

# Virtualization

## Introduction

G. Lettieri

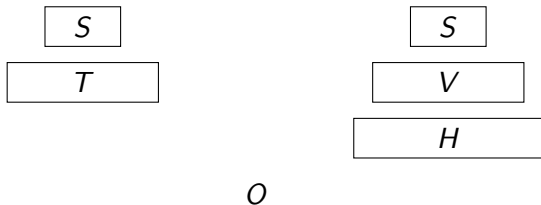
Dipartimento di Ingegneria dell'Informazione  
Università di Pisa

A/A 2014/15

What do people mean when they talk about “virtualization” w.r.t. computers?

- *Anything* you do on computers/Internet
- it cannot be touched  $\implies$  it is not real
- “fake” vs “real” experience

A fake  $V$  behaves like some real  $T$  w.r.t. some observer  $O$ .



Typically (but not necessarily):

- real  $T$  (Target): some hardware
- Virtual  $V$ : made in software (running on hardware  $H$ , Host)
- Observer  $O$ : someone using software ( $S$ ) originally made for  $T$

## Computer Science definition



Typically (but not necessarily):

- real  $T$  (Target): some hardware
- Virtual  $V$ : made in software (running on hardware  $H$ , Host)
- Observer  $O$ : someone using software ( $S$ ) originally made for  $T$

So, we have some software  $S$  that works on  $T$ . We want to replace the real  $T$  with the virtual  $V$  and make  $S$  work like before (to the satisfaction of some observer  $O$ ).

Why do we say “Typically (*but not necessarily*)”?

- $T$  may be software itself, e.g., Windows emulated by Wine on Linux
- $V$  may make use of some hardware designed specifically for the virtualization (we will see several examples of this), or may be entirely in hardware (e.g., old PC peripherals emulated by modern I/O chip-sets)

It is sometimes useful to regard the software  $S$  as the observer, since  $O$ 's interaction with  $T$  is often limited to the software she can run. For a software to “observe” something we mean that it changes behavior (e.g., it prints different outputs) depending on something.

# Why virtualization?

- We don't want to change  $S$
- *and* one or more of:
  - Hardware  $T$  is not available;
  - $V$  is *less expensive* than  $T$
  - $V$  is *more flexible* than  $T$
  - $V$  offers a good protection model for  $S$

## └ Why virtualization?

- We don't want to change  $S$
- and one or more of:
  - Hardware  $T$  is not available;
  - $V$  is less expensive than  $T$
  - $V$  is more flexible than  $T$
  - $V$  offers a good protection model for  $S$

Why don't we just change  $S$ ? It's software, isn't it?

Unfortunately, software may come in forms that are very hard to change:

- huge blobs of machine language;
- very large set of interacting applications and libraries;

Anyways, sometimes we do change  $S$ : this is called *paravirtualization* and we will examine it later.

## └ Why virtualization?

- We don't want to change  $S$
- and one or more of:
  - Hardware  $T$  is not available;
  - $V$  is less expensive than  $T$
  - $V$  is more flexible than  $T$
  - $V$  offers a good protection model for  $S$

The  $T$  hardware may not be available because:

- it no longer exists (historical emulation);
- it does not exist yet (e.g., simulation of new hardware done internally by hardware producers, so that they can start writing and debugging the software while the hardware is still being assembled)
- it never existed at all (where is the Java *Real* Machine?)
- we are already using it for something else (e.g., we have installed Linux on our PC and we want to run some Windows application at the same time).

## └ Why virtualization?

- We don't want to change  $S$
- and one or more of:
  - Hardware  $T$  is not available,
  - $V$  is less expensive than  $T$
  - $V$  is more flexible than  $T$
  - $V$  offers a good protection model for  $S$

Even if  $T$  is in principle available,  $V$  may be a cost-effective solution: less expensive, but still offering sufficiently good performance.

In these scenarios we run the software using less hardware resources than in the original, intended or possible deployment (e.g., less physical memory than actually addressable, less physical machines than number of services).

The idea is that the software is not going to need all the resources at the same time, so we can multiplex the available resources among the active users with (hopefully) little impact on performance.



## └ Why virtualization?

- We don't want to change *S*
- and one or more of:
  - Hardware *T* is not available;
  - *V* is less expensive than *T*
  - *V* is more flexible than *T*
  - *V* offers a good protection model for *S*

Independently from considerations of availability and cost, the fact that virtualization replaces some *hardware* with some equivalent *software* offers more flexibility in the management of the resources and opens the possibility for new functionality:

- live migration: migrating an entire running system from on physical machine to another
- checkpointing the state of a system for fault tolerance
- using applications running on different OSes on the same desktop.

