ILC 2005

LISP in an Electronic Design application

Engineering Numbers

- Engineers use numbers like 1k – meaning 1000, or 3.3u meaning 3.3e-6.
  - Lisp doesn't know about these numbers:

  -
    ```
    (* 1k 3.3u)
    Error: Attempt to take the value of the unbound variable `1k'.
    [condition type: unbound-variable]
    ```

Engineering Numbers

- In Common LISP it is not clear how to have LISP read these numbers (it may not be possible within the spec.)
    - In Franz Allegro LISP we can wrap the reader functions and check for a symbol that is actually intended to be a number:
    - (* 1k 3.3u) => 3.3M

Refer to the file "real-eng.lisp" if you would like the details

Engineering Numbers

- We can define a new format directive to print these kinds of numbers
  - (format t "~u ~s" (* 1230.0 3.5e-4) (* 1230.0 3.5e-4))
    430.5m 0.4305
  - (format t "~u ~u" -20db 3%)
    100.0m 30.0m

Refer to the file "formatu.lisp" if you would like the details

Logarithmic Iteration

- Many engineering tasks require analysis the log domain. We can add a new iteration path to loop:
    - (loop for i being log from 1k to 10k dec 5 collect (format nil "~4u" i))
      ("1K" "1.585K" "2.512K" "3.981K" "6.31K" "10.0K")

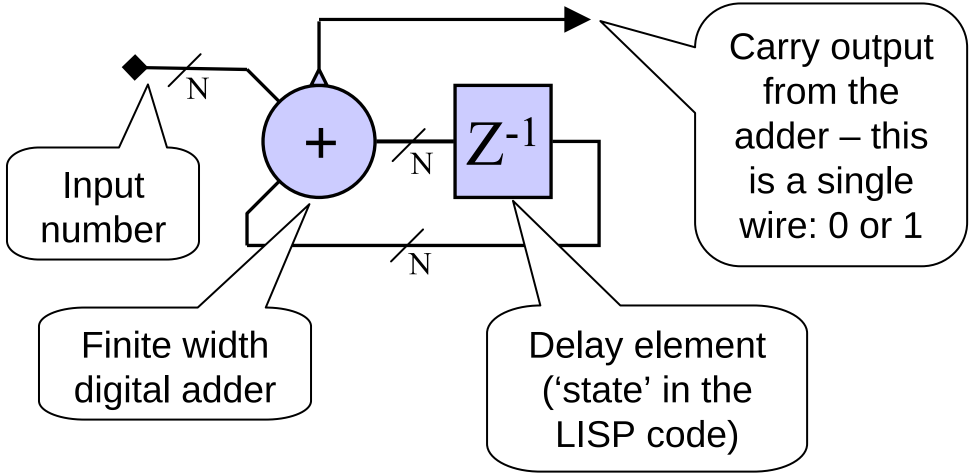
Refer to the file "log-loop-path.lisp" if you would like the details

Modeling Digital Hardware

- Verilog and VHDL – special purpose languages - are used to model hardware – so why use LISP?
- => Its faster and more flexible
  - Less time in the whole process from “I have a new idea to try” to “I have it running on a chip”
  - => It’s faster in LISP because advanced test and verification functions can be abstracted and they are generic.

Modeling a ΣΔ modulator

- This is what engineers call a 1st order Sigma-Delta Modulator (ΣΔ) – it's easy to understand – it is an adder that overflows:


N
N
N
+
Z^{-1}
Input number
Finite width digital adder
Delay element ('state' in the LISP code)
Carry output from the adder – this is a single wire: 0 or 1

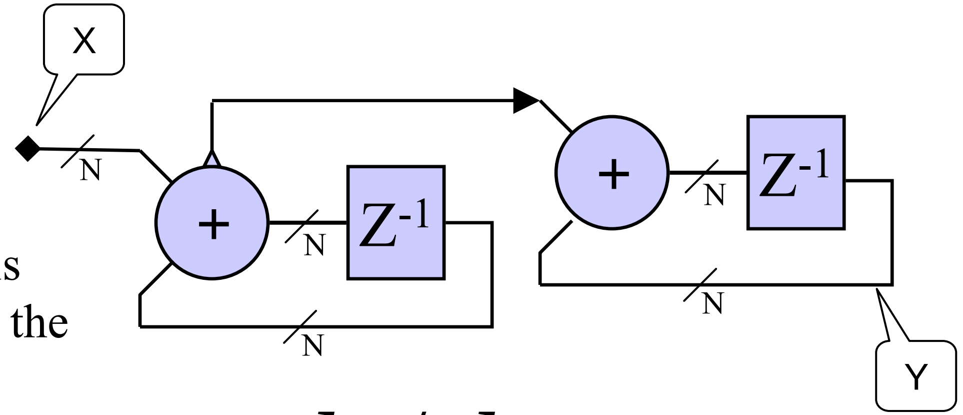
Modeling a ΣΔ modulator

- This is the LISP code:
  - It builds a function to represent the modulator which it returns
  - ‘state’ is the lamba bound state variable
  - It returns two values, 1 and -1

