Octtools 5.2
Part I: User Guide

Electronics Research Laboratory
University of California at Berkeley

May 25, 1993

Contents

Chapter 1

Using the Tools

The Octtools are a collection of programs and libraries that together form an integrated system for IC design. The system includes tools for PLA and multi-level logic synthesis, standard cell placement and routing, and custom cell design, as well as a variety of utility programs for manipulating schematic, symbolic, and geometric design data. Most tools are integrated with the Oct data manager and the VEM user interface.

These tools can be used together in various ways. This chapter consists of a tutorial introduction to some of the more important tools, presenting examples of some of the ways they can be used.

The remainder of this first volume contains in-depth tutorials and user's manuals for various tools. Volume Two presents implementation details about the Oct database and some of the tools. Volume Three collects all the UNIX man pages for the tools (which are available on-line via the octman command). Volume Four contains other assorted documentation (also available on-line through octman).

If you need help setting up your UNIX environment for the Octtools, or installing the Octtools, see the first chapter of Volume Two.

1.1 Synthesis Using Different Layout Strategies

The following section shows the use of the synthesis tools for building a cell. Three different layout techniques are used: standard-cell, pla, and gate-matrix. The cell is initially described in a high-level language, BDS, translated to logic equations, minimized and then implemented. See the man pages and users guides for detailed information on each tool.

1.1.1 Specification

The problem is to count the number of zero bits in an N-bit binary string. The input is an N-bit binary string expected to consist of a string of 1's, a string of 0's, and a string of 1's. Any of these strings may be zero length. For example, with N=8, '11001111', '10000001', and '11110000' are all valid strings, and '11001100' is a mal-formed string. The problem is to design a circuit which, given the 'N' bit string, returns the count of number of consecutive zeros in the string, or returns an error condition if the string is mal-formed. The output is undefined if the input string is mal-formed.

Here is the BDS program to do this:

```
! Problem (Count Zeros):  The input is an N-bit binary string expected to
! consist of a string of 1's, a string of 0's, and a string of 1's.  Any
! of these strings may be zero length.  For example, with N=8,
! '11001111', '10000001', and '11110000' are all valid strings, and
! '11001100' is a mal-formed string.  The problem is to design a circuit
! which, given the 'N' bit string, returns the count of number of
! consecutive zeros in the string, or returns an error condition if the
! string is mal-formed.  The output is undefined if the input string is
! mal-formed.
```

```
macro N = 8 $endmacro;
macro LOGN = 3 $endmacro;

model count error<0>, out<LOGN-1:0> = in<N-1:0>;

    ! find first bit matching 'val' (return index of the bit)
    routine ff<LOGN:0>(x<N-1:0>, val<0>);
state i<>;
for i from 0 to N-1 do
    if x<i> eql val then
return i;
return N;
    endroutine;


    routine main;
STATE x<N-1:0>;

! Shift off the first string of 1's (by finding first 0)
x = in sr1 ff(in, 0);

! The count of zeros is then the position of the first 1 in the string
out = ff(x, 1);

! Check for a mal-formed string by shifting off the 0's, and
! then checking that more 0's are found.
x = x sr1 out;
error = ff(x, 0) neq N;
if error then
    out = DONT_CARE;

    endroutine;
endmodel;
```

1.1.2 Standard Cell Implementation

This subsection shows how to use the Oct tools to create standard-cell layouts. The program **wolfe** is used to create the final layout (see Figure 1.1).

- bdsyn (high-level language to logic equation translator)

  `% bdsyn -A -o count.blif -z count.bds`

  Translate the BDS6 functional description into a set of logic equations in `blif`?? format. The '-z' maps all don't cares in the input file to zero. The '-o' option specifies that the output should go in the file `count.blif`, and the '-A' option is needed because in an earlier version of **bdsyn** the option '-o' had a completely different meaning.

- misII (multiple-level logic synthesis)

  `% misII -f script.msu -T oct -o count:logic count.blif`

  Takes the `blif` description and optimizes the logic equations using a standard script (`script.msu`); this includes technology mapping required to create a netlist of the MSU standard-cells. This netlist is placed in Oct as the *logic* view of the *count* cell.

  misII can also be run interactively and the above command can be executed as:

  ```
  % misII
  misII> read_blif count.blif
  misII> source script.msu
  misII> write_oct count:logic
  misII> quit
  ```

- musa (logic and switch-level simulator)

  `% musa count:logic`

  Simulate the logic description produced by **misII**.

```
musa> mv in in<7:0>     ! create an input vector
musa> mv out out<2:0>   ! create output vectors
musa> mv err error<0>
musa> set in B00011100  ! set the input vector (in binary)
musa> ev                ! evaluate the circuit
musa> show err out      ! display the output vectors
 err:1  out:H0          ! error flag is set
musa> set in B11100011   ! change the input vector
musa> ev                ! re-evaluate the circuit
musa> show err out      ! display the new output vector
 err:0  out:H3          ! error flag is not set
                        !    out is 3 (H = hexadecimal)
musa> set in B10000001   ! and so on....
musa> ev
musa> show err out
 err:0  out:H6          ! error flag is not set
                        !    out is 6
musa> quit              ! exit musa
```

  Notice that it is generally convenient to prepare a script containing all the simulation commands, because this aliminates the need for tedious retyping. Assuming that the script is called `count.musa` you could then invoke the command

  `% musa -i count.musa count:logic`

- vem (graphics editor)

  `% vem`

  Open the **vem** console window, and type the following command in the console window (*i.e.,* place the mouse cursor in the console window):

  `vem> "count:logic" o`

  Notice that all cells are drawn on top of each other and that there is no interconnection implementation.

  To exit **vem**, type the following command in the console window:

  `vem> :  close-window`

  Typing the `close-window` command in the window displaying the cell will close just that window rather than all of the **vem** windows.

  Notice that two different command entry mechanisms have been used, the single-key binding and the full command name. There are three ways to enter commands in **vem**, the two shown above and by selecting the command from the menu. The menu can be popped-up by pressing down the middle mouse button.

- padplace (assigning terminal locations)

  `% padplace -D count.padlist count:logic`

  The program **padplace** is used to assign terminal implementations and locations. In this example, terminal types, directions, edges, and positions are assigned. To get a list of the formal terminals and locations, run:

```
% padplace -l count:logic > count.padlist
```

Then you can use a text editor to rearrange the pads and run **padplace** with the **-D** flag to assign the locations.

Some tools may not guarantee that the terminals will be on the edges and in the order specified with **padplace**.

```
#
# Formal Terminal description for count:logic
# padp recognizes the following keywords:
# TERMTYPE (SUPPLY | GROUND | SIGNAL | CLOCK) : DEFAULT = SIGNAL
# DIRECTION (INPUT | OUTPUT | INOUT) : DEFAULT = INOUT
# PEAKCURRENT ( x (in Amperes)) : DEFAULT = 0.0
#   NOTE: PEAKCURRENT applies only to SUPPLY and GROUND terminals
# TERM_FLOATING_EDGES (ALL | LEFT|RIGHT|TOP|BOTTOM)
# TERM_EDGE (LEFT | RIGHT | TOP | BOTTOM) : DEFAULT = none
# TERM_ROW ( 0 | 1 | 2 | ... | n) : DEFAULT = 0
# TERM_RELATIVE_POSITION ( 0.0 <= x <= 1.0) : DEFAULT = none
# TERM_RELATIVE_POSITION_STEP ( 0.0 <= x <= 1.0) : DEFAULT = 0.0

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
        TERMTYPE SIGNAL
        DIRECTION INPUT
        TERM_ROW 0
        TERM_EDGE LEFT
        TERM_RELATIVE_POSITION 0.0
        TERM_RELATIVE_POSITION_STEP 0.1
FORMAL_TERMINAL in<4>
FORMAL_TERMINAL in<3>
FORMAL_TERMINAL in<2>
FORMAL_TERMINAL in<1>
FORMAL_TERMINAL in<0>
FORMAL_TERMINAL in<5>
FORMAL_TERMINAL in<7>
FORMAL_TERMINAL in<6>
        DIRECTION OUTPUT
        TERM_EDGE RIGHT
        TERM_RELATIVE_POSITION 0.0
        TERM_RELATIVE_POSITION_STEP 0.1
FORMAL_TERMINAL error<0>
FORMAL_TERMINAL out<2>
FORMAL_TERMINAL out<1>
FORMAL_TERMINAL out<0>
```

- wolfe (standard cell place and route)

  ```
  % wolfe -o :wolfe count:logic
  ```

  This runs **wolfe** on the logic view of the cell and creating the placed and routed view of the count cell. *Logic* and *wolfe* are names with no real 'meaning'; both views conform to the Oct Symbolic Policy. We just use the name *logic* to refer to the cell where the instances are not placed or routed, and *wolfe* to refer to the placed and routed cell.

- musa

  ```
  % musa -i count.musa count:wolfe
  ```

  Verify that the logic is still the same.

- chipstats (statistics about the layout)

  ```
  % chipstats count:wolfe
  ```

  Output information about a layout: area, routing area, via count, net lengths, *etc.*.

```
chipstats: Version 1.0 : Chip  count:wolfe

Number of instances    = 90
Number of nets         = 100
Number of connectors   = 553
Number of vias         = 390

Average instance area  =        2665.07
St.dev. instance area  =         624.33
Chip Area W x H        =    624 x   720
Chip Area              =      449280.00 (100.00%)
Total Instance Area    =      239856.00 ( 53.39%)
Routing + Empty Area   =      209424.00 ( 46.61%)
Total   net area       =      197317.50
Total   net length     =       48012.50
Average net length     =         480.12
St.dev. net length     =         562.58

***** 6 LONGEST NETS
(---) NAME                  LENGTH (lambda)  AREA (lambda^2)
(  1) GROUND                        3432.00            37056.00
(  2) SUPPLY                        3384.00            36672.00
(  3) [705]                         1550.00             4650.00
(  4) [700]                         1503.50             4510.50
(  5) [699]                         1440.50             4321.50
(  6) [885]                         1396.00             4188.00
```

- vem

  `% vem`

  Open the **vem** console window and give the commands:

  `vem> "count:wolfe" o`

  Notice that now the cells have been placed and routed.

  You could also say:

  `% vem -F count:wolfe`

- oct2ps (convert Oct to PostScript for printing)

  `% oct2ps -f bb count:wolfe > count.ps`

  This converts an OCT' facet into `PostScript` for printing on a PostScript compatible printer (*e.g.,* Apple LaserWriter, DEC LPS-40). The '-f bb' flag tells **oct2ps** only to generate code to print the bounding box and name of each subcell, not the details.

1.1.3 PLA Implementation

This subsection shows the same example as above, but using **octpla** to generate the layout (see Figure 1.2).

- bdsyn (convert `BDS` to logic equations)

  `% bdsyn -A -o count.blif -z count.bds`

- octpla (generate PLA from `blif`)

  `% octpla -nofold -o pla count`

  octpla reads the blif file, runs **misII** to convert the logic into `pla` format, runs **espresso** to minimize the logic, **genie** to generate the pla array, **octopus** to connect up the cells, **sparcs** to compact the layout, and **vulcan** to create the interface facet of the final layout (the interface facet contains an abstraction of the PLA, that is a collection of protection frames on each of the layers used in the PLA). The output is `count:pla`. The option `nofold` is required

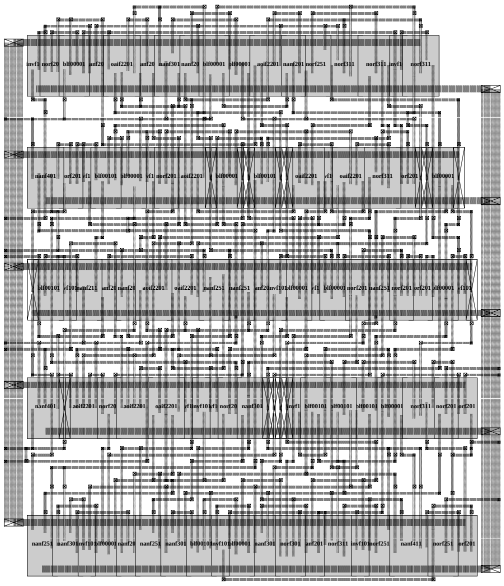

Figure 1.1: The standard cell implementation of "count".

because the code that performs PLA folding is not reliable (maybe you would want to try to run **octpla** without this option). The order of the terminals is based on the BDS description (same order as given in the `INPUT` and `OUTPUT` statements), thus **padplace** is not needed to assign locations.

- Alternative to **octpla**: **anchor**

  `% misII -T pla -o count.pla count.blif`

  `% anchor -a -F count:pla`

  It is necessary to run **misII** to produce the required input for **anchor**, a file in the "PLA" format. **Anchor** is a new tool, with a better control on the annealing process. **Anchor** has already proven that it can complete a PLA in cases where **octpla** gets stuck in an infinite loop. The use of **anchor** is encouraged. The option -a forces **anchor** to start PLA optimization from scratch, and -F asks for no folding. In this `count:pla` is the output. The default input is `count.pla`.

- musa (simulate the PLA)

  `% musa -i count.musa count:pla`

  Simulate the transistor level representation, using the same input as in the previous example.

- vem

  `% vem`

  `vem> "count:pla" o`

  Take a look at the **PLA**.

  `vem> "count:pla:interface" o`

  Take a look at the interface facet as well. Please notice that **vem** performs automatic file and facet name completion if you press the tab key.

- chipstats

  `% chipstats count:pla`

  Find out the area of the **PLA**.

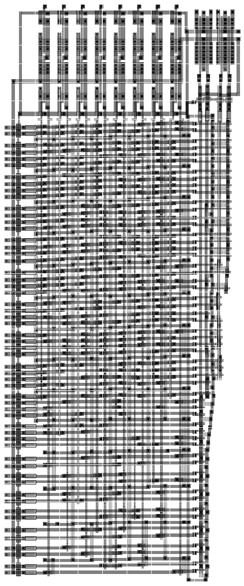

Figure 1.2: The PLA implementation of count.

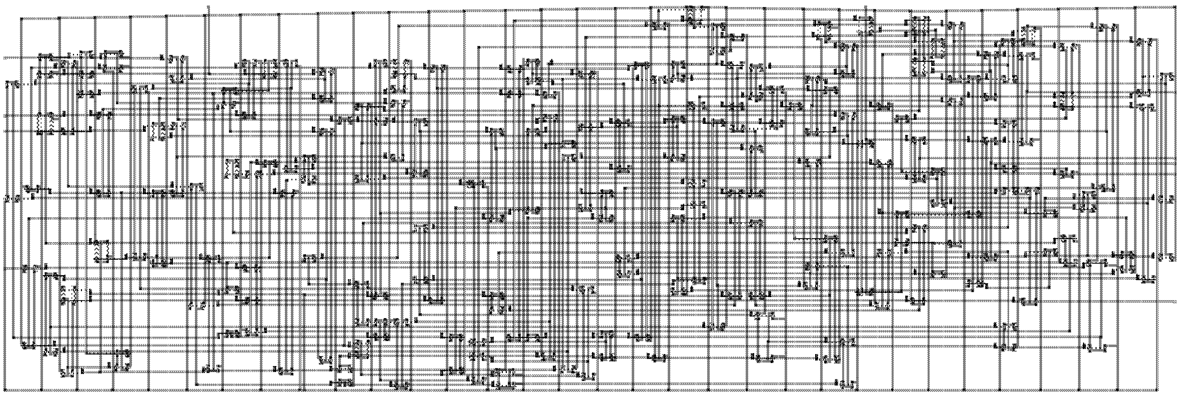

Figure 1.3: Gate matrix layout of "count".

1.1.4 Gate Matrix Implementation

This subsection shows the same example as above, but using **gem** to generate the layout (see Figure 1.3). Currently **gem** does not produce acceptable layout: it is a bit too sparse and it lacks substrate contacts. It is nevertheless interesting to take a look at another layout style.

This time the sequence of command should be:

```
% bdsyn -A -o count.blif -z count.bds
% gem count
% musa -i count.musa count:gem
% vem -F count:gem
% chipstats count:gem
```

In this example, the **gem** layout is much larger than the **octpla** layout. On smaller examples, the opposite may happen.

1.2 Simple Finite State Machine

This subsection shows how to build a simple finite-state machine, in this case, a traffic light controller. The example covers the preparation of the netlist, which consists of a PLA and a set of latches.

1.2.1 Specification

See pages 85 through 88 in "Introduction to VLSI Systems" by Carver Mead and Lynn Conway for a description of the traffic light controller.

- **bdsyn** (convert the `BDS` description to logic equations)

  `% bdsyn -A -o fsm.blif -z fsm.bds`

```
MODEL traffic_light
    nextState<1:0>,      ! next state
    hl<1:0>,             ! highway light
    fl<1:0>,             ! farm light
    st<0>                ! start interval time
    =
    c<0>,                ! car on the farm road
    ts<0>,               ! short interval timer timeout
    tl<0>,               ! long interval timer timeout
    presentState<1:0> ;  ! current state

! state assignments
CONSTANT HG = 0, HY = 1, FG = 2, FY = 3;

! symbolic output assignments
CONSTANT GREEN = 0, RED = 1, YELLOW = 2;

ROUTINE traffic_light_controller;

        SELECT presentState FROM
        [HG]:
            BEGIN
                hl = GREEN;
                fl = RED;
                IF c AND tl THEN BEGIN
                    nextState = HY;
                    st = 1;
                END
                ELSE BEGIN
                    nextState = HG;
                    st = 0;
                END;
            END;

        [HY]:
            BEGIN
                hl = YELLOW;
                fl = RED;
                IF ts THEN BEGIN
                    nextState = FG;
                    st = 1;
                END
                ELSE BEGIN
                    nextState = HY;
                    st = 0;
                END;
            END;

        [FG]:
            BEGIN
                hl = RED;
                fl = GREEN;
```

```
                IF NOT c OR tl THEN BEGIN
                    nextState = FY;
                    st = 1;
                END
                ELSE BEGIN
                    nextState = FG;
                    st = 0;
                END;
            END;

        [FY]:
            BEGIN
                hl = RED;
                fl = YELLOW;
                IF ts THEN BEGIN
                    nextState = HG;
                    st = 1;
                END
                ELSE BEGIN
                    nextState = FY;
                    st = 0;
                END;
            END;
        ENDSELECT;

ENDROUTINE traffic_light_controller;
ENDMODEL traffic_light;
```

- **nova** (choose the optimal state assignment)

  `% nova fsm.st > fsm.out`

  Currently, the state assignment program, **nova**, is not well integrated into the system, so you will have to create your own state table input for **nova**. The state table can be trivially generated from the BDS file or from the blif file. The output of **nova** can then be placed back into the original BDS file, and **bdsyn** can be run again.

```
# c tl ts ps ns st hl0 hl1 fl0 fl1
0-- state0 state0 00010
-0- state0 state0 00010
11- state0 state1 10010
--0 state1 state1 00110
--1 state1 state3 10110
10- state3 state3 01000
0-- state3 state2 11000
-1- state3 state2 11000
--0 state2 state2 01001
--1 state2 state0 11001
```

```
# nova
.start_kiss
# c tl ts ps ns st hl0 hl1 fl0 fl1
0-- state0 state0 00010
-0- state0 state0 00010
11- state0 state1 10010
--0 state1 state1 00110
--1 state1 state3 10110
10- state3 state3 01000
0-- state3 state2 11000
-1- state3 state2 11000
--0 state2 state2 01001
--1 state2 state0 11001
.end_kiss
#
# Prod.-terms of minimized MV cover = 10
# Prod.-terms of encoded cover = 9
```

```
# Pla area = 153
#
# .start_codes
.code state0 00
.code state1 11
.code state3 01
.code state2 10
# .end_codes
#
# .start_network
.model SISa01306.esp
.inputs v0 v1 v2
.outputs v5.2 v5.3 v5.4 v5.5 v5.6
.latch v5.0 v3 0
.latch v5.1 v4 0
.names v0 v1 v3 v4 [0]
1001 1
.names v0 v1 v3 v4 [1]
1100 1
.names v2 v3 [2]
01 1
.names v2 v3 [3]
11 1
.names v3 v4 [4]
00 1
.names v3 v4 [5]
10 1
.names v0 v3 v4 [6]
001 1
.names v1 v3 v4 [7]
101 1
.names v3 v4 [8]
11 1
.names [1] [2] [6] [7] [9]
0000 1
.names [9] v5.0
0 1
.names [0] [1] [8] [11]
000 1
.names [11] v5.1
0 1
.names [1] [3] [6] [7] [13]
0000 1
.names [13] v5.2
0 1
.names [0] [5] [6] [7] [15]
0000 1
.names [15] v5.3
0 1
.names [8] [17]
0 1
.names [17] v5.4
0 1
.names [4] [8] [19]
00 1
.names [19] v5.5
0 1
.names [5] [21]
0 1
.names [21] v5.6
0 1
.end
# .end_network
#
# CPU time = 0.0 seconds
```

- **octpla** (minimize and generate a PLA)

```
% octpla -nofold fsm
```

- **bdnet** (wire the latches)

```
% bdnet fsm.bdnet
```

```
MODEL fsm:unplaced;
TECHNOLOGY scmos;
VIEWTYPE SYMBOLIC;
EDITSTYLE SYMBOLIC;

INPUT hl<1:0>, fl<1:0>, st<0>;
OUTPUT c<0>, ts<0>, tl<0>;
SUPPLY Vdd;
GROUND GND;
CLOCK clk;

INSTANCE fsm:pla
        hl<1:0>: hl<1:0>;
        fl<1:0>: fl<1:0>;
        st<0># st<0>;
        c<0># c<0>;
        ts<0># ts<0>;
        tl<0># tl<0>;
        nextState<1:0>: latchin<1:0>;
        presentState<1:0>: latchout<1:0>;
        Vdd: Vdd;
        GND: GND;

ARRAY %i FROM 0 TO 1 OF
INSTANCE "$OCTTOOLS/tech/scmos/msu/stdcell2_2/dfnf311":physical
        DATA1: latchin<%i>;
        Q: latchout<%i>;
        Q_b: UNCONNECTED;
        CLK2: clk;
        "Vdd!": Vdd;
        "GND!": GND;

INSTANCE decode:logic
        in<1:0> : hl<1:0>;
        red: hred;
        yellow: hyellow;
        green: hgreen;

INSTANCE decode:logic
        in<1:0> : fl<1:0>;
        red: fred;
        yellow: fyellow;
        green: fgreen;

INSTANCE "$OCTTOOLS/lib/musa/greenled":physical
        [0, 20]
        ANODE: hgreen;
        CATHODE: GND;
INSTANCE "$OCTTOOLS/lib/musa/yellowled":physical
        [0, 10]
        ANODE: hyellow;
        CATHODE: GND;
INSTANCE "$OCTTOOLS/lib/musa/redled":physical
        [0, 0]
        ANODE: hred;
        CATHODE: GND;

INSTANCE "$OCTTOOLS/lib/musa/greenled":physical
        [0, 60]
        ANODE: fgreen;
        CATHODE: GND;
INSTANCE "$OCTTOOLS/lib/musa/yellowled":physical
```

```
        [0, 50]
        ANODE: fyellow;
        CATHODE: GND;
INSTANCE "$OCTTOOLS/lib/musa/redled":physical
        [0, 40]
        ANODE: fred;
        CATHODE: GND;

ENDMODEL;
```

- **musa** (simulate the FSM)

  `% musa -i fsm.musa fsm:unplaced`

```
! outputs
makevector hl hl<1:0>
makevector fl fl<1:0>
makevector st st<0>
! inputs
makevector tl tl<0>
makevector ts ts<0>
makevector car c<0>

macro clock
    set clk 1
    ev
    set clk 0
    ev
    show hl fl st
$end

macro test
    set tl $1
    set ts $2
    set car $3
    clock
$end

! do some test
test 0 0 0
test 0 0 1
test 0 1 0
test 1 1 1
```

Now that the FSM is ready, you could integrate it into a large chip. You would then use **bdnet** to describe the chip, **padplace** and **puppy** to place the chip, and **mosaico** to route it, as shown, for example in Section 1.5.

1.3 A 4-bit Up/Down Counter

Paul Cohen (M2C)

For some time now, **M2C** has been evaluating the OCT tool suite from The University of California at Berkeley. The tools are now included as an unsupported tool on the **M2C** release tape. In order to assist those faculty and students who are now evaluating the tools at their school, we are providing the following sample (simple) design (a 4 bit up/down counter) to illustrate the interaction of different tools in the OCT tool suite.

1.3.1 Specification

The inputs to the counter will be a clock, `clk`, a clear signal, `clr<0>`, and an up/down control signal, `ud<0>`. The outputs are `q<3:0>`, where `q<3>` is the most significant bit of the counter. The operation of the counter is:

clr<0>	ud<0>	q<3:0>
0	X	Synchronous clear
1	1	Count up
1	0	Count down

1.3.2 Design

The control logic for the counter can be described in the OCT high level language, **BDS**. The code for this might look like:

```
model updown d<3:0> = q<3:0>, clr<0>, ud<0>;
routine count;
    if clr EQL 0 then d = 0 else
        if ud EQL 1 then d = q + 1 else d = q - 1;
endroutine;
endmodel;
```

The next step is to convert the **BDS** description to **BLIF**. This is done with the program **bdsyn**. The command line is (assuming the above file is named `updown.bds`):

```
% bdsyn -A -o updown.blif updown.bds
```

The control logic can then be optimized, mapped into the MSU (Mississippi State University) standard cells, and saved in OCT format with the command:

```
% misII -f script.msu -T oct -o updown:logic updown.blif
```

With the control logic complete, the logic is connected to four flip flops (cell dfnf311 in the MSU library) to complete the counter. This is done with **bdnet**, the netlisting tool in OCT. The netlist file (`updown.bdnet`) might look like:

```
MODEL updown:unplaced;          ! Choose a name for this cell.
TECHNOLOGY scmos;               ! All these properties are required.
VIEWTYPE SYMBOLIC;
EDITSTYLE SYMBOLIC;

INPUT clk, ud<0>, clr<0>;       ! Declare formal terminals of this circuit.
OUTPUT q<3:0>;
SUPPLY Vdd;
GROUND GND;
```

```
! Put down an instance of the synthesized logic
! and a bunch of latches to keep the state.

INSTANCE updown:logic PROMOTE;

ARRAY %i FROM 0 TO 3 OF
    INSTANCE "$OCTTOOLS/tech/scmos/musa/stdcell2_2/dfnf311":physical
        DATA1:d<%i>;
        CLK2:clk;
        Q:q<%i>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;      !! Need quotes because of the funny char in name
        "GND!":GND;
ENDMODEL;
```

The command:

```
% bdnet updown.bdnet
```

will create an Oct cell, `updown`, representing the logic for the counter.

1.3.3 Simulation

You can simulate the circuit with the multi-level simulator, **musa**. A sample simulation control file (updown.sim) is shown below.

```
! Make a vector
mv q q<3:0>
! Define a macro for the clock
ma c
  se clk 1
  ev
  se clk 0
  ev
  sh ud<0> clr<0> q
$end

! Set input and then crank the clock.
se clr<0> 0
rp 32 c
! Different input now
se clr<0> 1
se ud<0> 1
rp 32 c
se ud<0> 0
rp 32 c
```

The above file first creates a vector, `q`, to simplify references to the outputs, `q<3:0>`. Next a macro, `c`, is created to cycle the clock and display the signals. The `rp` command is short hand for `repeat`.

Typing:

```
% musa -i updown.sim updown:unplaced
```

invokes the simulator and executes the simulation command file.

1.3.4 Layout

wolfe can be used to place and route the cells. Before you use **wolfe**, you will need to flatten the hierarchy with **octflatten**. The command line is:

```
% octflatten -t LEAF -o updown:flat updown:unplaced
```

If we do not specify where the terminals of the counter go **wolfe** will place them at random around the periphery of the cell. That is the worst possible way to do layout. Elsewhere you have seen examples of how the -D option in **padplace** can be used to specify terminal position. In this example we use the -c option, which places the terminals in a clockwise fashion around the cell (beats placing them at random, because at least the bit vectors will be adjacent and in the right order).

```
% padplace -c updown:flat
```

You can now use **wolfe** to place and route the cells with the command:

```
% wolfe -o updown:placed updown:flat
```

wolfe will pass the circuit to the standard cell place and route program, **TimberWolfSC**. When this is done, use **vem** or a plotting program, such as **oct2ps**, to look at the layout (`oct2ps -f bb updown:placed`).

Alternatively you could have used the -h option in **wolfe** to do the flattening of the hierarchy on the fly:

```
% padplace -c updown:unplaced
% wolfe -h -o updown:placed updown:unplaced
```

1.4 Synthesis and Optimization

This section uses three different optimization scripts for **misII** to show the effectiveness of different optimization strategies. The same logic input will be used for all examples:

- let `ex.blif` be any BLIF file obtained, for example, by running **bdsyn** (see examples at the end of the **bdsyn** manual). In our test run we have used one the BDS description shown in Figure 6.2.

  ```
  % bdsyn -A -o ex.blif count.bds
  ```

- minimize the logic via three different techniques, each represented by a different optimization script. The first script `ex.zero` does no optimization at all. The second uses algebraic optimization techniques, the third uses the more powerful, but computationally more expensive, techniques of boolean optimization. The scripts are shown in Table 1.1.

  ```
  % misII -f ex.zero -T oct -o ex:zero ex.blif
  % misII -f ex.algebraic -T oct -o ex:algebraic ex.blif
  % misII -f ex.boolean -T oct -o ex:boolean ex.blif
  ```

- place and route the results. Place the terminals in a clockwise fashion using **padplace**, then place and route the cells using **wolfe**. For more realistic results, you should not use the -f option in **wolfe**. This option shortens the duration of the annealing algorithm to its minimum: **wolfe** produces a result faster, but the quality is probably very poor.

  ```
  % padplace -c ex:zero
  % wolfe -f -o :zero-pandr ex:zero

  % padplace -c ex:algebraic
  % wolfe -f -o :algebraic-pandr ex:algebraic

  % padplace -c ex:boolean
  % wolfe -f -o :boolean-pandr ex:boolean
  ```

- compare the areas and cell counts. Your numbers may differ, but the relative ratios should be similar.

  ```
  % chipstats ex:zero-pandr
  ```

  117 instances, 674264 square lambda

  ```
  % chipstats ex:algebraic-pandr
  ```

  67 instances, 376272 square lambda

  ```
  % chipstats ex:boolean-pandr
  ```

  29 instances, 159544 square lambda

ex.setup

This short script is used by all the other scripts in this exercise to map the logic network into standard cells of the MSU2.2 library. The script simply loads the library information. The actual mapping is performed by the `map` command in the other scripts.

```
#
set OCT-CELL-VIEW physical
set OCT-CELL-PATH $OCTTOOLS/tech/scmos/msu/stdcell2_2
set OCT-TECHNOLOGY scmos
set OCT-EDITSTYLE SYMBOLIC
```

```
set OCT-VIEWTYPE SYMBOLIC
#
# loading standard-cell library description
#
read_library stdcell2_2.genlib
#
```

ex.zero	ex.algebraic	ex.boolean
#	#	#
sr ex.setup	sr ex.setup	sr ex.setup
#	#	#
# mapping logic to	# doing algebraic	# doing boolean
# standard-cells	# minimization	# minimization
#	#	#
map	sw	el 10
phase -g	el 5	sim1 *
print_stats	sim1 *	asb
print_gate -s	asb	gkx
#		asb
# verify result	gkx -abt 30	el 0
#	asb; sw	asb
verify ex.blif	gcx -bt 30	gkx
	asb; sw	asb
		el 0
	gkx -abt 10	sim2 *
	asb; sw	gd *
	gcx -bt 10	asb
	asb; sw	el -1
		asb
	gkx -ab	sim2 *
	asb; sw	el -1
	gcx -b	#
	asb; sw	# mapping logic to
		# standard-cells
	el 0	#
	gd *	map
	#	phase -g
	# mapping logic to	print_stats
	# standard-cells	print_gate -s
	#	#
	map	# verify result
	phase -g	#
	print_stats	verify ex.blif
	print_gate -s	
	#	
	# verify result	
	#	
	verify ex.blif	
ex.zero	ex.algebraic	ex.boolean

Table 1.1: Three possible optimization scripts for **misII**

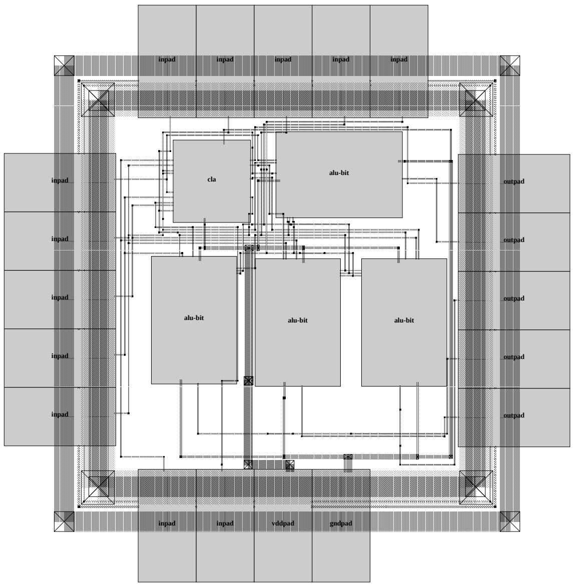

Figure 1.4: The small chip generated in the place and route example

1.5 Place and Route Example

This shows a more complete design example: subsections are designed, placed and routed, and connected to a pad frame (see Figure 1.4).

A 4-bit alu is described via alu bitslices and a carry look ahead circuit, the cells are connected together, pads added, and placed and routed.

- convert alu bitslice to a macro cell (and simulate)

```
% bdsyn -A -o alu-bit.blif -z alu-bit.bds
% misII -f script.msu -T oct -o alu-bit:logic alu-bit.blif
% musa -i alu-bit.musa alu-bit:logic
% padplace -D alu-bit.padlist alu-bit:logic
% wolfe -o alu-bit:symbolic alu-bit:logic
% musa -i alu-bit.musa alu-bit:symbolic
```

- convert carry look ahead to a macro cell (and simulate)

```
% bdsyn -A -o cla.blif -z cla.bds
% misII -f script.msu -T oct -o cla:logic cla.blif
% musa -i cla.musa cla:logic
% padplace -D cla.padlist cla:logic
% wolfe -o cla:symbolic cla:logic
% musa -i cla.musa cla:symbolic
```

- wire them together (and simulate the result)

```
% bdnet alu.bdnet
% musa -i alu.musa alu:unplaced
```

- place the pads

```
% padplace -u m2c_2u -P -D alu.padlist -o alu:pads alu:unplaced
```

The '-u' flag specifies the *pad family* to use. The '-P' flag says that we expect a pad limited chip, the '-D' option selects the pad description file `alu.padlist`, and the '-o' option tells **padplace** the name of the output view. A pad family consists of pads for input, output, input-output, vdd, and ground.

- place and route (and simulate the result)

```
% puppy -o alu:puppy alu:pads
% mosaico -o final alu:puppy
% musa -i alu.musa alu:final
```

- alternative for routing

  The pads selected for this example have terminals for power and ground rings. In this case it is appropriate to stop the **mosaico** sequence just before the final compaction:

```
% mosaico -s alu:puppy
```

The output of **mosaico** in this case is called `alu:puppy.mizer`, because **mizer** is the last tool executed by the **mosaico** sequence.

Then we route the rings using **padplace** and perform the final compaction and spacing using **sparcs**:

```
% padplace -R -o alu:puppy.rings alu:puppy.mizer
% sparcs -t -o alu:puppy.spaced alu:puppy.rings
```

Use **chipstats** to compute area and net length of the final chip.

Behavioral Description (alu-bit.bds)

```
model alubit s, cout = a, b, cin, opcode<2:0>;
routine main;
    select opcode FROM
        [000#2] : BEGIN
                  s = cin xor (a xor b);
                  cout = (a and b) or (cin and (a or b));
                  END;
        [001#2] : BEGIN
                  s = a and b;
                  cout = 0;
                  END;
        [010#2] : BEGIN
                  s = a or b;
                  cout = 0;
                  END;
        [011#2] : BEGIN
                  s = a xor b;
                  cout = 0;
                  END;
        [100#2] : BEGIN
                  s = not a;
                  cout = 0;
                  END;
        [101#2] : BEGIN
                  s = 0;
                  cout = 0;
                  END;
        [110#2] : BEGIN
                  s = 1;
                  cout = 0;
                  END;
```

```
        [111#2] : BEGIN
                  s = not (a and b);
                  cout = 0;
                  END;
    endselect;
endroutine;
endmodel;
```

Simulation Input (alu-bit.musa)

```
! should define a macro
mv op opcode<2:0>
macro add
    set a<0> $1
    set b<0> $2
    set cin<0> $3
    set op 000
    ev
    show cout<0> s<0>
$end
macro xor
    set a<0> $1
    set b<0> $2
    set op 011
    ev
    verify cout<0> 0
    show cout<0> s<0>
$end
macro clr
    set op 101
    ev
    verify cout<0> 0
    verify s<0> 0
    show cout<0> s<0>
$end
add 0 0 0
add 0 0 1
add 0 1 0
add 0 1 1
add 1 0 0
add 1 0 1
add 1 1 0
add 1 1 1
!
xor 0 0
xor 0 1
xor 1 0
xor 1 1
!
clr
!
exit
```

Port Placement Description (alu-bit.padlist)

```
#
# Formal Terminal description for alu-bit:logic
# padplace recognizes the following keywords:
# TERMTYPE (SUPPLY | GROUND | SIGNAL | CLOCK)
# DIRECTION (INPUT | OUTPUT | INPUT)
# TERM_FLOATING_EDGES (LEFT|RIGHT|TOP|BOTTOM)
# TERM_EDGE (LEFT | RIGHT | TOP | BOTTOM)
# TERM_ROW ( 0 | 1 | 2 | ... | n)
# TERM_RELATIVE_POSITION ( 0.0 <= x <= 1.0)
# TERM_RELATIVE_POSITION_STEP ( 0.0 <= x <= 1.0)
```

```
# PEAKCURRENT ( x (in Amperes)) // Applies only to SUPPLY and GROUND terminals

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
        TERMTYPE SIGNAL
        DIRECTION INPUT
        TERM_ROW 0
        TERM_EDGE LEFT
FORMAL_TERMINAL cin<0>
FORMAL_TERMINAL a<0>
FORMAL_TERMINAL b<0>
        TERM_EDGE TOP
FORMAL_TERMINAL opcode<0>
FORMAL_TERMINAL opcode<1>
FORMAL_TERMINAL opcode<2>
        DIRECTION OUTPUT
        TERM_EDGE RIGHT
FORMAL_TERMINAL s<0>
FORMAL_TERMINAL cout<0>
```

Behavioral Description (cla.bds)

```
model cla cout<3:0> = a<3:0>, b<3:0>, c;
routine main;
    STATE i<>, g<3:0>, p<3:0>, cin<4:0>;
    cin<0> = c;
    FOR i FROM 0 TO 3 DO
BEGIN
    g<i> = a<i> and b<i>;
    p<i> = a<i> xor b<i>;
    cout<i> = g<i> or (p<i> and cin<i>);
    cin<i+1> = cout<i>;
END;
endroutine;
endmodel;
```

Simulation Input (cla.musa)

```
MISSING
```

Port Placement Description (cla.padlist)

```
#
# Formal Terminal description for cla:logic
# padplace recognizes the following keywords:
# TERMTYPE (SUPPLY | GROUND | SIGNAL | CLOCK)
# DIRECTION (INPUT | OUTPUT | INPUT)
# TERM_FLOATING_EDGES (LEFT|RIGHT|TOP|BOTTOM)
# TERM_EDGE (LEFT | RIGHT | TOP | BOTTOM)
# TERM_ROW ( 0 | 1 | 2 | ... | n)
# TERM_RELATIVE_POSITION ( 0.0 <= x <= 1.0)
# TERM_RELATIVE_POSITION_STEP ( 0.0 <= x <= 1.0)
# PEAKCURRENT ( x (in Amperes)) // Applies only to SUPPLY and GROUND terminals

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
        TERMTYPE SIGNAL
        DIRECTION INPUT
        TERM_ROW 0
        TERM_EDGE TOP
```

```
FORMAL_TERMINAL c<0>
FORMAL_TERMINAL a<3>
FORMAL_TERMINAL a<2>
FORMAL_TERMINAL a<1>
FORMAL_TERMINAL a<0>
FORMAL_TERMINAL b<3>
FORMAL_TERMINAL b<2>
FORMAL_TERMINAL b<1>
FORMAL_TERMINAL b<0>
        TERM_EDGE BOTTOM
        DIRECTION OUTPUT
FORMAL_TERMINAL cout<3>
FORMAL_TERMINAL cout<2>
FORMAL_TERMINAL cout<1>
FORMAL_TERMINAL cout<0>
```

Wiring Description (alu.bdnet)

```
MODEL alu:unplaced;
    TECHNOLOGY scmos;
    VIEWTYPE SYMBOLIC;
    EDITSTYLE SYMBOLIC;

    INPUT a<3:0> b<3:0> cin:c<0> opcode<2:0>;
    OUTPUT sum<3:0>, cout:c<4>;
    SUPPLY VDD;
    GROUND GND;

    INSTANCE cla:symbolic
        c: c<0>;
        a<3:0> : a<3:0>;
        b<3:0> : b<3:0>;
        cout<3:0> : c<4:1>;
        "wolfe_Vdd" : VDD;
        "wolfe_Gnd" : GND;

    ARRAY %i from 0 to 3 of
    INSTANCE "alu-bit":symbolic
        opcode<2:0> : opcode<2:0>;
        cin : c<%i>;
        a : a<%i>;
        b : b<%i>;
        s : sum<%i>;
        cout: UNCONNECTED;
        "wolfe_Vdd" : VDD;
        "wolfe_Gnd" : GND;
ENDMODEL;
```

Simulation Input (alu.musa)

```
! should define a macro
mv a a<3:0>
mv b b<3:0>
mv sum sum<3:0>
mv op opcode<2:0>
macro add
    set a $1
    set b $2
    set cin $3
    set op 000
    ev
    show a b cin cout sum
$end
macro xor
    set a<0> $1
```

```
    set b<0> $2
    set op 011
    ev
    verify cout 0
    show a b sum
$end
macro clr
    set op 101
    ev
    verify cout 0
    verify sum 0
$end
add 0000 0000 0
add 0000 0000 1
add 0110 1101 0
add 1111 1111 1
add 1001 1000 0
add 0101 0100 1
add 0001 0001 0
add 0001 0001 1
!
xor 0000 0000
xor 0000 0001
xor 0101 0100
xor 0001 0001
!
clr
!
exit
```

Pad Placement Description (alu.padlist)

```
#
# Formal Terminal description for alu:unplaced
# padplace recognizes the following keywords:
# TERMTYPE (SUPPLY | GROUND | SIGNAL | CLOCK)
# DIRECTION (INPUT | OUTPUT | INPUT)
# TERM_FLOATING_EDGES (LEFT|RIGHT|TOP|BOTTOM)
# TERM_EDGE (LEFT | RIGHT | TOP | BOTTOM)
# TERM_ROW ( 0 | 1 | 2 | ... | n)
# TERM_RELATIVE_POSITION ( 0.0 <= x <= 1.0)
# TERM_RELATIVE_POSITION_STEP ( 0.0 <= x <= 1.0)
# PEAKCURRENT ( x (in Amperes)) // Applies only to SUPPLY and GROUND terminals

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
        TERMTYPE SIGNAL
        DIRECTION INPUT
        TERM_ROW 0
        TERM_EDGE LEFT
FORMAL_TERMINAL cin
FORMAL_TERMINAL a<0>
FORMAL_TERMINAL a<1>
FORMAL_TERMINAL a<2>
FORMAL_TERMINAL a<3>
        TERM_EDGE TOP
FORMAL_TERMINAL opcode<0>
FORMAL_TERMINAL opcode<1>
FORMAL_TERMINAL opcode<2>
FORMAL_TERMINAL b<0>
FORMAL_TERMINAL b<1>
        DIRECTION OUTPUT
        TERM_EDGE RIGHT
FORMAL_TERMINAL sum<0>
FORMAL_TERMINAL sum<1>
```

```
FORMAL_TERMINAL sum<2>
FORMAL_TERMINAL sum<3>
FORMAL_TERMINAL cout
        TERM_EDGE BOTTOM
        TERMTYPE SUPPLY
        DIRECTION INOUT
FORMAL_TERMINAL VDD
        TERMTYPE GROUND
FORMAL_TERMINAL GND
        TERMTYPE SIGNAL
        DIRECTION INPUT
FORMAL_TERMINAL b<2>
FORMAL_TERMINAL b<3>
```

1.6 A BCD Counter with 7 Segment Output

Paul B. Cohen
Massachusetts Microelectronics Center

In this lab we will create a chip which will include a BCD (**B**inary **C**oded **D**ecimal) counter and a BCD to 7 segment decoder. This simple design will serve as an example of the use of the Oct tools to create a complete chip. Table 1.2 lists the tools used in this exercise. (The notation "(Script)" indicates the tool is actually a "shell script" which runs other tools to accomplish the stated function.)

NOTE: given the complexity of this exercise, you may find it convenient to using the design manager VOV (see the VOV tutorial).

Tool Name	Function
BDNET	Netlist to OCT translator
BDSYN	BDS translator for logic networks
MUSA	Multi-level simulator
MOSAICO	Macro cell router (Script)
OCTPLA	Static CMOS PLA generator (Script)
OCTPROP	Add Oct properties to a facet
PADPLACE	Pad placement tool
TIMBERWOLFSC	Standard cell place & route
WOLFE	OCT Interface to TimberWolf

Table 1.2: Tool list

1.6.1 Specification

Figure 1.5 shows a symbol for the chip, with the I/O signals labeled. Table 1.3 lists the signals along with a brief description.

For simplicity, we will label the segments `seg<6>` through `seg<0>`, rather than the standard `a` – `g`, as shown in figure 1.6. The segment pattern for each number is shown in table 1.4

When the `reset` signal is low, the counter is asynchronously reset, and the 7 segment outputs will all be high except for `seg<0>`, which will be low. This will display the number "0" on a 7 segment display. The outputs will not change until the `reset` signal is set to logic 1.

When the `reset` signal is high, each high to low (1 to 0) transition of `clk` (clock) will cause the outputs to increment the displayed number until that number reaches "9". When the output value is the 7 segment pattern for a "9", the next 1 to 0 transition on `clk` will cause the output pattern to return to the "0" pattern.

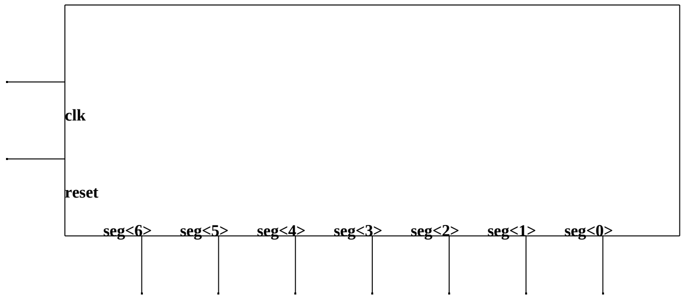

Figure 1.5: BCD counter / 7 Segment Decoder Chip

Signal Name	Direction	Description
clk	Input	Master Clock
reset	Input	Master reset (Asynchronous)
seg$< 6 : 0 >$	Output	7 Segment Display Outputs

Table 1.3: Signal Descriptions

1.6.2 Partitioning

Though there are alternatives, we will partition this design into three blocks. The BCD to 7 segment decoder will be one block, while the BCD counter will actually be broken into two blocks, with the flip-flops and control logic each becoming a block.

1.6.3 Block Design

Counter control logic

We can use BDS, the behavioral language in the OCT suite, to define the functionality of the counter control logic. A sample BDS file is shown below.

```
model count ns<3:0> = ps<3:0>;
routine main;
   ns = ps + 1;
   if (ps GEQ 9) then ns = 0;
endroutine;
endmodel;
```

Using a text editor, enter the above description into a file named `count.bds`. (Filenames are not critical, but using the suggested names will avoid confusion later.)

