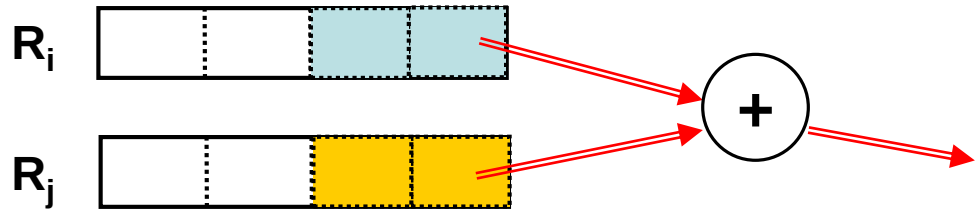
The effect of the ADD instruction is shown below

*Format **8**: Before*



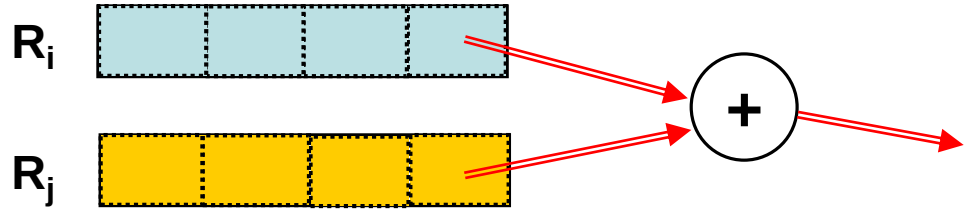
*Format **8**: After*

*Format **16**: Before*

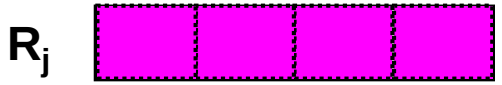


*Format **16**: After*

*Format **32**: Before*

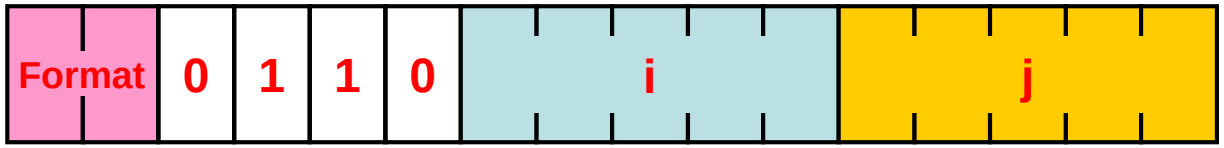


*Format **32**: After*



SUB i j

- **The SUB instruction denotes the two's complement arithmetic subtraction. The contents of register Ri is subtracted from the contents of the register Rj, and the result is put into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- If the subtraction gives a result which cannot be put into the format specified in the instruction, then **……** happens: ?????

Suggested assembly statement for the SUB instruction:

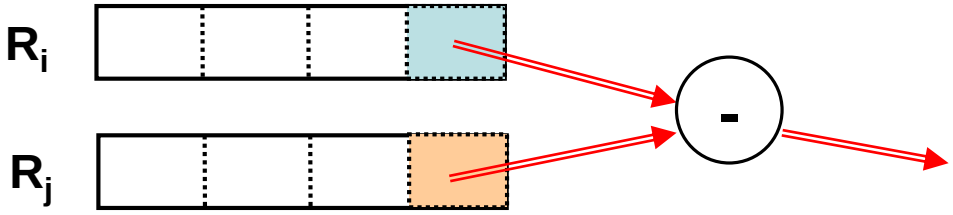
```
Rj -= Ri;
```

Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

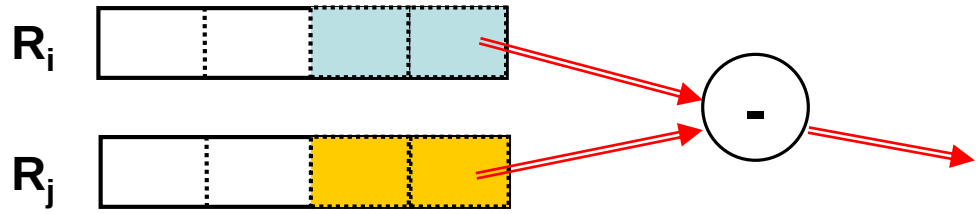
The effect of the SUB instruction is shown below

*Format **8**: Before*



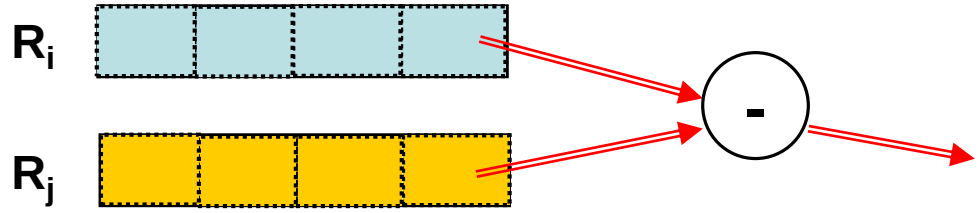
*Format **8**: After*

*Format **16**: Before*



*Format **16**: After*

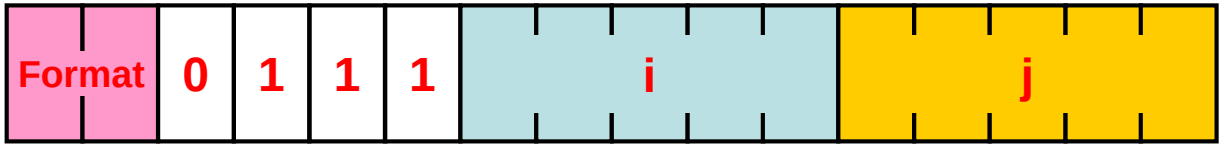
*Format **32**: Before*



*Format **32**: After*

ASR i j

- **The ASR instruction arithmetically shifts the contents of the register Ri <u>one bit right</u>, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.

