Inside an Emulator

Maarten ter Huurne
T-DOSE 2014

About the speaker

openMSX

DOLPHIN EMU

MSX

GameCube + Wii

openmsx.org

dolphin-emu.org

About the speaker

openMSX

2001-2014

2992 commits

DOLPHIN EMU

2008, 2011

235 commits

Contents

- What is emulation?
- How does it work?
  - CPU emulation
  - Peripheral emulation
  - Synchronization
- Thinking non-linearly
  - Determinism
  - Tool-Assisted Speedruns
  - Debugger with single-step-back

What is an emulator?

A **program** to duplicate the behavior of **one machine** on **another machine**.

Host: machine running the emulator

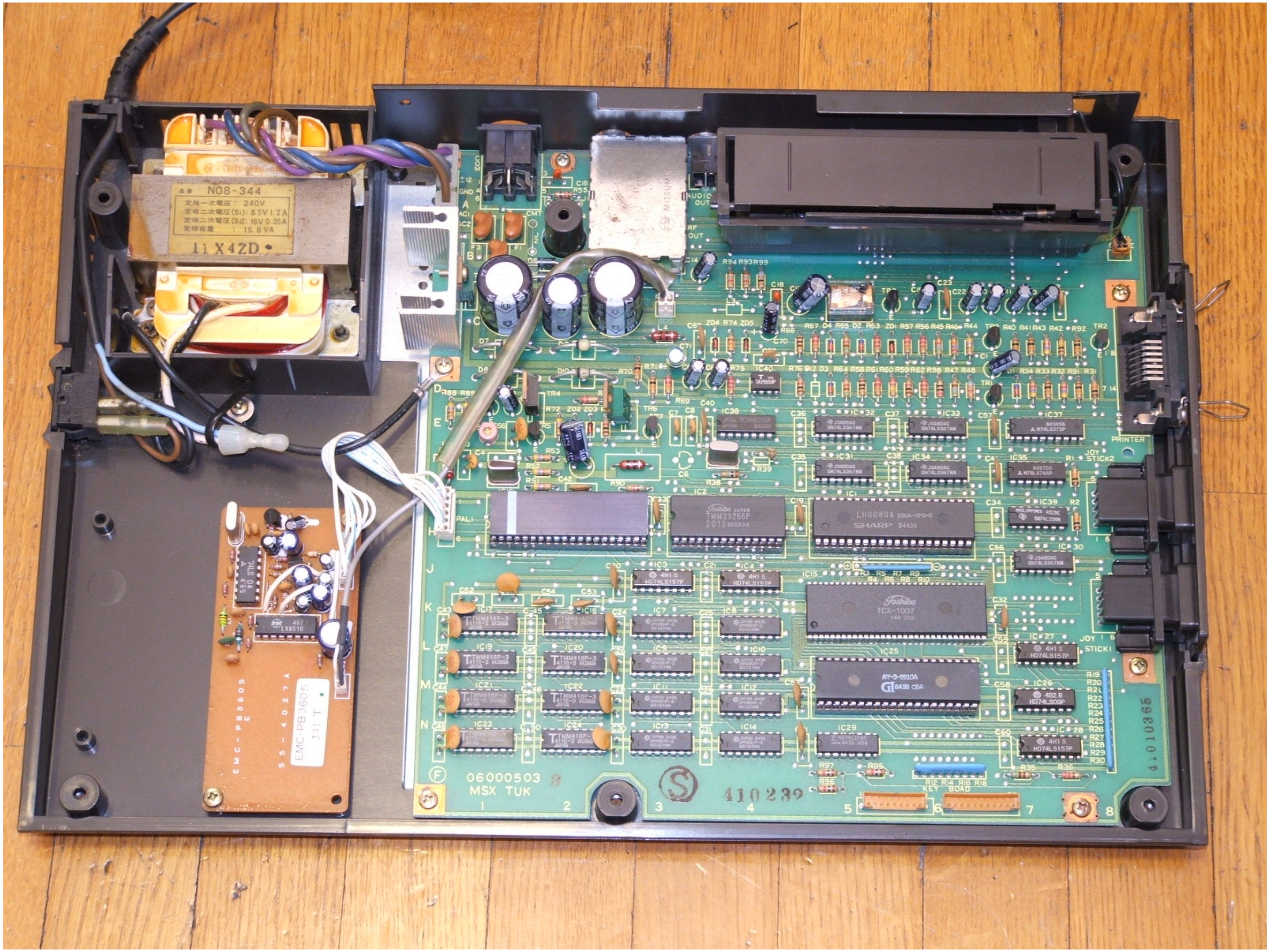
Guest: machine being emulated

Types of emulation

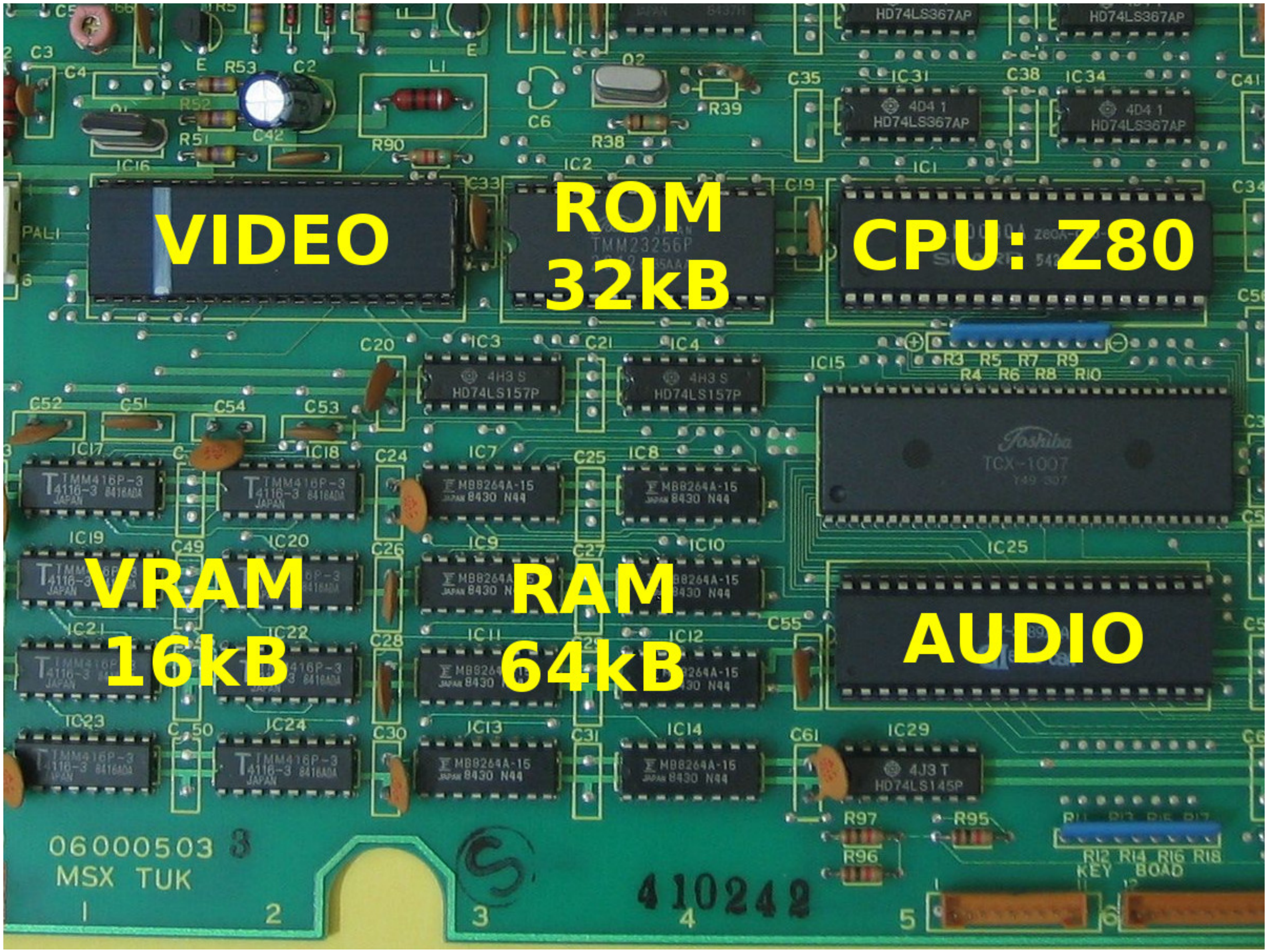
Type:	Host:	Guest:
Full-system	machine A	machine B
Virtualization	machine A	machine A
Virtual machine	machine A	imaginary machine

