

# System software

**Simulation, emulation,  
and virtual machines**



# Basic notions

- Simulator
  - imitation of a computer system based on a **model** for the system
    - model represents key characteristics (functions, behaviour, state) of the system
  - focus on internal state as represented by the model
    - can run much slower than the real system
    - „good“ simulation may also be considered as emulation
  - used for analysis and study
  - examples
    - Flight simulator, physics engines, weather simulation

# Basic notions

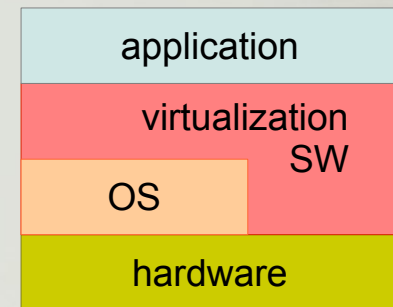
- Emulator
  - software for one computer system (**host**) that mimicks another computer system (**guest**)
  - focus on observable behaviour of guest
    - internal state of the guest may not be accurately emulated
  - used as a substitute
    - can replace the original system
  - examples
    - emulators of ZX Spectrum, Commodore 64, ...
    - DOSBox, Bochs, terminal emulators, ...

# Basic notions

- Virtualization
  - mechanism that creates something virtually
    - mostly refers to computer systems
  - maps virtual system to some real system
    - interfaces and resources of virtual device are mapped to interfaces and resources of real device (which imitates the virtual one)
    - may not use emulation
      - e.g. some virtual instructions may be directly executed by the host processor, virtual system may see real I/O devices
  - examples
    - VMWare, VirtualBox, Virtual PC

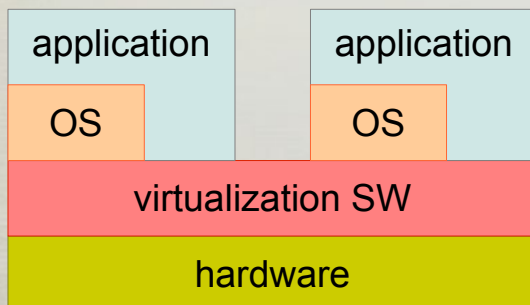
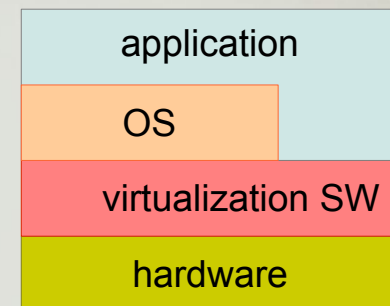
# Virtualization

- Process virtual machines
  - support for individual process
  - **runtime system**: virtualization software
    - uses host OS and hardware
    - emulates processor
    - portability of applications
  - examples
    - execution of intermediate code of programming languages
    - e.g. JVM, CLI, Parrot, Neko, Lua, Python



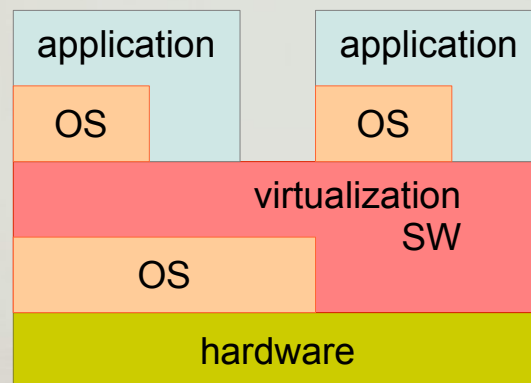
# Virtualization

- System virtual machines
  - emulates whole computer system
    - processor, memory, devices etc.
  - **hypervisor:** virtualization software
    - virtual machine monitor
    - executes directly on hardware
  - replication of resources



# Virtualization

- System virtual machines
  - **hosted** virtual machine
  - examples
    - Vware Player, VirtualBox, ...