```
(defun sd1 (&key (bits 8))
  (let ((state 0)
        (max (ash 1 bits)))
    #'(lambda (x)
        (if (when (>= (incf state x) max)
              (decf state max))
            1
           -1))))
```

ΣΔ modulators can solve equations

- A ΣΔ modulator can solve differential equations…
  - the rate of occurrence of overflow from the first ΣΔ is proportional to X, therefore the rate of increase of Y is also proportional to X.



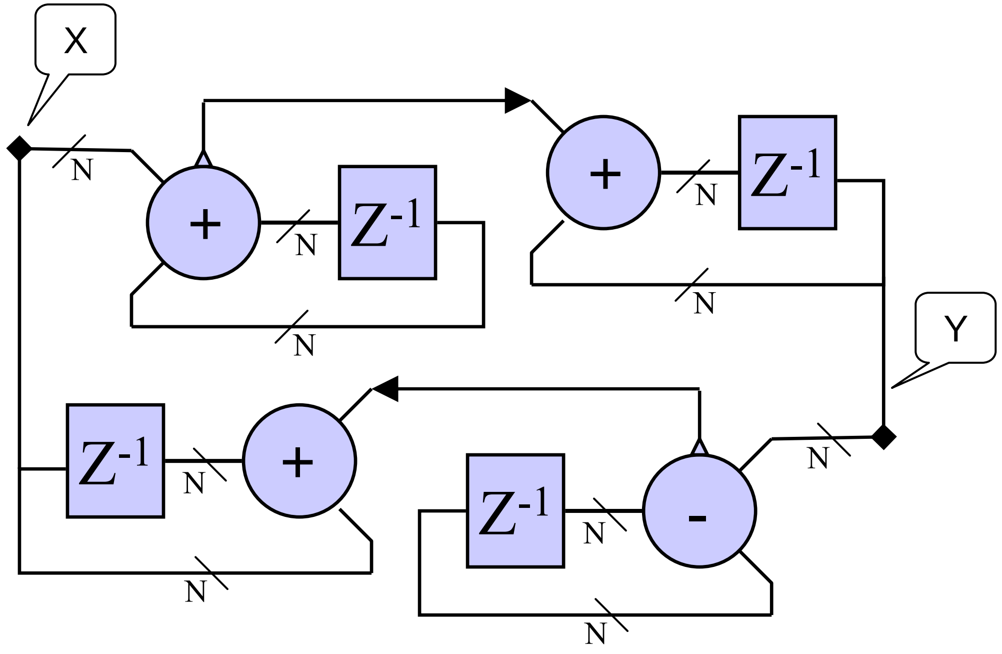
$$dy/dt = x$$

Example system with ΣΔ modulators

- ..and so can be made to model complex systems:

$$d^2y/dt^2 = -y$$

  – This one will create a quadrature oscillator – a sine and cosine wave



Example system with ΣΔ modulators

- This is the LISP code for that system.
  - Note it returns the function to model the system

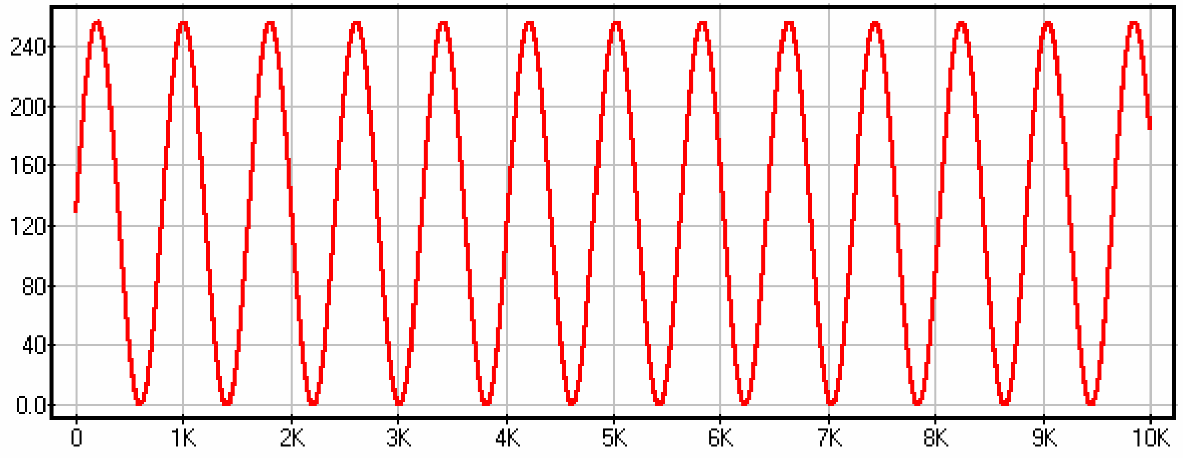
$$d^2y/dt^2 = -y$$

