Compiler Construction: Introduction

Project ERA
Simulator
Assembler
System-level Language

Eugene Zouev

Fall Semester 2018

Innopolis University

Project ERA: Contents

1. ERA logical architecture:
   Memory, registers, common instruction format.
2. ERA Typical memory structure.
3. ERA Assembler & System-level language.
4. Instruction set and instruction mnemonics
5. Overall compilation & execution model
6. Some comments for assembler & language implementation.
7. Appendix: The code example

Project ERA:
The tasks & recommended workload

ERA Instruction set simulator

- Design and implement the ERA overall memory structure model
- Implement instruction execution over memory
- Recommended team size: **2 persons**

ERA Assembler (as a separate subproject)

- Implement the Assembler & a test suite
- Recommended team size: **2 persons**

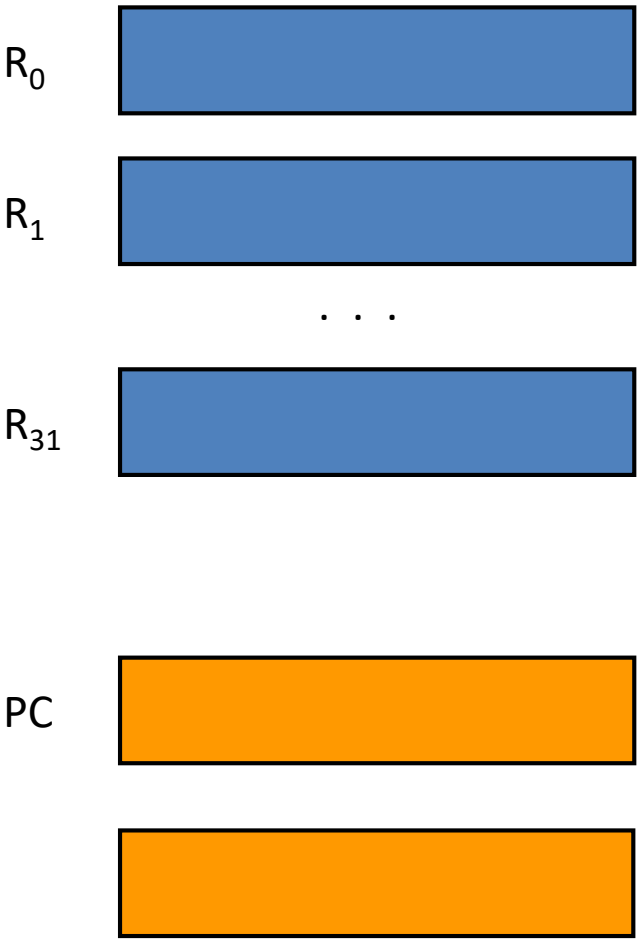
ERA System-level language

- Design and implement the language compiler (without assembler part) & corresponding test suite
- Integrate assembler part into the language implementation
- Recommended team size: **3-4 persons**

Project ERA

1. ERA Logical Architecture

CPU Logical Structure

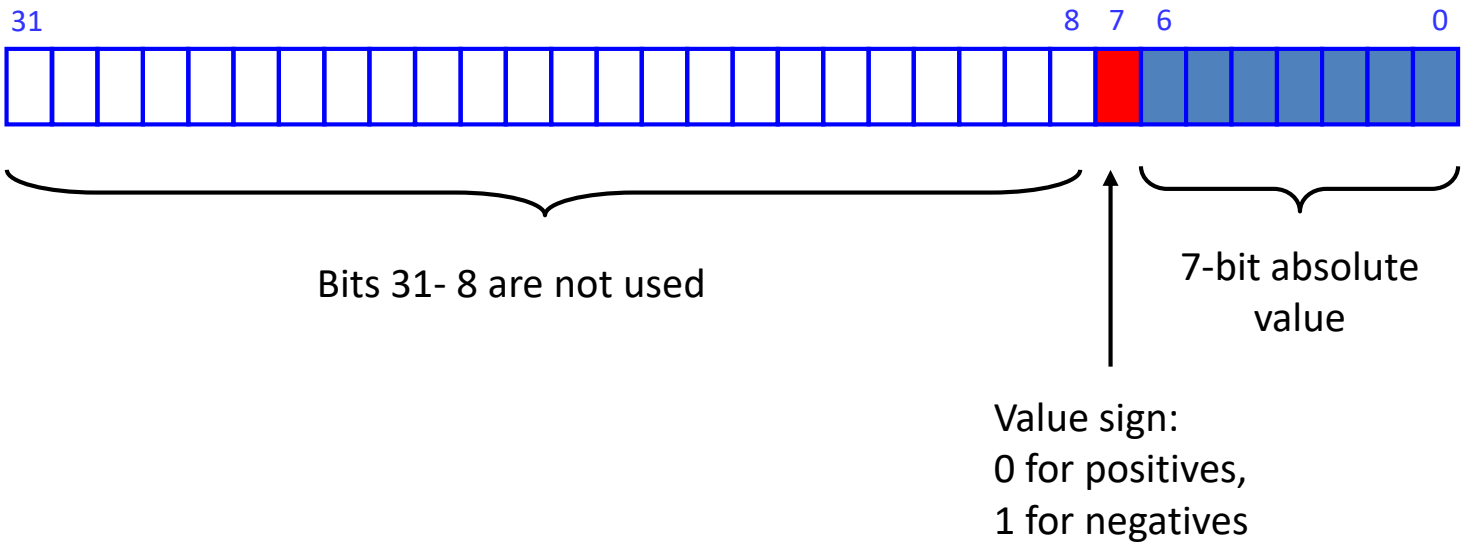


- 32 common registers (R0…R31) each of which is of 32 bits. Every register can contain either a value or an address of a memory location.
  An address in a register can address any byte of memory.
- The CPU performs all actions on the operands taken from common registers. There are instructions for loading 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

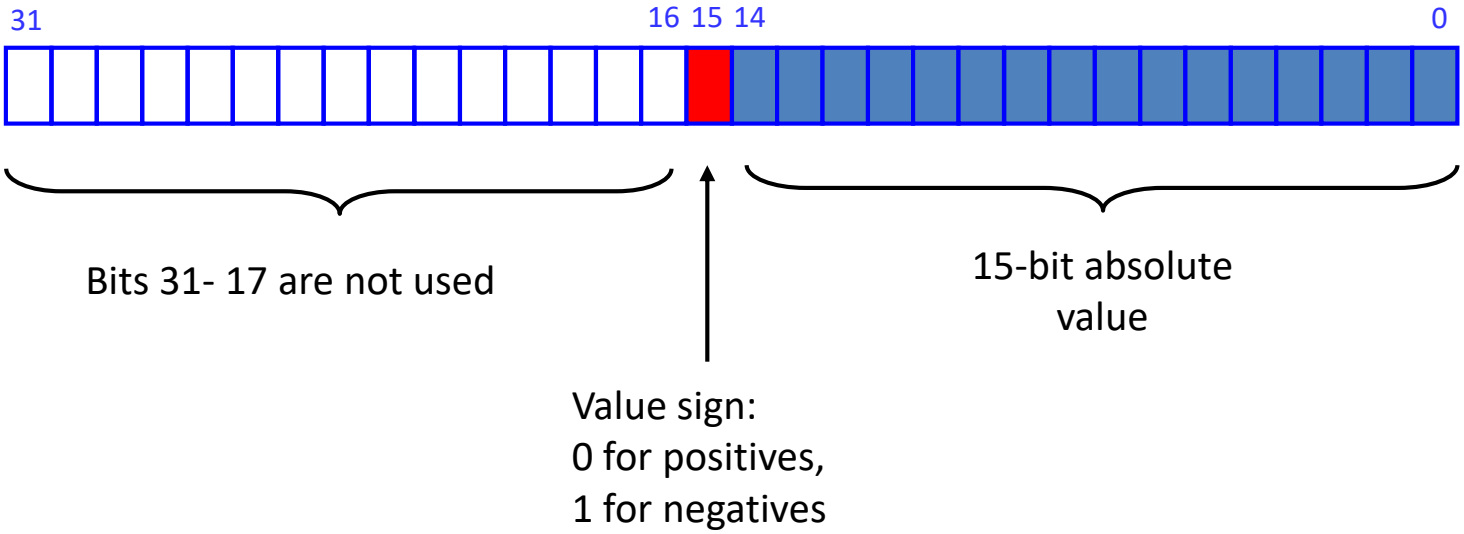
- Alignment?

- 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

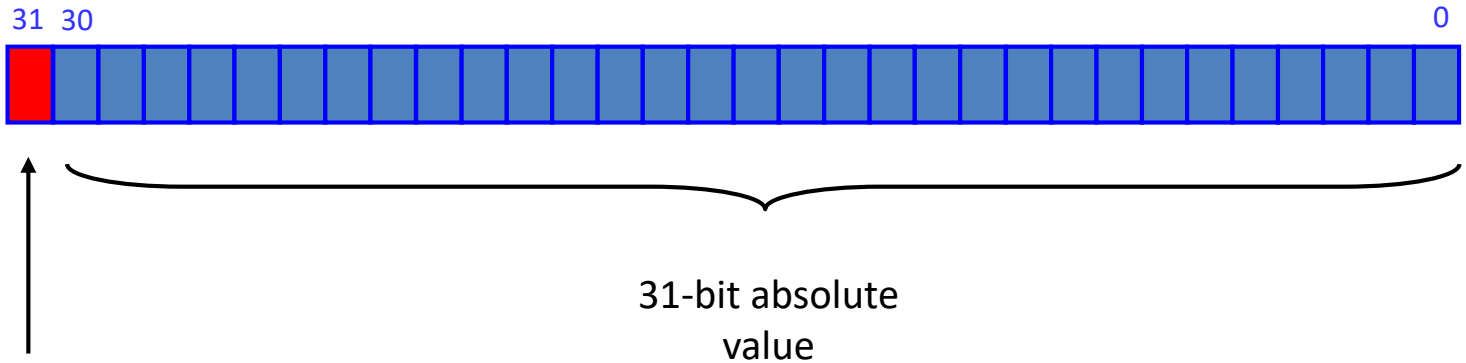
- The range of possible 16-bit integer values is [-32768..32767].
- The format of 16-bit signed integers is shown below:



- 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:

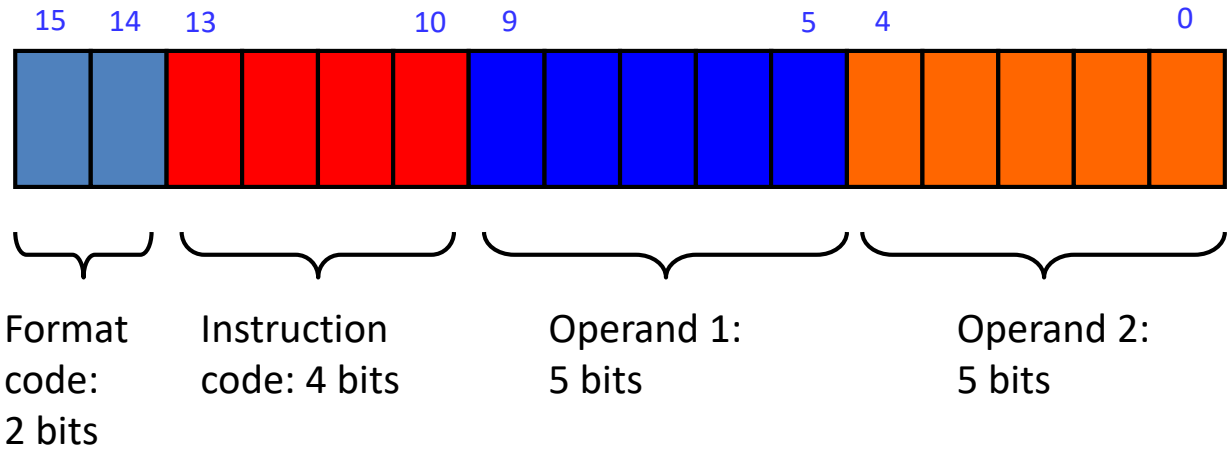


- Alignment?

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).

