



Debugging - Debugger architectures and infrastructures, breakpoints implementation

Steven Costiou

steven.costiou@inria.fr

RMoD / Inria Lille - Nord Europe

September 2021

Summary

1. An overview of debugger architectures
2. Software breakpoints
3. Hardware breakpoints
4. An introduction to Java Debug Interface

Debuggers architectures overview

1

2

3

4

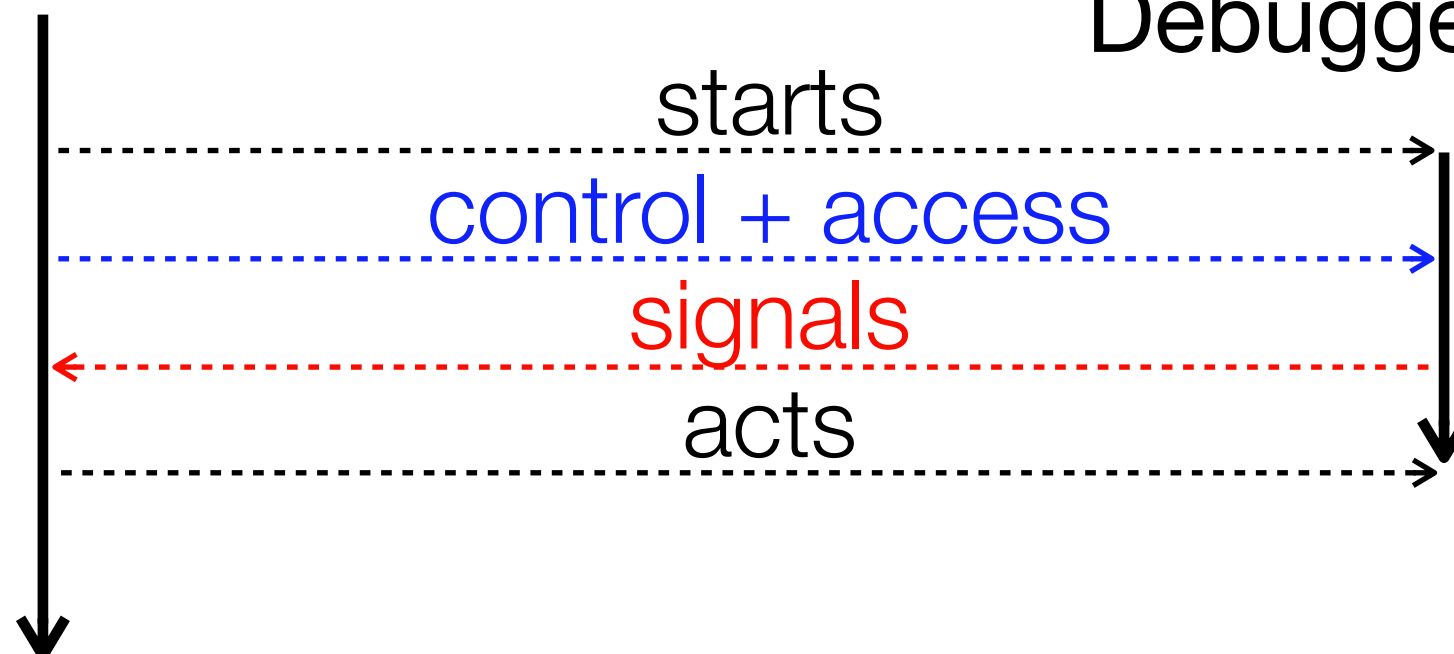
Debugging architecture - general concepts (1)

◆ Simplified overview

- ▶ The debugger process starts and stops the debuggee process
- ▶ The debugger applies control operators to the debuggee process
 - ▶ breakpoints, stepping, memory inspection...
- ▶ The debuggee process (in reality, the operating system) sends signals (e.g., interruptions) to the debugger, that can react upon reception of those events

Debugger process

Debuggee process

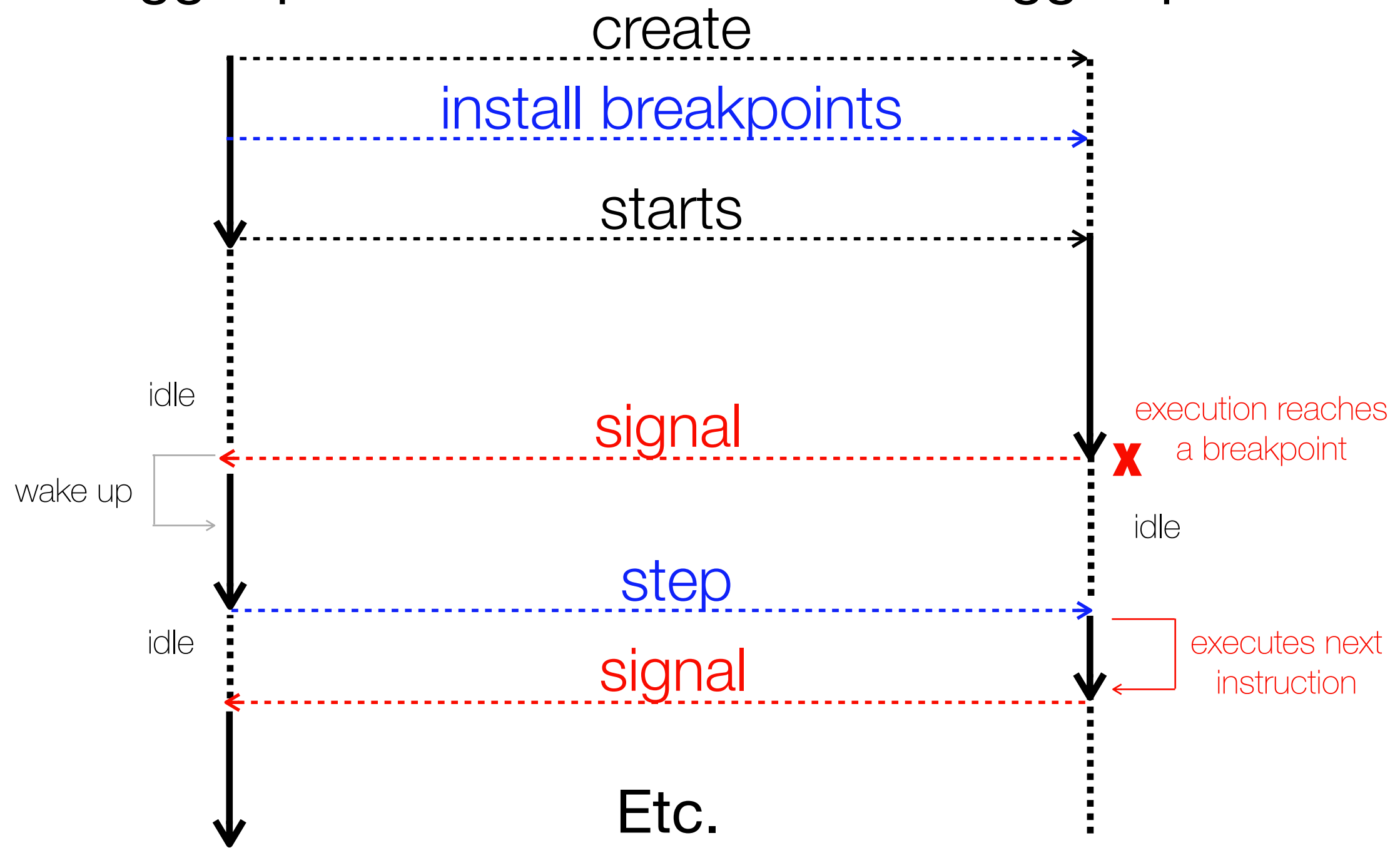


Debugging architecture - general concepts (2)