Guest machine: MSX

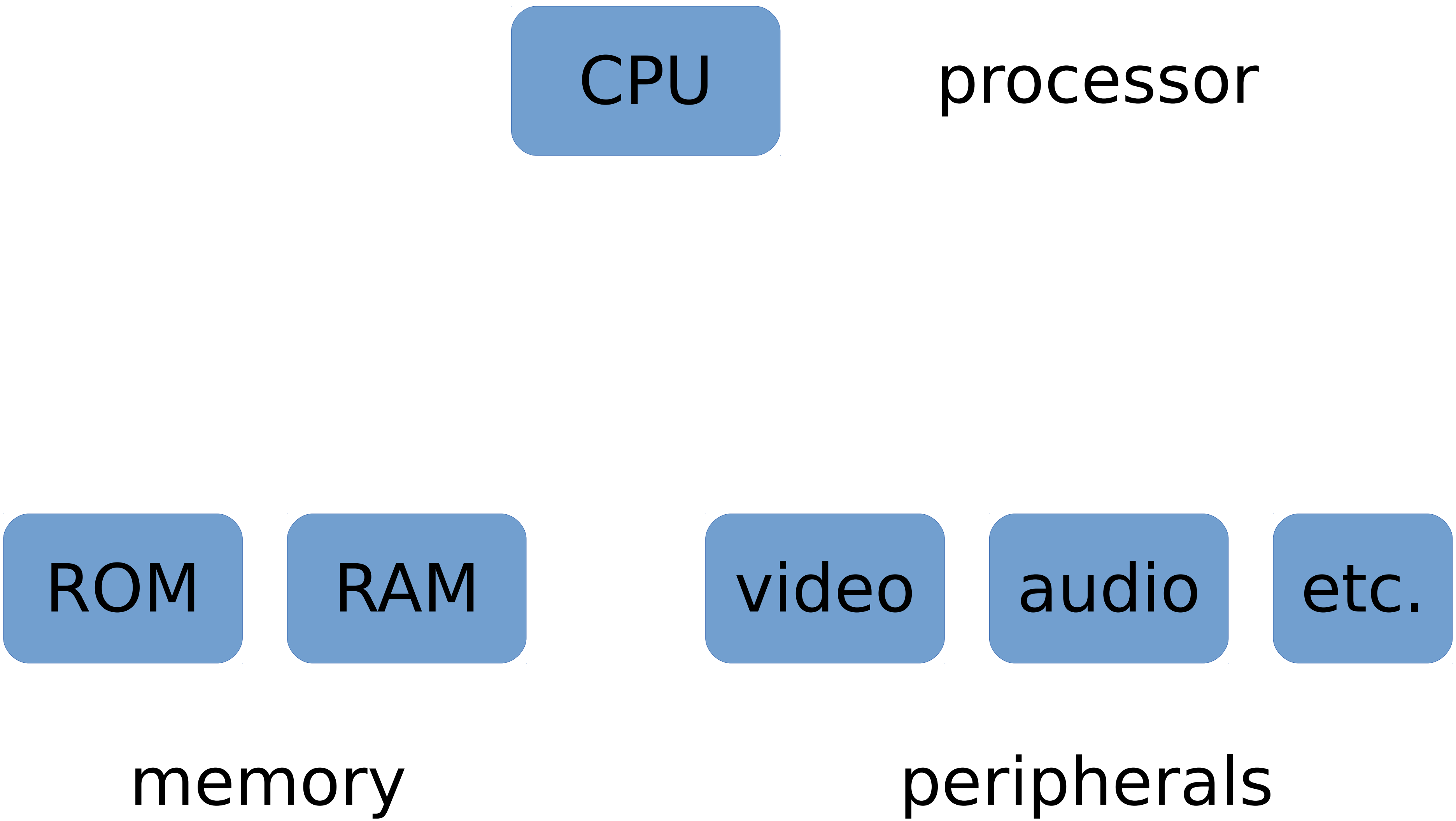
Guest machine: inside



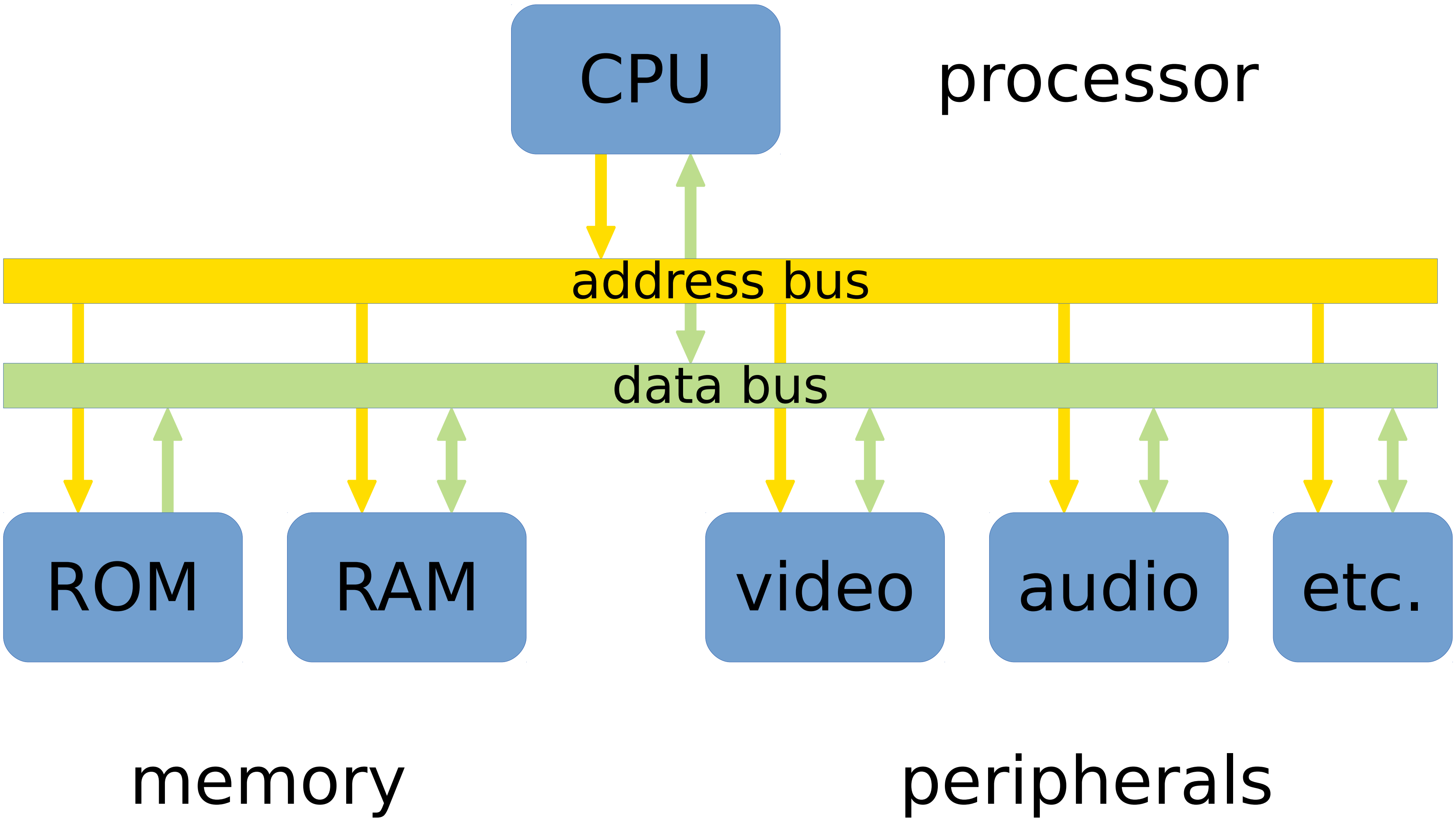
Guest machine: components



Guest machine: components



Guest machine: components



CPU: executing an instruction

- Fetch
  - read instruction from memory
  - increase program counter
- Decode
  - figure out what operation to execute
- Execute
  - perform the actual operation

CPU: interpreter

Fetch, decode, execute instruction every time.

Advantage:

- Simple

Disadvantage:

- Slow

CPU: Just-In-Time compiler

Fetch and decode instructions once, generate host code for execution.

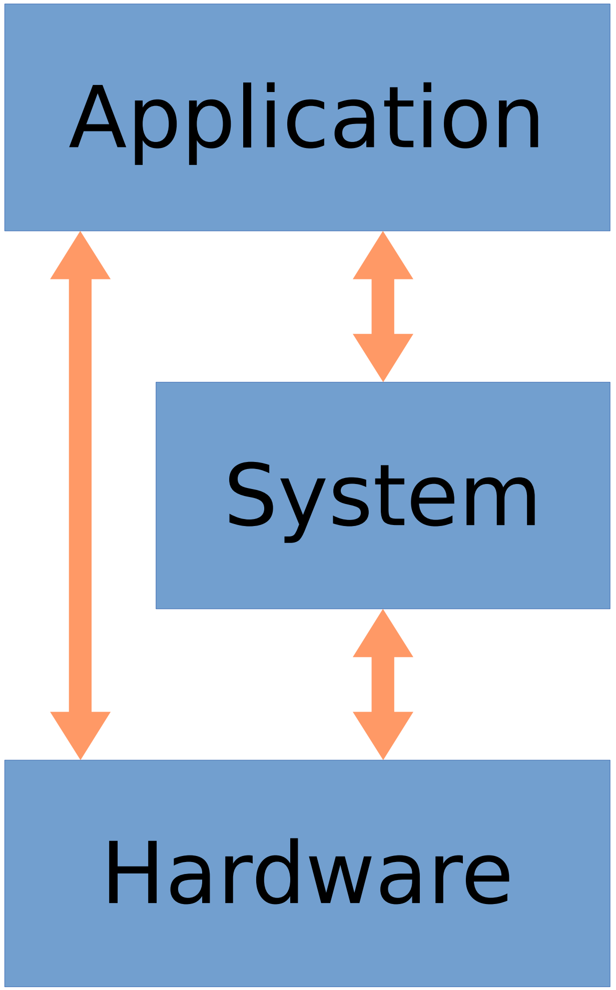
Advantage:

- Fast

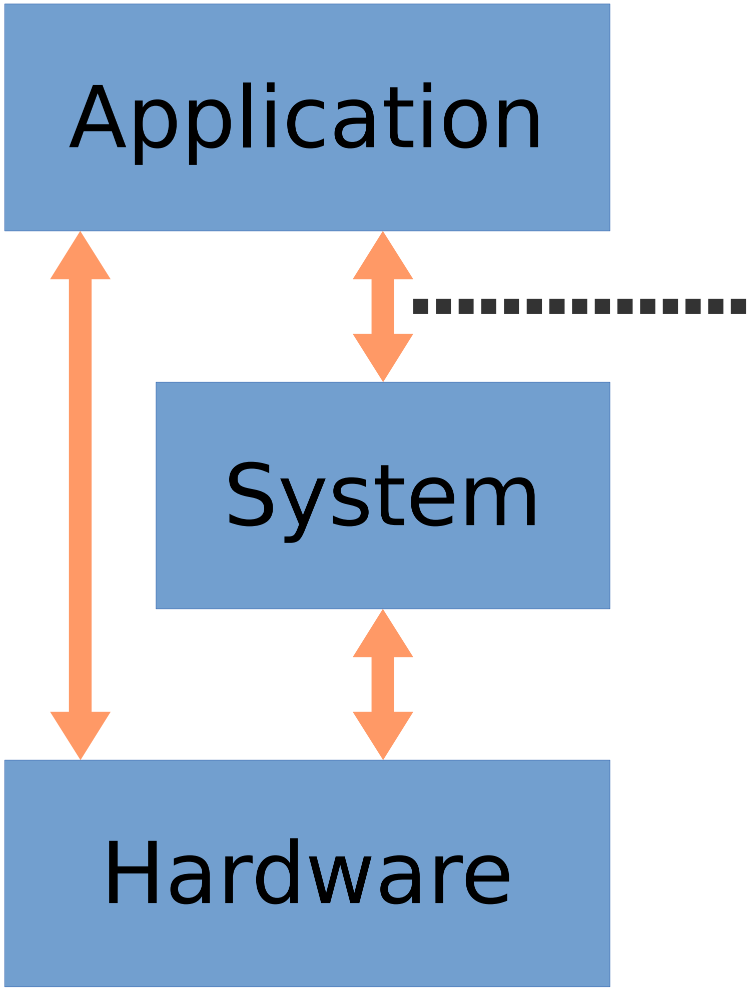
Disadvantages:

- Complex
- Host code generation is not portable

Peripherals



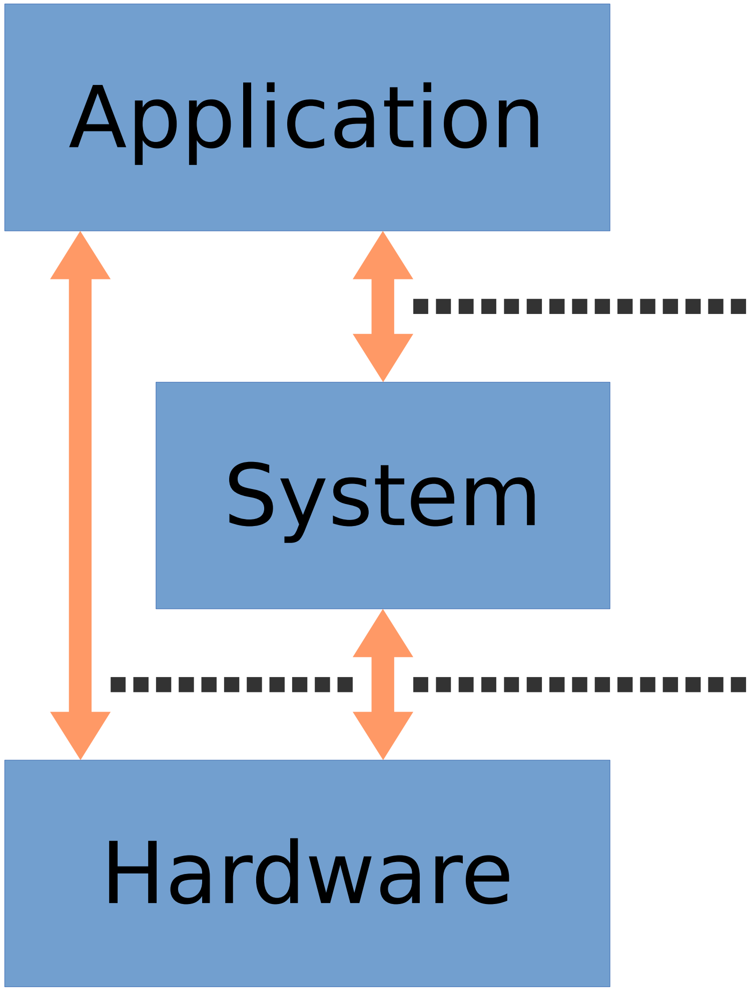
Peripherals



High-level emulation

- fast
- more direct mapping to host system

Peripherals



High-level emulation

- fast
- more direct mapping to host system

Low-level emulation

- accurate
- no assumptions about (use of) system layer

High-level emulation

Interpreter: (Fetch ; Decode ; Execute)*

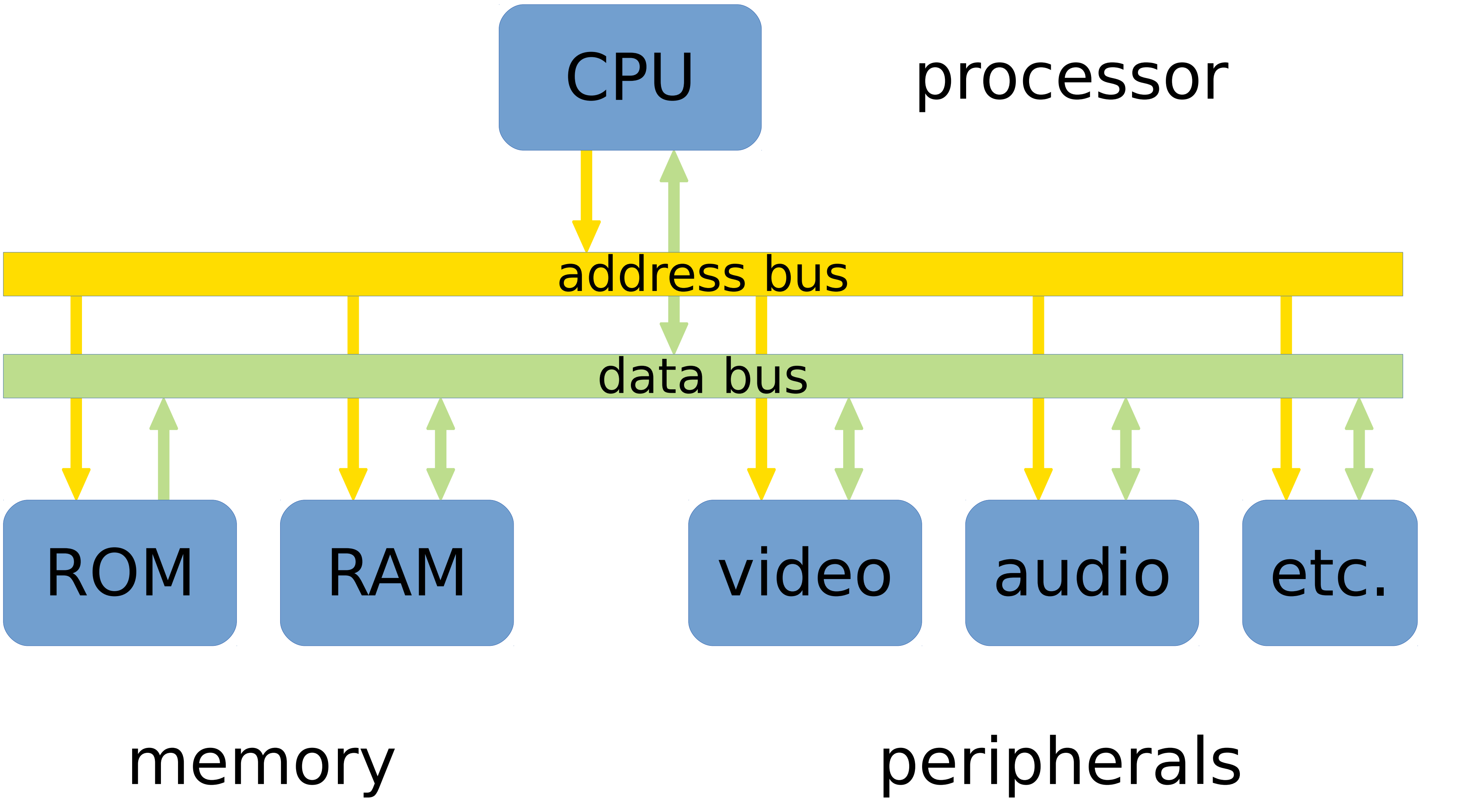
- Intercept at fetch:
  high overhead
- Intercept at decode:
  patch system with illegal instruction

High-level emulation

JIT Compiler: Fetch ; Decode ; Execute*

- Intercept at fetch (code generation): instead of generating code for a system routine, jump to an emulation routine

Low-level: input/output



Low-level: input/output

Instruction reads from or writes to a peripheral

I/O mapped I/O:

- dedicated in/out instructions
- peripheral selected by I/O port number

Memory mapped I/O:

- general load/store instructions
- peripheral selected by special memory address

I/O address mapping

```
<RTC id="Real time clock">
    <io base="0xB4" num="2" type="O"/>
    <io base="0xB5" num="1" type="I"/>
</RTC>

<WD2793 id="Memory Mapped FDC">
    <mem base="0x7FF8" size="4"/>
</WD2793>
```

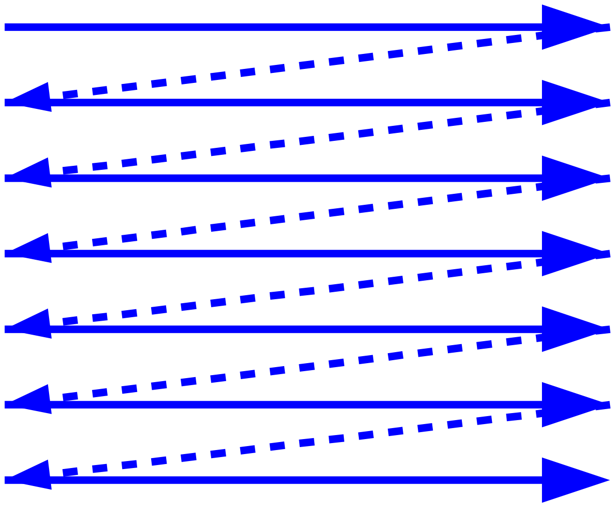
Peripheral emulation

```
Data Device::read(Addr a) {
   return reg[a];
}
```

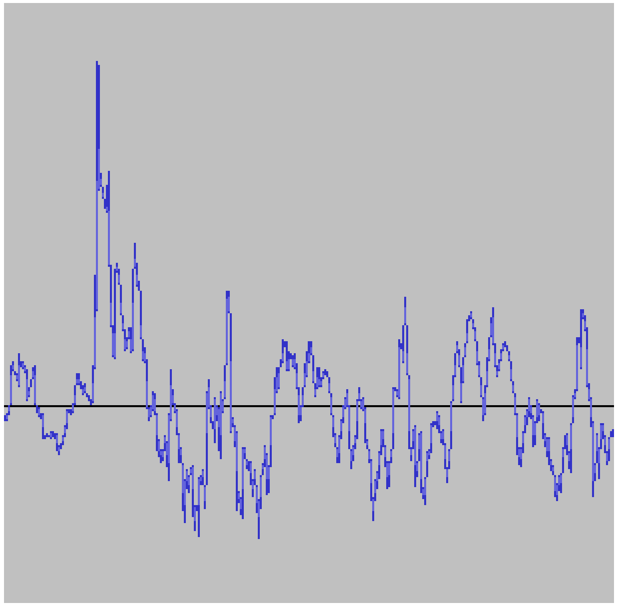
```
void Device::write(Addr a, Data d) {
   reg[a] = d;
}
```

Active peripherals

Change state or produce output in between I/O operations



Video



Audio

Multi-threading

Emulate active peripheral in host thread

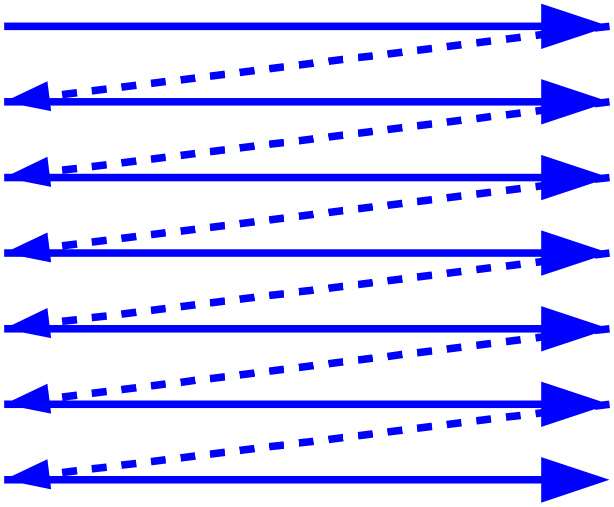
- Low timing accuracy
- Synchronization is expensive