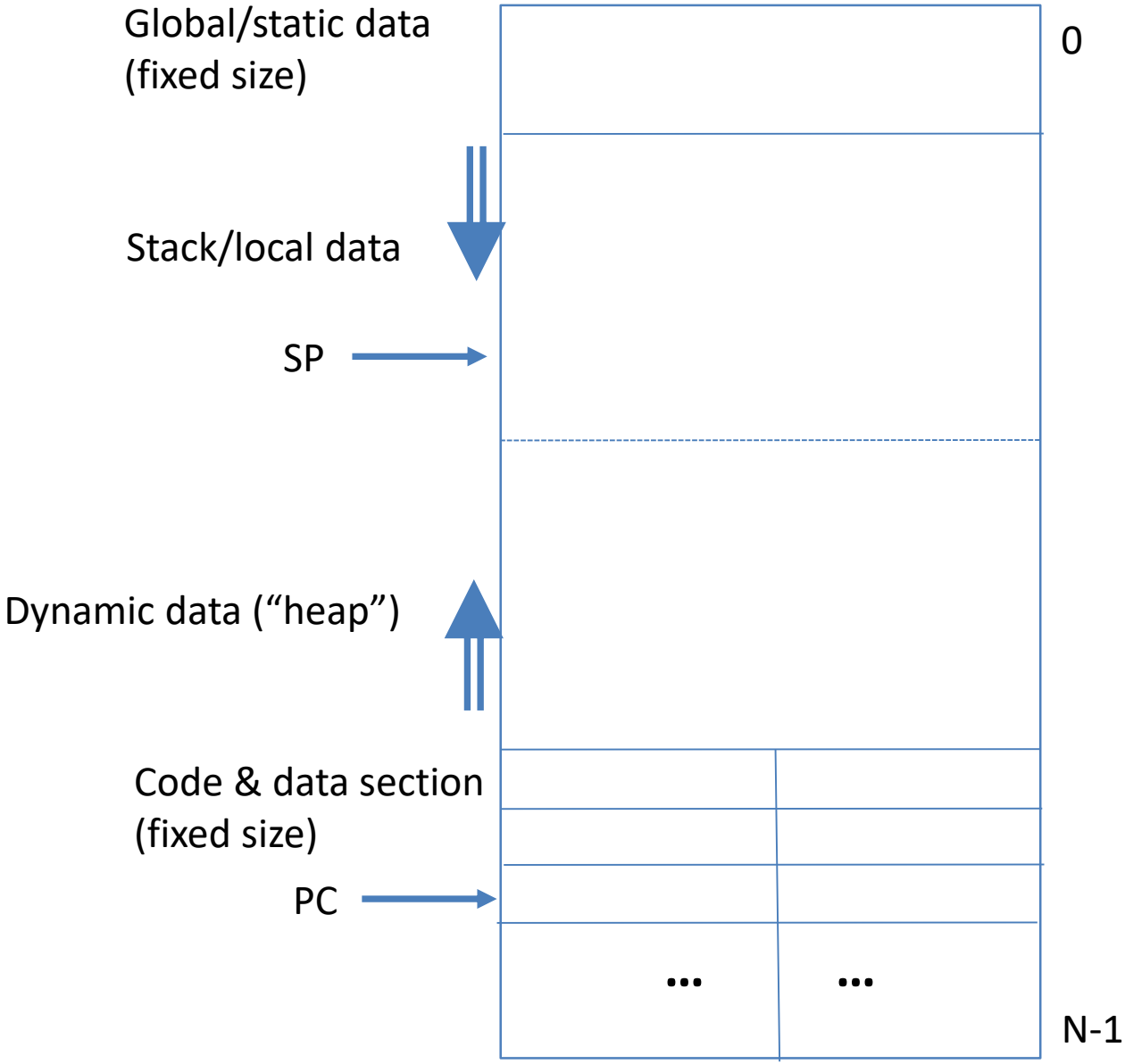
Common Instruction Format



- 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?

Project ERA

2. ERA Memory execution image: some raw ideas


Global/static data
(fixed size)
0
Stack/local data
SP
Dynamic data (“heap”)
Code & data section
(fixed size)
PC
...
...
N-1

Some general-purpose registers should be assigned for the following special needs:

IR
Instruction Register holding the current execution instruction.

PC
Program Counter Register holding the address of the next execution instruction.

SB
Static Base Register holding the *Static Base* address of global data.

SP
Stack Pointer Register holding the address of the top of stack.

FP
Frame Pointer Register holding the base address of data local to procedures.

Project ERA

3. ERA Assembler & System-level language

```
Program    : { Unit }

Unit       : Code | Data | Module | Routine

Code       : code { Variable | Constant | Statement } end

Data       : data Identifier [ Literal { , Literal } ] end

Module     : module Identifier { Declaration } end

Declaration : Variable | Constant | Routine

Variable   : Type VarDefinition { , VarDefinition } ;

Type       : int | short | byte

VarDefinition
           : Identifier [ := Expression ]
           | Identifier [ Expression ]

Constant   : const ConstDefinition { , ConstDefinition } ;

ConstDefinition : Identifier = Expression

Statement    : { Label } ( AssemblerStatement
                         | ExtensionStatement
                         | Directive)

Label        : < Identifier >
```

```
Routine   : [ Attribute ]
            routine Identifier [ Parameters ] [ : Results ]
            ( ; | do RoutineBody end )

Attribute : start| entry

Parameters  : ( Parameter { , Parameter } )

Parameter   : Type Identifier | Register

Results     : Register { , Register }

RoutineBody : { Variable | Constant | Statement }
```

```
Primary    : VariableReference  // Identifier
           | Dereference        // *VariableReference | *Register
           | ArrayElement       // VariableReference[Expression]
           | DataElement        // DataIdentifier[Expression]
           | Register           // R0 | R1 | ... R31
           | ExplicitAddress    // * Literal

Expression : Operand [ Operator Operand ]

Operand    : Receiver
           | Literal
           | Address   // &VariableReference

Operator   : + | - | * | & | | | ^ | ? | PrimitiveOperator

PrimitiveOperator : = | /= | < | >
```

```
AssemblerStatement
       : skip [ Expression ]          //                        NOP
       | stop [ Expression ]          //                        STOP
       |   Register := * Register     //    Rj :=  *Ri          LD
       |   Register :=   Expression   //    Rj :=   Const       LDA
       | * Register :=   Register     //   *Rj :=   Ri          ST
       |   Register :=   Register     //    Rj :=   Ri          MOV
       |   Register +=   Register     //    Rj +=   Ri          ADD
       |   Register -=   Register     //    Rj -=   Ri          SUB
       |   Register >>=  Register     //    Rj >>=  Ri          ASR
       |   Register <<=  Register     //    Rj <<=  Ri          ASL
       |   Register |=   Register     //    Rj |=   Ri          OR
       |   Register &=   Register     //    Rj &=   Ri          AND
       |   Register ^=   Register     //    Rj ^=   Ri          XOR
       |   Register <=   Register     //    Rj <=   Ri          LSL
       |   Register >=   Register     //    Rj >=   Ri          LSR
       |   Register ?=   Register     //    Rj ?=   Ri          CND
       | if Register goto Register    // if Ri goto Rj          CBR

Register   : R0 | R1 | R2 | R3 | ... | R30 | R31

Directive  : format ( 8 | 16 | 32 ) ;
```

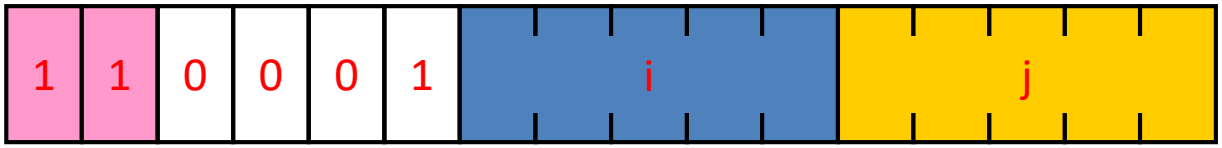
```
ExtensionStatement
            : Assignment | Call | If | While | For
            | Break | Swap | Goto

For         : [ for Identifier ]
                 [ from Expression ]
                    [ to Expression ]
                       [ step Expression ]
              LoopBody

While       : [ while Expression ] LoopBody

LoopBody    : loop { Statement } end

Break       : break

Swap        : Primary <=> Primary ;

Goto        : goto Identifier ;

Assignment : Primary := Expression ;
```

Project ERA

4. ERA Instruction set and instruction mnemonics

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:

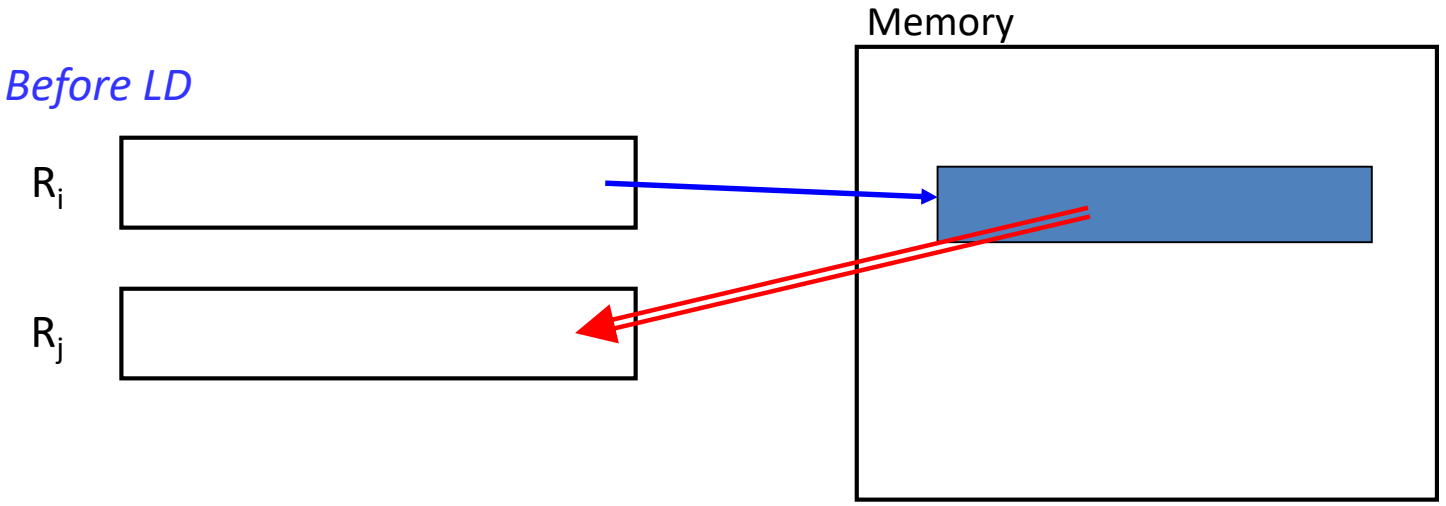


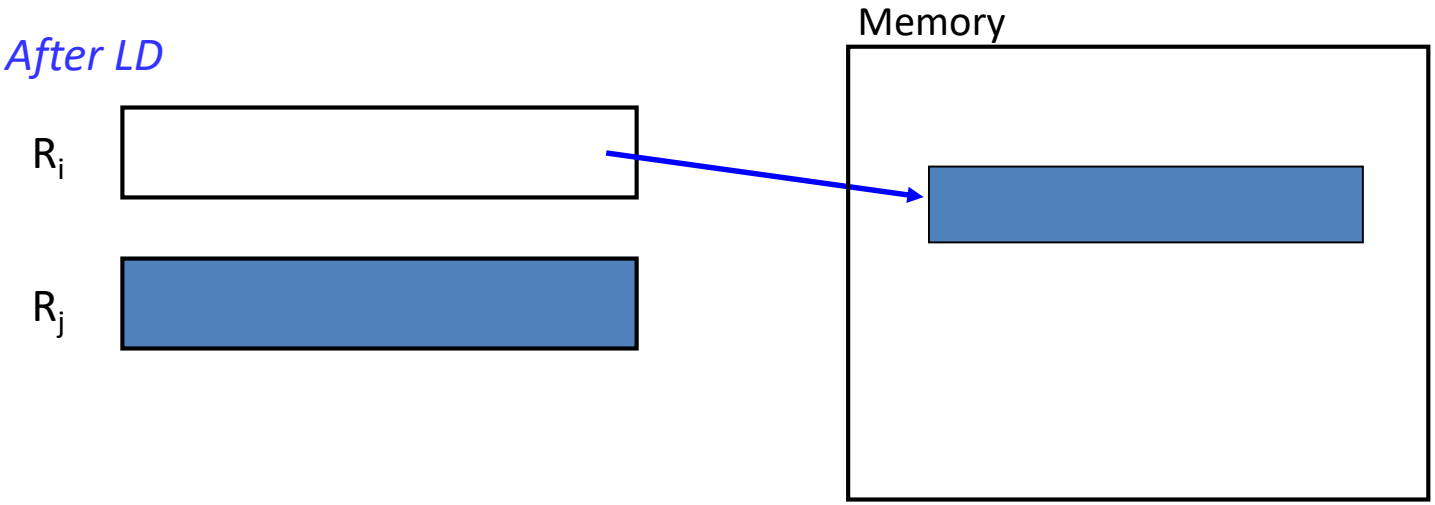
- The contents of the register Ri is considered as a 32-bit address of a 32-bit memory word.
- Instruction format is always 32, i.e., the entire 32-bit word is copied from the memory to the register.
- When the instruction is completed, the original contents of the register Rj is lost. The contents of 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

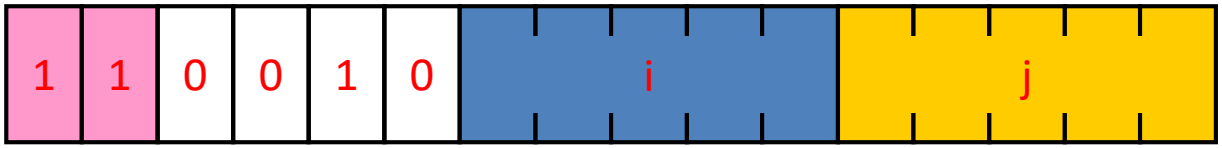




LDA i j

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:



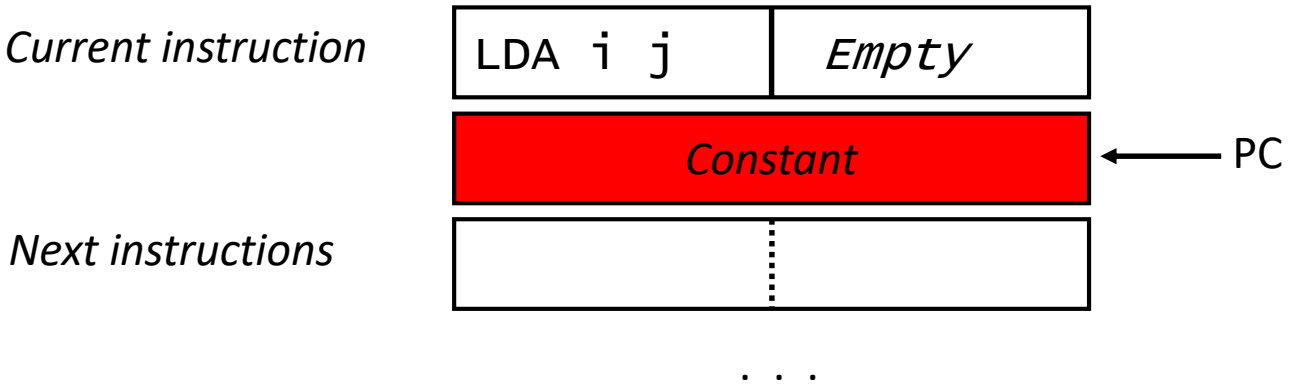
- The contents of the register Ri is considered as a 32-bit address of a 32-bit memory word.
- The next 32-bit word (pointed to by the PC register) is considered as an offset relatively to the “base” from the Ri register.
- Instruction format is always 32, i.e., the entire 32-bit word is copied from 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:

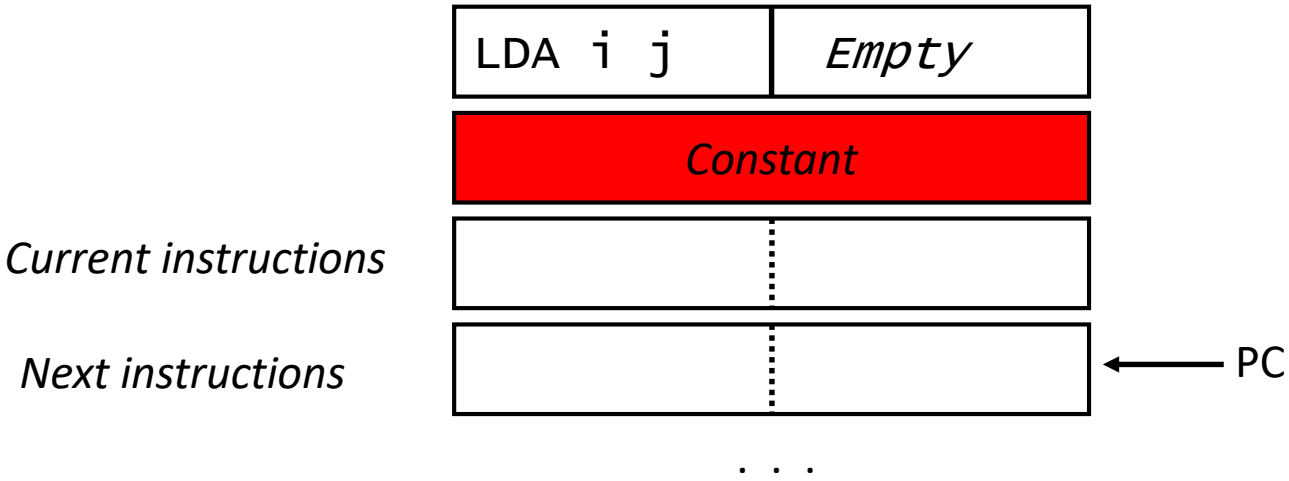
```
Rj := Ri + constant;
```

The scheme of how code is being processed is shown below

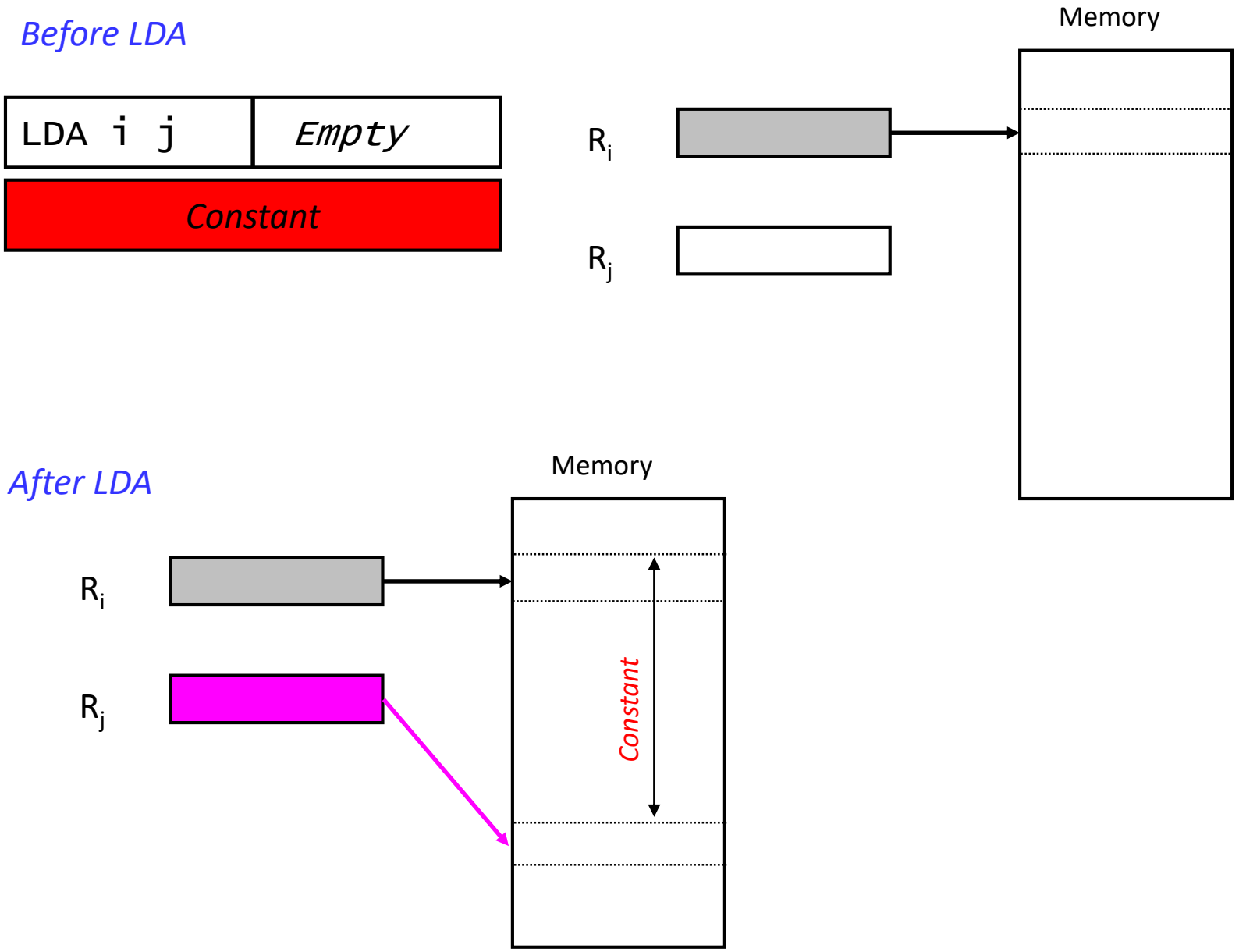
Before LDA



After LDA

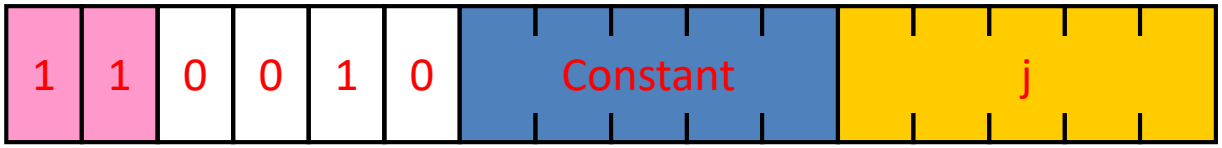


The effect of the LDA instruction is shown below



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.

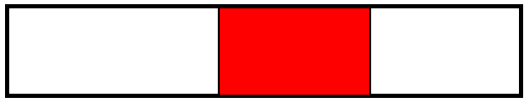
Suggested assembly statement for the LDC instruction:

```
Rj := Constant;
```

The effect of the LDC instruction is shown below

Before LDC

LDC instruction



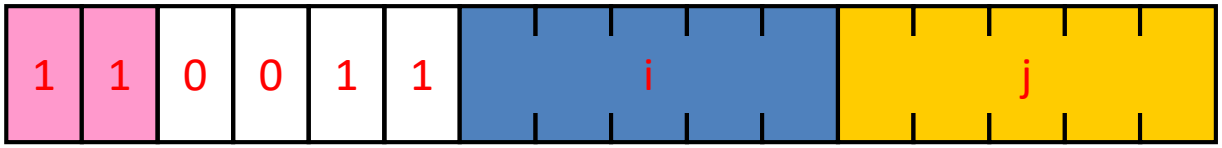
R_j

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:

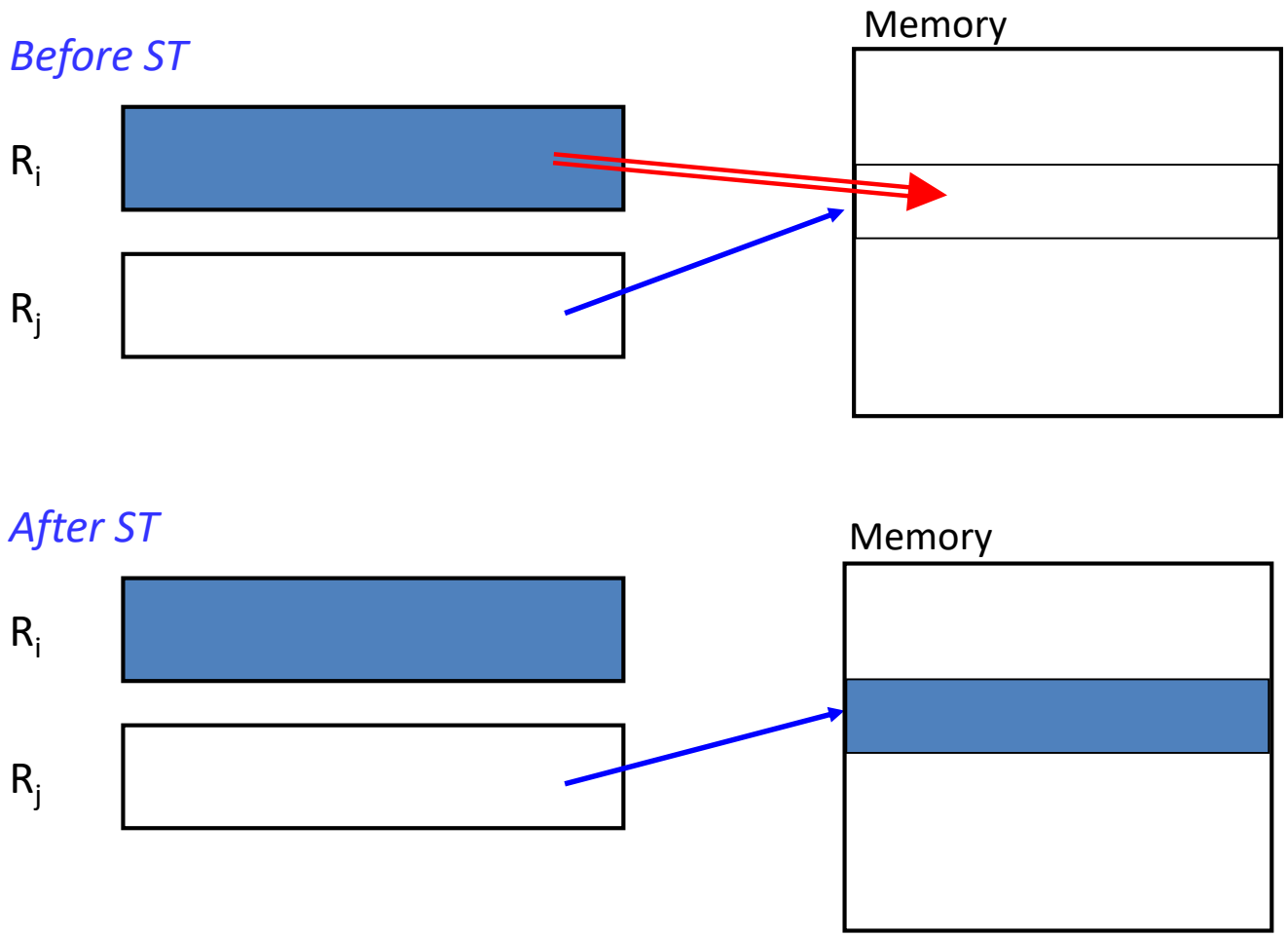


- The contents of the register Ri is treated as an arbitrary value. The contents of the register Rj is considered as a 32-bit address of a 32-bit memory word.
- Instruction format is always 32, i.e., the entire 32-bit register is copied to the memory.
- Memory state is not considered in the instruction,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