## └ Why virtualization?

- We don't want to change  $S$
- and one or more of:
  - Hardware  $T$  is not available;
  - $V$  is less expensive than  $T$
  - $V$  is more flexible than  $T$
  - $V$  offers a good protection model for  $S$

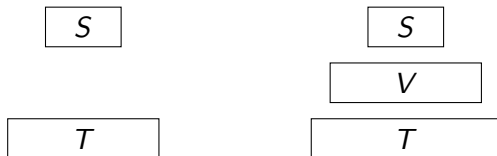
In the last scenario we are not necessarily concerned with availability, cost or flexibility: we want to run several applications on the same hardware and we want to be able to precisely control the way they are allowed (or disallowed) to interact.

Hardware usually gives a clear and well defined interface between separate modules. Two different machines can only communicate through the network that interconnects them, for example, and each machine can access the network only through a network adapter, which has a well defined interface made of registers and shared memory. Since virtualization typically has to reproduce these interfaces, we can leverage them to implement the isolation we need.

Note that there is an overlap here with what OSes traditionally are already meant to provide. Indeed, OS-based solutions alternative to virtualization are pushed forward in this area (containers, jails).

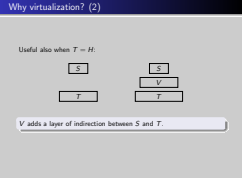
## Why virtualization? (2)

Useful also when  $T = H$ :



$V$  adds a layer of indirection between  $S$  and  $T$ .

## └ Why virtualization? (2)



The motivations involving cost, flexibility and protection can be summarized by noting that they make use of the new layer of indirection that  $V$  interposes between the software  $S$  and the hardware  $T$ .

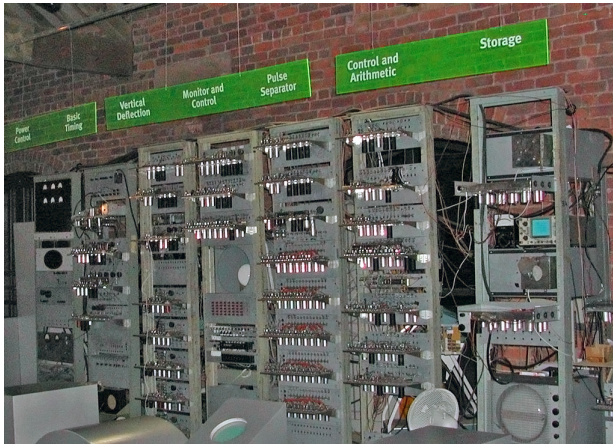
This is the reason why it may make perfect sense to introduce  $V$ , even when  $T$  and  $H$  are exactly the same.

# How to virtualize?

We are going to examine several techniques:

- emulation (Bochs, original JVM)
- binary translation (QEMU, recent JVMs)
- hardware-assisted (KVM, Virtualbox)
- paravirtualization (original Xen)

# The Small Scale Experimental Machine (1948)



**Figure :** A modern replica of the SSEM, aka “Baby”, at the Museum of Science and Industry, Manchester. (credit: Wikipedia)

## └ The Small Scale Experimental Machine (1948)



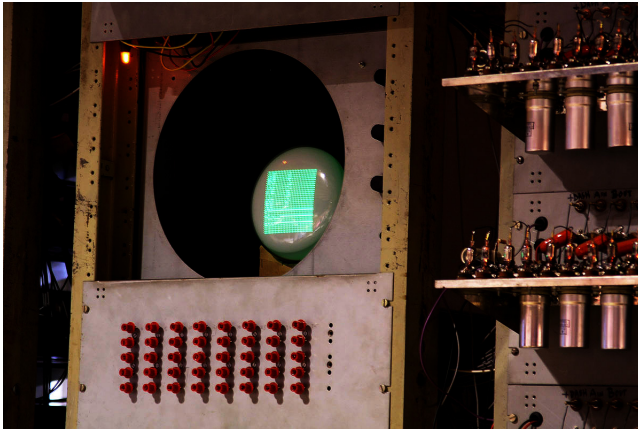
Figure : A modern replica of the SSEM, aka "Baby", at the Museum of Science and Industry, Manchester. (credit: Wikipedia)

We will write an emulator for the SSEM or Baby machine in Manchester. This is the very first Stored Program computer, completed the 21 June 1948 at the University of Manchester under the direction of F.C. Williams and Tom Kilburn.

