Virtual Hardware

or Virtue and Virtuality

John Kozak

jk@xylema.org

Introduction

Project of six years ago

Source lost

All I have is memories

Trip around dusty corners

like win95, VxDs, parallel ports

Problem and Solution

- Problem: how to talk to a non-existent peripheral?
- Solution: use a non-existent CPU!
- Want a scriptable emulator

Emulator Types

- Hypervisor
  - *e.g.* VM/370, VMware
- Interpreter
  - *e.g.* bochs, usim
- Dynamic Translation
  - *e.g.* FX!32, Dynamo

Emulator Issues

- Emulating CPU is relatively easy
  - well and accurately documented
  - (N.B. i386 presents special challenges for hypervisors)
- Emulating peripherals can be horrible
  - underspecified or mis-implemented
  - big variance in implementations

Emulators: PC on PC

- VMware
  - hypervisor
  - fast, solid
  - closed, no SDK
- bochs
  - interpreter
  - slower, less stable/complete
  - free software
- others?
  - not then...

bochs.sf.net Blurb

Bochs is a highly portable open source IA-32 (x86) PC emulator written in C++, that runs on most popular platforms. It includes emulation of the Intel x86 CPU, common I/O devices, and a custom BIOS. Currently, Bochs can be compiled to emulate a 386, 486, Pentium, Pentium Pro or AMD64 CPU, including optional MMX, SSE, SSE2 and 3DNow instructions.

Bochs is capable of running most Operating Systems inside the emulation including Linux, Windowsfi 95, DOS, and Windowsfi NT 4. Bochs was written by Kevin Lawton and is currently maintained by this project.

Bochs can be compiled and used in a variety of modes, some which are still in development. The'typical' use of bochs is to provide complete x86 PC emulation, including the x86 processor, hardware devices, and memory. This allows you to run OS's and software within the emulator on your workstation, much like you have a machine inside of a machine. For instance, let's say your workstation is a Unix/X11 workstation, but you want to run Win'95 applications. Bochs will allow you to run Win 95 and associated software on your Unix/X11 workstation, displaying a window on your workstation, simulating a monitor on a PC.

bochs - My Views

- great for systems programming
- performance not up to all production uses, yet
- really odd code
  - `.cc` files
  - premature optimisations ‘R’ us
  - bizarre config system
- very impressive piece of work, but really wants to be a library

SCM - why?

- ran on windows with reasonable FFI (SCM/w)
- clean C/scheme interface
  - interpreter
  - **`setjmp`** / **`longjmp`** and stack-copying
  - conservative GC
  - native thread support (avoids *synchromesh* problem)

Making bochs Scriptable from SCM

- disable various bochs optimisations
- build bochs as a library (DLL)
- call via FFI

Driving bochs from Scheme

```
(define (make-single-byte-device! port name)
  (letrec ((byte   0)
           (writer (lambda (addr value len)
                     (set! byte value)))
           (reader (lambda (addr len)
                     byte)))
    (register-iowrite-handler writer port name 1)
    (register-ioread-handler reader port name 1)))


(make-single-byte-device! 384 "demo s/b/d 1")
(make-single-byte-device! 385 "demo s/b/d 2")
```

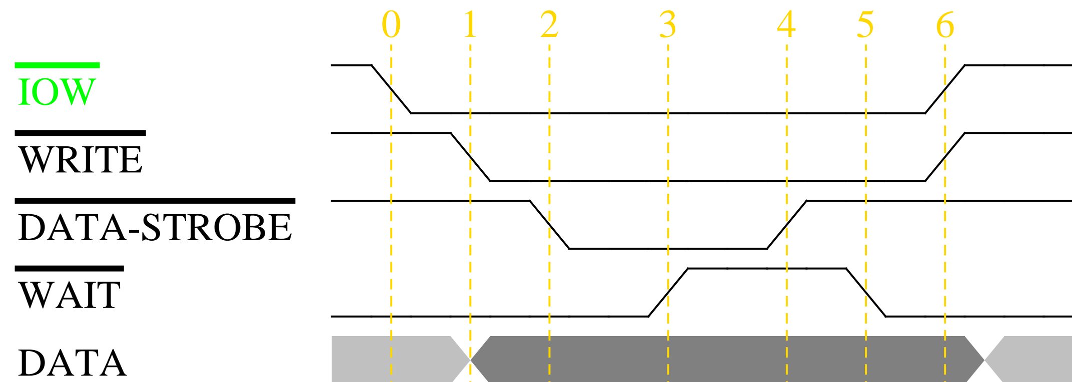
Other Uses

- scriptable low-level debugger
- higher-level logic analyser
- security analysis
- fault emulation
- randomised testing