Bdsyn will convert the BDS description into a BLIF (**B**erkeley **L**ogic **I**ntermediate **F**ormat). **Octpla** uses the BLIF description to generate a PLA layout of the logic. Synthesize the layout with the commands:

```
% bdsyn -A -t -o count.blif count.bds
```

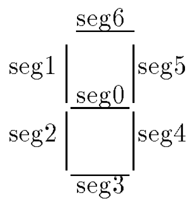

Figure 1.6: 7 segment display

Decimal Number	seg < 6 >	seg < 5 >	seg < 4 >	seg < 3 >	seg < 2 >	seg < 1 >	seg < 0 >
0	on	on	on	on	on	on	off
1	off	on	on	off	off	off	off
2	on	on	off	on	on	off	on
3	on	on	on	on	off	off	on
4	off	on	on	off	off	on	on
5	on	off	on	on	off	on	on
6	on	off	on	on	on	on	on
7	on	on	on	off	off	off	off
8	on	on	on	on	on	on	on
9	on	on	on	on	off	on	on

Table 1.4: 7 segment display decoder truth table

```
% octpla -nofold count
```

You can ignore the warning messages produced by vulcan.

(The **octpla** script invokes a number of tools, and will print a few messages to the screen as it runs.)

Vem can be used to view the resulting layout. If you want to do this, enter the command:

```
vem -G-0-0 -F count:pla -G+100+100 &
```

This will create a **vem** console window in the lower right hand corner of the screen, and a graphics window with the layout near the upper left of the screen. When you are done examining the layout, move the cursor to the **vem** console window and enter `^d` (Control d) or `:close-window` to exit **vem**.

It would be a good idea to simulate the counter control logic at this point. If there are any problems, they will be easier to diagnose on a small block. We will use the multi-level simulator **musa** to check the logic. Enter the command:

```
% musa count:pla
```

The following series of **musa** commands will check the counter control logic. (Anything after the "!" is a comment.) *As the commands are entered,* **musa** *will perform the indicated action.*

```
mv ps ps<3:0> ! create a 4 bit vector named ``ps''
mv ns ns<3:0> ! create a 4 bit vector named ``ns''
ma in         ! Define a macro called ``in''
se ps $1          ! set vector ps to the 1st argument
ev                ! evaluate the circuit
sh "%h %h" ps ns ! show ps and ns in hex format
$end              ! end the macro definition
in h0         ! invoke macro ``in'', check hex 0
in h1         ! continue to invoke ``in'', with
in h2         ! different arguments.
in h3         ! ``h'' (hex) notation expands
in h4         ! argument to 4 bits.
in h5
in h6
in h7
in h8
in h9
in ha         ! make sure illegal case is handled OK
quit
```

A sample simulation is shown below. Note that the prompt changes from `musa0>` to `>` while the macro `in` is being defined, and simulation results are indented 1 space.

```
musa0> mv ps ps<3:0>
musa0> mv ns ns<3:0>
musa0> ma in
> se ps $1
> ev
> sh "%h %h" ps ns
> $end
musa0> in h0
 ps:H0   ns:H1
musa0> in h1
 ps:H1   ns:H2
musa0> in h2
 ps:H2   ns:H3
musa0> in h3
 ps:H3   ns:H4
musa0> in h4
 ps:H4   ns:H5
musa0> in h5
 ps:H5   ns:H6
musa0> in h6
 ps:H6   ns:H7
musa0> in h7
 ps:H7   ns:H8
musa0> in h8
 ps:H8   ns:H9
musa0> in h9
 ps:H9   ns:H0
musa0> in ha
 ps:HA   ns:H0
musa0> quit
```

The counter control logic is now complete.

Flip-Flops

We will use flip-flops from the MSU (Mississippi State University) library for this design. We can create an OCT cell with 4 flip-flops using the netlist to OCT translator, **bdnet**. To do this, create a file, `reg.bdnet` with the following contents:

```
model reg:logic;

TECHNOLOGY scmos;
VIEWTYPE SYMBOLIC;
EDITSTYLE SYMBOLIC;

INPUT reset;
INPUT d<3:0>;
CLOCK clk;
SUPPLY Vdd;
GROUND GND;
OUTPUT q<3:0>;

ARRAY %x from 0 to 3 of
INSTANCE $OCTTOOLS/tech/scmos/msu/stdcell2_2/dfrf301:physical
     DATA1:d<%x>;
     CLK2:clk;
     RST3:reset;
     Q:q<%x>;
     "Vdd!":Vdd;
     "GND!":GND;
ENDMODEL;
```

The OCT view `reg:logic` is created with the command:

```
% bdnet reg.bdnet
```

Again, it would be a good idea to simulate the cell using **musa**. A sample simulation run is shown below. (**Musa** commands entered by the user in the sample run are preceeded by the prompt, `musa0>`. **Musa** uses the letters ``I'' and ``O'' to indicate logic levels set to high and low, respectively, by the user, and the numbers ``1'' and ``0'' to indicate simulation results.)

```
% musa reg:logic

musa0> mv d d<3:0>
musa0> mv q q<3:0>
musa0> ma cycle
> se clk 1
> ev
> se clk 0
> ev
> sh "%b %b %b" reset d q
> $end
musa0> se reset 0 ! reset all 4 ff's
musa0> ev
musa0> sh "%b %b %b" reset d q
 reset:O   d:xxxx   q:0000
musa0> se reset 1 ! won't do much with reset low
musa0> se d h5    ! set inputs to 0101
musa0> cycle      ! cycle clock 1 time
 reset:I   d:OIOI   q:0101
musa0> se reset 0 ! make sure reset is asynch
musa0> ev
musa0> sh "%b %b %b" reset d q
 reset:O   d:OIOI   q:0000
musa0> se reset 1 ! try a few more values
musa0> se d ha
musa0> cycle
 reset:I   d:IOIO   q:1010
musa0> se d h0
musa0> cycle
 reset:I   d:OOOO   q:0000
musa0> se d hf
musa0> cycle
 reset:I   d:IIII   q:1111
musa0> quit       ! try more if you'd like
```

We will use **padplace** to define the I/O for this cell. Create a list of signals with the command:

```
% padplace -l reg:logic > reg.padlist
```

This will create a file, `reg.padlist` which will look something like:

```
#
# Formal Terminal description for reg:logic
# padp recognizes the following keywords:
# TERMTYPE (SUPPLY | GROUND | SIGNAL | CLOCK) : DEFAULT = SIGNAL
# DIRECTION (INPUT | OUTPUT | INOUT) : DEFAULT = INOUT
# PEAKCURRENT ( x (in Amperes)) : DEFAULT = 0.0
#   NOTE: PEAKCURRENT applies only to SUPPLY and GROUND terminals
# TERM_FLOATING_EDGES (ALL | LEFT|RIGHT|TOP|BOTTOM)
# TERM_EDGE (LEFT | RIGHT | TOP | BOTTOM) : DEFAULT = none
# TERM_ROW ( 0 | 1 | 2 | ... | n) : DEFAULT = 0
# TERM_RELATIVE_POSITION ( 0.0 <= x <= 1.0) : DEFAULT = none
# TERM_RELATIVE_POSITION_STEP ( 0.0 <= x <= 1.0) : DEFAULT = 0.0

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
     TERMTYPE SIGNAL
     DIRECTION INPUT
     TERM_ROW 0
FORMAL_TERMINAL reset
FORMAL_TERMINAL d<3>
FORMAL_TERMINAL d<2>
FORMAL_TERMINAL d<1>
FORMAL_TERMINAL d<0>
     TERMTYPE CLOCK
```

```
FORMAL_TERMINAL clk
     TERMTYPE SUPPLY
     DIRECTION INOUT
FORMAL_TERMINAL Vdd
     TERMTYPE GROUND
FORMAL_TERMINAL GND
     TERMTYPE SIGNAL
     DIRECTION OUTPUT
FORMAL_TERMINAL q<3>
FORMAL_TERMINAL q<2>
FORMAL_TERMINAL q<1>
FORMAL_TERMINAL q<0>
```

We will once again use **padplace** to add the I/O information to the OCT cell, this time adding the information to the original facet with the command:

```
% padplace -D reg.padlist reg:logic
```

The last step in assembling the register cell is to actually place and route the flip-flops. **Bdnet** created logical connections, we will use **wolfe** (which invokes **TimberWolfSC**) to perform the actual place and route of the individual cells. Enter the commands:

```
% wolfe -o reg:placed reg:logic
```

When **TimberWolf** is invoked, each step of the simulated annealing process will be indicated by a number. There are 123 steps.

The register cell is now complete. If you wish, it can be re-simulated (as above, except type `musa reg:placed` instead of `musa reg:logic`).

Vem can be used to view the resulting layout. If you want to do this, enter the command:

```
% vem -G-0-0 -F reg:placed -G+100+100 &
```

This will create a **vem** console window in the lower right hand corner of the screen, and a graphics window with the layout near the upper left of the screen. When you are done examining the layout, move the cursor to the **vem** console window and enter `^d` (Control d) or `:close-window` to exit **vem**.

Decoder

We will define the logic for the BCD to 7 segment decoder block with the BDS hardware behavioral language. Create a file named `decode.bds` with the following contents:

```
model bcd_7seg seg<6:0> = data<3:0>;
routine decode;
     select data from
          [0#16]: seg=7E#16;
          [1#16]: seg=30#16;
          [2#16]: seg=6D#16;
          [3#16]: seg=79#16;
          [4#16]: seg=33#16;
          [5#16]: seg=5B#16;
          [6#16]: seg=5F#16;
          [7#16]: seg=70#16;
          [8#16]: seg=7F#16;
          [9#16]: seg=7B#16;
          [otherwise]: seg=00#16;
     endselect;
endroutine;
endmodel;
```

We will need to create a BLIF file for the decoder (`decode.blif`) with **bdsyn** and create the PLA layout **octpla**. Enter the following 3 commands:

```
% bdsyn -A -t -o decode.blif decode.bds
% octpla -nofold decode
```

Musa can be used to simulate the decoder. Start **musa** with the command:

```
% musa decode:pla
```

A sample simulation run follows. Note the use of the verify command. Since we know what the output should be, the verify command instructs **musa** to advise us only if the output differs from our expectations.

```
musa0> mv s seg<6:0>
musa0> mv d data<3:0>
musa0> ma disp
> se d $1
> ev
> sh "%h %b" d s
> $end
musa0> ma chk
> se d $1
> ev
> verify s $2
> $end
musa0> disp h0
 d:H0   s:1111110
musa0> disp h1
 d:H1   s:0110000
musa0> chk h2 h6d
musa0> chk h3 h79
musa0> chk h4 h33
musa0> chk h5 h5b
musa0> chk h6 h5f
musa0> chk h7 h70
musa0> chk h8 h7f
musa0> chk h9 h7b
musa0> chk ha h00
musa0> chk hb h00
musa0> chk hc h01 ! intentionally wrong to show what happens
musa: Warning: Verify failed: s has value 0000000 not 00000001.
musa0> chk hc h00
musa0> chk hd h00
musa0> chk he h00
musa0> chk hf h00
musa0> quit
```

The decoder is now complete **Vem** can be used to view the resulting layout. If you want to do this, enter the command:

```
% vem -G-0-0 -F decode:pla -G+100+100 &
```

This will create a **vem** console window in the lower right hand corner of the screen, and a graphics window with the layout near the upper left of the screen. When you are done examining the layout, move the cursor to the **vem** console window and enter `^d` (Control d) or `:close-window` to exit **vem**.

1.6.4 Assembling the Chip

Connecting the Blocks

Bdnet, the netlist to OCT translator, will be used to connect the blocks. Create a file named `bcd.bdnet` with the following contents:

```
model bcd:logic;

TECHNOLOGY scmos;
VIEWTYPE SYMBOLIC;
EDITSTYLE SYMBOLIC;

INPUT reset;
CLOCK clk;
```

```
SUPPLY Vdd;
GROUND GND;
OUTPUT seg<6:0>;

INSTANCE reg:placed PROMOTE;
INSTANCE decode:pla PROMOTE
     data<3>:q<3>;
     data<2>:q<2>;
     data<1>:q<1>;
     data<0>:q<0>;
INSTANCE count:pla PROMOTE
     ns<3>:d<3>;
     ns<2>:d<2>;
     ns<1>:d<1>;
     ns<0>:d<0>;
     ps<3>:q<3>;
     ps<2>:q<2>;
     ps<1>:q<1>;
     ps<0>:q<0>;
ENDMODEL;
```

Create the OCT cell `bcd:logic` with the command:

```
% bdnet bcd.bdnet
```

When this is complete, you may want to simulate the resulting circuit. Since we will want to resimulate the circuit after the pads are added and the blocks are placed and routed, we should create a simulation file. For this simulation, we will plot the results on the screen as the simulation proceeds. Create a file, `bcd.sim`, with the following contents:

```
mv seg seg<6:0> ! create a vector, "seg"
ma c            ! create a macro, "c" to cycle the clock
se clk 1
ev
se clk 0
ev
sh "%b %b" reset seg
$end
setplot plot1 1000 300 0 500
plot plot1 clk seg<6> seg<5> seg<4> seg<3> seg<2> seg<1> seg<0>
se reset 0      ! make sure reset works
ev
sh "%b %b" reset seg
c
se reset 1
rp 10 c
se reset 0
ev
sh "%b %b" reset seg
se reset 1
rp 10 c
```

The `quit` *command should not be part of this file, since that would cause the plot to disappear before it could be examined.*

To invoke **musa** and read the command file, enter:

```
% musa bcd:logic
```

When the musa prompt appears, type:

```
musa> source bcd.sim
```

The plot window has its own menu, which you are encouraged to explore. A particularly useful command is `show-all`. To execute this command, move the mouse to the plot window, and press the middle mouse button. This will cause a small menu similar to figure 1.7 to appear on the screen.

xg	
Display	$\Rightarrow$
Flags	$\Rightarrow$
Misc	$\Rightarrow$

Figure 1.7: Plot menu

Move the cursor over the small arrow to the right of the Display entry, and a sub-menu will appear. Move the cusor over the `show all` entry, and release the mouse button. (If you already released it, just click it again over the `show all` command.)

When you are satisfied with the results, return the cursor to the terminal window and type:

```
quit
```

to leave **musa**.

Adding Pads

Padplace will be used to add bonding pads to the design. Often, it is desirable to constrain the order of the pads around the chip. We will do this by first creating a file with the I/O signals listed, then editing that file to constrain the order of the pads.

Create the I/O listing with the command:

```
% padplace -l bcd:logic > bcd.padlist
```

This will create a file, `bcd.padlist` with the signals for the chip listed. To constrain the order of the pads, edit `bcd.padlist` so it resembles the sample file, below.

```
#
# Formal Terminal description for bcd:logic
# padp recognizes the following keywords:
# TERMTYPE (SUPPLY | GROUND | SIGNAL | CLOCK) : DEFAULT = SIGNAL
# DIRECTION (INPUT | OUTPUT | INOUT) : DEFAULT = INOUT
# PEAKCURRENT ( x (in Amperes)) : DEFAULT = 0.0
#   NOTE: PEAKCURRENT applies only to SUPPLY and GROUND terminals
# TERM_FLOATING_EDGES (ALL | LEFT|RIGHT|TOP|BOTTOM)
# TERM_EDGE (LEFT | RIGHT | TOP | BOTTOM) : DEFAULT = none
# TERM_ROW ( 0 | 1 | 2 | ... | n) : DEFAULT = 0
# TERM_RELATIVE_POSITION ( 0.0 <= x <= 1.0) : DEFAULT = none
# TERM_RELATIVE_POSITION_STEP ( 0.0 <= x <= 1.0) : DEFAULT = 0.0

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
        TERM_EDGE TOP
        TERM_RELATIVE_POSITION 0
        TERM_RELATIVE_POSITION_STEP .2
        TERMTYPE SIGNAL
        DIRECTION OUTPUT
FORMAL_TERMINAL seg<0>
        DIRECTION INPUT
FORMAL_TERMINAL reset
        TERMTYPE SUPPLY
FORMAL_TERMINAL Vdd
        TERM_EDGE BOTTOM
        TERM_RELATIVE_POSITION 0
        TERM_RELATIVE_POSITION_STEP .3
        TERMTYPE CLOCK
FORMAL_TERMINAL clk
        TERMTYPE GROUND
FORMAL_TERMINAL GND
        TERMTYPE SIGNAL
        DIRECTION OUTPUT
        TERM_EDGE LEFT
```

```
        TERM_RELATIVE_POSITION 0
        TERM_RELATIVE_POSITION_STEP .1
FORMAL_TERMINAL seg<1>
FORMAL_TERMINAL seg<2>
FORMAL_TERMINAL seg<3>
        TERM_EDGE RIGHT
        TERM_RELATIVE_POSITION 0
        TERM_RELATIVE_POSITION_STEP .1
FORMAL_TERMINAL seg<4>
FORMAL_TERMINAL seg<5>
FORMAL_TERMINAL seg<6>
```

Once this has been done, we can add the bond pads to the chip. At this time, there are two pad libraries included with the OCT tools. We will use the pad library `m2c_2u`. The alternative is `sc2`. Create a new OCT facet, `bcd:padp` with the pads included by entering:

```
% padplace -D bcd.padlist -u m2c_2u -o bcd:padp bcd:logic
```

Place and Route the Blocks

Puppy can be used to place the blocks. Puppy will try to optimally place the blocks within the pads, which were added by padplace. The command:

```
% puppy -o bcd:pup bcd:padp
```

will perform the block placement. If this is the first time you are running puppy on this design, you will see some messages:

```
puppy: Message: Could not find bcd.parPuppy
puppy: Message: Cannot even find parPuppy
puppy: Message: You can create the parPuppy file using the menu
puppy: Message: Copying bcd:padp into bcd:pup
puppy: Message: Could not find bcd.constraintsPuppy
puppy: Message: Cannot even find constraintsPuppy
```

which can be ignored. Puppy will open a series of windows so you can watch and alter the progress. The "-g" option on puppy is useful if you are not at a graphics terminal.

Mosaico can now be used to route the blocks. **Mosaico** is actually a shell script, which invokes a host of tools to perform the routing, and to perform various checks along the way. Mosaico will take quite a while to run, and many of the tools are quite verbose as they run. Warnings from some of the tools are to be expected, mosaico will stop if there is a fatal error.

Enter the command:

```
% mosaico -c -s -o bcd:wired bcd:pup
```

Vem can be used to view the resulting layout and wiring created by puppy and mosaico. If you want to do this, enter the command:

```
% vem -G-0-0 -F bcd:wired -G+100+100 &
```

This will create a **vem** console window in the lower right hand corner of the screen, and a graphics window with the layout near the upper left of the screen. When you are done examining the layout, move the cursor to the **vem** console window and enter `^d` (Control d) or `:close-window` to exit **vem**.

The power and ground rings will need to be competed, and the entire chip compacted. Power and ground rings are added by padplace:

```
% padplace -R -o bcd:rings bcd:wired
```

You may see some warnings:

```
padplace: Message: PADS are not moved
padplace: Warning: Net Vdd_ring exists already. Cannot route it as a ring.
padplace: Warning: Net Vdd_guard_ring exists already. Cannot route it as a ring.
```

which can be ignored.

Final compaction is done using sparcs.

```
% sparcs -t -o bcd:final bcd:rings
```

Vem can be used to view the final chip layout. If you want to do this, enter the command:

```
% vem -G-0-0 -F bcd:final -G+100+100 &
```

This will create a **vem** console window in the lower right hand corner of the screen, and a graphics window with the layout near the upper left of the screen. When you are done examining the layout, move the cursor to the **vem** console window and enter `^d` (Control d) or `:close-window` to exit **vem**.

1.6.5 Summary

Congratulations on the completion of the chip!

We have used a variety of "paths" in the OCT tool suite to implement a chip using a variety of layout styles for the different blocks. This is by no means the only way we could have completed this chip. As you become more familiar with the tools, you may wish to explore alternate implementations.

1.7 Wiring to Fixed Padframes

Paul B. Cohen
Massachusetts Microelectronics Center
paul@m2c.org

This tutorial covers the two primary methods for wiring some arbitrary logic to a fixed padframe in the Octtool suite. **Mosaico** can, in many cases, automate the task of wiring the logic to the pads. When mosaico cannot be used, VEM can be used to accomplish the padframe wiring manually.

In this tutorial, we will cover the wiring of small block of logic to an M^2C standard 40 pin frame both automatically and manually. The procedures for wiring to a MOSIS tiny frame are similar. We will create a small block of logic, and demonstrate how to create an M^2C padframe, instantiate the logic, and perform the wiring.

It is assumed the reader is familiar with the Oct tools, specifically VEM. You should have completed the the "Symbolic Editing with Vem" or have gained familiarity with symbolic editing on your own. Other tools used in this tutorial include:

- bdsyn
  - The BDS hardware description language
- bdnet
  - The bdnet netlist language
- misII
- wolfe
- padplace
- mosaico

Since files and instructions will be provided, extensive knowledge of the additional tools is not required.

For "advanced" users of the Octtools, you may find it sufficient to simply read this tutorial, then jump right in and wire your own design to the pads. If you choose to do this, then you can skip the section 1.7.1, "Creating the Logic", but you should read the rest of the tutorial.

Fixed Padframes

Fixed padframes are fairly common for designs which are to be fabricated on multi-project wafers. Although the area occupied by the logic on each design will vary, subsequent steps in the fabrication and packaging of the individual designs argue for a fixed die sie, or at least a small number of fixed sizes. Within those fixed die sizes, it is also advantageous to standardize the location of the bond pads. This simplifies the bond process, especially when only a few copies of each design are being packaged, and eliminates violations of bonding rules.

M^2C and MOSIS distribute standardized pad frames, which fix the locations of the bond pads, and also include input and output buffer circuitry. In general, the type of pad (input, output, bidirectional, V_{DD}, or GROUND) doesn't matter (MOSIS places some restrictions on V_{DD} and GROUND), but the pad cells are generally designed such that the bond pad itself is in the same location regardless of the driver circuitry.

1.7.1 Creating the Logic for the Sample Circuit

Since the creation of the logic itself isn't really the purpose of the tutorial, we will create a simple 4 bit binary adder, with a carry in and carry out. Create a file, `adder.bds`, with the following contents:

Pin Number	Signal Name	Pin Number	Signal Name
1	$a<3>$	40	V_{DD}
2	$a<2>$	39	$cout$
3	$a<1>$	38	$sum<3>$
4	$a<0>$	37	$sum<2>$
5	$b<3>$	36	$sum<1>$
6	$b<2>$	35	$sum<0>$
7	$b<1>$		
8	$b<0>$		
9	cin		
20	GND		
Unspecified pins are GND			

Table 1.5: Pin configuration for the adder chip.

```
model adder sum<3:0>,cout = a<3:0>,b<3:0>,cin;
state c<3:0>;
routine main;
c = a + b + cin;
sum = c;
if (c GEQ F#16) then cout = 1 else cout = 0;
endroutine;
endmodel;
```

To implement this logic using the Mississippi State University (MSU) standard cells, execute the following 2 commands;

```
% bdsyn -A -t -o adder.blif adder.bds
% misII -f script.msu -T oct -o adder:logic adder.blif
```

Do not create the layout for the adder yet. We will do this in section 1.7.3, after we add some layout constraints.

1.7.2 Adding Signal Placement Constraints

For manual wiring, it helps to constrain the layout of the logic so the order of the input and output signals to the logic is roughly the same as the order of the pads. Although the choice is quite arbitrary, we will use the pinout in table 1.5 for the chip.

Since it is not a good idea to have floating input signals in a CMOS circuit, we will use GROUND pads (gndpad) for all unused signals.

Padplace is the tool which allows us to constrain the order of the pads around the logic when the layout is created. The first step is to create a file with all of the signals listed. This is accomplished with padplace:

```
% padplace -l adder:logic > adder.padplace
```

You can then edit the file `adder.padplace` to set the order for the signals. A sample file is shown in appendix 1.8.1. You can then use this file to add the placement constraints for the signals.

```
% padplace -D adder.padlist adder:logic
```

1.7.3 Creating the Layout

We can now use **wolfe** to create the layout of the adder. Since this is a small layout, wolfe may not be able to keep the signals in exactly the order we want, but the outcome should be close.

```
% wolfe -o adder:placed adder:logic
```

1.7.4 Positioning the Layout in the Padframe

In order to position the logic near the center of the padframe, we will need to know the size of the layout. **Chipstats** will provide this information.

```
% chipstats adder:placed
```

The size of the layout is approximately $449\mu m \times 383\mu m$. The "origin" of the layout cell (coordinates [0,0]), are near the lower left, so we know that the center of this layout is near [224.5,191.5]. (The V_{DD} connection has a negative X coordinate, but this is only a few microns wide, and won't affect the positioning significantly.)

From the **Design Preparation Guide**[1] we know that the size of the M^2C standard frame is $4560\mu m \times 4560\mu m$. We can determine that the center of the padframe, therefore, is at coordinates [2280,2280].

In order to center the logic, coordinates [224.5,191.5] on the logic should correspond to [2280,2280] on the padframe. We can subtract and determine the origin of the logic should be at [2055.5,2088.5] in order to be centered.

All of these coordinates were computed in terms of microns, but the Octtools use Oct unit internally. Normally, the relationship is 20 $Oct\ Units = 1\lambda$, and for a $2\mu m$ process, $1\lambda = 1\mu m$. So, we can translate the coordinates of the logic in the padframe by multiplying by 20, resulting in [41110,41770].

1.7.5 Creating the Padframe with Bdnet

A template for the M^2C and MOSIS padframes is included in the Octtools distribution from M^2C or UC Berkeley. The template for the M^2C standard frames (40 and 64 pins) can be found in the directory:

```
$OCTTOOLS/tech/scmos/pads/m2cframe
```

To create an M^2C standard 40 pin frame, you should copy the file

```
$OCTTOOLS/tech/scmos/pads/m2cframe/m2cframe.bdnet
```

from the `m2cframe` directory to your working directory, since you will need to edit the file to customize the pad frame. The template is a netlist file in bdnet format. There are 40 INSTANCE statements, one for each pin. Each INSTANCE statement instantiates a pad and driver, and places it in a specific location. The placement is done by the coordinate pair in brackets ([X,Y]) and may also rotate the pad ([X,Y,ROTATION]). These numbers and rotations are *critical*, and should not be changed.

Although the pad location and rotation cannot be changed, the various driver types have been designed such that they are interchangeable. In the template, pin 1 is an output with the input to the pad cell connected to GND.

```
INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/output":physical
        [38000,83600] NAME=pin1;
        "io_pad":UNCONNECTED;
        "to_pad":GND;
        "Vdd_ring":UNCONNECTED;
        "GND_ring":UNCONNECTED;
```

In our circuit, we want pin 1 to be an input, with the output of the buffer connected to $a<3>$. We can edit our copy of the file to change the type of driver located at pin 1, and connect the *from_pad* signal to $a<3>$ as follows:

[1] The **Design Preparation Guide** is available from M^2C. It is included in the documentation shipped with the M^2C CAD Tools Release, and is available "on-line".

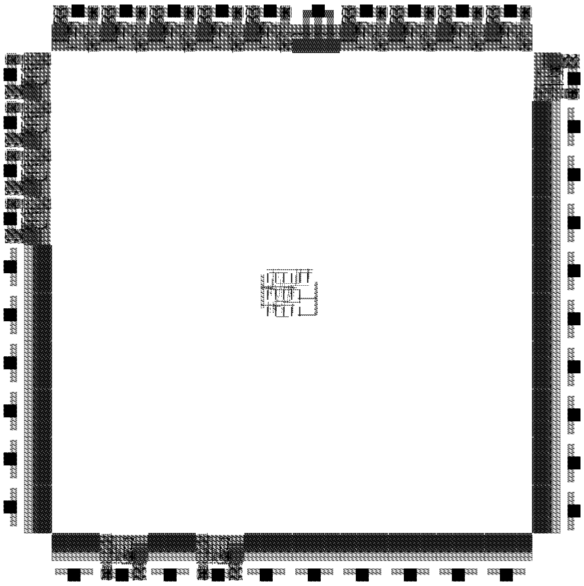

Figure 1.8: Adder chip, with M^2C padframe, before wiring.

```
INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/input":physical
        [38000,83600] NAME=pin1;
        "io_pad":UNCONNECTED;
        "from_pad":a<3>;
        "Vdd_ring":UNCONNECTED;
        "GND_ring":UNCONNECTED;
```

Note that the coordinates [38000,83600] are unchanged.

We can also instantiate the logic by adding an INSTANCE command, and positioning the logic according to our calculations in section 1.7.4. The INSTANCE statment would look something like:

```
INSTANCE adder:placed [41110,41770] PROMOTE;
     "wolfe_Vdd":Vdd;
     "wolfe_Gnd":GND;
```

Since wolfe will, by default, add signals `wolfe_Vdd` and `wolfe_Gnd`, instead of `Vdd` and `GND`, we need to connect these signals in the bdnet file explicitly. Normally, if we used a bdnet file to assemble various parts of our design, and if that bdnet file indicated the SUPPLY net was `Vdd` and the GROUND net was `GND`, then the explicit connections would be unnecessary.

We can now proceed to edit the rest of the file to place the right pad type at each location, as specified in table 1.5 (page 36). This file is included in Appendix 1.8.2.

In the sample file, the name of the model was changed from `m2cframe:symbolic` to `adder:frame`. If you change the name of the model, you should also change the name of the facet in the portion of the file which adds properties to the padframe. After pin 40, you will note the FACET keyword, followed by `m2cframe:symbolic`. You should edit this `cell:view` pair to be the same as the `cell:view` in model name (`adder:frame`, in our case). In general, you should not edit anything else in after the ENDMODEL statement.

After you have edited the file (I called my edited version adderframe.bdnet), you must run bdnet:

```
% bdnet adderframe.bdnet
```

The resulting padframe, with the adder logic instantiated, should resemble figure 1.8. You can verify this with VEM.

At this point, the V_{DD} and GROUND rings through the pad cells are *not* completed near the corners of the chip. The corner cells to complete the V_{DD} and GROUND rings will be added later.

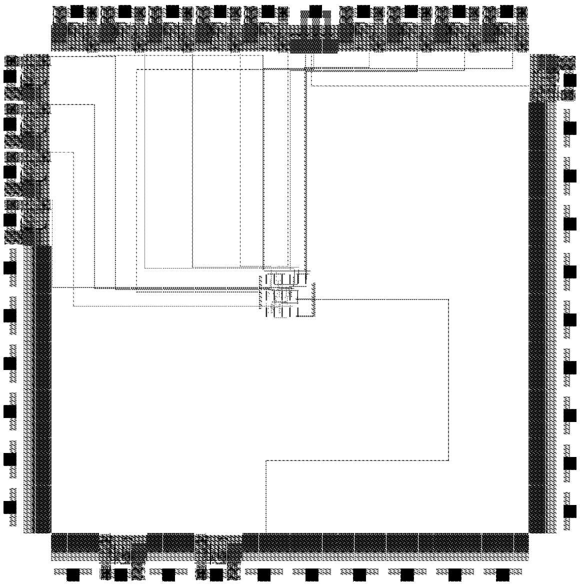

Figure 1.9: Adder chip, with M^2C padframe, after automatic wiring.

1.7.6 Automatic Wiring

Mosaico is a tool which will route a layout composed of arbitrary blocks. Once the padframe, with logic, has been created, simply execute the command:

```
% mosaico -c -s -o adder:wired adder:frame
```

You can type `mosaico -help` for an explanation of the command line options.

After a few minutes, your layout will look something like figure 1.9. You can use VEM to examine the layout.

As of this writing, mosaico sometimes moves the pads, even though they are supposed to be fixed. Near the end of the end of the mosaico run, `chipstats` is called to gather some statistics on the chip. One of the key statistics to look for is the chip area. For my mosaico run, it indicated the chip size was:

```
Chip Area W x H        =   4560 x  4570
```

Since the chip size should be $4560\mu m \times 4560\mu m$, this indicates mosaico moved the pads. One of our options at this point is to wire the padframe manually. Using the symbolic editing style in VEM, this isn't too bad a task.

There are also occasions where layout constraints, special placements, etc. indicate that manual wiring is preferred to an automatic solution.

1.7.7 Manual Wiring

Since we may want to try some additional experiments later on, we can make a copy of our original placed, unrouted layout. Since Oct uses directories to store the cells and views, we can use the UNIX copy command[2]:

```
% cp -r adder/frame adder/manual
```

While the "optimum" way to accomplish a specific task in VEM is largely a matter of personal style, I have found the following methodology to be effective.

[2] This is cheating. In general it is best to make no assumption on the structure used to store Oct facets, because it may change in the future.

Options editor for chip:frame				
[-]	Expert mode	Snap (oct units)	20	__I
[+]	Visible Grid	Major Grid Spacing	100	______I
[-]	Expand Instances	Bounding Threshold	5	_I
[+]	Visible Title Bar	Abstraction Threshold	0	I
Ok		Dismiss	Apply	Help

Figure 1.10: Window Options Dialog Box (simpler than actual dialog)

Start VEM and open a relatively small window with the layout. Since the re-draw time of the whole chip can be significant, I prefer this small window to be near the edge of the screen, so I don't cover it very often. The command:

```
% vem -G-0-0 -Fadder:manual -G400x400+0+350 &
```

will open a VEM console window in the lower right corner of the screen, and a graphics window near the lower left.

The width of the MET1 and MET2 wires at the edge of the core of the chip are all 3λ, with one edge on a λ grid. In order to connect to the centerline, we will need to be able to position our wiring on a $\frac{1}{2}\lambda$ grid. Since there are normally 20 Oct units per λ, we should set the "snap" parameter to 10 Oct units, or $\frac{1}{2}\lambda$. To do this, use the `window-options` command. This will bring up a dialog box similar to figure 1.10 . Note the option "Snap (oct units)" in the upper right. Using the left and right mouse buttons, set this to 10, instead of the default 20, by clicking the buttons with the cursor in the "slider" box next to the option.

Once this is set, click on **OK** to remove the dialog box.

It will be useful to have a palette of the available layers available. You can create one with the `palette` command in VEM.

Since there is a certain amount of randomness in the placement and routing of the logic, I will describe in detail the wiring of a few of the signals, as they appear on the layout I created while writing this tutorial. Your layout may vary, but the basic idea will remain the same.

It is a good idea to wire V_{DD} and GROUND first, so the number of vias can be minimized. Open another window, zoomed in to the upper left hand corner of the logic. This is done by creating a small box around the area of interest, and using the `open-window` command. In the new window, keep zooming in (create a box and use the `zoom-in` command) until you can read the label *wolfe_Vdd* near the top of the V_{DD} wire.

Position the cursor somewhere over the V_{DD} wire, and execute the `select-net` command. Note that the V_{DD} wire is highlighted. If you look in the "full chip" window, you will notice that the bottom edge of pin 40, the V_{DD} pad, is also highlighted. This is useful as a guide to determine where the "other end" of a signal is located. You can confirm your selection in the console window, where you should see the message:

```
net Vdd has been selected
```

Create a third window, zoomed in to the bottom edge of pin 40. Exactly where on the V_{DD} terminal you zoom is not important, but you should see the highlight on the screen. It is also a good idea to try to keep the three graphics windows, the palette, and the console window from overlapping.

Unselect everything with Ctrl-u (û).

It might be a good idea to read this entire paragraph before actually doing the operations called for. In the window zoomed to the top of the *wolfe_Vdd* signal, begin drawing a line upwards from the center of the series of vias near the top of the wire. Remaining in line drawing mode (ie, don't release the mouse button) move the cursor to the full chip window, and continue to draw the line upwards, until you are around half-way between the logic and the top of the pad frame. Bend the line (release and press the left mouse button, and begin moving the cursor to the right. When you are roughly under the V_{DD} pad, bend the line again and continue drawing the line upwards. Move the cursor to the window which is zoomed to the V_{DD} pad, and complete the line somewhere over the V_{DD} terminal. The position isn't critical, but sure to avoid the GROUND ring just above the V_{DD} terminal. Now move the cursor over the MET2 block on the palette, and execute the `create` command. The wire will be created connecting V_{DD} on the logic to the V_{DD} pad

You should be able to repeat the above procedure for the GROUND connection. Be careful not to get your screen too cluttered with windows. You don't need the windows zoomed to the ends of the V_{DD} wire any more - so you can re-use these windows. Create a small box around the lower right hand corner of the layout, then move the cursor to one of the zoomed windows and execute the `zoom` command. Zoom in until you the minor grid is enabled. The label *wolfe_Gnd* will be at the top of the wire, so it won't be visible here. Use the `select-net` command to find the other end of the *GND* net, which is pin 20, just to the left of center along the bottom row of pads. Zoom the other window to the *GND* terminal on pin 20.

Using a procedure similar to the V_{DD} net wiring, wire the *GND* net, also using MET2.

Normally, V_{DD} and GROUND connections should be fairly wide. The width of the wire at the end of the logic is 25λ for V_{DD} and GROUND, so let's make our wires 25λ also.

From any of the graphics windows, execute the command:

```
"25""MET2"w
```

which will set the wire width for MET2 to 25λ.

You should have a total of 6 wire segments – 3 each for V_{DD} andGROUND. Select all 6 segments. The simplest way to do this is by drawing 6 lines, each perpendicular to, and intersecting, one of the wire segments, then using the `select-objects` command. The number of objects selected will be confirmed in the console window. (At the scale of the full chip graphics window, some of the wire segments may not appear to be highlighted. Use the console window to verify the number of segments. If you're still not sure, zoom in on a segment to be sure it is selected.)

Move the cursor over the MET2 block in the palette, and execute the `change-layer` command. The wires will be changed to 25λ wide. Note that you could have changed the wire width before you created the wires, but it is very useful to know about the `change-layer` command.

Now change the wire width for MET2 back to 3λ by typing:

```
"3""MET2"w
```

from the graphics window.

It's a good idea to save your work once in a while, so do so now.

Having wired V_{DD} and *GND*, we can begin to wire the signals. Once again, zoom in near the upper left corner of the logic, and keep zooming until you can read the label on the "top edge" signal nearest the upper left. For my layout, this is $b < 3 >$. Yours may vary. From the pad list, we know that $b < 3 >$ should connect to pin 5, but select the net to confirm this. Zoom the other graphics window to the appropriate signal on the padframe for your layout.

Note that all of the signals along the top of the logic terminate in MET2. Also note that the signals at the pad drivers are available on both layers. You will need to connect your wire to the logic in the appropriate layer – MET2 for the top and bottom, MET1 for the left and right.

Since we're starting along the top, we'll need to connect to the logic using MET2. Beginning over the leftmost wire on the top of the logic, ($b < 3 >$ in my case), draw a line connecting the logic to the pin (pin 5 in my case). This will likely be a multiple segment line, just as you created for V_{DD} and *GND*. Create the wire in MET2.

For my layout, I crossed the $b < 3 >$ and V_{DD} signals, and both are on MET2. This will create a short on my chip. I can use the `change-layer` command to fix this.

Select the segment of the signal wire which crosses V_{DD}. Position the cursor over the MET1 block on the palette, and execute the `change-layer` command. Note that the segment is now on MET1. Zoom in to the ends of that segment, and note that VEM also added vias where there should be connections between MET1 and MET2.

Which Terminal?	
+	'cin' of 'adder:placed'
-	'wolfeVdd' of 'adder:placed'

Continue	Cancel	Help

Figure 1.11: Ambiguous terminal dialog box

Inappropriate Terminal
One of the segment endpoints intersects a terminal that is not the right size or layer. Press the Continue button to try to create a simple connector and connect the endpoint to that connector. Or Cancel the operation and try again.

Continue	Cancel

Figure 1.12: Inappropriate Terminal dialog box

Continue the procedure for the remaining signals. For simplicity, it may be useful to keep all horizontal segments in MET1, and all vertical segments in MET2. This makes subsequent wire crossings easier.

As you complete the wiring, there are a few dialog boxes you may come across. Most, such as the ambiguous terminal dialog (figure 1.11), are self-explanatory. The "Inappropriate Terminal dialog (figure 1.12) may seem more serious, but it really isn't. The geometry for the terminals on the pad cells are really on MET2, even though both layers are present in the layout. Thus, if you try to connect to the pads with MET1, VEM will note that there is not on the right layer. If you click on the "Continue" box, VEM will simply terminate the wire over the pad terminal, and make a logical connection. Since MET1 is actually present, the connection really will be made.

Manual Wiring to a MOSIS Frame

The procedures for wiring to a MOSIS Frame are similar, but the terminals on the pad cells may not be available on both MET1 and MET2. Be sure to make the connections on the right layer!

1.7.8 Adding the Corner Cells

Now that the wiring is complete, we will need to add the corner cells. For an M^2C padframe, these cells complete the V_{DD} and GROUND rings. A test inverter is also added by the standard template. For the MOSIS padframe, these cells are special V_{DD} and GROUND pads, which also complete the V_{DD} and GROUND rings.

Again, only the procedure for the M^2C padframe will be outlined. The procedures for the MOSIS frame are similar.

Copy the file:

```
$OCTTOOLS/tech/scmos/pads/m2cframe/m2cfinal.bdnet
```

to your working directory. This file is simply instantiates your wired padframe, the 4 corner cells, and a test inverter. You'll need to edit the file to instantiate your wired padframe, `adder:manual` for this tutorial. I renamed my copy `addercorners.bdnet`, and it is provided in appendix 1.8.3.

You should not change any of the coordinates in this file. Once you have edited the file to your satisfaction, run bdnet:

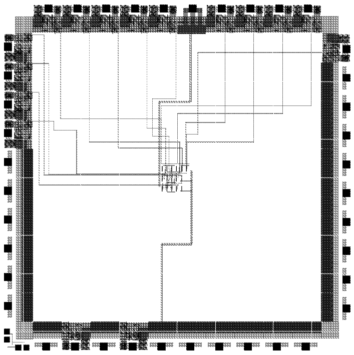

Figure 1.13: Final (manual) layout of the adder chip

```
% bdnet addercorners.bdnet
```

to complete your chip.

A sample layout of the adder chip is shown in figure 1.13.

1.7.9 Summary

We have gone through an exercise in wiring a block of logic to an M^2C standard 40 pin pad frame using the Oct tools. You should now understand how to wire your own chip to a padframe, using a similar methodology.

1.8 Files

1.8.1 Padplace File

```
#
# Formal Terminal description for adder:logic
# padp recognizes the following keywords:
# TERMTYPE (SUPPLY | GROUND | SIGNAL | CLOCK) : DEFAULT = SIGNAL
# DIRECTION (INPUT | OUTPUT | INOUT) : DEFAULT = INOUT
# PEAKCURRENT ( x (in Amperes)) : DEFAULT = 0.0
#   NOTE: PEAKCURRENT applies only to SUPPLY and GROUND terminals
# TERM_FLOATING_EDGES (ALL | {LEFT|RIGHT|TOP|BOTTOM})
# TERM_EDGE (LEFT | RIGHT | TOP | BOTTOM) : DEFAULT = none
# TERM_ROW ( 0 | 1 | 2 | ... | n) : DEFAULT = 0
# TERM_RELATIVE_POSITION ( 0.0 <= x <= 1.0) : DEFAULT = none
# TERM_RELATIVE_POSITION_STEP ( 0.0 <= x <= 1.0) : DEFAULT = 0.0

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
     TERMTYPE SIGNAL
     DIRECTION INPUT
     TERM_EDGE TOP
     TERM_RELATIVE_POSITION 0
     TERM_RELATIVE_POSITION_STEP 0.1
FORMAL_TERMINAL b<3>
FORMAL_TERMINAL a<3>
FORMAL_TERMINAL a<2>
FORMAL_TERMINAL a<1>
FORMAL_TERMINAL a<0>
     DIRECTION OUTPUT
FORMAL_TERMINAL cout
FORMAL_TERMINAL sum<3>
FORMAL_TERMINAL sum<2>
FORMAL_TERMINAL sum<1>
```

```
FORMAL_TERMINAL sum<0>
     DIRECTION INPUT
     TERM_EDGE LEFT
     TERM_RELATIVE_POSITION 0.5
     TERM_RELATIVE_POSITION_STEP 0.15
FORMAL_TERMINAL cin
FORMAL_TERMINAL b<0>
FORMAL_TERMINAL b<1>
FORMAL_TERMINAL b<2>
```

1.8.2 Padframe File Fragment

```
model adder:frame;

TECHNOLOGY scmos;
EDITSTYLE symbolic;
VIEWTYPE symbolic;

INSTANCE adder:placed [41110,41770] PROMOTE;
     "wolfe_Vdd":Vdd;
     "wolfe_Gnd":GND;


INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/input":physical
     [38000,83600] NAME=pin1;
     "io_pad":UNCONNECTED;
     "from_pad":a<3>;
     "Vdd_ring":UNCONNECTED;
     "GND_ring":UNCONNECTED;
INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/input":physical
     [30400,83600] NAME=pin2;
     "io_pad":UNCONNECTED;
     "from_pad":a<2>;
     "Vdd_ring":UNCONNECTED;
     "GND_ring":UNCONNECTED;
INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/input":physical
     [22800,83600] NAME=pin3;
     "io_pad":UNCONNECTED;
     "from_pad":a<1>;
     "Vdd_ring":UNCONNECTED;
     "GND_ring":UNCONNECTED;

 .... a lot more of the same ...

INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/input":physical
     [7600,53200,OCT_ROT90] NAME=pin9;
     "io_pad":UNCONNECTED;
     "from_pad":cin;
     "Vdd_ring":UNCONNECTED;
     "GND_ring":UNCONNECTED;
INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/gndpad":physical
     [7600,45600,OCT_ROT90] NAME=pin10;
     "gndpad":UNCONNECTED;
     "GROUND":UNCONNECTED;
     "Vdd_ring":UNCONNECTED;
     "GND_ring":UNCONNECTED;

 .... a lot more of the same ...

INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/vddpad":physical
     [45600,83600] NAME=pin40;
     "vddpad":UNCONNECTED;
     "Vdd":Vdd;
     "Vdd_ring":UNCONNECTED;
     "GND_ring":UNCONNECTED;
```

```
ENDMODEL;

(FACET adder:frame
     (IPROP PAD_MINIMUM_SEPARATION 0)
     (NET "Vdd" (SPROP NETTYPE "SUPPLY"))
     (NET "GND" (SPROP NETTYPE "GROUND"))
     (INSTANCE "pin1" (SPROP PLACEMENT_CLASS "PAD")
                (SPROP MOBILITY "FIXED"))
     (INSTANCE "pin2" (SPROP PLACEMENT_CLASS "PAD")
                (SPROP MOBILITY "FIXED"))
     (INSTANCE "pin3" (SPROP PLACEMENT_CLASS "PAD")
                (SPROP MOBILITY "FIXED"))
     (INSTANCE "pin4" (SPROP PLACEMENT_CLASS "PAD")
                (SPROP MOBILITY "FIXED"))
     (INSTANCE "pin5" (SPROP PLACEMENT_CLASS "PAD")
                (SPROP MOBILITY "FIXED"))
     (INSTANCE "pin6" (SPROP PLACEMENT_CLASS "PAD")
                (SPROP MOBILITY "FIXED"))

 .... a lot more of the same ...

     (INSTANCE "pin40" (SPROP PLACEMENT_CLASS "PAD")
                (SPROP MOBILITY "FIXED"))
)
```

1.8.3 File for Adding Corner Cells

```
model adder:final;

TECHNOLOGY scmos;
EDITSTYLE symbolic;
VIEWTYPE symbolic;

INSTANCE adder:manual [0,0] NAME=chip PROMOTE;

INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/corner":physical
     [7600,7600,OCT_ROT90] NAME=llcorner PROMOTE;
INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/corner":physical
     [83600,7600,OCT_ROT180] NAME=lrcorner PROMOTE;
INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/corner":physical
     [83600,83600,OCT_ROT270] NAME=urcorner PROMOTE;
INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/corner":physical
     [7600,83600] NAME=ulcorner PROMOTE;

INSTANCE "$OCTTOOLS/tech/scmos/pads/m2cframe/testinv":physical
     [0,0] NAME=testinv PROMOTE;


ENDMODEL;
```

1.9 A Short Introduction to Phyt

Paul B. Cohen
Massachusetts Microelectronics Center
M^{2}C

Phyt and **rpc-phyt** are utilities which allow you to add text to a chip design. The text will be present on the final chip. A common use is to add your name or initials, or other textual information which you would like to appear when your chip is fabricated. If the text is created on first or second metal (MET1 or MET2), then it will be readable on the chip with suitable magnification or very good eyes.

Phyt (which stand for **phy**sical **t**ext), reads a font palette, which can be created from the X11 font libraries. The palette is created by **bdf2oct**. The default font is 8x13. (Not all of the available fonts have been converted to palettes, so use 8x13 for now.)