```
(defun osc1 (&key (bits 8))
  (let ((sin-sd (sd1 :bits bits))
        (cos-sd (sd1 :bits bits))
        (max (ash 1 bits))
        (cos (ash 1 bits))
        (sin (ash 1 (1- bits))))
    #'(lambda ()
        (psetq
          cos (max 0 (min max (- cos (funcall sin-sd sin))))
          sin (max 0 (min max (+ sin (funcall cos-sd cos)))))
        (values sin cos))))
```

Example system with ΣΔ modulators

- We can easily test it:

```
(loop
  repeat 10k
  with osc = (osc1)
  collect (funcall osc) into y
  finally (plot y))
```



Real example: Audio digital filter

No approximations or simplifications – this is a complete first order filter*

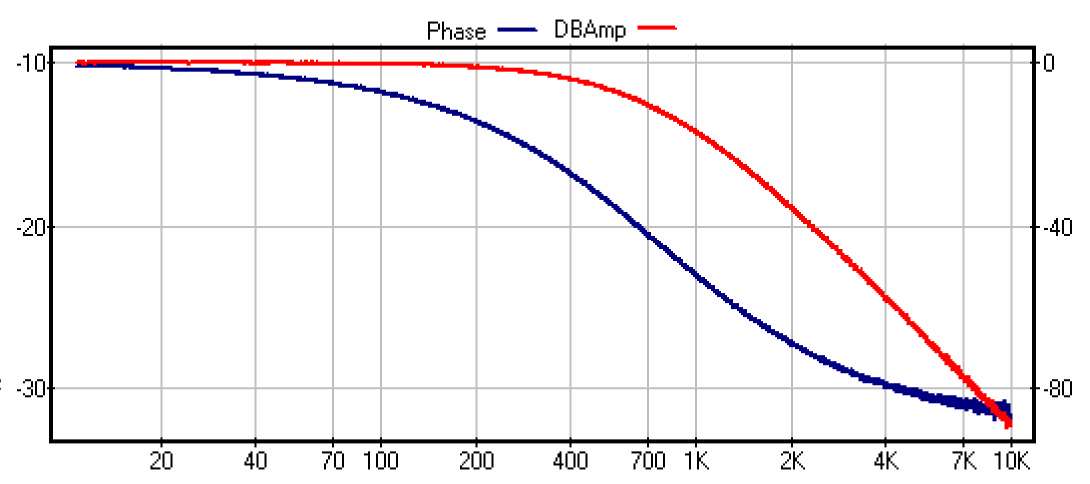
```
(defun filter (&key (bits 12))
  (let ((input-dm    (sd1 :bits bits))
        (feedback-dm (sd1 :bits bits))
        (half-full-scale (ash 1 (1- bits)))
        (state (ash 1 (1- bits))))
    (labels ((de-normalize (x) (round (+ half-full-scale (* x half-full-scale))))
             (normalize (x) (/ (- x half-full-scale) half-full-scale))
             (filter (i) (normalize
                          (incf state
                                (- (funcall input-dm (de-normalize i))
                                   (funcall feedback-dm state))))))
      #'filter)))
```

**This class of filter is physically smaller that the conventional IIR or FIR filters and is "patent pending" by ESS. Refer to USPTO Publication number 20040193665 for more details*

Example of a Generic Test Function