*Stored Program* means that the program is input and stored in an internal memory, and only then obeyed. Once in memory, it can be read at high speed and it can also modify itself (a feature often used in early times). The Stored Program concept originated in the USA within the ENIAC group, lead by J.P. Eckert and J. Mauchly. It was then disseminated in a draft paper by J. von Neumann and during the very first computer course, in the summer of 1946.

The main obstacle to the actual implementation the stored program idea was to build a device which could store a sufficiently large number of digits and operate at electronic speed. The ENIAC group was working on mercury delay lines, invented by Eckert. Williams and Kilburn came up with a different idea.

# The SSEM CRT output



**Figure :** The CRT output showing the memory contents as a matrix of  $32 \times 32$  big/small dots (credit: Wikipedia)



## └ The SSEM CRT output



Figure The CRT output showing the memory contents as a matrix of 32x32 big/small dots (credit: Wikipedia)

They decided to make computer memories from CRT's, which were already used in radars. The screen of the tube is used as a matrix of dots, each in one of two possible configurations, to store a bit. A bit can be read by trying to change it (a destructive read): if a current is produced, then the stored bit was different from the one we had tried to write. The current was observed using a metal plate that covered the screen. Since the cannon could be deflected to read and write from any desired location in a relatively constant time, this was a RAM. The memory content had to be periodically refreshed, so it was actually a DRAM.

└ The SSEM CRT output

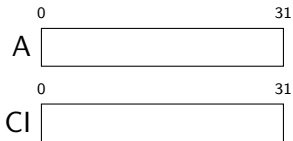
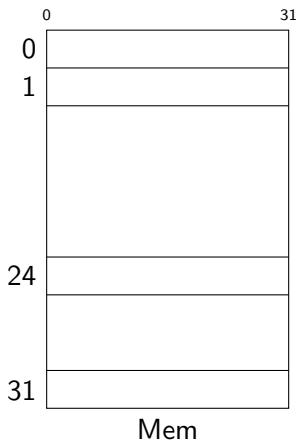


Figure The CRT output showing the memory contents as a matrix of 32x32 big/small dots (credit: Wikipedia)

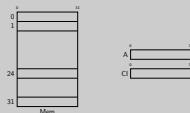
The tube that shows the contents of memory (the one in the picture) is not the one actually used for storage: that one had the metal plate covering it, and was also protected from interferences. However, the display tube is receiving the same signal as the memory one. Note that this was the only output device in the machine. Input was carried on using a keyboard and a set of switches to select a location in memory and write a digit into it.

The Baby was actually built only to test the memory tube (later known as Williams Tube) and was deliberately a very simple machine, for which (apparently) only three programs were ever written (one of them by A. Turing). Once it worked, it was soon expanded into the more practical Manchester Mark I computer.

# The SSEM ISA (1)



## └ The SSEM ISA (1)



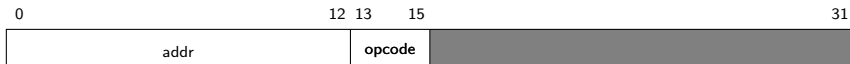
The machine has 32 memory locations 32 bits wide. Each location can store either an instruction or a number. Numbers are integers represented in 2's-complement.

The machine has only one register, the accumulator A.

CI is the program counter. It is always incremented by 1 before fetching an instruction. Since it starts at 0, the (default) entry point is at address 1.

Note the peculiar way of representing the numbers with the least significant bit on the left. The CRT also shows the memory contents in this way.

# The SSEM ISA (2)



opcode	<i>effect</i>	
0	CI	$\leftarrow \text{Mem}[\text{addr}]$
1	CI	$\leftarrow \text{CI} + \text{Mem}[\text{addr}]$
2	A	$\leftarrow -\text{Mem}[\text{addr}]$
3	Mem[addr]	$\leftarrow A$
4,5	A	$\leftarrow A - \text{Mem}[\text{addr}]$
6	<i>if</i> $A < 0$ , $\text{CI} \leftarrow \text{CI} + 1$	
7	<i>halt</i>	

# The SSEM ISA (2)

opcode	effect
0	CI $\leftarrow$ Mem[addr]
1	CI $\leftarrow$ CI + Mem[addr]
2	A $\leftarrow$ -Mem[addr]
3	Mem[addr] $\leftarrow$ A
4,5	A $\leftarrow$ A - Mem[addr]
6	if A < 0, CI $\leftarrow$ CI + 1
7	halt