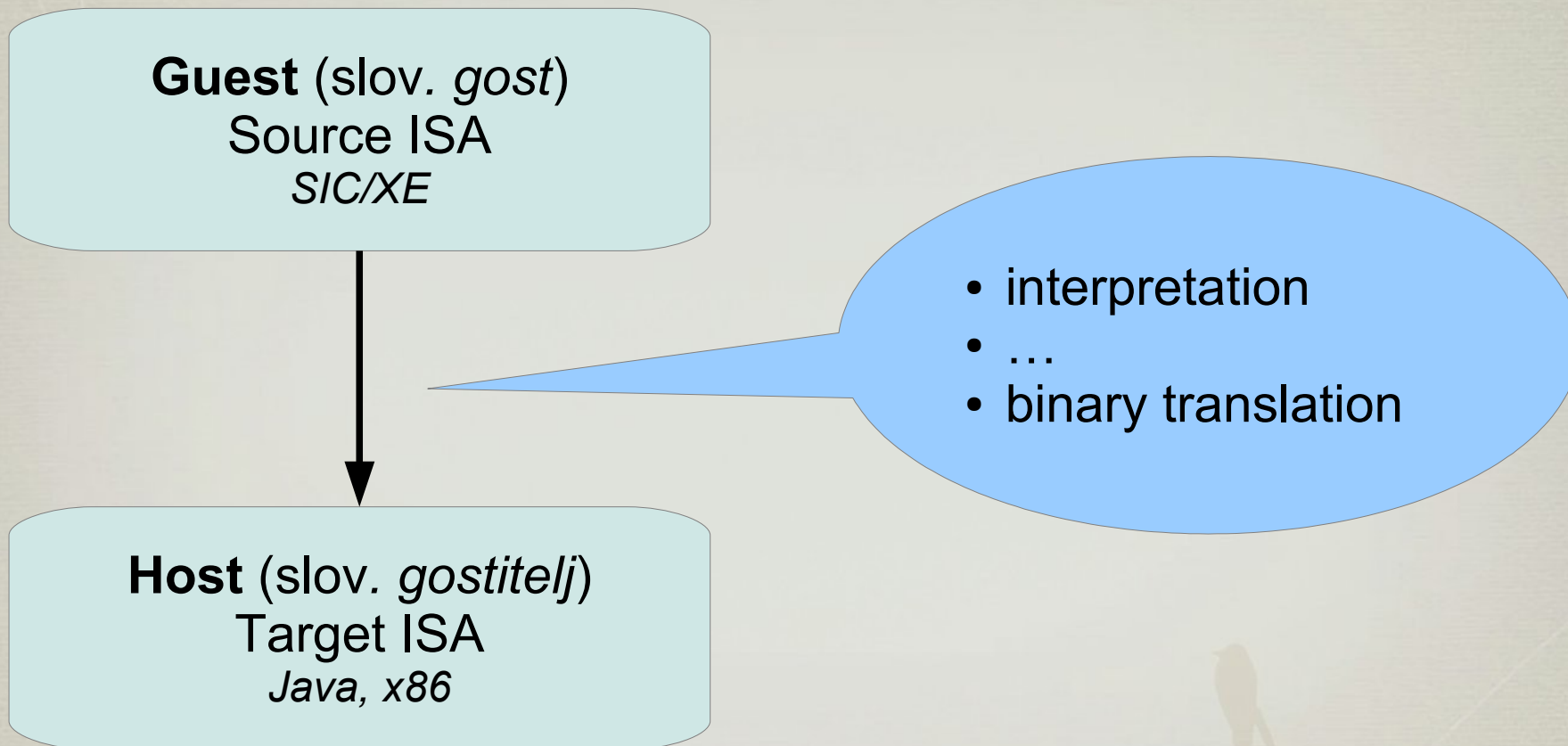


# Emulation

- Emulator of computer system
  - memory
    - e.g. array of bytes
  - registers
  - processor
    - emulation of all or selected set of instructions
  - devices
    - stream (character) devices
    - memory-mapped devices



# Emulation



# Interpretation

- Similar as in a processor
  - fetch instruction from the PC address
  - decode instruction
  - fetch operands
  - decode operands
  - execute instruction

# Interpretation

- *Decode-and-dispatch loop*

```
while (!halt) {  
    opcode = fetch();  
  
    switch (opcode) {  
        case Opcode.LDA:  
            regA = fetch() << 8 | fetch();  
            break;  
  
        case Opcode.STA:  
            ...;  
            break;  
  
        case Opcode.ADD:  
            ...;  
            break;  
  
        ...  
    }  
}
```



One  
big  
switch



# Interpretation

- *Indirect threaded interpretation*

**LDA:**

```
// execute the instruction  
regA = fetch() << 8 | fetch();  
// decode and jump to the next instruction  
opcode = fetch();  
routine = dispatch[opcode];  
goto routine;
```

**STA:**


...

**ADD:**

...

A table mapping  
opcodes to routines.

Routines are represented  
with labels  
(symbolic addresses).