1.9.1 Using rpc-phyt

Phyt can be used from a terminal (or terminal emulator window), but an "rpc" version also exists, which can be used from a **VEM** graphics window. To use rpc-phyt, start VEM, and open a **VEM** graphics window with the wired version of your design. I generally use the ":wired" version, which does not have the corner cells.

With the cursor in the **VEM** graphics window, enter the command:

`"$OCTTOOLS/bin/rpc/rpc-phyt"r`

This will start the rpc-phyt program. After a few seconds, the message "`rpcphyt initialized`" will appear in the **VEM** console window. You should not try to use rpc-phyt until it is initialized.

As with most rpc applications, rpc-phyt has its own menu, which can be invoked by holding down the Shift and clicking the middle mouse button. Don't do this now, though.

The default font is 8x13 (the terminal windows use the 8x13 font), and the default layer for the text is MET1, which is fine. The option you may want to change is the text height. You should position the cursor in the **VEM** graphics window, and type:

`"textHeight 100"`

You should then hold down the Shift key, and click the middle mouse button. A small menu will appear, with the text: `miscellaneous` ⇒. Move the text over the arrow, then down to the `phyt-set-options` command, and click the middle mouse button again. You have set the text height to 100λ.

To enter text, find a suitable place on the layout, which is not near any of the wiring, especially the wiring on MET1. There should be some empty space on your chip somewhere. In a moment, you will create a point to indicate where you want the text. As a guideline, the text will be above and to the right of the point. So, choose a spot with plenty of room above and to the right of the cursor.

Once you have chosen the spot for the text, create a point by clicking the left mouse button. Now type:

`"your name here"`

(You should substitute your name, initials, or whatever). Hold the Shift key, and click the middle mouse button. Select `phyt-create-text` from the miscellanous sub-pane of the menu. After a while, the text will appear in MET1 on your chip.

You can repeat the process to add other text, (your lab partner's name might be nice). If you don't like the location of the text, you can select it and move it as you would any other cell in a layout. When your are done, save the layout.

Since the Octtools are hierachical in nature, the text has been added to the ":final" version of the chip. You can open a **VEM** graphics window with the final version of the chip just to see it.

Phyt

Phyt also exists as a stand-alone program. It will create an Oct cell, using your text (minus spaces and punctuation) as a cell name. You will also need to specify the technology, so phyt can scale the text properly. The command to generate the text "your name here" using phyt would be:

```
% phyt -T scmos "your name here"
```

This would create a new Oct cell and view, `yournamehere:physical`. You should run **vulcan**,

```
vulcan  yournamehere:physical
```

Once this is done, you can instantiate the cell "`yournamehere:physical`" as you would any other cell.

Chapter 2

VEM tutorials

This tutorial consists of four parts:

common: in which the basic functionality of VEM is described. You should be familiar with the content of this section before starting with any of the other sections.

physical editing: in which the commands for physical editing are described. You should read this section if you plan to design layout at the physical level, that is at the level of boxes on various layers.

symbolic editing: in which the commands for symbolic editing are described. The **Octtools 5.2** emphasize symbolic editing, and you should probably become familiar with this section.

schematic editing: in which the commands to edit schematics are explained.

These tutorials were in originally developed by David Harrison, the author of VEM. They were then expanded and updated by Paul Cohen. They have now been consolidated into one tutorial by Andrea Casotto.

2.1 Common part

VEM is an interactive graphics editor for designs represented using the OCT Data manager. VEM currently supports physical and symbolic editing of IC designs, as well as schematic capture.

The tutorials cover many of the commands available in VEM, but the coverage is not complete. After completing the tutorials, you should be able to discover how to use the remaining commands by referring to the manual. (Also known as the "man page".)

There are tutorials covering each of the three editing styles available in VEM (physical, symbolic and schematic) as of **Octtools 5.2**. Table 2.1 lists the categories of tools you will need to complete these tutorials. It is assumed that you are familiar with the use of at least one text editor, and that you are familiar with managing windows on your system.

Tool class	Examples
UNIX	
X Window System	DECwindows, Openwindows X11R4
Window Manager	dxwm, uwm, twm
Text Editor	vi, emacs
Octtools	VEM, musa, wolfe

Table 2.1: Tools required for the tutorials

All of the tutorials assume you are logged in to a graphics workstation running the X Window System (X11R4), UNIX, a window manager, and **Octtools 5.2** or higher. Commands you should type are noted as follows:

`type this command`

Quotation marks (") are very important in VEM. When they appear in a command such as

`"nand:physical":open-window`

you should enter the quotation marks as they appear in the text.

Many VEM commands are *bound* to a key so they can be invoked with a single keystroke. These keys will be noted with a box, as in $\boxed{\text{o}}$, and are not followed by the return key. Like UNIX, *key bindings* are case sensitive. The notation $\boxed{\text{\^{}e}}$ indicates you should press and hold the control (Ctrl) key while pressing the indicated key.

- Paragraphs like this describe an operation which is more complicated than simply typing in a command. Such operations generally require the mouse. You should read the paragraph before proceeding.

⋆ Experience has shown that there are certain operations which novice users often find difficult or confusing. These operations are noted with the symbol ⋆. Read these paragraphs especially carefully before proceeding.

Some VEM operations require that you *click* one of the mouse buttons, others require you to *press* a mouse button. When you are asked to *click* a mouse button, that means you should press and immediately *release* the button without moving the mouse. When you are asked to *press* one of the buttons, you should press the button and *hold* it down while performing the indicated operation (generally involving mouse movement), then *release* it when the operation is complete.

2.1.1 Window Managers

VEM assumes that a window manager is running. Some window managers place a title bar at the top of each window. If your terminal window has a title bar, then there is a window manager running.

Some window managers require you to *click* one of the mouse buttons (usually the left mouse button) to *focus* keyboard input on particular window. Other window managers (**uwm**, for example), *focus* input automatically, based on which window is directly under the cursor. It is assumed that you are familiar with the operation of whatever window manager you are using. The remainder of this tutorial assumes that input is focused on the proper window for each operation.

2.1.2 OCTTOOLS Variable and Search Paths

In order to use the Octtools, you need to set the environment variable `OCTTOOLS` and you need to have the directory `$OCTTOOLS/bin` in your search path. The `OCTTOOLS` variable tells the tools where to find some useful information, such as technological information, standard libraries, and documentation. The value to be given to this variable depends of course on the particular installation. The rule is that `$OCTTOOLS/bin` should be the directory with all the **Octtools 5.2** executables for the machine that is being used: for example if the machine is a VAX, then `$OCTTOOLS/bin` should point at the VAX binaries.

It is convenient to edit the `set path` command in your `.cshrc` file (not your `.login` !!) (in your home directory) to include `$OCTTOOLS/bin`. This will take effect next time you start a csh (or login again). To add the Octtools to your search path immediately, type:

```
# This works for some Suns at UCB. Your path will probably be different.
setenv OCTTOOLS /projects/octtools/octtools/sun4

set path=($OCTTOOLS/bin $path)
```

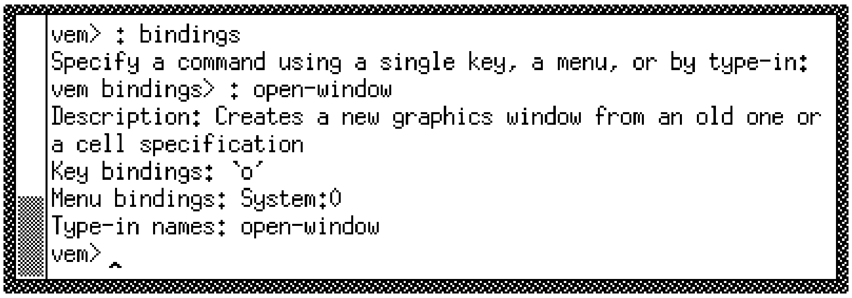

Figure 2.1: VEM bindings command

2.1.3 Key Bindings

As you become more familiar with VEM, you will find key bindings an effective way so speed up the layout process. There is a special command, `bindings`, which allows you to determine some information, including the key binding, for each command.

- *Click* the middle mouse button to invoke the VEM main menu, locate the `bindings` command at the bottom of the **System** sub-pane.
- *Click* the left mouse button to select the `bindings` command.
- In the **VEM** console window, you will see the command `:bindings` echoed, followed by the text:

  `Specify a command using a single key, a menu, or by type-in:`
- You will also notice that the **VEM** console window prompt now reads:

  vem bindings>
- The next command you enter will not actually execute, but will be described by the bindings command. Press the $\boxed{\text{o}}$ key.
- In the **VEM** console window, a one line description of the command (`:open-window`) will appear, along with the key binding, and the sub-pane of the menu the command appears on. Figure 2.1 is a sample of the **VEM** console window after the `bindings` command has been executed. The notation "System:0" indicates it is the top item on the **System** sub-pane of the VEM main menu.
- After the command is described, VEM returns to normal command entry.

2.1.4 Starting VEM

This section explores some very basic operations in VEM, such as opening windows.

- Start VEM with the command:

  `vem -G+0-0 &`

  (If the directory `$OCTTOOLS/bin` is not included in your search path, you should either type:

  `$OCTTOOLS/bin/vem -G+0-0 &`

  or add `$OCTTOOLS/bin` to your search path.)

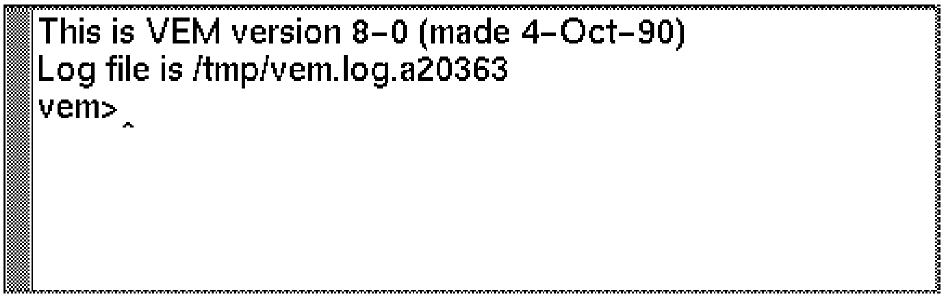

Figure 2.2: Sample VEM console window

After a few seconds, the **VEM** console window will appear in the lower left hand corner of your screen. Figure 2.2 shows a sample VEM console window.

The command line switch `-G+0-0` indicates the X–Y coordinates of the initial position of the **VEM** console window on the screen. If you do not specify the position, you will need to position the **VEM** console window manually as you would any newly created window.

- Move the cursor to the newly created **VEM** console window. (If you need to manually focus input on a window, be sure to do so.) Type:

  `"nand:physical"`

  and then press the $\boxed{\text{o}}$ key.

The letter "o" is the default *key binding* for the VEM command `open-window`. Thus, when you press the $\boxed{\text{o}}$ key, the command
`"nand:physical" :  open-window`
actually appears in the **VEM** console window.

- Since the `nand` cell does not yet exist, it will be created. The message: "Buffer does not exist. New buffer created" will appear in the **VEM** console window.

- VEM will need to know a few basic facts about the cell you are creating. These facts are known as *properties*. Since the `nand` cell does not yet exist, the essential properties are considered to be missing. VEM will prompt you for the properties by opening a small window called a *dialog box*. (See figure 2.3 for a sample dialog box.) VEM will try to choose some reasonable default values for the properties. The **Technology** property should be **scmos**, while the **EDITSTYLE** and **VIEWTYPE** properties should both be **PHYSICAL**[1]. Move the cursor over the box labeled "OK" and *click* the left mouse button.

- The behavior of VEM at this point is slightly dependent on the type of window manager you are using. With some window managers, a **VEM** graphics window will be created at some location on the screen. (Be sure to change the focus!) Other window managers will create the outline of the **VEM** graphics window at the current cursor location. If this happens, move the cursor such that the outline is roughly centered on the screen and *click* the left mouse button to actually create the **VEM** graphics window.

Since VEM runs on top of the X Window System, the **VEM** graphics window and the **VEM** console window can be moved and resized in the same manner as any other X application.

Figure 2.4 shows a newly created **VEM** graphics window for the cell `nand`. The window is divided into two areas, the *graphics display* area and the *title bar*, located at the top of the window.

In the center of the title bar is the name of the cell. In this case, it is `nand:physical`. At the right hand edge of the title bar, VEM displays the current location of the cursor when it is in the graphics window, in λ (lambda).

[1] If the properties are not correct, it is probably due to a typographical error when you entered the argument to the open-window command. Click the left mouse button over the Cancel box and re-enter the command line.

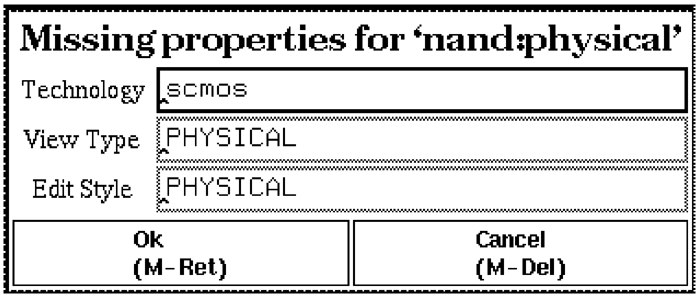

Figure 2.3: Sample missing property dialog box

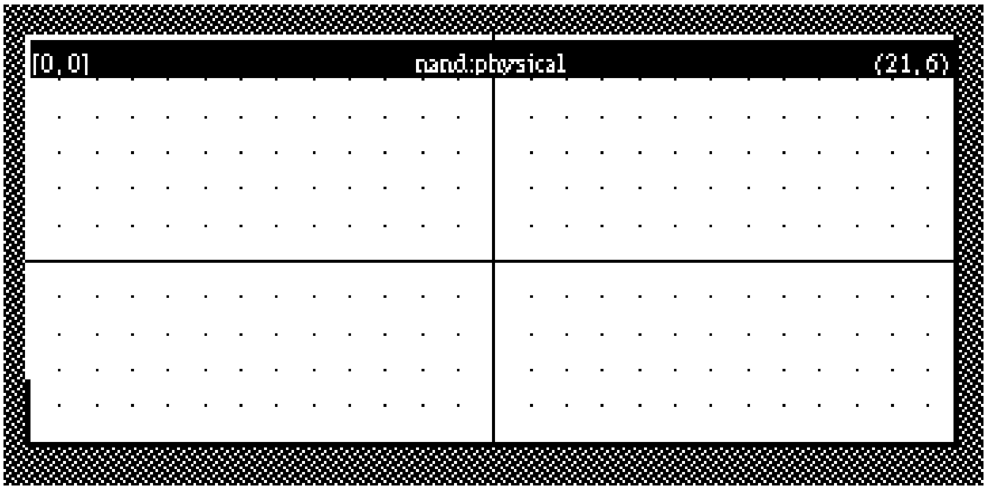

Figure 2.4: VEM graphics window

- Move the cursor around the **VEM** graphics window and watch the numbers in the title bar change.

The numbers on the left indicate the size of a graphics argument. This will be covered shortly. For now, these numbers should be [0,0].

2.1.5 Entering Arguments in VEM

VEM commands are specified in *post-fix* notation. That is, you first specify the arguments to a command, then the command itself. For example, when you opened the graphics window, you first specified the cell `nand:physical`, then the command `open-window`. VEM accepts 5 types of arguments. These are **points**, **lines**, **boxes**, **text**, and **objects**.

Graphics arguments (points, lines, and boxes) are entered into VEM via the mouse. The left mouse button is used for entry of graphics information, the right button is used to modify graphics arguments. The middle button invokes the VEM commands menu. As arguments and commands are entered, user feedback is provided via the **VEM** console window. Thus, it is advisable to avoid covering the **VEM** console window with other windows.

- Move the cursor to the **VEM** graphics window, and *click* the left mouse button. This will create a point in the **VEM** graphics window, and the message:

  vem> points(1)

  echos to the **VEM** console window.

- Move the cursor around the **VEM** graphics window, and enter a few more points. (Be sure to stop the cursor before entering each point.) Note that each time a point is entered in the **VEM** graphics window, the number of points in the **VEM** console window is incremented.

- Once you have entered a few points, hit the $\boxed{\text{delete}}$ key and observe that the points you entered are deleted in reverse order. The number of points in the **VEM** console window is decremented each time you delete a point.

- While you still have a few points remaining (enter a few new points if necessary), and press $\boxed{\hat{}\text{u}}$. This will delete all of the points from the **VEM** graphics window, and remove the argument count from the **VEM** console window.

Next we will enter some boxes. The numbers on the left hand side of the title bar are used to indicate the size of a graphics argument. As we enter box arguments, these numbers will change.

⋆ To enter a box, *press* the left mouse button, and *hold* it down while you move the mouse. As you move the mouse, a box will appear and will change size as you move the mouse. *Release* the left mouse button to complete the box.

- The argument:

  vem> boxes(1)

  will appear in the **VEM** console window.

- Note that the argument size numbers at the left hand edge of the title bar indicate the size of the box you just drew in λ.

- Enter a few more boxes. As you draw each box, note that the argument size numbers in the title bar change as the boxes are created. Figure 2.5 shows a sample **VEM** graphics window with a few boxes entered. The argument size indicates the last box drawn had a size of $6\lambda \times 5\lambda$.

- As with points, boxes can be removed in reverse order with the $\boxed{\text{delete}}$ key. Try deleting a few boxes.

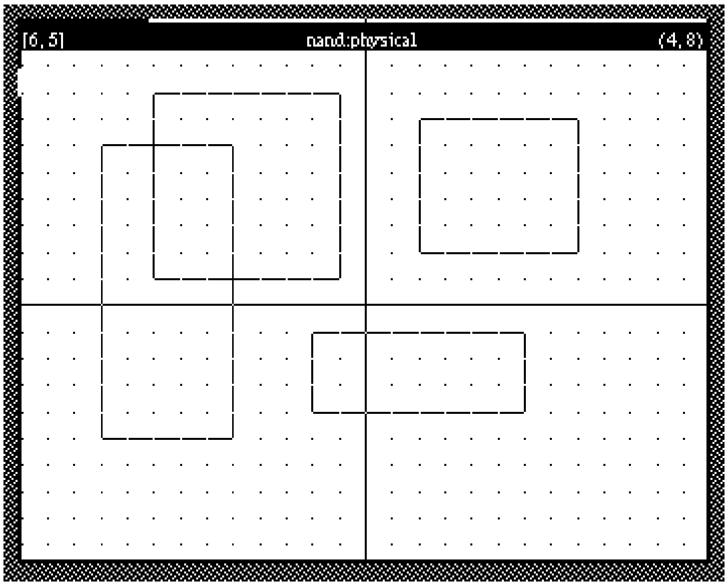

Figure 2.5: VEM graphics window with some boxes

- All of the arguments can be deleted by pressing ^u. Try this now.

So far we have entered graphics arguments by *clicking* the left mouse button, and by *pressing* the mouse button and *moving* the mouse before we *released* the button. This has allowed us to enter **points** and **boxes**. **Lines** arguments are entered into VEM via a combination of mouse *click* and *press* operations.

⋆ To draw a line, move the mouse to the place you would like one end of the line to be and *click* the left mouse button. *Do not move the mouse after the point has been entered.*

⋆ With the cursor over the point you just entered, *press* the left mouse button again. With the button held down, move the mouse around a bit. Note how the other end of the line tracks the cursor.

- When you want to complete the line, *release* the left mouse button.

- When you first create the point, the argument:

  vem> points(1)

  appears in the **VEM** console window. When you complete the line, the argument changes to:

  vem> lines(1)

- Try entering a few more lines and watch the response in the **VEM** console window and the argument size entry in the title bar.

- To delete a line, you will need to hit the delete key twice. The first time you hit the delete key, the last line argument in the **VEM** console window will revert to a point argument. The second time you hit the delete key, the point will be deleted. Try hitting the delete key a few times and see what happens.

- As with points and boxes, the entire argument list can be removed with ^u. If you have any lines remaining, delete them with ^u now.

2.1.6 Changing Arguments

Sometimes you may enter arguments which are not the right size, or are not in the right place. It is not necessary to delete an argument in order to change its position or size.

- Enter 3 or 4 points in the **VEM** graphics window. (Remember, just *click* the left mouse button to create a point.)

⋆ Move the mouse over one of the points, and *press* and *hold* the right mouse button. While *holding* the button down, move the cursor to some other location in the **VEM** graphics window and *release* the right mouse button. Note that the point has been moved to a new location.

- Delete all of the points by typing ^u.

- Enter a box somewhere in the **VEM** graphics window. (*Press* the left mouse button, move the mouse, and *release* the button.)

- Move the cursor over any of the corners of the box. *Press* and *hold* the right mouse button and move the mouse. Notice how the corner of the box moves with the cursor. *Release* the right mouse button to leave the vertex in a new location.

- Try changing the box again, noting that the argument size indicator now indicates the change in the size of the argument.

- Delete the box with with either the delete or ^u keys.

- Now enter a line somewhere in the **VEM** graphics window. (*Click* the left mouse button, then *press* the left mouse button again and move the mouse before *releasing* the button.) Position the cursor over one end of the line, and try changing the endpoint by *pressing* the right mouse button, moving the mouse, and *releasing* the button.

- Try changing the position of the other end of the line. Once a line is entered, either endpoint can be re-positioned.

- Delete the line by hitting ^u.

We have seen how to enter 4 types of arguments in VEM. **Points**, **lines**, and **boxes** were entered via the mouse. When we first created the **VEM** graphics window `nand:physical`, we entered a **text** argument. We will see how to enter and manipulate **object** arguments shortly.

2.1.7 OCT Notation

VEM uses the OCT Data Manager to store and retrieve all graphics information. Some understanding of the OCT notation is useful to use VEM effectively.

In OCT, the basic design unit is the *cell*. Each cell is an arbitrarily complex portion of the design, and may include other cells in a hierarchical fashion. Figure 2.6 is an example of the cell hierarchy in OCT.

At the top of the figure is a `4 bit adder cell`. This adder includes 4 copies of a `1 bit adder cell`. Each `1 bit adder cell` might include `AND cells`, `OR cells`, and `XOR cells`. Each of these cells might include `transistor cells` or they may be "leaf" cells. (The transistor or leaf cells are not shown in figure 2.6.)

A cell may have many *views*, depending on what design style you use and where you are in the design cycle. We have just created the `physical` view of the `nand` cell (though we haven't added any geometries to that cell yet). The OCT notation for the *physical view* of the *nand cell* is `nand:physical`. We could have other views of the same cell. For example, we might also want to have a *schematic* view of the `nand` cell, where we would create the transistor level schematic. We might call this `nand:schematic`. OCT provides a mechanism for representing CAD data, but places no meaning on the data. *Policy* is used to assign meaning to the data. Currently, policies are

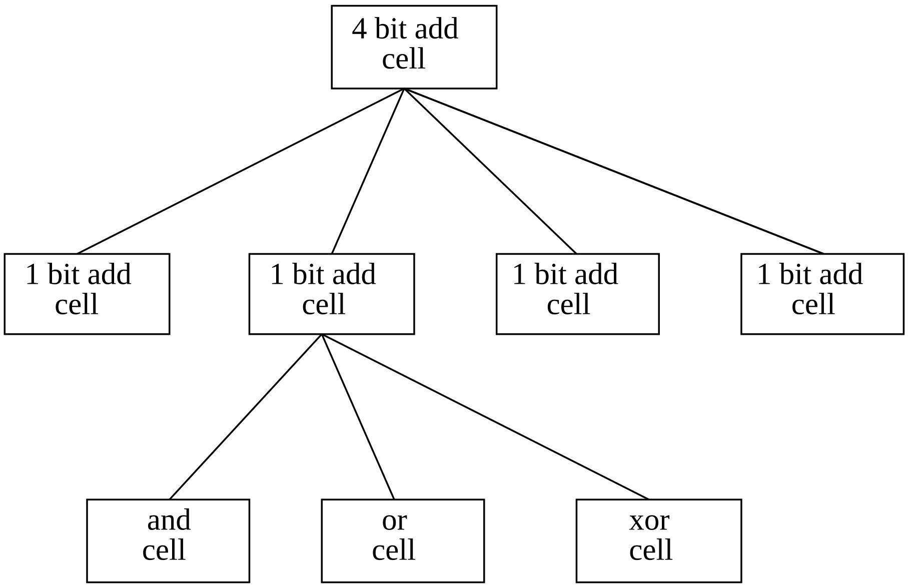

Figure 2.6: OCT cell hierarchy

defined for 3 *view types*. These are *physical*, *symbolic*, and *schematic*. The VIEWTYPE property must correspond to one of the defined types, but the name of the view is arbitrary. For example, we may have 2 different views of the **`nand`** cell, **`nand:test1`** and **`nand:test2`**, which both have the property VIEWTYPE=PHYSICAL.

Finally, each view contains some number of *facets*. Each view has a facet named "**`contents`**", which contains the actual definition of the view. The facet is the fundamental unit that you edit in OCT. An "**`interface`**" facet has been defined, which contains an abstraction of the contents facet. The interface facet is usually generated from the contents facet by a tool such as **vulcan**.

When we opened the **VEM** graphics window we specified the OCT cell:view of **`nand:physical`**. Since we did not specify the facet, the **`contents`** facet was assumed. When we actually add geometries to the cell, we will be adding them to **`nand:physical:contents`**.

2.1.8 OCT and UNIX

Each OCT cell corresponds to a UNIX directory, and each view is a subdirectory of the associated cell. Facets are stored as files in the corresponding view subdirectory. Thus, the OCT structure **`nand:physical:contents`** corresponds to the UNIX file system pathname **`nand/physical/contents;`**. (The semicolon is part of the file name.)

There are no restrictions on where in the UNIX file structure each of the cell directories is located. In the 4 bit adder structure of figure 2.6, it is reasonable to have the 4 bit adder cell and 1 bit adder cell in a directory named **`project`**, while the leaf cells (gates) are actually in a common cell library.

2.2 Physical Editing with VEM

Paul B. Cohen
Massachusetts Microelectronics Center

You should be familiar with the content of the Section 2.1. Start VEM on a cell called `nand:physical`, with TECHNOLOGY=scmos, VIEWTYPE=PHYSICAL, EDITSTYLE=PHYSICAL

2.2.1 Drawing Geometry

So far, we have learned how to enter arguments, but we have not actually entered any geometries into VEM. In general, geometry is created with the `create-geometry` command, using **points**, **lines**, **boxes**, and **text** to define the location and type of the geometry.

In VEM **points**, **lines**, and **boxes** are used to indicate where a geometry is to be created. You also will need to specify which *layer* you want the geometry to be created on. While a text argument can be used for this, the simplest way is via "*layer inheritance*". Layer inheritance means that when you invoke the `create-geometry` command, VEM will check under the cursor and use that layer information to create a new geometry.

2.2.2 The Palette and the Menu

When you opened the **VEM** graphics window on the cell `nand:physical`, VEM confirmed certain default property values for the cell. (See figure 2.3 on page 53.) The first property was the "TECHNOLOGY" property, which has a default value of **scmos**. In order to determine what layers are available in a given technology, and to create a small sample of each layer for the `create-geometry` command to use for layer inheritance, we will open a *palette window* for the scmos technology.

- Make sure the cursor is in the **VEM** graphics window, and *click* the middle mouse button.
- The main VEM menu will appear with 7 major categories, each with an arrow indicating commands specific to that category are on a sub-pane of the main menu. (See figure 2.20.)

(If you accidentally invoke the VEM menu and don't want to actually execute any of the commands, simply move the cursor to some blank spot on the screen and *click* any of the mouse buttons.)

- Move the cursor over the arrow (⇒) to the right of the **System** entry on the main menu. The System sub-menu will appear, as shown in figure 2.7.
- Move the cursor to the `palette` command, and *click* the middle mouse button.
- A palette window will appear, with the title `layers:physical`. (Remember, you may need to position the outline and *click* the left mouse button to create the window.) Figure 2.8 shows a sample physical editing palette for the scmos technology.
- If the palette window is covering the **VEM** graphics window, you should move one or both of the windows so both the palette window and the **VEM** graphics window are completely visible.

(The layers available in the palette do not correspond exactly to CMOS mask layers. Later in this tutorial we will use most of the layers in the palette, so we will see where each of the layers should be used. Conversion of OCT layer names to correct mask layer names for fabrication is done by **fabprep** or **octtocif**.)

	browser
o	**open-window**
^d	**close-window**
?	**where**
P	**palette**
L	**list-palettes**
R	**re-read**
S	**save-window**
^S	**switch–facet**
b	**bindings**

Figure 2.7: System sub-pane of VEM menu

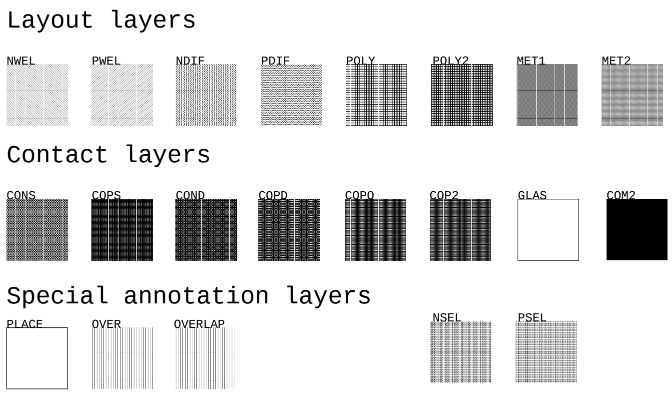

Figure 2.8: Physical editing palette for scmos

2.2.3 Creating Geometry

In this and subsequent sections, you will be creating the layout for a 2 input NAND gate. The purpose of this tutorial is to learn to use VEM, not to create the best NAND gate layout possible. There will be times when the tutorial calls for the creation of a geometry which you will notice, if you are familiar with the "MOSIS Scalable and Generic CMOS Design Rules", will violate some of these rules. Please enter the geometry as specified, since the purpose will be to learn how to use **octdrc** to locate these violations.

- Move the cursor to the center of the **VEM** graphics window. The coordinates at the left hand edge of the title bar should read (0,0).
- Using the box creation method discussed earlier (*press* the left mouse button, *move* the mouse, and *release* the button) draw a box exactly $+16\lambda$ in the X direction and $+4\lambda$ in the Y direction. Remember you can use the argument size entry in the title bar to determine the box size, or you can count grid points.

⋆ Move the cursor to the `layers:physical` palette, and position it over the green square labeled "NDIF".

⋆ *Click* the middle mouse button to invoke the VEM main menu.

⋆ Move the cursor to the arrow (⇒) next to the **Edit** entry on the main menu. The edit sub-pane will appear.

⋆ The first item on the edit sub-pane is `create-geometry`. Move the cursor over this entry and *click* the middle mouse button to actually create a rectangle of NDIF in the **VEM** graphics window.

In OCT terminology, what you have just done is to *attach* an object (the box) to a mask layer (NDIF). Thus, NDIF now "contains" the box object.

- Move the cursor to coordinates (0,14) in the **VEM** graphics window. (The title bar indicates the file is `nand:physical`.)
- Draw a box exactly $16\lambda \times 4\lambda$.
- Move the cursor to the palette window, and position it over the box labeled "PDIF".
- *Click* the middle mouse button to invoke the VEM menu, and position the cursor over the arrow (⇒) next to the **Edit** entry.
- When the edit sub-pane appears, move the cursor over the `create-geometry` entry, *but do not press the middle mouse button yet.*
- Notice the letter "c" next to the `create-geometry` entry. This is the default key binding for the `create-geometry` command.
- With the cursor over the `create-geometry` command, go ahead and *click* the middle mouse button to execute the `create-geometry` command.

The $\boxed{\text{c}}$ key is *bound* to the `create-geometry` command. In general, there are 3 ways to enter VEM commands, though not all commands can be entered in all 3 ways. The `create-geometry` command, for example, can be entered via the menu, as we have done, by pressing the $\boxed{\text{c}}$ key, or by typing the command, preceded by a colon (:). We will take a closer look at key bindings shortly.

- Move the cursor to coordinates (5,-3) in the **VEM** graphics window titled `nand:physical`.
- Draw a box exactly $2\lambda \times 23\lambda$, (The argument size entry in the title bar should read [2,23].) The upper right hand corner of the box should be at coordinates (7,20).

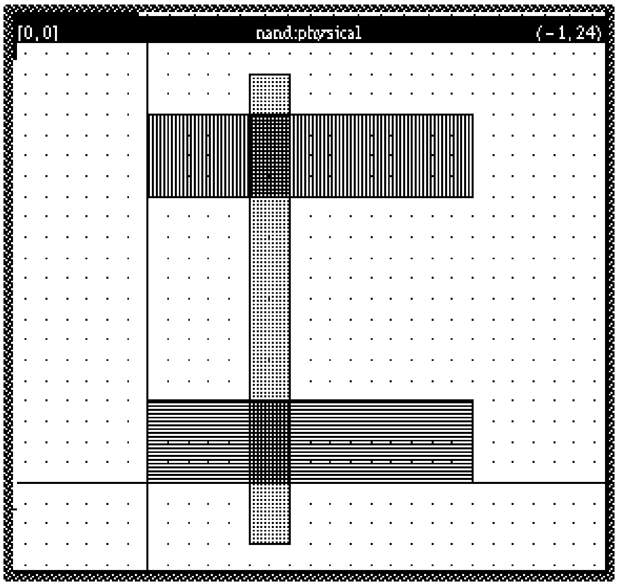

Figure 2.9: Unfinished NAND gate

- Move the cursor to the box labeled "POLY" in the palette window, and press the $\boxed{\text{c}}$ key. This will create a rectangle of POLY in the **VEM** graphics window.

At this point, your layout should resemble figure 2.9.

2.2.4 Saving Your Work

It is generally a good idea to save you work periodically. There are 2 commands in VEM for saving you work. These are `save-window` and `save-all`. Both commands are bound to the same key, $\boxed{\text{S}}$. This is an example of the context sensitive nature of VEM.

- Move the cursor to the **VEM** console window, and press the $\boxed{\text{S}}$ key. (Remember, VEM is case sensitive.)
- You will see the command `:save-all` echoed in the **VEM** console window. VEM will open a **dialog box** asking you which buffers you would like to save. Figure 2.10 shows what this dialog box will look like. In general, VEM will use this dialog box only if you have changed the layout since the last `:save-all` or `:save-window` command.
- Move the cursor over the box next to `nand:physical:contents;`, and *click* the left mouse button. A check ($\surd$) will appear in the box.
- Move the cursor over the box labeled "OK", and *click* the left mouse button again to confirm the `save-all` operation.
- Move the cursor to the **VEM** graphics window, and press the $\boxed{\text{S}}$ key again.
- Notice the command `save-window` is echoed in the **VEM** console window.

2.2.5 Creating More Geometry

If you are familiar with the layout for a NAND gate, you may notice that the PDIF box will not be large enough to complete the layout without violating the layout rules. We will deal with this problem shortly.

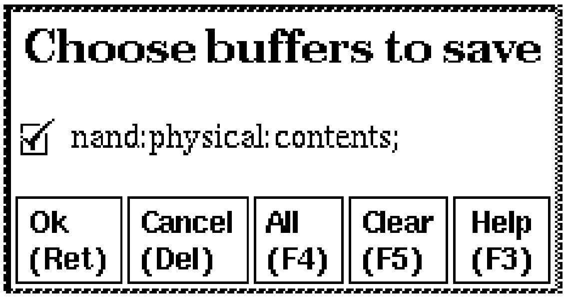

Figure 2.10: VEM dialog box for the save-all command

- Move the cursor to coordinates (1,1). (This is 1λ "inside" the NDIF box.)
- Draw a $2\lambda \times 2\lambda$ box. The upper right corner of the box should be at (3,3).
- Move the cursor to coordinates (15,3). (1λ "inside" the opposite corner of the NDIF box.)
- Draw a $-2\lambda \times -2\lambda$ box. The lower left will be at coordinates (13,1).
- Move the cursor to the palette window, and position it over the box labeled "COND". This stands for **CO**ntact to **ND**iffusion.
- Press the $\boxed{\text{c}}$ key to create the geometry for the contacts. (You can use the `create-geometry` command from the VEM menu if you prefer.)
- Note that both contacts have been created.
- Move the cursor to coordinates (1,15) and draw a $2\lambda \times 2\lambda$ box.
- Draw another $2\lambda \times 2\lambda$ box beginning at coordinates (9,15).
- Move the cursor to the palette window, and position it over the box labeled "COPD" (**CO**ntact to **PD**iffusion.)
- Press the $\boxed{\text{c}}$ key to create the geometry for the 2 contacts.

We will need to add a third contact to the PDIF (COPD), and extend the PDIF itself, but before we do that we will add the POLY to make the next two transistors. This will allow us to quickly determine how far the PDIF needs to be extended.

- In the **VEM** graphics window (`nand:physical`), move the cursor to coordinates (9,-3).
- Draw a box exactly $2\lambda \times 12\lambda$ ([2,12]).
- Move the cursor to a point over the POLY box you created earlier. Be sure, for now, that there is only POLY under the cursor. (Coordinates (6,8) will work.)
- Press the $\boxed{\text{c}}$ key to create a POLY box.

Notice that layer inheritance works with the cursor over any sample of the layer you wish to create a geometry on. The palette is simply a convenient way to get a sample of each of the layers for the technology.

- Move the cursor to coordinates (9,9) and draw a $4\lambda \times 2\lambda$ box.
- Move the cursor to coordinates (13,9) and draw a $2\lambda \times 10\lambda$ box.
- Move the cursor over the POLY sample in the palette window and press the $\boxed{\text{c}}$ key to create 2 additional boxes on POLY. (Those familiar with the layout rules will notice a violation. This will be corrected later.)

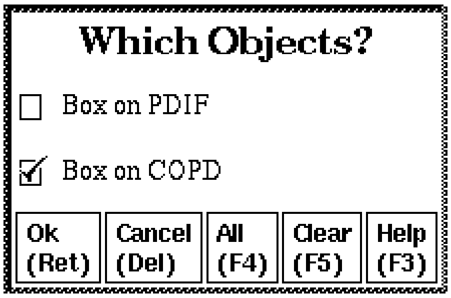

Figure 2.11: VEM selection dialog box

2.2.6 Selecting, Copying, and Altering Geometry

Often, while creating a layout, it is useful to be able to copy and change various geometries.

- Position the cursor at coordinates (10,16) in the **VEM** console window. This should put the cursor over the right hand box on COPD.

Rather than create a new contact on COPD, we will simply make a copy of an existing contact. Before we can do this, we will need to select the contact with the `select-objects` command.

- *Click* the middle mouse button to invoke the VEM main menu, and position the cursor over the arrow (⇒) next to the **Selection** entry. This will cause the selection sub-pane to appear.
- The first entry on the selection sub-pane is `select-objects`. Move the cursor over that entry and *click* any mouse button. (Note the small "s", indicating the `select-ojects` command is bound to the $\boxed{\text{s}}$ key.)
- When the VEM main menu was invoked, there were 2 geometries under the cursor. One was the COPD box we wanted to select, the other was the PDIF box. We will need to let VEM know which one (or both) of these geometries we want to select. VEM will create a dialog box (see figure 2.11) to resolve the ambiguity.
- Move the cursor over the box next to the entry "Box on COPD" and *click* the left mouse button.
- The box on COPD will be highlighted to indicate which item you have "checked off" in the dialog box. Move the cursor over the box labeled "Ok" and *click* the left mouse button again to confirm the selection.

You have now selected an **object**, which is the fifth type of argument in VEM. In the **VEM** console window, you will see that you have selected one object.
vem> objects(1)

- Move the cursor to coordinates (9,15). This is the lower left hand corner of the contact.
- *Click* the left mouse button to create a point.
- Move the cursor to coordinates (17,15), and *click* the left mouse button to create a second point. The command line in the **VEM** console window will read:

  vem> objects(1) points(2)

  Thus, you are using the mouse in the **VEM** graphics window to build an argument list.
- *Click* the middle mouse button to invoke the VEM main menu, and move the cursor over the arrow next to the **Selection** entry.

- The fourth entry on the selection sub-pane is `copy-objects`. (Note the small "x", indicating the key binding for the `copy-objects` command.) Move the cursor over the `copy-objects` entry, and *click* the left mouse button to execute the command.
- The new copy of the contact is selected. Position the cursor over the new copy of the contact, and *click* the middle mouse button. When the VEM main menu appears, move the cursor to the arrow next to the **Selection** entry.
- Move the cursor over the second entry, `unselect-objects`, and *click* the left mouse button to unselect the object on COPD. (Note the key binding is "u".)

The `copy-objects` command requires a list of objects to be copied, as well as 2 points indicating the displacement of the copy. VEM will copy the selected objects, and displace the copy by the amount indicated by the X–Y difference between the points. In this case, the points were 8λ apart in the X direction, and 0λ apart in the Y direction.

- The new contact may be fairly close to the edge of the window. This may make the next operation a bit difficult. We can use the `show-all` command to fit what we have drawn so far into the existing **VEM** graphics window. Press the $\boxed{\text{f}}$ key, which is the key binding for the `show-all` command. (`Show-all` is on the **Display** sub-pane of the VEM menu.)
- Position the cursor at coordinates (0,14) in the **VEM** graphics window.
- Draw a box exactly $20\lambda \times 4\lambda$. The upper right hand corner should be 1λ above and to the right of the contact on COPD we just copied.

⋆⋆ Move the cursor over the PDIF rectangle in the **VEM** graphics window. *Be sure the cursor is in the* **VEM** *graphics window, not the palette.)* Coordinates (12,16) will do nicely.

⋆⋆ Press the $\boxed{\text{a}}$ key, which is the key binding for the `alter-geometry` command. (`Alter-geometry` is on the **Edit** sub-pane of the VEM menu.) The PDIF area in the **VEM** graphics window will change from $16\lambda\times 4\lambda$ to $20\lambda\times 4\lambda$.

2.2.7 Pan, Zoom, and Multiple Windows

We will need to add some metal interconnect to our layout. The first place we will add metal is near the top of the NAND gate, to create a V_{DD} connection. There is not enough room near the top of the **VEM** graphics window to allow for the minimum metal width and the spacing we will need to other metal we will create later on. We can move the window with the `pan` command.

- Position the cursor over the middle COPD contact, near the top of the **VEM** graphics window. (Coordinates (10,16), for example.)
- Press the $\boxed{\text{p}}$ key, which is the key binding for the `pan` command. (The `pan` command is on the **Display** sub-pane of the VEM menu.)
- Note that the contact is now positioned at the center of the **VEM** graphics window, and there is plenty of space "above" the layout for more geometry.
- Move the cursor to coordinates (0,14) in the **VEM** graphics window, and draw a $4\lambda \times 7\lambda$ box.
- Move the cursor to coordinates (16,14) in the **VEM** graphics window, and draw another $4\lambda \times 7\lambda$ box.
- Move the cursor to coordinates (0,21) in the **VEM** graphics window, and draw a $20\lambda \times 3\lambda$ box.
- Though we could create the geometry on first metal via the palette, as we have done before, this time we will enter the rest of the command "longhand". The command line in the **VEM** console window should read:

```
vem> boxes(3)
```

⋆ *Remembering that quotes should be entered as shown*, and keeping the cursor in the **VEM** graphics window, type:

```
"MET1":create-geometry
```

The command line will now read:

```
vem> boxes(3) "MET1":create-geometry
```

(Remember to hit the $\boxed{\text{Return}}$ key after you enter the command.) Note that even though the cursor was in the **VEM** graphics window, the characters you typed appeared in the **VEM** console window. (MET1 is an abbreviation for **MET**al **1**.)

- Press the $\boxed{\text{f}}$ key (`show-all`) to bring the entire layout into the **VEM** graphics window.

VEM is a multiple window application, using the X Window System. This means we can have multiple **VEM** graphics windows open on a design, or we can have different designs in different **VEM** graphics windows on the screen at the same time. VEM allows us to work between windows on the same design.

- Move the cursor to coordinates (6,13) in the **VEM** graphics window, and draw an $8\lambda \times 7\lambda$ box.
- *Click* the middle mouse button, and move the cursor to the first entry on the **System** sub-pane, which is the `open-window` command. *Click* the left mouse button to execute the command.

⋆ If you are using a window manager which creates windows immediately (**dxwm**, for example), then window will be opened with the contact on COPD near the middle. If so, then this is fine for now, even if it partially obscures the original **VEM** graphics window.

⋆ If you are using a window manager which allows you to position and size the window before actually creating it (such as uwm), then the default size for the secondary **VEM** graphics window may be too large. We can position and size the new window in one operation. Move the mouse to a point near the upper left hand corner of the screen. *Press* the middle mouse button, and move it a slightly towards the lower right hand corner of the screen. You will notice a small box in the upper right hand corner of the screen, with the text:

```
VEM cell nand:physical : (x dimension) (y dimension)
```

Slowly move the mouse until the X and Y dimensions are both about 400 (the units are pixels), and *release* the middle mouse button.

⋆ You will now have two graphics windows opened on the cell `nand:physical`. Move the cursor to the newly created **VEM** graphics window, and press $\boxed{\text{Z}}$ (`zoom-out`). (Remember, you may have to *focus* the commands on the new window for the `zoom-out` command to work.)

- If the newly created **VEM** graphics window is obscuring the NDIF box in the original **VEM** graphics window, move one of the windows to uncover the NDIF. (If only the top (PDIF) is obscured, that is fine.)

We can open yet another **VEM** graphics window on this design.

- In the original **VEM** graphics window, move the cursor to coordinates (3,-2), and draw a $15\lambda \times 8\lambda$ box.

⋆ Press the $\boxed{\text{o}}$ key to open another **VEM** graphics window, this time with the NDIF and one of the COND contacts filling most of the window.

⋆ It is likely that this third window is covering part of the first and second **VEM** graphics windows. Covering the first window is OK, but if the second **VEM** graphics window is covered, move one of the windows so you can see all of the second and third **VEM** graphics windows.

One of the features of VEM is that it allows us to work between windows. This allows us to have windows "zoomed in" on various parts of a design so we can position geometry accurately.

⋆ Move the cursor to coordinates (8,18) in the second **VEM** graphics window (the one with the COPD box in the center). *Click* the left mouse button to create a point.

⋆ Move the cursor to coordinates (8,0) in the third **VEM** graphics window (NDIF and COND). *Click* the left mouse button to create another point.

⋆ Keeping the cursor in the third **VEM** graphics window, create points at coordinates (16,0), (16,4), and (12,4), *in that order.*

⋆ Moving back to the second **VEM** graphics window, position the cursor at coordinates (12,18) and *click* the left mouse button to create a point.

- Move the cursor to the palette window, and position it over the box labeled MET1.
- Press the $\boxed{\text{c}}$ key to execute the **`create-geometry`** command. Notice there is an "L" shaped polygon on MET1, with a vertex at each point you created.

After we create numerous **VEM** graphics windows, we may want to close a few windows to keep the screen from becoming too cluttered and confusing. The **`close-window`** command allows us to close some or all of the VEM windows.

- Move the cursor to the second **VEM** graphics window, (make sure input is properly focused), and *click* the middle mouse button to invoke the VEM menu.
- On the **System** sub-pane, locate the **`close-window`** command, and *click* the left mouse button to execute it.
- Leave the third **VEM** graphics window on the screen, but move it so it is not covering the first **VEM** graphics window or the palette.

`Create-geometry` is not the only command that can be used between windows.

- Move the cursor to coordinates (0,2) in the original **VEM** graphics window, and create a point by *clicking* the left mouse button.
- Move the cursor to the other **VEM** graphics window (it was the third, but there are only two **VEM** graphics windows now), and press the $\boxed{\text{p}}$ key (**`pan`**).
- Notice that the position "pointed" to in the original window is now centered in the other window.
- Keeping the cursor in the third **VEM** graphics window, press $\boxed{\text{Z}}$ to execute the **`zoom-out`** command.
- Draw a $4\lambda \times 4\lambda$ box beginning at coordinates (-8,0).
- Move the cursor to the PDIF box in the palette (**`layers:physical`**), and press $\boxed{\text{c}}$ to create the geometry. Note the new geometry appears in both windows.
- Move the cursor back to the original **VEM** graphics window, and press $\boxed{\text{f}}$.

2.2.8 More Selection, Copy, Move, and Transform

Earlier, we selected an object by creating a point and executing the `select-objects` command. We can also indicate the items we would like to select by using a **box** or a **line** argument to the select command.