Note that the accumulator only had a subtracter and that reading from memory reverses the sign. This was done to spare electronic circuits at the expense of execution time (and program storage): you can still do additions via  $a + b = -(-a - b)$ . E.g., to have  $a + b$  in A starting with  $a$  in Mem[20] and  $b$  in Mem[21] we can do:

A	$\leftarrow$ -Mem[20]	read $-a$ in A
A	$\leftarrow$ A - Mem[21]	compute $-a - b$ in A
Mem[22]	$\leftarrow$ A	temporary store of $-a - b$
A	$\leftarrow$ -Mem[22]	read $a + b$ back into A

Note that the conditional branch (opcode 6) just skips a single instruction if the test is true (another 1 will be added to CI before fetching the next instruction).

Note, finally, that the unconditional jumps (opcodes 0 and 1) are always indirect.

# The emulator (1)

```
int32_t Mem[32];
int32_t A;
int32_t CI;

void exec() {
    for (;;) {
        /* advance CI */
        CI++;

        /* fetch the next instruction */
        int32_t PI = Mem[CI];

        /* decode the instruction */
        int32_t opcode = (PI & 0xE000) >> 13;
        int32_t addr = PI & 0x1FFF;
```

## └ The emulator (1)

```
int32_t Mem[32];
int32_t A;
int32_t CI;

void exec() {
    for (;;) {
        /* advance CI */
        CI++;

        /* fetch the next instruction */
        int32_t PI = Mem[CI];

        /* decode the instruction */
        int32_t opcode = (PI & 0xE000) >> 13;
        int32_t addr = PI & 0x1FFF;
```

We use `int32_t` which, if available, is guaranteed to be a 32 bit integer represented in two's-complement (C99 standard).



## The emulator (2)

```
/* execute the instruction */  
switch (opcode) {  
  case 0: CI = Mem[addr];           break;  
  case 1: CI = CI + Mem[addr];      break;  
  case 2: A = -Mem[addr];           break;  
  case 3: Mem[addr] = A;            break;  
  case 4: /* below */  
  case 5: A = A - Mem[addr];        break;  
  case 6: if (A < 0) CI = CI + 1; break;  
  case 7: return; /* terminates emulation */  
}  
}  
}
```

# The (amended) first program

1	00011000000000100000000000000000	$A \leftarrow -\text{Mem}[24]$
2	01011000000001100000000000000000	$\text{Mem}[26] \leftarrow A$
3	01011000000001000000000000000000	$A \leftarrow -\text{Mem}[26]$
4	11011000000001100000000000000000	$\text{Mem}[27] \leftarrow A$
5	11101000000001000000000000000000	$A \leftarrow -\text{Mem}[23]$
6	11011000000000100000000000000000	$A \leftarrow A - \text{Mem}[27]$
7	00000000000001100000000000000000	<i>if</i> $A < 0$ , $\text{CI} \leftarrow \text{CI} + 1$
8	00101000000001000000000000000000	$\text{CI} \leftarrow \text{CI} + \text{Mem}[20]$
9	01011000000000100000000000000000	$A \leftarrow A - \text{Mem}[26]$
10	10011000000001100000000000000000	$\text{Mem}[25] \leftarrow A$
11	10011000000001000000000000000000	$A \leftarrow -\text{Mem}[25]$
12	00000000000001100000000000000000	<i>if</i> $A < 0$ , $\text{CI} \leftarrow \text{CI} + 1$
13	00000000000001110000000000000000	<i>halt</i>
14	01011000000001000000000000000000	$A \leftarrow -\text{Mem}[26]$
15	10101000000000100000000000000000	$A \leftarrow A - \text{Mem}[21]$
16	11011000000001100000000000000000	$\text{Mem}[27] \leftarrow A$
17	11011000000001000000000000000000	$A \leftarrow -\text{Mem}[27]$
18	01011000000001100000000000000000	$\text{Mem}[26] \leftarrow A$
19	01101000000000000000000000000000	$\text{CI} \leftarrow \text{Mem}[22]$
<hr/>		
20	10111111111111111111111111111111	-3
21	10000000000000000000000000000000	1
22	00100000000000000000000000000000	4
23	00000000000000000001111111111111	-262144
24	11111111111111111110000000000000	262143