We need  
labels and  
gotos

# Interpretation

- *Predecoding*
  - instructions are decoded before execution
  - important data (operands) are stored in easily accessible data structure (array of records)
    - TPC ... target PC (table index)
    - SPC ... source PC
      - manipulated separately, it may be possible to calculate SPC from TPC

## LDA:

```
// execute the instruction  
regA = code[TPC].operand;  
// decode and jump to the next instruction  
TPC++;  
SPC += 3;  
opcode = code[TPC].opcode;  
routine = dispatch[opcode];  
goto routine;
```

# Interpretation

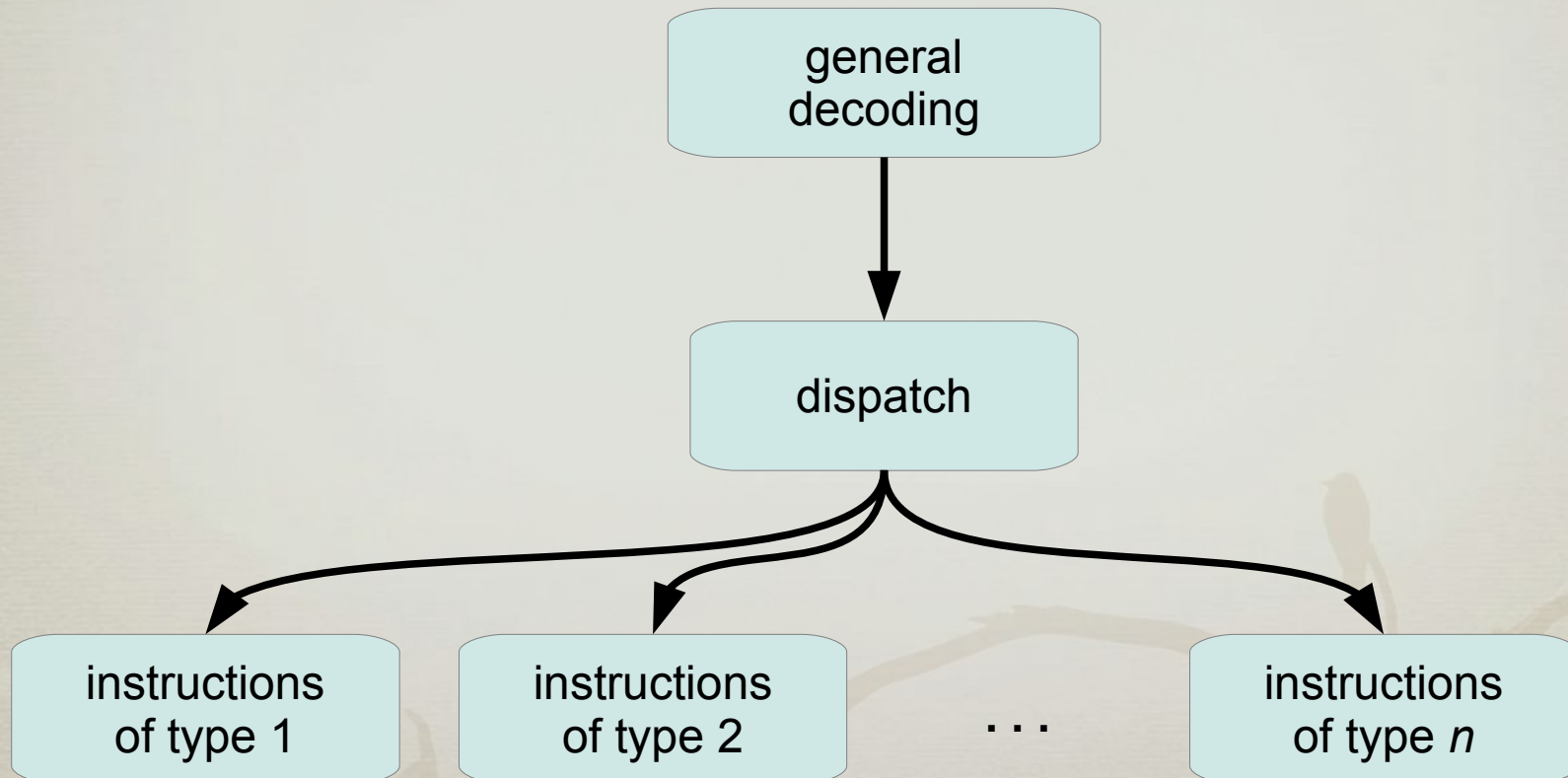
- *Direct threaded interpretation*
  - predecoding where we also pre-calculate addresses of routines