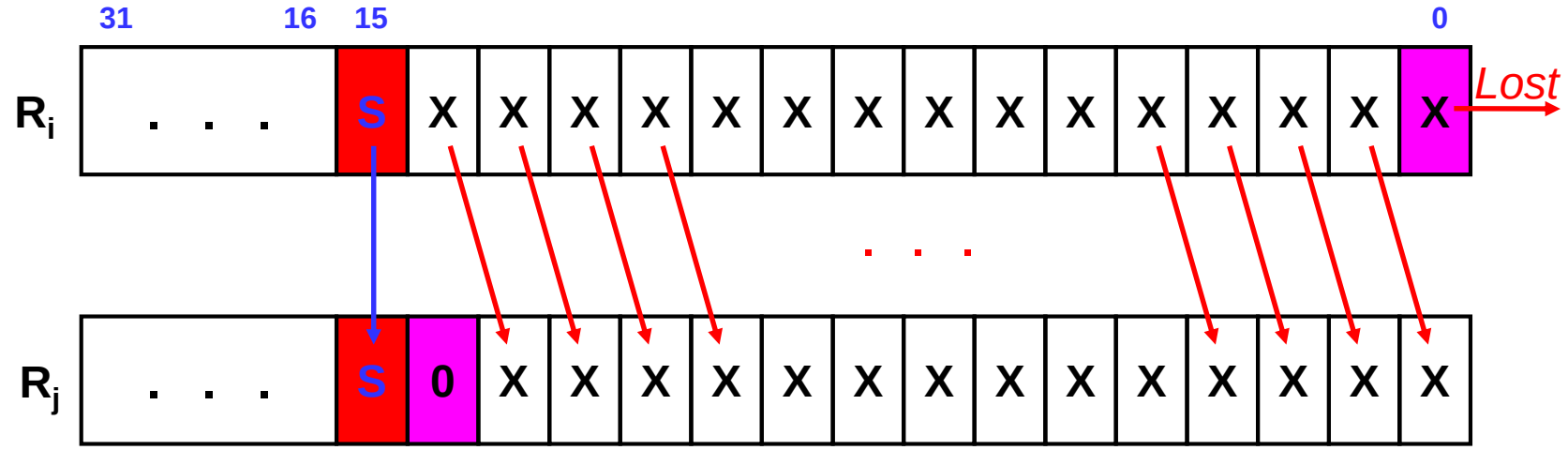
Suggested assembly statement for the ASR instruction:

```
Rj >>= Ri;
```

Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

- *Arithmetic* shift means that the sign bit does not participate in the operation but remains on its usual place.
- The leftmost bit of the operand gets the value of 0. The rightmost bit of the operand is always lost.
- The contents of the Ri register does not change.
- The effect of the ASR instruction for format 16 is shown below. The operation for formats 8 and 32 is performed in the similar way.

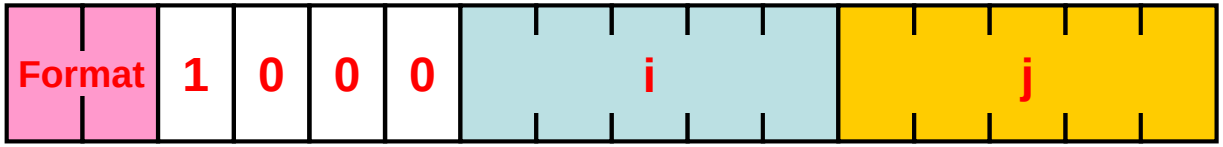


What to do with the high bits of the result for 8 and 16 formats?

- Copy them from the source register.
- Remain them as they were (no modifications).
- Set them to 0s.

ASL i j

- **The ASL instruction arithmetically shifts the contents of the register Ri <u>one bit left</u>, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.

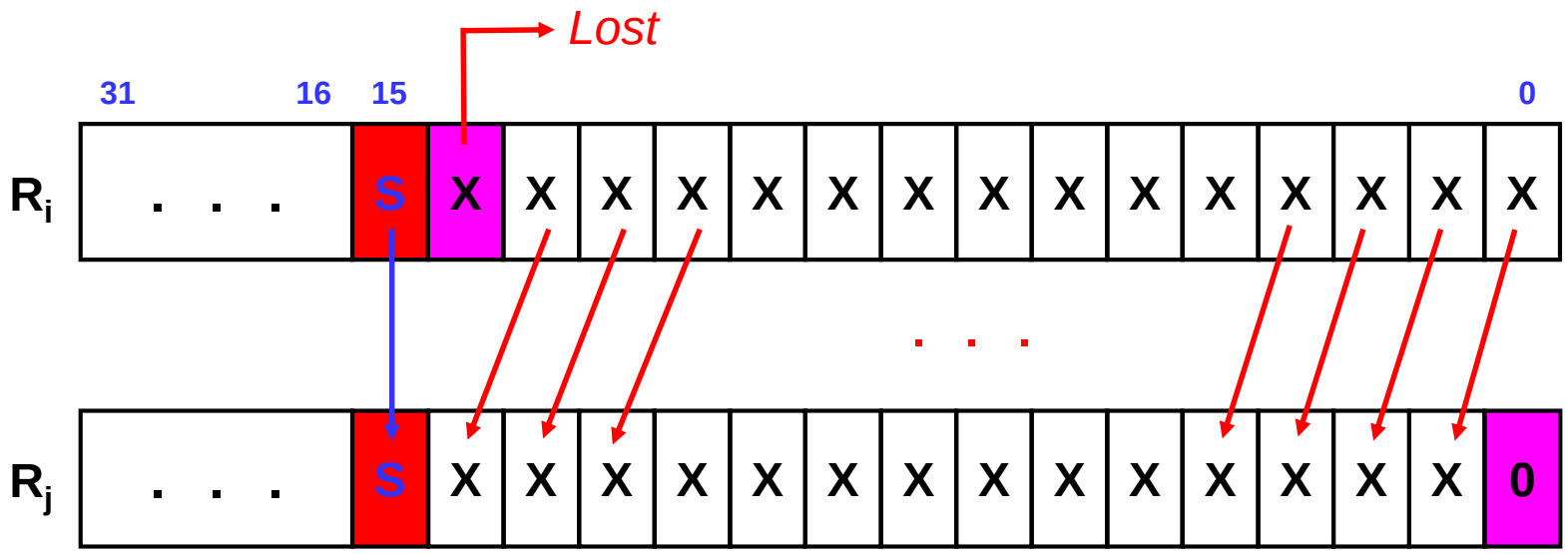
Suggested assembly statement for the ASL instruction:

```
Rj <<= Ri;
```