└ The (amended) first program

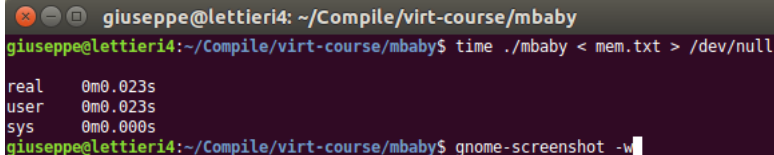
The (amended) first program		
1	0001 10000000001000000000000000000000	A ← -Mem[26]
2	0101 1000000001 10000000000000000000000000	Mem[26] ← A
3	0101 1000000001 10000000000000000000000000	A ← -Mem[26]
4	1101 1000000001 10000000000000000000000000	Mem[27] ← A
5	1110 1000000001 10000000000000000000000000	A ← -Mem[27]
6	1101 10000000000100000000000000000000	A ← A - Mem[27]
7	00000000000001100000000000000000	if A < 0, G ← G + 1
8	0101 1000000001 10000000000000000000000000	C1 ← C1 + Mem[26]
9	0101 10000000000100000000000000000000	A ← A - Mem[26]
10	1001 1000000001 10000000000000000000000000	Mem[25] ← A
11	1001 1000000000 10000000000000000000000000	A ← -Mem[25]
12	00000000000001100000000000000000	if A < 0, C2 ← C2 + 1
13	00000000000001100000000000000000	halt
14	0101 1000000001 10000000000000000000000000	A ← -Mem[26]
15	1010 10000000000100000000000000000000	A ← A - Mem[21]
16	1101 1000000001 10000000000000000000000000	Mem[27] ← A
17	1101 1000000001 10000000000000000000000000	A ← -Mem[27]
18	0101 1000000001 10000000000000000000000000	Mem[26] ← A
19	0110 10000000000000000000000000000000	C1 ← Mem[25]
20	00111111111111111111111111111111	?
21	10000000000000000000000000000000	1
22	00100000000000000000000000000000	4
23	00000000000000000000000000000000	-262144
24	11111111111111111111111111111111	262144

The program finds the the greatest proper divisor  $b$  of a number  $a$ . Initially, word 23 must contain  $-a$  and word 24 must contain  $a - 1$  (first factor to try). The program tries each potential factor from  $a - 1$  to 1 in turn. Division is implemented by repeated subtraction (words 6–8). When the program halts (word 13) we have  $b$  in word 27 and  $-b$  in word 26.

The original program has been lost and then reconstructed from memory. Moreover, this is an amended, slightly improved version.

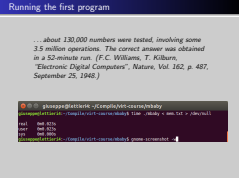
# Running the first program

*... about 130,000 numbers were tested, involving some 3.5 million operations. The correct answer was obtained in a 52-minute run. (F.C. Williams, T. Kilburn, "Electronic Digital Computers", Nature, Vol. 162, p. 487, September 25, 1948.)*

A terminal window titled "giuseppe@lettieri4: ~/Compile/virt-course/mbaby" with standard window controls. The prompt is "giuseppe@lettieri4:~/Compile/virt-course/mbaby\$". The command "time ./mbaby < mem.txt > /dev/null" has been executed. The output shows timing statistics: "real 0m0.023s", "user 0m0.023s", and "sys 0m0.000s". The prompt is now "giuseppe@lettieri4:~/Compile/virt-course/mbaby\$". The next command "gnome-screenshot -w" is partially visible.

```
giuseppe@lettieri4: ~/Compile/virt-course/mbaby
giuseppe@lettieri4:~/Compile/virt-course/mbaby$ time ./mbaby < mem.txt > /dev/null
real    0m0.023s
user    0m0.023s
sys     0m0.000s
giuseppe@lettieri4:~/Compile/virt-course/mbaby$ gnome-screenshot -w
```

## └ Running the first program



The emulator is available at  
<http://lettieri.iet.unipi.it/virtualization/mbaby.tgz>  
Clearly, we are not emulating execution time.

- Model both  $T + S$  and  $V + S$  as *State Machines*:

$\langle T\text{-state}, T\text{-next} \rangle$

$\langle V\text{-state}, V\text{-next} \rangle$

- Define *interp*:  $V\text{-state} \rightarrow T\text{-state}$  (interpretation)
- Agree with  $O$  that she will only look at  $T\text{-states}$  (either directly from  $T$  or from  $V$  through *interp*)
- Require that  $V\text{-next}$  preserves the interpretation.