```
(plot-frequency-response-from-quadrature-time-domain
  #'filter
  :fstart 10
  :fstop 10k
  :ain -10db
  :foperate 10me
  :run-time 100m)
```

- This, rather long named function, can test the time domain description and create the frequency domain result

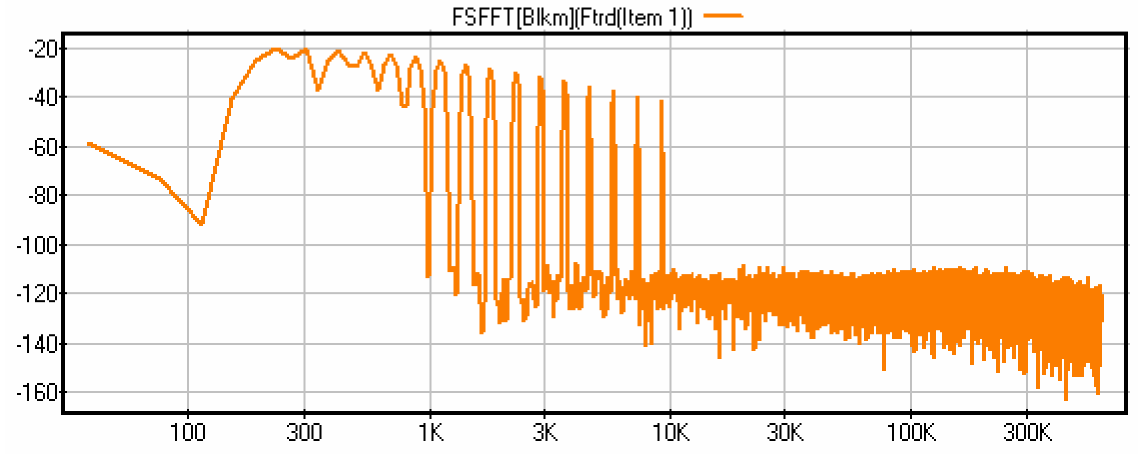


There are more details explaining how this function actually works in the presentation notes

Multi-tone Generic Test Function

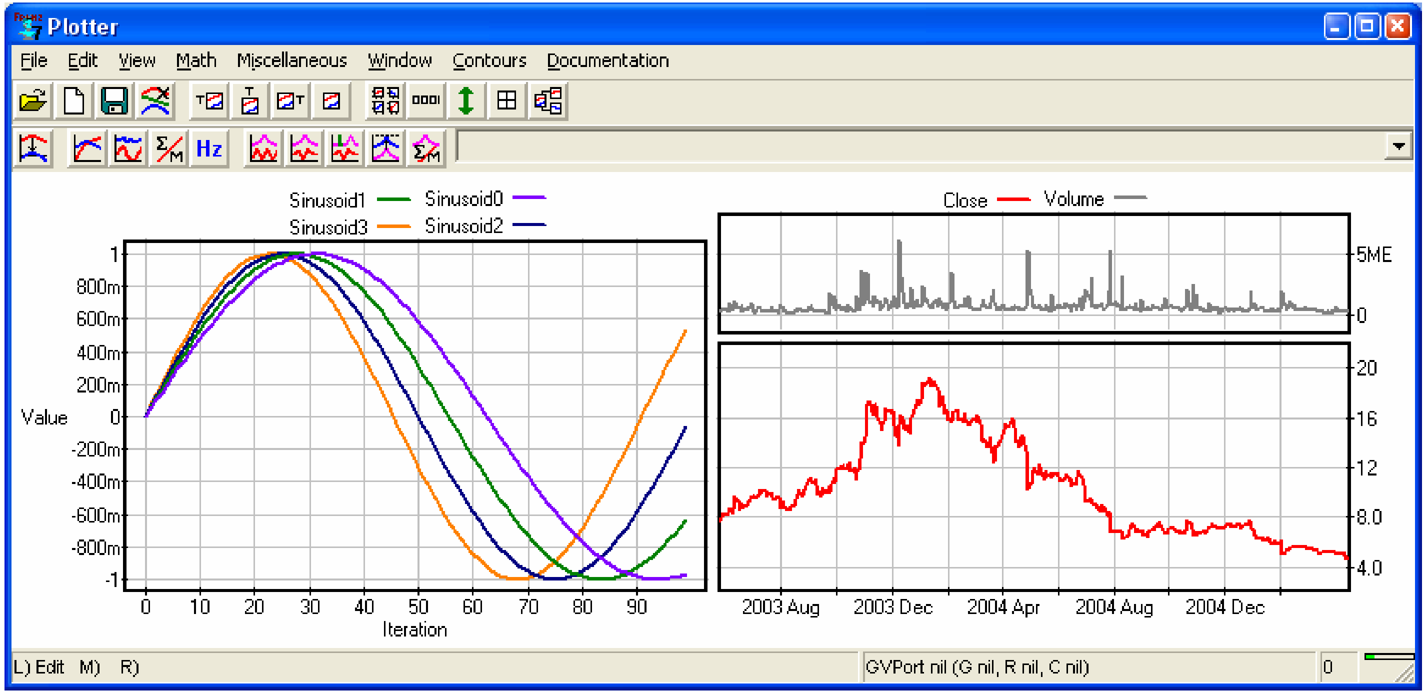
```
(run-a-system #'filter
  :ain -20db
  :fclk 10me
  :ftest '(200 10k)
  :f-dec 10)
```

- This function drives the system with orthogonal multi-tones and plots the FFT of the result



Brief overview of the Plotter

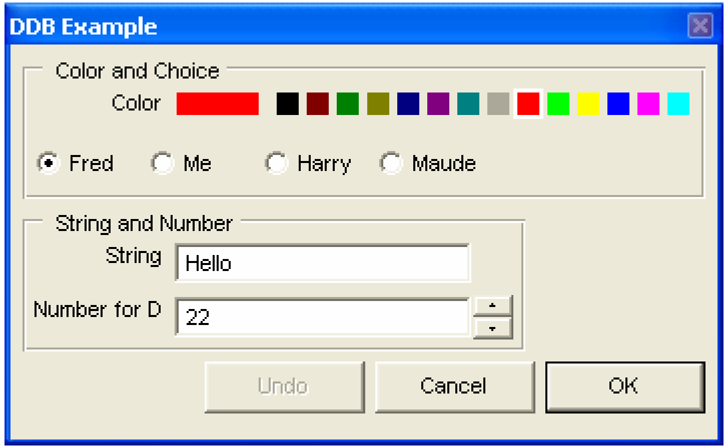
- The plotter plots many forms of data.
- It is based on a Presentation and Gesture system similar to CLIM
- It has many data analysis functions



Dynamic Dialog Boxes

- Windows Dialog boxes are created at run time:

```
(let ((a red)
      (b :fred)
      (c "Hello")
      (d 22))
  (in-dialog-box (:name "DDB Example" :direction :v
                  :prompt-width "Number for D")
    (dialog-column (:name "Color and Choice")
      (dsetf a 'color)
      (dsetf b 'alist :choices
        '(("Fred" . :fred) ("Me". :me) :harry :maude)))
    (dialog-column (:name "String and Number")
      (dsetf c 'string :direction :h)
      (dsetf d 'integer :range '(0 100)
                        :prompt "Number for D"))))
```



(More examples are given in the presentation notes)
Refer to "dynamic-dialog.lisp" if you would like the details

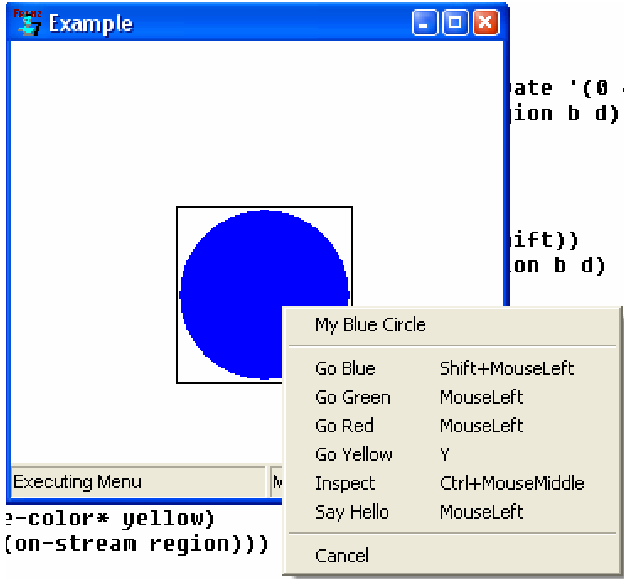
Presentations and Gestures

- The Plotter and all other UI tools are based on presentations and gestures – similar to CLIM:

```
(with-output-as-presentation
    (s :type 'my-circle :object "My Blue Circle")
  (with-foreground-color (s *circle-color*)
    (fill-circle s (make-position ..) ..)))))

(define-gesture :go-blue :buttons :left :chord '(:shift))

(define-presentation-gf :go-blue ((p my-circle) ..)
  (setf *circle-color* blue)
  (invalidate (on-stream region)))
```



Refer to "presentations.lisp, gestures.lisp and select.lisp" if you would like the details

Saving and Distributing Objects

- Objects need to be saved in a file:
  - ASCII “print” format is convenient and, by indexing objects, relatively easy to implement
  - But it is not efficient to save large data sets (such as simulation results) in this format

```
;;; A typical dump file begins something like this:
(1 def notes (2 list (3 list (4 list (5 "Spice
Simulations") (6 "Trans")
(7 "Debug") (8 "Sweep")) \. (9 "tran 1n 100n")) (10
list (11 list (5) (6) (7)) \.
(12 ".save all(v) all(i)")) (13 list (14 list (15
"Spice Simulations") (16 "AC")
(7) (17 "Sweep")) \. (18 "ac dec 100 10 10g")) (19
list (20 list (15) (16) (7)) \.
(21 ".save all(v) all(i)")) (22 list (23 list (24
"Spice Simulations") (25 "DC")
(7) (26 "OP")) \. (27 "op")) (28 list (29 list (24)
(25) (7)) \. (30 ".save all(v) all(i)")))
name (31 "V-Source") blobs (32 list (33 wire-blob x
1344 y 1332 blobs.....
```

Saving and Distributing Objects

- It is more efficient (smaller files, faster) to define two reader macros:
  - #{ reads a binary format list
  - #[read a binary format array

```
;;; A dump file begins something like this:
(2 plot-object contours (3 list (4 2d-contour name (5
list (6 "Sinusoid0"))
creation-time 3325952101 color #S(rgb :red 0 :green 0
:blue 128) single-key
t x-data (7 list-data-item data (8 :data #100{xxxxxxxx
.... )))))
```

#n{ causes the following n file bytes to be read as binary double float values and creates a list of them

Refer to "dump-form.lisp" if you would like the details.

Distributing Associated Files

- Commonly an “object” – for example a schematic drawing – has other information in other files. (The library files provided by the manufacturer for example).
- A new type of “pathname” called a “data-file” behaves in all respects like a regular pathname, except when the object that has a reference to it is dumped to a file, the contents of that file are also dumped.
- When the dumped object is read, possibly at a different site, the embedded file is recreated in the appropriate logical pathname at that new site.