- In the original **VEM** graphics window, draw a $6\lambda \times 6\lambda$ box beginning at coordinates (-9,-1). (This will surround the PDIF you just created.)
- Press the $\boxed{\text{s}}$ key to select all objects completely contained within the box. In this case, it is only the PDIF.
- ⋆ Move the cursor to coordinates (-4,0), and *press* the right mouse button. While *holding* the right mouse button down, move the cursor up to coordinates (-4,14), then *release* the button.
- Press the $\boxed{\text{x}}$ key to copy the PDIF box.

The `move-objects` command in VEM operates much like the `copy-objects` command, except the original objects are displaced. We will use the `move` command later, but in case you put something in the wrong place, you might want to try the `move-objects` command.

You might have deduced that we are creating *well* and *substrate* contacts, though we're not done yet. In order to make ohmic contact to the N-well, we will need NDIF, not PDIF.

- (The PDIF box you just copied should still be selected. If it isn't, select it now.) Move the cursor over the box labeled NDIF in the palette window, and *click* the middle mouse button.
- ⋆ Locate the `change-layer` command on the **Edit** sub-pane of the VEM menu, and *click* the left mouse button to execute it. Note that the selected PDIF box is now changed to NDIF.
- Move the cursor to (-7,15) and create a $2\lambda \times 2\lambda$ box. Move to the box labeled COPS in the palette window (yes, this is an error) and press the $\boxed{\text{c}}$ key to create the geometry. (COPS is **CO**ntact to **P** **S**ubstrate.)
- Move the cursor to (-7,1) and create a $2\lambda \times 2\lambda$ box. Move to the box labeled CONS in the palette window (yes, this is another error) and press the $\boxed{\text{c}}$ key to create the geometry. (CONS is **CO**ntact to **N** **S**ubstrate.) Notice the contact appears in both graphics windows.
- We'll need to correct the two errors we just made. Move the cursor to (-7,1) and draw a $2\lambda \times 16\lambda$ box. Press the $\boxed{\text{s}}$ to select the two contacts. Note that even though there was NDIF and PDIF enclosed in the box, only the contacts (COPS and CONS) were <u>completely</u> enclosed by the box.
- With the cursor still in the **VEM** graphics window, type:

  ```
  "my"
  ```

  and then press the $\boxed{\text{t}}$ key. *Nothing will appear to have happened. This is OK.* Key $\boxed{\text{t}}$ is the binding for `transform`, with "my" indicating the transform is to mirror the object in the y-direction.
- In order to confirm the transformation, press the $\boxed{\text{m}}$ key (equivalent to the `move` command) to actually move both of the contacts.

The `transform` command does not actually move the objects, but rather moves the selection boxes to the new location and waits for a subsequent command to complete $\boxed{\text{m}}$ or abort $\boxed{\hat{}\,\text{u}}$ the actual move. Since in this case the transformation was completely symmetrical, the transformation of the selection boxes was not entirely obvious.

- Unselect all of the objects by pressing $\boxed{\hat{}\,\text{u}}$.

- It has been a while since the layout has been saved. From the original **VEM** graphics window, press $\boxed{\text{S}}$ to save your work.

- We really don't need the other **VEM** graphics window any more, so move the cursor to the other **VEM** graphics window and press $\boxed{\text{\^{}d}}$ to close it.

A few more connections and the NAND gate is complete.

- Draw a $12\lambda \times 4\lambda$ box beginning at (-8,0), and another box, of size $8\lambda \times 4\lambda$ beginning at (-8,14).

- Move the cursor over a box of MET1 (try (2,20)), and press $\boxed{\text{c}}$. (If there was some ambiguity as to which layer you wanted, VEM will ask you to resolve the ambiguity with a dialog box.)

- Move the cursor to (8,-2) and press $\boxed{\text{p}}$ to `pan` that coordinate to the center of the **VEM** graphics window.

- Create a $4\lambda \times 4\lambda$ box beginning at coordinates (9,-7), move the cursor over a bit to the POLY box you created (much) earlier, and press the $\boxed{\text{c}}$ key.

- Create another $4\lambda \times 4\lambda$ box beginning at coordinates (9,-7), move the cursor over a bit to the MET1 polygon you created (much) earlier, and press the $\boxed{\text{c}}$ key. (If a dialog box appears, select MET1.)

- Create a $2\lambda \times 2\lambda$ box beginning at coordinates (10,-6), move the cursor to the box labeled COPO (**CO**ntact to **PO**ly) in the palette window, and press $\boxed{\text{c}}$.

- Create yet another $4\lambda \times 4\lambda$ box beginning at coordinates (9,-7), and press the $\boxed{\text{s}}$ to select everything inside the box. There should be 3 objects, the POLY, MET1 and COPO boxes.

- Using the left mouse button, create points at (13,-3) and (7,-3), in that order. press the $\boxed{\text{x}}$ to execute the `copy-objects` command.

- Oops! We've violated the metal1 spacing rule of 3λ. Notice that the new copy of the POLY, MET1 and COPO boxes are still selected. Place the cursor anywhere on the **VEM** graphics window, and *press* the right mouse button. While *holding* the right mouse button down, move the cursor 1λ to the left and 0λ down ([-1,0]), then *release* the button.

- Press $\boxed{\text{m}}$ to execute the `move-objects` command.

- Move the cursor to the middle of the selected objects (coordinates (4,-5)), and press the $\boxed{\text{u}}$ key to execute the `unselect-objects` command.

- A dialog box will appear, asking you which objects to unselect. Choose *all 3*, and *click* the left mouse button over the Ok box to confirm the operation and unselect all 3 objects.

- Move the cursor to coordinates (2,-7), and draw a $5\lambda \times 4\lambda$ box. Move the cursor to coordinates (5,-4) and press the $\boxed{\text{a}}$ key to invoke the `alter-geometry` command. When the dialog box appears, select the "Box on POLY" by *clicking* the left mouse button in the box to the left of the text, and then *click* the left mouse button again over the box marked "Ok" to confirm the selection. The $4\lambda \times 4\lambda$ box on POLY will change to $5\lambda \times 4\lambda$.

- Move the cursor to (9,-7), draw a $4\lambda \times 4\lambda$ box, and press $\boxed{\text{s}}$ to select the 3 objects enclosed by the box.

- *Press* the right mouse button and move it 1λ to the right, then *release* the button. Press $\boxed{\text{m}}$ to move the POLY, COPO, and MET1.

- Unselect everything with $\boxed{\text{\^{}u}}$.

- Beginning at (9,-7), draw a $5\lambda \times 4\lambda$ box, and alter the geometry on POLY with the $\boxed{\text{a}}$ key. (Tell VEM to use POLY either by careful positining of the cursor or the dialog box.)
- Press the $\boxed{\text{f}}$ key to restore the entire layout to the **VEM** graphics window.
- Press $\boxed{\text{Z}}$ to `zoom-out`.
- Move the cursor to (-15,6) and draw a $42\lambda \times 25\lambda$ box. Press the $\boxed{\text{z}}$ to `zoom-in` on the box.
- Move the cursor to (-13,9) and draw a $38\lambda \times 14\lambda$ box. Move the cursor to the palette window, position it over the box labeled NWEL, and press the $\boxed{\text{c}}$ key.
- Press $\boxed{\text{f}}$, then save your work by pressing $\boxed{\text{S}}$.

2.2.9 Adding Text

The layout of the NAND gate is electrically complete, but it is useful to add labels and text so others can understand what we have done.

- Move the cursor over the MET1 near the top of the layout. (Coordinates (9,22) will do nicely.)

⋆ *Do not create a point!* With the cursor in the **VEM** graphics window, type:

```
"Vdd"
```

and then press the $\boxed{\text{T}}$ key. ($\boxed{\text{T}}$ is bound to `create-terminal`, which is on the **Edit** sub-pane of the VEM menu). This will create a terminal on the metal box, and label it `Vdd`.

- Move the cursor over the metal connecting the P substrate contact and the NDIF, (coordinates (-2,2), for example), and type:

```
"GROUND"
```

then press $\boxed{\text{T}}$ to create the terminal and label it `GROUND`.

- Label one of the inputs `a` by moving the cursor over one of the $4\lambda \times 4\lambda$ MET1 boxes near the bottom of the window (try (3,-7)), and type:

```
"a"
```

then press $\boxed{\text{T}}$.

- Move to the other input contact (10,-7) and type:

```
"b"
```

then press $\boxed{\text{T}}$ to label the other input `b`.

- Move the cursor to coordinates (14,2) and type:

```
"out"
```

then press $\boxed{\text{T}}$ to label the output `out`. Since the label will actually be over the contact, it may be difficult to see.

Just to be sure that there is no mistake about the type of gate, we can add some text to the layout.

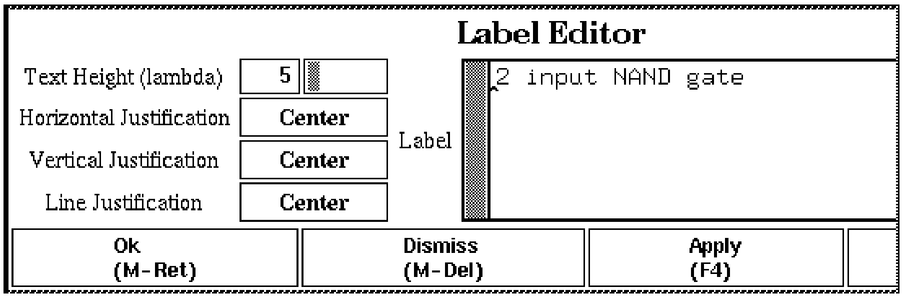

Figure 2.12: VEM Label Editor

- Move the cursor to a point just above the middle of the NAND gate layout. (Around (8,25) is fine, though not at all critical.)
- *Click* the left mouse button to create a point, then move the cursor to the over the box labeled "COM2" in the palette window. (The layer is not critical for this operation.)
- Press E (`edit-label`, also on the **Edit** sub-pane of the VEM menu). This will open an "Edit Label" dialog box, shown in figure 2.12.
- Move the cursor to the large open area in the dialog box, and enter some text. A descriptive label, your name, or both will be fine.
- When you are done, *click* the left mouse button over the box marked "Ok" at the bottom of the dialog box.
- Save the design by pressing S with the mouse in the **VEM** graphics window.

The layer chosen for the label is not really critical. Before this layout is used for fabrication, we will need to convert the OCT database to another format, such as CIF (Caltech Intermediate Form). When this is done, the label will be on a text layer in the CIF file, so it will not actually appear on the masks.

2.2.10 Design Rule Checking

The minimum width and spacing of the geometries on the various layers are determined by the *layout rules* for the technology. These rules vary a bit from one fabrication line to another, but for now we will want to be sure our NAND gate adheres to the MOSIS scaleable CMOS rules, version 6.

Octdrc

Octdrc is a tool which will check a layout for design rule violations.

- Move the cursor to the terminal window where you first invoked VEM.
- Enter the command:

  ```
  octdrc -P -n nand:physical
  ```

  Octdrc will take a few seconds to check all of the layout rules.
- When octdrc is finished, move the cursor to the **VEM** console window and type:

  ```
  "nand:drc"
  ```

  and press the o key to open a new **VEM** graphics window to display the contents of `nand:drc`.

The new **VEM** graphics window (`nand:drc`) should look like `nand:physical`, except for a yellow box over the end of one of the POLY boxes.

- Move the cursor to the new **VEM** graphics window (`nand:drc`), and type:

  ```
  :layer-display
  ```

  VEM will open a dialog box with all of the layers listed, and a selection box beside each layer.

Notice that all of the layers you created are listed in the dialog box, along with a new layer named **Rule3.3**. This is one way of determining which rule has been violated. The `layer-display` command is a useful way to check a layout with numerous violations, since each rule can be checked individually.

- Another way to determine which layout rule has been violated is to move the cursor over the yellow violation box, and press $\boxed{?}$. In the **VEM** console window, the location and layer name of all of the boxes under the cursor will appear, and one of the layers listed will be **Rule3.3**.

MOSIS rule 3.3 states that the gate overlap of active must be at least 2λ. In terms of the OCT layers, the POLY must extend at least 2λ beyond the PDIF or NDIF. This assures that the source and drain won't be shorted if the POLY and PDIF or NDIF are slightly misaligned during fabrication.

- To correct the violation, move the cursor the **VEM** graphics window displaying your original layout, `nand:physical`.
- Move the cursor to coordinates (13,9), and draw a $2\lambda \times 11\lambda$ box.
- Place the cursor over the POLY. Coordinates (14,19) will work.
- Press $\boxed{\text{a}}$ to invoke the `alter-geometry` command, and select POLY when the dialog box appears.
- Save the NAND gate again by pressing $\boxed{\text{S}}$ with the cursor in the **VEM** graphics window with `nand:physical` displayed.

With the layout corrected, you should re-run octdrc to verify you didn't create a new violation when you fixed the overlap error.

- Move the cursor to the terminal window, and once again type:

  ```
  octdrc -P -n nand:physical
  ```

  To check the layout again.

⋆ Move the cursor to the **VEM** graphics window displaying `nand:drc`, and press $\boxed{\text{R}}$ (`re-read`). A dialog box will appear with a warning message; you should confirm the `re-read` by *clicking* the left mouse button over the box marked `Yes`.

- You can close the `layer-display` dialog box by *clicking* the left mouse button over the box marked `OK`.
- Close the `nand:drc` window by moving the cursor over the **VEM** graphics window with `nand:drc` and pressing $\boxed{\hat{}\text{d}}$.

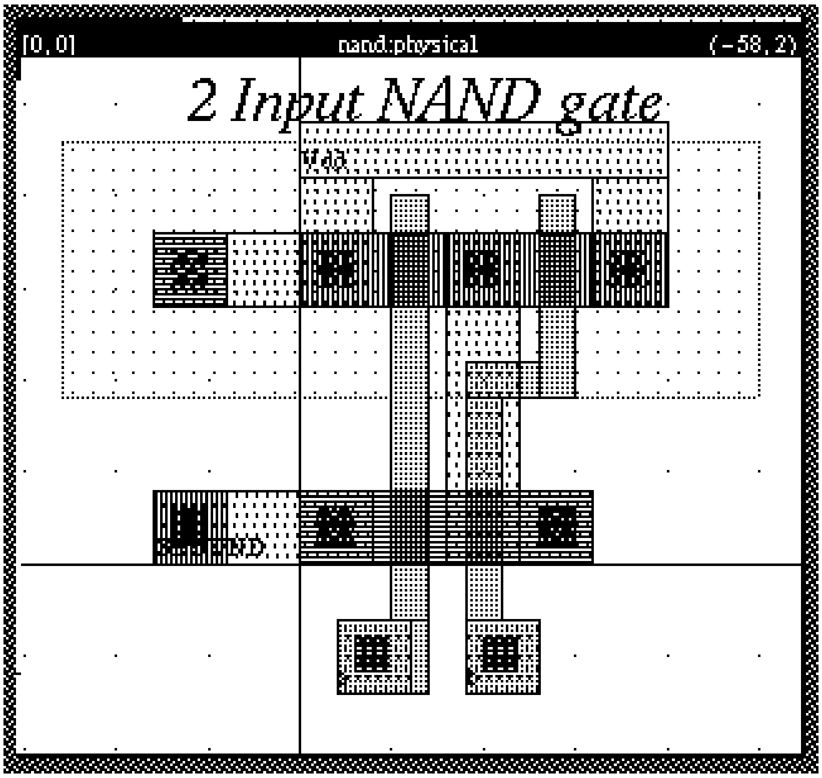

Figure 2.13: NAND gate layout

2.2.11 Wrapping Up!

Congratulations on the successful layout of the 2 input NAND gate! If everything went well, your layout will look approximately like figure 2.13.

- After all of this work, be sure to save you efforts by placing the cursor in the **VEM** graphics window and pressing the $\boxed{S}$ (`save-window`) key.

- Move the cursor to the **VEM** console window, and press $\boxed{{}^\wedge d}$. Closing the **VEM** console window will close all of the other VEM windows.

If there are any unsaved changes in any open windows, VEM will use a dialog box to ask which buffers you want to save. If you do not save all of your work, VEM will confirm the close with another box, warning you that unsaved changes will be lost. Since you do not have any unsaved changes, you will not see these dialog boxes this time.

2.2.12 Adding Properties

Now that you have completed the layout of the NAND gate, you may want to use this cell as part of a larger design. It would be useful if you could simlulate the larger design to be sure it does what you want.

The logic simulator, **musa** cannot simulate physical layouts directly. In order to simulate a circuit which uses this cell, you will need to add some information, called **PROPERITES** to the OCT database for the NAND gate. There are three ways to add properties. The program **octprop** can add properties to a cell, as can the netlist program, **bdnet**. VEM can be used to add properties, but this is not the easiest way to do it. For this example, we will use bdnet to add simulation properties to the NAND gate.

Simulation properties are added to the interface facet, so you will need to be sure one exists for the NAND gate. Simply type:

```
vulcan nand:physical
```

to create the interface facet.

Create a file, `nandProps.bdnet`, with the following contents:

```
(FACET nand:physical:interface
    (SPROP "TECHNOLOGY" "scmos")
    (SPROP "VIEWTYPE" "PHYSICAL")
    (SPROP "EDITSTYLE" "PHYSICAL")
    (SPROP CELLCLASS "LEAF")
    (SPROP CELLTYPE "COMBINATIONAL")
    (BAG PERMUTABLE
(FTERM a)
(FTERM b)
    )
    (FTERM out
(SPROP LOGICFUNCTION "(! (* a b))")
(SPROP DIRECTION "OUTPUT")
(SPROP TERMTYPE "SIGNAL")
    )
    (FTERM a
(SPROP DIRECTION "INPUT")
(SPROP TERMTYPE "SIGNAL")
    )
    (FTERM b
(SPROP DIRECTION "INPUT")
(SPROP TERMTYPE "SIGNAL")
    )
    (FTERM "Vdd"
(SPROP TERMTYPE "SUPPLY")
(RPROP PEAKCURRENT 0.0001 )
    )
    (FTERM "GND"
(SPROP TERMTYPE "GROUND")
(RPROP PEAKCURRENT 0.0001 )
    )
)
```

The names and allowable values for the various properties are listed in the Octtools policies guide. For more information, see Part II: Reference, of this book.

Once you have created the file, you should run **bdnet** to actually add the properties to the database.

```
bdnet nandProps.bdnet
```

If you use this cell as part of another circuit, this cell will now simulate correctly in **musa**.

2.2.13 Printing the Layout (Optional)

If you have a PostScript printer, you can print the NAND gate layout. You will need to determine the name of the printer.

- Move the cursor to the terminal window, and enter the command:

```
oct2ps nand:physical | lpr -Ppostscript_printer
```

where you substitute the name of your PostScript printer in the lpr command.

2.3 Symbolic Editing with VEM

Paul B. Cohen
Massachusetts Microelectronics Center

You should already be familiar with the content of the Section 2.1. Start VEM on a cell called `nand:symbolic`, with TECHNOLOGY=scmos, VIEWTYPE=SYMBOLIC and EDITSTYLE=SYMBOLIC.

2.3.1 Drawing Circuitry

So far, we have learned how to enter arguments, but we have not actually added any circuitry into VEM. We can add wiring, transistors, or other OCT cells using the `create` command. **Points**, **lines**, **boxes**, and **text** are used to define which **object** we would like to add and where to add it.

In VEM interconnect, contacts, terminals, and transistors are considered "primitives". The primitives are stored in the OCT library. (For scmos, the library is in `$OCTTOOLS/tech/scmos`). Each time we add one of these **objects** to the cell we are working on, we *create* an *instance* of that **object**. Other cells, whether they be from a common library or of our own design, can also be instantiated with the `create` command.

2.3.2 Palettes and the VEM Menu

In VEM, **points**, **lines**, and **boxes** are used to indicate where an instance is to be created. VEM will also need to know what you would like to create. A simple way to add objects to your cell is via *instance inheritance*. (If you have tried the physical editing tutorial, *instance inheritance* is similar in concept to *layer inheritance*).

When you opened the **VEM** graphics window on the cell `nand:symbolic`, VEM confirmed certain default property values for the cell. The first property was the "TECHNOLOGY" property, which has a default value of **scmos**. In order to determine what layers are available for wiring in a given technology, and to create a small sample of each layer for the `create` command to use for instance inheritance, we will open a *layer palette window* for the scmos technology.

- Make sure the cursor is in the **VEM** graphics window, and *click* the middle mouse button.
- The main VEM menu will appear with 8 major categories, each with an arrow indicating commands specific to that category are on a sub-pane of the main menu.
- Move the cursor over the arrow ($\Rightarrow$) to the right of the **System** entry on the main menu. The System sub-menu will appear, as shown in figure 2.7.
- Move the cursor to the `palette` command, and *click* the left mouse button.
- A palette window will appear, with the title `.../layers:symbolic`. (Remember, you may need to position the outline and *click* the left mouse button to create the window.) Figure 2.14 shows a sample symbolic editing layer palette for the scmos technology.
- If the layer palette window is covering the **VEM** graphics window, you should move one or both of the windows so both the layer palette window and the **VEM** graphics window are completely visible.

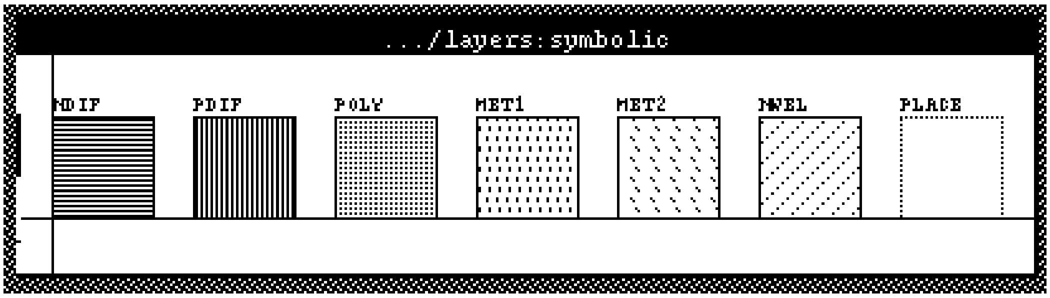

Figure 2.14: Symbolic editing palette for scmos

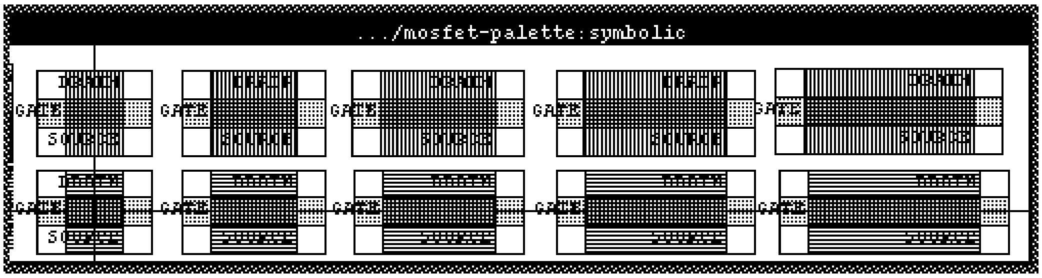

Figure 2.15: Mosfet Palette

(The layers available in the palette do not correspond exactly to CMOS mask layers. Later in this tutorial we will use most of the layers in the palette, so we will see where each of the layers should be used.)

There are a number of other palettes available for symbolic editing. A particularly useful one is the mosfet palette, which contains a variety of PMOS and NMOS transistors.

- Be sure the cursor is in the **VEM** graphics window, and type:

  `"mosfet"`

  Notice that the text echos in the **VEM** console window.

- With the cursor in the **VEM** graphics window, *click* the middle mouse button to invoke the VEM main menu again.

- Move the cursor over the arrow (⇒) to the right of the **System** entry on the main menu. The System sub-menu will appear, as shown in figure 2.7.

- Move the cursor to the `palette` command, and *click* the left mouse button.

- A second palette window will appear. The new palette will have the name `.../mosfet-palette:symbolic` in the title bar. The mosfet palette contains 10 transistors (5 PMOS along the top, 5 NMOS along the bottom) as shown in figure 2.15.

- If necessary, move the mosfet palette so it is not covering either the **VEM** graphics window or the layer palette.

There are a number of other palettes available, but the two palettes currently on the screen are the most useful. Additional transistor sizes are available on the xmosfet palette. Contacts and terminals are included on the connector palette. Since contacts and terminals are usually added automatically by VEM, users do not generally need to open the connector palette. For the purposes of this tutorial, we will need only the layer palette and the mosfet palette. Once you are familiar with VEM, feel free to explore the other palettes available.

- To find out what palettes are available, make sure the cursor is in the **VEM** graphics window, and type:

  ```
  :list-palettes
  ```

  The command will echo to the **VEM** console window, and a list of the palettes available will be printed in the **VEM** console window

2.3.3 Adding Transistors

In this and subsequent section, we will create the layout for a 2 input NAND gate. The gate will then be simulated and compacted. Once this is done, we will instantiate the NAND gate in another cell to demonstrate the hierarchical nature of the tools. The goal of this tutorial is to learn the use of VEM, not to create the best NAND gate layout possible. There may be times when you are asked to perform an operation which you suspect may not be correct. Please eperform the operations as specified, since the purpose will be to show how to correct the errors.

- In the **VEM** graphics window, move the cursor to coordinates -12,20. You can use the cursor information at the right hand edge of the title bar to determine where (-12,20) is on the screen, or you can simply count grid points.
- *Click* the left mouse buttonto create a point at (-12,20).
- ⋆ Move the cursor to the mosfet palette (the title bar reads `.../mosfet-palette:symbolic`) and position the cursor over the PMOS transistor in the upper left hand corner of the mosfet palette.
- *Click* the middle mouse button to invoke the VEM main menu.
- When the main menu appears, move the cursor to the arrow (⇒) to the right of the **Edit** entry on the VEM main menu.
- ⋆ The first item on the edit sub-pane is `create`. Move the cursor over this entry and *click* the left mouse button to create an instance of the transistor in the **VEM** graphics window. (Part of the transistor may extend off the top of the **VEM** graphics window. This is OK.)

In OCT terminology, you have created an instance of
`$OCTTOOLS/tech/scmos/mosfets/pmos.4.2:physical`
in your circuit, `nand:symbolic`. The cell, `nand:symbolic` "contains" the instance of the transistor.

- Move the cursor to coordinates (0,-6) in the **VEM** graphics window.
- Create a point by *clicking* the left mouse button.
- Move the cursor to coordinates (0,-15) in the **VEM** graphics window and create another point by *clicking* the left mouse button.
- Move the cursor to the mosfet palette window, and position it over the NMOS transistor in the lower left hand corner of the palette.
- *Click* the middle mouse button to invoke the VEM menu, and position the cursor over the arrow (⇒) next to the **Edit** entry.
- ⋆ When the edit sub-pane appears, move the cursor over the `create` entry, *but do not click the left mouse button yet.*
- Notice the small letter "c" next to the `create` command. This is the default *key binding* for the `create` command.

- Now go ahead and *click* the left mouse button. You will see 2 NMOS transistors instantiated in the **VEM** graphics window.

The $\boxed{\text{c}}$ key is "bound" to the `create` command. In general, there are 3 ways to enter VEM commands, though not all of the commands can be entered in all 3 ways. The `create` command, for example, can be entered via the menu, as we have done, be pressing the $\boxed{\text{c}}$ key, or by typing the command preceeded by a colon(:). We will take a closer look at key bindings later in this tutorial.

2.3.4 Selecting and Copying Objects

While we could have instantiated 2 PMOS transistors, we can also use the `copy-objects` command to create the second transistor. Before we can copy the transistor, we will need to select it with the `select-objects` command.

- Move the cursor to coordinates (-12,20) in the **VEM** graphics window. The cursor should be centered over the PMOS transistor.
- Press the $\boxed{\text{s}}$ key to select the PMOS transistor.
- Note that the entire transistor is selected as a single object.

In the **VEM** console window, the command line indicates we have selected the fifth type of VEM argument, the **object**. The command line in the **VEM** console window should read:
vem> objects(1)

- *Click* the left mouse button to create a point at coordinates (-12,20).
- Move the cursor to coordinates (12,20) in the **VEM** graphics window, and *click* the left mouse button again to create a second point. The command line in the **VEM** console window will read:

  vem> objects(1) points(2)

  Thus, you are using the mouse in the **VEM** graphics window to build an argument list.
- *Click* the middle mouse button to invoke the VEM main menu, and move the cursor over the arrow next to the **Selection** entry.
- Locate the `copy-objects` entry on the selection sub-pane. (Note the small "x" indicating the key binding for the `copy-objects` command). *Click* the left mouse button to execute the `copy-objects` command.
- A second PMOS transistor has been instantiated in the **VEM** graphics window. This transistor is still selected. Move the cursor over the new transistor (coordinates (12,20) will work) and press the $\boxed{\text{u}}$ key, which is bound to the command `unselect-objects`. (You may have noticed earlier that `unselect-objects` can also be found on the **Selection** sub-pane of the VEM main menu).

The `copy-objects` command requires a list of **objects** to be copied, as well as 2 **points** indicating the displacement of the copy. VEM will copy the selected objects, and displace the copy by the amount indicated by the X–Y difference between the points. In this case, the points were 24λ apart in the X direction and 0λ apart in the Y direction.

The 2 PMOS transistors probably extend a bit off the top of the **VEM** graphics window. We can use the `show-all` command to fit what we have drawn so far into the **VEM** graphics window.

- With the cursor anywhere in the **VEM** graphics window, press the $\boxed{\text{f}}$ key, which is the key binding for the `show-all` command. (`Show-all` is on the **Display** sub-pane of the VEM menu).

At this point, your layout should resemble figure 2.16.

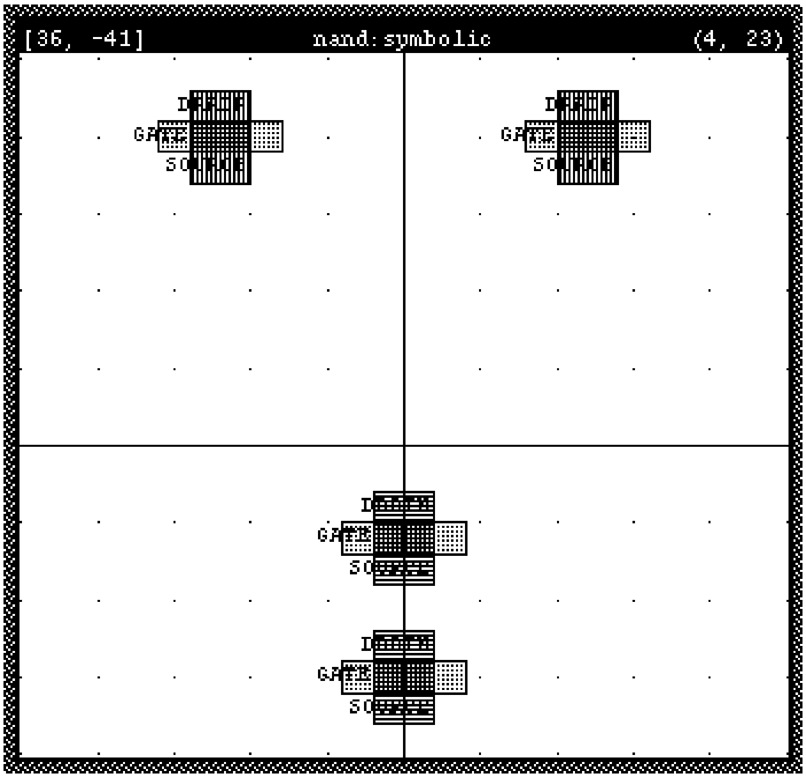

Figure 2.16: Partially completed NAND gate layout

2.3.5 Saving Your Work

It is generally a good idea to save you work periodically. There are 2 command in VEM for saving you work. These are `save-window` and `save-all`. Both commands are bound to the same key, $\boxed{\text{S}}$. This is an example of the context sensitive nature of VEM.

- Move the cursor to the **VEM** console window, and press the $\boxed{\text{S}}$ key again.
- Notice the command `save-all` is echoed in the **VEM** console window.

VEM will open a dialog box asking you which buffers you would like to save. Figure 2.17 shows what this dialog box will look like.

- Move the cursor over the box next to `nand:symbolic:contents;`, and *click* the left mouse button. A check ($\surd$) will appear in the box.

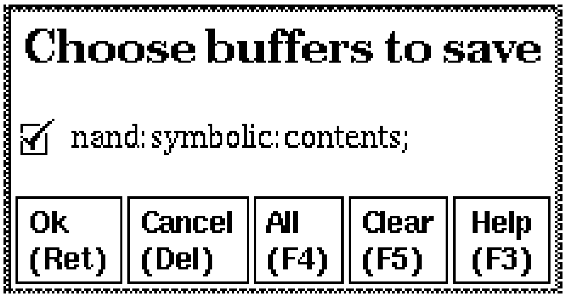

Figure 2.17: VEM dialog box for the save-all command

- Move the cursor over the box labeled "OK", and *click* the left mouse button again to confirm the `save-all` operation.

You can also save your work from the **VEM** graphics window. The $\boxed{\text{S}}$ is bound to the `:save-window` command when it is pressed in a **VEM** graphics window.

- Move the cursor to the **VEM** graphics window, and press the $\boxed{\text{S}}$ key. (Remember, VEM is case sensitive.)
- You will see the command `:save-window` echoed in the **VEM** console window.

2.3.6 More Selecting, Moving Transforming and Unselecting

It will make the connection of the transistors a bit simpler if the drain terminals of the PMOS transistors were "facing" each other. We can accomplish this with the `transform` command.

- In the **VEM** graphics window, move the cursor to coordinates (-17,16). This is just below and to the left of the left hand PMOS transistor.
- Draw a box exactly $10\lambda \times 8\lambda$. Remember, to draw a box you should *press* the left mouse button and move the mouse while *holding* the button. *Release* the button when the box is the right size. (Use the argument size entry in the title bar to determine the size of the box.)
- Press the $\boxed{\text{s}}$ key. This will select all objects completely enclosed by the box. In this case, the only object completely enclosed by the box is the PMOS transistor.

⋆ Keep the cursor in the **VEM** graphics window and type:

  `"270"`

  and press the $\boxed{\text{t}}$ key. ($\boxed{\text{t}}$ is the bound to the `transform` command on the **Selection** page of the VEM menu).

⋆ The selection box has changed, but the transistor itself hasn't moved. Press the $\boxed{\text{m}}$ key to actually move the transistor to its transformed location.

- Place the cursor over the selected transistor.
- *Click* the middle mouse buttonto invoke the VEM menu, and the locate the `unselect-objects` command on the **Selection** sub-pane of the VEM menu.
- *Click* the left mouse button to execute the `unselect-objects` command. Note that the key binding is "u".
- Move the cursor to coordinates (6,20) in the **VEM** graphics window.
- Draw a line, beginning at (6,20) and extending 8λ to the right ([8,0]). (Remember, *click* the left mouse button to create a point, then *press* the left mouse button over that point and move the mouse to the right. *Release* the left mouse button when the end of the line is where you want it.) The end of the line will be over the right hand PMOS transistor.
- Press the $\boxed{\text{s}}$ key. This will select all objects the line intersects, which in this case is only the right hand transistor.
- Press the $\boxed{\text{t}}$ key to transform the selection box.
- Press the $\boxed{\text{m}}$ key to move the transistor to its transformed location.
- With the cursor over the selected transistor, press $\boxed{\text{u}}$ to unselect the object under the cursor (the transistor).

In VEM, we can use **points**, **boxes**, or **lines** as arguments to the `select-objects` command. The `unselect-objects` command, and many other VEM commands, operate in a similar fashion, as we shall soon see.

2.3.7 Pan, Zoom, and Multiple Windows.

In order to complete the NAND gate, we will need to interconnect the transistors. In order to connect the source terminals of the PMOS transistor, we will need to pan the **VEM** graphics window up so we have more room above the layout to work.

- Place the cursor between the 2 PMOS transistors in the **VEM** graphics window, at coordinates (0,20).
- Press the p key to `pan` the point under the cursor to the center of the **VEM** graphics window.

We won't be needing the mosfet palette for the remainder of this tutorial. In order to keep the screen uncluttered, we can close the mosfet palette window. We will still need the layer palette.

- Move the cursor over the mosfet-palette window, and press ^d. This is the key binding for `close-window`, which is on the **System** sub-pane of the VEM menu.

We can now begin to interconnect the NAND gate.

- Move the cursor over the drain of the left hand PMOS transistor. Coordinates (-10,20) in the **VEM** graphics window will place the cursor over the center of the drain.
- Draw a line from (-10,20) to (10,20). (*Click* the left mouse button, then *press* the left mouse button with the cursor over the point and move the mouse). The argument size entry will read [20,0].
- Move the cursor to the layer palette, and position it over the box labeled PDIF. (PDIF stands for **P DIF**fusion).
- Press the c key to create a path on the layer PDIF.

In the symbolic editing sytle of VEM, connections are by default *snapped* to 90° angles. In order to make the accurate positioning of vertices simpler, the last vertex created in VEM remains movable until a new vertex is created or the line is terminated.

- Move the cursor to the source of the left hand PMOS transistor, at coordinates (-14,20).
- Draw a line from (-14,20) to (-20,20).

⋆ With the cursor over the end of the line at (-20,20), *press* the left mouse button and move the cursor upwards to coordinates (-20,30). This will continue the line you just created, but a vertex will be created. As you move the cursor upwards, move it left and right a bit, and notice how the vertex at moves with the cursor. *Release* the left mouse button at (-20,30)

⋆ With the cursor at coordinates (-20,30), *press* the left mouse button, move the cursor to coordinates (20,30), and *release* the left mouse button.

⋆ With the cursor at coordinates (20,30), *press* the left mouse button, move the cursor to coordinates (20,20), and *release* the left mouse button.

⋆ With the cursor at coordinates (20,20), *press* the left mouse button, move the cursor to coordinates (14,20), and *release* the left mouse button.

- Move the cursor to coordinates (0,20), and press the c key to create a path on PDIF connecting the sources of the PMOS transistors.

Note that *instance inheritance* can be used with the cursor over any sample of the object you want to create. The various palettes are just a simple way to get a sample of various useful objects on the screen to use for instance inheritance.

At each vertex, VEM has automatically created a terminal.

- For the next operation, we will need to be able to see the NMOS transistors in the **VEM** graphics window, so press $\boxed{\text{f}}$ to execute the `show-all` command.

VEM is a multiple window application using the X Window System. This means we can have multiple **VEM** graphics windows open on a design, or we can have different designs in different **VEM** graphics windows on the screen at the same time. VEM allows us to work between windows on the same design.

- Move the cursor to coordinates (-7,-9) and draw a $14\lambda \times 16\lambda$ box (the argument size entry in the title bar will read [14,16]).

⋆ Press the $\boxed{\text{o}}$ key to open another **VEM** graphics window, and position it on the screen so it is not covering the the Y axis of the original **VEM** graphics window. You should leave at least 20λ on either side of the axis uncovered. (Covering the edge of the original window is OK. This is not a VEM restriction, but if you cover the Y axis, the next step will be hard to do). Be sure to leave the layer palette window uncovered.

⋆ Move the cursor to coordinates (-4,15) and draw an $8\lambda \times 10\lambda$ box. (If you've covered the argument size entry, the other corner of the box will be at (4,25).

⋆ Press the $\boxed{\text{o}}$ key to open a third **VEM** graphics window, and position it so it is not covering the second **VEM** graphics window. Covering the original **VEM** graphics window is OK at this point, but if you can avoid it, do so. Be sure to leave the layer palette window uncovered.

- Move the cursor to coordinates (0,-4) in the second **VEM** graphics window, and draw a 6λ vertical line. The argument size entry will be [0,6], and the endpoint will be at (0,2).

- Move the cursor to the layer palette window, and position it over the lox labeled NDIF. (NDIF stands for **N DIF**fusion).

- Press the $\boxed{\text{c}}$ key to create a path on NDIF.

⋆⋆ Position the cursor at coordinates (0,2) in the second **VEM** graphics window.

⋆ *Click* the left mouse button, and then *press* it again to begin to draw a line. Move the mouse to coordinates (0,20) in the third **VEM** graphics window, and *release* the left mouse button. (The end of the line will be centered over the PDIF path we created earlier).

- Move the mouse to the layer palette window, and position it over the box labeled MET1. Press the $\boxed{\text{c}}$ key to create a path on first metal. (MET1 stands for **MET** al **1**).

- Notice the contacts between MET1 and NDIF in the second **VEM** graphics window and between MET1 and PDIF in the third **VEM** graphics window have been added automatically.

When a newly created path terminates on an existing path ($\top$), VEM assumes you want the 2 paths connected, and adds the appropriate contact, if possible. When a new path crosses an existing path (+), no connection is assumed.

- Close the third **VEM** graphics window by moving the cursor into that window and pressing $\boxed{\hat{\ }\text{d}}$ to close it. Notice the metal connection appears in the original **VEM** graphics window also.

Many other VEM commands also operate between windows.

- In the original **VEM** graphics window, move the cursor to a point midway between the 2 NMOS transistors. Coordinates (0,-11) will do nicely.

- *Click* the left mouse button to create a point.

- Move the cursor to the second **VEM** graphics window, and press $\boxed{\text{p}}$ to `pan` the point in the original **VEM** graphics window to the middle of the second **VEM** graphics window.

- Move the cursor down a bit until it is over the bottom NMOS transistor. (Coordinates (0,-13) in the second **VEM** graphics window).

- Beginning at coordinates (0,-13), draw a line 5λ upwards. This line should end at (0,-8).

- Move the cursor to the layer palette window, and position it over the NDIF box. Press $\boxed{\text{c}}$ to create a path on NDIF in the graphics window.

Once we connect the gate terminals of the transistors, our NAND gate will be electrically complete.

- Close the second **VEM** graphics window by moving the cursor to that window and pressing $\boxed{\hat{}\text{d}}$ to execute the `close-window` command.

- Move the cursor to coordinates (-3,-6) in the **VEM** graphics window, which will put it over the gate of the top NMOS transistor.

- Draw a $-4\lambda \times 0\lambda$ line.

- With the cursor at (-7,-6), *press* the left mouse button and move the cursor to coordinates (-12,17). This is over the gate of the left hand PMOS transistor.

- Move the cursor to coordinates (12,17), and draw a $0\lambda \times -32\lambda$ line (32λ downward), then continue the line with a $-9\lambda \times 0\lambda$ segment (9λ to the left). The end of the second line will be over the gate of the bottom NMOS transistor.

- Since the lines are not contiguous, VEM will note that you have drawn 2 lines with 2 segments each by echoing:

  vem> lines(2) lines(2)

  in the **VEM** console window.

⋆ Move the cursor over the box labeled POLY in the layer palette window. *Do not try to use instance inheritance by putting the cursor over the poly gate on the transistor.* (Remember that the transistors are objects, and instance inheritance creates a copy of the object under the cursor. If you place the cursor over the gate of the transistor and invoke `create`, VEM will try to create another instance of the transistor, not the POLY gate.)

- Press $\boxed{\text{c}}$ to create 2 paths on POLY, connecting the transistors.

- Save your work to this point by pressing $\boxed{\text{S}}$ with the cursor in the **VEM** graphics window.

2.3.8 Labeling Terminals

At this point, the NAND gate is electrically complete, but we should label some terminals for simulation.

- Move the cursor to coordinates (-20,30). This is over one of the terminals on the PDIF layer.

⋆⋆ *Do not create a point!* Type:

```
"Vdd"
```

and press the $\boxed{\text{c}}$ key to `create` the label for the terminal.

The terminal at (-20,30) is known as an *actual terminal*. Before we can use this symbolic cell as part of another cell, we need to define the legal connection points. The connection points are known as *formal terminals*. The **text** argument to the `create` command indicated to VEM that we wanted to create a formal terminal. Only actual terminals of instances or connectors can be used as implementations of formal terminals.

- Move the cursor to coordinates (0,-17), which will put it over the source of the bottom NMOS transistor.
- Press the $\boxed{\text{p}}$ key to pan this transistor to the center of the screen.
- Move the cursor to the center of the source of the NMOS transistor, and draw a line extending 6λ downwards ([0,-6]). Create and NDIF wire by moving the cursor over the NDIF box in the layer-palette and pressing $\boxed{\text{c}}$.
- Move the cursor to (0,-23) in the **VEM** graphics window and create a formal terminal named "GND" by typing:

  ```
  "GND"
  ```

  and pressing the $\boxed{\text{c}}$ key.
- Press $\boxed{\text{f}}$ to center the layout in the **VEM** graphics window.
- Move the cursor to (-12,-6) and type:

  ```
  "a"
  ```

  and press $\boxed{\text{c}}$ to create the formal terminal "a".
- Move the cursor to (12,-15) and type:

  ```
  "b"
  ```

  and press $\boxed{\text{c}}$to create the formal terminal "b".
- Move the cursor to coordinates (0,10), and create a short line extending 5λ to the right ([0,5]). Create a wire on MET1 by moving the cursor over the nearest MET1 wire ((0,10) will do), and pressing $\boxed{\text{c}}$.
- Move the cursor to (5,10) and type:

  ```
  "out"
  ```

  and press $\boxed{\text{c}}$ to create the formal terminal "out".

2.3.9 Adding Properties

We can add various *properties* to the NAND gate with the `edit-property` command.

- Move the cursor to coordinates (0,30) and invoke the `select-net` command by pressing $\boxed{\text{\^{}n}}$. Note that the entire path on PDIF is selected, as are the source terminals of the PMOS transistors.
- Press $\boxed{\text{\^{}e}}$ to invoke the `edit-property` command. This will bring up a dialog box labeled "Property Editor". Move the cursor to the dialog box, and position it in the area next to the word "Name". Type:

```
NETTYPE
```

- Keep the cursor in the dialog box, but move it to the large box next to the word "Value", and *click* the left mouse button to focus input on this field.
- Type:

```
SUPPLY
```

- Make sure the property type is "String".
- Move the cursor over the box along the bottom of the dialog box marked "New", and *click* the left mouse button, then move the cursor over the box labeled "Ok" and *click* the left mouse button again to remove the dialog box.
- Move the cursor to coordinates (0,-23) and invoke the `select-net` command by pressing ^n. This will select the GND net. (A dialog box may appear - select either entry).
- Press ^e to invoke the `edit-property` command. This will bring up a dialog box labeled "Property Editor". Move the cursor to the dialog box, and position it in the area next to the word "Name". Type:

```
NETTYPE
```

- Keep the cursor in the dialog box, but move it to the large box next to the word "Value". Click the left mouse button to focus input on this field, and type:

```
GROUND
```

- Make sure the property type "String".
- Move the cursor over the box along the bottom of the dialog box marked "New", and *click* the left mouse button, then move the cursor over the box labeled "Ok" and *click* the left mouse button again to remove the dialog box.

2.3.10 Wrapping up!

Congratulations on the completion of the NAND gate layout. If everything went well, your NAND gate should look something like figure 2.18.

2.3.11 Creating the interface facet

You should always create both a contents and an interface facet for each view of any cell you make. The interface facet contains an abstraction of the contents and it is used by many tools because it is normally much faster to open it rather than the contents. You will normally use **vulcan** to create the interface facet of a cell. VEM provides the command `update-interface`, which if invoked on a contents facet, will update the corresponding interface. You need to open the interface facet to see the effect of this command. The interface facet is not saved automatically. You have to remember to save it yourself, just like you remember to save the contents facet.

You can always forget about the interface of the cell you are editing until you need to instantiate it in some other cell. A good practice is to edit the contents facet, save it to disk, immediately run **vulcan** on it.

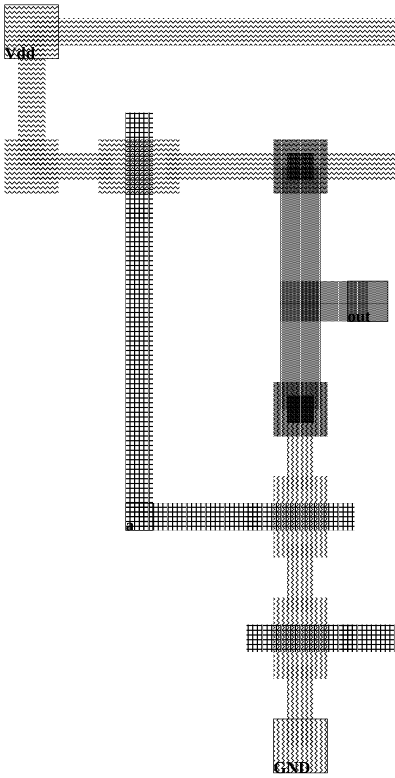

Figure 2.18: NAND gate symbolic layout

2.3.12 Closing VEM

Before you close VEM, you should save your work.