## └ A formalization

- Model both  $T + S$  and  $V + S$  as State Machines
  - ( $T\text{-state}, T\text{-next}$ )
  - ( $V\text{-state}, V\text{-next}$ )
- Define  $\text{interp}: V\text{-state} \rightarrow T\text{-state}$  (interpretation)
- Agree with  $O$  that she will only look at  $T\text{-states}$  (either directly from  $T$  or from  $V$  through  $\text{interp}$ )
- Require that  $V\text{-next}$  preserves the interpretation.

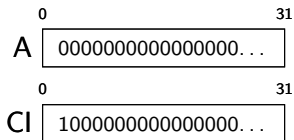
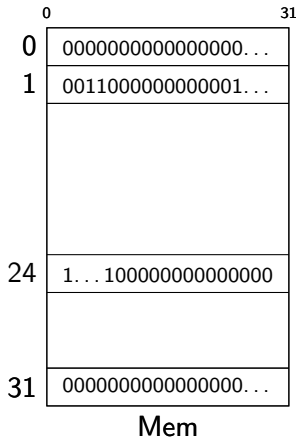
Let us try to generalize what we have done.

The idea is to let the  $T$  system run and take snapshots at some instants. Then we work with *descriptions* of these snapshots. The descriptions do not contain *all* the things that can be observed in the system, but only some features we are interested in. The interval between two consecutive snapshot must last for sufficiently long as to observe a different state, but we are free to choose any interval that lasts longer than that (thus jumping over intermediate states). Therefore, our sequence of descriptions is an abstraction of the real system. It is *this* abstraction that we want to reproduce.

We are assuming *deterministic* state machines for now:  $T\text{-next}$  is a function  $T\text{-next}: T\text{-state} \rightarrow T\text{-state}$  (and the same goes for  $V\text{-next}$ ).

There are several ways to model “final” states. We choose to have  $T\text{-next}(s) = s$  whenever  $s$  is final.

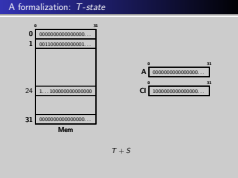
# A formalization: $T$ -state



$$T + S$$



└ A formalization:  $T$ -state

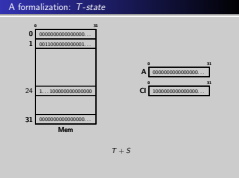


What do we put in each  $T$ -state?

- First of all, it depends on what the observer can see or is interested in. E.g., the observer may want that a given set of programs (maybe all possible programs) produce the exact same output as in the original machine. In the SSEM example, the output is the state of the memory when the programs stops (opcode `0x7`), therefore the observer should at least be able to see the memory contents when the machine stops.
- Then, we may need to include other details. The idea is that from each snapshot description we should be able to predict the next one. This is why we also add A and CI.

Note that the state includes the actual contents of the registers and the memory. For convenience, we consider the state of the machine immediately *after* the increment of CI.

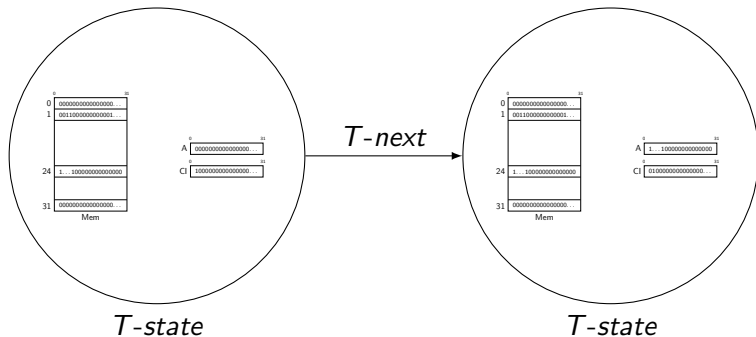
└ A formalization: *T-state*



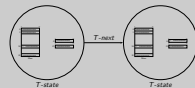
What we put in the description determines the level of detail/abstraction of our emulation. In the SSEM example we have included the accumulator, the program counter and the memory, but we have left out many other things. For instance, the SSEM was a serial machine, processing one bit at a time, but we have not put an indicator of “the current bit within the word” in the state.

As an additional example, we could have put the state on/off of each vacuum tube in the virtual-state. Then, our emulation would have been at the logic level. We have omitted these details from the state because we “feel” that they are not needed, if we only want to reproduce the memory contents. This feeling can be justified more rigorously.