Suggested assembly statement for the LDC instruction:

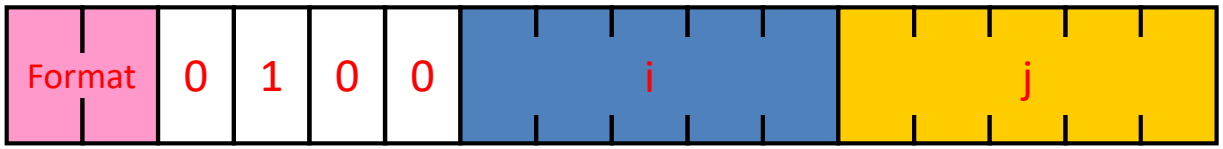
```
*Rj := Ri;
```

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:



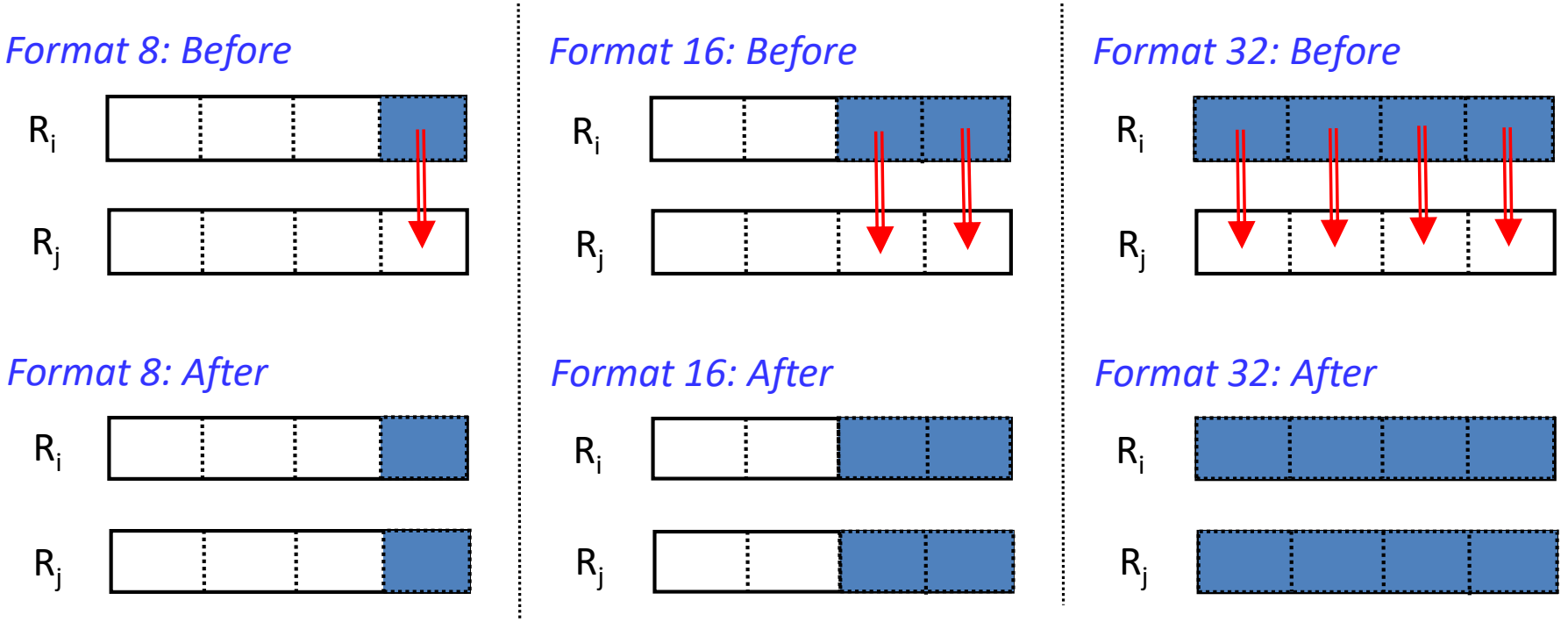
Suggested assembly statement for the MOV instruction:

```
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.

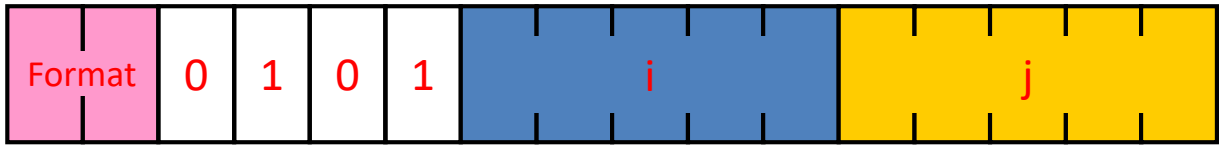
  Instruction format 16: two lowest bytes are copied; two highest bytes of Rj remain the same. The original value of Rj's two lowest bytes is lost.

  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:

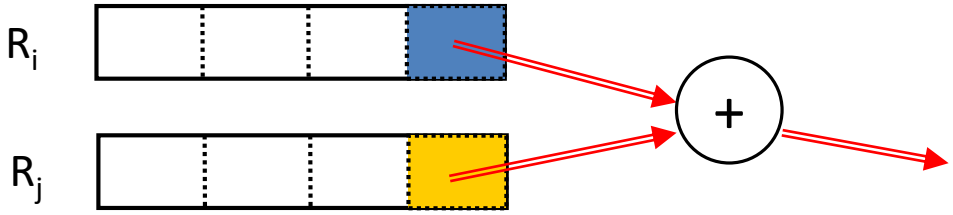
```
Rj += Ri;
```

Additional assembly directives specifying the current instruction format:

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

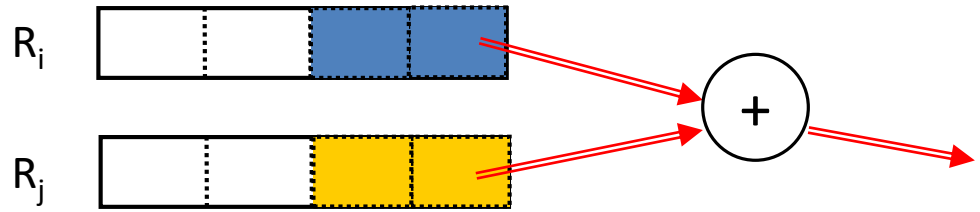
The effect of the ADD instruction is shown below

Format 8: Before



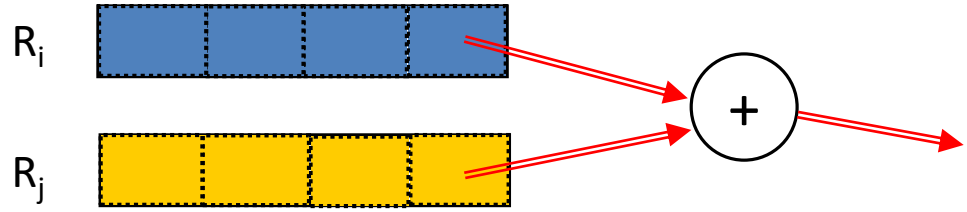
Format 8: After

Format 16: Before



Format 16: After

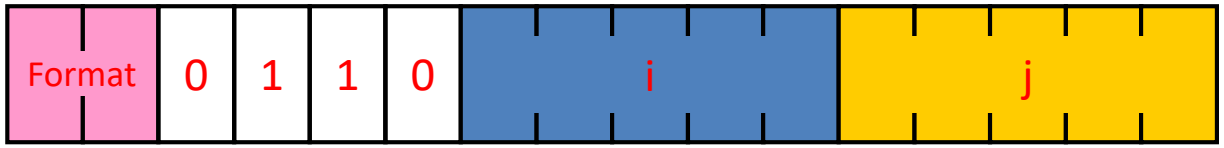
Format 32: Before



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:

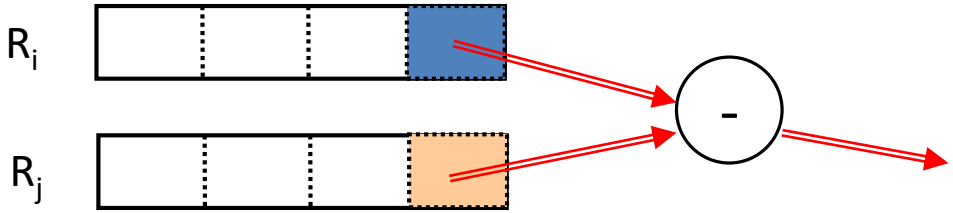
```
Rj -= Ri;
```

Additional assembly directives specifying the current instruction format:

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

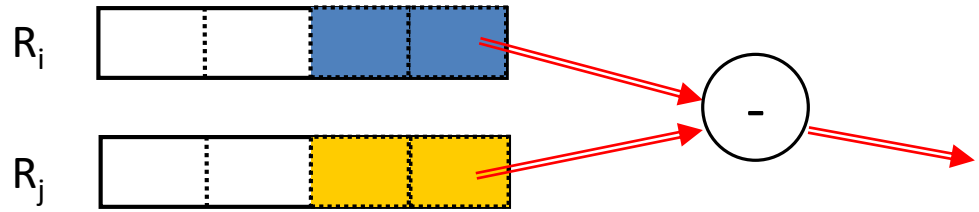
The effect of the SUB instruction is shown below

Format 8: Before



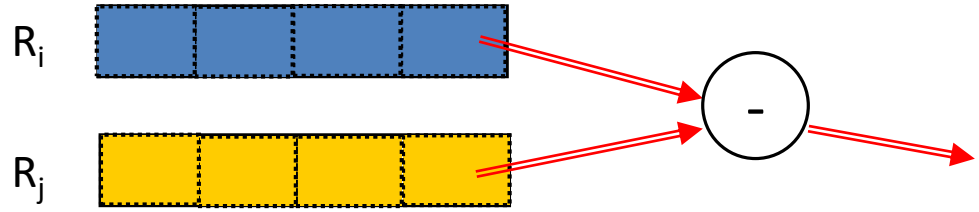
Format 8: After

Format 16: Before



Format 16: After

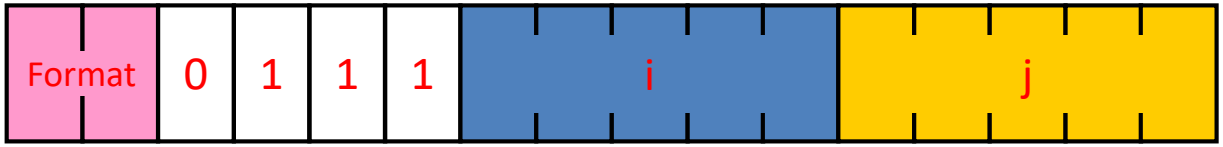
Format 32: Before



Format 32: After

ASR i j

- The ASR instruction arithmetically shifts the contents of the register Ri one bit right, 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.

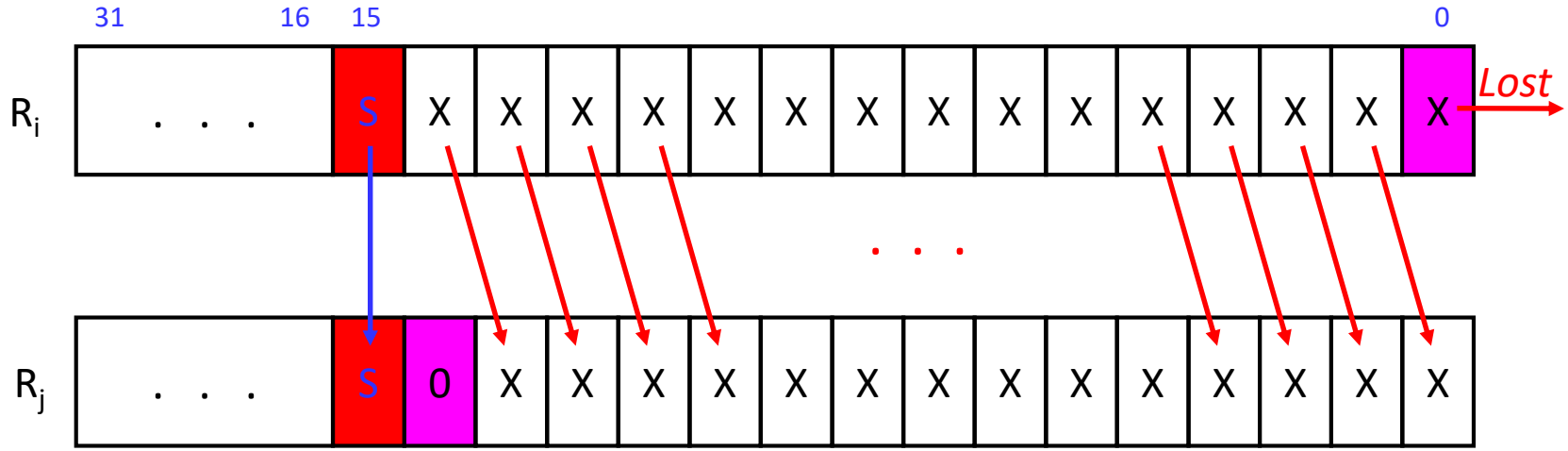
Suggested assembly statement for the ASR instruction:

```
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.