- With the cursor in the **VEM** graphics window, press S to save your work.
- Move the cursor to the **VEM** console window and press ^d to close VEM. Closing the **VEM** console window will close all of the other VEM windows. If there are any windows with unsaved work, vem will use the `:save-all` dialog box to find out which windows you want to save. If you do not save all of the windows which have unsaved changes, VEM will confirm the `:close-window` operation with another dialog box.

2.3.13 Simulation

We can use the **MU**lti Level **S**imul**A**tor, **musa** to simulate the NAND gate and be sure it is correct. We will use a terminal window to run the simulator.

- From the terminal window, type:

  ```
  musa nand:symbolic
  ```

  Musa will respond with a version message, and the prompt:

  musa>

  to indicate you have invoked the simulator.

- The `set` command is used to set a signal (a formal terminal) to a value. Set “a” to 0 by typing, at the musa> prompt,

```
set a 0
```

- The `set` can be abbreviated "se". Set "b" to 0 by typing:

```
se b 0
```

- We do not need to set V_{DD} and GND, since **musa** recognizes the 2 NETTYPE properties, SUPPLY and GROUND, that we assigned earlier. If we had not assigned these properties to any net, then we would need to set V_{DD} and GND explicitly, as we do all other inputs.

⋆ **Musa** does not actually run the simulation until we enter the "evaluate" command. This allows us to set a number of inputs before actually running a simulation. Since both inputs have been set, we can simulate the circuit. Type:

```
evaluate
```

- The `show` command is used to inform **musa** which signals we would like to see. Type:

```
show a b out
```

- **Musa** will respond with:

```
 a:O b:O out:1
```

Musa uses the letter "O" and the number "0" to distinguish between nodes which have been set low by the user (letter "O") and nodes which evaluate to a low value (number "0"). The letter "I" and the number "1" are used to indicate whether a high value has been set by the user or is a simulation result.

- We can save some typing by setting a "watch list" in **musa**. Type:

```
watch io a b out
```

(The `watch` command can be abbreviated `wa`). Continue the simulation by typing the following sequence of commands:

```
se a 1
ev
sh io
```

In this case the `show` command (abbreviated `sh`), was used with the name of the watch list ("io") as an argument. We could have multiple watch lists.

- Finish the simulation by typing:

```
se b 1
se a 0
ev
sh io
se a 1
ev
sh io
quit
```

A complete simuation of the NAND gate is shown below.

```
musa> set a 0
musa> se b 0
musa> evaluate
musa> show a b out
 a:0 b:0 out:1
musa> watch io a b out
musa> se a 1
musa> ev
musa> sh io
 a:I b:0 out:1
musa> se b 1
musa> se a 0
musa> ev
musa> sh io
 a:0 b:I out:1
musa> se a 1
musa> ev
musa> sh io
 a:I b:I out:0
musa> quit
```

2.3.14 Compacting the Layout

When we created the layout for the NAND gate, we did not pay particular attention to the layout rules or to trying to arrive at a reasonably dense layout. **Sparcs** is a layout compaction tool which will minimize the area of the layout and satisfy the spacing rules. (If you have violated a spacing rule, **sparcs** will actually increase the area of your circuit to satsify the rule).

Note: previous versions of this tutorial used **rpc-sparcs**, which is no longer part of the standard distribution. Use **sparcs** instead.

```
# Do the compaction. If the layout cannot be compacted
# sparcs will tell you and give you some hints on how to
# resolve the problem.
sparcs nand:symbolic

# Look at the compacted layout.
vem -F nand:spaced
```

2.3.15 Creating the Interface Facet and Adding Properties

Before we can instantiate the NAND gate in another cell, we will need to create the interface facet. **Vulcan** will create the interface facet.

- From the terminal window, enter the command:

  ```
  vulcan -k nand:spaced
  ```

  We should also add the "CELLTYPE" property to the interface facet. The value of the property should be "MODULE". We can use **octprop** to do this.

  - From the terminal window, type:

    ```
    octprop -pCELLTYPE=MODULE nand:spaced:interface
    ```

2.3.16 Instantiating One Cell in Another

With the NAND gate complete, we can instantiate it in another cell as often as we wish.

- From the **VEM** console window, type:

  `"ao1:symbolic"o`

- Since the cell `ao1:symbolic` does not exist, VEM will open a missing property dialog box, The correct property values should be:

Technology:	scmos
View Type:	SYMBOLIC
Edit Style:	SYMBOLIC

- If the dialog box is correct, *click* the left mouse button over the "OK" box. (Remember, you may need to position the **VEM** graphics window and *click* the left mouse button to actually create the **VEM** graphics window).

We will use 2 instances of the compacted NAND gate to create a circuit with the logic function:

$$output = (in1 \cdot in2) + \overline{in3}$$

which will be implemented using 2 input NAND gates as:

$$output = \overline{\overline{(in1 \cdot in2)} \cdot in3}$$

- Move the cursor to the newly created, blank **VEM** graphics window. Position it at coordinates (0,0).
- *Click* the left mouse button to create a point.
- Type:

  `"nand:spaced"`

  and press $\boxed{\text{c}}$ to create an instance of the NAND gate.

- The NAND gate will probably not be centered in the **VEM** graphics window. Press $\boxed{\text{f}}$ to execute the `show-all` command.
- With the cursor anywhere over the NAND gate, press $\boxed{\text{s}}$ to select the gate. Notice that the entire NAND gate is selected as a single entity. The command line in the **VEM** console window reads:

  vem> objects(1)

- Press $\boxed{\text{Z}}$ to `zoom-out`. The design will appear smaller.
- Move the cursor to coordinates (-25,0) and *click* the left mouse buttonto create a point.
- Create another point at (5,0) by *clicking* the left mouse button at coordinates (5,0).
- Execute the `copy-objects` command by pressing $\boxed{\text{x}}$. (If you prefer to use the menu, feel free to do so).
- Press $\boxed{\text{f}}$ again to fit everything inside the **VEM** graphics window.
- Press $\boxed{\hat{}\text{u}}$ to unselect everything.

We have created 2 instances of the NAND gate in our cell ao1. We can connect them to form the new function. For the purposes of this tutorial we will keep the number of open windows to a minimum, even though this may not be the most efficient way to proceed. Once you are familiar with VEM, you should feel free to experiment to find a style which is comfortable for you.

- In the **VEM** graphics window, create a $15\lambda \times 6\lambda$ box beginning at (-3,7) and ending at (12,13).
- Press z to `zoom-in` such that the box now occupies the entire **VEM** graphics window.
- Press P top open a layer palette window. This is the only palette you will need for this circuit.
- Draw an 11λ line from (-1,10) to (10,10).

⋆ Move the cursor to the PDIF box in the layer palette window, and press c to create the path on PDIF. (*Do not try to use instance inheritance with the cursor over the PDIF paths in the* **VEM** *graphics window*).

If you try to use instance inheritance by placing the cursor over one of the PDIF paths in the NAND gate, the results won't be quite what you expect. When you invoke the `create` command, VEM will try to create an instance of whatever is under the cursor. Remember when you selected to NAND gate to copy it, the entire NAND gate was selected, regardless of where (over the gate) the cursor was. So, while it may look like the cursor is over a path on PDIF, it is really over an instance of a NAND gate.

- Press f (from the **VEM** graphics window) to restore the center the full layout in the window.
- Move the cursor to coordinates -10,-23 and draw a line extending 5λ downward [0,-5] to coordinates (-10,-28).
- Continue the line (by *pressing* the left mouse button) 30λ to the right [30,0]. The end will be at (20,-28).
- Continue the line 5λupward to coordinates (20,-23).
- Move the cursor to the layer palette window, and position it over the NDIF box. Press c to create a path on the NDIF layer in the **VEM** graphics window.
- Move the cursor to (-12,-4) and draw a line 15λ to the right, ending at (3,-4).
- Move the cursor to the MET1 box in the layer palette window, and press c to create a path on 1^{st} metal.
- Move the cursor back to (3,-4), and draw a line 10λdownwards, to (3,-15).
- Continue the line to the right to coordinates (14,-15). The new segment will be 11λ long.
- Move the cursor over the POLY box in the layer palette window, and press c.
- Draw a 10λ line beginning at (17,-4) and ending at (27,-4). Create a path on 1^{st} metal by moving the cursor to the MET1 box in the layer palette and presing c.
- Draw a line from (-16,-15) extending 5λ to the left.
- Draw a line from (-5,-20) extending 5λ to the right.
- Draw a line from (25,-20) extending 5λ to the right.
- Create 3 paths on POLY by moving the cursor over the POLY box in the palette window and pressing c.

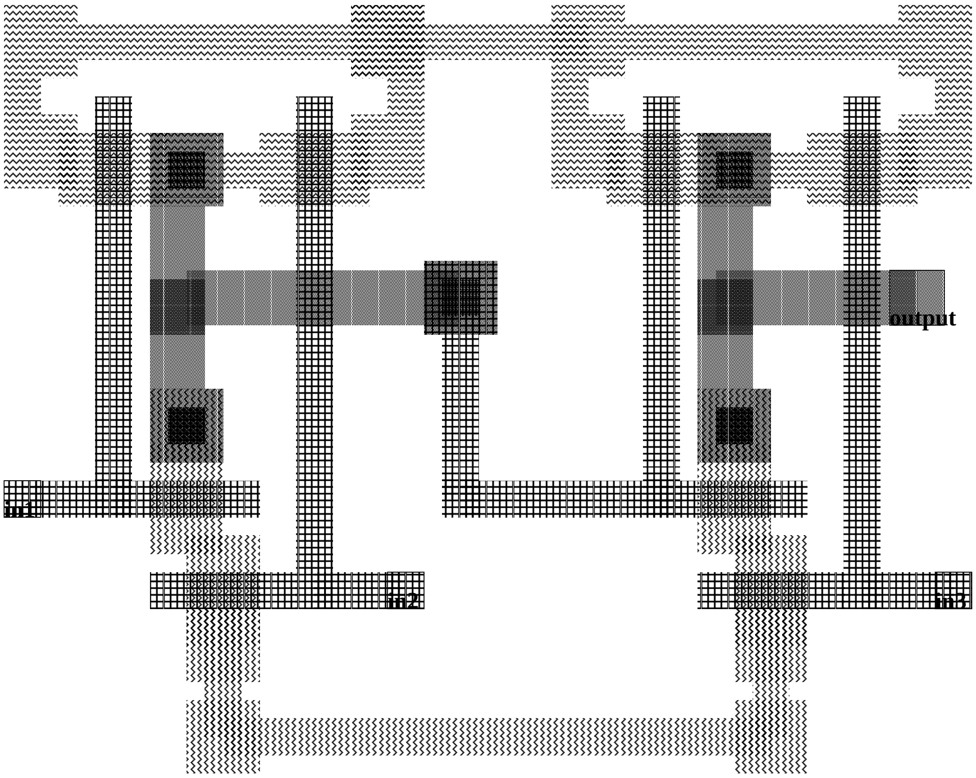

Figure 2.19: AOI symbolic layout

2.3.17 Label the I/O

- Move the cursor to coordinates (-21,-15) and type:

  `"in1"`

  and press c to create the formal terminal named "in1".
- Move the cursor to coordinates (0,-20) and type:

  `"in2"`

  and press c to create the formal terminal named "in2".
- Move the cursor to coordinates (30,-20) and type:

  `"in3"`

  and press c to create the formal terminal named "in3".
- Move the cursor to coordinates (28,-4) and type:

  `"output"`

  and press c to create the formal terminal named "output".

Your circuit should resemble the circuit if figure 2.19.
We're ready to simulate the cell, be first we should save you work.

- With the cursor in the **VEM** graphics window, press S to save the cell.
- Move the cursor to the **VEM** console window and press ^d to close VEM.

2.3.18 Simulating the New Cell

We are now ready to simulate the new cell. Instead of entering the simulation commands directly into **musa**, we can create a file with the simulation commands.

- Using a text editor, create a file `ao1.sim` with the following contents::

```
se in1 0
se in2 0
se in3 0
ev
wa io in1 in2 in3 output
sh io
se in3 1
ev
sh io
se in1 1
ev
sh io
se in2 1
ev
sh io
```

- When the file is complete, enter the command:

```
musa -i ao1.sim ao1:symbolic
```

  to simulate the circuit.

2.3.19 Two useful RPC applications

If you are gong to design symbolic layout, you should become familiar with two RPC applications: **rpc-symhelp** and **rpc-steps**. Both applications can be started form the "Application" menu in VEM.

An RPC application adds commands to VEM. **Rpc-symhelp** knows about rules of symbolic layout, and performs some checks to ensure that other tools, down the line, will be able to perform their task. For example, the simulator **musa** is much easier to use if all transistors and nets in a symbolic layout have a short name. The `fix-names` command avaiable through RPC gives a name to all instances and nets that do not have any. Names with strange characters (such as spaces, or shell meta-characters) are also spotted and possibly fixed, because many tools cannot deal with such names. **Rpc-symhelp** performs many more checks, and it also allows you to annotate formal terminals, and to estimate parasitics (capacitance and resistance) and you should definitely experiment with it.

Rpc-steps knows about SPICE and can give you a SPICE deck starting from a symbolic layout (or from a schematic). This RPC application allows you to define models for all transistors in the layout, to specify waveforms on all INPUT terminals, loads on all OUTPUT terminals, what kind of analysis you want, and the output you would liek to get from SPICE. When everything is ready, the command `prepare-spice-deck` will produce the SPICE deck.

Each vem window can have at most one RPC application, but since you can have mamy windows opened on the same facet, it is possible to have different RPC applications running on the same facet. For example, you can open two windows on `nand:symbolic` and have **rpc-symhelp** running on one and **rpc-steps** runing on the other.

Both RPC applications have a stand alone version, called **symhelp** and **steps**. Use these versions if the circuit is large – they run much faster.

2.3.20 Printing the Layout (Optional)

If you have a PostScript printer, you can print the NAND gate layout. You will need to determine the name of the printer.

- With the cursor in the terminal window, enter the command:

  ```
  oct2ps ao1:symbolic | lpr -Ppostscript_printer
  ```

  where you substitute the name of your PostScript printer in the lpr command.

2.4 Schematic Capture with VEM

Paul B. Cohen

Massachusetts Microelectronics Center

You should be familiar with the content of the Section 2.1. Start VEM on a cell called `1bit:schematic`

2.4.1 Schematic Capture with VEM

The schematic capture commands in VEM allow the user to create and edit designs in a technology independent form using standard symbols for logic and circuit elements. Since schematic capture in VEM is hierarchical, circuits may be created, simulated, and used in a higher level circuit. For logic designs, the schematic may be used as input to **misII**, which will optimize the logic and map it into a variety of layout styles.

2.4.2 Palettes and the VEM Menu

In VEM, **points**, **lines**, and **boxes** are used to indicate where an instance is to be created. VEM will also need to know what you would like to create. A simple way to add objects to your cell is via *instance inheritance*. (If you have tried the physical editing tutorial, *instance inheritance* is similar in concept to *layer inheritance*).

When you opened the **VEM** graphics window on the cell `1bit:schematic`, VEM confirmed certain default property values for the cell. The first property was the "TECHNOLOGY" property, which has a default value of **schematic**. In order to determine what gates and devices are available in a given technology, and to create a sample of each gate and device for the `create` command to use for instance inheritance, we will open two *palette windows* for the schematic technology. One palette will contain the gates we will use in this tutorial, the other contains some discrete devices, though we will not use the discrete devices in this tutorial.

- Be sure the cursor is in the **VEM** graphics window, and type:

  `"gate"`

  Notice that the text echos in the **VEM** console window.

- With the cursor in the **VEM** graphics window, *click* the middle mouse button to invoke the VEM main menu. Figure 2.20 shows the VEM main menu.
- Move the cursor over the arrow (⇒) to the right of the **System** entry on the main menu. The System sub-menu will appear, as shown in figure 2.7.
- Move the cursor to the `palette` command, and *click* the left mouse button.
- The gate palette window will appear. The palette will have the name `.../gates-palette:schematic` in the title bar. The gate palette contains an assortment of logic gates as shown in figure 2.21.
- If necessary, move the gate palette so it is not covering the **VEM** graphics window.

(If you accidentally invoke the VEM menu and don't want to actually execute any of the commands, simply move the cursor to some blank spot on the screen and *click* the right mouse button.)

Another useful palette for VEM schematic capture is the device palette. This palette contains a number of discrete devices such as resistors, capacitors and transistors. Figure 2.22 is a sample of the device palette in the VEM schematic editing style.

- With the cursor in the **VEM** graphics window, type:

  `"device"`

schematic	
System	⇒
Display	⇒
Options	⇒
Props & Bags	⇒
Undo	⇒
Edit	⇒
Selection	⇒
Application	⇒

Figure 2.20: Main VEM menu

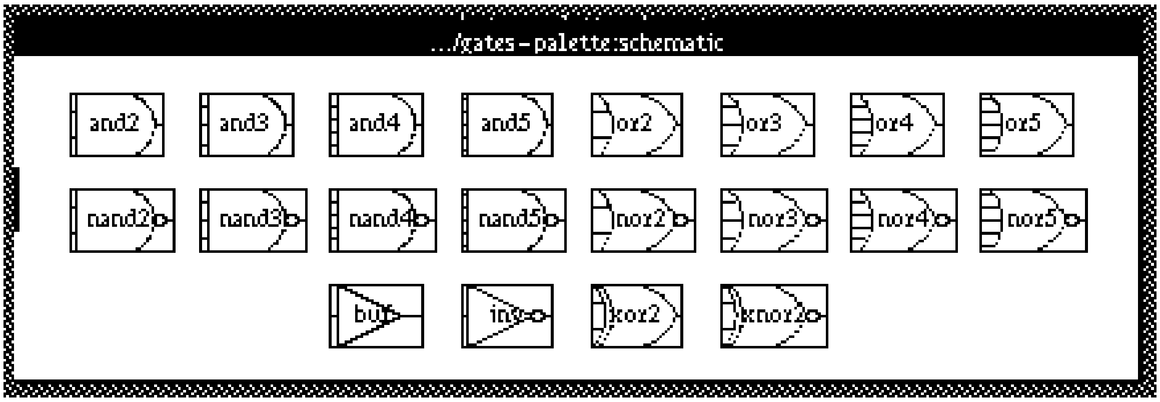

Figure 2.21: Gates Palette

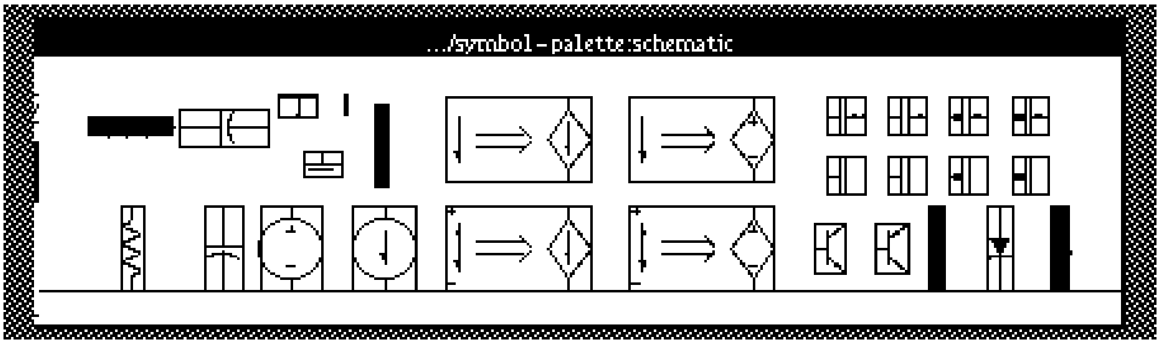

Figure 2.22: Device Palette

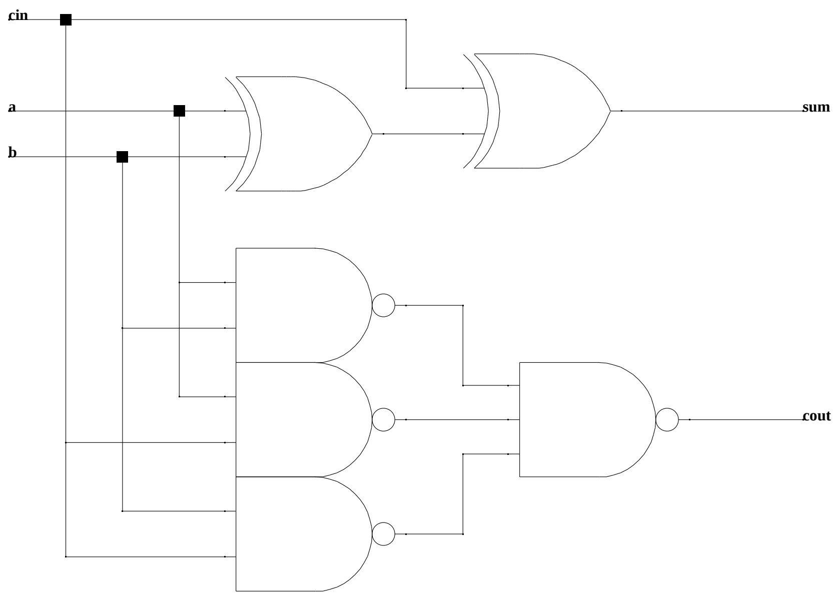

Figure 2.23: 1 bit adder schematic

- With the cursor in the **VEM** graphics window, *click* the middle mouse button to invoke the VEM main menu again.
- Move the cursor over the arrow (⇒) to the right of the **System** entry on the main menu. The System sub-pane will appear, as shown in figure 2.7.
- Move the cursor to the `palette` command, and *click* the left mouse button.
- The device palette window will appear. The palette will have the name `.../symbol-palette:schematic` in the title bar. The symbol palette contains an assortment of discrete device symbols as shown in figure 2.22.
- You won't be needing this palette for the tutorial, so move the cursor to the symbol palette and press $\boxed{\text{\^{}d}}$ to close the palette window.

There are a number of other palettes available, but the gate and device palettes are the most useful. For the purposes of this tutorial, we will need only the gate palette, Once you are familiar with VEM, feel free to explore the other palettes available.

- To find out what palettes are available, make sure the cursor is in the **VEM** graphics window, and type:

  ```
  :list-palettes
  ```

  The command will echo to the **VEM** console window, and a list of the palettes available will be printed in the **VEM** console window

2.4.3 Adding Gates

For this tutorial, we will create a 4 bit binary ripple carry adder. We will do this by first creating a 1 bit full adder, then creating a 4 bit adder using 4 instances of the 1 bit adder. A sample schematic is shown in figure 2.23 as a reference.

- With the cursor in the **VEM** graphics window, move the cursor to coordinates (0,0) and *click* the left mouse button to create a point.

- Move the cursor to the gate palette window and position it over the XOR2 gate.
- *Click* the middle mouse button to invoke the VEM main menu. Move the cursor over the arrow (⇒) next to the **Edit** entry, and the Edit sub-pane will appear.
- The first entry on the Edit sub-pane is `create`. Note the small letter "c" next to the `create` command. This is the default *key binding* for the `create` command.
- Move the cursor over the `create` entry and *click* the left mouse button to `create` an *instance* of the XOR2 gate in your schematic.

The c key is *bound* to the `create` command. In general, there are 3 ways to enter VEM commands, though not all of the commands can be entered in all 3 ways. The `create` command, for example, can be entered via the menu, as we have done, be pressing the c key, or by typing the command preceeded by a colon(:). We will take a closer look at key bindings later in this tutorial.

2.4.4 Selecting and Copying Objects

While we could have instantiated the XOR2 gate twice by creating 2 points, we can also use the `copy-objects` command to create the second instance.

- Move the cursor over the newly instantiated XOR2 gate in the **VEM** graphics window, (coordinates (5,5), for example) and press s to execute the `select-objects` command. In the **VEM** console window, the command line will read:

  vem> objects(1)
- Move the cursor to coordinates (-1,5) and *click* the left mouse button to create a point.
- Move the cursor to coordinates (20,5) and create another point. The command line in the **VEM** console window will read:

  vem> objects(1) points(2)

  Thus, you are using the mouse in the **VEM** graphics window to build and argument list. We will use this argument list for the `select-objects` command.
- *Click* the middle mouse button to invoke the VEM main menu, and move the cursor over the arrow next to the **Selection** entry.
- Locate the `copy-objects` entry on the **Selection** sub-pane. (Note the small "x" indicating the key binding for the `copy-objects` command). *Click* the left mouse button to execute the `copy-objects` command.
- A second XOR2 gate has been instantiated in the **VEM** graphics window. This gate is still selected, and extends off the **VEM** graphics window. Press f to center the two XOR2 gates in the **VEM** graphics window. (f is the default key binding for the `show-all` command, which is on the **Display** sub-pane of the VEM menu.
- Move the cursor over the new XOR2 gate (coordinates (30,5) will work) and press the u key, which is the key binding for the command `unselect-objects`. (You may have noticed earlier that `unselect-objects` is on the **Selection** sub-pane of the VEM main menu).

The `copy-objects` command requires a list of **objects** to be copied, as well as 2 **points** indicating the displacement of the copy. VEM will copy the selected objects, and displace the copy by the amount indicated by the X–Y difference between the points. In this case, the points were 21λ apart in the X direction and 0λ apart in the Y direction.

2.4.5 Pan and Zoom

There is not enough room at the bottom of the **VEM** graphics window to add the 3 NAND gates as shown in figure 2.23. We can use the `pan` command to move the **VEM** graphics window down so we have some space to work.

- Move the cursor to coordinates (0,-10) in the **VEM** graphics window and press [p] to `pan` those coordinates to the center of the **VEM** graphics window.
- There still isn't enough space for 3 gates, so we can `zoom-out` to create still more space to work. *Click* the middle mouse button to invoke the VEM main menu, and move the cursor to the **Display** sub-pane. Move the cursor to the `zoom-out` command (note the key binding is [Z]), and *click* the left mouse button.
- With the cursor in the **VEM** graphics window, place three points at coordinates (0,-15), (0,-25) and (0,-35). (You will probably need to use the coordinate field in the title bar to position the cursor.)
- Move the cursor to the gate palette window, and position it over the NAND2 gate.
- Press [c] to `create` 3 instances of the NAND2 gate.
- Press the [f] key to execute the `show-all` command.

It is not really critical to get the 3 gates aligned perfectly when you instantiate them, since you can always use the `move-objects` command (bound to the [m] key) to reposition them later.

- Create a point in the **VEM** graphics window at coordinates (20,-25).
- Instantiate a 3-input NAND gate in your schematic by moving the cursor over the NAND3 symbol in the gate-palette and pressing the [c] key.
- In the **VEM** graphics window, draw a line beginning at coordinates (25,-10) and ending at (25,-20). (*Click* the left mouse button, then *press* the right mouse button and move the cursor downward.) The argument size box will read [0,-10], indicating the line is 10λ long.
- Press [s] to select the NAND3 gate. When using a line as an argument to the `select-objects` command, any object the line intersects will be selected.
- With the cursor in the **VEM** graphics window, *press* the right mouse button and *hold* it down.
- Move the cursor 5λ to the right (slowly). Note the selection box tracks the cursor movment. When you have moved the selection box 5λ to the right (the argument size entry in the title bar will be [5,0]), *release* the right mouse button.
- Press [m] to execute the `move-objects` command and compete the move.
- With the cursor over the NAND3 gate (which is still selected) press the [u] key to unselect the object.
- Move the cursor to coordinates (5,-5), and draw a $35\lambda \times 15\lambda$ box. The opposite corner of the box will be at (40,10).
- Once again, press [s] to select the XOR2 gate. When a box argument is used with the `select-objects` command, all objects completely enclosed by the box are selected. The first XOR2 gate was intersected by the box, but not completely enclosed.
- *Press* the right mouse button, and move the cursor up 2λ. *Release* the right mouse button, and press [m] to complete the move.
- Press [^u] to be sure everything is unselected.

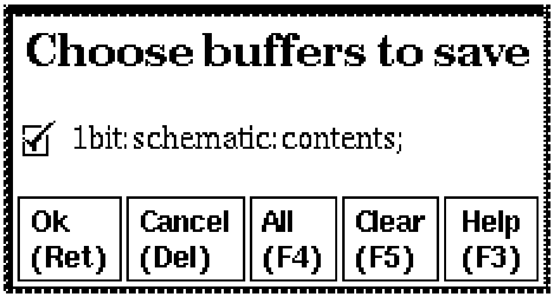

Figure 2.24: VEM dialog box for the save-all command

2.4.6 Saving Your Work

It is generally a good idea to save you work periodically. There are 2 command in VEM for saving you work. These are `save-window` and `save-all`. Both commands are bound to the same key, $\boxed{S}$. This is an example of the context sensitive nature of VEM.

- Move the cursor to the **VEM** graphics window, and press the $\boxed{S}$ key. (Remember, VEM is case sensitive.)
- You will see the command `:save-window` echoed in the **VEM** console window.
- Move the cursor to the **VEM** console window, and press the $\boxed{S}$ key again.
- Notice the command `save-all` is echoed in the **VEM** console window.

Since there are three open graphics windows (`1bit:schematic`, `symbol-palette:schematic` and `gates-palette:schematic`), VEM will open a dialog box asking you which buffers you would like to save. Figure 2.24 shows what this dialog box will look like.

- Move the cursor over the box next to `1bit:schematic:contents;`, and *click* the left mouse button. A check ($\surd$) will appear in the box.
- Move the cursor over the box labeled "OK", and *click* the left mouse button again to confirm the `save-all` operation.

2.4.7 Adding Wiring

We will now connect the gates to form the 1 bit full adder. Note that when creating wires on the schematic, the start and end points will snap to the terminal on the logic symbol if you are in close proximity to the terminal. The definition of "close" defaults to 10 pixels, but can be changed. To change this value, see the manual *Customizing VEM* by David Harrison.

- In the **VEM** graphics window, place the cursor over the output of the first XOR2 gate and draw a line to the bottom input of the second XOR2 gate.
- Press $\boxed{c}$ to `create` a wire where the line was drawn. The 2 XOR2 gates are connected.

There is an important distinction in VEM between lines and wires. The line you draw is just a line, which could be used to (un)select, create, or delete objects. When you invoke the `create` command, you are then letting VEM know you want to create *wires* where the *lines* were drawn.

- Move the cursor to the output of the top NAND2 gate (coordinates (15,-10)).

⋆ Draw a line from the output of the NAND2 gate extending 5λ to the right.

- ⋆ With the cursor over the end of the line you just completed (coordinates (20,-10)), *press* the left mouse button and move the cursor down 5λ to coordinates (20,-15) and *release* the left mouse button. This will bend the line.
- ⋆ With the cursor over the end of the new line segment (20,-15), *press* the left mouse button again and move the cursor toward the uppermost terminal on the NAND3 gate. Notice how the last vertex you created in the line moves with the cursor.
- When the cursor is over the NAND3 input (coordinates (24,-17)), *release* the left mouse button.
- The command line in the **VEM** console windowwill read:

  vem> lines(3)
- ⋆ Move the cursor to the midde input of the NAND3 gate, and draw a line to the output of the middle NAND2 gate. (This should be a straight line.)
- ⋆ Move the cursor down to the output of the bottom NAND2 gate, and draw a 3 part line (as above) connecting the NAND2 gate with the bottom input of the NAND3 gate.
- The command line in the **VEM** console windowwill read:

  vem> lines(3) lines(1) lines(3)
- ⋆ Press the [c] key to `create` all 3 wires connecting the NAND2 gates with the NAND3 gate.

We can complete the wiring of the 1 bit adder fairly quickly using multiple line arguments to the `create` command. Use figure 2.23 as a reference.

- Move the cursor to coordinates (0,0) and press [p] to `pan` the **VEM** graphics window up and to the left.
- Move the cursor over the upper input to the second XOR2 gate (20,9), and draw a line 5λ to the left.
- Continue the line with a segment extending 6λ upward, and another segment extending 35λ to the left.
- Press [c] to `create` the wire.
- ⋆ From each of the inputs of the original XOR2 gate, (-1,7) and (-1,3) draw a line extending 19λ to the left. Press the [c] key to `create` the 2 wires.
- From anywhere in the **VEM** graphics window, press the [f] key.
- From the upper input of the topmost NAND2 gate, draw a line 4λ to the left and 15λ up. The line should end on the wire connected to the top input of the XOR2 gate.
- From the bottom input of the topmost NAND2 gate, draw a line 9λ to the left and 15λ up.
- From the top input of the middle NAND2 gate, draw a line extending 4λ to the left and 10λ up. This should end at the vertex of the line connected to the top input of the upper NAND2 gate.
- From the other input of the middle NAND2 gate, draw a line 14λ to the left and 37λ up. This will terminate on the wire connected to the second XOR2 gate.
- From the upper input of the last NAND2 gate, draw a line 9λ to the left and 16λ up.
- (One more.) From the bottom input of the bottom NAND2 gate, draw a line 14λ to the left and 10λ up.

- Press $\boxed{\text{c}}$ to `create` all of the wires.

We'll need to connect some wires to the outputs. Since VEM is a multiple window application, we can open some additional windows on the schematic.

- Press the $\boxed{\text{f}}$ key to center your schematic in the **VEM** graphics window.
- Move the cursor to coordinates (40,0) and press $\boxed{\text{p}}$ to pan the window to the right.
- Move the cursor to coordinates (48,5) and draw a $4\lambda \times 5\lambda$ box.
- Press the $\boxed{\text{o}}$ key (`open-window`). VEM will open a second graphics window, "zoomed" to the box you drew in the orginal graphics window. (Depending on which window manager you are using, you may need to position the window outline and *click* the left mouse button to create the new window.) Move the new **VEM** graphics window so it is not covering the right hand side of the original **VEM** graphics window.
- Move the cursor to the unconnected output of the XOR2 in the original **VEM** graphics window.

⋆⋆ Draw a line beginning in the original **VEM** graphics window, and ending at coordinates (50,7) in the second **VEM** graphics window. Note that the line appears in both **VEM** graphics windows.

⋆ Press the $\boxed{\text{c}}$ key to create the wire.

- In the original **VEM** graphics window, draw a line from the output of the NAND3 gate extending 10λ to the right, and press $\boxed{\text{c}}$ to `create` the wire. (The endpoint will be at (50,-20).)
- When you are done with the wiring, be sure to save your work be pressing the $\boxed{\text{S}}$ key.

2.4.8 Labeling the Terminals

A few of the wires have one end unconnected. These will become the inputs and outputs of the circuit. We can label these terminals with the `create` command.

In the second **VEM** graphics window, you will see a small box at the end of the wire you created. This is called an *actual* terminal in VEM. When we label it, we will *promote* this terminal to a *formal* terminal.

⋆⋆ Move the cursor over the box at the end of the wire in the second **VEM** graphics window. *Do not create a point.*

- Type:

  `"sum"`

  and press $\boxed{\text{c}}$ to `create` the formal terminal and name it `sum`.
- In the original **VEM** graphics window, draw a $4\lambda \times 4\lambda$ box beginning at (48,-22) and ending at (52,-18).
- Move the cursor to the second **VEM** graphics window and press $\boxed{\text{z}}$ to zoom this window to the box. Note how the second **VEM** graphics window "zooms" to the coordinates and size of the box you drew in the first **VEM** graphics window.
- Move the cursor over the actual terminal (the small box) at the end of the wire, and type:

  `"cout"`

  and press the $\boxed{\text{c}}$ key.

⋆ Close the second **VEM** graphics window by pressing $\boxed{\hat{}\,\mathrm{d}}$ with the cursor in the second **VEM** graphics window.

It is not really necessary to zoom in on the end of the wire to label it, but the first few times it is useful to see the label appear. In this tutorial we won't zoom in on the rest of the signals as we label them, but if you prefer to zoom in, feel free to do so.

- In the **VEM** graphics window, move the cursor to (-20,15). This is over the topmost input signal. Type:

  `"cin"`

  and press $\boxed{\mathrm{c}}$. (Refer to figure 2.23 for the proper signal names.)
- Move the cursor to (-20,7), and type:

  `"a"`

  and press $\boxed{\mathrm{c}}$.
- Move the cursor to (-20,3), and type:

  `"b"`

  and press $\boxed{\mathrm{c}}$.
- Save the 1 bit adder by pressing $\boxed{\mathrm{S}}$.
- The next step is to simulate the circuit. We don't have to close VEM to do this, but it will make the screen less cluttered. Move the cursor to the **VEM** console window and press $\boxed{\hat{}\,\mathrm{d}}$ to close the **VEM** console window and all of the **VEM** graphics windows.

Congratulations on the completion of the 1 bit adder cell!

2.4.9 Simulating the cell

Before we use the 1 bit adder as part of a larger circuit, we will simulate it (with **musa**) to verify that it is functioning correctly. **Musa** can read commands from a file, but for this simple circuit it is not necessary.

2.4.10 Running the Simulation

Most **musa** commands can be abbreviated. For example, the state of the circuit can be displayed by typing

```
show signal_list
```

or

```
sh signal_list
```

A sample simulation run is shown at the end of this subsection.

- Invoke **musa** with the 1 bit adder as an input by typing (*from a terminal window*): `musa 1bit:schematic`
- Create a group of signals to list to the screen by typing:

  ```
  watch pins cin a b cout sum
  ```

This groups the signals `cin`, `a`, `b`, `cout` and `sum`, and names the group *pins*.

- Set each of the inputs low by typing:

```
set a 0
se b 0
se cin 0
```

- So far, all you have done is set the inputs to logic 0. the `evaluate` command actually runs the simulation. Type:

```
evaluate
```

to cause **musa** to simulate the logic with the given inputs.

- We can examine the state of the signals with the `show` command. Type:

```
show pins
```

to cause the current state of the pins to be displayed.

Outputs `cout` and `sum` should both be 0. If not, go back to VEM and check your wiring. Note that power and ground are not needed here since **musa** gets the equations for each cell from the **LOGICFUNCTION** property of that cell.

Note that **musa** uses the letters "O" and "I" to indicate signals that are set to 0 (low) or 1 (high) by the user. This distinguishes these signals in the output list from signals that evaluate to a 0 or 1, since these signals use the numbers "0" and "1" to indicate low or high values.

- Set the `a` input to logic 1 with the command:

```
se a 1
```

- Simulate the circuit by typing:

```
ev
sh pins
```

The abbreviation for `evaluate` is `ev`.

- Try a few combinations of inputs on your own until you are convinced the schematic is correct.
- When you are convinced, type:

```
quit
```

to exit **musa**.

A sample simulation of the 1 bit adder is shown below.

```
musa> watch pins cin a b cout sum
musa> set a 0
musa> se b 0
musa> se cin 0
musa> evaluate
musa> show pins
 cin:O a:O b:O cout:0 sum:0
musa> se a 1
```

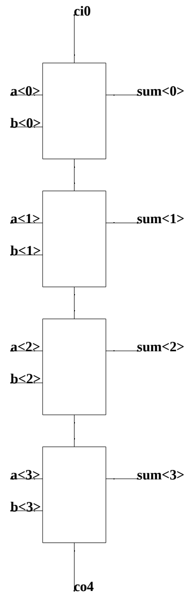

Figure 2.25: 4 bit adder schematic

```
musa> ev
musa> show pins
 cin:0 a:I b:0 cout:0 sum:1
musa> se b 1
musa> ev
musa> show pins
 cin:0 a:I b:I cout:1 sum:0
musa> se a 0
musa> se cin 1
musa> show pins
 cin:I a:0 b:I cout:1 sum:0
musa> quit
```

2.4.11 Using the cells hierarchically

We will now use 4 instances of the 1 bit adder to create a 4 bit binary adder. For reference, a schematic of the 4 bit adder is shown in figure 2.25 on page 104.

2.4.12 Creating a Symbol

Before we use the 1 bit adder, we will create a symbol for the circuit, and save it in the interface facet.

- Using a text editor, create a file `1bit.sym` with the following contents:

```
name = "1bit"
left = "a b"
right = sum
top = cin
bottom = cout
```

- Now create the symbol with the command (*from the terminal window*):

```
symgen -f 1bit.sym 1bit:schematic:interface
```

2.4.13 Adding Properties

The 1 bit adder cell should have **CELLCLASS** and **CELLTYPE** properties. We can add these property with **octprop**. (**Bdnet** can also be used to add properties.)

- Add the properties **CELLCLASS** with a value **MODULE** and **CELLTYPE** with a value of **MODULE** with the command:

```
octprop -pCELLCLASS=MODULE -pCELLTYPE=MODULE 1bit:schematic:interface
```

2.4.14 Instantiating the 1 Bit Adder

We can now create a new schematic cell, `4bit:schematic`, which will contain 4 instances of the 1 bit adder cell, `1bit:schematic`.

- Start VEM again with the command:

```
vem -G+0-0 -F 4bit:schematic -G+100+100 &
```

- Since this is a new cell, VEM will open a missing propery dialog box for the cell `4bit:schematic`. The dialog box will be similar to figure 2.3 on page 53.

⋆ Place a point at coordinats (10,10) the **VEM** graphics window and create an instance of the 1 bit adder by typing:

```
"1bit:schematic"
```

and pressing c to `create` an instance of the 1 bit adder. VEM will use the symbol you created with symgen, rather than the entire schematic drawing.

- Press f to zoom the window to the drawing.
- Press Z to zoom out a little.
- Select the instance of the 1 bit adder by moving the cursor over the symbol and pressing s.
- Place a point at the upper left hand corner of the selection box, coordinates (6,20).
- Place another point at coordinates (6,4).
- Press x to copy the symbol, followed by f to bring both symbols into the **VEM** graphics window. Note that the copy of the symbol is selected.
- Move the cursor over the original symbol (the one which is not selected) and press s to select it. You will now have 2 selected adder symbols in the **VEM** graphics window.
- Place a point at coordinates (6,21), and a second point at coordinates (6,-11).
- Press x to copy both symbols, and f to bring all 4 symbols into the **VEM** graphics window.
- Unselect everything by pressing ^u.
- Save your work by pressing S.

We will need to connect the `cout` signals to the `cin` signals. The simplest way to do this is with a second **VEM** graphics window, zoomed in to the `cout` and `cin` signals.

- Move the cursor to coordinates (9,2) in the **VEM** graphics window, and draw a $4\lambda \times 7\lambda$ box.

- Press $\boxed{\text{o}}$ to open a new **VEM** graphics window, and position it so that the four 1 bit adder symbols are not covered. The new **VEM** graphics window will be "zoomed-in" to the `cout` and `cin` terminals of the first and second 1 bit adder symbols.
- In the second **VEM** graphics window, draw a line from the `cout` signal at coordinates (11,6) to the `cin` signal at coordinates (11,4).
- Press $\boxed{\text{c}}$ to create the wire.
- In the original **VEM** graphics window, move the cursor to a point between the `cout` and `cin` signals on the second and third 1 bit adder symbols. Coordinates (11,-11) will work well. *Click* the left mouse button to create a point.
- Move the cursor to the second **VEM** graphics window and press $\boxed{\text{p}}$ to `pan` the window to the point.
- As before, draw a line from `cout` to `cin`, and press $\boxed{\text{c}}$ to create the wire.
- Place a point at coordinates (11,-27) in the original **VEM** graphics window. The point will be between the 3rd and 4th 1 bit adder symbols. Once again, move the cursor to the second **VEM** graphics window and press $\boxed{\text{p}}$ to pan the window to the point.
- One last time, draw a line from `cout` to `cin`, and press $\boxed{\text{c}}$ to create the wire.

We can label the signals on the 4 bit adder in the same way we labeled the signals in the 1 bit adder. As before, it is not necessary to zoom in on a terminal to label it, but you may do so if you prefer.

- In the original **VEM** graphics window, move the cursor over the end of the `cin` terminal of the uppermost `1bit` symbol (11,20) and create a point.
- Move the cursor to the second **VEM** graphics window and press $\boxed{\text{p}}$, then $\boxed{\text{Z}}$ to pan the second **VEM** graphics window to the `cin` terminal and zoom out a bit.
- In the second **VEM** graphics window, create a wire extending 5λ upwards. (Draw a line and press $\boxed{\text{c}}$.)
- Move the cursor over the upper end of the new wire (11,25) and name this terminal `ci0` by typing

  ```
  "ci0"
  ```

  and pressing the $\boxed{\text{c}}$ key.
- When the dialog box appears, select "SIGNAL" and "INPUT" for the terminal type and direction.
- In the original **VEM** graphics window, move the cursor over the end of the `cout` terminal of the bottom `1bit` symbol (11,-42) and create a point.
- Move the cursor to the second **VEM** graphics window and press $\boxed{\text{p}}$.
- Create a wire from the `cout` terminal (11,-42) extending 5λ downwards.
- Label the bottom end of this wire `co4` by moving the cursor over the bottom end of the wire and typing

  ```
  "co4"
  ```

and pressing $\boxed{\text{c}}$.

- When the dialog box appears, select “SIGNAL” and “OUTPUT” for the terminal type and direction.

Next, we add short wire stubs to the adder inputs, and label the signals using a vector notation. The `a` and `b` inputs will be labeled `a<0>` through `a<3>` and `b<0>` through `b<3>`, respectively. In the following steps, the signal type and direction should all be “SIGNAL” and “INPUT”.

- Move the cursor to a point midway between the `a` and `b` inputs of the uppermost 1BIT symbol (coordinates (6,13) will be fine) and *click* the left mouse button to create a point.
- Move the cursor to the second **VEM** graphics window and press $\boxed{\text{p}}$.
- Create wires beginning at the `a` and `b` terminals of the symbol and extending 3λ to the left (2 wires).
- Move the cursor to left hand end of the wire attached to the the `a` input of the uppermost symbol, and type:

  ```
  "a<0>"
  ```

  and press $\boxed{\text{c}}$ to label this signal `a<0>`.
- Move the cursor to left hand end of the wire attached to the the `b` input of the uppermost symbol, and type:

  ```
  "b<0>"
  ```

  and press $\boxed{\text{c}}$ to label this signal `b<0>`.

⋆ Begining with the second symbol from the top, label the remaining `a` inputs `a<1>` through `a<3>` and the `b` inputs `b<1>` through `b<3>` by creating a point midway between the inputs of each 1BIT symbol in the original **VEM** graphics window, panning the second **VEM** graphics window to the point, creating a 3λ wire on each input, and labelling the left hand end of the wire, as you did with the `a<0>` and `b<0>` signals.

⋆⋆ In a similar fashion, label the `sum` outputs begining with `sum<0>` and ending with `sum<3>`. You can use the “point and pan” technique to pan the second **VEM** graphics window to each of the `sum` outputs. Remember to choose “SIGNAL” and “OUTPUT” for the terminal type and direction.

- Save your work, ($\boxed{\text{S}}$), and close VEM by moving to the **VEM** console window and pressing $\boxed{\hat{}\,\text{d}}$.

2.4.15 Simulate the 4 bit Adder

In a little while, we will use **musa** again to simulate the 4 bit adder. Since we will want to simulate a few different views of the adder, it is worth the time to create a simulation control file. As we proceed from schematic to layout, we expect that at each step the same simulation control file will produce the same output results.

2.4.16 Creating a Simulation Control File

The sequence of musa commands which follows should be entered into a file (`4bit.ctl` is a good name for the control file) using any text editor you like.

It will be convenient to refer to the inputs and outputs of the 4 bit adder as *vectors*. We can define these with the **musa** command `make_vector`, abbreviated `mv`.

- Enter the following commands into your control file.

```
mv a a<3:0>
mv b b<3:0>
mv s sum<3:0>
```

- Since there are a number of signals which must be set to perform an addition, it will be simpler to define a macro to perform the addition and display the output. We can do this with the `macro` command in **musa**. Enter the following lines into the file.

```
macro add
se a $1
se b $2
ev
sh ci0 a b co4 s
$end
```

  The `$end` statement ends the definition of the macro named `add`.

- Set the `ci0` signal to 0 by adding the line

```
se ci0 0
```

  to your file.

The macro definition above makes it fairly easy to add 2 numbers and evaluate the result. A few sample input lines are shown below; feel free to add more numbers to the list.

- Simulate the addition of a few numbers by entering the following lines into your command file.

```
add h0 h0
add h2 h2
add h7 h8
add h8 h8
add hf h0
    .
    .
```

  You may continue to add pairs of numbers until you are satisfied the adder is functioning correctly.

You might also try setting `cin` to `1` (`se ci0 1`) to test the carry in. With some careful thought, it should be possible to test every path within the circuit with only a few pairs of numbers.

Save the file, and you are ready to simulate the 4 bit adder.

2.4.17 Simulation from a Command File

This section deals again with **musa**, so the commands should be entered in a terminal window, not a VEM window.