# A formalization: $T$ -transitions



└ A formalization:  $T$ -transitions



What we do in the transitions determines a further level of abstraction: how much do we want to let the  $T\text{-state}$  change before we take another snapshot? In the SEEM we have considered the full execution of single instruction as a step.

# A formalization: *V-states*

```
int32_t Mem[32]; // 0, 0x100C, ...,  
                // [24] 0x3FFFF, ..., 0  
int32_t A;       // 0  
int32_t CI;      // 1  
  
void exec() {  
    for (;;) {  
        CI++;  
⇒      int32_t PI = Mem[CI];  
  
        int32_t opcode = (PI & 0xE000) >> 13;  
        int32_t addr = PI & 0x1FFF;  
  
        switch (opcode) {  
        case 0: CI = Mem[addr];          break;  
        case 1: CI = CI + Mem[addr];      break;  
        case 2: A = -Mem[addr];           break;  
        case 3: Mem[addr] = A;            break;  
        case 4:                           break;  
        case 5: A = A - Mem[addr];         break;  
        case 6: if (A < 0) CI = CI + 1;    break;  
        case 7: return;  
        }  
    }  
}
```

└ A formalization: *V-states*

```

inc32_p Mem[32]; // 0x0000
inc32_p A; // 0x00000000...0
inc32_p C1; // 1

void main() {
    Mem[0] = 1;
    C1 = 0;

    inc32_p Pl = Mem[C1];

    // ...

    inc32_p Pl = Mem[C1];

    inc32_p membase = (Pl & 0xFFFF) * 16;
    inc32_p addr = Pl & 0xFFFF;

    while (true) {
        case 0: C1 = Mem[addr]; break;
        case 1: C1 = C1 + Mem[addr]; break;
        case 2: A = Mem[addr]; break;
        case 3: Mem[addr] = A; break;
        case 4: A = A + Mem[addr]; break;
        case 5: A = A + C1 - C1 + 1; break;
        case 7: return;
    }
}

```

The *V-state* is the state of our emulator program, including the contents of all variables (here written in the comments) and the current execution point (here marked by the double arrow). In the example, we are considering the moment in time between the increment of C1 and the assignment to Pl.

# A formalization: *V-state* interpretation

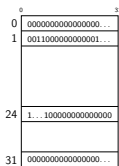
```
int32_t Mem[32]; // 0, 0x00C..., [24] 0x3FFFF, ..., 0
int32_t A; // 0
int32_t CI; // 1

void exec() {
  for (;;) {
    CI++;
    => int32_t PI = Mem[CI];

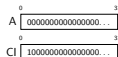
    int32_t opcode = (PI & 0xE000) >> 13;
    int32_t addr = PI & 0x3FFF;

    switch (opcode) {
      case 0: CI = Mem[addr]; break;
      case 1: CI = CI + Mem[addr]; break;
      case 2: A = -Mem[addr]; break;
      case 3: Mem[addr] = A; break;
      case 4:
      case 5: A = A - Mem[addr]; break;
      case 6: if (A < 0) CI = CI + 1; break;
      case 7: return;
    }
  }
}
```

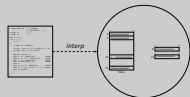
*interp*



Mem



└ A formalization:  $V$ -state interpretation



The equivalence between a virtual-state and a target-state is given by an  $interp$  map, such that

$$interp: V\text{-state} \rightarrow T\text{-state}$$

(answering the question: “What  $T$ -state does this  $V$ -state is equivalent to?”). Note that each  $V$ -state is equivalent to just one  $T$ -state, but each  $T$ -state may be equivalent to several  $V$ -state, i.e., we do not require  $interp$  to be 1-to-1. For simplicity, however, we do require  $interp$  to be onto: the virtual system must have a way to represent any target state.



# A formalization: V-transitions

*V-next*

```
int32_t Mem[32]; // 0, 0x100C, ...,
int32_t A; // [24] 0x3FFF, ..., 0
int32_t Cl; // 1

void exec() {
  for (;;) {
    Cl++;
    =>
    int32_t Pi = Mem[Cl];

    int32_t opcode = (Pi & 0xE000) >> 13;
    int32_t addr = Pi & 0x1FFF;

    switch (opcode) {
    case 0: Cl = Mem[addr]; break;
    case 1: Cl = Cl + Mem[addr]; break;
    case 2: A = -Mem[addr]; break;
    case 3: Mem[addr] = A; break;
    case 4:
    case 5: A = A - Mem[addr]; break;
    case 6: if (A < 0) Cl = Cl + 1; break;
    case 7: return;
    }
  }
}
```