◆ A more detailed example

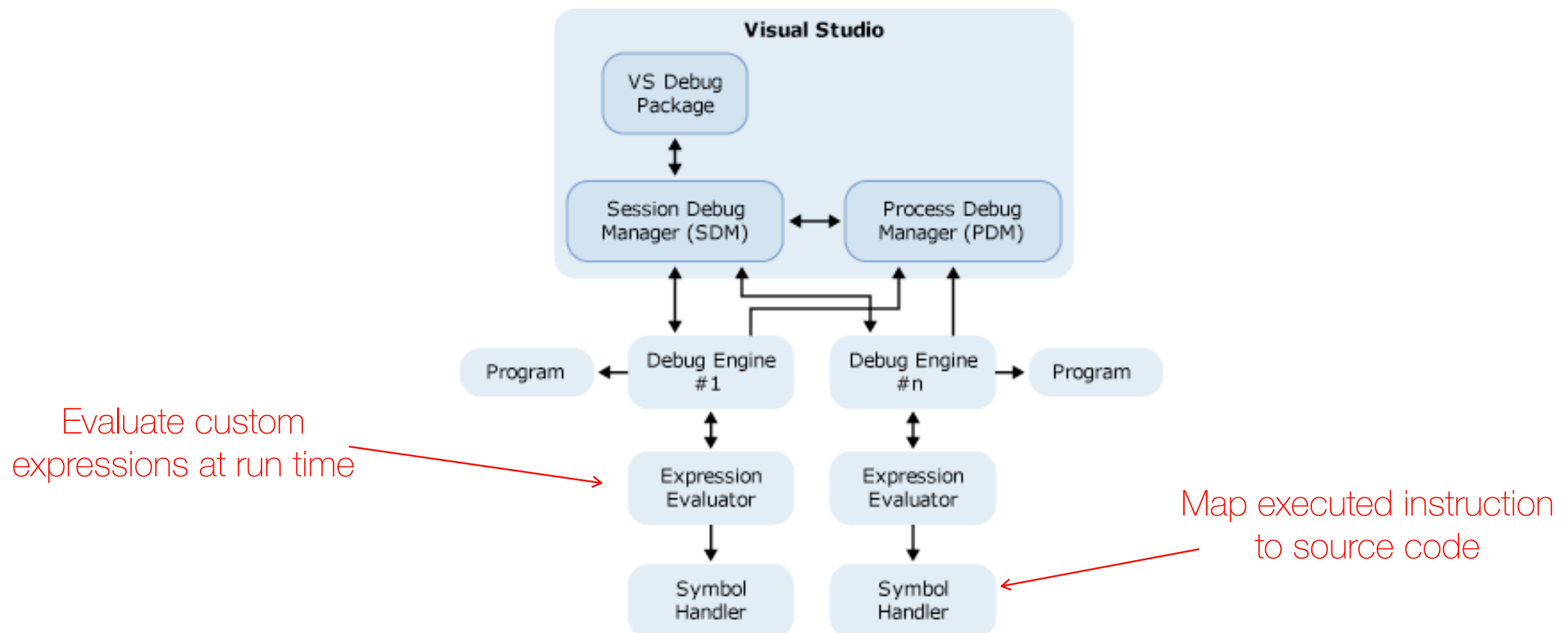
Debugger process

Debuggee process



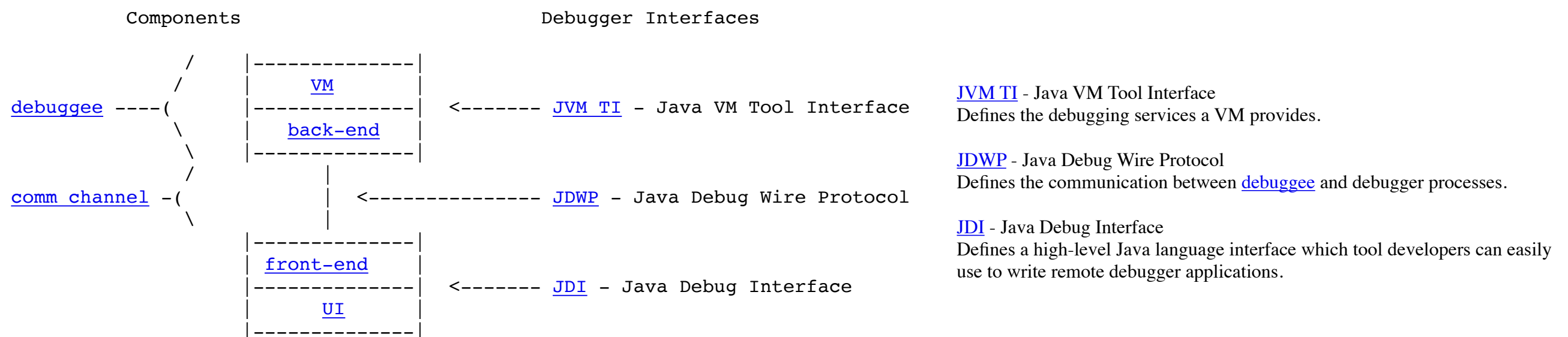
Example: the Visual Studio debugger architecture

- ◆ The VS IDE provides interfaces to plug debug engines
- ◆ Debug engines provide run-time support for different languages
 - ▶ They attach to a run time (*i.e.*, a running program)
 - ▶ They provide an expression evaluator for the supported language
 - ▶ They provide a symbol handler to map debugging symbols to the running program



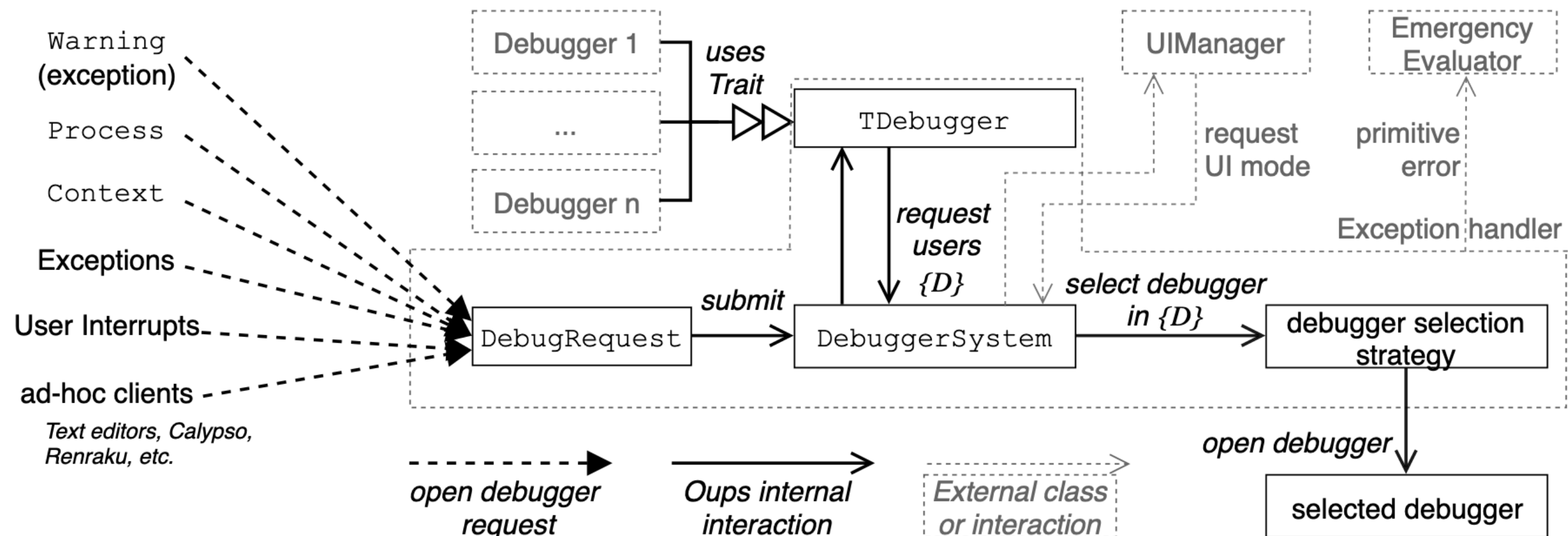
Example: the JVM debugging architecture

- ◆ Java programs are executed by **a virtual machine (JVM) that implements the JVM TI interface**
- ◆ **A back-end handles communication between the VM and the debugger front-end**
 - ▶ Communication with the VM is made through the JVM TI interface
 - ▶ Communication with the debugger is made through the JDWP protocol
- ◆ **Tools are extendable** from any of these layers:
 - ▶ New back-ends can be directly plug to the VM via JVM TI
 - ▶ New front-ends can be interfaced with a back-end the implements the JDWP protocol
- ◆ A front-end (JDI) allows to build custom java debugging tools directly in java



Example: the Pharo debugger infrastructure

- ◆ All signals go through a single entry point in the system
- ◆ The debugger system find all available debuggers in the runtime and a dedicated debugger strategy choses which debugger to use
 - ▶ The debugger selection strategy is customisable, extendable to provide fine-grained debugger selection
 - ▶ New debuggers can be added to the runtime and take over specific kind of signals



Software breakpoints

1

2

3

4

About interrupts (1)

◆ **What is an interrupt?**

- ▶ A signal to the processors
- ▶ This signal means something needs immediate attention from the processor

About interrupts (1)

◆ **What is an interrupt?**

- ▶ A signal to the processors
- ▶ This signal means something needs immediate attention from the processor

◆ **What happens when an interrupt happens?**

- ▶ The processor stops its current activities
- ▶ The processor then executes an interrupt handler
- ▶ That handler «*handles*» the event at the origin of the interrupt

About interrupts (2)

◆ **Hardware interrupts**

- ▶ Electronic signal from an external device
- ▶ They can be asynchronous

◆ **Software interrupts**

- ▶ Triggered by the processor when executing particular instructions
- ▶ Sometimes called «traps» or «exceptions» (e.g., breakpoints or divide-by-zero exceptions)

What is a software breakpoint?

◆ **Instrumentation of the runtime**

- ▶ Special instructions are inserted into the binary (or byte code for managed runtimes)
- ▶ These special instructions raise an interrupt signal when they are hit at run time

What is a software breakpoint?

◆ Instrumentation of the runtime

- ▶ Special instructions are inserted into the binary (or byte code for managed runtimes)
- ▶ These special instructions raise an interrupt signal when they are hit at run time