**LDA:**

```
// execute the instruction  
regA = code[TPC].operand;  
// decode and jump to the next instruction  
TPC++;  
SPC += 3;  
routine = code[TPC].routine;  
goto routine;
```

# Interpretation

- What about CISC ISA?
  - much more complicated decoding than RISC



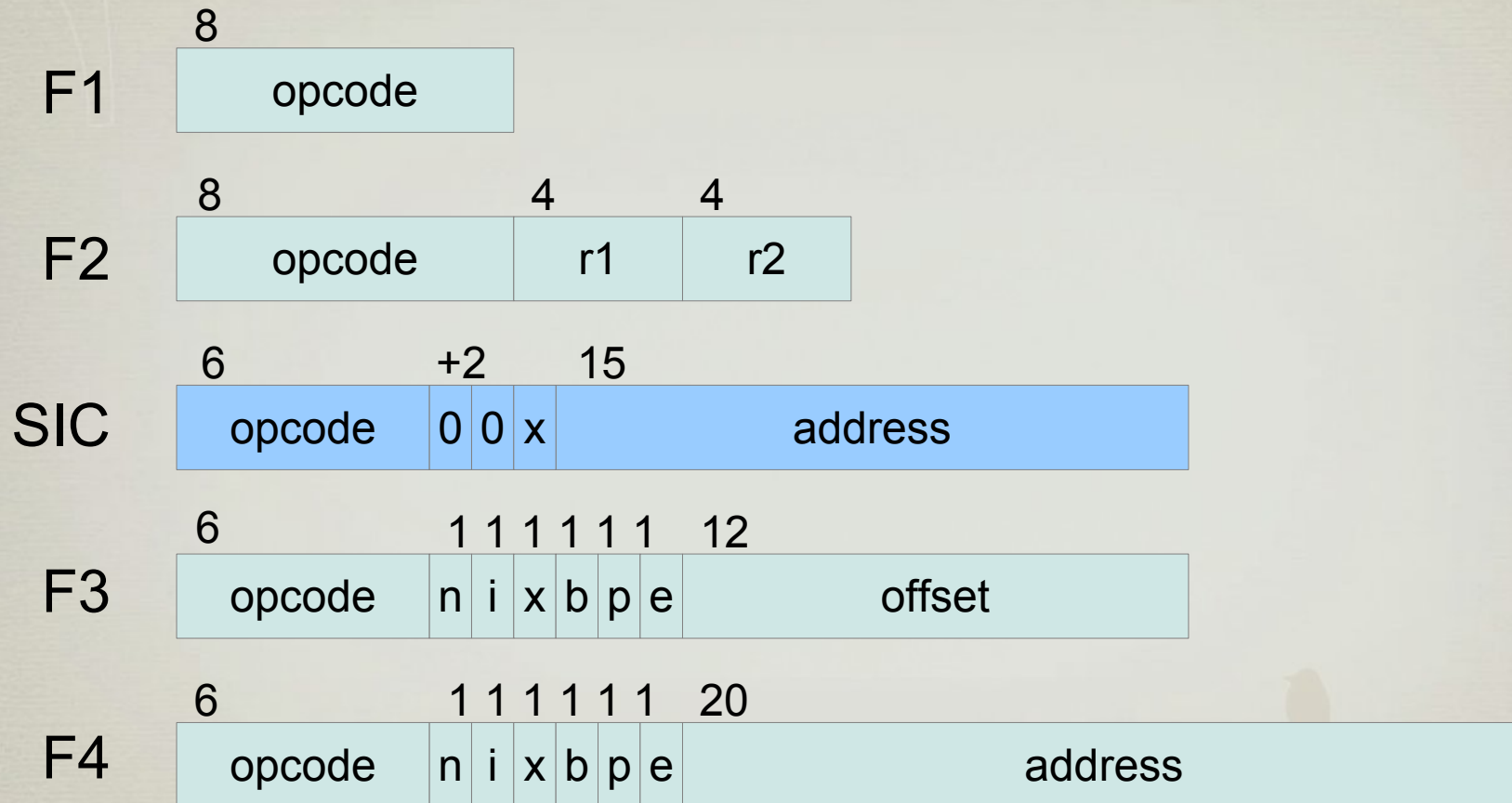
# System software

## SIC/XE emulation

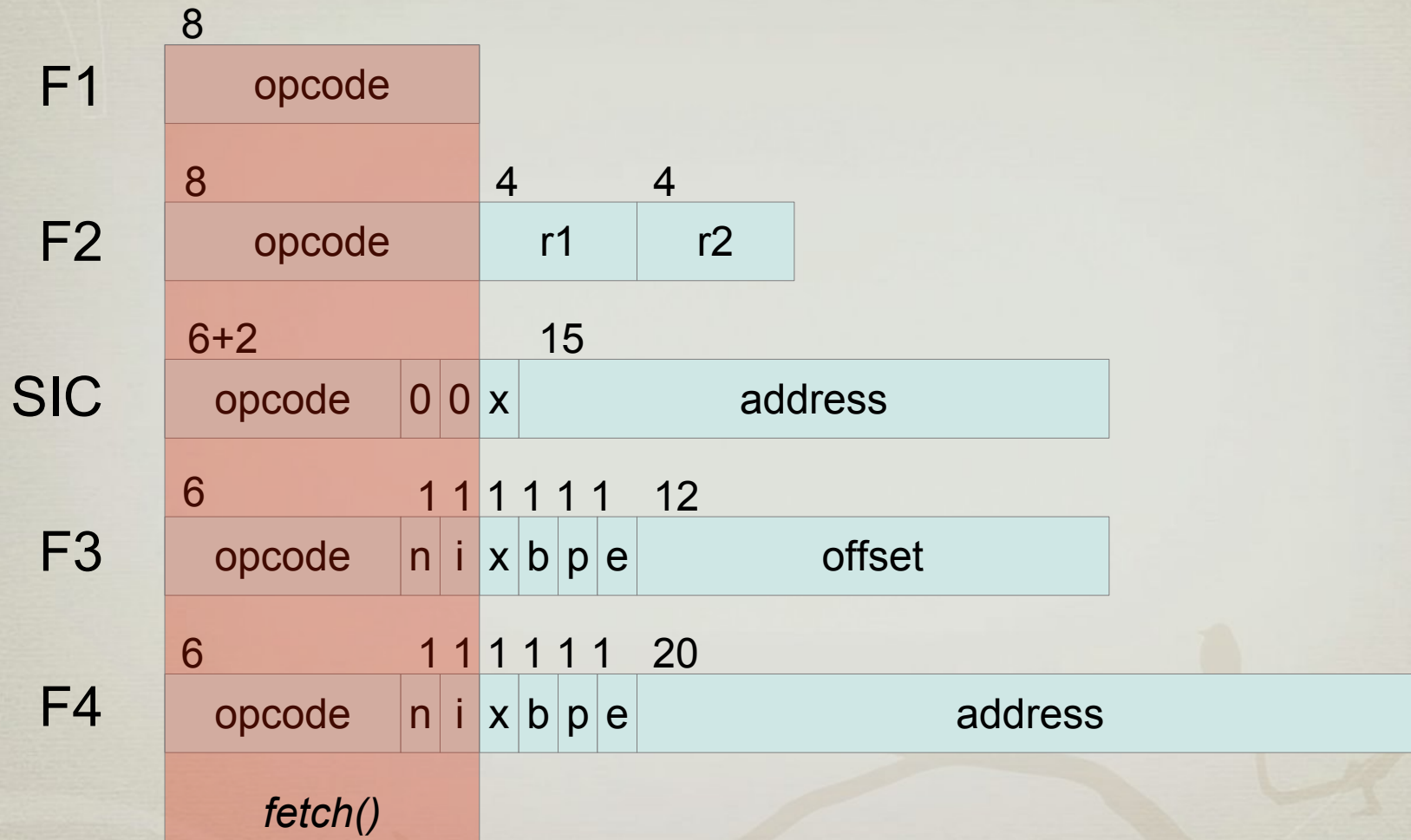




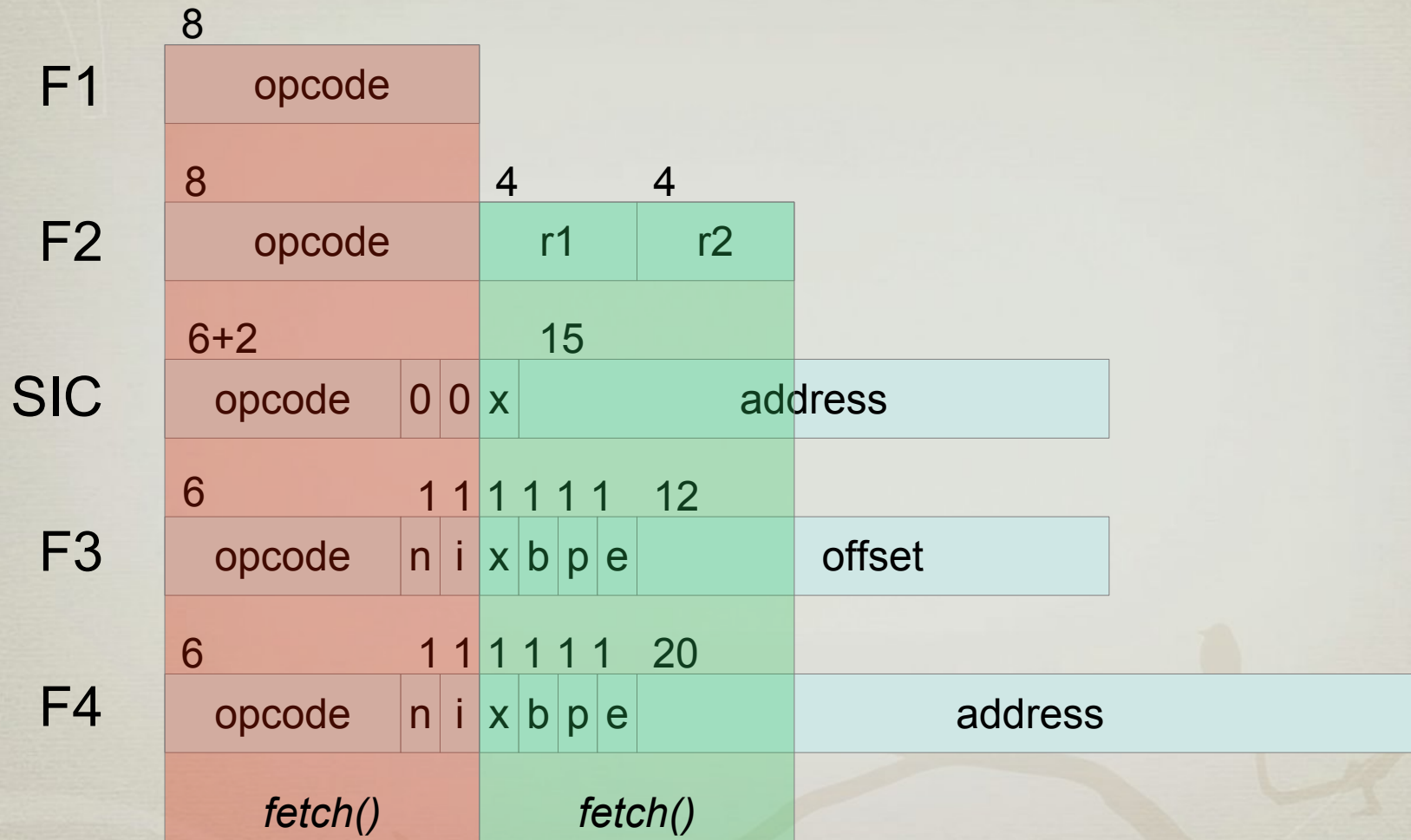
# SIC/XE instruction formats



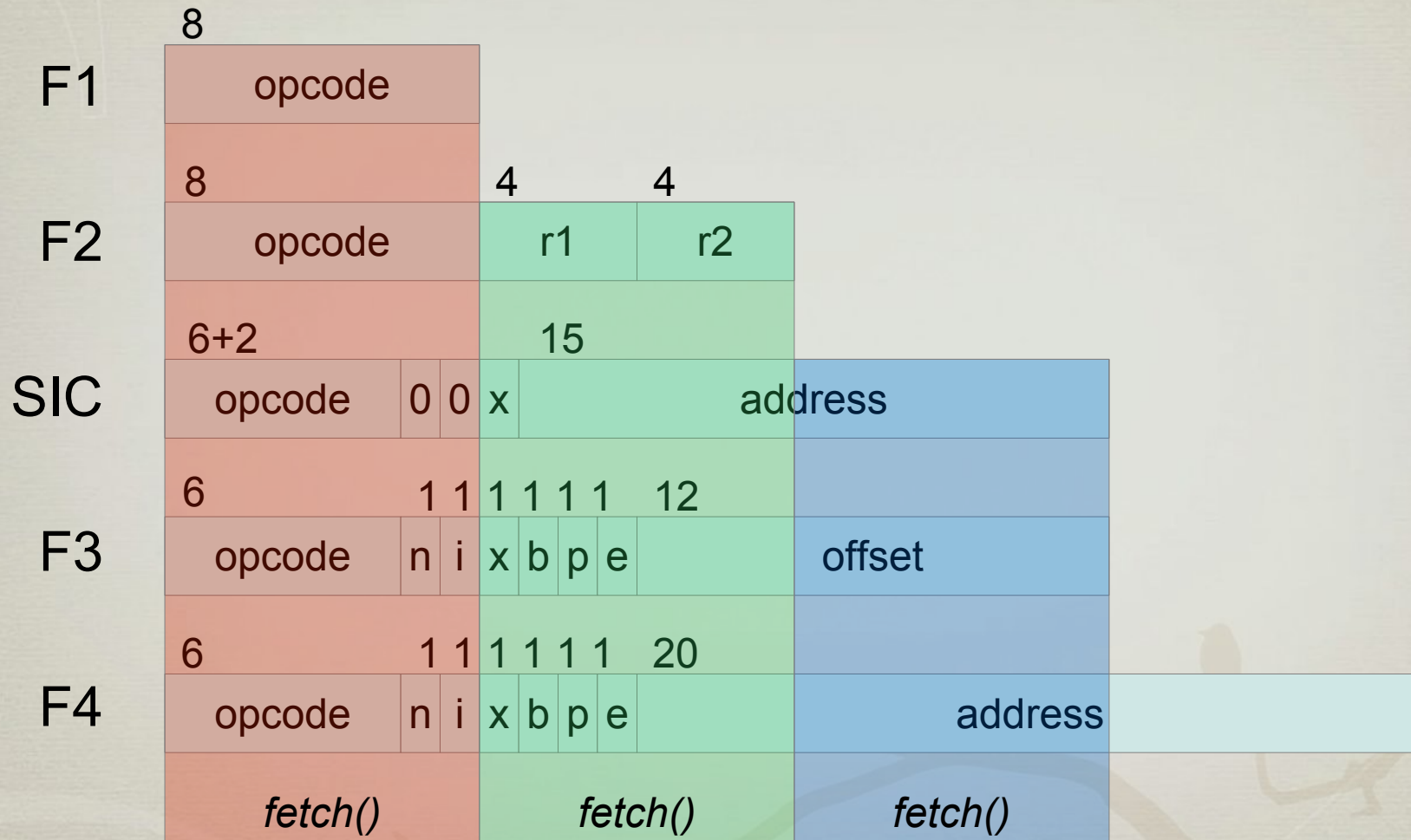
# SIC/XE instruction formats



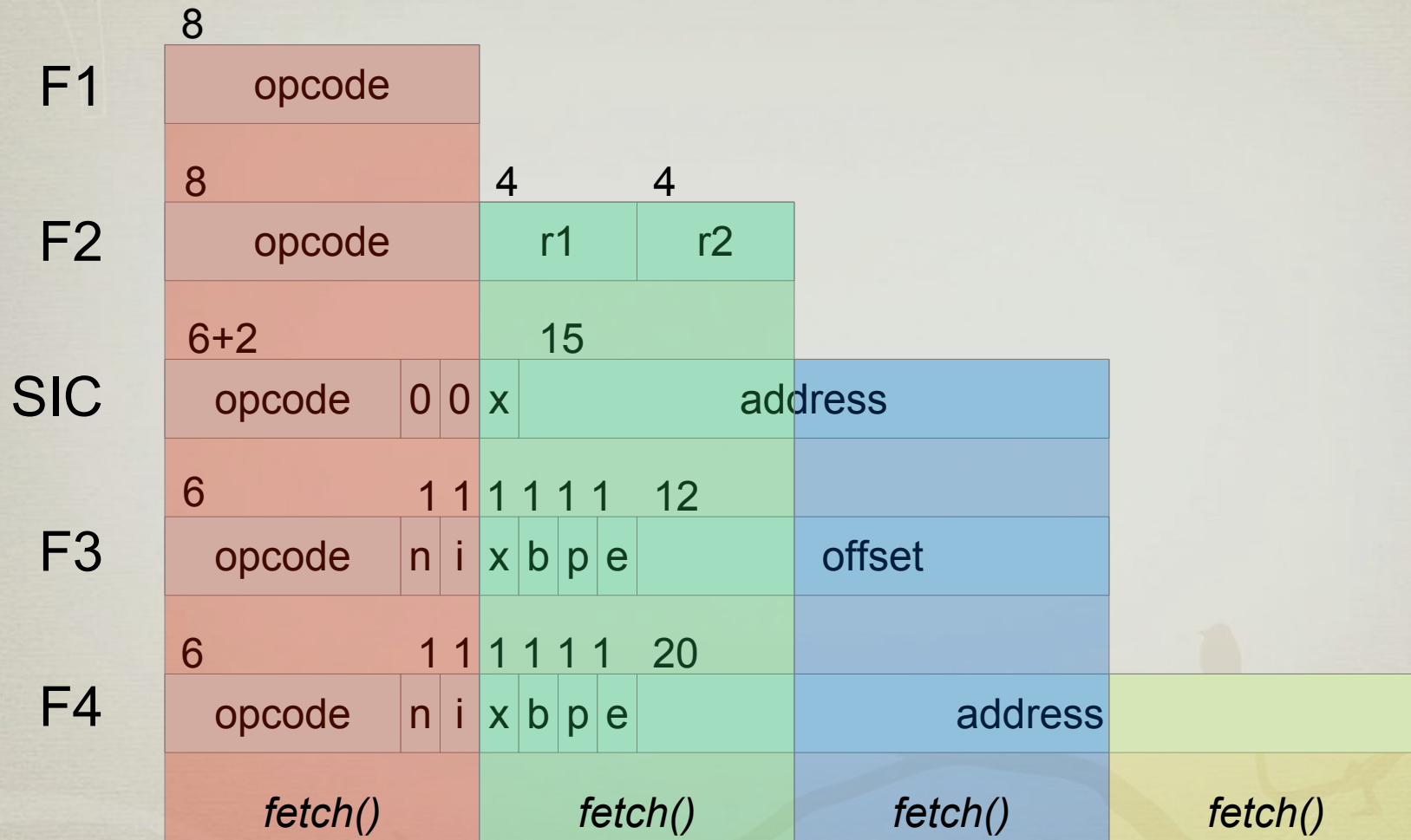