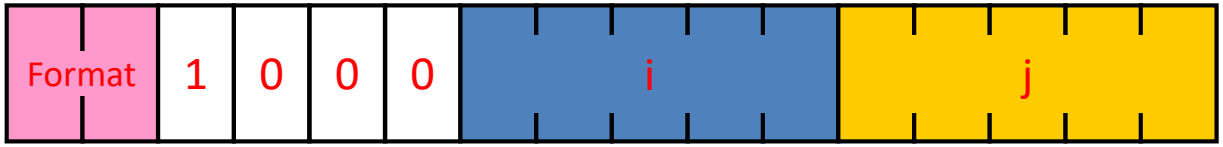


What to do with the high bits of the result for 8 and 16 formats?

- 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 one bit left, 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.

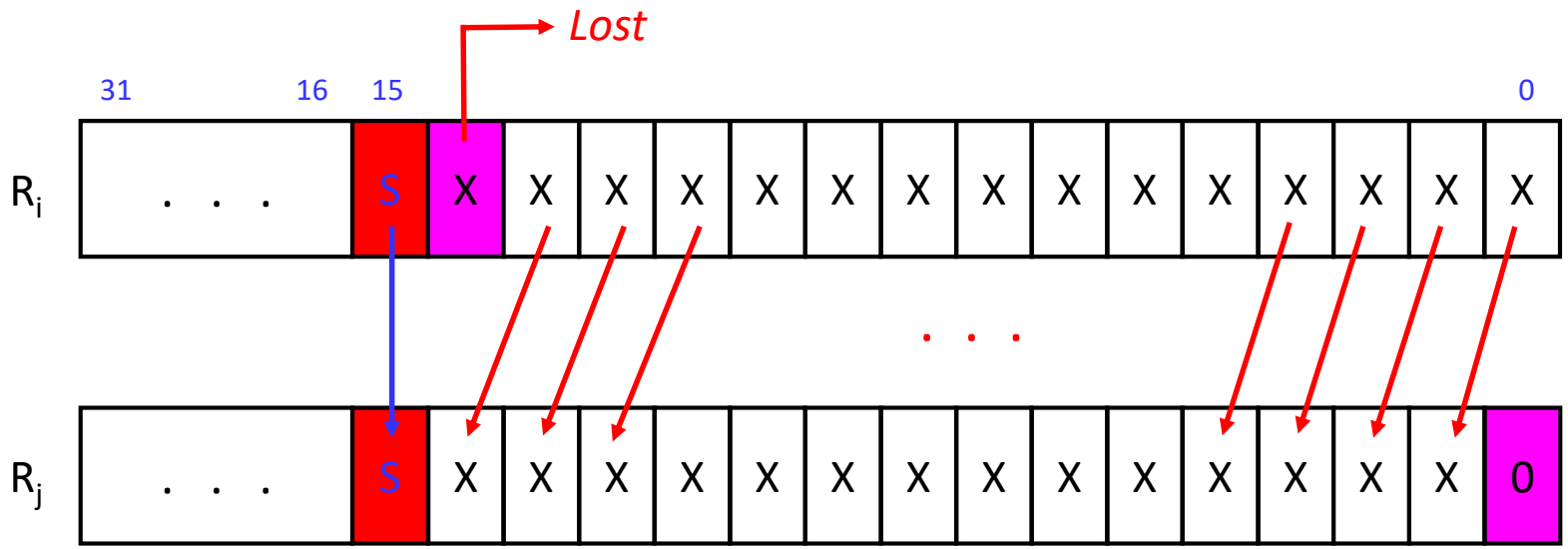
Suggested assembly statement for the ASL instruction:

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

Additional assembly directives specifying the current instruction format:

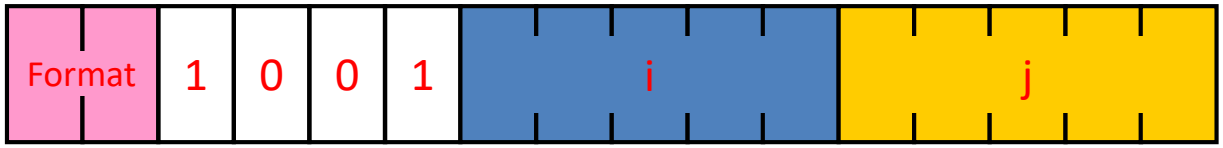
`.format 8;` or `.format 16;` or `.format 32;`

- *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;
```

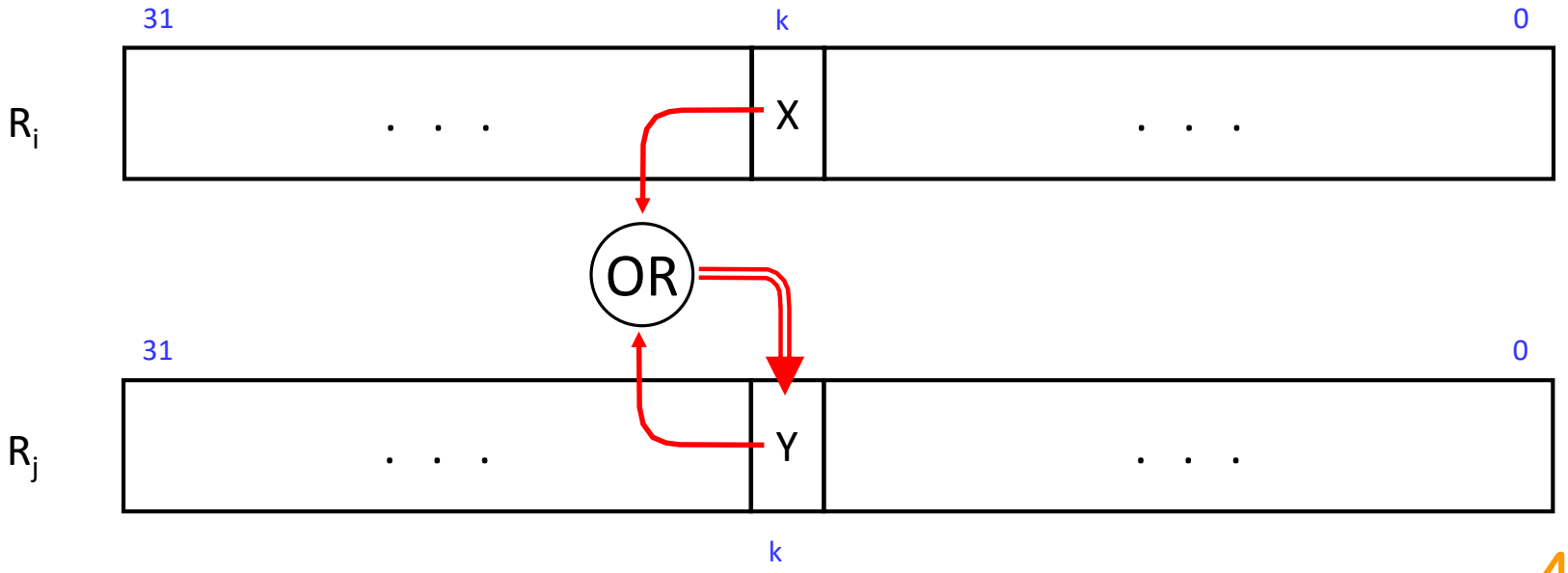
Additional assembly directives specifying the current instruction format:

`.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:

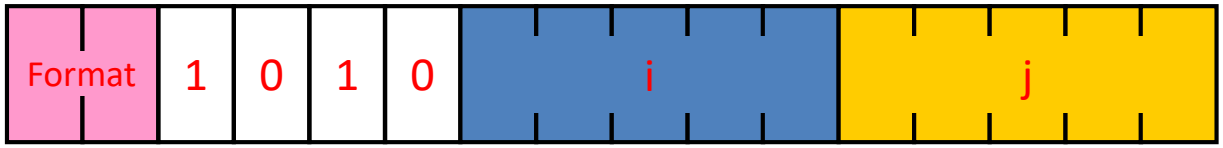
X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	1

- 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;
```

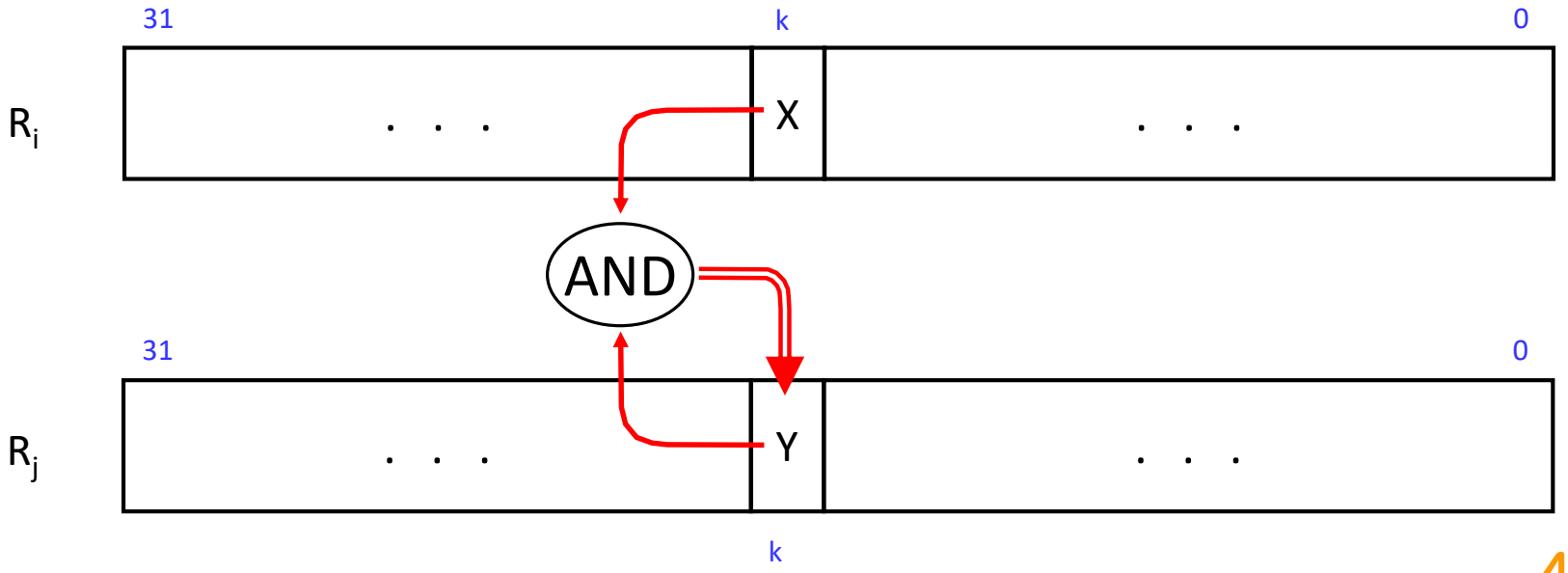
Additional assembly directives specifying the current instruction format:

`.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:

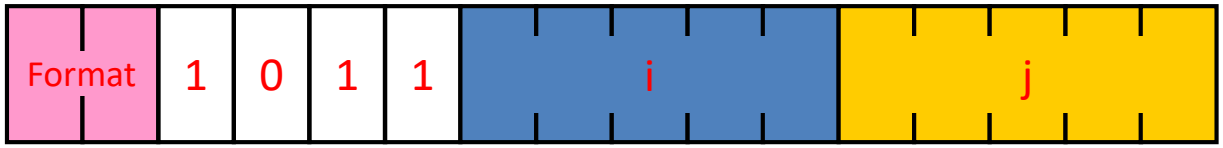
X	Y	Result
0	0	0
0	1	0
1	0	0
1	1	1

- 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;
```

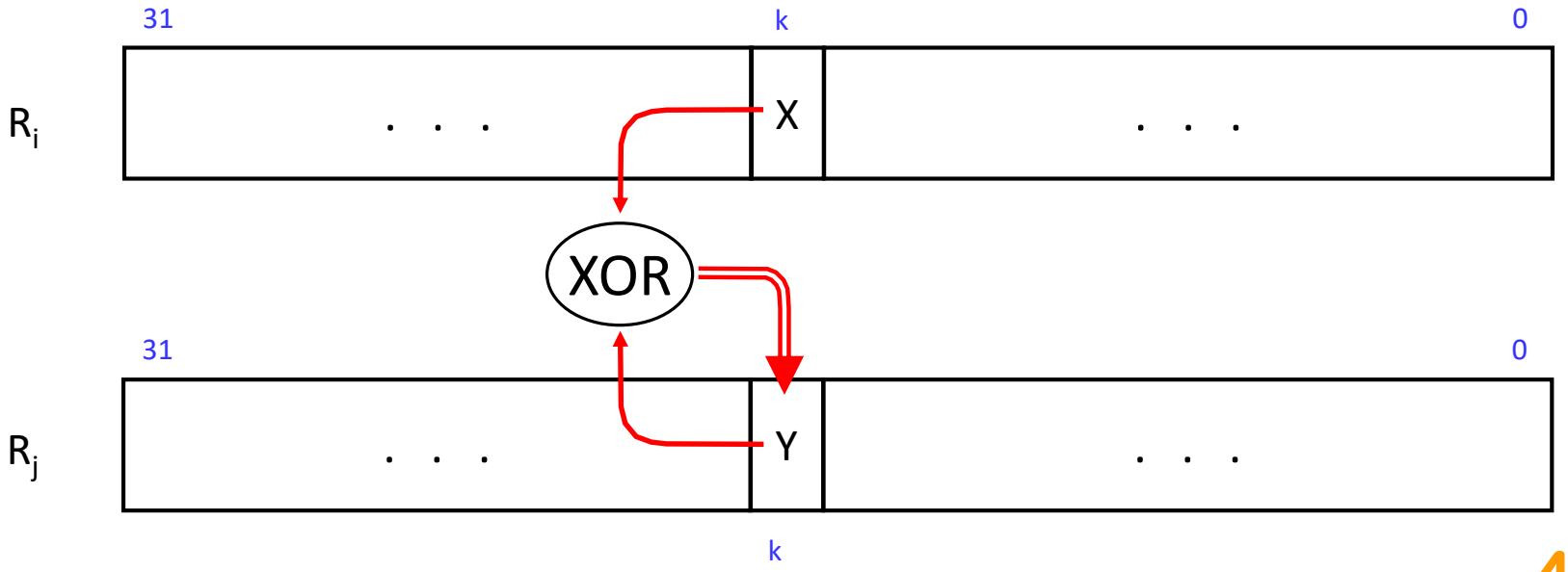
Additional assembly directives specifying the current instruction format:

`.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:

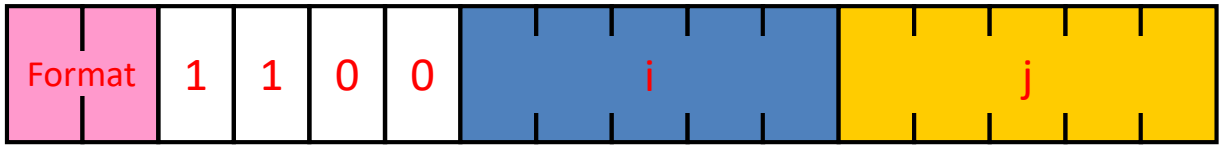
X	Y	Result
0	0	0
0	1	1
1	0	1
1	1	0

- 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 one bit left, 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.

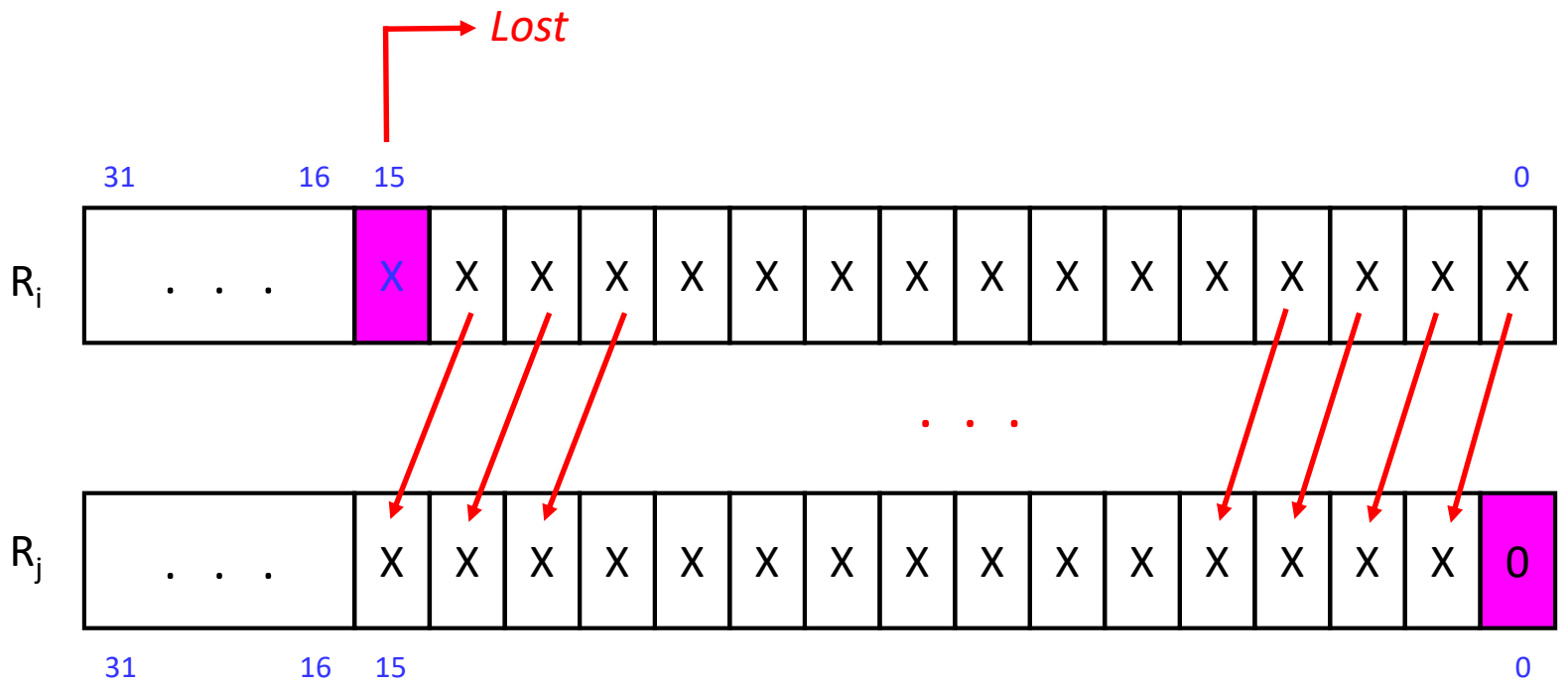
Suggested assembly statement for the AND instruction:

```
Rj <= Ri;
```

Additional assembly directives specifying the current instruction format:

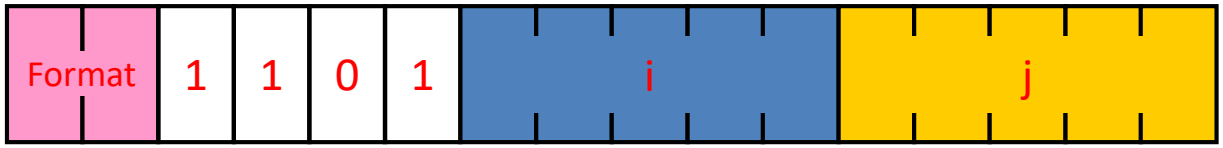
`.format 8;` or `.format 16;` or `.format 32;`

- 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 one bit right, 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.

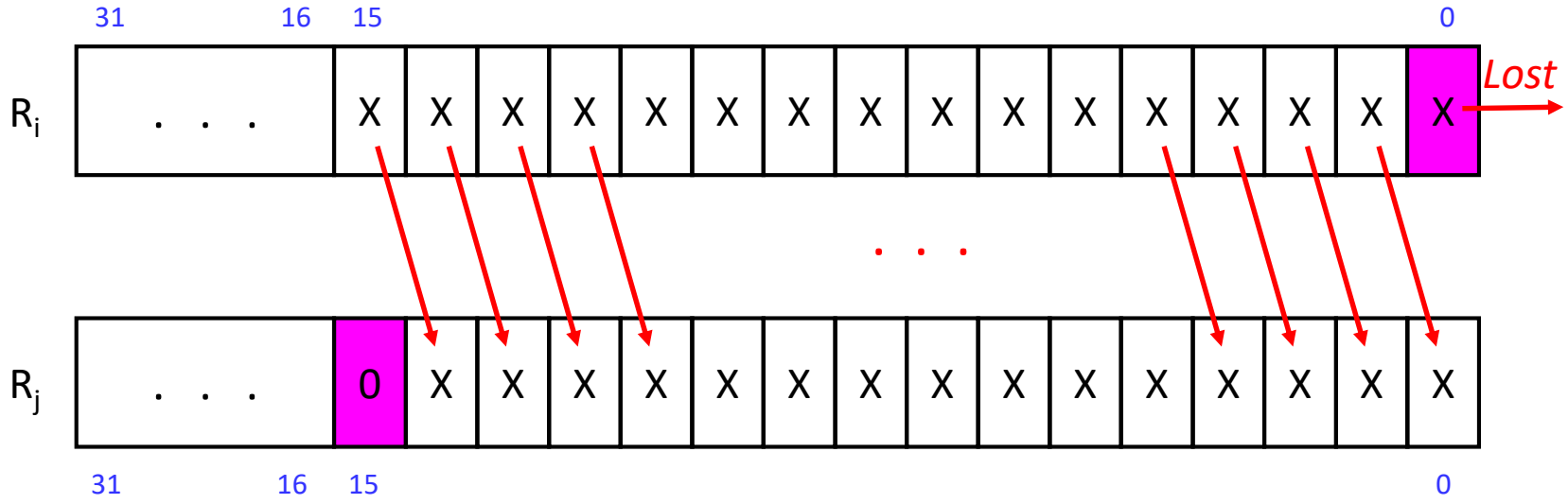
Suggested assembly statement for the AND instruction:

```
Rj >= Ri;
```

Additional assembly directives specifying the current instruction format:

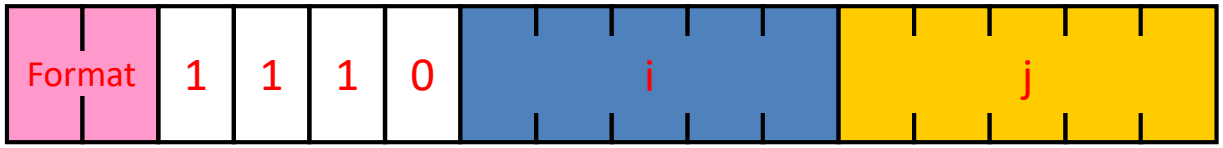
`.format 8;` or `.format 16;` or `.format 32;`

- 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:

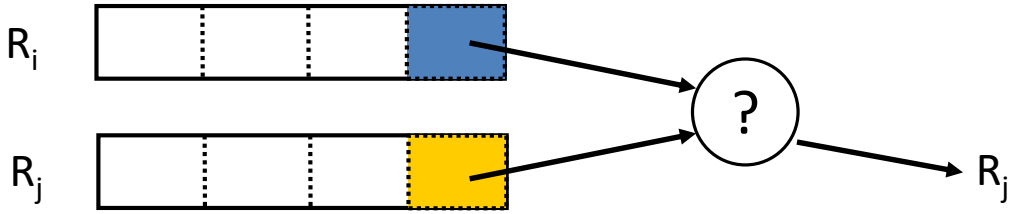
```
Rj ?= Ri;
```

Additional assembly directives specifying the current instruction format:

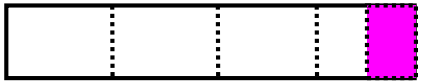
`.format 8;` or `.format 16;` or `.format 32;`

The effect of the CND instruction is shown below

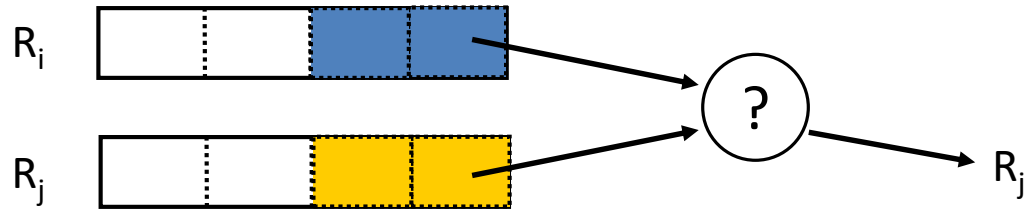
Format 8: Before



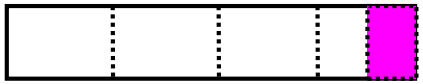
Format 8: After



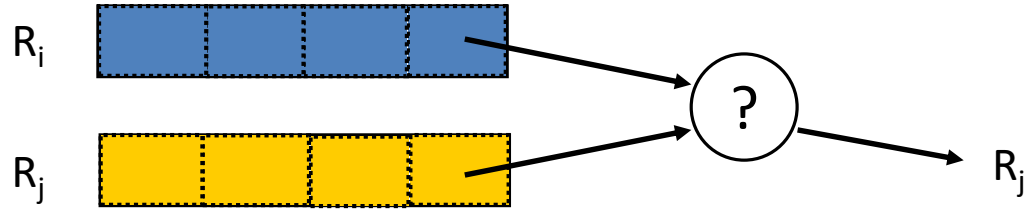
Format 16: Before



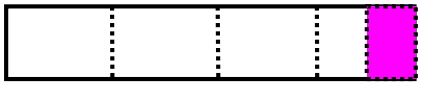
Format 16: After



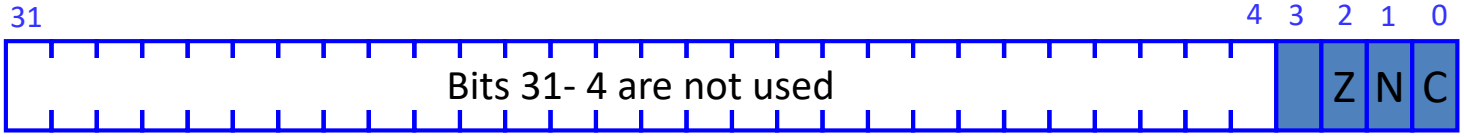
Format 32: Before



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.

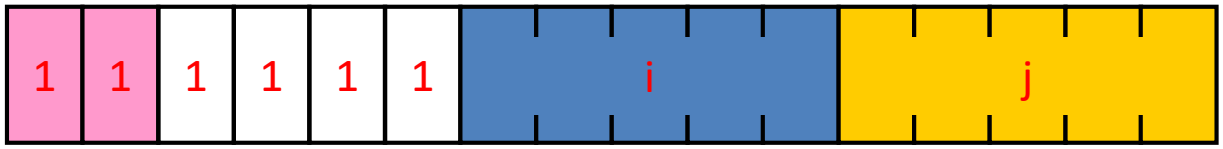
  For example, the $\leq$ condition can be treated as either $<$ or $=$ conditions, and the corresponding mask is binary 110. Similarly, the $\geq$ condition is either $>$ or $=$ conditions, and the mask is binary 101. Finally, the inequality $\neq$ condition is either $\leq$ or $\geq$ conditions, and the mask for selecting is binary 011.

  Therefore, all six possible comparison results ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) can be extracted using AND instruction with the following masks:

Relation	*Mask*
$<$	010
$\leq$	110
$>$	001
$\geq$	101
$=$	100
$\neq$	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:

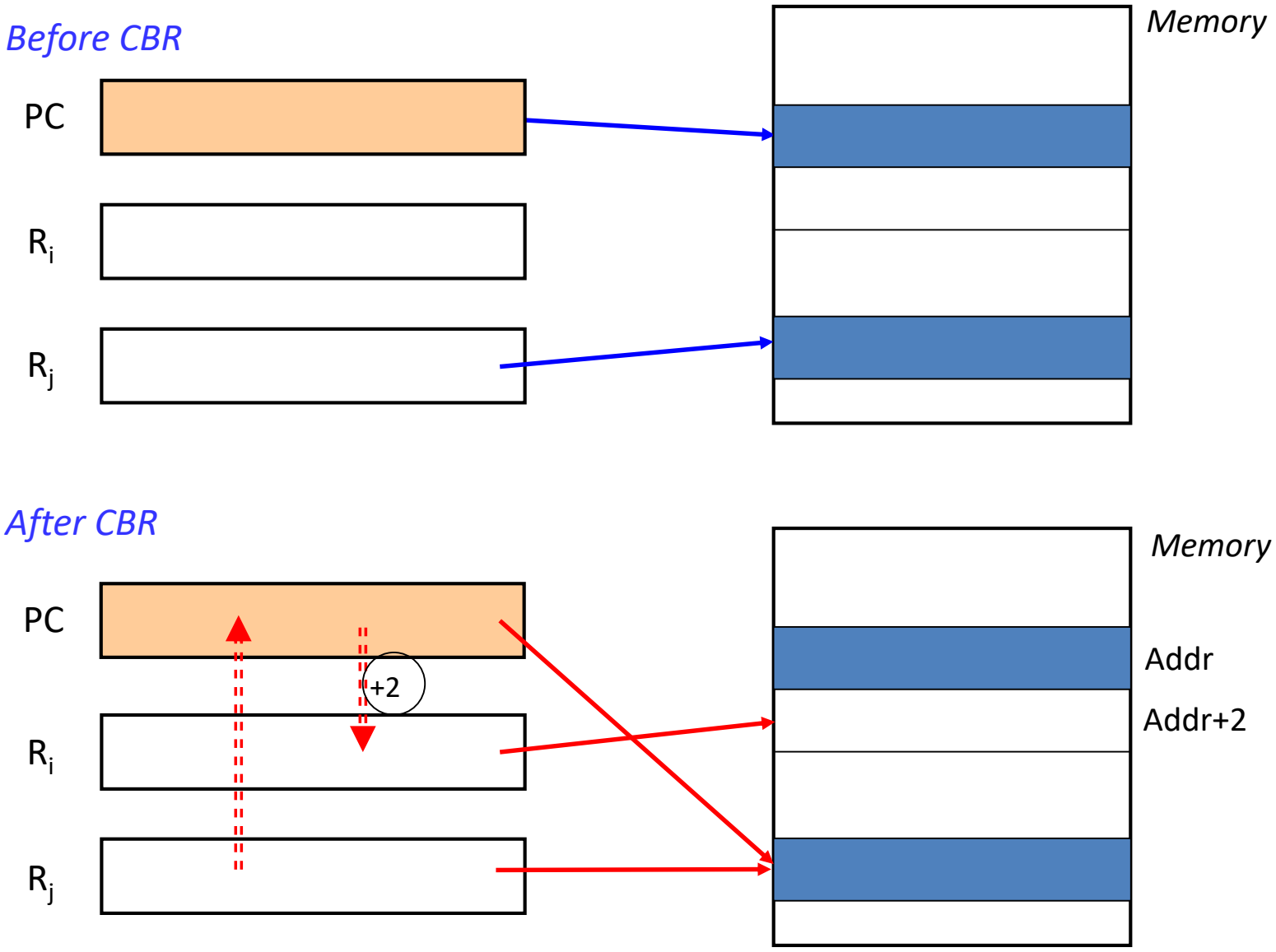


- Memory state is not considered in the instruction, and the memory state does not change.
- Format code does not affect the instruction's execution.
- The contents of the Rj register does not change.

Suggested assembly statement for the CBR instruction:

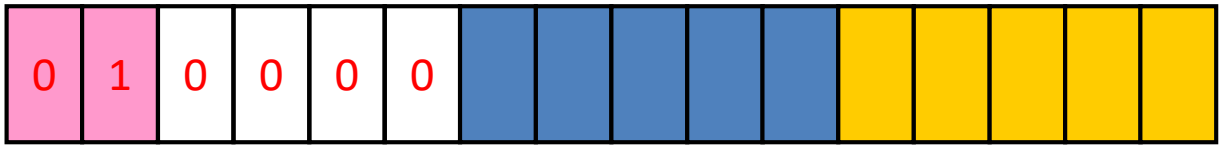
```
if Ri goto Rj;
```

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:

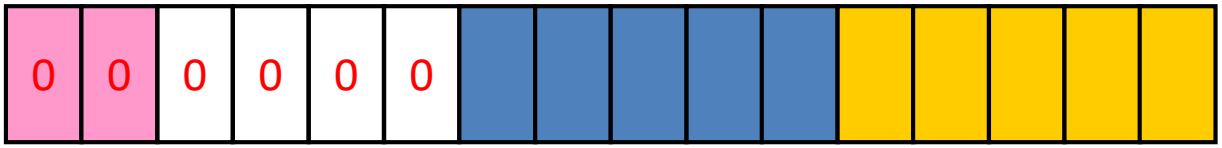


- Memory state is not considered in the instruction, and the memory state does not change.
- The value of the operand part of the instruction (bits from 9 to 0) does not affect the execution. If necessary it can be used as a place for keeping constants. In this case, the possible value range of the constant 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:

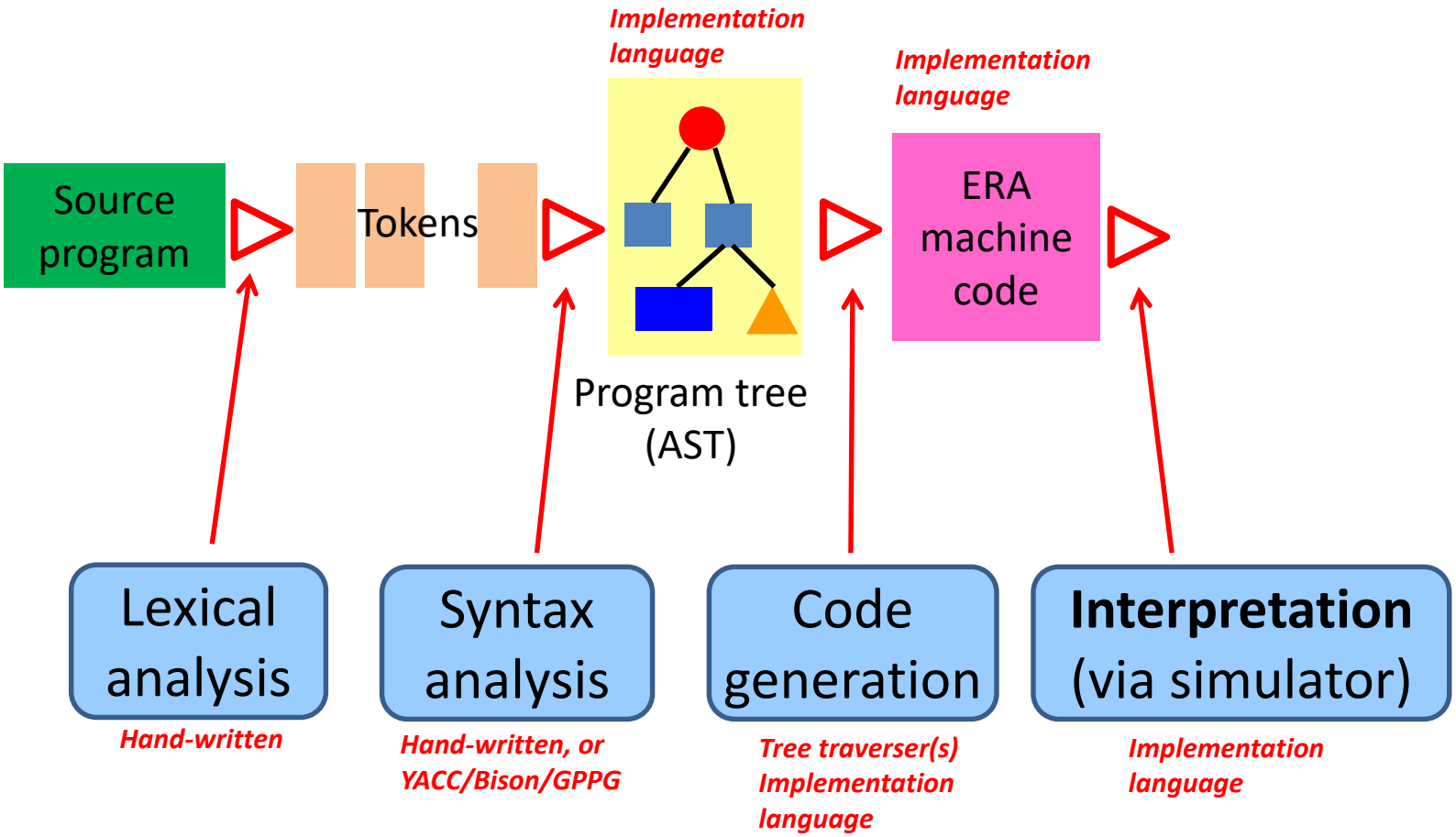


- Memory state is not considered in the instruction, and the memory state does not change.
- The value of the operand part of the instruction (bits from 9 to 0) does not affect the execution.

Suggested assembly statement for the STOP instruction:

```
stop
```

5. Project ERA: Implementation Model



6. Project ERA: Some comments for assembler & language implementation

Выражение

1) Операнд
2) Операнд Операция Операнд

Операция : + | - | * | & | '|' | ^ | ?

Операнд :	Reference	x
	Dereference	*x
	Address	&x
	Array Element	x[Операнд]
	Literal	7
	Register	R3
	Explicit Address	*Literal

x + 3

Значение, содержащееся в переменной x, увеличивается на **3**.

*x + 3

Значение по адресу, записанному в переменной x, увеличивается на 3.

&x + 3

Значение **адреса переменной x** увеличивается на 3.

label + 3

Адрес метки увеличивается на 3.

Простое присваивание

Получатель := Выражение

Получатель :	Reference	x := expr
	Dereference	*x := expr
	Array Element	x[Операнд] := expr
	Register	R3 := expr
	Explicit Address	*33 := expr

x := Что-то

Значение записывается в то место, которое занимает переменная x.

*x := Что-то

Значение записывается по адресу, который хранится в переменной x.

&x := Что-то

А вот в этом нет смысла: значение из правой части всегда помещается по адресу переменной x. Это, стало быть, в точности то же самое, что и x := Что-то.

&x	Берётся адрес переменной x
&label	Неверно: у меток нет адреса
&fun	Бессмысленно: функция рассматривается как адрес начала её кода
*x	Берётся/записывается значение по адресу, взятому из переменной x
*label	То же, что и *x
*fun	То же, что и *x, но только «по чтению»
*число	Берётся/записывается значение по адресу, заданному явно.

fcall fun

(«Быстрый» вызов). Прямая реализация перехода с возвратом – без стека, непосредственно командой if R1 goto R2.

Применяется для реализации простых функций без (прямой или косвенной) рекурсии. Параметры и локальные переменные функции (если они есть) располагаются непосредственно в коде. Они адресуются относительно начала кода функции (смещение относительно начала кода).

call fun

Вызов общего характера с использованием кадров стека, со статической и динамической цепочками и запоминанием адреса возврата в стеке. Параметры и локальные переменные функции адресуются относительно «базы» – адреса начала текущего кадра стека в фиксированном регистре (R31).

Может использоваться для реализации как обычного процедурного механизма языка Си (без вложенных функций), так и более сложных механизмов (например, для реализации вложенных функций).

Проблемы адресации

Проблема заключается в том, что работа с метками (как и система адресации в целом) сильно зависит от способа расположения кода и данных в памяти.

Если код располагается по абсолютным (физическим) адресам, то метка представляет собой константу (конкретный физический адрес) и потому может быть непосредственно представлена числом. То же относится к случаю адресации локальной переменной «быстрой» процедуры: в данном случае адрес также представляется непосредственной константой.

При адресации локальной переменной обычной процедуры используются два значения, которые не могут быть вычислены при компиляции: адрес «базы» (обычно содержится в некотором фиксированном регистре) и смещение адресуемого объекта или команды относительно базы (смещение известно при компиляции). Сложение базы из регистра и константы-смещения и даёт нужный адрес.

Наиболее сложный случай – адресация внутри кода (попросту говоря, переходы по меткам, включая вызовы процедур). В данном случае также возможны два способа адресации: непосредственный, по конкретным адресам, когда расположение кода известно заранее, и относительный, когда адрес начала кода находится в некотором регистре, а смещение внутри кода вычисляется статически, при компиляции.

Проблема с адресацией по коду заключается в том, что ни база (адрес начала кода), ни смещение внутри кода заранее (при компиляции) не известны. Более того, они не известны вплоть до окончания генерации финального кода. С другой стороны, реализация переходов для абсолютного и относительного вариантов предполагает использование различных последовательностей команд и тем самым влияет на размер смещение внутри кода.

Получается, что две проблемы мешают друг другу... «Развязать» ситуацию придётся путём разделения процесса генерации на два прохода и введение псевдокоманды «загрузки метки».....

```
==========================================
int32 a = ...;
...
goto a;  // допустимо; подразумевается, что в a лежит некоторый адрес

goto &a; // так тоже можно (это ассемблер, детка ☺), хотя и глупо:
          // переход по адресу, по которому лежит переменная a.
          // То есть, попытка «выполнения» числа, как если бы это была
          // некоторая команда.
==========================================
label L;
...
a:= &L; // адрес метки - ошибка
==========================================
data Messages is
   <msg1> “this is a message”;
   <msg2> “this another message”;
end Messages;
...
a:= Messages.msg1[7];     // OK
Messages.msg2[3] := “f”;  // Тоже можно (почему нет?)
==========================================
```

Две псевдокоманды:

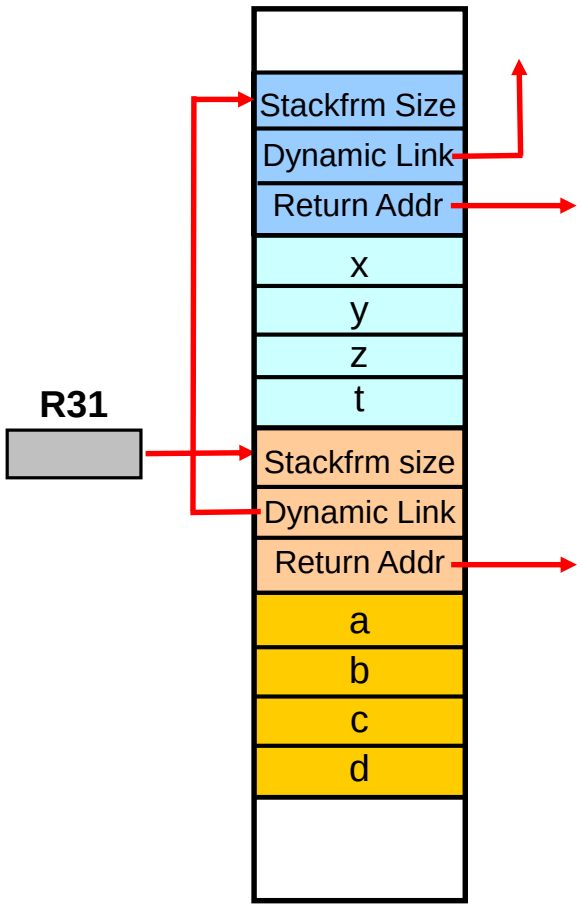
Load Address
LDA, регистр, Указатель на Unit, offset

Если указатель на Unit есть null, то на **регистр** загружается адрес локальной переменной (база + offset, «база» определяется по типу текущего unit'а). Иначе – адрес переменной из другого модуля.

Load Label
LDL, регистр, Указатель на Unit, указатель на метку

На **регистр** загружается адрес, взятый из определения метки. Метка берётся либо из модуля по адресу Unit'а, либо из текущего модуля.
Псевдокоманды появляются в коде после парсинга; в них компилируются все обращения к переменным и меткам. После того, как оттранслирована вся программа (и, соответственно, становятся известными все адреса, прежде всего, адреса начала кодовых сегментов всех единиц компиляции), псевдокоманды заменяются на «нативные» команды загрузки адресов, сложения их со смещениями и записи полученных физических адресов в заданный регистр.

```
call f(a,b,c) using R0,R1;
...
return using R0;
```



Алгоритм вызова функции f

```
R0 := R31;    // Взяли адрес базы
R31 += size;  // Передвинули базу на новый stackframe
*R31 := size_new; // Положили в stackframe размер
R1 := 1;
R1 += R31; // R1 смотрит на след. слово в stackframe
*R1 := R0; // дин.цепочка (адрес пред. stackframe)
<загрузка аргументов вызова, начиная со смещения 3>
if R1 goto f; // в R1 сохранился адрес возврата
```

Стандартный пролог функции

```
R0 := 2;
R0 += R31;  // R2 смотрит на третье слово stackframe
*R0 := R1;  // Сохранили в 3-ем слове адрес возврата
...
<Код тела функции>
...
```

Стандартный эпилог функции

```
R0 := 2;
R0 += R31; // R0 смотрит на третье слово stackframe
*R31 = R31;// Вернули адрес базы в положение до вызова
*R0 := R0; // Взяли в R2 адрес возврата из функции
if R0 goto R0; // Возврат на след.команду после вызова
```

a > b (a и b – локальные)

```
R0 := offset-a;
R0 += R31;  // R0 смотрит на место a в стеке
R0 := *R0;  // В R0 – значение a

R1 := offset-b;
R1 += R31;  // R1 смотрит на место b в стеке
R1 := *R1;  // В R1  - значение b

R0 ?= R1;   // Сравнили a и b
R1 : = 1;   // Маска сравнения на «больше»
R0 &= R1;   // Выделили из результата бит «больше»;
            // в R0 лежит 1, если a>b,
            // и 0 в противном случае
```

X := a > b (a и b – локальные)

```
...
R2 := offset-x;
R2 += R31;  // R2 смотрит на место x в стеке
*R2 := R0;  // В R0 – значение результата сравнения
```

(Если a, b или x – глобальные, то вместо R31 используется R30.)

```
if a > b then
    Statements1
else
    Statements2
end;
```

```
<Вычисление условия; инвертирование результата.
Пусть инвертированный результат находится в R0.>

R1 := L1;       // В R1 – адрес метки L1
if R0 goto R1; // Если условие ложно, переход на L1
<Statements1>
R1 := L2;
if R1 goto R1; // Безусловный переход на L2
<L1>
<Statements2>
<L2>
```

```
while a > b loop
    Statements
end;
```

```
<L0>
<Вычисление условия; инвертирование результата.
Пусть инвертированный результат находится в R0.>
R1 := L1;       // В R1 – адрес метки L1
if R0 goto R1; // Если условие ложно, выход из цикла
<Statements>
R1 := L0;
if R1 goto R1; // Безусловный переход на начало цикла
<L1>
```

```
for R0 := C1 to C2 step C3 loop
    Statements
end;
```

```
R0 := C1;
<L>
Statements
R1 := C3;
R0 += R1;
R1 := C2;
R0 ?= R1;  // Сравнение текущего значения R0 и C2
R1 := 6;   // Маска для «меньше или равно»
R0 &= R1;
R1 := L;
if R0 goto R1;  // если R0<=C2, то идём на след. итерацию
```

swap a, b; **a <=> b**

```
R0 := offset-a;
R0 += R31;   // В R0 – адрес a
R2 := *R0;   // Загрузили a в R2

R1 := offset-b;
R1 += R31;   // В R1 – адрес b
R3 := *R1;   // Загрузили b в R3

*R0 := R3;   // По адресу a помещаем значение b
*R1 := R2;   // По адресу b помещаем значение a
```

R1 <=> R2 using R3;

```
R3 := R1;
R1 := R2;
R2 := R3;
```

7. Appendix: The Code Example

Source code:

```
routine Sort ( int[] a, int size )
do
   for i := 1 to size loop
      for j := i to 0 by -1 loop
         if a[j-1] < a[j]
         then
            w := a[j-1];
            a[j-1] := a[j];
            a[j] := w;
         end
      end
   end
end
```

Assembler code for sort 1/3

```
 // i: R1;  j: R2;  intermediate values: R3, R4, R5

    R1 := 1;          // i := 1
<LoopOuter>
    R3 := Size;
    R3 := *R3;        // R3: Size
    R4 := 1;
    R3 -= R4;
    R3 ?= R1;         // Compare Size-1 and i
    R4 &= R3;         // Extract > sign using R4=1 as mask for >
    R3 := OutOuter;
    if R4 goto R3;    // if i>Size goto OutOuter

    // Organize inner loop
    R2 := R1; NOP;    // j := i
<LoopInner>
    R3 := 0;          // w := 0
    R3 ?= R2;         // Compare j with 0
    R4 := 4;          // Mask for equality
    R3 &= R4;         // Extract equality sign
    R4 := OutInner;
    if R3 goto R4;    // if j=0 exit the inner loop

    // Otherwise, compare two array elements
    // to decide if we need to exchange them.
    // R10: address of j-th element
    // R11: a[j]
    // R12: a[j-1]
```

Ext 2

Assembler code for sort, page 2/3

```
    R10 := Array;
    R10 += R2;       // array base address+j
    R11 := *R10;     // R11 := a[j]
    R12 := R10;
    R13 := 1;
    R12 -= R13;      // a+j-1
    R12 := *R12;     // R12 := a[j-1]

    R3  := R12;      // w := a[j-1]
    R3  ?= R11;      // Compare a[j-1] and a[j]
    R4  := 5;        // Mask for >=
    R4  &= R3;       // Extract > and = signs
    R3  := OutExchange;
    if R4 goto R3;  // if a[j-1] >= a[j] do not perform exchange

    // Otherwise, perform exchange
    R3 := R10;
    R4 := 1;
    R3 -= R4;        // R5: address of (j-1)th element
   *R3 := R11;       // a[j-1] := a[j]
   *R10:= R12;       // a[j] := a[j-1]
<OutExchange>
    // Decreasing j (inner loop)
    R3 := 1;
    R2 -= R3;        // j := j-1
    R4 := LoopInner;
    if R3 goto R4;  // goto LoopInner
```

Ext 3

Assembler code for sort, page 3/3

```
<OutInner>
    // Increasing i (outer loop)
    R3 := 1;
    R1 += R3;       // i := i+1
    R4 := LoopOuter;
    if R3 goto R4;  // goto LoopOuter
    NOP;

<OutOuter>
    R15 := Size;
    R15 := *R15;
    R16 := Array;
    TRACE R15,R16;
    STOP; NOP;
<Size>
    DATA  20
<Array>
    DATA 537, 242, 114, 436, 337, 296, 285, 655, 639, 436
    DATA 912, 520, 624, 551, 600, 741, 612, 943, 871, 735
```