◆ Examples of special instructions

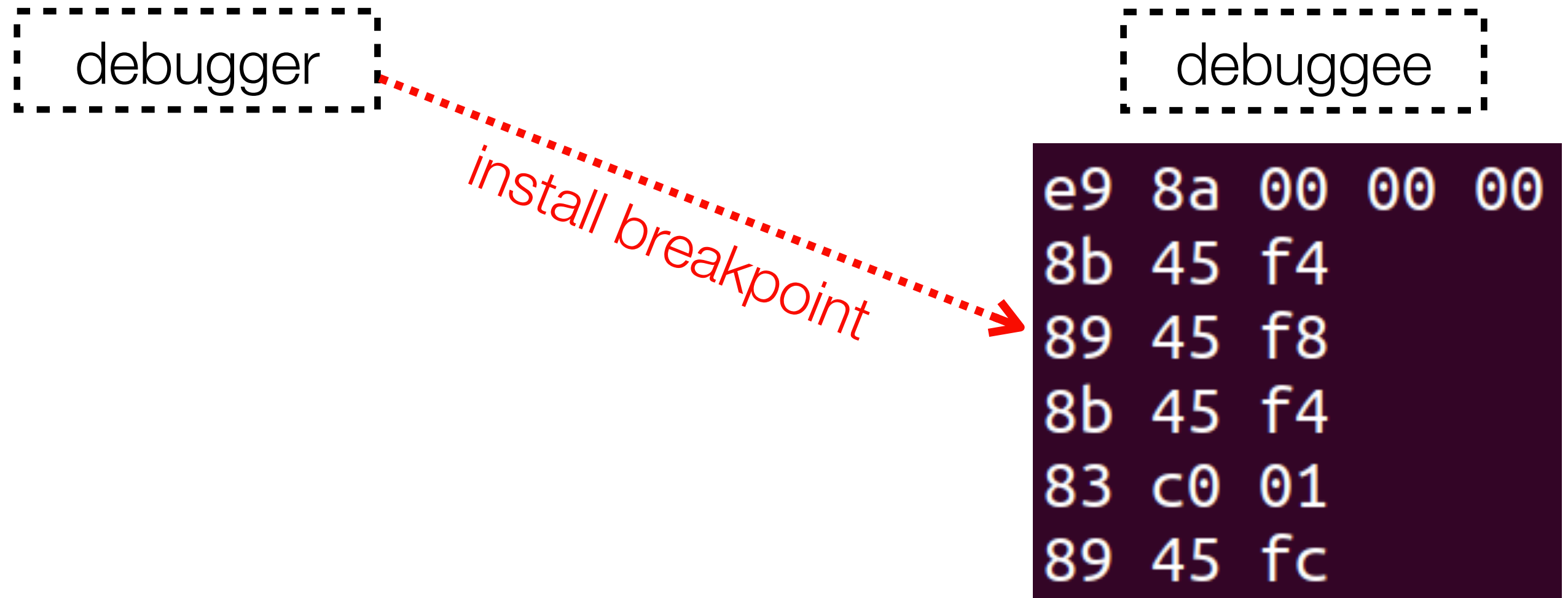
- ▶ Native code: **0xcc** (or INT3)
- ▶ Java: **0xca** opcode
- ▶ Smalltalk: **self halt.** instruction (inserted directly in the source code)

How does it work (native code example)?

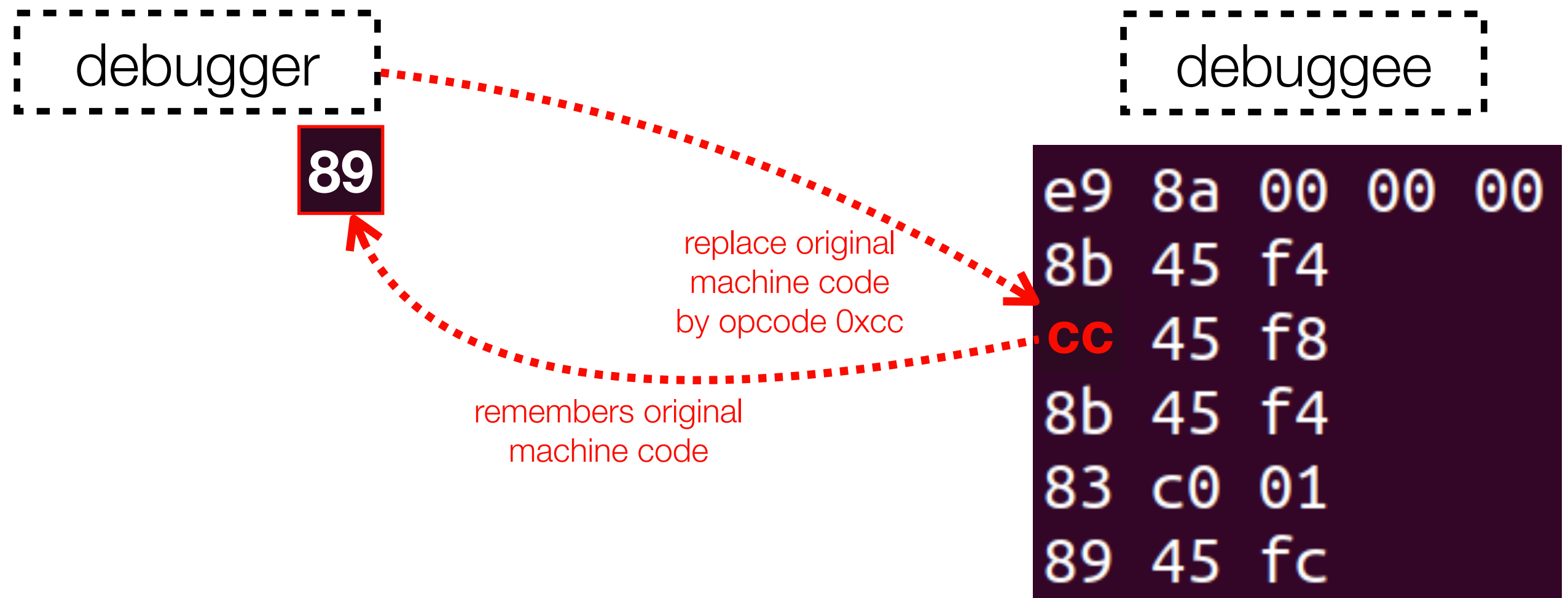


e9	8a	00	00	00
8b	45	f4		
89	45	f8		
8b	45	f4		
83	c0	01		
89	45	fc		

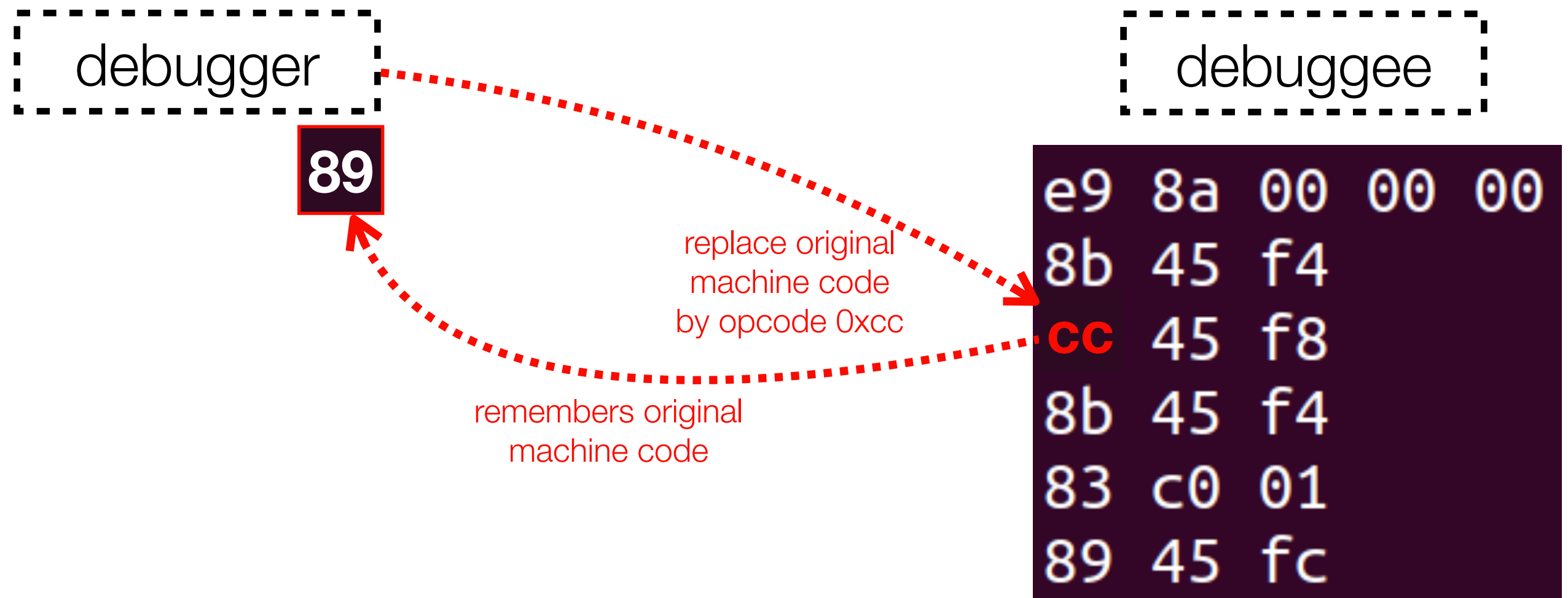
How does it work (native code example)?



How does it work (native code example)?