- Enter the command

```
musa -i 4bit.ctl 4bit:schematic
```

  from the terminal window. This will cause **musa** to read the schematic (`4bit:schematic`) and use the file `4bit.ctl` to control the simulation. (If you named the control file something other than `4bit.ctl`, remember to use that file name.)

musa will simulate the circuit just as if you had entered the commands from within **musa**. If the circuit does not appear to be behaving properly, open a VEM graphics window with to be sure the adder is wired correctly and the signal names in the circuit agree with the signal names in your control file. If you find errors in the schematic, be sure to save the schematic before you try to re-run **musa**. (Hint: If the `sum` outputs have a value of "X" (unkown), try usign the musa command `show "%b" sum`. This will show the individual bits of the `sum` output.)

2.4.18 Preparing the Schematic for the Optimizer

Before you can optimize the logic and map it into the cell library, the circuit must be reduced to an interconnection of primitive elements which exist in the library. (We will be using the *Mississippi State University* (MSU) library for this tutorial.)

2.4.19 Creating the Interface Facet

Before we can use **octflatten**, we will need to create an interface facet. **Vulcan** is the tool we will use to create this facet.

- Create an interface facet, "4bit:schematic:interface", with the command:

  ```
  vulcan -k 4bit:schematic
  ```

- We will use the tool **octflatten** to "flatten" the hierarchy. Enter the command:

  ```
  octflatten -t LEAF -o 4bit:flat 4bit:schematic
  ```

  This will create a new view of the 4bit adder called "flat". *Do not run vulcan on the 1 bit schematic.*

2.4.20 Resimulate the Circuit

If you would like to verify that the circuit was flattened correctly, or that you didn't modify the view, you can run **musa** with `4bit:flat` as an argument. The control file `4bit.ctl` can be used again, and the results should be the same as for `4bit:schematic`.

- The command line to re-simulate the circuit now would be:

  ```
  musa -i 4bit.ctl 4bit:flat
  ```

2.4.21 Running the Optimizer, misII

The next step is to optimize the logic and map it into the MSU cells in preparation for the final layout.

- Enter the command:

  ```
  misII
  ```

  from the terminal window. After a few seconds, the prompt "`misII>`" will appear.

- Read the flattened schematic into **misII** with the command

  ```
  read_oct  4bit:flat
  ```

- *Optional.* There is a command in misII which displays the progress of the optimization graphically. (See figure 2.26 for a sample.) If you would like to "watch" misII, enter the command

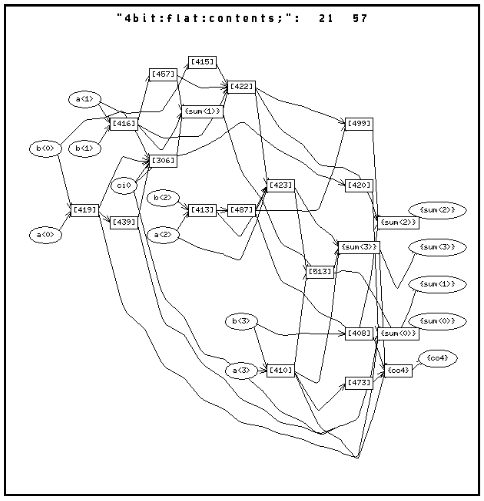

Figure 2.26: Sample misII graphics window

```
set autoexec plot
```

- Source the script for the MSU library with the command

```
source $OCTTOOLS/lib/misII/lib/script.msu
```

(This might take a few minutes to execute the entire script). If you used the plot command, you will see the graph change every few seconds as **misII** progresses. When **misII** is finished with the optimization and mapping, the graphics window will approximately resemble figure 2.26.

- When this completes (the `misII>` prompt re-appears), write a new view of the cell with the command

```
write_oct 4bit:logic
```

This view will be of viewtype SYMBOLIC, rather than the SCHEMATIC viewtype we had been working with.

- Leave **misII** by typing `quit`.

The schematic has now been optimized and mapped into the MSU cells. (**MisII** will create the interface facet, `4bit:logic:interface` as well as the `contents` facet. You should be aware of this.)

- Open a VEM window on the logic view with the command:

```
vem -G-0-0 -F4bit:logic -G+50+10 &
```

- You should see a series of standard cells "piled up". That is, the origin of each cell is in the same place.

- Close VEM by moving the cursor into the **VEM** console window and pressing ^d.

2.4.22 Simulate the Mapped Logic

Though each of the cells is placed at the same point in the logic view, the functionality and connectivity has been preserved. If you like, you can verify this by invoking **musa** with the logic view as an argument.

- From the terminal window, enter the command:

  ```
  musa -i 4bit.ctl 4bit:logic
  ```

  The results should be the same as for the previous simulations of the 4 bit adder.

2.4.23 Place and Route the Cells

An interface from OCT to the **TimberWolfSC** standard cell place and route software has been provided, and is called "**wolfe**".

From the terminal window, execute the command:

```
wolfe -o 4bit:placed 4bit:logic
```

to start the place and route software.

Wolfe will randomly assign locations for the terminals (all of the terminals you created should be in the assignment list), and pass the information to **TimberWolfSC**. This step will take a few minutes, but **TimberWolfSC** should print a number to the screen every few seconds as it runs.

2.4.24 Simulating and Viewing the Final Layout

As before, the **musa** control file `4bit.ctl` can be used to simulate the SYMBOLIC view of the cell.

- Type:

  ```
  musa -i 4bit.ctl 4bit:placed
  ```

  to simulate the placed and routed version of the logic.

- You can also use the verify function in **misII** to check the logic. To do this, type

  ```
  misII
  ```

  When the **misII** prompt appears, type the following sequence of commands:

  ```
  misII> read_oct 4bit:logic
  misII> write_blif 4bit.blif
  misII> read_oct 4bit:placed
  misII> verify 4bit.blif
  misII> quit
  ```

You may want to look at the final layout for the 4 bit adder. The layout created by **wolfe** should resemble the layout in figure 2.27.

- You can open a VEM window with the command:

  ```
  vem -G+0-0 -F 4bit:placed -G+100+100 &
  ```

- Close VEM by pressing $\boxed{\text{\^{}d}}$ when you are finished.

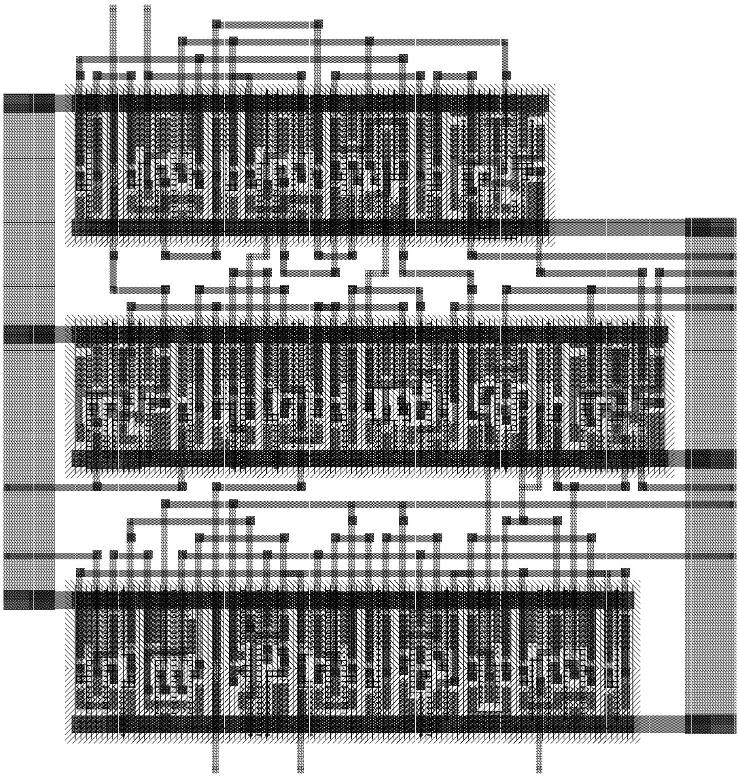

Figure 2.27: 4 bit adder layout

2.4.25 Printing the Results

If you have a PostScript printer, you can print the schematics. You will need to determine the name of the printer.

- To print the schematic for the 1 bit adder, move the cursor to the terminal window, and enter the command:

  ```
  oct2ps -t 1bit:schematic | lpr -Ppostscript_printer
  ```

  where you substitute the name of your PostScript printer in the lpr command.

- To print the schematic for the 4 bit adder, move the cursor to the terminal window, and enter the command:

  ```
  oct2ps -t -finterface 4bit:schematic | lpr -Ppostscript_printer
  ```

  where you substitute the name of your PostScript printer in the lpr command.

Chapter 3

VOV tutorial

The objective of this tutorial is to help you become familiar with VOV, a design management system developed at UC Berkeley and integrated with the Octtools. In a simple exercise, you will use some of the Octtools to layout a combinational logic circuit starting from a behavioral description of the circuit. No knowledge of the Octtools is required. You will describe the behavior of a seven segment display driver (SSDD) using the language BDS, convert the description into logic equations using **bdsyn**, optimize the logic equations with **misII**, simulate the description with **musa**, implement the layout with **padplace** and **wolfe**; all this under VOV's supervision, and with VOV's assistance.

The expected duration of the exercise is about an hour, but you can suspend it at any time and continue later.

3.1 Introduction

VOV is a design management system that provides many services: coordination of team design, history tracking, design data monitoring, data dependency analysis and others.

VOV non-intrusively monitors the activity of individuals as well as teams of designers, and maintains a record of each transaction invoked by the designers. All the design activity is captured in the **design trace**. The design trace is a bipartite directed graph, in which nodes are called either "places" or "transitions." Places represent design data, while transitions represent tool invocations, also referred to as "CAD transactions." Each transaction is characterized by a set of input places and a set of output places.

The trace can also be interpreted as a data-dependency graph. All the outputs of a transition are dependent upon the inputs of that transition. If any of the inputs changes, the transition must be repeated, that is, it must be "retraced."

A useful feature of VOV is its ability to automatically retrace (re-execute) a design sequence by using a previously generated design trace. This provides something like an automatic `make` facility, with the difference that you do not have to write any `makefile`. Instead, you simply have to execute a transaction once; after that, VOV knows when the transaction should be repeated and can repeat it automatically.

3.2 TUTORIAL: Design of a seven segment display driver

A seven segment display driver is a circuit with one 4-bit input called `data<3:0>` and 8 outputs `a,b,c,d,e,f,g,dp`. The top segment is called "a", then, if you move clockwise, you find segments "b" "c" "d" (the bottom) "e" and "f". The middle horizontal segment is "g". The decimal point is usually called "dp". The input is interpreted as an hexadecimal digit, while each of the outputs controls the corresponding segment of a seven segment display unit, so that the segment is lit if it is needed to represent the input number. The outputs are active high.

3.2.1 Start VOV

Make sure you have both `$OCTTOOLS/bin` and `$OCTTOOLS/bin/vov` in your path. If they are not, please edit your `~/.cshrc` file and modify your `path` variable.

Decide a name to describe the activity you will perform to complete this exercise. The name is important, because it identifies your work from that of other people who might be working on the same machine. The project name can be any alphanumeric string with no spaces. Examples are: `cpu badge microprocessor goophy xyz99`. In a classroom, it may be a good idea to use a name which includes your initials, or your login name. Suppose that you choose the name `ACtut`.

Move in whatever directory you plan to do your work. For example it could be `~/vovtutorial`. There you should type:

```
mkdir ~/vovtutorial
cd ~/vovtutorial
vov_ez_start ACtut
```

The **vov_ez_start** command starts what is called the "vov server," a program that runs in the background and monitors the activity of the tools you invoke. To get information from the server use the program `vov_sh`. For example,

```
vov_sh -I
```

asks the server to provide information about the status of the design. Like almost all the tools in the Octtools, **vov_sh** with no options has the effect of producing a usage message, with all the options understood by the tool. Try this, even if at this point the number of options of **vov_sh** might appear overwhelming.

Using `vov_sh -I` will also show you if the server is up and running. If nothing happens within a minute, or in the unlikely event that `vov_ez_start` fails, you should choose another name for your activity and try again.

Every time a tool is executed, whether it is invoked by you or by VOV, the tool connects to your server and declares all of its inputs and outputs. The server checks that all inputs are up to date and that the outputs can be overwritten. If that is the case, the tool proceeds with its task, otherwise a conflict has been detected and the user is queried to decide on how to resolve the conflict.

For this exercise, it is best if you work only from the shell that **vov_ez_start** gives you. If you want to open another window (even if running on another machine) and use that window as well, you can do it, provided that you set the two environment variables `VOV_HOST_NAME` and `VOV_PROJECT_NAME` to the appropriate values. This can be done easily with an alias created by `vov_ez_start` called the same as the project name.

If the server is running, let's begin our design. If you knew the **Octtools 5.2**, you could start using them as if VOV did not exist. Thus you would probably write a BDS file with the description of the SSDD and then you would invoke the various tools **bdsyn, musa, misII, wolfe** in the right order, with the proper command lines.

But, let's assume that you do not know much about the **Octtools 5.2**. You need help, and you wonder if anybody else has ever done anything similar to what you have to do.

3.2.2 Enter the assistant

The steps that transform a behavioral description of a circuit into layout have been performed many hundreds of times by many expert designers. Each time, the tools have left a trace of their execution. Some traces have been saved and installed in a library. The program `vov_assist` can be used to extract information from such existing traces.

For this exercise, it is best to use `vov_assist` in its interactive mode:

```
vov_assist -i
```

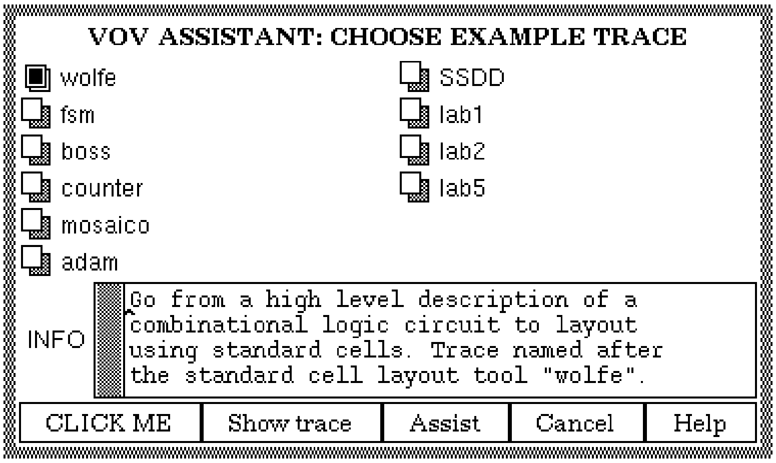

Figure 3.1: The first pop-up dialog from the assistant. The trace called fsm has been selected, and the INFO field shows a short description of the trace. The designer can take a look at the trace (Show trace) or decide to use this trace as example (Assist).

You will get a pop-up dialog, with a list of all the example traces currently present in the VOV library (Figure 3.1). The assistant is designed to be self explanatory. Make sure to click the buttons labeled `CLICK ME` and "Help" (or Ctrl-h) whenever you do not know what to do. For this exercise, here is the hint: choose the trace labeled `wolfe`.

(If the dialog does not pop-up, refer to section 3.5.)

Use of dialogs

The operation of the dialogs in VOV is mostly done with the left button of the mouse to select items in lists and to click on control buttons. The scroll-bar widget uses all three buttons, in a rather intuitive way: the left button moves the bar to the left, the right to the right, the middle allows you to drag the bar continuously.

The left button is also used to select text: you just have to click on it from two to five times. Two clicks select a word, three select a line, four a paragraph, five the entire text. One click simply moves the cursor. The selected text can be deposited into another window using the middle button.

If there are several edit fields in a dialog, you can use Tab and Meta-Tab to move from one to the other. Some control buttons have "accelerators," that is you can type a key rather than pointing and clicking. In VOV, the *Help* button is accelerated by both Ctrl-h and by the Help key. *Cancel* and *Dismiss* are accelerated by Meta-Del.

3.2.3 The graphical interface

It is interesting to see what the trace looks like. You can do this by clicking `Show trace`. This starts the editor VEM on the trace, and an associated process, called an "RPC application," that lets you browse the trace. At this point you are probably not familiar with either VEM or RPC. This is not a big problem if you are a bit adventurous and use this hint: Clicking the middle button on a VEM window pops up the VEM menu, while clicking the middle button with the "Shift" key pops up the RPC menu. The most useful commands have a "key binding" for quick invocation. The command you need is `view`, which is bound to the lower case `v`. To quit VEM use Ctrl-d in the console window.

The trace (see Figure 3.2) is a bipartite directed graph with "places" which represent data, and "transitions" which represent tools. Each transition has some inputs and outputs. Move the mouse over the trace and view some of the nodes. Notice that at the top there is a place which is a behavioral description of a counter, while at the bottom there is a place which represent the layout

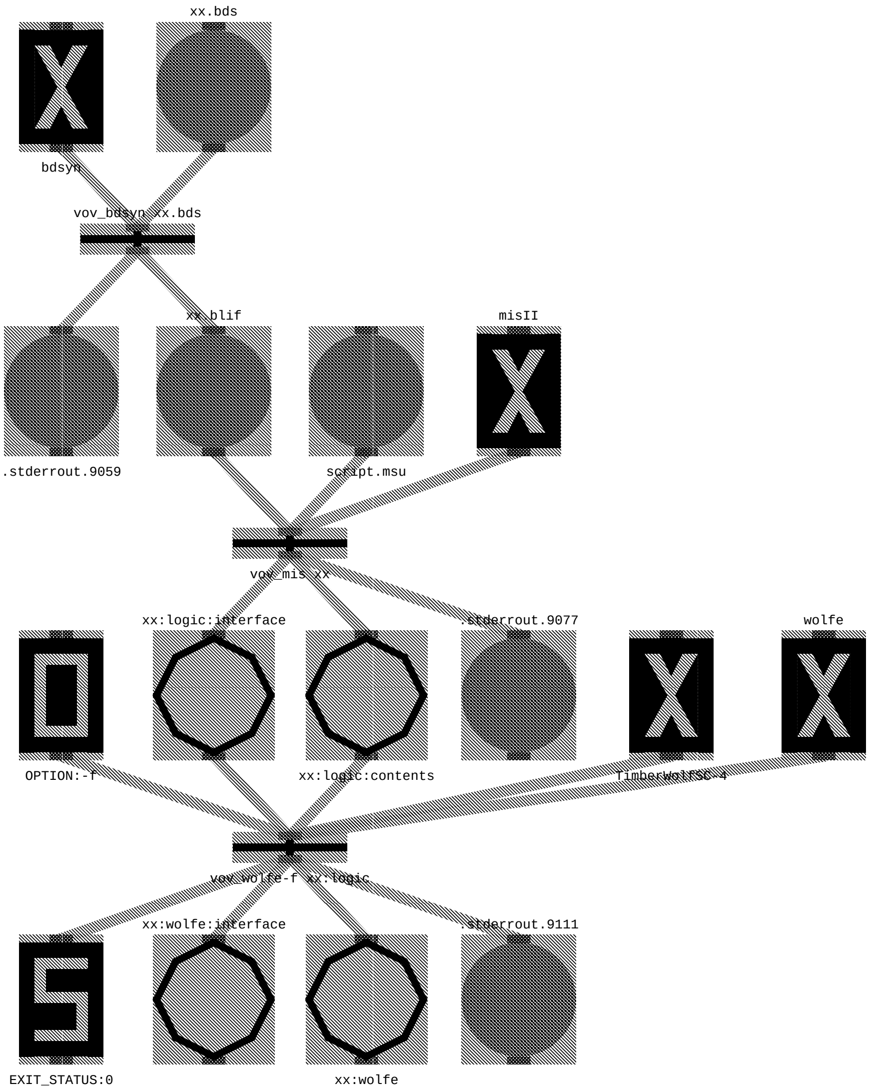

Figure 3.2: This example of a trace should be read from top to bottom, inputs are on top, outputs on the bottom. Three transactions are represented in this trace. Although all the places are treated similarly, they are represented graphically by different icons depending on the type of the place: circles represent **UNIX** data files, octagons are used for oct facets, X's denote executables, O's command line options and S's exit status of transactions.

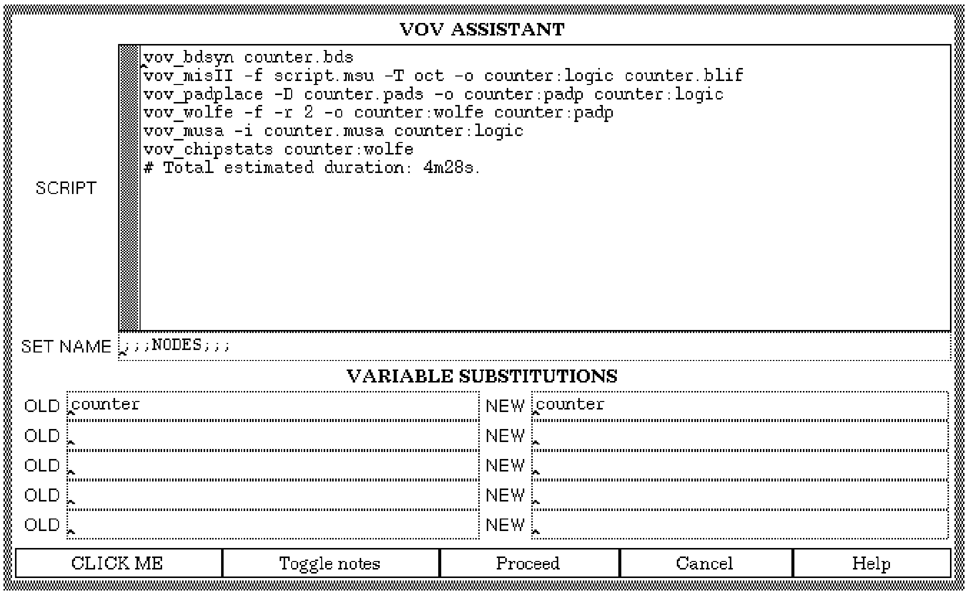

Figure 3.3: The second dialog from the assistant.

of the counter. It is the purpose of this exercise that you learn what each of the intermediate steps does.

3.2.4 Getting assistance from the assistant

Now go back to the `vov_assist` dialog and click `Assist`. You will get another dialog, with more help (Figure 3.3). Please spend some time to understand this dialog.

This dialog contains a script, consisting of all the command lines that have been used to generate this example trace. This is an alternative way to represent the trace. The trace has been fully annotated to help you understand it. To see some of the notes, click on the `Toggle notes` button. Please notice that the first transition to be executed was `bdsyn -A -o counter.blif counter.bds`. Why did the designer run this tool? What other tools where used? Why was **wolfe** used?

You will soon notice that the example trace refers to a counter. In fact, all the files mentioned in the trace are called `counter.someSuffix` and all the OCT facets are called `counter:someViewName`. The assistant uses some heuristics to determine the "topic" of an example trace. For this trace, the topic is described by the string `counter`, which appears in the dialog next to the `OLDROOT` label.

You want to design a SSDD, not another counter. Thus, it is better to rename the files involved in the trace. Instead of `counter.something`, you might want to call your files `7seg.something`. Write `7seg`, or the new name of your choice, in the `NEWROOT` box and click on `Proceed`.

You will get yet another menu, titled `VOV ASSISTANT: COPYING A TRANSITION`. This is the first transition that you have to perform to go from a BDS description to layout: `bdsyn -A -o 7seg.blif 7seg.bds` The annotations below the transition explain why you want to run this tool. Please read the annotations. Since you probably trust your assistant, you should simply read the information in the dialog, and click `Copy`. This means that the transition in the example trace is copied into your own trace, with the appropriate modifications to the command line. **bdsyn** is not actually run at this point, but now VOV knows that you want to execute this transition, sooner or later.

Now another dialog has popped up. This says `COPYING A PRIMARY INPUT`. In order to run **bdsyn** you need a file called `7seg.bds` which does not yet exists. You could very well write it from scratch, but here the assistant gives you an opportunity to get a good start. It offers you the possibility to copy the description of the counter into the file `7seg.bds`. At least, you get a syntactically correct file that should be relatively easy to modify later with a text editor like **vi** or **emacs**. Again, click on `Copy`.

Keep going: you will get about ten dialogs, in each of which you will have to click the `Copy` button. Eventually, you get a dialog that informs you that the transfer of information has been completed. Dismiss the last dialogs in `vov_assist` and go back to your shell.

For future reference

You could have obtained the same result you have just obtained by simply saying:

```
vov_assist -p wolfe -N 7seg
```

This batch use of the assistant will be useful if you want to use the example trace to process another cell. You will still use the dialogs if you want to copy only a part of an example trace.

3.2.5 Your turn to act intelligent

The assistant has done the best it could to help you, but it is not smart enough to do the design for you. You still have to make an effort to read some manuals and to understand the steps involved in your design. You won't have to read any manual to complete this tutorial, but you are encouraged to take a look at them soon, so that you can become a "power-user" of the **Octtools 5.2**.

At this point, the assistant has transferred into your design trace a modified version of a design flow which was used by some expert to build a counter. The SSDD can be built with a very similar flow. Also, the assistant has given you a good start on all the primary inputs that you need to design the SSDD. But now it is up to you to modify those files so that they contain meaningful information. Suppose that you have chosen the root name "7seg" for your files. In your working directory you will find the following files: `7seg.bds 7seg.musa 7seg.pads`. By now, you should know the purpose of each file, and you should modify them appropriately.

You should describe the behavior of the SSDD in the `7seg.bds` file. Then, you must also specify the position of the terminals in the layout, say inputs at the bottom, outputs on top, in the file `7seg.pads`. Finally, you must provide a complete simulation script to be used with `musa`, by modifying `7seg.musa`. Figures 3.4, 3.5 and 3.6 offer a very clear suggestion on how to do this. It would be nice if you could do this exercise without copying the figures[1].

3.2.6 The trace as a dependency graph

From the trace, we know that the final layout is going to be an OCT facet called `7seg:wolfe`. Even if it does not even exist yet, the server already knows how to obtain it. Try

```
vov_sh -h 7seg:wolfe
```

to get a short history of this piece of data. The use of the term "history" might sound strange at this point: how can the design have a history if we have done nothing, if no tool has ever been run? In fact, the term history is not being used properly. Instead we should say "dependency." `7seg:wolfe` depends upon all the transitions shown by `vov_sh -h 7seg:wolfe` (Figure 3.7).

Try also

```
vov_sh -h 7seg.blif
vov_sh -h 7seg.bds
```

to see the history of other places.

The trace can be interpreted as a dependency graph. By default, with the `-h` option of `vov_sh`, you get a report on the visited transitions, four levels backwards, that is looking at the nodes "above" the one you query about. Using the `-H` option, you can also ask a report on all the places visited, or on all the nodes visited (i.e., both transitions and places), with the graph traversed either backward or forward, going as many levels deep as you want. Try for example:

[1] If you are lazy, the files are in `$OCTTOOLS/lib/vov/DATA/SSDD`.

```
!! Description of seven segment display driver.
MODEL "7seg"                    !! Declare the name of this description.
   a,b,c,d,e,f,g,dp             !! Comma separated list of outputs.
        =                       !! The '=' is the separator between outputs and inputs.
   data<3:0>
        ;                       !! Terminator of I/O list.
CONSTANT
        Xzero  = 1111110#2,     Xone   = 0110000#2,
        Xtwo   = 1101101#2,     Xthree = 1111001#2,
        Xfour  = 0110011#2,     Xfive  = 1011011#2,
        Xsix   = 0011111#2,     Xseven = 1110000#2,
        Xeight = 1111111#2,     Xnine  = 1111011#2,
        XA     = 1110111#2,     XB     = 0011111#2,
        XC     = 1001110#2,     XD     = 0111101#2,
        XE     = 1001111#2,     XF     = 1000111#2;

STATE out<6:0>;                 !! Intermediate signal.
ROUTINE main_routine;           !! Only one routine in this description.
    dp = 0;                     !! Decimal point is always off.
    SELECTONE data FROM
        [0] : out =  Xzero;     [1] : out =  Xone;
        [2] : out =  Xtwo;      [3] : out =  Xthree;
        [4] : out =  Xfour;     [5] : out =  Xfive;
        [6] : out =  Xsix;      [7] : out =  Xseven;
        [8] : out =  Xeight;    [9] : out =  Xnine;
        [10] : out = XA;        [11] : out = XB;
        [12] : out = XC;        [13] : out = XD;
        [14] : out = XE;        [15] : out = XF;
    ENDSELECTONE;
    a = out<6>;     b = out<5>;     c = out<4>;     d = out<3>;
    e = out<2>;     f = out<1>;     g = out<0>;
ENDROUTINE;
ENDMODEL;
```

Figure 3.4: File 7seg.bds.

```
        TERMTYPE SIGNAL
        DIRECTION INPUT
        TERM_EDGE BOTTOM
        TERM_RELATIVE_POSITION 0.1
        TERM_RELATIVE_POSITION_STEP 0.2
FORMAL_TERMINAL data<3>
FORMAL_TERMINAL data<2>
FORMAL_TERMINAL data<1>
FORMAL_TERMINAL data<0>
        DIRECTION OUTPUT
        TERM_EDGE TOP
        TERM_RELATIVE_POSITION 0.1
        TERM_RELATIVE_POSITION_STEP 0.1
FORMAL_TERMINAL a
FORMAL_TERMINAL b
FORMAL_TERMINAL c
FORMAL_TERMINAL d
FORMAL_TERMINAL e
FORMAL_TERMINAL f
FORMAL_TERMINAL g
FORMAL_TERMINAL dp
```

Figure 3.5: The file 7seg.pads specifies the desired position of the terminals of the SSDD.

```
!! Make some vectors for convenience.
!!
mv OUT a b c d e f g
mv IN  data<3:0>

!! Macro with one parameter.
macro test
   set #c = #c + 1 % 16 !! Increment auxiliary variable mod16.
   set IN #c            !! Set the input.
   ev                   !! Evaluate circuit.
   show IN OUT          !! Show signals.
$end

!! Now try all possible inputs.
set #c = 1111          !! So we start simulation from 0
rp 16 test
```

Figure 3.6: File 7seg.musa.

```
$ vov_sh -h 7seg:wolfe

.../vovtutorial/andrea/7seg:wolfe:contents
NOT VALID  Fri Aug 24 14:56:56 1990(fornax)

NOT VALID wolfe -f -r 2 -o 7seg:wolfe 7seg:padp
NOT VALID padplace -D 7seg.pads -o 7seg:padp 7seg:logic
```

Figure 3.7: The history of `7seg:wolfe`.

```
vov_sh -H -t2 -h 7seg:wolfe     # transitions, 2 levels back
vov_sh -H -p2 -h 7seg:wolfe     # places, 2 levels back
vov_sh -H -n20 -h 7seg:wolfe    # nodes, 20 levels back
vov_sh -H -tpn400 -h 7seg:wolfe # everything, many many levels back
vov_sh -H +t3 -h 7seg.bds       # transitions, 3 levels forward
```

3.2.7 Validity of nodes

Each node in the trace, whether it is a place or a transition, has a *status*. Normally the status is either `VALID` or `NOT VALID`, but there are other possibilities described in the second part. A `VALID` status means that the node is "good." If the node is a transition, the transition has successfully completed. If the node is a place, it is up to date and consistent with all the other places it depends upon. If a transition is `NOT VALID`, it has not been run successfully yet, or it should be run again, probably because one of its inputs has been modified since the last time the transition was executed. If a place is `NOT VALID`, it is the output of a transition which is also NOT VALID.

The status of the nodes is managed by VOV. It is possible for the user to change the status of a node, but it is best that you do not do it in this exercise.

3.2.8 Automatic retracing

When all of your files are ready, you can ask VOV to run all the tools for you, that is to retrace your design. It is always a good idea to begin slowly. In this case, try retracing one step at a time. The first transition to be executed should be `bdsyn -A -o 7seg.blif 7seg.bds` which outputs `7seg.blif`. If you just want to regenerate `7seg.blif` you'll say:

```
vov_sh -r 7seg.blif
```

The server determines what needs to be done to bring `7seg.blif` up to date (in our case, it will simply run **bdsyn**).

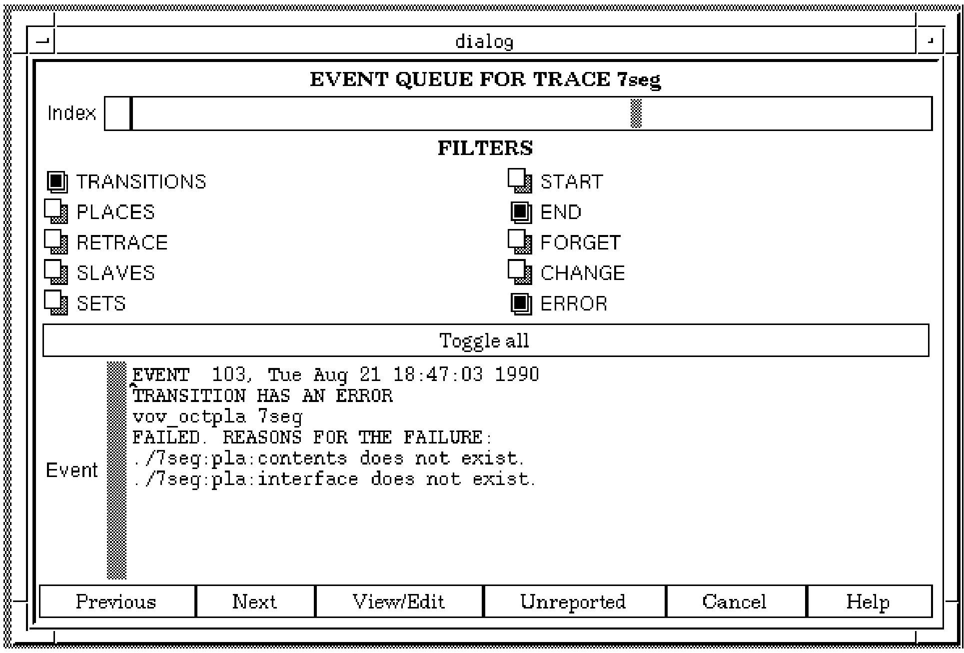

Figure 3.8: The dialog to browse the event queue.

VOV estimates the time it should take to do the retracing by adding up the estimated duration of each transition which is scheduled to be retraced, with the assumption that each transition is going to require as much time as when it was executed last. This estimation is sometimes wrong, but it tends to become accurate as the design data becomes more stable.

There might be some syntax errors in your BDS file. Read carefully the output you get from VOV. If there are errors, try to fix them, and do the retracing again.

If you are more confident in the correctness of your data, you could say, for example:

```
vov_sh -r 7seg:wolfe   # All the way down to 7seg:wolfe
vov_sh -k 7seg.bds     # Anything that depends on 7seg.bds
vov_sh -k 7seg.pads    # Anything that depends on 7seg.pads
vov_sh -R              # Retrace all in the current directory
vov_sh -AR             # Retrace all
```

Please note that some of these retracings will fail, as explained later in section 3.2.11.

It is OK to ask for a global retrace (option `-AR`) as long as you work alone. If you are part of a team, retracing everything would also retrace parts of the trace which "belong" to your teammates. Although this is not dangerous, it might result in a waste of CPU cycles.

If there are mistakes in your files, you will probably have to iterate this process a few times: modify files, retrace, modify, retrace.

3.2.9 Review what has happened

Use `vov_sh -n` to get a dialog that allows you to examine the most recent events related to the evolution of your design trace . This is especially useful to find out what, if anything, went wrong.

All the important events in the trace are recorded in a queue until a designer has inspected them. The inspection of the event queue gives you quick access to data and transitions, by a simple click of the `Edit` button. Please spend some time to experiment with this dialog (Figure 3.8), because it is very useful. Try the `Previous` and the `Next` button, and see how they interact with the "FILTER" flags. Also notice how the horizontal scroll-bar on the top of the dialog gives you fine control on the event displayed on the lower window.

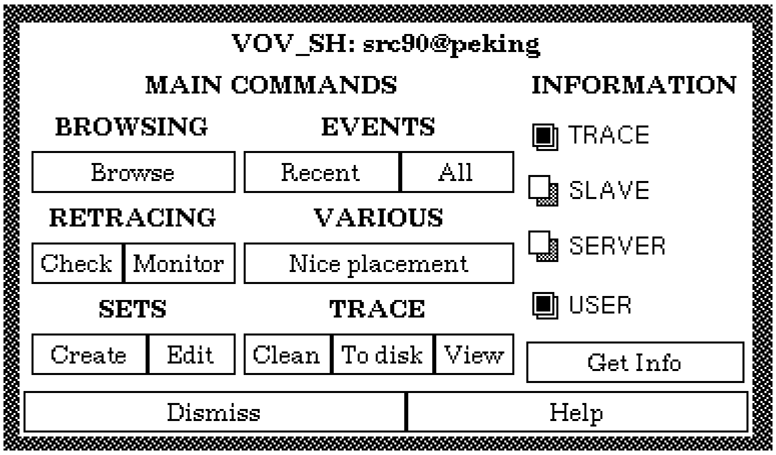

Figure 3.9: Use `vov_sh -i` to get this control panel.

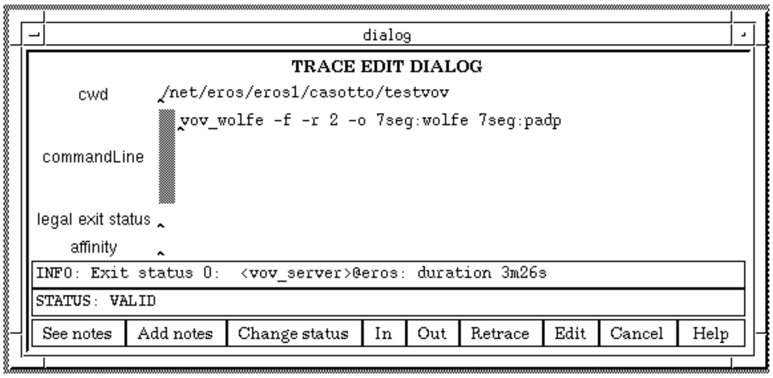

Figure 3.10: The dialog to edit a transition.

You can also interact with VOV using `vov_sh -i`, which brings up the dialog shown in Figure 3.9. You are encouraged to experiment with this dialog, because you will find it useful.

3.2.10 Possible problems

It is not possible to list all the problems that can arise during your exercise. There can be syntax errors in some of the files, which can be resolved by studying the output of the tool which has detected the error.

Sometimes VOV will not retrace a transition because one or more of its inputs in not valid. This can happen for several reasons:

- VOV has just realized that an input has become invalid. Just ask for another retrace.

- The invalid input is bogus, it is never going to become valid. This is a situation of deadlock from which VOV cannot get out without your help. You can either repeat the transition by hand (type it again from your shell), or you can decide to remove the input from the dependency list. From the event-queue dialog locate the event describing the failed retracing of the transition and click on `Edit`. You get a dialog describing the transition (Figure 3.10). Click on `In` to look at the inputs of the transition(Figure 3.11). Select the offending bogus input (one at a time) and click on `Disconnect`. This removes the place from the input list for the transition thus effectively changing the criterion for the re-execution of the transition.

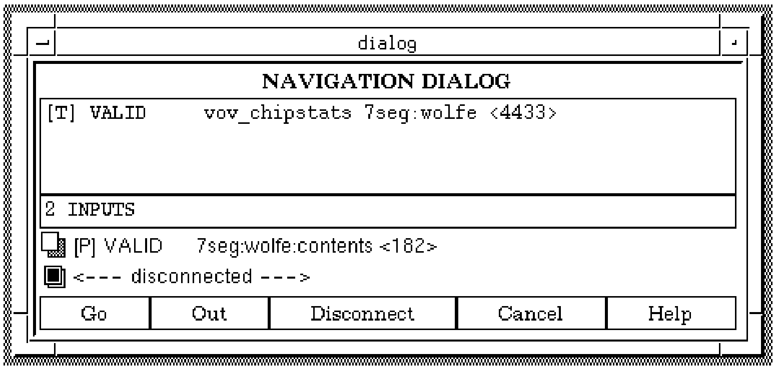

Figure 3.11: This dialog shows the inputs of a node and allows the user to navigate the trace or to disconnect nodes.

3.2.11 Substitution a transition

The example trace is not going to work with the data described in this tutorial. The reason is rather obscure for a novice: you should run `bdsyn` with the `-t` option, which will omit the trailing `<0>` for 1-bit variables — what is expected by the files `7seg.pads` and `7seg.musa`. Thus, instead of the `bdsyn -A -o 7seg.blif 7seg.bds`, as suggested to you by the assistant, you should use

```
bdsyn -t -A -o 7seg.blif 7seg.bds
```

How do you get rid of the old transition and add the new one instead? Just run the transition by hand. Since you are effectively suggesting a new way to produce the file `7seg.blif`, VOV will ask you to confirm what you are doing. This is a case of "output conflict" detected by VOV; a place that is already the output of a transition is being declared as the output of another transition, i.e. the place is being "defined" again. Since the definition of a place must be unique, either the new transition is a mistake, or the old one must be removed (forgotten): you must decide.

For such a small change in a command line, you could have also edited the transition directly, using the event queue to find an event that mentions the transition and clicking on `Edit`.

3.2.12 Check the results

When the retracing has completed successfully, somewhere you have obtained the layout of the SSDD, in an OCT facet called `7seg:wolfe`. You can use VEM to take a look at it. The easiest way is to scan the event queue until you find the event describing the change of `7seg:wolfe`, click on `Edit`, double-click on the VEM command displayed in the edit dialog, and then deposit the command in the VEM console by clicking the middle button of the mouse (easier done than said.)

In a similar way you should find the event relative to a change in the output of **chipstats** and take a look at the file. How large is the layout? How many nets are there? How many cells (instances)?

3.2.13 Try something new

Until now, you have followed a prepared trace, and it is finally time to start breaking new ground. Suppose that, by reading the Octtools manuals, you find out that **musa** offers a nice graphical way to simulate seven segment displays, and you become convinced that this graphics would be the best way to test the correctness of the SSDD.

What is needed is a way to connect together your SSDD with a model of a seven segment display. You need a new circuit with one *instance* of a driver and one *instance* of a seven segment display. The *terminals* of the SSDD should be appropriately connected to the terminals of the display by means of *nets*. The new circuit should also have some *formal terminals* to talk to the outside world. All this can be done with the BDNET language, which is very useful to write netlists.

```
MODEL 7seg:symbolic;
TECHNOLOGY scmos; !! Required property.
VIEWTYPE SYMBOLIC; !! Required property.

!! Terminals of this circuit.
INPUT  data<7:0>;  SUPPLY Vdd;  GROUND Gnd;

INSTANCE 7seg:wolfe NAME = HIDEC
data<3:0> : data<7:4>;
a : hiA;  b : hiB;
c : hiC;  d : hiD;
e : hiE;  f : hiF;
g : hiG;  dp :  UNCONNECTED;
wolfe_Vdd : Vdd; wolfe_Gnd : Gnd;

INSTANCE 7seg:wolfe NAME = LODEC
data<3:0> : data<3:0>;
a : loA;  b : loB;
c : loC;  d : loD;
e : loE;  f : loF;
g : loG;  dp : UNCONNECTED;
wolfe_Vdd : Vdd; wolfe_Gnd : Gnd;


INSTANCE "$OCTTOOLS/lib/musa/redseg":physical [50,0] NAME =  HISEG;
COMMON : Gnd;
A : hiA;  B : hiB;
C : hiC;  D : hiD;
E : hiE;  F : hiF;
G : hiG;  DP  :  Gnd;

INSTANCE "$OCTTOOLS/lib/musa/yellowseg":physical [200,0] NAME =  LOSEG;
COMMON : Gnd;
A : loA;  B : loB;
C : loC;  D : loD;
E : loE;  F : loF;
G : loG;  DP  :  Gnd ;

ENDMODEL;
```

Figure 3.12: The file 7seg.bdnet describes a netlist of two decoders each connected to a display.

In the manuals you discover the syntax for BDNET and write the file shown in Figure 3.12, which actually uses two displays, a red one and a yellow one, just for the fun of it.

Type:

```
bdnet 7seg.bdnet
```

The tool **bdnet** connects to the server and registers itself, with its inputs and outputs. From now on, VOV knows that you want to run **bdnet** on the file `7seg.bdnet`. If you change the file, you can ask for a retrace using, for example,

```
vov_sh -k 7seg.bdnet
```

When your netlist is correct, you can run `musa` on the new circuit. Of course you need a completely different script to simulate this circuit, so you write the file shown in Figure 3.13 which you call `double7seg.musa`.

Then you run the simulator:

```
musa -i double7seg.musa 7seg:symbolic
```

This transition is interactive. It does not make sense to run it if there is nobody watching the graphical output to decide if it is correct or not. If, for any reason, you ask VOV to retrace the transition, VOV will obediently do it, but you will loose all the graphical display. If you want more control on retracing of interactive transitions, please read the second part of this tutorial.

```
!! Make some vectors for convenience.
!!
mv DATA data<7:0>

set #count = 0
macro test
     set DATA #count
     ev
     sleep 200
     set #count = #count + 1 % 256
$end

!! Now try some vectors.

repeat 256 test

sleep 2000   !! Rest for a while.
```

Figure 3.13: The simulation script to test the circuit with two seven segment displays.

3.2.14 Suspension or end of the exercise

If you want to end or suspend the tutorial, it is better that you kill the server. This is optional, more than anything it is a courtesy to your colleagues that might want to use the machine. For a real design, it will be better to keep the server running continuously, until the project is completed.

```
vov_sh -AK   ## Kill the server gently.
```

The server saves the trace one last time in the directory `~/vovez` (with the name `<projectname>.page.0:symboli` and exits. You can obtain a hardcopy of your trace using the program `vov_hardcopy`:

```
vov_hardcopy <projectname> [printername]
```

You can use **vov_ez_start** again to restart the server.

3.3 A deeper look

The program `vov_ez_start`, used in the tutorial, is a convenient way to get VOV started.

3.3.1 What does vov_ez_start really do?

The steps involved in `vov_ez_start` are:

- Create, if it does not exist already, the directory `~/vovez`.
- Prepare a "slaves" file describing the slaves connected to the server. This file is called `<projectname>.vov.slaves`. If the file `$OCTTOOLS/lib/vov/vov.slaves.default` exists, the slaves file will be a copy of it, otherwise a new file will be created containing only one entry for a slave to be run on the local machine.
- Start the program `vov_server` with the options `-B -S`. The option `-B` makes the server run in batch mode, which means that the server automatically recovers in case of errors (e.g. errors in the communication protocol). The `-S` option makes the server start the slaves described in the "slaves" file.
- Create a new alias in the file `~/.vovprojects`. The name of the alias is the prpojectname. The alias can be used to set the two environment variables used by the VOV clients to locate the server in the among the machines in the network. The two environment variables are: `VOV_HOST_NAME` and `VOV_PROJECT_NAME`.

```
#   Slave list for project: BRIC
#   Lines beginning with '#' are comments.
#   Each line MUST contain:
#   HOST_NAME:  name of the host where you want the slave to run.
#   COEFF:      An integer number used to balance the relative
#               power of the slaves.
#               COEFF = 1    means: Use actual power of the slave
#               COEFF = 4    means: Divide actual power of slave by 4.
#   MAX_LOAD:   Maximum load allowed on the host in order to dispatch a
#               job to the slave. If the load on the host is greater
#               than MAX_LOAD, no job will be dispatched to the slave.
#   Optionally, you can add a list of resources provided by each slave
#   within a pair of parentheses. Example: hostname 1 4 ( vax verilog )
#   IMPORTANT: leave blanks before and after the paretheses.
#       HOST_NAME       COEFF   MAX_LOAD (AFFINITY)
        aahz            2       0.9
        chumly          2       0.9
        eros            1       8  ( bigvax vax )
        sequent         1       0.8
```

Figure 3.14: Example of a "slaves" file.

If you want to work with your server from another window (possibly in another machine), you simply have to invoke this alias and then all clients will connect to the server.

You can have more control on VOV if you perform each of these steps by hand. For example, you can decide the working directory of the server, or you can have a dozen slaves to work for you.

3.3.2 Add many slaves to your server

A slave is a special client, which simply waits to be assigned something to do. When the server has something to be retraced, it chooses the best possible slave for that job. It is convenient to have one slave running on all the machines on which you have accounts (of course, all the slaves must be able to get to the design data, possibly through NFS).