# SIC/XE instruction formats



# SIC/XE instruction formats



# SIC/XE instruction formats



# SIC/XE executor

- Decode & fetch sequence
  - Fetch opcode (one byte)
  - Is it F1?
    - if yes then execute it
  - Otherwise it is F2, SIC, F3 or F4
    - fetch another byte
  - Is it F2?
    - if yes then decode the two registers and execute
  - Otherwise it is SIC, F3 or F4
  - ...

# SIC/XE executor

- F3 and F4 instructions are the same
  - only the operands are of different sizes
- SIC is as strict subset of F3/F4
  - the book standard
- Suggestion
  - first, decode operand depending on format
  - then, treat the execution of SIC, F3 and F4 together
    - SIC is extended to F3/F4

# SIC/XE executor

- Executor for particular formats
  - `boolean execF1(int opcode)`
  - `boolean execF2(int opcode, int operand)`
  - `boolean execSICF3F4(int opcode, int ni, int operand)`
  - Checks the opcode, and if it is of the right format it is executed and returns `true`.





# SIC/XE executor

- General executor:
  - `void execute()`

```
public void execute() {  
    int opcode = fetch();  
    if (execF1(opcode)) return;  
    int op = fetch();  
    if (execF2(opcode, op)) return;  
    . . .  
}
```

# SIC/XE executor

- Decoding of SIC, F3, F4 operand
  - Check if the bits  $n=0$  and  $i=0$ ?
    - Thus, we have SIC operand.
  - Is the bit  $e$  set?
    - Thus, we have extended format – F4.
  - Otherwise it is F3 operand.
    - Način izračuna operanda je podan v bitih  $b_p$ :
      - direktno, bazno relativno, PC-relativno ali napačno naslavljanje.

# SIC/XE executor

- Indexed addressing
  - specified in the bit  $x$
  - possible only with simple addressing

```
if (Opcode.isIndexed(op))  
    if (Opcode.isSimple(ni)) operand += regX;  
    else invalidAddressing();
```

# SIC/XE executor

- Execution
  - Bits `ni` specify a use of TA
  - Call `execSICF3F4(opcode, ni, operand)`
    - remember to reset bits `ni` in the `opcode` field
    - if it returns `false` then invalid `opcode`

# SIC/XE executor

- Load & store confusion
  - store instructions implicitly contain one level of indirection
  - `STA 42`  $\rightarrow$  `m[42] = regA`

instruction	description	operand
LDA #42	$A \leftarrow 42$	42
LDA 42	$A \leftarrow m[42]$	m[42]
LDA @42	$A \leftarrow m[m[42]]$	m[m[42]]
STA #42	$42 \leftarrow A$	-
STA 42	$m[42] \leftarrow A$	42
STA @42	$m[m[42]] \leftarrow A$	m[42]