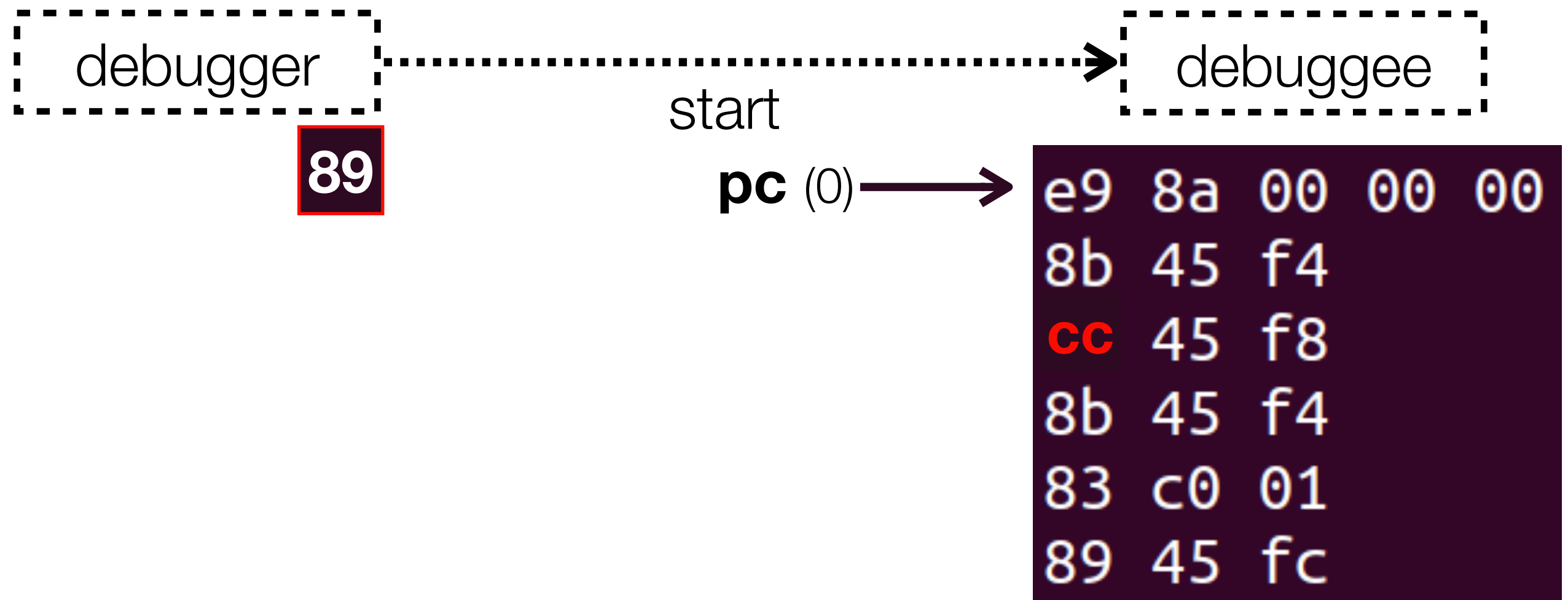


How does it work (native code example)?



Note: if you disassemble the code in which you put a breakpoint, you will most probably not see the **cc** opcode. For example, gdb only shows you the original opcode, and not the instrumentation.

How does it work (native code example)?



How does it work (native code example)?

debugger

89

while the program
executes, the debugger
just wait for signals

debuggee

pc (i) →

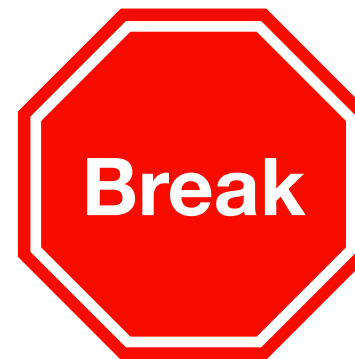
e9	8a	00	00	00
8b	45	f4		
cc	45	f8		
8b	45	f4		
83	c0	01		
89	45	fc		

How does it work (native code example)?



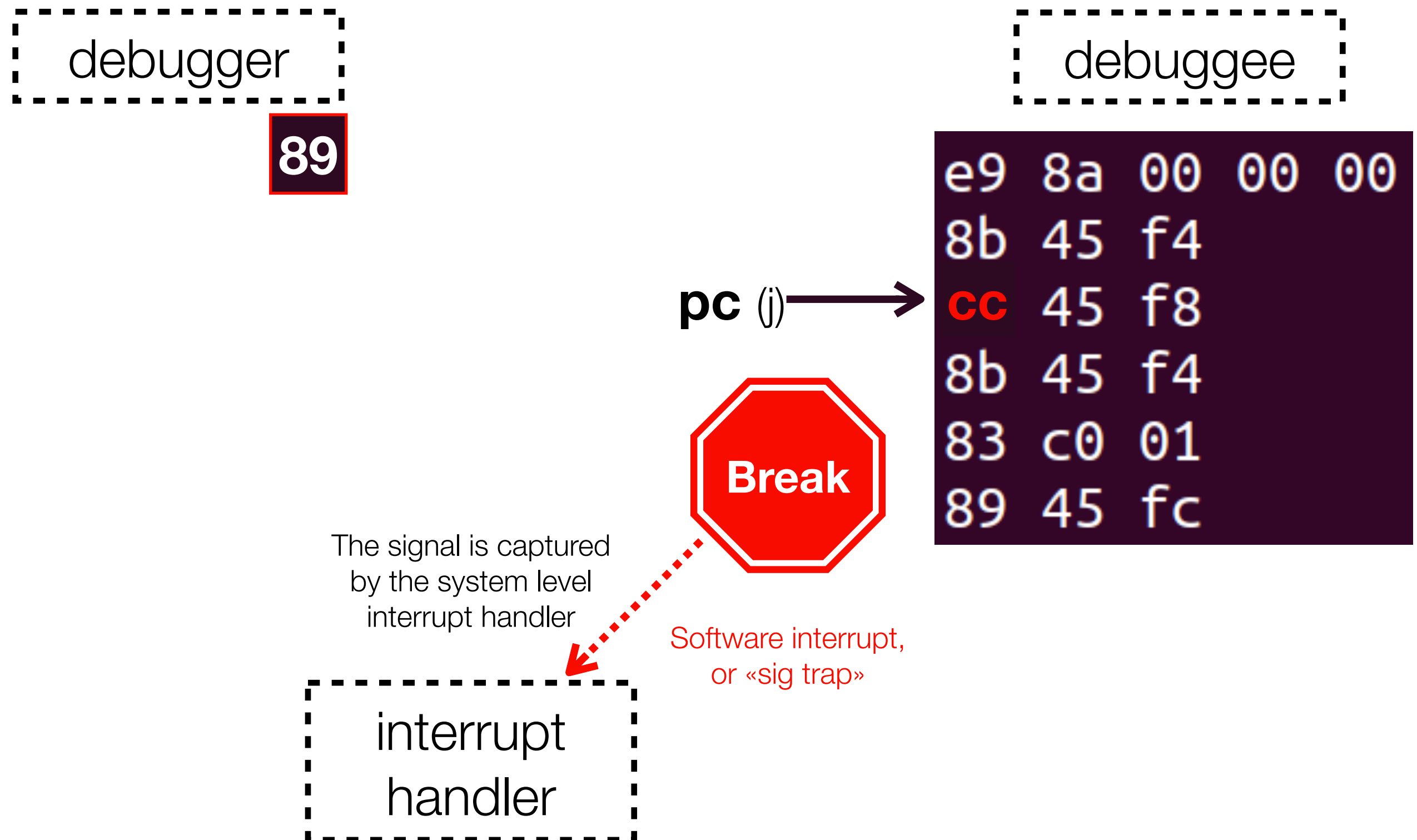
when the **cc** opcode
is hit, an interrupt signal
breaks the execution