Static interleaved execution

Emulate active peripheral every *N* guest clock ticks

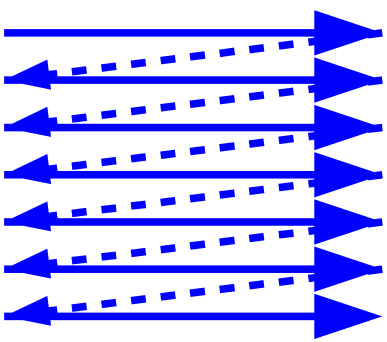
Emulate video chip

- once per frame,
- once per line,
- once per pixel?

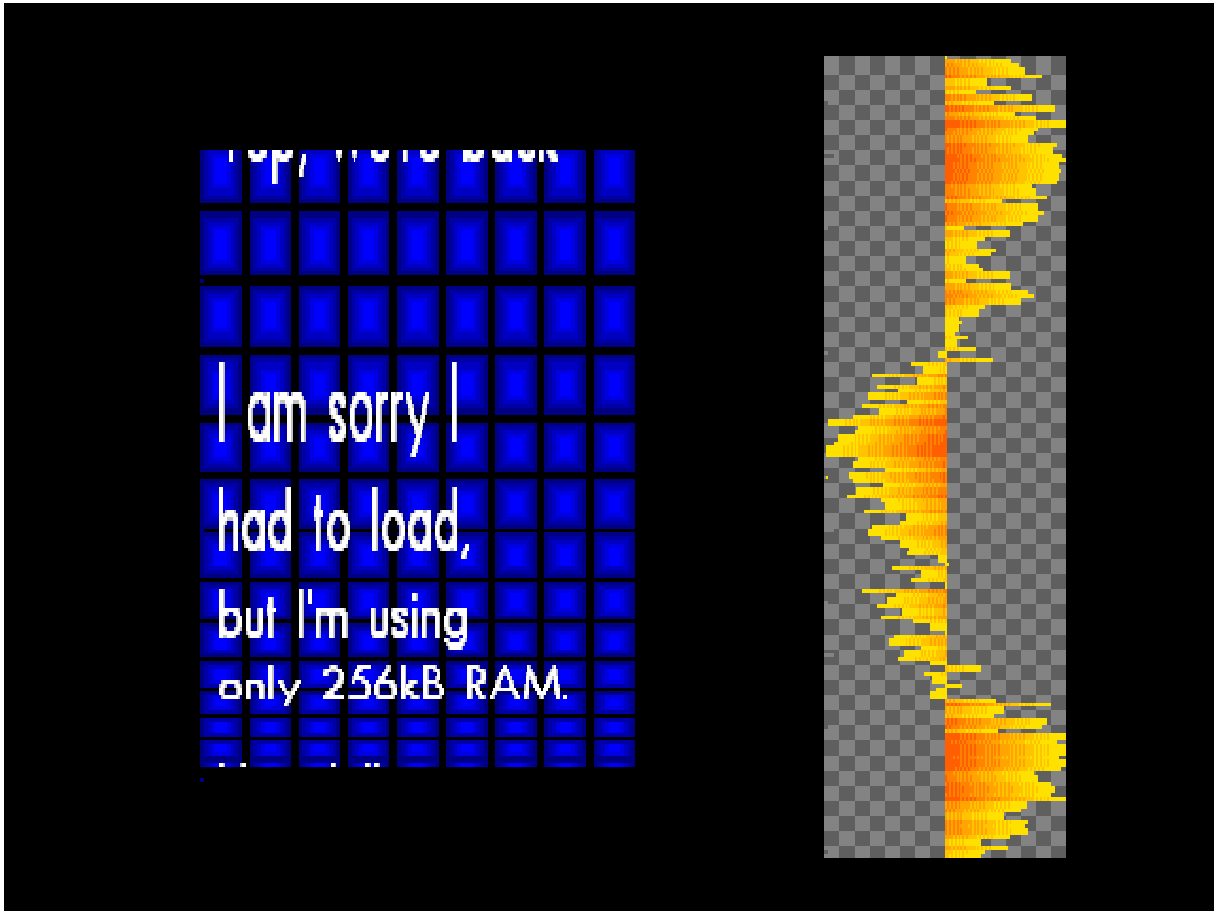


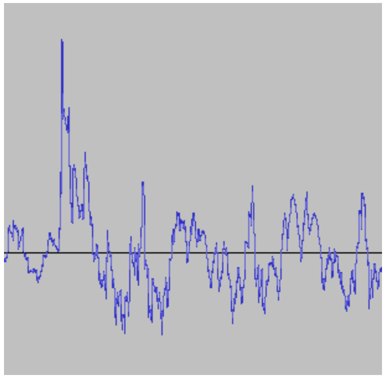
The challenger

Unknown Reality by NOP



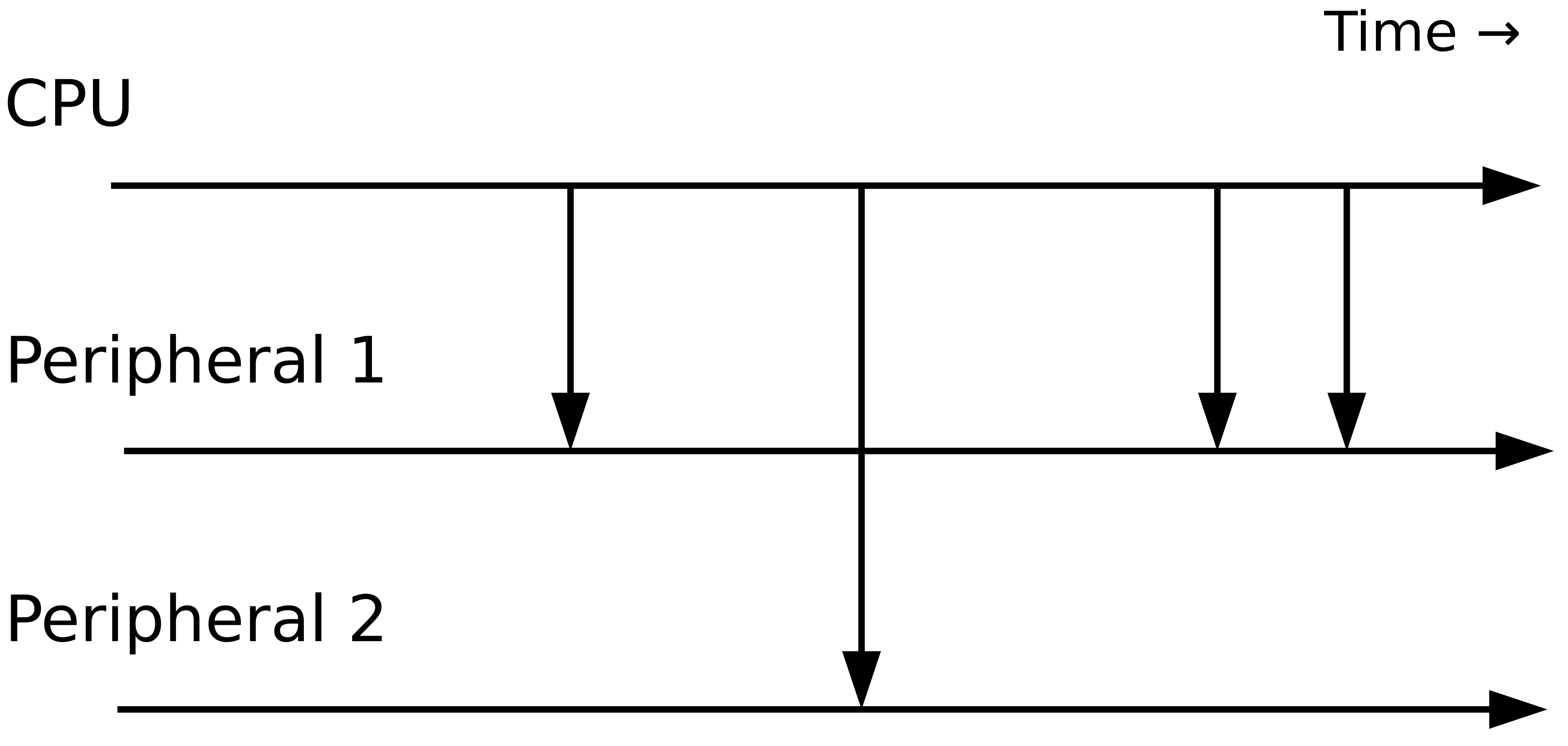
Video



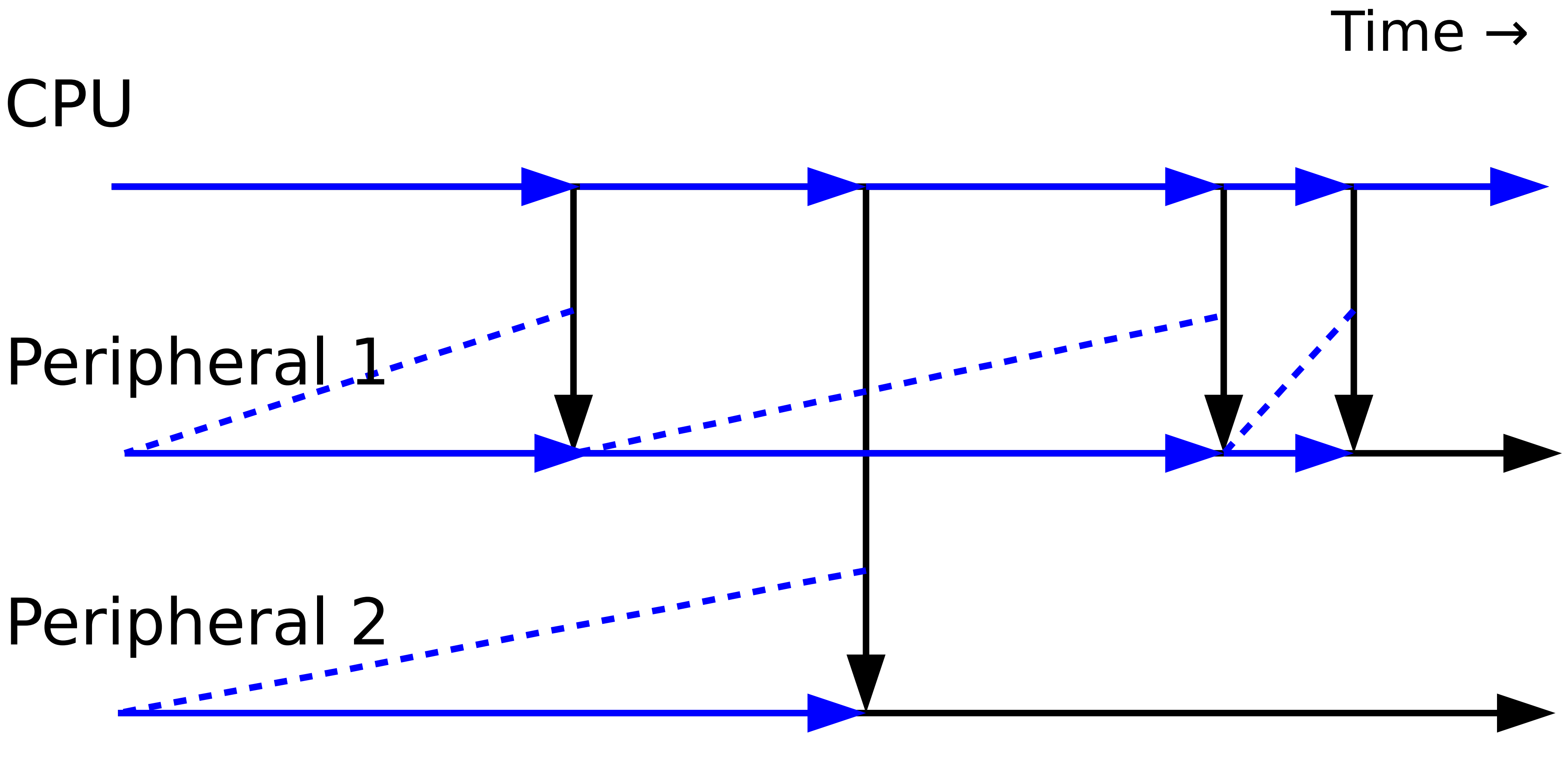


Audio

Timestamped on-demand sync



Timestamped on-demand sync

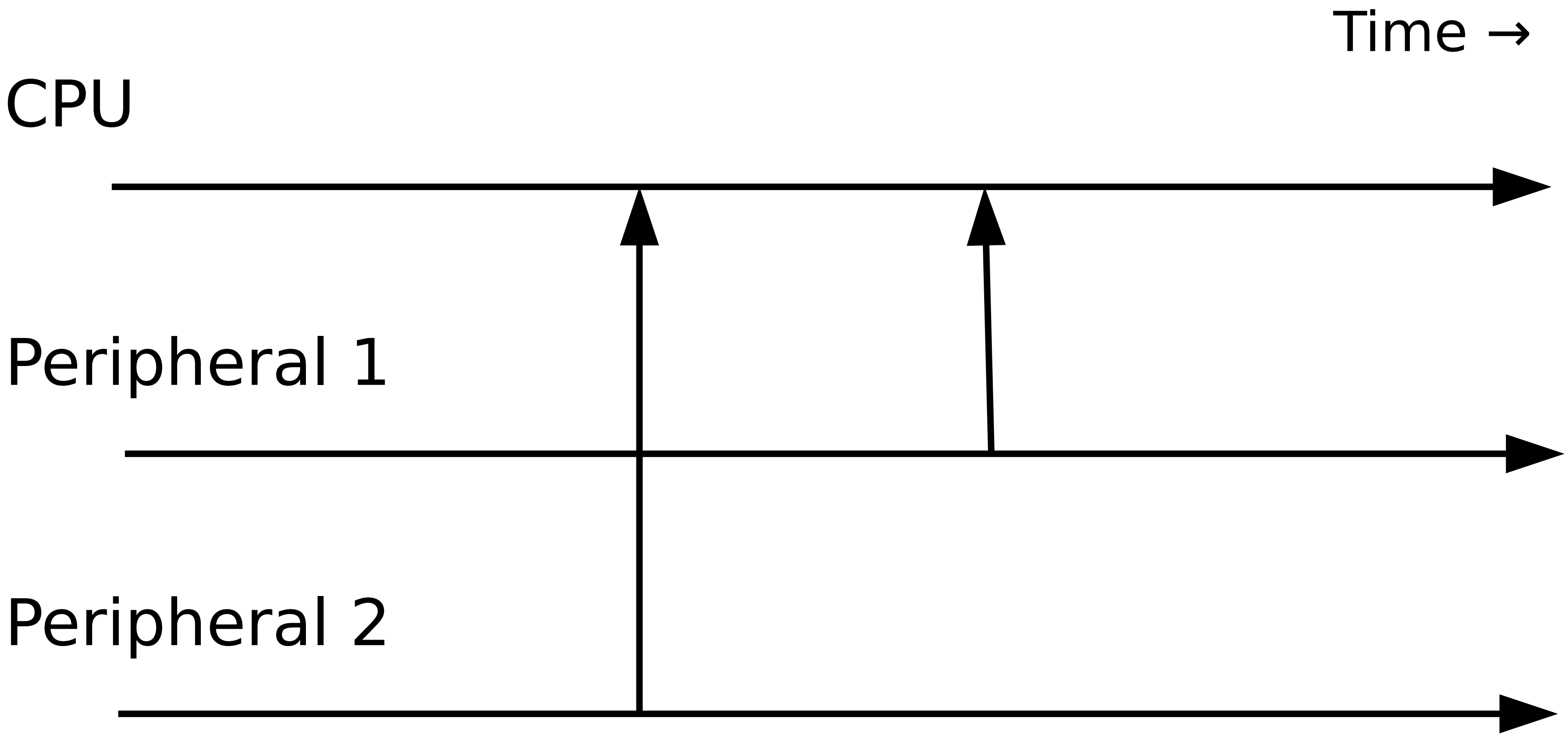


Timestamped on-demand sync

```
Data Device::read(Addr a, Time t) {
   sync(t);
   return reg[a];
}
```

```
void Device::write(Addr a, Data d, Time t) {
   sync(t);
   reg[a] = d;
}
```

TS on-demand sync: interrupts



TS on-demand sync: interrupts

Sync point:

Guest time stamp at which peripheral emulation needs to run

- Peripheral registers sync point at predicted time of interrupt request
- CPU will execute until first sync point, then corresponding peripheral runs and can raise interrupt request

Determinism