Notations for Hardware

- how to describe hardware?
  - VHDL
  - various nice formal methods offerings
  - timing diagrams are what you get in practice

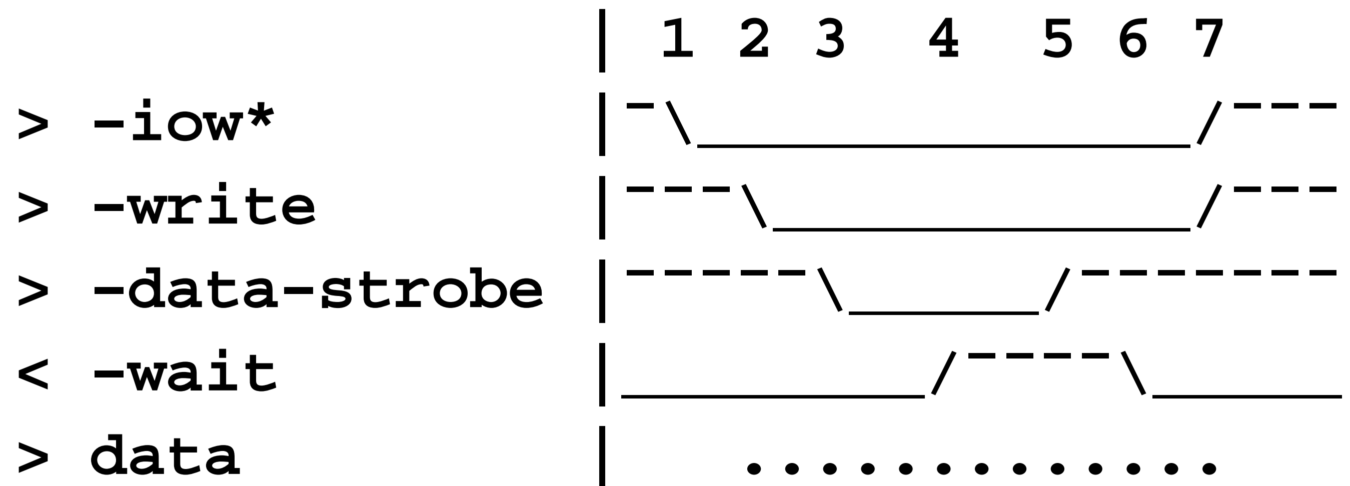
Timing Diagrams



ASCII Art Timing Diagrams

name: EPP-data-write

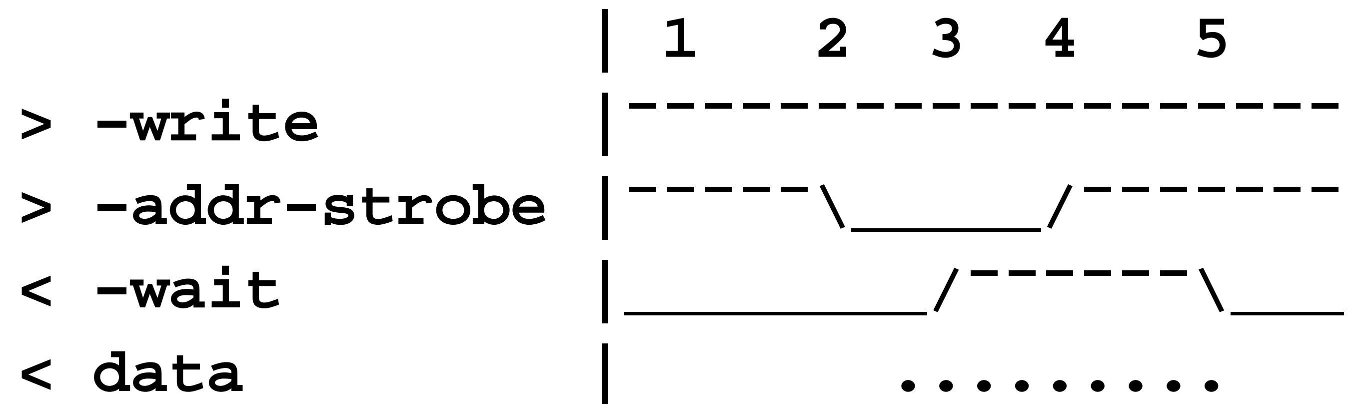
spec: EPP



ASCII Art Timing Diagrams

name: EPP-address-read

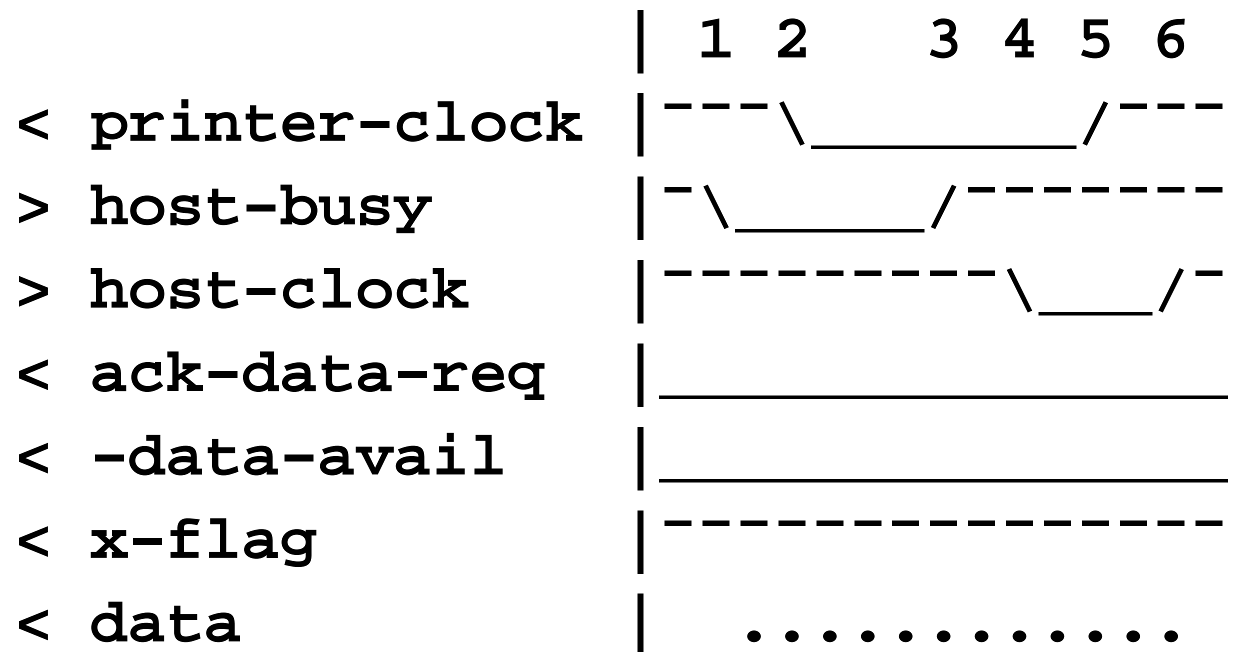
spec: EPP



ASCII Art Timing Diagrams

```
name: byte-mode-read
spec: SPP

timing: 1->2 0   35
timing: 3->4 0 1000
timing: 6->7 0 1000
```



Working with ASCII Art Timing Diagrams

- Easy to parse
- Easy to work with
  - emacs mode?
  - emacs as logic analyser??
- Declarative specification of behaviour/protocol
- Produce pretty pictures via postscript

Deriving State Machines

Each column of a timing diagram defines the state of an interface at a point in time.

By matching up the initial and final states of each diagram we can build a state machine for the interface.

Example here is ’monadic’, though

Inferring Behaviour

- *Post hoc, ergo propter hoc*
  - Extremely useful principle...
  - ... as endorsed by David Hume
  - Can infer lots about behaviour, but do need to annotate with code.

Code Annotation for Timing Diagrams

- glue
  - to emulated system
  - to emulated peripheral
- 1-for-*n* cases
  - *e.g.* **`data`** in previous timing diagrams
- exceptional conditions
  - *e.g.* timeouts
  - *e.g.* contention on bi-directional line

Doing It Again Today

- would probably use:
  - mzscheme
  - qemu

Last Slide Brought To You By:

```
(define (go-demo)
  (let* ((frame (instantiate frame% ("#x80") (min-height 80) (min-width 80)))
         (2-lcd (instantiate 2-lcd% (frame))))
    (send frame show #t)
    (thread (lambda ()
              (qemu:startup
               (lambda ()
                 (qemu:register-ioport-read  385 1 1
                                             (lambda (a)
                                               (send 2-lcd get-value)))
                 (qemu:register-ioport-write 128 1 1
                                             (lambda (a d)
                                               (send 2-lcd set-value! d))))
               "-cdrom" "fdbootcd.iso")))))
```