pc (j) →

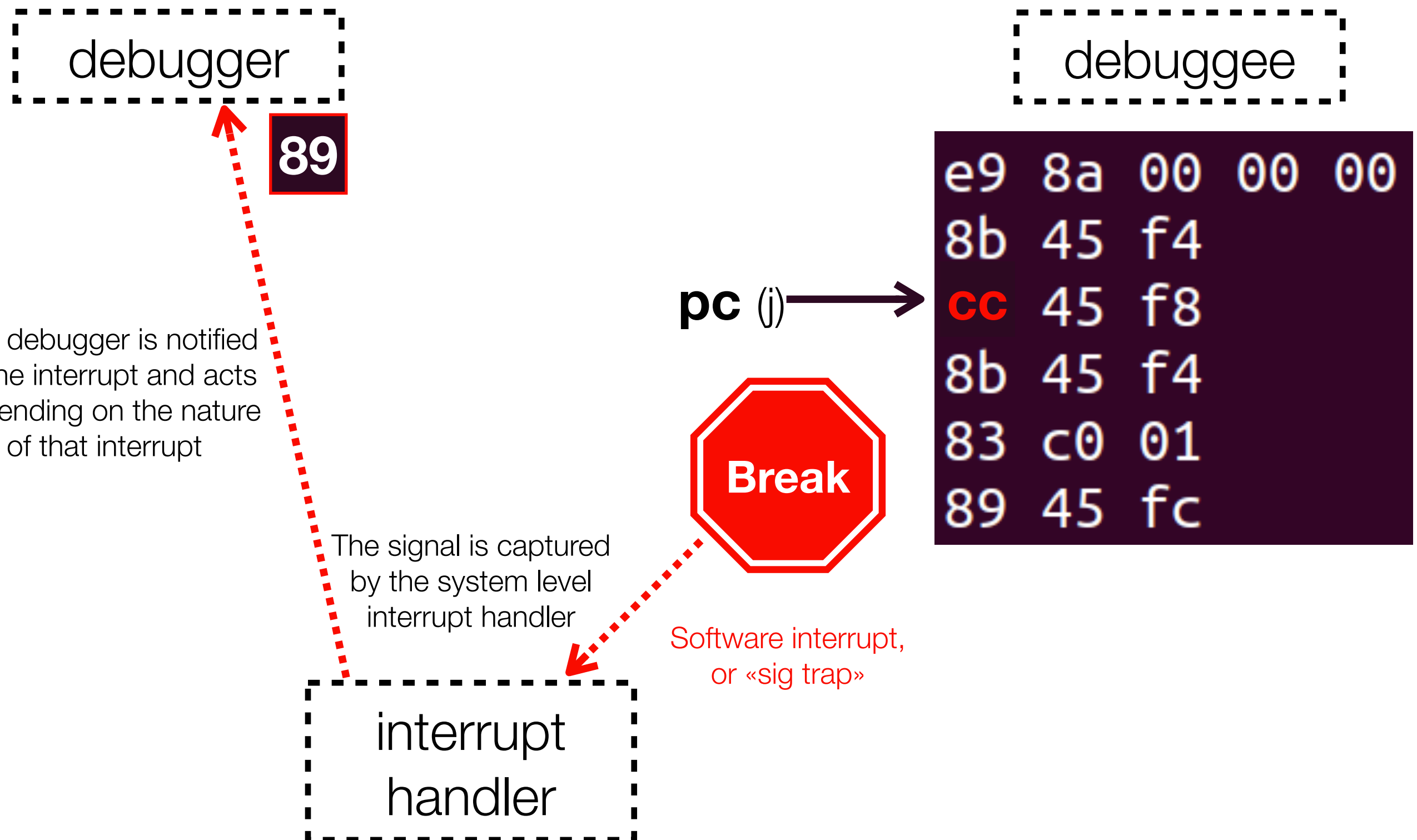


e9	8a	00	00	00
8b	45	f4		
cc	45	f8		
8b	45	f4		
83	c0	01		
89	45	fc		

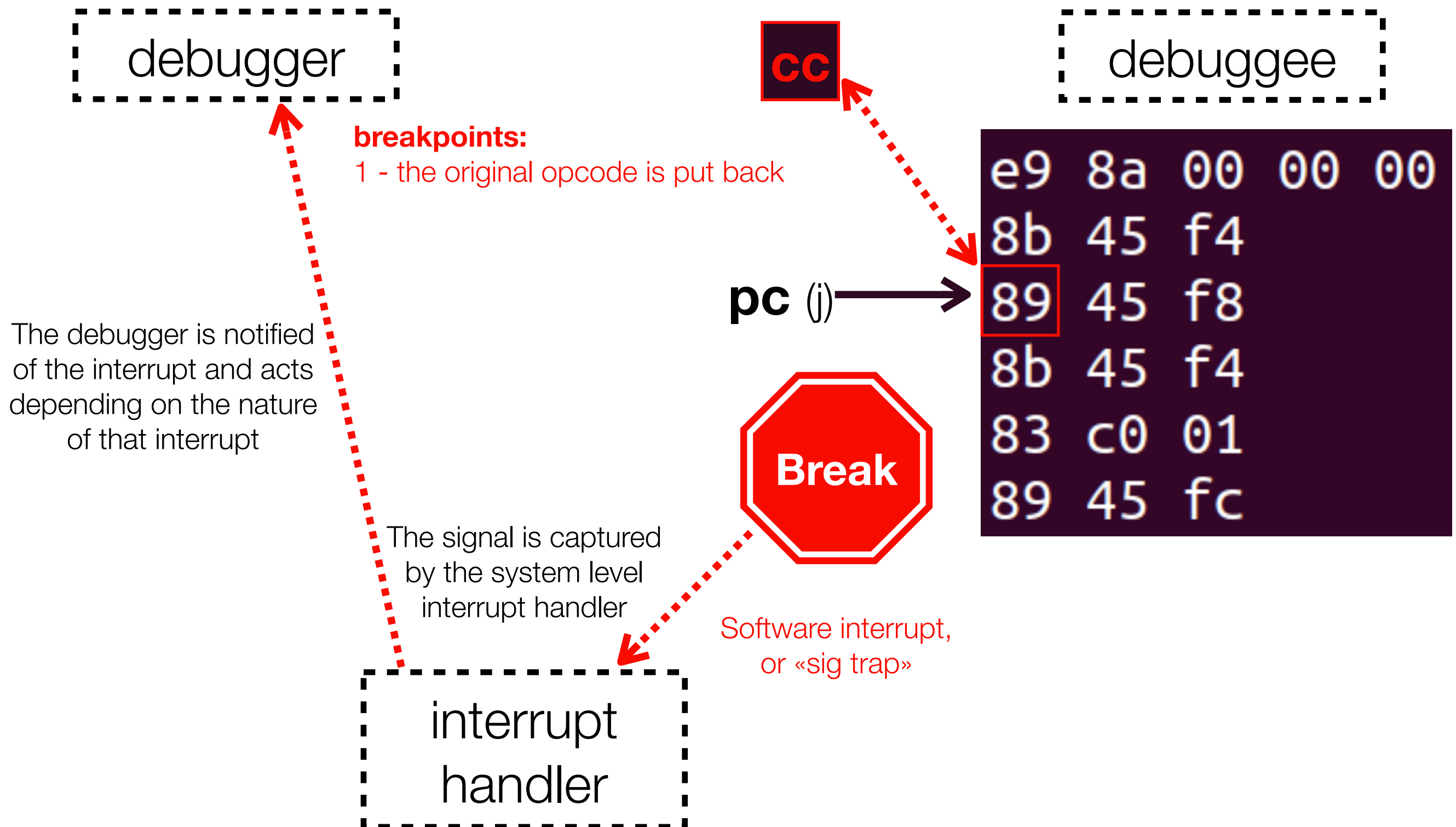
How does it work (native code example)?



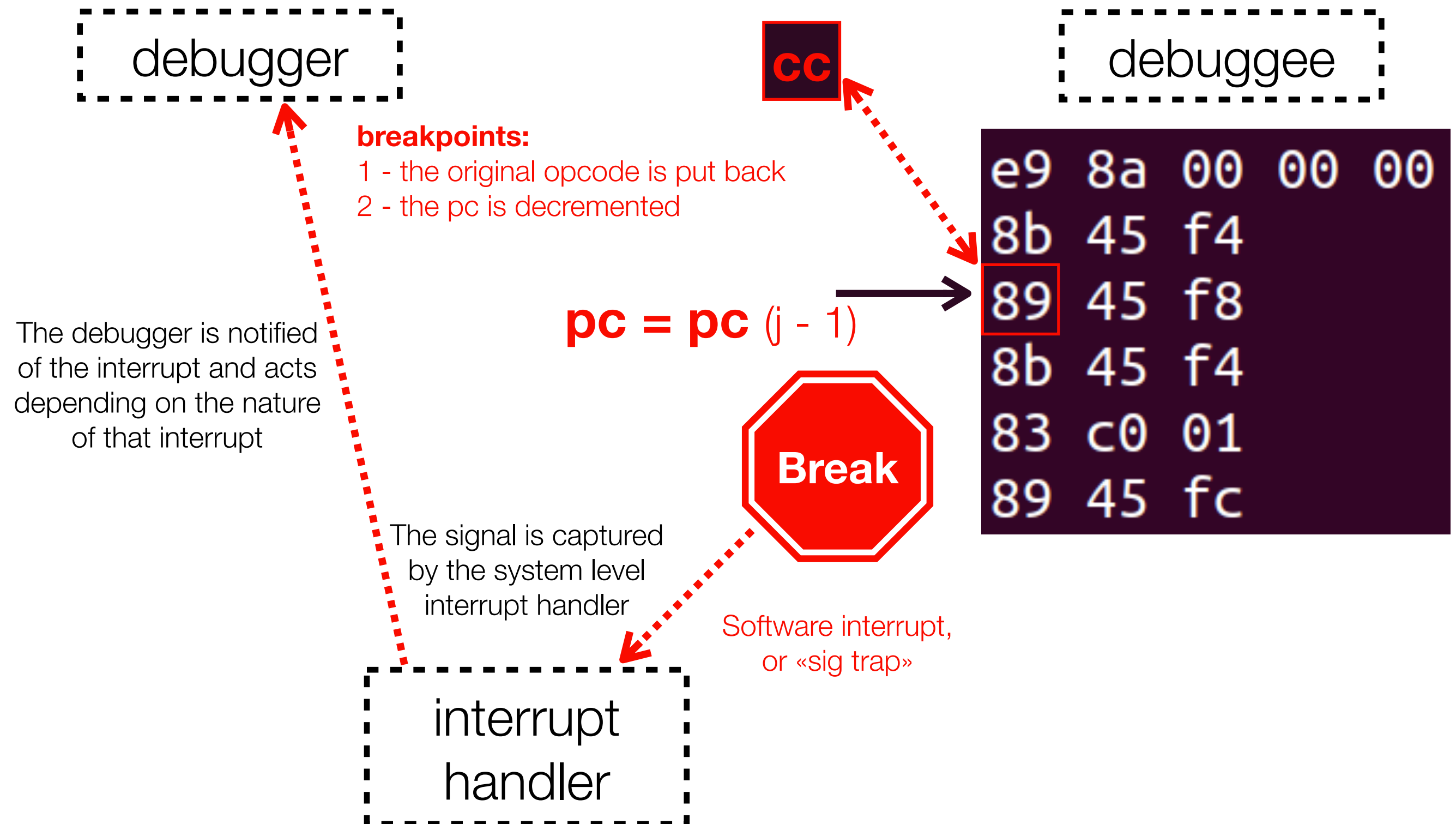
How does it work (native code example)?