The best way to connect several slaves is to prepare a "slaves" file, such as the one shown in Figure 3.14, which should be called `<projectname>.vov.slaves` and should reside in the current working directory of the server (`~/vovez` if you use `vov_ez_start`).

All slaves are ranked by "power," a number which takes into account the speed of the CPU and the load on the machine. The coefficient `COEFF` is a politeness coefficient: it should be a positive integer. It is used as a divisor in computing the power of a slave. Thus, a slave with a `COEFF=2` has half the power of an identical slave with `COEFF=1`. If you start a slave on a machine normally used by somebody else, it is a good practice to have VOV use that slave only if all other slaves running on your machines are already busy. This behavior can be controlled by `COEFF`. In most cases, however, it is always a good idea to leave `COEFF=1`. The number in the `MAX_LOAD` field specifies that the slave should not accept jobs if the load on the machine is greater that the number. The `AFFINITY` is an optional field. Be careful if you use it: you need to put spaces between the parenthesis and anything else. See section 3.3.9 for more details.

When the file is ready, you should type:

```
vov_start_slaves <hostname> <projectname>
```

to get all the slaves started. `<hostname>` is the name of the host in which the server is running. You can also have the server start the slaves automatically by using the "-S" option in **vov_server**. The status of the slaves can be checked with `vov_sh -m`, or with the "Monitor" button in `vov_sh -i`.

You should not have more than one slave running on each machine, or you will only overload that machine. The server can have at most 61 clients connected at the same time. It is better not to connect more than about 20 slaves, or you will run out of connection points.

Slaves can be killed at any time with an ordinary `ctrl-c` or with `kill -9`.

You can also start slaves on the fly. Just log onto a machine, set the correct environment variables and say:

```
vov_slave
```

When you no longer need this slave, just kill it with `ctrl-c`.

Each slave makes available a set of resources, identified by a set of words. By default, a slave provides two resources, the name of the host on which the slave is running and the machine type. For example a slave running on the VAX called "fornax" offers the resources "vax fornax". You can override the set of resources offered by the slave with the `-A` option. A special slave which uses the `-A` option is

```
vov_interactive
```

which offers the resource "interactive", which makes the slave eligible to accept transitions whose affinity is "interactive".

3.3.3 Start the server manually

You must decide which host you want the server to run on. A good host is one that has NFS access to all files used for the project. You can run several servers on the same host, as long as they are given different project names. Typically, there is one server per project.

To restart the server use:

```
cd <server_working_directory>
vov_server -S -B <projectname} >& outfile &
```

If you are starting a new project, you have to use the `-C` option as well. Do not use this option when you RESTART the server, or you will lose the trace.

The server has been designed to run continuously for the whole duration of the project. You should not kill the server if you log out.

3.3.4 Clients

How do clients find which server to connect to? They use two environment variables: `VOV_HOST_NAME` and `VOV_PROJECT_NAME`. These variables are set with the `setenv` command:

```
setenv VOV_HOST_NAME <hostname>
setenv VOV_PROJECT_NAME <projectname>
```

It is convenient to create an alias to set those environment variables. All this is done for you by `vov_setup`, which updates or creates the file `~/.vovprojects`. For example, if you have a project called `BRIC` with the server running on `eros`, you would say:

```
vov_setup eros BRIC
```

and you would then have a new alias

```
alias BRIC 'setenv VOV_HOST_NAME eros; setenv VOV_PROJECT_NAME BRIC'
```

In this way, you just have to type `BRIC` to set the variables.

Another environment variable used by the clients is `VOV_TRACE_ONLY` which, if set to `TRUE`, tells the client that we are only interested in the trace left by the execution, and not in the execution itself. This is useful when you want to build a large trace involving many time consuming transactions. Setting `VOV_TRACE_ONLY`, you have a quick way to produce a large trace and than you can let VOV do the retracing for you.

3.3.5 The event queue and the journal

The server keeps a record of everything that happens in the trace. All events are recorder in a journal file which resides in the working directory of the server. The journal is called `journal.<projectname>`. The journal is of little practical utiility, and it is mostly used to do detective work to find out why something strange and unexpected happened.

(Note: the journal log may have been turned off in your version of VOV.)

3.3.6 The event queue

To take a look at the latest events, you can also use `vov_sh -n`, where "`-n`" should remind you of the word "notify." You should already be familiar with the dialog that allows you to browse the event queue. It offers you the quickest way to the objects in the trace. For example, if you want to change the command line of a transition, without retyping everything, you can find an event that mentions the transition, click the "Edit" button, and do the changes you want to do in the transition editing dialog that has popped up.

3.3.7 The trace

The trace is stored in an OCT symbolic facet, called `BRIC.page.0:symbolic`. It might be interesting to look at the trace with `vem`. The most current version of the facet is always in the memory of the server, which saves the trace occasionally (about once every hour). If you want to see the latest version of the trace you can force the server to save the trace onto disk with `vov_sh -w`. Also you can use **vov_rpc** to browse through the trace.

vov_sh is the main tool to get information from the server. In particular:

```
vov_sh -I       : Get information on what is going on.
vov_sh -h place : Gives you the history of a place (file or facet).
vov_sh -w       : Force the server to write the trace to disk.
vov_sh -t place : Toggle the status of the place.
vov_sh -AR      : Retrace everything that has to be retraced.
vov_sh -R       : Retrace everything under the current working directory.
vov_sh -r place : Retrace what is needed to make ''place'' consistent.
vov_sh -k place : Kick ''place'', i.e. retrace all its dependents.
vov_sh -n       : Be notified about the latest events.
vov_sh -i       : Start the interactive interface for vov_sh.
vov_sh -Ai      : Start the advanced interactive interface for vov_sh.
vov_sh -AK      : Kill the server gently.
```

You will most frequently use the interactive interface of **vov_sh**.

3.3.8 Annotations

Adding annotations to the trace is like adding comments to a program. Everyone agrees that it is important, but no one really does it because it is a hassle. In case you are a well disciplined designer and want to carefully document what you are doing, for your own benefit of for the benefit of your teammates, VOV gives you the possibility to add **annotations** to any element in the design trace. An annotation is a piece of text attached to the object. Use annotations to explain why you did something, or the meaning of a place. Whenever you are editing or viewing an object, please notice in the lower left corner of the dialog the buttons labeled "See notes" and "Add notes". The "See notes" button appears only if the object already has annotations attached to it, the other button is always there. If you click either button you get a dialog to browse, add or forget the annotations for the object.

3.3.9 Affinity of transition, interactive transitions

In VOV it is possible to connect to the server several slaves, each running on a different machine in the LAN. The slaves provide VOV with CPU cycles to perform the necessary retracing. Two slaves are **equivalent** if any transition performed on one slave would give the same results if executed on the other slave.

It would be ideal if all slaves were equivalent: the dispatching of transitions could be based only on the greedy strategy that the best slave gets the transition with longest expected duration. In practice, different machines offer different resources and slaves can lose equivalence for several reasons:

architecture: the output of a compilation depends on the architecture of the machine;

hardware resources: some transaction could be very time consuming, or require a large amount of memory, so that it should only be executed on the large machines in the network, rather than on the smaller workstations;

software resources: some commercial software such as "verilog" can run only on the machines which have been licensed.

One unsatifactory solution is to limit VOV to the largest subset of equivalent slaves. This might not work because in general, a subset of equivalent slave does not completely cover the set of resources needed to complete a design, or the subset could be too small for the design.

VOV deals with the problem of non-equivalent slaves by considering the **affinity** of each transition. If a transition, for licensing reasons, can only run on the machines called "pinocchio" and "geppetto," its affinity is "pinocchio geppetto". If a transition can only run on vax'es, its affinity would be "vax". For the special case of a transition that can run on any machine, the affinity would be an empty string. By default, the affinity list of a transition contains only the machine type.

In a similar fashion, each slave has an associated list of resources, which defaults to a list containing the machine name and the machine type. Thus, the resource list of a slave running on the VAX "pinocchio" would be "pinocchio vax".

Both the affinity list of a transition and the resorce list of a slave can be overridden by the user.

A *match* between an affinity list and a resource list exists if either string is empty or if the two lists have one word in common. When dispatching a transition to one of the slaves, VOV scans the slaves in decreasing order of power, and chooses the first idle slave whose resource list matches the affinity list of the transition.

3.3.10 Graphical interface using vem/RPC

An RPC application is available to browse the trace. Move to your project directory before you type: `vov_rpc BRIC` This starts **vem** and an RPC application called **vovRpc**.

The color coding used has the following meaning:

```
pink (MET2)     : valid trace
blue (MET1)     : invalid trace
red  (POLY)     : active tracing (a transaction is currently executing)
orange (NWEL)   : active retracing.
```

The places in the trace use different icons to ease the understanding of the graph. Here is the key to interpret the icons:

```
circle  : Data file (often an ASCII file)
octagon : OCT facet
X       : Executable
S       : Exist status
O       : Command line option
```

3.3.11 Status of the trace

Each node in the trace has a "status," which can be one of the following:

VALID: The node is "good." If the node is a transition, the transition has successfully completed. If the node is a place, it is up to date and consistent with all the other places it depends upon.

NOT VALID: If the node is a transition, it has not been run successfully yet, or it should be run again, probably because one of its inputs has been modified since the last time the transition was executed. If the node is a place, it is the output of a transition which is also NOT VALID.

TRACING: If the node is a transition, it is currently being executed for the first time. All the outputs of such transition are also in the same status.

RETRACING: If the node is a transition, it is currently being retraced. All the outputs of such transition are also in the same status.

DEAD: The dead parts of the trace are ignored by the server. A node can become DEAD only if a designer decides so. Dead nodes are kept for documentation purposes, or to prevent the server from retracing them.

MISSING: This status is used for places which suddenly disappear.

WEIRD: You might occasionally see this as a possible status, but in reality it is not used, and it will soon disappear.

3.3.12 Protection

Currently VOV is an open system, which ignores protections. All the data generated by VOV is not protected. Although this is only a temporary situation, it should not be a problem.

3.3.13 Sets

You can create sets of nodes. You can do this from the `vov_sh -i` menu. When you create a set you must specify a name for the set, and a **selection rule**. The selection rule is complicated: you can select nodes by type (PLACE or TRANSITION), by status (VALID or NOT VALID or DEAD or any of the others), by type of place (UNIX FILE, OCT FACET, EXECUTABLE, etc.) or by regular expression matching. You can also select nodes with no inputs, with no outputs, or isolated nodes, those with no inputs and no outputs.

The regular expression matching routines use the "emacs" syntax. The dot `.` stands for any character, the star `*` stands for an arbitrary repetition of the previous expression. Thus the regular expression `.*` matches every string (any character repeated any number of times). If you want to make a set of all the files whose name begin with `.std.` you should specify the regular expression `.*\.std\..*` in which `\.` matches the dot. For a more complete description of the syntax, ask emacs.

Why should you want to make sets? A number of reasons, the most important being documentation and because you can use sets to delete (forget) nodes from the trace.

3.3.14 Forgetting nodes

VOV makes it purposely hard to eliminate nodes from the trace. The simplest way to tell VOV to ignore parts of the trace is to change the status of the uninteresting nodes to DEAD. Do this with the "Change status" button you can see in many dialogs. But if you are positively sure that you want VOV to forget about something, here is what you do. First you must start `vov_sh -i` to get the menu that allows to create and edit sets. Then you create a set containing the nodes you want to forget. Then you edit the set and ask for a list of its elements. Then you select the nodes in the set that you want to forget and hit the "Forget" button.

Sometimes, before you forget a place, you might want to delete it from disk and throw it in the trash. Thus you can select some nodes and hit "Trash". Actually VOV does not remove the files form disk, it just puts them in another directory called `VOV_GARBAGE_CAN` in the working directory of the server.

Responding to complaints that the procedure just described is too involved, another way to forget a node has been impemented: use the option -f in **vov_sh**. Use this with caution:

```
% vov_sh -Af <placename>
```

3.3.15 Moving stuff around the file system

Would you like to change the path name of a place in the trace? Use the "Mvlib" button in the `vov_sh -Ai` dialog. Alternatively you can use the `-O/-N` pair of options in **vov_sh**. Example. Suppose you have built a pla in a directory `/users/joe/mypla` and have decided to move all the data in another directory, for example by doing:

```
cd /users/joe
mv mypla reallygoodpla
```

Now VOV is surely confused. It knew about some files in the old directory and now it is unable to find them. So you now just tell VOV what happened:

```
vov_sh -O /users/joe/mypla -N /users/joe/reallygoodpla
```

3.3.16 Handy utilities

vov_cleanup : get a report on what files are useful and what can be removed.

vov_setup : create a new alias to set the VOV environment variables.

vov_start_slaves : start slaves described in the slaves file.

vov_kill_slaves : kill slaves described in the slaves file.

vov_parallel : uses the VOV dispatching mechanism to parallelize the execution of a set of indeoendent transitions. See `octman vov_parallel` for more information.

3.4 Working in a network

If you are working in a network environment, you need to devise a scheme to name your files and facets so that all the machines that you are going to use will understand the names correctly. Full path names are not adequate, because they may make sense on one machine but not in another. VOV gives you the possibility of using logical names. A logical name uses environment variables to point at the root of a directory tree which contains the files in your design.

For example, you can have an environment variable called MYPROJECT with value `/users/designer1/project` and then refer to your files and facets with names such as `$MYPROJECT/alu:symbolic`.

On another machine the value of MYPROJECT may have to be set differently, for example `/net/host1/users/designer1/project`, but the name `$MYPROJECT/alu:symbolic` can be made to mean the same on both machines.

You can tell VOV which environment variables to use in its names with the file `$HOME/.vovequivalence` which is a shell script that is going to be executed. The stdout of the script consists of any number of pairs, with each pair consisting of a name of an environment variable and of a full path name. For every occurrence of the path name, VOV will replace the string `${VARIABLE_NAME}`

If your `~/.vovequivalence` file specifies the equivalence

```
MYPROJECT /net/host2/users/designer1/project
```

then a file name `/net/host2/users/designer1/project/alu.bds` will become `${MYPROJECT}/alu.bds`.

Figure 3.15 shows an example equivalence file and Figure 3.16 shows it output.

```
#!/bin/csh -f

if ( -e /bin/machine ) set MACHINE = `/bin/machine`
if ( -e /bin/arch ) set MACHINE = `/bin/arch`

set oct = $OCTTOOLS
set OCTROOT = $oct:h

echo OCTTOOLS $OCTROOT/sun4
echo OCTTOOLS $OCTROOT/sun3
echo OCTTOOLS $OCTROOT/ibmrt
echo OCTTOOLS $OCTROOT/mips
echo OCTTOOLS $OCTROOT/vax
echo OCTTOOLS $OCTROOT/common
echo OCTTOOLS /projects/octtools/octtools/$MACHINE
echo OCTTOOLS /projects/octtools/octtools-5.0/$MACHINE

if ( $MACHINE == mips ) then
endif
if ( $MACHINE == sun4 )  then
     echo OCTTOOLS /tmp_mnt/net/canova3/octtools
     echo OCTTOOLS /tmp_mnt/net/canova2/octtools
endif
```

Figure 3.15: Example of a .vovequivalence file

```
OCTTOOLS /projects/octtools/octtools-5.0/sun4
OCTTOOLS /projects/octtools/octtools-5.0/sun3
OCTTOOLS /projects/octtools/octtools-5.0/ibmrt
OCTTOOLS /projects/octtools/octtools-5.0/mips
OCTTOOLS /projects/octtools/octtools-5.0/vax
OCTTOOLS /projects/octtools/octtools-5.0/common
OCTTOOLS /projects/octtools/octtools/mips
OCTTOOLS /projects/octtools/octtools-5.0/mips
```

Figure 3.16: Output obtained executing the script in the previous figure.

3.5 Problems with dialogs

Make sure that the DISPLAY variable is correct, that the display is in fact willing to serve requests from the machine in which you are running. Ask your X11 guru for help.

Sometimes the dialogs refuse to pop up with strange error messages. This seems to be due to strange combination of widget defaults. These problems are generally fixed by changing those defaults. Here is a list of working defaults (You will have to add it to your `~/.Xdefaults` file.

```
Dds*dialog.borderWidth: 0
Dds*dialog*borderColor: black
Dds*background: #cc9
Dds*description.borderWidth: 0
Dds*description.font: -*-helvetica-medium-r-normal-*-*-120-*-*-*-*-*-*
Dds*description.foreground: DarkSlateBlue
Dds*title.font: *-new century schoolbook-bold-r-normal-*-*-120
Dds*title.foreground: cyan
Dds*title.borderWidth: 0
Dds*button.font: *-new century schoolbook-medium-r-normal-*-*-120
Dds*button.borderWidth: 1
Dds*button.highlightThickness: 3
Dds*button.foreground: yellow
Dds*edit*font: -*-courier-medium-r-*-*-*-120-*-*-*-*-*-*
Dds*edit*foreground: white
Dds*edit*borderWidth: 1
Dds*edit*wrap: word
Dds.dialog*edit*borderColor: cyan
Dds*Table.borderWidth: 0
Dds*Table.internalWidth: 1
Dds*Table.internalHeight: 1
Dds*Table.rowSpacing: 0
Dds*Table.columnSpacing: 0
Dds*Tgl.borderWidth: 0
Dds*vport.forceBars: True
Dds*value.font: *-new century schoolbook-bold-r-normal-*-*-120-*
Dds*value.foreground: cyan
Dds*numvalue.borderWidth: 1
Dds*numvalue.font: times_bold14
Dds*numvalue.foreground: yellow
Dds*valscroll.foreground: cyan
Dds*valscroll.length: 40
Dds*choice.font: *-new century schoolbook-medium-i-normal-*-*-120-*
Dds*choice.foreground: yellow
Dds*showtext*font: -*-courier-medium-r-*-*-*-120-*-*-*-*-*-*
Dds*showtext*foreground: black
Dds*showtext.wrap:  word
```

Chapter 4

Macro-cell placement: puppy

Andrea Casotto
Dept. of Electrical Engineering
University of California, Berkeley

4.1 Getting started

puppy is a tool for placement and floorplanning of integrated circuits in the macro cells design style. The program can also be used for placement of components on a PCB. The algorithm used by **puppy** is Simulated Annealing ([4], [3], [1]).

The program originated [1] on the Sequent Balance 8000, a multiprocessor with shared memory. On the Sequent, **puppy** can use as many processors as desired, with almost linear speedup. The program can also run on sequential machines (VAX, SUN). Sequential and parallel versions are essentially identical from the user's point of view.

A friendly user interface has been developed, with graphics and menus. It is highly recommended that **puppy** is run under an X-server, even if the program can run on an alphanumerical terminal.

4.1.1 Starting puppy

puppy uses two auxiliary and optional files:

- the first file is used to set the parameters of the annealing. **puppy** looks first for a file called `<chip>.parPuppy`. If such file is missing, **puppy** searches for `parPuppy`. The file can be created by **puppy** itself, using one of the menu commands or using the `-l` option. The file is self-documenting. You can edit it before a **puppy** run. Any parameter in the file can also be set interactively.

- The second file defines constraints on the placement. The file is called either `<chip>.constraintsPuppy` or just `constraintsPuppy`. The use of this file is optional. All the constraints can be represented by OCT properties attached to instances and to nets. If such properties are already present in the chip to be placed, you do not have to use the file. This file provides a simple mechanism to enter these properties in the OCT database. In particular you can set `MOBILITY` and `LEGAL_TRANSFORMATIONS` for each instance, and `NET_CRITICALITY` and `MAX_NET_LENGTH` for each net. All of these properties are described later in this manual and also in the *OCT Symbolic Policy*document.

  This file can be created by using the `-l` option. Use `-l C` if you are only interested in constraining the placement of the cells, `-l N` if you want to care about the criticality of the nets, or `-l B` if you are going to constrain both instances and nets. Once the file is created you can edit it. The file is self-explanatory.

Suppose that the chip that you want to place is `chip:unplaced`. Type
`puppy -o chip:placed chip:unplaced`

First, **puppy** copies the input view `chip:unplaced` into the output view `chip:placed`. From now on, the input view will not be touched. Then **puppy** begins to read the `chip:placed`. If the chip is large, with many instances of many different masters and many nets, reading the view can take a couple of minutes.

If you are running under X and the environment variable `DISPLAY` is properly set, **puppy** opens the following windows on your display:

Placement window: this window shows the current placement being considered by **puppy**. The window has an active menu, which can be called up by pressing and holding any button while inside the window. Use this menu to recenter the placement area in the window, to decide whether the nets should be displayed, to see which processor owns which cell, to see the expansion of the cells.

Console window: this window shows the components of the cost function, the speed of the algorithm measured in moves per second and the number of processors being used. Other information might also be displayed, depending on the version you are actually running.

Interactive Variables window: this window allows the user to control most of the annealing parameters. This window is implemented by the `iv` package. There are active buttons in the window, to change the values of the parameters. Try pressing some buttons on this window to see what happens. You can change the value of a variable also by typing the new value in the appropriate window. Notice at the bottom the `Confirm` button. You need to push on this button to confirm some actions that are potentially dangerous.

Plot windows: there can be 3 or more plot windows, depending on the version of **puppy** that you are running. The windows are managed by the `xg` package. There is a menu in each of these windows. The menu can be activated by pressing the middle button of the mouse. The left button is used for zooming. Press the button, hold it, drag it to define the zooming area, then release. The right button lets you make measurements on the graph. Push the button and hold it. A large cross will appear in the window, and the coordinates of its center will be printed at the top of the window. Release the button to continue.

X-event processing: the parent process in **puppy** checks the event queue after each move. Some delay in response to mouse and button actions could be experienced on very loaded machines, or on large chips if the action happens while **puppy** is performing measurements and double checks after each temperature step.

4.1.2 Interaction with puppy

You can just watch **puppy** do its job or you can try to interact with it. The most useful things you can do are:

Gently abort: you can abort because you see that the placement that **puppy** is finding is clearly unsatisfying, maybe because you have not specified correctly all the constraints, or you might have any reason you want to abort. Press any button on the console window, hold it until the menu comes, and select *exit* in the *Debug* pane. Then you will have to push the *Confirm* button in the IV window.

Adjust weights: (See section on tuning for more details.);

Adjust temperature: Simulated Annealing stops when the chip is "frozen" and the parameter called "temperature" is rather small. For annealing to produce the best result **you should not play with the temperature**. However, in situations in which you are in a time crunch and want a quick solution, you will find that manually lowering the temperature will speed things up significantly. In particular, if you set the temperature to 0.0 (negative values are illegal !!) you will reduce Simulated Annealing to a more simple minded algorithm called *Iterative Improvements* which most surely will get stuck in some poor local minimum of the cost function. See also Section 4.7 for more information on the cooling schedule and on the stopping criterion.

Change the placement area: The dashed box that you can see in the placement window represents the placement area. A high penalty is applied if a cell is placed outside the box. There are 4 different ways to specify the placement area, described in section 4.5. You can change the placement area with the menu entry *changeChipBoundary*, which rotates among the 4 possible definitions. In the case of the `AUTOMATIC` boundary, you can interactively specify total area and aspect ratio using the IV-window. When you are satisfied with the placement area, using the menu, select *recenter* to recenter the placement window.

Change expansion of the cells: (See Section 4.3.2). Cells are expanded by **puppy** to accommodate some routing area. You can see the effect of the expansion by toggling the `show-BB` toggle in the `DISPLAY` pane of the menu. The cell expansion can be changed using the IV-window.

4.1.3 Exiting puppy

puppy will stop when the stopping criterion described in Section 4.7 is satisfied. If you see a satisfactory placement in the placement window, you can *save-and-exit* with the menu. **puppy** creates a log file called `PUPPYdata000` containing statistical data about the annealing. You can normally discard such file.

4.1.4 Advanced features: Floating terminals

In a limited way **puppy** can help you do some floorplanning of a chip, in the sense that it can handle cells in which you are free to choose the position of the terminals. See Section 4.6.2 for more details. See also the `MOSAICO` manual for more details on floorplanning.

4.1.5 Placement constraints

The following properties can be used to specify special constraints on the annealing variables:

MOBILITY "FIXED" means that the cell's coordinates cannot be changed by **puppy**. The default is "MOVABLE".

LEGAL_TRANSFORMATIONS is a string valued property. The string is a list of words from the following vocabulary:

- "ALL", the default value, means that all the transforms are legal;
- Any OCT transformation, without the prefix "OCT_": "NO_TRANSFORM", "MIRROR_X", "MIRROR_Y", "ROT180", "ROT90", "ROT270", "MX_ROT90", MY_ROT90";
- "THIS_ASPECT_RATIO" is a short hand for the list of 4 transformations which maintain the aspect ratio of the cell: "NO_TRANSFORM MIRROR_X MIRROR_Y ROT180";
- "NOT" means that all the following transformations are not legal.

Examples: "NOT THIS_ASPECT_RATIO", "ROT90 ROT270 NO_TRANSFORM".

TERM_FLOATING_EDGES is a string valued property attached to each floating terminal, specifying the edges in which the terminal is allowed to float. The default value is expressed by the empty string "" meaning that all the edges are legal. Example: "RIGHT LEFT".

FORBIDDEN_EDGES is a property attached to the "FLOORPLAN" bag in the interface of the master of a soft cell. It is an economic way to put edge constraints on all the floating pins of the cell. Example: "LEFT TOP".

4.2 Description of the algorithm

The input for a placement problem consists of a net list, that is a description of a set of cells and of their interconnections. The cells can be of any rectilinear shape, but rectangular cells are the most common. Each cell has a number of terminals, that are the places where the nets implementing the interconnections must be attached. The terminals can have a fixed location or they can be placed at will along some of the edges of the cell. In this second case we say that the terminals are "floating".

To describe a placement we must specify the position of each cell as well as the position of all the floating terminals. In general, a placement can be considered as a point s in a space S of all possible placements. The point $s \in S$ is specified by the values assigned to some "variables", each representing the position of a cell or of a floating terminal. In consideration of the fact that we are using Simulated Annealing to determine the optimal value of those variables, we refer to them as the "annealing variables."

The annealing variables in **puppy** are:

cells , for which a position must be selected, represented by a triplet for X and Y coordinates of the center of the cell and orientation of the cell; in future versions of the program, the shape of the cell can be chosen as well;

floating terminals , which must be placed along the perimeter of the cell they belong to, subject to some constraints;

nets , for which the shape of the minimum spanning tree must be determined.

In the parallel version, the set of all the annealing variables is partitioned among all the processes. In particular, each annealing variable is assigned an "owner," that is a process that has the responsibility of performing the annealing on that variable. Each process has permission to move only variables that it owns (or that it has temporarily borrowed). The current value of a variable is accessible, for reading, by all the processes. On the Sequent this is achieved by allocating the annealing variables in the shared segment of the memory.

Simulated Annealing attempts to perform a global minimization of a function defined over a finite space. In our case the finite space is the set S of all the placements and the function is a suitable "cost" defined by the user so that the final result has some desired properties, such as a minimum total net length, or minimum area, or a combination of the two. This characterization of the cost function is obviously faulty: it is expressed in terms of the final result which is itself influenced by the choice of the cost function, in the sense that it is one of the points of global minimum. However faulty, this characterization seems to capture what of empirical still exists in the art of using Simulated Annealing for the minimization of a complicated cost function.

4.3 The cost function

The cost function that **puppy** attempts to minimize is the weighted sum of four components;

- Estimated wire length;
- Overlap penalty;
- Chip area;
- Violation of constraints on length of some nets.

Four additional components are considered when dealing with analog circuits;

- Penalty for asymmetric placement;
- Penalty for device mismatching;
- Penalty for discontinued wells;

- Violation of performance constraints.

More precisely, let S be the set of all possible placements of the cells belonging to C_T, and define a cost function $v : S \rightarrow \mathcal{R}^+$.

The *cost function* v is the weighted sum of four components

$$v(s) = \alpha_w l f_w(s) + \alpha_a f_a(s) + \alpha_{ov} f_{ov}(s) + \alpha_v f_v(s)$$

and, for analog chips,

$$v(s) = \alpha_{wl} f_{wl}(s) + \alpha_a f_a(s) + \alpha_{ov} f_{ov}(s) + \alpha_v f_v(s) + \alpha_{sy} f_{sy}(s) + \alpha_{ma} f_{ma}(s) + \alpha_{we} f_{we}(s) + \alpha_{cd} f_{cd}(s) \tag{4.1}$$

where:

- $s \in S$;
- $f_w l$ is the *estimated total wire length* ; the length of a net is estimated by computing the semi-perimeter of the bounding box including all the pins connected to the net;
- f_a is the *total area of the chip* ;
- f_{ov} is the *total overlapping area* between cells; this component has to be zero at the end of the algorithm, to achieve a feasible placement;
- f_v is the total number of violations of constraints on the maximum length of some nets.
- f_{sy} is a measure for the asymmetry of the circuit.
- f_{ma} is a measure for the mismatch of the circuit devices.
- f_{we} is a measure for the discontinuity of well regions.
- f_{cd} is a measure for the performance degradation of the circuit.
- The coefficients α are non-negative weights, that can be set for each experiment; usually, a number of experiments help determine a satisfactory set of weights. The weights are chosen by the user, using empirical criteria. See section 4.8 on tuning.

4.3.1 Estimated wire length

puppy estimates the length of a net in either of two ways. The first method is the bounding box metric, by which the length of a net is estimated as the half-perimeter of the minimum rectangle including all the terminals belonging to the net. In the second method, each net is represented using a spanning tree of its terminals. Each edge in the tree is called a subnet. The length of a subnet is the Manhattan distance between the two terminals on the subnet, and the length of the net is estimated as the sum of the lengths of its subnets. The spanning tree of a net is not necessarily the minimum spanning tree. Instead, **puppy** performs an annealing on the shape of the tree in order to minimize its length.

See section 4.6 for a description of the move which allows the annealing of the spanning trees. The subnets are represented by blue segments in the placement window. Red segments are used for those nets which have only two terminals.

In terms of speed, the subnet method is intrinsically faster than the bounding box method. However, due to the need to perform extra moves to anneal the spanning trees, there is no clear speed advantage in using the new method.

In any case, the subnet method is recommended, because it is more accurate. The bounding box metric is provided only for comparison with other placement tools which also use such a metric.

One can switch between one method to the other using the “enable subnets” switch, either in the parameter file or in the interactive variable window.

4.3.2 Routing estimation

In **puppy** the perimeter of each cell is expanded to take in to account routing area. The expansion is depends on the pin density on each edge of the cell and on a parameter called "cellExpansion", which can be changed interactively. You can see the effect of the expansion by toggling the `show-cell-expansion` toggle in the `DISPLAY` pane of the menu. Changing the cell expansion interactively is a lot of work for **puppy**. Try to do it carefully. **puppy** can only compute overlap of rectangular cells. Generic rectilinear cells are expanded into rectangular cells.

The routing estimation performed by **puppy** is not always satisfactory. In particular, it fails in the case of abutting cells, because the expansion prevents the cells from abutting. Good results have been obtained setting "cellExpansion" to small values (1 or 2) or even to zero.

4.3.3 Overlap penalty

The final placement should have no overlap, but experimentation has shown that the finite time behavior of the Simulated Annealing algorithm for placement improves if overlapping is allowed in the intermediate steps of the algorithm.

The cost of overlap is proportional to the overlapping area.

In order to keep the cells inside some prescribed boundaries, an overlap penalty is computed, proportional to the amount of area outside the boundary. This type of overlap is more costly than ordinary overlap depending on a factor called `chipBoundaryOverlapFactor` in the `parPuppy` file.

A small amount of overlap in the final placement does not necessarily mean that the placement is illegal, because such overlap is often due to overlap of routing regions around the cells.

4.3.4 Chip area

The total area of the chip is calculated as the area of its bounding box, i.e. the smallest rectangle including all the cells. If the chip contains fixed pads, they are not considered for the purpose of computing the chip area.

4.3.5 Special constraints

This component of the cost function is experimental. It can change in future versions of the program.

Sometimes one wants a critical net to be no longer than some fixed length. This is expressed in OCT by attaching the property "NET_MAX_LENGTH" to the critical net. This is an integer valued property which represents the maximum length of the net in lambdas.

puppy interprets this constraint differently depending on which method is used to estimate the length of a net. If the bounding box method is used, then **puppy** counts a violation every time the net has an estimated length that exceeds "NET_MAX_LENGTH". If the subnet method is being used, **puppy** counts a violation for every subnet that exceeds the specified length. In this way we might have the case of a critical net with many subnets, each shorter than NET_MAX_LENGTH, but such that the total length exceeds NET_MAX_LENGTH. Such situation is not penalized by the subnet method. The behavior has been chosen because it has a faster implementation and because the author believes that it is very likely that critical nets have few pins.

Every violation has a cost specified by `conViolWeight` in the `parPuppy` file.

4.4 Analog circuit placement using puppy

Analog circuits require additional constraints during the placement phase. There are topological constraints and performance constraints. In the next two sections a breef explanation of the motivation and the use of these constraints is presented[1].

[1] The analog placement methodology has been designed and implemented by Edoardo Charbon.

4.4.1 Topological constraints

Several analog circuits are fully or partially differential, to achieve improved performance (in terms of noise, gain etc.) To match the layout parasitics in differential circuits it is required that placement and routing be symmetric, with respect to certain pairs of blocks. The component f_{sy} in equation (4.1) penalizes the symmetric cells not equidistant with respect to a symmetry axis and those that do not have mirrored orientation. In case of a self-symmetric cell, the cost will be high if the cell is far from the symmetry axis.

Axis orientation and position in the space are currently user-defined. Every axis is represented by a bag called "SYM". It is attached to a bag called "XSYMMETRIES" if the axis is horizontal "YSYMMETRIES" otherwise. These two bags are directly attached to the facet. "SYM" contains a property called "POSITION", which carries the coordinate of the axis, and a non-limited number of bags called "COUPLE", to which the symmetric cells have to be attached. "COUPLE" bags must contain one or two instances only. If only one instance is attached to the "COUPLE" bag, it will be placed on the symmetry axis. The position of the axes will be maintained fixed throughout the placement process unless the option -v is entered. In the latter case the axes will be periodically updated following the scheme presented in [5]. An unlimited number of axes with different positions and orientations can coexist in one layout.

Several important performance functions of analog circuits (such as offset voltage in differential pairs and accuracy of current mirrors) depend on the matching of the electrical behaviors of circuit devices. Such electrical matching is in turn determined by the physical separation between devices and their mutual orientation. The component f_{ma} in (4.1) penalizes placements in which matched cells are too far apart or have different orientations. Matched cells must be attached to bags called "MATCH" which are contained in the bag "MATCHING". Each "MATCH" bag contains a property called "DISTANCE", which represents the maximum allowed distance between two matched modules.

In analog circuits it is desirable to place devices having the same substrate potential in a single well region. In fact a limited number of wells, in conjunction with a suitable distribution of substrate connectors, minimizes the potential gradient of silicon substrate where devices reside. Moreover the silicon area can be used more efficiently due to the reduction of separation areas between different well regions. The component f_{we} in (4.1) quantifies the discontinuity of well regions and the separation between wells of different types. In case of MOS technology **puppy** automatically recognizes an instance with n or p diffusion as n or p channel device. Make sure to assign the value "MOS" to the property CELLTYPE and "LEAF" to the property CELLCLASS in every module's interface facet. If more than one diffusion is present in the module, then it is assumed to be independently biased and as such its position won't affect f_{we} at all. The user can also define an instance as self-biased by attaching it to a bag called "SELF".

The weights of $f_{sy}, f_{ma} and f_{we}$ can be set by the user, and useful defaults are provided.

4.4.2 Performance constraints

In our approach the placement is targeted toward meeting a set of high-level constraints on performances, depending on net capacitances. During the placement phase, net capacitances are not precisely known, since they depend on the routing details. However, for a given placement conservative estimates of minimum and maximum values of capacitances can be made for each net on the ground of net length estimate and the possible layers in which the net can be routed. Hence, minimum and maximum values can be derived for the degradation of performance functions during each S.A. iteration.

If the maximum degradation for a given performance is below the bound given by specifications, then no cost function penalty is imposed. In this case the constraint will always be met during routing if the maximum estimate is a conservative one.

If the allowed bound is between the minimum and maximum degradations, a contribution proportional to the constraint violation [2] is added to the cost function. In this case there is a possibility that

[2] constraint violation is defined as the difference between the maximum degradation and the allowed bound.

the constraint be met during routing. However, when the minimum possible degradation exceeds the allowed bound, the constraint cannot be met during routing.

For these three possibilities we assigned a value to the cost function f_{cd} which also drives the placement algorithm. The sensitivity information of the circuit is provided to **puppy** trough a file called "cell.CDpuppy". This file is automatically generated using PARCAR [2].

4.5 Chip boundary

The chip boundary is the area where the cells can be placed. It should not be confused with the "chip area" which is the area of the bounding box containing all the cells. In the placement window the chip boundary is represented by a dashed box and the chip area by a solid box.

puppy can use any one of four chip boundaries:

1. AUTOMATIC: this boundary is computed by **puppy** taking into account the total area of the cells to be placed (fixed cells are ignored) and the desired aspect ratio; both the area of the automatic boundary and the aspect ration can be set in the parameter file or they can be changed interactively.
2. INITIAL_POSITION: this boundary is the bounding box of the cells in their initial placement.
3. FIXED_CELLS: this boundary is the bounding box of the cells which are FIXED; normally, this option is useful when there is a fixed pad frame.
4. SPECIFIED_BY_OCT: this boundary is obtained form the OCT data base; it corresponds to a box attached to the "FLOORPLAN" bag in the interface facet of the chip to be placed.

The chip boundary can be changed interactively using one of the menu entries.

4.6 The move set

Different sets of moves are possible on each type of annealing variable. A "cell" can be translated, rotated or interchanged with another cell. When **puppy** is running on the Sequent, the processes can also change the ownership of the cells. A "floating terminal" can be moved from one site to another, a "net" with more than 2 terminals can have its spanning tree modified.

Each process runs through the list of its own annealing variables and performs a move attempt on each of them. In particular, during a 1-processor annealing, the annealing variables are moved always in the same order.

In the following sections the moves are described in detail.

4.6.1 Moves of a cell

In the following we assume that process p has decided to move a cell named A, which is one of the cells owned by p. The process can attempt any of the moves described below.

Translation: The size of the translation vector is controlled by a range limiting mechanism ([7],[6]). The *range limiter* in **puppy** is a pair of positive numbers (RX, RY). The *range limiter window* for a cell is a rectangular area of width RX and height RY centered in the center of cell. Let W and H represent the width and height of the chip boundary. Initially the range limiter is set to $(W/8, H/8)$. It is then decreased by a factor 0.7 whenever the fraction of accepted over attempted translations is less than 0.25%. The range limiter cannot become smaller than $(4, 4)$.

In a translation, the center of cell A is translated to a random position in an area determined by the intersection of the chip boundary and the range limiter window. If the intersection is empty, the cell center is placed at random inside the chip boundary. The intersection can be empty if the cell is initially outside the chip boundary or if the chip boundary itself is changed

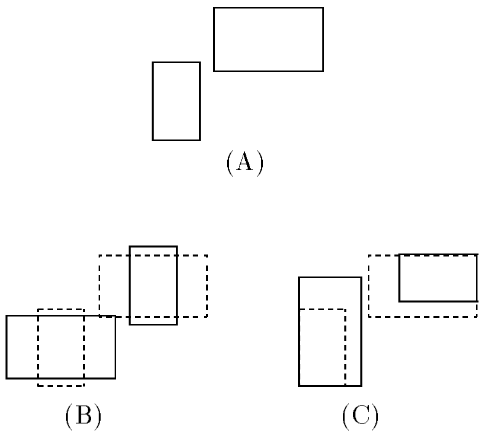

Figure 4.1: Types of pairwise interchange. (A): initial position; (B): interchange of the centers; (C): interchange keeping the bounding box

interactively, leaving the cell outside. The shaking vector (see Sec.4.6.4) is always added to the translation vector.

Alignment: (new move, description still missing).

Transformation: All the eight Manhattan transformations are possible for a cell, unless some are forbidden by using the "LEGAL_TRANSFORMATIONS" property. The coordinates of the cell center remain fixed during a transformation.

Translation and transformation: This move is simply a combination of a translation and a transformation.

Pairwise interchange: The cell to be interchanged with cell A is selected at random from the totality of the movable cells (pads are excluded). If the cell is owned by another process, e.g. process q, p tries to borrow it from q. If the borrowing is not possible the interchange is aborted. There are two flavors of interchanges (Fig.4.1), one of which involves the interchange of the centers of the cells, with or without an additional transformation. The other possibility is that of maintaining constant the bounding box of the two cells to be interchanged. The orientation of the cells is also adjusted in an attempt to exchange the aspect ratios as well.

Ownership change: *(parallel version only)* Process p selects another process q which can accept another cell. This move does not affect the placement.

A number of flags and menu options are provided to enable or disable some types of moves.
The selection of which move to perform is at random. The probabilities of selecting a particular move are determined in one of three possible ways:

1. Constant probabilities, which are hard-wired in **puppy**;
2. Global quality factors (default); the quality factor of each type of move is the product of the expected absolute value of the cost of the move times the frequency at which that type of move is accepted. Selection probabilities are made proportional to the quality factors.
3. Cell based quality factors (not fully implemented);

The rightmost graph displays the selection probabilities for all the cell moves. The quality factors are estimated using statistics from the previous temperature step.

4.6.2 Moves of a floating terminal

A cell with floating terminals is called a "soft" cell. In the current version of **puppy**, soft cells are limited to be rectangular. The floating terminals of a soft cell can be placed on a number of precomputed **sites** along the edges of the cell. The number of sites can be controlled indirectly with the `maxSiteCapacity` parameter in the `parPuppy` file. The `maxSiteCapacity` represents the maximum number of floating terminals that can occupy a site. The larger this number, the fewer sites will be necessary to hold all the floating terminals. On each edge of a soft cell, **puppy** will put the minimum number of sites, equally spaced, so that the number of terminals that can be physically squeezed on the edge is smaller than the combined capacity of the sites and each site has a capacity no greater than `maxSiteCapacity`.

Breaking the symmetry: If the sites are placed symmetrically around the cell, all the orientations that can be obtained by mirroring or by 180 degree rotations would be equivalent, and **puppy** would have a hard time in selecting a particular one among those orientations. To break the symmetry, the left and top edge of the soft cell are assigned one site more than the right and bottom edge. This has resulted in improved finite time behavior of **puppy**, which is able to focus and improve on a particular solution earlier than it would if the soft cells were symmetrical.

Only one type of move can be performed on a floating terminal, namely the change of its site. The new site is selected at random among all the sites of the cell, except the current site of the terminal. The new site can be rejected if it is already filled to capacity or if it lies on an edge which is not legal for the floating terminal (see "TERM_FLOATING_EDGES" property). If the new site is rejected, then one of the two sites adjacent to the current site is selected. If even the second site is rejected, the move of the terminal is aborted, and the algorithm continues with the next annealing variable.

In parallel, two or more processors can simultaneously and independently move a floating terminal to the same site, even if the site is almost filled to capacity. In this way, it may sometime happen that the site capacity is exceeded. This condition is a benign error which in most cases is spontaneously fixed. However the condition is detected by the double-check routines and generates a warning message, which can be ignored.

4.6.3 Moves of a net

When the length of a net is estimated using the "subnet method", **puppy** performs an annealing also on the shape of the spanning tree of the nets. For two terminal nets there is only one possible spanning tree, so that the following discussion applies only to nets with 3 or more terminals.

The steps performed for this move are:

- At random, select a node n in the spanning tree with 2 or more incident edges;
- Again at random, select 2 edges $E_m = (n, m)$ and $E_f = (n, l)$ incident on n; let E_m be the "moving edge" and let E_f be the fixed edge;
- The idea is to let E_m slide along E_f from n to l, so that $E_m = (n, m)$ is replaced by $\hat{E_m} = (m, l)$. This move maintains the connectedness of the graph.

Fig.4.2 shows an example of subnet move.

4.6.4 Shaking

The shaking corresponds to a small bias in the direction of the translations. A "shaking vector" of length "shakingStrength" in the `parPuppy` file is added to all translation vectors. The shaking vector rotates counterclockwise by 90 degrees every time the temperature is updated. The length of the shaking vector is internally bounded in the range between 0 and 10.

Shaking seems to have two positive effects: it helps remove overlap in the later stages of the annealing algorithm and it helps get a more squared chip by pushing the cells towards the corners of the available area.

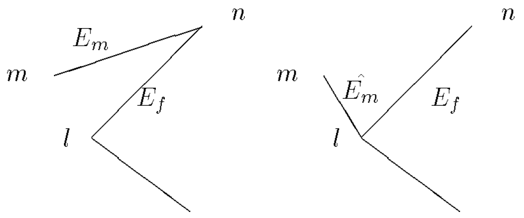

Figure 4.2: Example of subnet move

```
for I = 1 to movesPerAnnealVar do {
    foreach annealVar in list of my annealing variables do {
        move(annealVar);
    }
}
```

Figure 4.3: Inner loop

Some more statistical study is necessary to definitely establish the effect of shaking. Set shaking strength to 0 if you want to turn off any shaking.

4.7 Cooling schedule

The current version of **puppy** uses a "vanilla" cooling schedule. The algorithm performs a sequence of **steps**. During each step the annealing temperature is kept constant. At each step, the algorithm can be represented as a nested loop (see Fig.4.3). In the inner loop, each process runs through the list of all its annealing variables and attempts a move on each of them. The inner loop is repeated, at least, a number of times controlled by the parameter `movesPerAnnealVar`. We say "at least", rather that "exactly", because in the parallel version a process will continue to execute the inner loop even after it has completed `movesPerAnnealVar` iterations, until all the parallel processes have also completed that number of iterations.

The temperature used in step $k+1$ is obtained from the temperature used for step k by multiplication by a constant α whose default value is 0.9.

The annealing stops when all the following conditions are satisfied:

- the current temperature is less than `minTemperature`, which is a parameter controllable by the user;
- the cost of the placement measured at the end of the last 4 steps has not changed by more than 0.1%.

The initial temperature can be set by the user, otherwise it is automatically computed. For the automatic computation of the starting temperature, **puppy** performs a number of moves at infinite temperature (represented by -1.0), and then it computes the initial temperature based on some statistics on the cost of the performed moves.

As a rule of thumb, the initial temperature should be rather high, meaning that the acceptance ratio should be close to 1. Good values of the initial temperature vary dramatically from chip to chip.

From the menu, it is possible to set the temperature to zero, to check the behavior of the greedy algorithm, or to infinity (actually represented by the illegal temperature -1.0), to check the behavior of the totally random algorithm.

An adaptive cooling schedule has also been implemented. Early experiments show some promise. Look at the usage message for hints on how to tell **puppy** to use it.

4.8 Tuning

puppy is a blind annealing machine. It does not know what it is doing, it does not understand the meaning of the tradeoff between the various components of the cost function. The quality of the resulting placement is entirely dependent on the choice of the weights used in the cost function. The user has total control on these weights, both in the parameter file as well as interactively. This section is intended to give some advise to the novice user.

From our experience, no setting is good for all kind of circuits. Although some heuristics seem to give reasonable choices in many cases, a circuit can always be found for which those heuristics do not work.

- Given that only the relative magnitude of the weights is important, assume that the weight for the wire length is 1.
- As a rule of thumb, set the overlap weight so that at high temperatures the overlap costs about the same as the total estimated wire length.
- If you run **puppy** on the Sequent, make sure that the cluster weight is small enough, especially if the moves are selected according to their quality factors. A weight of $1E-8$ is a good start. To tune this weight turn on the debugging flag and check the quality factor for the change of ownership move at high temperatures and decrease the cluster weight until the quality factor is about tree orders of magnitude smaller than all other quality factors. This will prevent **puppy** from wasting too much time in ownership changes at high temperatures.
- If many of the cells are identical, a **grid** structure is probably the most desirable placement for those cells. It is a good idea to:
  - Use LEGAL_TRANSFORMATIONS to constraints the orientation of the identical cells;
  - Reduce the weight of nets in the cost function, and increase the area and overlap weights; by how much ?! I wish I knew, I would have coded it into **puppy**.
  - Explicitly specify the placement area (using the box in the FLOORPLAN bag).
- **Pad limited chip:** this kind of chips give problems if the core is relatively sparse. It is preferable to suggest to **puppy** the placement area (using the box in the FLOORPLAN bag). Alternatively, place the pads first and let **puppy** do the placement of the core.