```
sas(1): #F"MySchemtics:fred.txt"
#F"MySchemtics:fred.txt"
sas(2): (describe *)
#F"MySchemtics:fred.txt" is an instance of #<standard-
class data-file>:
 The following slots have :instance allocation:
  logical-pathname        "MySchemtics:fred.txt"
  object-file-pathname    nil
  offset-in-object-file   nil
  length-in-object-file   nil
  file-cache              nil
  file-cache-complete     nil
```

There are more details in the presentation notes
Refer to “dump-form.lisp” if you would like to see the code.

Controlling Other MSWindows Apps.

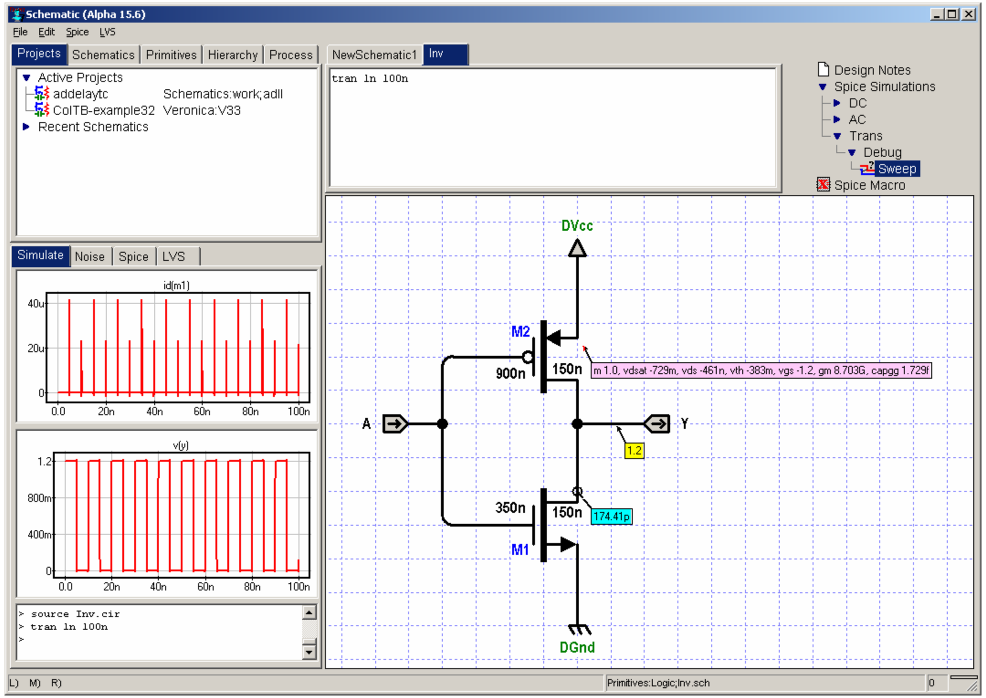
- Assuming we don’t want to use DDE (perhaps it is not available in the remote application) and no other documented API is known, we can always control the remote application as though from the keyboard.
- This is the code for #’advance-ppt:

```
(defun advance-ppt
     (&optional (name "Microsoft PowerPoint"))
  (let ((ppw (caar (find-window-in-tree
                   :name name
                   :substring-ok t)))
        (this
          (development-main-window *system*)))
    (unwind-protect
        (progn
          (setf (topmost this) t)
          (win:SetForegroundWindow ppw)
          (do-keypress ppw vk-pagedown))
      (setf (topmost this) nil)
      (set-foreground-window this)))
  (values))
```

There are many more details in the presentation notes.
Refer to “zombie.lisp” if you would like to see a complete example of controlling the keyboard of one computer from a second computer.

Complete Example: Schematic Editor

- ESS Technology makes Digital Media Chips: DTV, DVD, VCD Audio parts etc.
- These designs have significant analog content: ADC, DAC, SSCG circuits etc.
- All these circuits are designed in ESS's design center in Kelowna, Canada using this LISP based tool.



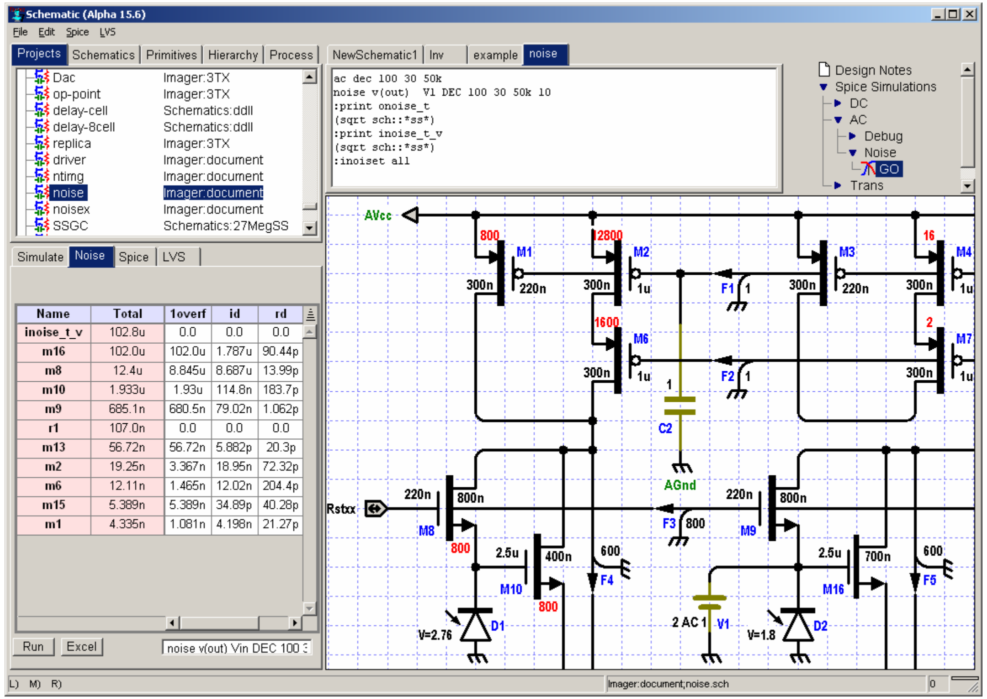
```
tran 1n 100n
```

```
> source Inv.cir
> tran 1n 100n
>
```

m 1.0, vdsat -729m, vds -461n, vth -383m, vgs -1.2, gm 8.703G, capgg 1.729f

Schematic Editor

- The Schematic Editor is a standalone executable: no knowledge of LISP is need to run it – in fact there is no evidence of it being written in LISP to the user.
- It is a competitive and feature rich production quality tool – one of many such tools from many vendors used to design chips.



Dac	Imager:3TX
op-point	Imager:3TX
delay-cell	Schematics:ddll
delay-8cell	Schematics:ddll
replica	Imager:3TX
driver	Imager:document
ntimg	Imager:document
noise	Imager:document
noisex	Imager:document
SSGC	Schematics:27MegSS

```
ac dec 100 30 50k
noise v(out)  V1 DEC 100 30 50k 10
:print onoise_t
(sqrt sch::*ss*)
:print inoise_t_v
(sqrt sch::*ss*)
:inoiset all
```

Name	Total	1overf	id	rd
inoise_t_v	102.8u	0.0	0.0	0.0
m16	102.0u	102.0u	1.787u	90.44p
m8	12.4u	8.845u	8.687u	13.99p
m10	1.933u	1.93u	114.8n	183.7p
m9	685.1n	680.5n	79.02n	1.062p
r1	107.0n	0.0	0.0	0.0
m13	56.72n	56.72n	5.882p	20.3p
m2	19.25n	3.367n	18.95n	72.32p
m6	12.11n	1.465n	12.02n	204.4p
m15	5.389n	5.389n	34.89p	40.28p
m1	4.335n	1.081n	4.198n	21.27p

Summary

- LISP has proved useful and productive as an aid to developing electronic circuits
- More than 20 new (patented or patent pending) circuit have been developed
- It does not seem to be reasonable to expect any new hires to know LISP – it seems to work best when LISP based tools can be used with no LISP knowledge, then, when the user asks for more, one can begin to expose more of the code and eventually users become used to LISP and its syntax.

- Most of the contents of this presentation are available for download.

Martin Mallinson
ILC June 2005