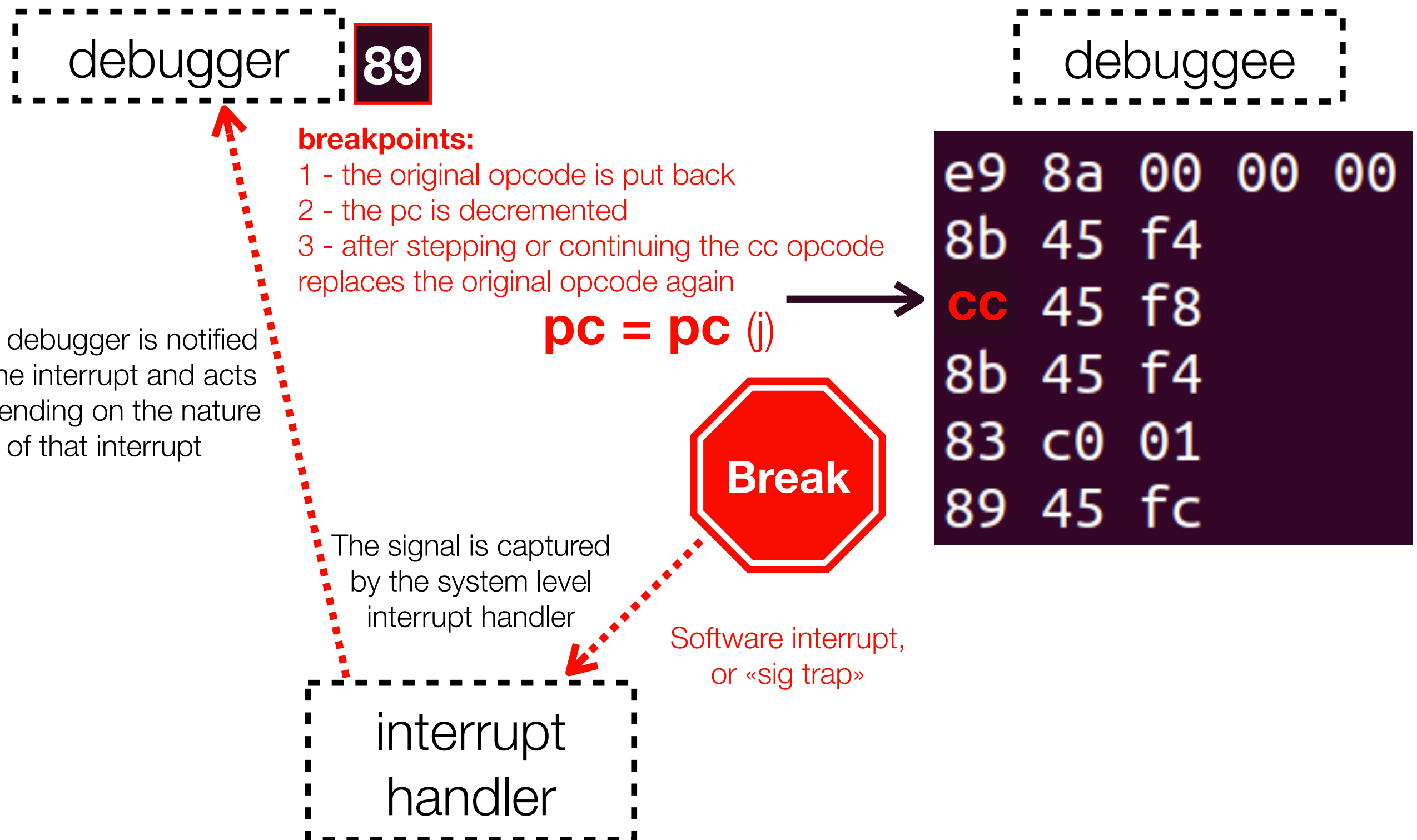
How does it work (native code example)?



How does it work (native code example)?



How does it work (native code example)?



Implementing breakpoints through reflection: the case of Pharo

source code

ensureUpstream

```
| remote |  
remote := self iceRepository remotes first.  
self iceRepository branch setUpstreamIfMissing: remote
```

bytecode

```
65 <4C> self  
66 <80> send: iceRepository  
67 <81> send: remotes  
68 <82> send: first  
69 <D0> popIntoTemp: 0  
70 <4C> self  
71 <80> send: iceRepository  
72 <83> send: branch  
73 <40> pushTemp: 0  
74 <94> send: setUpstreamIfMissing:  
75 <D8> pop  
76 <58> returnSelf
```

Implementing breakpoints through reflection: the case of Pharo

source code

bytecode

ensureUpstream

```
| remote |  
remote := self iceRepository remotes first.  
self iceRepository branch setUpstreamIfMissing: remote
```

```
65 <4C> self  
66 <80> send: iceRepository  
67 <81> send: remotes  
68 <82> send: first  
69 <D0> popIntoTemp: 0  
70 <4C> self  
71 <80> send: iceRepository  
72 <83> send: branch  
73 <40> pushTemp: 0  
74 <94> send: setUpstreamIfMissing:  
75 <D8> pop  
76 <58> returnSelf
```

Implementing breakpoints through reflection: the case of Pharo

source code

ensureUpstream

```
| remote |  
remote := self iceRepository remotes first.  
self iceRepository branch setUpstreamIfMissing: remote
```

bytecode

```
65 <4C> self  
66 <80> send: iceRepository  
67 <81> send: remotes  
68 <82> send: first  
69 <D0> popIntoTemp: 0  
70 <4C> self  
71 <80> send: iceRepository  
72 <83> send: branch  
73 <40> pushTemp: 0  
74 <94> send: setUpstreamIfMissing:  
75 <D8> pop  
76 <58> returnSelf
```

new bytecode

```
89 <4C> self  
90 <80> send: iceRepository  
91 <81> send: remotes  
92 <82> send: first  
93 <23> pushConstant: a Breakpoint  
94 <52> pushThisContext  
95 <24> pushConstant: RBAssignmentNode(remote := self iceRepository remotes first)  
96 <A5> send: breakInContext:node:  
97 <D8> pop  
98 <D1> popIntoTemp: 1  
99 <4C> self  
100 <80> send: iceRepository  
101 <86> send: branch  
102 <41> pushTemp: 1  
103 <97> send: setUpstreamIfMissing:  
104 <D8> pop  
105 <58> returnSelf
```

bytecode insertion

Implementing breakpoints through reflection: the case of Pharo

new bytecode

89 <4C> self

90 <80> send: iceRepository

91 <81> send: remotes

92 <82> send: first

93 <23> pushConstant: a Breakpoint

Push receiver: Breakpoint signal

94 <52> pushThisContext

95 <24> pushConstant: RBAssignmentNode(remote := self iceRepository remotes first)

96 <A5> send: breakInContext:node:

97 <D8> pop

98 <D1> popIntoTemp: 1

99 <4C> self

100 <80> send: iceRepository

101 <86> send: branch

102 <41> pushTemp: 1

103 <97> send: setUpstreamIfMissing:

104 <D8> pop

105 <58> returnSelf

Implementing breakpoints through reflection: the case of Pharo

new bytecode

```
89 <4C> self
90 <80> send: iceRepository
91 <81> send: remotes
92 <82> send: first
93 <23> pushConstant: a Breakpoint    Push receiver: Breakpoint signal
94 <52> pushThisContext    Push arg 1: the execution context
95 <24> pushConstant: RBAssignmentNode(remote := self iceRepository remotes first)
96 <A5> send: breakInContext:node:
97 <D8> pop
98 <D1> popIntoTemp: 1
99 <4C> self
100 <80> send: iceRepository
101 <86> send: branch
102 <41> pushTemp: 1
103 <97> send: setUpstreamIfMissing:
104 <D8> pop
105 <58> returnSelf
```

Implementing breakpoints through reflection: the case of Pharo

new bytecode

```
89 <4C> self
90 <80> send: iceRepository
91 <81> send: remotes
92 <82> send: first
93 <23> pushConstant: a Breakpoint    Push receiver: Breakpoint signal
94 <52> pushThisContext    Push arg 1: the execution context
95 <24> pushConstant: RBAssignmentNode(remote := self iceRepository remotes first)    Push arg 2: the AST node
96 <A5> send: breakInContext:node:
97 <D8> pop
98 <D1> popIntoTemp: 1
99 <4C> self
100 <80> send: iceRepository
101 <86> send: branch
102 <41> pushTemp: 1
103 <97> send: setUpstreamIfMissing:
104 <D8> pop
105 <58> returnSelf
```


Implementing breakpoints through reflection: the case of Pharo

new bytecode

```
89 <4C> self
90 <80> send: iceRepository
91 <81> send: remotes
92 <82> send: first
93 <23> pushConstant: a Breakpoint    Push receiver: Breakpoint signal
94 <52> pushThisContext    Push arg 1: the execution context
95 <24> pushConstant: RBAssignmentNode(remote := self iceRepository remotes first) Push arg 2: the AST node
96 <A5> send: breakInContext:node:    Send message breakInContext:node: with args to receiver BREAK!
97 <D8> pop
98 <D1> popIntoTemp: 1
99 <4C> self
100 <80> send: iceRepository
101 <86> send: branch
102 <41> pushTemp: 1
103 <97> send: setUpstreamIfMissing:
104 <D8> pop
105 <58> returnSelf
```

Implementing breakpoints through reflection: the case of Pharo

new bytecode

```
89 <4C> self
90 <80> send: iceRepository
91 <81> send: remotes
92 <82> send: first
93 <23> pushConstant: a Breakpoint    Push receiver: Breakpoint signal
94 <52> pushThisContext    Push arg 1: the execution context
95 <24> pushConstant: RBAssignmentNode(remote := self iceRepository remotes first) Push arg 2: the AST node
96 <A5> send: breakInContext:node:    Send message breakInContext:node: with args to receiver BREAK!
97 <D8> pop    Pop the return value from stack
98 <D1> popIntoTemp: 1
99 <4C> self
100 <80> send: iceRepository
101 <86> send: branch
102 <41> pushTemp: 1
103 <97> send: setUpstreamIfMissing:
104 <D8> pop
105 <58> returnSelf
```

How to install breakpoints?

- ◆ Breakpoints can be installed directly on a memory address
- ◆ Breakpoints can also be installed in the source code, for example on a function or on a particular line of code

How to install breakpoints?

- ◆ Breakpoints can be installed directly on a memory address
- ◆ Breakpoints can also be installed in the source code, for example on a function or on a particular line of code
- ◆ **But how is the source code mapped to memory addresses after it has been compiled?**

How to install breakpoints?