Additional assembly directives specifying the current instruction format:

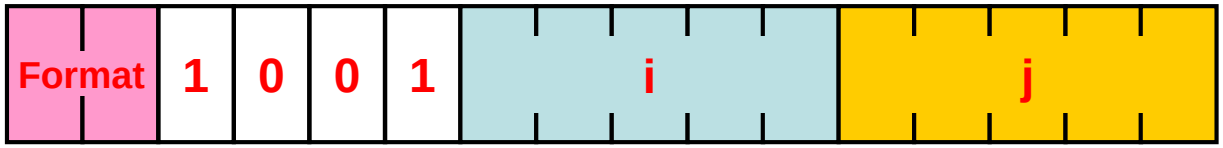
`.format 8;` **or** `.format 16;` **or** `.format 32;`

- *Arithmetic* shift means that the sign bit does not participate in the operation but remains on its usual place.
- The leftmost bit of the operand is always lost. The rightmost bit of the operand gets the value of 0.
- The contents of the Ri register does not change.
- The effect of the ASL instruction for format 16 is shown below. The operation for formats 8 and 32 is performed in the similar way.



OR i j

- **The OR instruction applies logical addition (“OR”) operator to every pair of bits taken from registers Ri and Rj, respectively, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- The contents of the Ri register does not change.

Suggested assembly statement for the OR instruction:

```
Rj |= Ri;
```

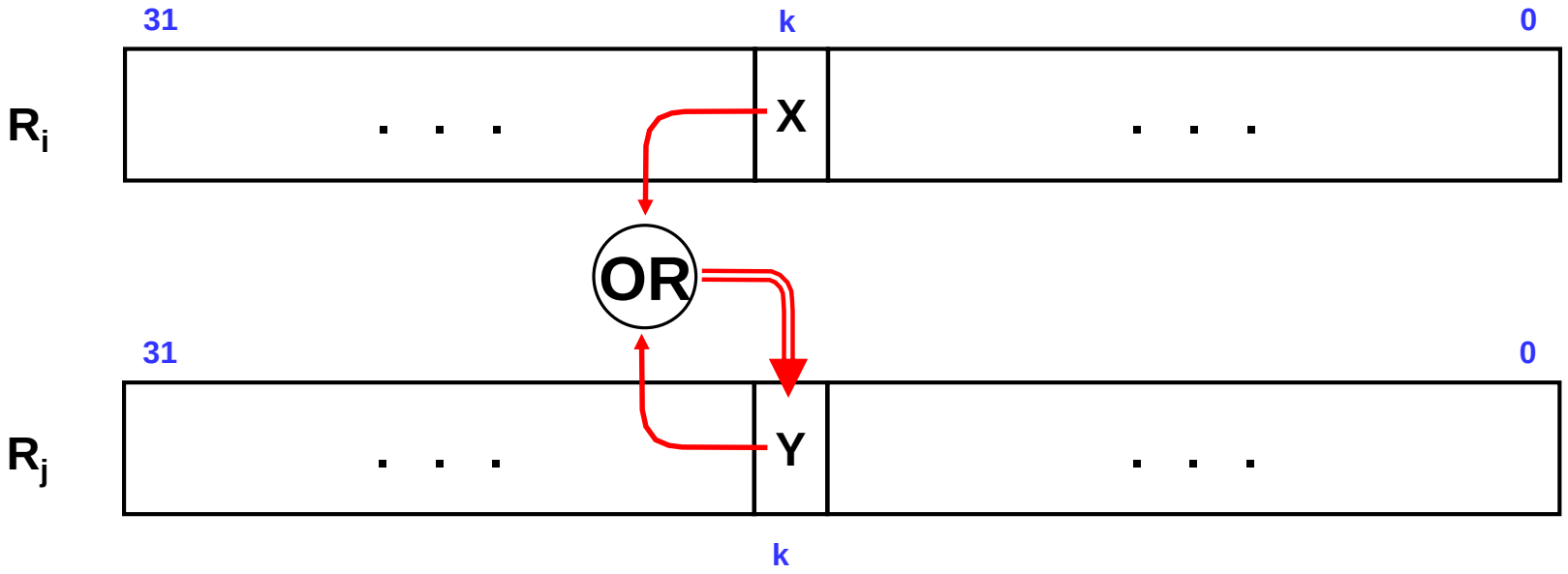
Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every pair of bits independently.
- The rule for the **OR** operation performed on each pair of bits is defined as follows:

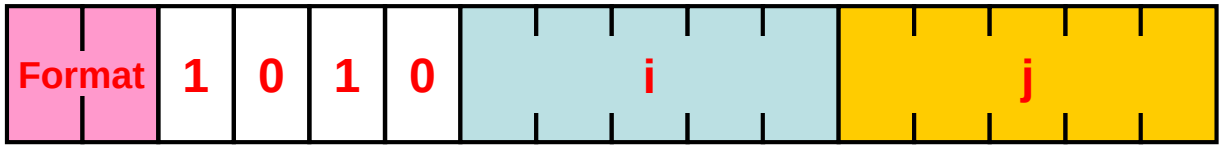
X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	1

- The mechanism of the OR instruction – for one pair of bits - is shown below. Here, k∈[0..31] for format 32, k∈[0..15] for format 16, and k∈[0..7] for format 8.



AND i j

- **The AND instruction applies logical multiplicative (“AND”) operator to every pair of bits taken from registers Ri and Rj, respectively, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- The contents of the Ri register does not change.