*V-state*

```
int32_t Mem[32]; // 0, 0x100C, ...,
int32_t A; // [24] 0x3FFF, ..., 0
int32_t Cl; // 1

void exec() {
  for (;;) {
    Cl++;
    int32_t Pi = Mem[Cl];
    =>
    int32_t opcode = (Pi & 0xE000) >> 13;
    int32_t addr = Pi & 0x1FFF;

    switch (opcode) {
    case 0: Cl = Mem[addr]; break;
    case 1: Cl = Cl + Mem[addr]; break;
    case 2: A = -Mem[addr]; break;
    case 3: Mem[addr] = A; break;
    case 4:
    case 5: A = A - Mem[addr]; break;
    case 6: if (A < 0) Cl = Cl + 1; break;
    case 7: return;
    }
  }
}
```

...

```
int32_t Mem[32]; // 0, 0x100C, ...,
int32_t A; // [24] 0x3FFF, ..., 0
int32_t Cl; // 2

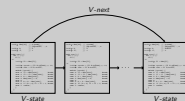
void exec() {
  for (;;) {
    Cl++;
    =>
    int32_t Pi = Mem[Cl];

    int32_t opcode = (Pi & 0xE000) >> 13;
    int32_t addr = Pi & 0x1FFF;

    switch (opcode) {
    case 0: Cl = Mem[addr]; break;
    case 1: Cl = Cl + Mem[addr]; break;
    case 2: A = -Mem[addr]; break;
    case 3: Mem[addr] = A; break;
    case 4:
    case 5: A = A - Mem[addr]; break;
    case 6: if (A < 0) Cl = Cl + 1; break;
    case 7: return;
    }
  }
}
```

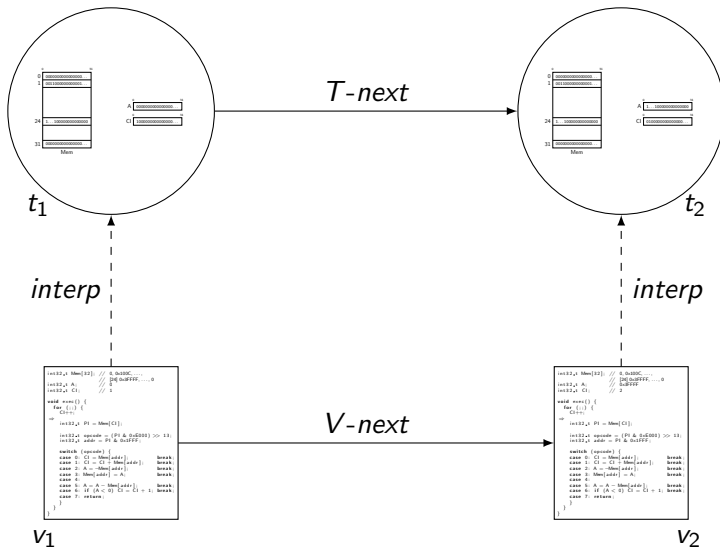
*V-state*

└ A formalization:  $V$ -transitions

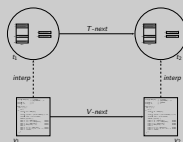


A  $V$ -next transition generally involves several transitions in our emulation program. We only consider a new  $V$ -state when the program state is again interpretable as a  $T$ -state. In our example, this only happens when the execution returns to the point between the increment to  $CI$  and the assignment to  $PI$ . What happens in-between does not have to correspond to anything in the target system: the observer will not see the intermediate states, and we are free to do anything we wish, as long that the “stepwise correctness” property holds.

# A formalization: stepwise correctness



└ A formalization: stepwise correctness



Assume the target system starts in a  $T$ -state state  $t_1$ . We find a  $V$ -state  $v_1$  whose interpretation is  $t_1$  (there must be at least one, since *interp* is onto). Assume  $T\text{-next}(t_1) = t_2$ . Then  $V\text{-next}$  must be implemented such that  $V\text{-next}(v_1)$  is a state  $v_2$  whose interpretation is exactly  $t_2$ . We start with “equivalent” states ( $t_1$  and  $v_1$ ), we end up (after a single step in both machines) with two new equivalent states ( $t_2$  and  $v_2$ ). If this property holds for any starting states, then the equivalence will be preserved for the entire execution.

Since  $O$  is only permitted to look at  $T$ -states, she will not be able to distinguish the target system from the virtual system.