- ◆ Breakpoints can be installed directly on a memory address
- ◆ Breakpoints can also be installed in the source code, for example on a function or on a particular line of code
- ◆ **But how is the source code mapped to memory addresses after it has been compiled?**
- ◆ We need to add debugging information to the executable program (e.g., in C compile with `-g`)
For example: the **DWARF** format



DWARF

- ◆ **DWARF** describes a program with debugging information entries:
 - ◆ Maps highly optimized binary code (the compiled program) with its source code
 - ◆ A tree structure representing types, variables, functions, lines and the relationship between these structural elements and their binary representation
 - ◆ Portable, extensible, memory efficient

DWARF

◆ DWARF examples: Debugging Information Entries and description of variables in a function

```
fig7.c:
1:  int a;
2:  void foo()
3:  {
4:      register int b;
5:      int c;
6:  }

<1>:  DW_TAG_subprogram
      DW_AT_name = foo
<2>:  DW_TAG_variable
      DW_AT_name = b
      DW_AT_type = <4>
      DW_AT_location = (DW_OP_reg0)
<3>:  DW_TAG_variable
      DW_AT_name = c
      DW_AT_type = <4>
      DW_AT_location =
          (DW_OP_fbreg: -12)
<4>:  DW_TAG_base_type
      DW_AT_name = int
      DW_AT_byte_size = 4
      DW_AT_encoding = signed
<5>:  DW_TAG_variable
      DW_AT_name = a
      DW_AT_type = <4>
      DW_AT_external = 1
      DW_AT_location = (DW_OP_addr: 0)
```

Figure 7. DWARF description of variables
a, b, and c.

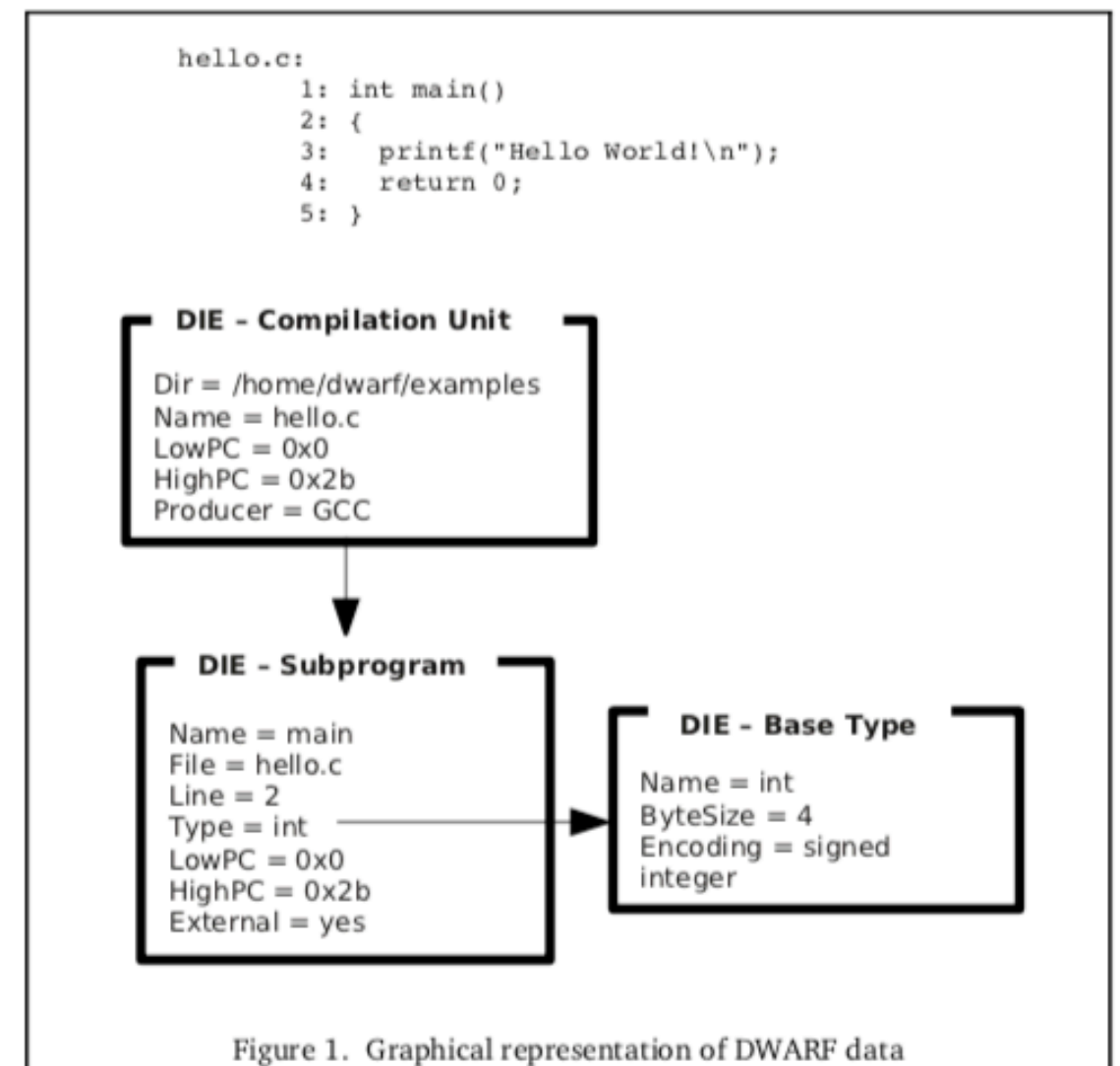


Figure 1. Graphical representation of DWARF data


DWARF

```
strndup.c:
1: #include "ansidecl.h"
2: #include <stddef.h>
3:
4: extern size_t strlen (const char*);
5: extern PTR malloc (size_t);
6: extern PTR memcpy (PTR, const PTR, size_t);
7:
8: char *
9: strndup (const char *s, size_t n)
10: {
11:     char *result;
12:     size_t len = strlen (s);
13:
14:     if (n < len)
15:         len = n;
16:
17:     result = (char *) malloc (len + 1);
18:     if (!result)
19:         return 0;
20:
21:     result[len] = '\0';
22:     return (char *) memcpy (result, s, len);
23: }
```

Figure 8a. Source for strndup.c.

<1>: DW_TAG_base_type DW_AT_name = int DW_AT_byte_size = 4 DW_AT_encoding = signed	<7>: DW_TAG_formal_parameter DW_AT_name = n DW_AT_type = <2> DW_AT_location = (DW_OP_fbreg: 4)
<2>: DW_TAG_typedef DW_AT_name = size_t DW_AT_type = <3>	<8>: DW_TAG_variable DW_AT_name = result DW_AT_type = <10> DW_AT_location = (DW_OP_fbreg: -28)
<3>: DW_TAG_base_type DW_AT_name = unsigned int DW_AT_byte_size = 4 DW_AT_encoding = unsigned	<9>: DW_TAG_variable DW_AT_name = len DW_AT_type = <2> DW_AT_location = (DW_OP_fbreg: -24)
<4>: DW_TAG_base_type DW_AT_name = long int DW_AT_byte_size = 4 DW_AT_encoding = signed	<10>: DW_TAG_pointer_type DW_AT_byte_size = 4 DW_AT_type = <11>
<5>: DW_TAG_subprogram DW_AT_sibling = <10> DW_AT_external = 1 DW_AT_name = strndup DW_AT_prototyped = 1 DW_AT_type = <10> DW_AT_low_pc = 0 DW_AT_high_pc = 0x7b	<11>: DW_TAG_base_type DW_AT_name = char DW_AT_byte_size = 1 DW_AT_encoding = signed char
<6>: DW_TAG_formal_parameter DW_AT_name = s DW_AT_type = <12> DW_AT_location = (DW_OP_fbreg: 0)	<12>: DW_TAG_pointer_type DW_AT_byte_size = 4 DW_AT_type = <13>
	<13>: DW_TAG_const_type DW_AT_type = <11>

Figure 8b. DWARF description for strndup.c.

 **DWARF** example:
description of a function

Figures from: **Introduction to the DWARF
Debugging Format**, Michael J. Eager

Another example: the Pharo debugger map

ensureUpstream

source code

```
| remote |  
remote := self iceRepository remotes first.  
self iceRepository branch setUpstreamIfMissing: remote
```

Another example: the Pharo debugger map

ensureUpstream

source code

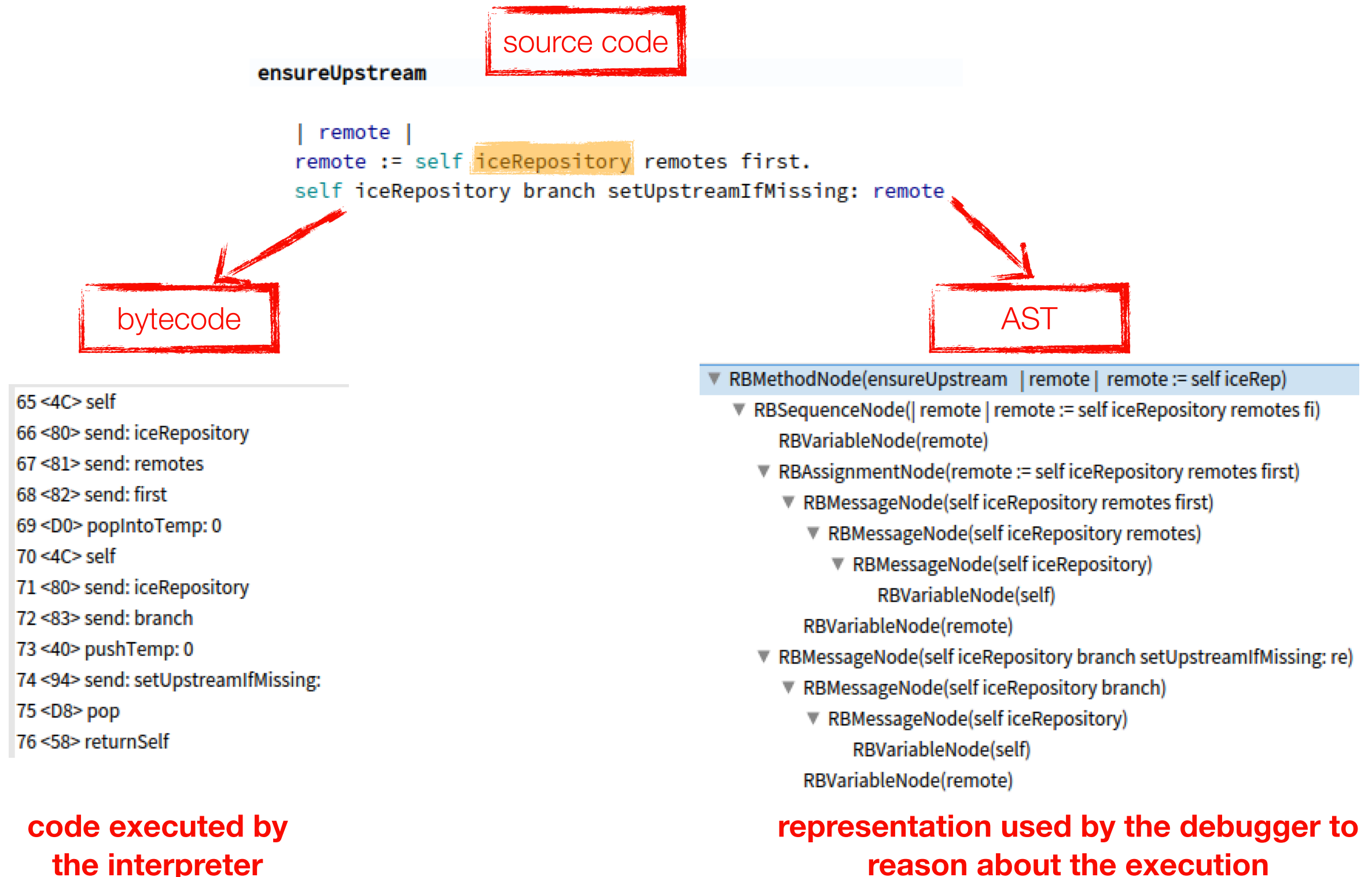
```
| remote |  
remote := self iceRepository remotes first.  
self iceRepository branch setUpstreamIfMissing: remote
```