Suggested assembly statement for the AND instruction:

```
Rj &= Ri;
```

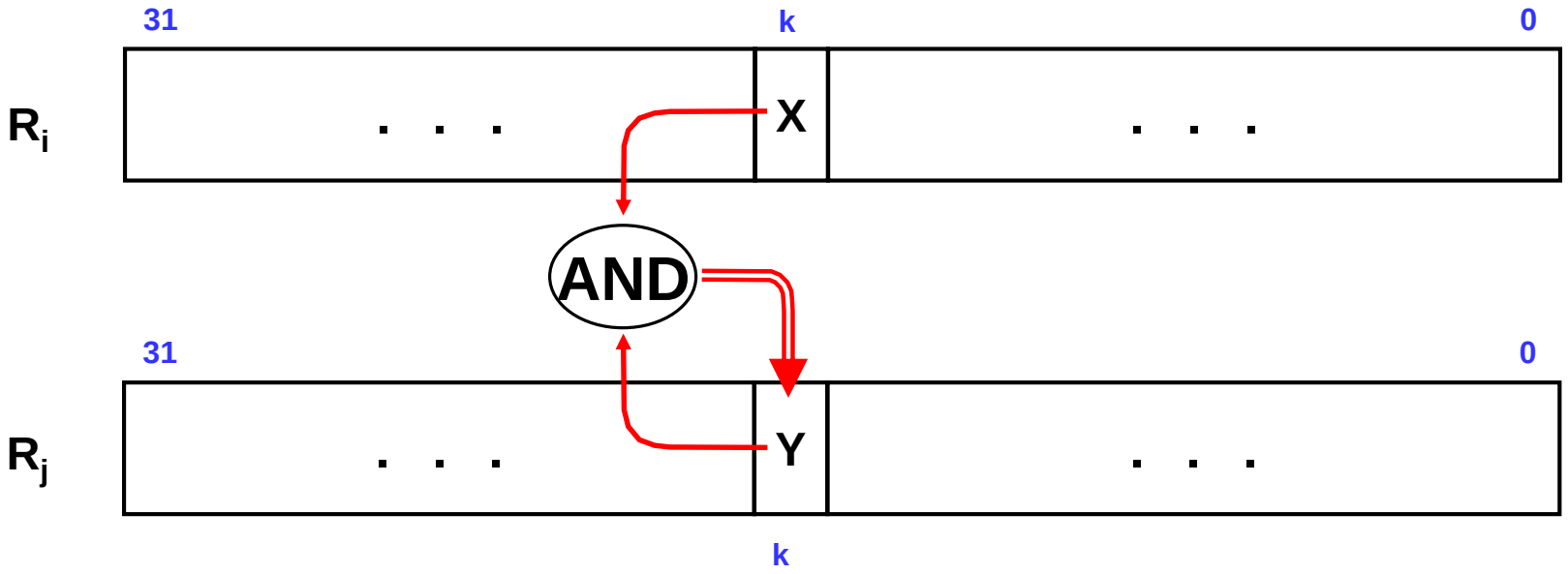
Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every pair of bits independently.
- The rule for the AND operation performed on each pair of bits is defined as follows:

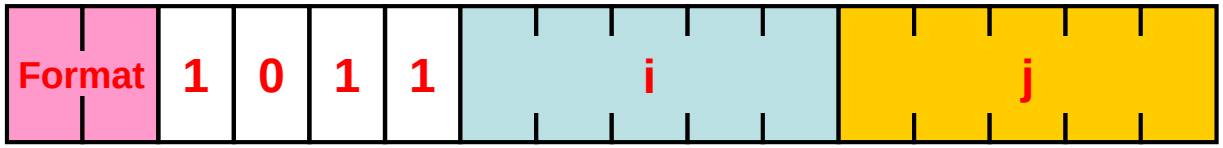
X	Y	Result
0	0	0
0	1	0
1	0	0
1	1	1

- The mechanism of the AND instruction – for one pair of bits - is shown below. Here, k∈[0..31] for format 32, k∈[0..15] for format 16, and k∈[0..7] for format 8.



XOR i j

- **The XOR instruction applies logical exclusive OR (“XOR”) operator to every pair of bits taken from registers Ri and Rj, respectively, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- The contents of the Ri register does not change.

Suggested assembly statement for the AND instruction:

```
Rj ^= Ri;
```

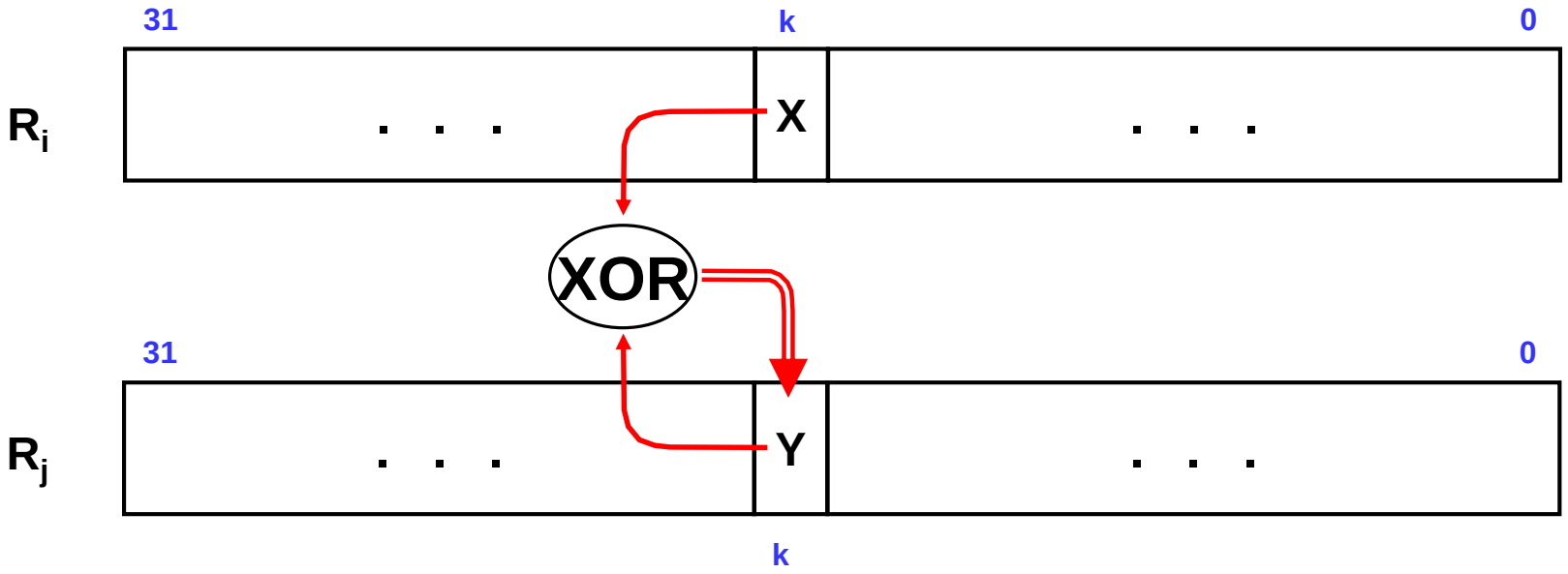
Additional assembly directives specifying the current instruction format:

`.format 8;` **or** `.format 16;` **or** `.format 32;`

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every pair of bits independently.
- The rule for the XOR operation performed on each pair of bits is defined as follows:

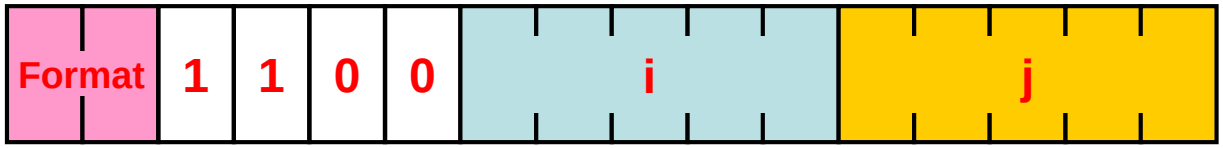
X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	0

- The mechanism of the XOR instruction – for one pair of bits - is shown below. Here, k∈[0..31] for format 32, k∈[0..15] for format 16, and k∈[0..7] for format 8.



LSL i j

- **The LSL instruction logically shifts the contents of the register Ri <u>one bit left</u>, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- The contents of the Ri register does not change.

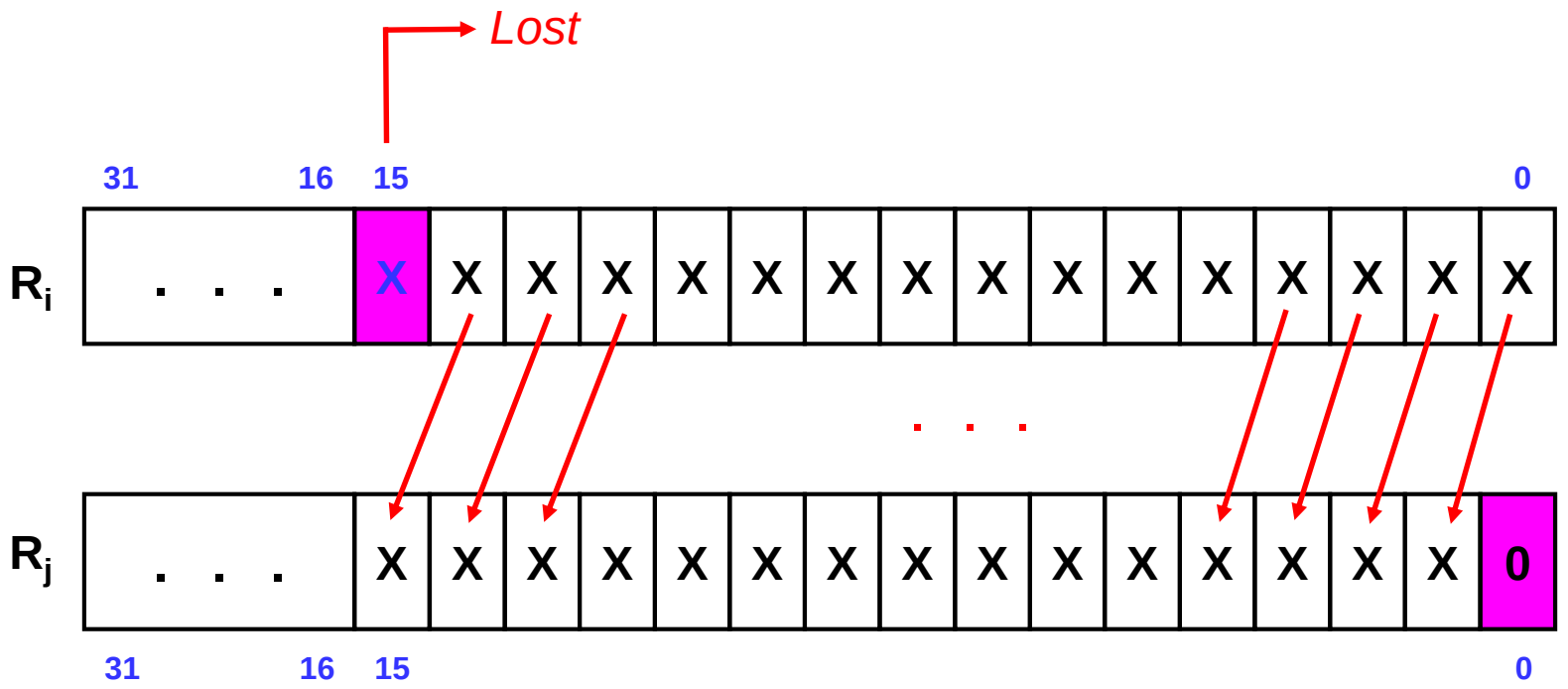
Suggested assembly statement for the AND instruction:

```
Rj <= Ri;
```