old state + input → new state

Deterministic:

New state depends on **nothing else**

Determinism: why?

old state + input → new state

Deterministic emulation is **reproducible**:

- helps with reproducing bugs
- required for accurate emulation
- enables emulator-only features

Determinism: how?

old state + input → new state

Non-determinism comes from host:

- if emulation result depends on host state
- if host timing is relevant

Determinism: how?

old state + input → new state

To eliminate non-determinism:

- model use of host state as an input
- do all timing using guest time stamps

Determinism: random?

old state + input → new state

Pseudo random:

only looks random to casual observer

Real random:

always comes from input

Determinism: replay

old state + input → new state

state snapshot

+

input recording

↓

reproducible replay

Speedruns

Complete a game as fast as possible:

- careful planning
- glitch abuse
- execution skill

Goals:

- challenge
- entertainment

Tool-Assisted Speedruns (TAS)

Complete a game as fast as possible:

- careful planning
- glitch abuse
- ~~execution skill~~ use of emulation tools

Goals:

- challenge
- entertainment
- find limits

Tool-Assisted Speedruns (TAS)

Emulation tools:

- per-frame recording
- re-recording
- disassembly
- luck manipulation
- algorithmic input generation

Tool-Assisted Speedruns (TAS)

Demo

Tool-Assisted Speedruns (TAS)

TASVideos – tasvideos.org

Video archive:

- rendered movies
- input recordings

Community:

- resources
- discussion

Debugger with single-step-back

- Input recording in memory:
  replay to any point in history
- Regular snapshots:
  replay **quickly** to any point in history
- Replay until timestamp:
  replay quickly to any point in history
  and **stop there**
- Combined:
  replay until just before previous
  instruction

Debugger with single-step-back

Demo

Conclusions

- Many different implementation options
- Often trade-off between accuracy and execution speed – but not always!
- Determinism + input recording → replay
- Emulation can provide unique ways to analyze system behavior