bytecode

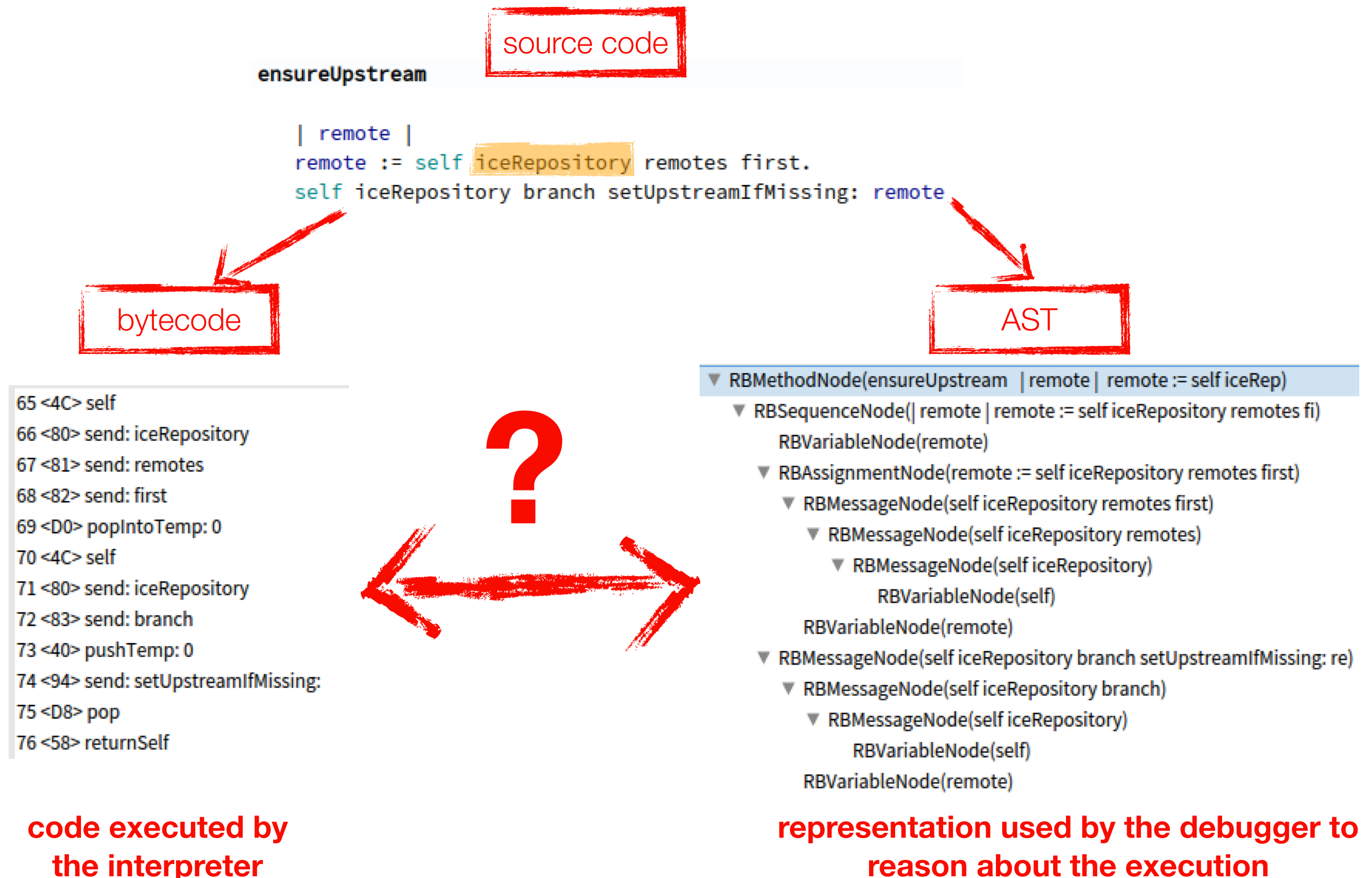
```
65 <4C> self  
66 <80> send: iceRepository  
67 <81> send: remotes  
68 <82> send: first  
69 <D0> popIntoTemp: 0  
70 <4C> self  
71 <80> send: iceRepository  
72 <83> send: branch  
73 <40> pushTemp: 0  
74 <94> send: setUpstreamIfMissing:  
75 <D8> pop  
76 <58> returnSelf
```

**code executed by
the interpreter**

Another example: the Pharo debugger map



Another example: the Pharo debugger map



Another example: the Pharo debugger map

- ◆ The system maintains **a map between the bytecode and the AST**, so that the debugger can track at any time **to which AST node** of a method **corresponds to the bytecode currently being executed**:

`ensureUpstream`

```
| remote |  
remote := self iceRepository remotes first.  
self iceRepository branch setUpstreamIfMissing: remote
```

69	RBAssignmentNode(remote := self iceRepository remotes first)
66	RBMessageNode(self iceRepository)
74	RBMessageNode(self iceRepository branch setUpstreamIfMissing: remote)
71	RBMessageNode(self iceRepository)
68	RBMessageNode(self iceRepository remotes first)
65	RBVariableNode(self)
76	ensureUpstream remote remote := self iceRepository remotes first. self iceRepos
73	RBVariableNode(remote)
70	RBVariableNode(self)
67	RBMessageNode(self iceRepository remotes)
75	RBMessageNode(self iceRepository branch setUpstreamIfMissing: remote)
72	RBMessageNode(self iceRepository branch)

- ◆ The debugger map is generated on-demand per-method, the first time the debugger executes a method, then cached in the system
- ◆ To generate the map, the debugger uses the internal representation of the compiler to compute an program counter index for each node of the AST

Breakpoints in Smalltalk and Python

◆ Smalltalk: insert the `self halt` instruction in the source code.

▶ When this instruction is executed, it sends a signal to the virtual machine that stops the execution and notifies the debugger

◆ Python: insert the `Breakpoint()` instruction (previously `pdb.set_trace()`)

▶ When this instruction is executed, it invokes the debugger

Hardware breakpoints

1

2

3

4

Hardware breakpoints

◆ Debugging facilities embedded in the hardware

- ▶ Dedicated registers are used to store a memory address (named «*DR*» for «*Debug Registers*»)
- ▶ When the pc (program counter) reaches an address that matches a value in the breakpoint registers, the hardware signals an exception
- ▶ On exception, control is transferred to the debugger
- ▶ Breakpoint can be configured to trigger when the memory address **is read, read/written or executed**

Hardware breakpoints

◆ **Limited amount of registers**

- ▶ Usually from 2 to 8 (*i.e.*, 2 to 8 breakpoints maximum)
- ▶ Depends on the hardware and its architecture

References

1. **How debuggers works**, Jonathan B. Rosenberg, 1996
2. **Introduction to the DWARF Debugging Format**, Michael J. Eager, 2012
3. **How C++ Debuggers work**, Simon Brand, Meeting C++ 2017 – <https://www.youtube.com/watch?v=Q3Rm95Mk03c>
4. **How C++ Debuggers Work**, Simon Brand , CppCon 2018 – <https://www.youtube.com/watch?v=0DDrseUomfU>
5. **GDB internals**, Free Software Foundation, 2008 – <https://sourceware.org/gdb/wiki/Internals/Breakpoint%20Handling>
6. **x86 debug register** – https://en.wikipedia.org/wiki/X86_debug_register
7. **Interrupt** – <https://en.wikipedia.org/wiki/Interrupt>
8. <https://docs.microsoft.com/en-us/visualstudio/extensibility/debugger/>
9. <https://docs.oracle.com/javase/8/docs/technotes/guides/jpda/architecture.html>
10. **How debuggers works**, Jonathan B. Rosenberg, 1996
11. **Handling Error-Handling Errors: dealing with debugger bugs in Pharo**, S. Costiou, T. Dupriez, D. Pollet, IWSST 2020