Additional assembly directives specifying the current instruction format:

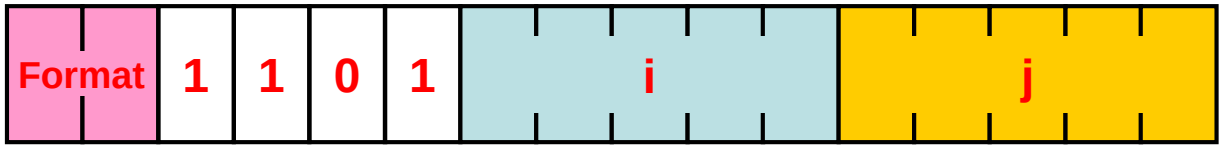
`.format 8;` **or** `.format 16;` **or** `.format 32;`

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every bit independently.
- The leftmost bit of the operand is always lost. The rightmost bit of the operand gets the value of 0.
- The effect of the LSL instruction for format 16 is shown below. The operation for formats 8 and 32 is performed in the similar way.



LSR i j

- **The LSR instruction logically shifts the contents of the register Ri <u>one bit right</u>, and puts the result into the register Rj.**
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- The contents of the Ri register does not change.

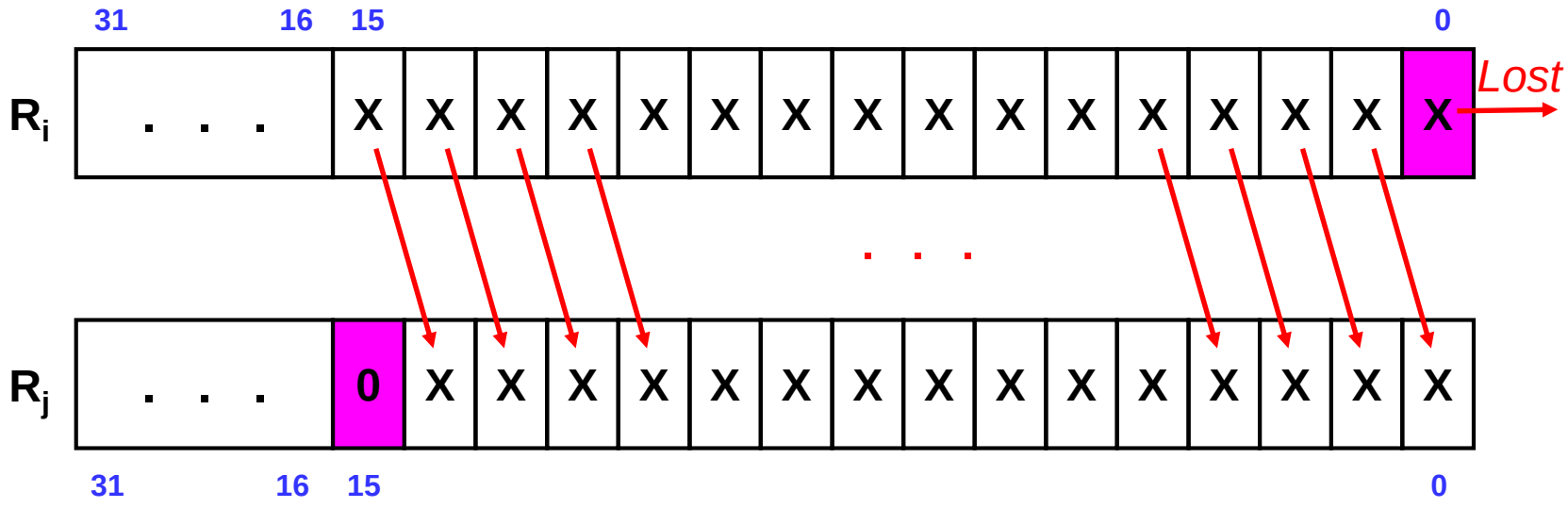
Suggested assembly statement for the AND instruction:

```
Rj >= Ri;
```

Additional assembly directives specifying the current instruction format:

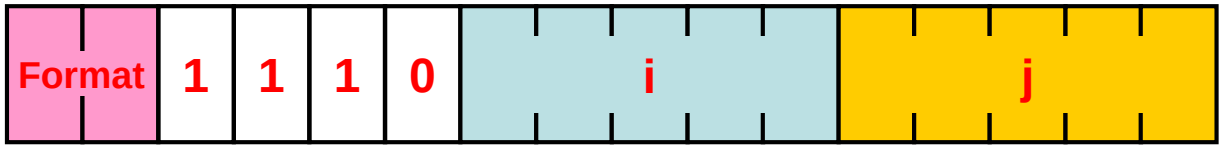
`.format 8;` **or** `.format 16;` **or** `.format 32;`

- In this instruction, the contents of registers Ri and Rj are considered as two bit scales. The operation is performed on every bit independently.
- The rightmost bit of the operand is always lost. The leftmost bit of the operand gets the value of 0.
- The effect of the LSR instruction for format 16 is shown below. The operation for formats 8 and 32 is performed in the similar way.



CND i j

- **The CND instruction arithmetically compares the contents of registers Ri and Rj and puts the result of the comparison (as a set of 1-bit signs) to the register Rj.**
- Signs occupy four lowest bits of the result (see next slides for details and for the meaning of the signs).
- The instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.

Suggested assembly statement for the CND instruction:

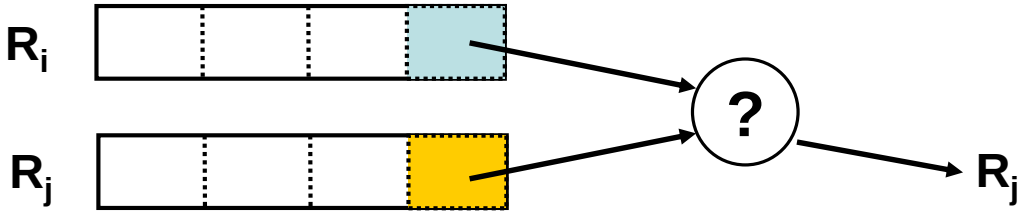
```
Rj ?= Ri;
```

Additional assembly directives specifying the current instruction format:

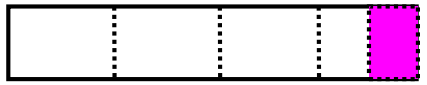
`.format 8;` **or** `.format 16;` **or** `.format 32;`

The effect of the CND instruction is shown below

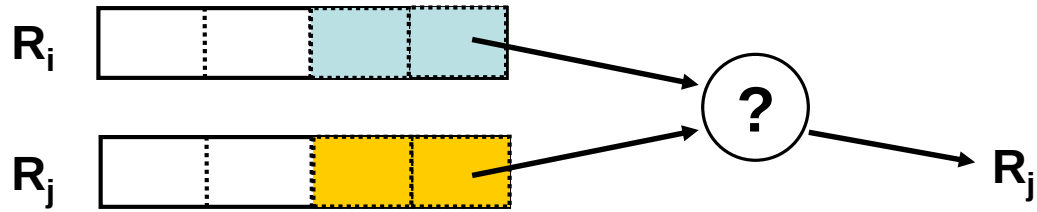
*Format **8**: Before*



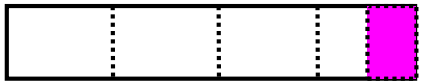
*Format **8**: After*



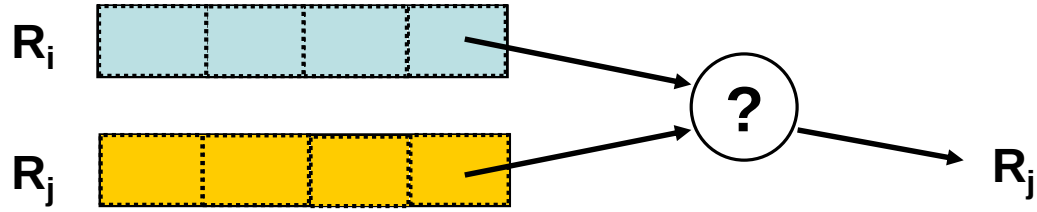
*Format **16**: Before*



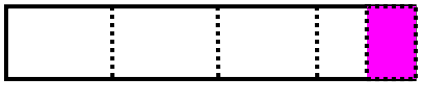
*Format **16**: After*



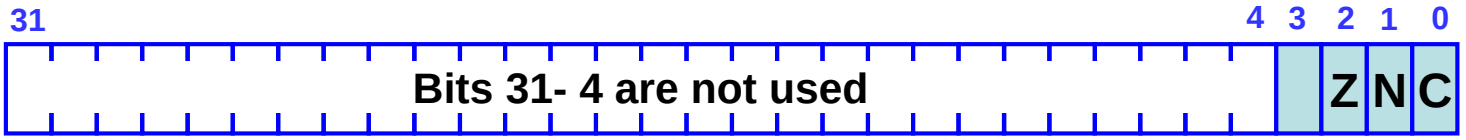
*Format **32**: Before*



*Format **32**: After*



- The contents of the Ri register does not change.
- The result of the instruction (i.e., the contents of the Rj register) is as follows:



- Bit 3: Reserved (always 0)
- Bit 2 (**Z**): **1**, if Ri = Rj,
  0, otherwise
- Bit 1 (**N**): **1**, if Ri < Rj,
  0, otherwise
- Bit 0 (**C**): **1**, if Ri > Rj,
  0, otherwise

- Signs are mutually exclusive: i.e., the semantics of signs assumes that the only one sign is set as the result of the comparison.
- The result of the comparison can be used in an arbitrary way. Perhaps the most important one is to use it for organizing conditional jumps (see CBR instruction).

- Signs can be checked using logical instructions (e.g., AND) together with appropriate masks.

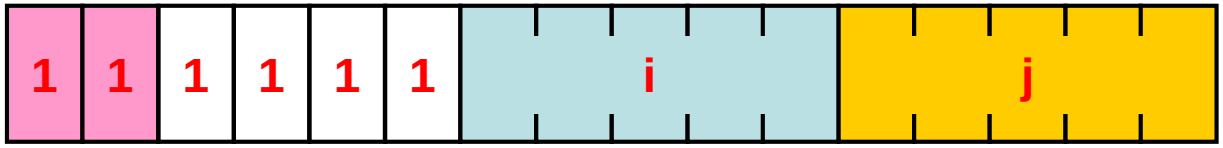
  For example, the ≤ condition can be treated as either < or = conditions, and the corresponding mask is binary 110. Similarly, the ≥ condition is either > or = conditions, and the mask is binary 101. Finally, the inequality ≠ condition is either ≤ or ≥ conditions, and the mask for selecting is binary 011.

  Therefore, all six possible comparison results (<, ≤, >, ≥, =, ≠) can be extracted using AND instruction with the following masks:

Relation	***Mask***
<	010
≤	110
>	001
≥	101
=	100
≠	011

CBR i j

- **The CBR instruction checks the contents of the Ri register. If it is non-zero, then 1) the address of the next instruction (i.e., current value of the PC register + 2) is stored in the Ri register, and 2) the value of the Rj register is set to the PC register. This means that the next instruction will be fetched by the address taken from the Rj register.**
- The instruction format is as follows:

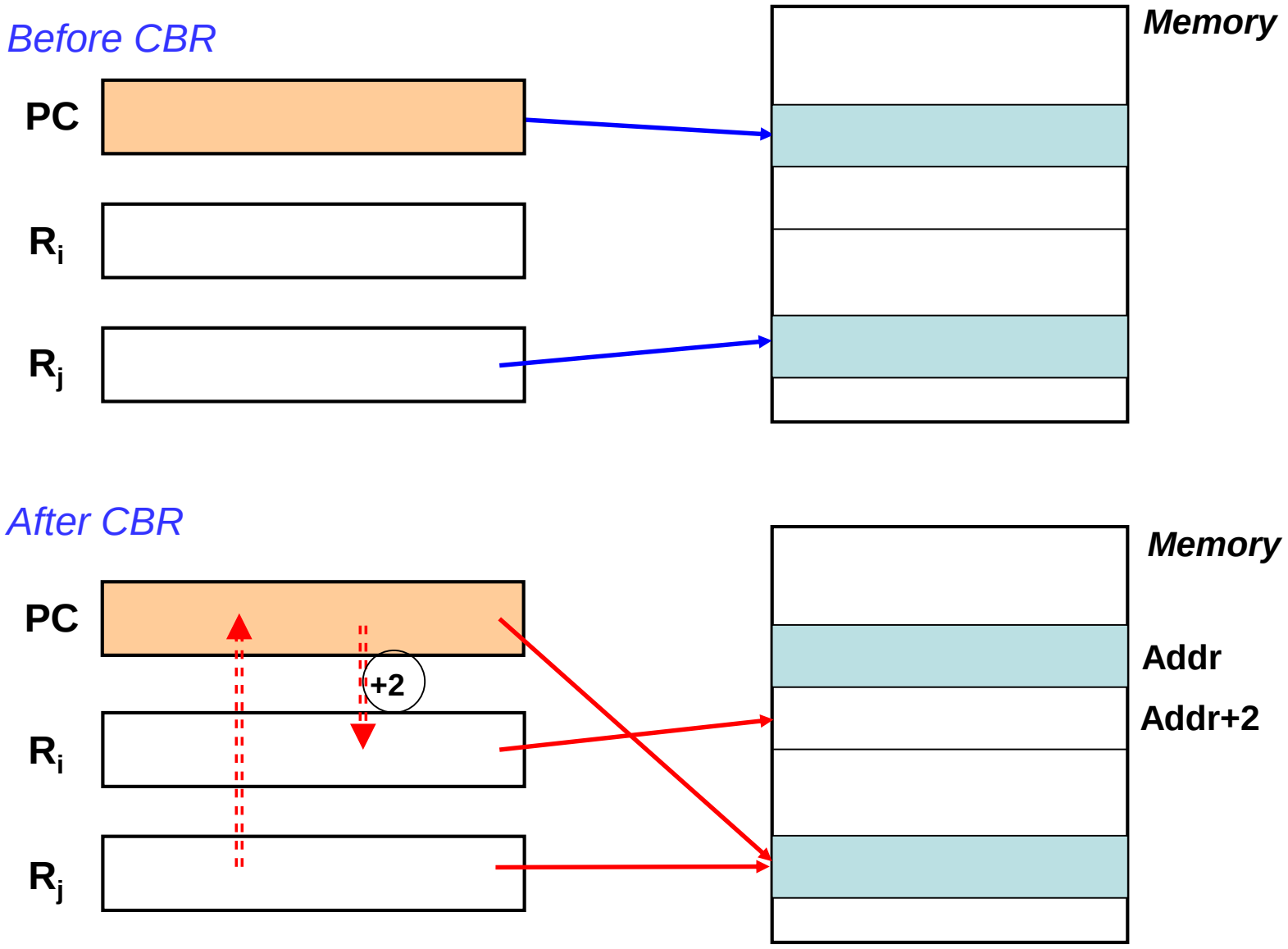


- Memory state is not considered in the instruction, and the memory state does not change.
- Format code does not affect the instruction's execution.
- The contents of the Rj register does not change.

Suggested assembly statement for the CBR instruction:

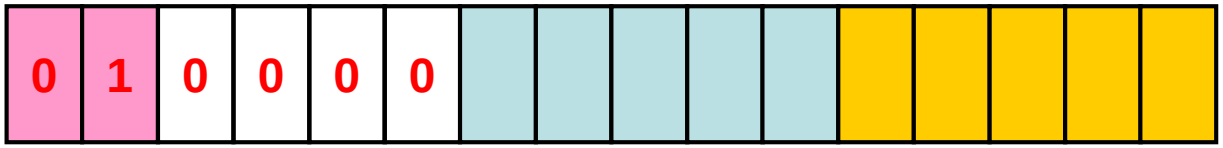
```
if Ri goto Rj;
```

The effect of the CBR instruction (for the case when Ri is non-zero) is shown below



NOP / STOP

- **The NOP instruction performs no actions.**
- The NOP instruction format is as follows:

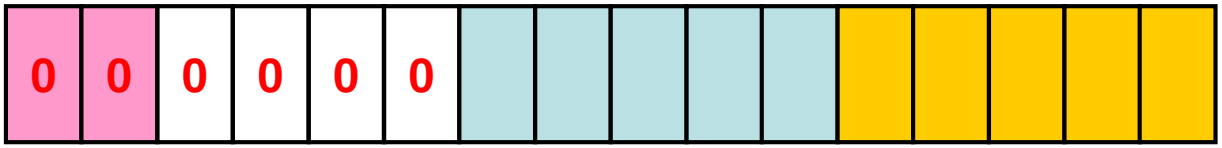


- Memory state is not considered in the instruction, and the memory state does not change.
- The value of the operand part of the instruction (bits from 9 to 0) does not affect the execution. If necessary it can be used as a place for keeping constants. In this case, the possible value range of the constant is [0..1023].
- The NOP instruction is used as a placeholder (for example to meet alignment requirements), or as a constant storage.

Suggested assembly statement for the NOP instruction:

```
skip
```

- **The STOP instruction causes the program execution to interrupt.**
- The STOP instruction format is as follows:



- Memory state is not considered in the instruction, and the memory state does not change.
- The value of the operand part of the instruction (bits from 9 to 0) does not affect the execution.

Suggested assembly statement for the STOP instruction:

```
stop
```