Chapter 5

MOSAICO: User manual

Octtools 5.2 includes several tools for floorplanning, placement, and routing of macro-cells. The script that performs routing is called **mosaico**. This manual describes use and features of **mosaico**, as well as interactions of other tools with **mosaico**.

The **mosaico** script consists of the following steps:

channel definition: performed by **atlas**;

global routing: performed by **mosaicoGR**;

detailed routing: performed by **mosaicoDR**; one channel at a time is routed, the result being a tall hierachy of channels instantiated into other channels.

flattening: performed by **octflatten**, to restore the original netlist hierarchy;

via minimization: performed by **mizer**;

symbolic compaction: performed by **sparcs**;

verification of routing: performed by **mosaicoRC**;

Various flags in the argument list passed to **mosaico** allow the user to skip some of the early steps or to exit early. For example `-a` skips **atlas**, `-r` starts the sequence with **mosaicoDR**, `-s` exits before **sparcs**.

5.1 Get a correct net list

The input for **mosaico** is a net list, that is a description of the cells that constitute the chip with all the nets that interconnect terminals of the cells. This net list must be stored in the OCT database in a facet obeying all the rules of the *OCT Symbolic Policy*.

The most common way to describe a netlist is with **bdnet**. See the **bdnet** man page for more information on the syntax of the language (`octman bdnet`). There are some details that should be kept in mind when you set up a **bdnet** file to describe a netlist for **mosaico**:

- Remember to attach the NETTYPE (`SUPPLY` or `GROUND`) property to your supply and ground nets; example:

```
(FACET chip:netlist:contents
   (NET VDD (SPROP NETTYPE "SUPPLY" ))
   (NET GND (SPROP NETTYPE "GROUND" ))
   )
```

- Remember to attach the TERMTYPE (`SUPPLY` or `GROUND`) property to all the supply and ground formal terminals in the interface facet of the chip.

```
(FACET chip:netlist:interface             !! <<<== INTERFACE    !
   (FTERM VDD (SPROP TERMTYPE "SUPPLY" )
              (SPROP DIRECTION "INPUT" ))
   (FTERM GND (SPROP TERMTYPE "GROUND" )
        (SPROP DIRECTION "INPUT" ))
   )
```

- In general, you should always all the pads in your **bdnet** file. In most cases, however, you do not need to worry about pads. Assuming that all your formal terminals are unidirectional (i.e. their DIRECTION property is either "INPUT" or "OUTPUT", but not "INOUT"), you can just provide one formal terminal for each pad, and then you can use **padplace** to implement the pads and to place them around the chip.

- You can use the **bdnet** file to add special constraints for the placement, like the FLOORPLAN bag (see section 5.2.2), or the LEGAL_TRANSFORMATIONS property for the instances (see manual for placement tool **puppy**).

If you have doubts about the correctness of a netlist, check it with **symhelp**. **symhelp** checks an OCT facet for conformity to the OCT symbolic policy and tries to fix problems.

```
% symhelp [-d] chip:placed
```

By default, **symhelp** will not modify the cell. The `-d` flag allows **symhelp** to make the necessary modifications.

5.1.1 Advanced editing of the net list

[1]

Suppose that you have a *correct* net list and that you would like to edit it. For example, you might want to partition your design into two or more smaller pieces. The basic tool to do this is the **nle** library. You can either use **rpc-nle** to work interactively or **alba** to work in batch mode.

In order to define a partition with **alba**, you need to create the following structure in the contents facet of your chip:

- A bag whose name begins with the letters "`PARTITION`". The name can be longer. For example "`PARTITION_my_subchip`" is a legal name. The bag must contain all of the following:

- All instances that belong to the partition;

- property `PARTITION_NAME` whose value is a string which is the name of the new cell to be created;

- property `PARTITION_TYPE` with value `HARD` or `SOFT`, according to the type of merging that you want to perform (see man page for **nle**);

- optional property `MODULE_GENERATOR`, with value the name of the program that you want to use to process the partition. For example, this could be **wolfe** or **mosaico**.

This structure is used by **padplace** to specify how the pads should be merged into pad blocks in order to simplify the job of the router.

[1] You can skip this section on your first reading.

5.1.2 Implement and place pads

It is important to implement the formal terminals before running **mosaico**. Unimplemented formal terminals will not be routed.

Suppose you have your chip correctly described in the OCT view `chip:unplaced`. The next thing to do is to implement the pads. First of all, get a list of all the formal terminals and of their properties using **padplace**.

```
padplace -l chip:unplaced > chip.pads
```

The file `chip.pads` contains the list of all the terminals. The format of this file is such that it can be fed back into **padplace**. You can edit the file to add placement information like TERM_EDGE and TERM_RELATIVE_POSITION, TERM_ROW. You can find several examples of a pads-file elsewhere in this guide.

Then you can ask **padplace** to implement all the pads. **Octtools 5.2** comes with the following pad families in the *scmos* technology:

m2c_2u tested and functional family of pads.

sc2 another good family of pads, its only problem being the large size of each pad.

mosis the pads normally used for the MOSIS tiny chip frame (look in `$OCTTOOLS/tech/scmos/pads/mosis` for more details on how to handle this pad frame.

m2cframe the pads normally used for chips fabricated at M2C.

connFamily consists of simple MET1-MET2 connectors. It can be used when you are dealing with a part of a chip, rather than with the whole chip.

smallConnFamily is the same as connFamily, but power and ground pads are also minimum size.

Suppose that for your chip you want to use the `m2c_2u`, and that you want **padplace** to use the information contained in `chip.pads` to decide where each pad should go. Then you can type:

```
% padplace -u m2c_2u -D chip.pads -o chip:with_pads chip:unplaced
```

If you do not have a clear idea on where to place the pads, you can use the `-n` option of **padplace** to obtain a pad placement that attempts to minimize the length of the nets connected to the pads. This is especially useful with the `connFamily`. Try:

```
% padplace -u connFamily -o chip:with_pads chip:unplaced
```

Use **vem** to check that everything is like you want it.

5.1.3 Place the chip

It is crucial to have a good placement before you run **mosaico**. In general, you should use **puppy**, a powerful and complex tool for macro-cell placement. Chapter 4 tells you how to use this tool. For example, you will invoke **puppy** directly with:

```
% puppy -o chip:placed chip:with_pads
```

If your chip is reasonably simple, you can consider using **bdnet** to describe the placement.

```
!! Fragment of a bdnet file showing how you can
!! describe the placement of an instance.

MACRO lambda = 20 $ENDMACRO

!! ...
INSTANCE shifter:spaced NAME = "shifter" [200*lambda,330*lambda,OCT_ROT90]
!! ...
```

5.1.4 Route the chip

You now have a placed chip in `chip:placed`. To route the chip type:

```
% mosaico chip:placed
```

By default the output view is `chip:placed.spaced`, but you can specify another output view with the `-o` option.

```
% mosaico -o routed chip:placed
```

Many intermediate files and directories are created by **mosaico**, these will deleted after they are no longer needed if you run **mosaico** with the addition of the `-c` option.

```
% mosaico -c chip:placed
```

If you have already run **mosaico** without the `-c` option, you can clean the unnecessary files using the `-C` (capital C) option:

```
% mosaico -C chip:placed
```

The whole process might take anywhere from 5 minutes for a small chip with less than 10 cells and about 20 nets, to a few hours for large chips with a few tens of cells and hundreds of nets. The two standard benchmarks (`ami33` and `xerox`) take about 30 CPU minutes each on a VAX 8800.

mosaico creates a number of intermediate files, directories and facets. The `RTstuff` directory contains all the channels before and after routing. It also contains the log files generated by spider. These files can be useful to the expert user for debugging, but under normal conditions they contain no useful information. Typically, each channel requires about 20 to 200 kBytes of disk storage, depending on the amount of routing. Large channels (hundreds of nets, 50 or more tracks) can grow up to half a MByte or more.

OCT cells with name of the form <chip>.ioleft, <chip>.ioright, <chip>.iotop, <chip>.iobottom contain the I/O pads, which are merged together in order to reduce the number of channels to be routed.

The file `mosaicoDemoRpc.seq` can be used together with the RPC application `mosaicoDemo` (see manual).

The file <chip>_mosaicoRC.out contains the output from `mosaicoRC` and it is created only if any error is detected in the routing.

The following views of the chip are also created:

`placed.cd` : containing the channel definition;

`placed.cd_after_gr` : containing the global routing information;

`placed.hier` : containing the complete hierarchy of channels;

`placed.flat` : which is the flattened version of `placed.hier`;

`placed.mizer` : is the view after via minimization.

5.2 If something goes wrong

To be realistic, sometimes something goes wrong, often due to some misunderstanding or some missing property. To find out where the problem is, it is always convenient to redirect the output from `mosaico` to a file:

```
mosaico chip:placed >& mout &
```

Study this file (`mout`) to discover all the errors that have occurred. Then refer to the documentation relative to the appropriate tool to solve the problem.

5.2.1 Common problems

Overlapping instances: If there are overlapping cells, the definition of channels impossible. If the ovelapping is minor, **mosaico** will first try to expand the placement by a fixed factor of 1.1. If overlap persists after the expansion, the run will be aborted. It is up to you to solve the problem. If you used **puppy** for the placement, you can try to rerun it using a **slightly**

higher weight for the overlap cost. Also, you could just try to rerun **puppy** keeping the same parameters. **puppy** uses a probabilistic algorithm and it will generally produce different placements in each run (however the cost of the final placement will be about the same).

Terminals too close: Terminals must be separated by a minimum amount that is dependent on the technology being used regardless of the layer(s) the terminals are on. **mosaico** (**spider** in particular) will complain somewhat like this:

```
ERROR - pins too close PIN VDD at loc 40620, 63300 width 400 PIN
HandXHS1 at loc 40450, 63760 width 60
```

This indicates that terminals VDD and HandXHS1 are too close and their locations and widths. This situation must be fixed because one of these terminals will be left unconnected.

Terminals on a wrong layer: known layers for standard scmos technology are: `MET1, MET2, POLY`. Terminals on other layers cannot be used.

Other routing errors: **mosaico** performs double checking of the routing using the program **mosaicoRC**. If an error is detected, you should check the file `<cellname>_mosaicoRC.out` for a textual description of the error. In the file you will also find instructions on how to use **vem** to see the location of the error.

Power and ground nets are not proper size: this is generally due to lack of properties in the right places. See section 5.3 for more details.

Inefficient use of empty rooms: **mosaico** performs at its best on dense placements, with no empty rooms (large empty areas). If your placement has empty rooms, they might stay empty.

5.2.2 Floorplanning and Placement

The FLOORPLAN bag

To indicate the fact that a cell is soft, a bag named "FLOORPLAN" must be attached to the interface facet of the cell. The FLOORPLAN bag contains constraints for floorplanner and module generators. The bag is always and only in the interface facet. It contains the following objects and properties:

A box This box has two meanings. For the floorplanner, the area of the box represents the area of the soft cell and its shape represents the desired aspect ratio of the cell. The aspect ratio can vary in the range specified by two properties **MIN_Y_OVER_X** and **MAX_Y_OVER_X** (see below). The box may be attached to a layer (for example layer PLACE) so that it can be seen in VEM, but such attachment is optional.

For a module generator, the box represents the area suggested by the floorplanner for the implementation of the cell. **mosaico** itself can be considered a module generator, and the box in the FLOORPLAN bag in the interface of the chip to be placed specifies the area available for floorplanning and placement.

MODULE_GENERATOR is a string property, naming the module generator which is supposed to implement the cell. This property is used by the floorplanner and by `alba`. Examples: "", "wolfe", "gem", "mosaico".

MIN_Y_OVER_X Minimum aspect ratio for a soft cell.

MAX_Y_OVER_X Maximum aspect ratio for a soft cell.

FORBIDDEN_EDGES (**puppy** only) Easy way to put edge constraints on the floating pins of the cell. Example = "LEFT TOP".

Tools

puppy does not consider soft cells with variable aspect ratio yet. See also "Puppy User's Manual".

alba is a program that performs a number of checks before and after floorplanning, allowing the user to change some of the information contained in the FLOORPLAN bag.

makeSoft is a program that creates the information necessary to consider a cell as a soft cell.

5.3 Power and Ground Routing

The nets with type "SUPPLY" or "GROUND" are treated specially by **mosaico**. The terminals on those nets are searched for the property PEAKCURRENT, and the width of the nets is computed in order to accomodate such current. The value of the maximum current density for each layer is currently stored in the `MOSAICOrules` file in the technology directory. The width of the power and ground wires in Lambda is quantized to the closest of the values in the sequence:

$$2, 3, 4, 5, 6, 8, 10, 12, 14, 17, 20, 25, 30, 35, 40, \ldots$$

NETTYPE : string valued property. Legal values are: `SIGNAL,SUPPLY,GROUND,CLOCK`.

TERMTYPE : string valued property. Legal values are: `SIGNAL,SUPPLY,GROUND,CLOCK`.

PEAKCURRENT : real valued property, expressing the peak current through the terminal in amperes. Example: `0.005`

5.3.1 Verification and evaluation

mosaicoRC

mosaicoRC performs 4 different checks to asses the correctness of the routing process. It checks the consistency, from a net-list point of view, of two facets; it checks for disconnected terminals, disconnected paths, and possible violations of the symbolic policy with respect to the routing procedures.

The consistency check is performed in two steps. In the first step, the first facet is selected and all the nets attached to it are generated. For each one of the nets, all the terminals are generated. Every terminal is matched by name with a corresponding terminal on the second facet. If the matching fails, an error is recorded. The procedure is then repeated with the facets exchanged.

To check for disconnected terminals, **mosaicoRC** proceeds as follows: For each net, generate the first terminal attached to it and start to follow the path originating from it. The path is followed until either it ends on a terminal or it branches to two or more other paths. Each one of these is then followed. The procedure is repeated recursively. All the terminals are marked as soon as they are reached by one of the paths starting from the first terminal. When, for the net under consideration, all paths have been examined, all terminals attached to it must be marked. For each one of the terminals that is not marked, an error is detected and recorded.

The checking for disconnected paths is accomplished by generating all paths attached to the layers used to perform the routing. For each of the paths it is checked that connects to exactly two between terminals and connectors.

The checking for violations of the routing policies comes for free while the other checks are being performed.

The inconsistencies eventually determined are reported in two different formats: a textual one and a graphical one. For the textual messages a file named *circuitName*`_mosaicoRC.out` is created and all the violations discovered are reported in it in an textual format. The violations that are related to geometric entities are collected into a bag which is stored in OCT.

Once the execution of **mosaicoRC** is completed, the errors can be displayed graphically by means of Vem by the following sequence of operations:

run Vem;

open the contents view of the facet on which `mosaicoRC` has been run;

select the bag named ROUTING_ERRORS;

At this point the location where errors occurred will be highlighted. Each of the object that is highlighted will have a property named ROUTING_ERRORS attached to it. The value of the property is a string that describes the type of the error detected. Some of the errors, like for example differences between two facets, cannot be represented graphically. In this case the error is reported in the textual format only. The most complete report of the inconsistencies detected by **mosaicoRC** is contained in the file *circuitName*`_mosaicoRC.out`.

5.4 Floorplanning

Suppose that your chip has one cell which is not yet implemented. Suppose also that the module generator that you are going to use to implement the cell gives you some flexibility on the shape of the cell as well as on the position of its terminals. The **mosaico** system provides some support to decide which constraints to set for the implementation of the cell. This will be illustrated by an example.

The floorplanning tool in **mosaico** is **puppy** which cannot handle cells with variable aspect ratio (at least for the time being). Thus you have to decide the shape of the cell. Suppose that you want this shape to be a rectangle 100 by 300. Type the following command:

```
makeSoft -w 100 -h 300 MYSOFTCELL:someview
```

And this is all we can do for the aspect ratio of the cell.

puppy does handle floating pins on soft cells. Some terminals are only allowed to be floating on some of the edges of the cell, others can float anywhere. One easy way to tell **puppy** that the terminals are floating is to use `makeSoft`. For example:

```
makeSoft -F LR MYSOFTCELL:someview
```

specifies that all the terminals of the cell can float but only on the left or on the right edge (`-F LR`).

A more detailed representation of the constraints on the floating pins can be expressed using **padplace**:

```
padplace -l MYSOFTCELL:someview > someFile
```

to get a list of all the terminals of the cell. Then you can edit the file `someFile` which might look something like Fig.5.1. The file says that the 3 input terminals can float only on top and left edges, while the output terminals can float only on the right edge. Give this information to **padplace** again:

```
padplace -D someFile MYSOFTCELL:someview
```

Important note on floating pins

Floating terminals are very hard to deal with, especially if there are many (hundreds) of them. **puppy** finds it hard to decide on an orientation for a cell whose terminals are all floating because at high temperature almost all the orientations are equivalent. This means that for reasonable running times, the resulting placement might not be very good (it might even be unfeasible).

It is always better to constrain the floating terminals as much as possible. The possibility to do this depends, of course, on the specific problem. In some cases it makes sense to constrain the input terminals all on one side, while letting the output terminals float on the opposite side.

```
#
# Formal Terminal description for mmtlb:symbolic
# padp recognizes the following keywords:
# TERMTYPE (SUPPLY | GROUND | SIGNAL | CLOCK) : DEFAULT = SIGNAL
# DIRECTION (INPUT | OUTPUT | INOUT) : DEFAULT = INOUT
# PEAKCURRENT ( x (in Amperes)) : DEFAULT = 0.0
#   NOTE: PEAKCURRENT applies only to SUPPLY and GROUND terminals
# TERM_FLOATING_EDGES (ALL | {LEFT|RIGHT|TOP|BOTTOM})
# TERM_EDGE (LEFT | RIGHT | TOP | BOTTOM) : DEFAULT = none
# TERM_ROW ( 0 | 1 | 2 | ... | n) : DEFAULT = 0
# TERM_RELATIVE_POSITION ( 0.0 <= x <= 1.0) : DEFAULT = none
# TERM_RELATIVE_POSITION_STEP ( 0.0 <= x <= 1.0) : DEFAULT = 0.0

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
        TERMTYPE SIGNAL
        DIRECTION INPUT
        TERM_FLOATING_EDGES TOP  LEFT
FORMAL_TERMINAL tlb0_acc_q[0]
FORMAL_TERMINAL tlb1_acc_q[0]
FORMAL_TERMINAL tlb2_acc_q[0]
FORMAL_TERMINAL tlb3_acc_q[0]
FORMAL_TERMINAL clk
        DIRECTION OUTPUT
        TERM_FLOATING_EDGES RIGHT
FORMAL_TERMINAL tlb0_acc_q[1]
FORMAL_TERMINAL tlb0_acc_q[2]
FORMAL_TERMINAL tlb1_acc_q[1]
FORMAL_TERMINAL tlb1_acc_q[2]
```

Figure 5.1: A pads file

Depending on the structure of your chip, there might be some pad arrangements that make more sense than others. Although it would be desirable that the tools themselves automatically detect this condition, this is not yet the case. Locally, for each soft cell, **padplace** with its option `-n` can generally do a good job.

Mosaico and wolfe

If a chip contains standard cells like those in the MSU library, you should use **wolfe** to obtain a dense layout of these cells.

You should determine an optimal partitioning of all the standard cells in your chip. Then place and route each partition using **wolfe**, then use **mosaico** to wire all the pieces together. The process of determining an optimal partition is currently not supported by any tool (i.e. it must be done by hand, trying to be a little smart about it).

Please consider the following ideas:

- You can control the aspect ratio of a **wolfe** cell using the `-r` option (specifying the number of rows). Alternatively, you can use **makeSoft** to specify a target bounding box for the cell.
- You can control the position of each terminal in a wolfe cell using the `TERM_EDGE` and `TERM_RELATIVE_POSITION` properties. **padplace** provides an easy ASCII interface to enter such properties, and **puppy** can choose the optimal value for those properties based on its own floorplanning criteria.
- It is not a good idea to have a partition containing only one standard cell, but it is disastrous to have many such partitions. You can merge standard cells into the same partition using **nle** or **alba -p** (please refer to the appropriate manuals).

Mosaico and hand layout

Some rules should be followed to allow a painless integration of hand layout in a chip which has to be routed by **mosaico**. The rules are listed here ordered by importance:

- Do not allow terminals to overlap.
- Keep the terminals at least 8 lambdas apart (for scmos)[2]. One easy way to guarantee this result even after compaction is to implement all the terminals using MET1-MET2 vias.
- The layers connected to are MET1, MET2, and POLY. A terminal must contain at least one geometry on these layers or it cannot be connected by **mosaico**.
- Place all the formal terminals along the periphery of the cell so that they can be easily accessed from outside the cell.
- Try to use the same layer to implement all the terminals that appear on the same side of a cell. Example: make all the terminals on the TOP of the cell in MET1 and all the terminals coming out form the LEFT edge of the cell in MET2. The effect of applying this rule is that of producing "better looking" routing. This is by far the least important of all the rules.

The restrictive rules mentioned above make it easier for **mosaico**. However, **mosaico** has been designed to handle much more complicated situations, like irregularly shaped cells, terminals internal to the cell, and overlapping terminals. Please be warned that each of these situations causes **mosaico** to use complicated heuristics to determine the best way to connect those terminals, and that those heuristics may fail.

Any internal terminal must be reachable from the outside using a straight segment in one of the layers used to implement the terminal. Unreachable terminals are generally detected by **vulcan**.

mosaico (and more precisely **mosaicoGR**) uses the concept of "projection of a terminal onto a channel". There are 4 channels surrounding each cell, independent of the shape of the cell. Each

[2] The recommended spacing between terminals is the minimum center to center distance of two MET1-MET2 vias

terminal is projected onto one channel. Sometimes a terminal is projected into more channels to give more degrees of freedom to the global router.

There are a number of ways in which terminals interact among themselves. If two terminals overlap, they must be projected onto two different channels. Similarly, if the straight segment necessary to connect to an internal terminal "A" overlaps another terminal "B", then terminals "A" and "B" must also be projected onto different channels.

If it is not possible to map interacting terminals onto different channels, **mosaico** will not connect one of them. A loud warning is generated in such a case.

5.5 Alphabetical Glossary

5.5.1 Floating terminals

To specify a floating terminal, attach the property "TERM_FLOATING_EDGES" to it in the interface facet. This is a string valued property consisting of a list of the edges in which the terminal is allowed to float. For example: "LEFT TOP". The empty string "" means that all edges are legal.

You can use **makeSoft** to make all terminals of a cell floating. If you want to be more detailed, use **padplace** with the `-l` and the `-D` options (see the man page for **padplace** for more details).

A cell with floating terminals is a SOFT cell. See also Sec.5.2.2.

5.5.2 Formal Terminals

Each formal terminal must have a distinct name.

No connection can be made to terminals without an implementation. Non-floating terminals without an implementation cause a fatal error for the routing system, although **puppy** can handle floating terminals without an implementation.

Following is a list of properties that are important for the place and route tools:

DIRECTION : string valued property: one of `INPUT, OUTPUT` or `INOUT`. This property is extremely important for power and ground terminals.

PEAKCURRENT : real valued property, expressing the peak current expected through the terminal in amperes for power and ground terminals only.

TERMTYPE : can be one of `SIGNAL, CLOCK, SUPPLY, GROUND`.

TERM_FLOATING_EDGES : list of edges on which a floating terminal is allowed to be placed. The empty string is the default, meaning that all edges are legal. Examples: `"LEFT", "LEFT RIGHT", "TOP BOTTOM", "LEFT RIGHT BOTTOM"` .

TERM_EDGE : one of `LEFT, RIGHT, TOP, BOTTOM`. This is the edge where the floorplanner recommends the terminal should be placed.

TERM_RELATIVE_POSITION is a real valued property in the range [0.0,1.0] representing the position of the terminal on the edge specified by "TERM_EDGE". For example, a value of 0.75 means that the terminal should be placed in a position corresponding to 75% of the edge, computed from left to right and from bottom to top.

TERM_ROW In the case in which you need more than 1 row of pads, this property specifies the row in which the pad should go. Row 0 is the outermost row, row 1 is the next row, and so on.

5.5.3 Instances

A chip has a number of instances. Every instance which is not a connector must have a distinct name. Every connector that implements a formal terminal should also have a distinct name.

MOBILITY FIXED means that the cell's coordinates cannot be changed by **mosaico**. Default is "MOVABLE".

PLACEMENT_CLASS Used especially to recognize the pads. See section 5.5.4.

LEGAL_TRANSFORMATIONS is a string valued property. The string is a list of words from the following vocabulary:

- "ALL" , the default value, means that all the transformations are legal;
- Any OCT transformation, without the prefix "OCT_": "NO_TRANSFORM", "MIRROR_X", "MIRROR_Y", "ROT180", "ROT90", "ROT270", "MX_ROT90", MY_ROT90";
- "THIS_ASPECT_RATIO" is a short hand for the list of 4 transformations which maintain the aspect ratio of the cell: "NO_TRANSFORM MIRROR_X MIRROR_Y ROT180";
- "NOT" means that all the following transformations are not legal.

Examples: "NOT THIS_ASPECT_RATIO", "ROT90 ROT270 NO_TRANSFORM".

The property can be attached to the interface of the master. If it is attached both to the instance and to its master, the instance property overrides.

5.5.4 Pads

Pads are special instances which want to be placed along the periphery of the chip. In general, pads are flagged by a special property, but in other cases pads are implicitly defined by the fact that they provide the implentation for the formal terminals of the chip.

PLACEMENT_CLASS This property is attached either to the instance or to the interface of the master. In both cases it has value "PAD".

Pad families

This section applies to version 3 of **padplace** and all later versions.

Pads are often designed in families, each containing a Vdd pad, a Gnd pad, an input pad, an output pad, and maybe others. Pads of the same family can abut and can create power and ground rings. In this section it is explained how a pad family can be described so that **padplace** can use the family to give a correct implementation to all the formal terminals of a chip. Each pad in the family must be assigned a type and a direction. **padplace** will match type and direction of the formal terminal, to find the pad which is the proper implementation. **padplace** also will take into account power consumption of the pads to determine how many power and ground terminal to add to the chip bejond those already present in the chip.

A pad family can be of two types: instances (the "normal" type) or connectors.

A facet decribing the family must be of type SYMBOLIC and contain:

- One instance of each pad. The instances should be attached to INSTANCES bag or to the CONNECTORS bag according to the type of the family.

  The instances should be rotated so that they can be placed on the top of the chip (inner edge facing down). Each instance should be annotated with the property DIRECTION (INPUT, OUTPUT, INOUT) and TERMTYPE (SUPPLY, GROUND, CLOCK , SIGNAL). Also each instance must have the real valued property PAD_VDD_GND_REQUIRED, expressing the amount of power required by the pin. For example: Suppose that the for a particular family, one should put one vdd/gnd pair for each 4 output pads. This will be expressed by attaching to the output pad the property PAD_VDD_GND_REQUIRED with value $0.25 = 1/4$.

- One net called PAD_BONDING_PINS containing all the bonding pins which will give the implementation to the formal terminals.
- One net called PAD_INTERNAL_PINS containing all the bonding pins which will give the implementation to the formal terminals.
- One net called PAD_SUPPLY_RING containing all the terminals that must be connected to form the supply ring for the pads. The net should also contain a path of the right width on the correct layer for the ring.
- One net called PAD_GROUND_RING containing all the terminals that must be connected to form the ground ring for the pads. The net should also contain a path of the right width on the correct layer for the ring.
- One property called PAD_MINIMUM_SEPARATION to specify the minimum distance between the pads. A separation of 0 means that the pads can be abutted.

Limitations: Only 1 power voltage is supported. Tristate outputs are not supported yet.

padplace understands which cells are pads, which side should face towards the inside of the chip, and can place the pads according to a number of different schemes. An instance is a pad if one of the following three conditions is satisfied:

1. The instance has the property PLACEMENT_CLASS with value PAD;
2. The master has the property PLACEMENT_CLASS with value PAD;
3. An actual terminal of the instance is attached to a formal terminal of the cell.

5.5.5 Layer PLACE

The layer "PLACE" is a dummy layer used by **mosaico**. Currently, the routing phase of **mosaico** requires the layer to be present in the interface facet of the master of each instances. The geometry on the layer PLACE is used to define the shape of the cells.

The layer PLACE is also used to give shape and implementation to soft cells (see 5.2.2).

5.5.6 Nets

Unique names for nets are necessary. Also names should not contain weird characters such as spaces and double quotes. Use **symhelp** to check the names.

NETTYPE one of `SIGNAL, CLOCK , SUPPLY, GROUND` (*OCT Symbolic Policy*).

NET_CRITICALITY integer valued property in the range [-1000,1000] (*OCT Symbolic Policy*). Criticality 0 is the default. The higher the criticality, the shorter the net should be.

NET_MAX_LENGTH integer valued property (**puppy** only) for the placement tools, this property suggests the maximum length of the net (in lambdas).

Chapter 6

BDSYN User's Manual

6.1 Introduction

BDSYN is a hardware description translator. It takes as input a textual description of a block of combinational logic and produces a collection of logic functions which implement the described function. BDSYN was written to be a front end to the Berkeley Synthesis System. Work on **bdsyn** was begun in the spring of 1986 as a class project for Berkeley's synthesis class. For the synthesis class, we were given permission to use Digital Equipment Corporation's proprietary multi-level simulator DECSIM. The functional simulation language BDS is used for high level simulation in DECSIM. We have chosen to use a subset of the BDS language as the input language to **bdsyn**. The output of **bdsyn** is a multiple-level logic representation of the described function. The output created is in BLIF (Berkeley Logic Interchange Format). This output is suitable input for MIS which is the multiple level logic minimizer in the Berkeley Synthesis System.

6.1.1 What Bdsyn Is

BDSYN is a tool for quickly describing and implementing combinational logic. It allows the user to describe a circuit function and produce an implementation of that circuit automatically. BDSYN's output is given in a "multiple-level" format. This means that **bdsyn** is capable of translating many types of logic configurations that would be impractical to represent in a two-level PLA form. BDSYN supports many high-level language constructs: subroutines, IF–THEN–ELSE, SELECT, constant bounded FOR loops, multiple assignment to variable, and multiple returns from a single routine. In addition, **bdsyn** supports complex operators such as addition, and shifts by a variable amount. (Note that complex operators may imply a large number of gates.)

When **bdsyn** is used in conjunction with MIS, it can be used to describe logic in standard cell, PLA, or a more complicated module generated form.

6.1.2 What Bdsyn Is Not

It is very important to keep in mind that **bdsyn** can be used for describing combinational logic blocks only. We define combinational logic to be any block of logic which does not use system clocks or signal latching of any kind. Combinational logic does not hold state information. In particular, describing a finite state machine in **bdsyn** requires the logic designer (or a program) to add external latches to the described combinational logic block that hold the current state of the machine.

The limitation of describing only combinational logic is common to PLA description programs as well as **bdsyn**. (Although there may be provisions for automatically adding the required external latches.) The reason we find it necessary to describe this limitation in such detail here, however, is that some of the **bdsyn** constructs (e.g. FOR loops and multiple assignment) appear very sequential in nature. In reality, they are merely interpreted as a short hand for describing parallel structure. There are many examples of the use of sequential-like descriptions in the last section of this guide.

6.2 Using Bdsyn

6.2.1 Describing Logic in Bdsyn

BDSYN descriptions are very similar to standard procedural programming languages (Pascal, C, *etc.*). Describing logic in **bdsyn** amounts to writing a program that accomplishes the desired function. Each flow-of-control structure and operation corresponds roughly to a small block of logic. Each variable corresponds to a "bit vector", which is a group of single bit signals in the hardware.

At the top of the **bdsyn** description it is necessary to define input and output "ports." In a programming point of view, these ports can be viewed as external variable declarations. In a hardware point of view, these ports represent primary input and outputs of the combinational logic block. Once the ports are declared, the logic designer should define a set of values for the input ports for which he wishes the hardware to operate. If he then writes a program that will generate desired output values for each of the interesting input combinations, this constitutes a legal description.

To say it in a different way, suppose a **bdsyn** "program" has been written. BDSYN will create logic that will act in an identical manner to that program. If a set of inputs were to be introduced to the **bdsyn** program and the program were executed (run in a software sense), a particular set of outputs would be generated. If these same inputs were introduced to the combinational logic block created by **bdsyn**, the same set of outputs would result.

6.2.2 Running Bdsyn

Once a hardware description has been written, it can be converted to logic by running **bdsyn**. The input description is read from the file given on the command line. The BLIF output is returned to standard output. There are several command line options for **bdsyn** which are outlined in the **bdsyn** manual page. Two of the options, however, will be discussed here.

The **-b** option tells **bdsyn** not to do its cleanup evaluation. This cleanup evaluation process in **bdsyn** eliminates redundant and unnecessary logic. Unfortunately, it may also have the undesired effect of eliminating user specified intermediate variables that **bdsyn** considers unimportant. The **-b** makes sure that the variable are preserved, at the price of extra logic.[1]

The **-c** option is followed by a number which tells **bdsyn** how much "collapsing" of logic that it should do. Collapsing is done in order to decrease the amount of output which **bdsyn** creates. Although quite rare, it is possible to create an input description which causes **bdsyn** to collapse too much logic and run for inordinate amounts of time. The option **-c1** will limit the amount of collapsing that is done, and will alleviate the problem. BDSYN uses MIS do to logic collapsing for it. The actual communication with MIS is accomplished through the use of UNIX pipes. On non-UNIX systems or on a system where MIS is not available, the option **-c0** should be used to turn off collapsing altogether.

6.2.3 What is New in Version 1.1

There are two main changes in **bdsyn** since version 1.0 that users may notice. The first change is that the intermediate variable names created by **bdsyn** are shorter. Each variable is followed by two numbers. The first number is the block in the description in which the variable is set. The second number represents how many times the variable has been revised in the block.

The second change is that **bdsyn** will create its own intermediate variables corresponding to routine return values and condition expressions from IF and SELECT statements. These variables are good candidates for having a large fanout. They are created in the hope that it will save MIS some work in extracting common factors. On occasion, **bdsyn** may create an internal variable which does not fan out. MIS will give a warning when such a variable is created. In general, it is safe to ignore these warnings for variables whose name begin with "$$COND."

[1] The extra logic could easily be reduced by MIS at a later time.

AND	BEGIN	BUF	BY	CONSTANT
DO	DOWNTO	ELSE	END	ENDMODEL
ENDROUTINE	ENDSELECT	ENDSELECTALL	ENDSELECTONE	EQL
EQV	FOR	FROM	GEQ	GTR
IF	LEAVE	LEQ	LSS	MACRO
MOD	MODEL	NAND	NEQ	NOR
NOT	OR	OTHERWISE	OXT	REQUIRE
RETURN	ROUTINE	SELECT	SELECTALL	SELECTONE
SL0	SL1	SLR	SR0	SR1
SRR	STATE	SXT	SYNONYM	THEN
TO	WIDTH	XOR	ZXT	

Table 6.1: Bdsyn Reserved Words

BEHAVIOR	ENDBEHAVIOR	ENDMODULE	ENTRY	FORWARD
FOURSTATE	HALT	INFORM	INPUT	INVALID
MODULE	NOENTRY	OUTPUT	PORT	REPEAT
REVISION	STATIC	TWOSTATE	UNTIL	WHILE

Table 6.2: Keywords Ignored by Bdsyn

6.3 Language Constructs

6.3.1 Names and Numbers

BDSYN accepts a subset of the BDS language as input. BDS is a Pascal-like language. The input consists of names, numbers, and reserved keywords separated by white space. White space is spaces, tabs, and carriage returns. Any characters following a '!' (exclamation point) are interpreted as comments by **bdsyn**. Legal names may contain the characters a-z, A-Z, 0-9, '_' (underscore), and '%'. Other characters are legal in the name as long as the name is quoted with double quotes ("). Names may not be any of the reserved keywords (given below) unless they are quoted. Names that begin with '$' are illegal. BDSYN, like DECSIM, is case insensitive, so the name "signal" is the same as the name "SIGNAL."

Numbers in **bdsyn** must be positive and may be given in any one of base 2, 4, 8, 10, or 16. Decimal numbers are represented as a simple integer (e.g. `511`). Other base numbers are represented as an integer followed by a '#' and the base (e.g. `1FF#16`). The width, in bits, of a decimal number is the minimum number of bits required to represent that number. The width of a number with a '#' is equal to the number of bits required to represent a number in that base with the given number of digits. For example `001#2` has width three, `123#8` has width nine, and `12345678#16` has width thirty-two. BDSYN has the limitation that numbers in the input must be thirty-two bits or fewer. This limitation may be overcome by the use of the concatenation operator. (e.g. `12#16 & 34567890#16` is a forty bit constant.)

6.3.2 Reserved Keywords

Table 6.1 gives a list of reserved words for **bdsyn**. These may not be used as names unless they are quoted. In addition to the reserved words there are several keywords which **bdsyn** explicitly ignores but are included so that complete BDS descriptions (not just our subset) may be parsed. They are listed in table 6.2.

6.3.3 Variables

BDSYN has two types of variables which it understands. The first, "logic variables," imply actual logic and can be thought of as wires in the actual layout. Logic variables are declared with bit subscripts that define their width. The lower bit numbers correspond to the less significant bits, and **bdsyn** requires that the high bit be first.

```
STATE name1<7:0>;
```

If the bit range of a variable is omitted in its declaration,

```
STATE name2;
```

bdsyn will assume a single bit vector (`name2<0>`).

When using a logic variable (as opposed to specifying one), the bit subscript is used to choose certain bits from the bit vector. If the bit vector is left out, the the entire bit vector is used.

In addition to logic variables, **bdsyn** defines "meta-variables." Their main application is as the index variable for FOR loops. Loop index variables are required to be meta-variables. Meta-variables are declared with null bit subscripts:

```
STATE name3<>;
```

Meta-variables **do not** imply any logic, and are used for temporary storage of constants. When meta-variables appear on the left hand side of an assignment, the right hand side is evaluated and assigned to the meta-variable. The right hand side of this assignment must be a meta-variable expression (i.e., an expression written from meta-variables and constants). When meta-variables appear on the right hand of an assignment, they are treated exactly like constants. This means that meta-variables may be used in places where logic variables are disallowed by **bdsyn**. Section 6.6 covers more about meta-variables.

6.3.4 Input Format

The following is a BNF description of a typical input program for **bdsyn**. Keywords are given in capital letters, and user names and productions are given in small letters. Optional phrases are given in square brackets '`[ ]`'. Phrases in curly braces '`{ }`' many be repeated zero or more times in the input. The character '`|`' (vertical bar) represents a choice of many lines.

The basic **bdsyn** input file consists of a "MODEL":

```
MODEL model_name [output_var {, output_var}] =
                             [input_var {, input_var}] ;
    {global_declaration ;}
    {routine}
ENDMODEL [model_name];
```

A "global_declaration" is defined as the following:

```
|   STATE global_var {, global_var}
|   CONSTANT constant_name = number {, constant_name = number}
|   SYNONYM var_name = var_name {, var_name = var_name}
```

A "routine" is defined as the following:

```
ROUTINE routine_name [bit_subscripts] [( var_param {, var_param} )] ;
    { local_declaration ;}
    { [ label_name: ] statement ;}
ENDROUTINE [routine_name];
```

A "local_declaration" is defined as the following:

```
|   STATE local_var {, local_var}
|   CONSTANT constant_name = number {, constant_name = number}
|   SYNONYM var_name = var_name {, var_name = var_name}
```

A statement of a routine is defined below. Note: The square brackets in the SELECT statements represent actual brackets in the input.

```
|   BEGIN {statement ;} END
|   variable = expression
|   IF expression THEN statement [ELSE statement]

|   FOR meta_var FROM const_expr TO const_expr
                        [BY const_expr] DO statement

|   FOR meta_var FROM const_expr DOWNTO const_expr
                        [BY const_expr] DO statement

|   SELECT expression FROM {[expression {, expression}]: statement;}
                        {[OTHERWISE]: statement ;} ENDSELECT

|   SELECTALL expression FROM {[expression {, expression}]: statement;}
                    {[OTHERWISE]: statement ;} ENDSELECTALL

|   SELECTONE expression FROM {[expression {, expression}]: statement;}
                        {[OTHERWISE]: statement ;} ENDSELECTONE

|   RETURN [expression]
|   LEAVE label_name
```

Note that these tables imply that a semicolon ';' separates every statement. As in Pascal, there is no semicolon before ELSE, but unlike Pascal there is a semicolon before END. Expressions and const_expr's (constant expressions) will be described in later sections.

6.3.5 Declaration Statements

The MODEL statement defines the primary inputs and outputs to the combinational logic block being described. The declared "ports" may not be meta-variables.

The STATE, CONSTANT, and SYNONYM may be used as either global declarations, applying to all routines, or local declarations, applying to only one routine. The STATE statement is for declaration of variables. All variables must be declared prior to use. Local variables can be used only within the scope of a routine and their values are lost between consecutive calls to a routine. Constant declarations are used to assign a name to a constant. Declared constants can be used anywhere a constant can. Synonyms are used to create an alternate name for a previously defined variable. They are particularly useful for defining subfields of a variable:

```
SYNONYM opcode<6:0> = instruction<31:25>;
```

6.3.6 Flow-of-control Statements

BDSYN offers a rich selection of flow-of-control statements. They are IF, SELECT, SELECTONE, SELECTALL, FOR, RETURN, and LEAVE. The IF statement implements optional execution of a statement. If the least significant bit of the IF expression evaluates to one, the THEN clause is used. If the low bit evaluates to zero, the ELSE clause (if present) is used. The SELECT statement is used to execute one statement out of many. If the expression following the SELECT keyword is equal to the expression appearing in the square brackets, the corresponding statement is executed.

SELECTONE and SELECTALL are simply a shorthand for several IF statements. SELECTONE implies nested IF statements:

```
SELECTONE expr FROM
    [expr1, expr2]: statement1;
    [expr3]: statement2;
    [OTHERWISE]: statement3;
ENDSELECTONE;
```

is equivalent to:

```
IF (expr EQL expr1) OR (expr EQL expr2) THEN
    statement1
ELSE IF expr EQL expr3 THEN
    statement2
ELSE
    statement3;
```

SELECTALL is equivalent to sequential IF statements:

```
SELECTALL expr FROM
    [expr1, expr2]: statement1;
    [expr3]: statement2;
    [OTHERWISE]: statement3;
ENDSELECTALL;
```

is equivalent to:

```
IF (expr EQL expr1) OR (expr EQL expr2) THEN
    statement1;
IF expr EQL expr3 THEN
    statement2;
IF (expr NEQ expr1) AND (expr NEQ expr2) AND (expr NEQ expr3) THEN
    statement3;
```

SELECTALL is different than SELECT in that SELECTALL implies sequentiality in its case evaluation. The SELECT statement can be viewed as having all of its cases evaluated in parallel. A SELECTALL allows two cases to evaluate to true. If two cases are true in a SELECT, incorrect logic may be produced. SELECT generally implies much less logic, however, because there is no implied ordering of the cases.

FOR statements are used to iterate a single piece of code several times. As a hardware description language, **bdsyn** must enforce several rules on the FOR statement. The expressions which specify the start, end, and step values for the loop index must be either constants, or constant expressions. This is because the number of iterations must be fixed. BDSYN also requires that the index variable of the FOR statement is a meta-variable. FOR loops may be nested, and since meta-variables are treated as constants by **bdsyn**, the inner loop may depend on the outer loop.

```
FOR i FROM 0 TO 7 DO
    FOR j FROM i TO 7 DO
        statement;
```

The RETURN statement is used to break out of a subroutine and return to the calling routine. If the subroutine returns a value (as described below) the RETURN statement must be followed by an expression which is the return value. The LEAVE statement is similar to RETURN in that can be used to break out of a block of statements. When a LEAVE statement is encountered, control will break out of a labeled statement. The LEAVE must occur within the labeled statement to which it points. For example:

```
loop: FOR i FROM 1 TO 10 DO BEGIN
    statement1;
    IF condition THEN LEAVE loop;
    statement2;
END;
```

Notice that by using the LEAVE it is possible to describe a loop whose number of iterations depends on a logic variable (*condition* in this case). It is only important that the FOR statement itself have extents which are constants or meta-variables.

6.3.7 Routines and Routine Calls

BDSYN has no concept of a main routine, per se. At runtime, **bdsyn** routines are classified as main routines or subroutines depending on whether or not they called by another routine. BDSYN allows many main routines in the same model, and these routines can be viewed as running in parallel. It is an error in **bdsyn** for two disjoint routines (one is not called by the other) to set the same global variable or primary output of the block. Two disjoint routines may call the same subroutine, however. It is illegal to have recursive routine calls.

Subroutines may return a single variable (regular or meta) to the calling routine. A subroutine that returns a value is declared by placing a bit subscript after the routine name in the ROUTINE declaration. The bit subscript specifies the width of the return value or if the value is a meta-variable. If the subscript is missing, the routine returns no value. Parameters passed to subroutines must be declared as part of the ROUTINE declaration. Parameter passing has the same semantics as assignment statements (described below).

A common error in **bdsyn** is to have a subroutine which is not called by another routine. This causes **bdsyn** to treat the subroutine as a main routine.

6.3.8 Assignment Statements

Assignment statements are written in the form:

```
variable = expression
```

They result is that the variable receives the value of the expression. If the width of the variable is wider than the width of the evaluated expression, extra zeros will be added to pad the high bits of the variable. If the expression is wider than the variable, the high bits are truncated. If the variable is a meta-variable the expression may contain only constants and other meta-variables.

It is possible to assign a value to the same variable twice in the same program. BDSYN will treat this situation in the same manner as any sequential programming language. In particular the following works correctly.

```
x = 1;   y = x;   x = 0;
IF x NEQ y THEN is_ok = 1;
```

Section 6.4 contains further details on multiple assignment.

6.3.9 Expressions

Legal expressions are of the following form. They are listed in order of precedence, and expressions higher on the list are evaluated first. The curly braces in the ZXT, OXT, and SXT expressions represent actual characters in the input, not a repeatable phrase.

```
|   number                          ! number as described above
|   constant                        ! a declared constant
|   meta_variable
|   regular_variable
```

```
|   (expression)                      ! parenthesized expression
|   expression<expression>            ! choose a bit
|   expression<const_expr:const_expr>       ! choose several bits
|   expression & expression           ! concatenation
|   expression SR0 expression         ! shift right filling with zeros
|   expression SR1 expression         ! shift right filling with ones
|   expression SRR expression         ! rotate right
|   expression SL0 expression         ! shift left filling with zeros
|   expression SL1 expression         ! shift left filling with ones
|   expression SLR expression         ! rotate left
|   expression * expression           ! multiplication
|   const_expr / const_expr           ! integer division
|   const_expr MOD const_expr         ! mod operation
|   expression + expression           ! addition
|   expression - expression           ! subtraction
|   expression EQL expression         ! Equal comparison
|   expression NEQ expression         ! Not equal comparison
|   expression LSS expression         ! Less than comparison
|   expression LEQ expression         ! Less than equal comparison
|   expression GTR expression         ! Greater than comparison
|   expression GEQ expression         ! Greater than equal comparison
|   BUF expression                    ! null operation
|   NOT expression                    ! bitwise NOT
|   expression AND expression         ! bitwise AND
|   expression NAND expression        ! bitwise NAND
|   expression OR expression          ! bitwise OR
|   expression NOR expression         ! bitwise NOR
|   expression XOR expression         ! bitwise XOR
|   expression EQV expression         ! bitwise XNOR
|   WIDTH expression                  ! return number of bits
|   ZXT {WIDTH = const_expr} expression     ! zero extend
|   OXT {WIDTH = const_expr} expression     ! one extend
|   SXT {WIDTH = const_expr} expression     ! sign extend
```

In the above "const_expr" represents a constant expression. A constant expression is an expression that contains only numbers, declared constants, and meta-variables. It may not contain regular variables.

The single bit selection operator, `<0>`, may take an arbitrary expression as the bit selector. When the bit selector is not a constant, a multiplexor is implied. The multiple bit selection operator, `<31:0>`, requires that both expressions be a constant expression and that the first number is larger than the second. The concatenation operator '`&`' joins two bit vectors into one. The expression before the '`&`' becomes the high order bits, the trailing expression becomes the low order bits.

The shift operations shift the first expression by an amount given by the second expression. When the shift amount is a variable expression, a barrel shifter is implied. Note: the shift operation does not change the width of the shifted expression. This means it is possible to shift bits off the end of the expression accidentally. Care should be taken to extend the expression (using ZXT) so no bits are lost.

'+', '-', '*', '/', and MOD implement integer arithmetic and are supported for constant expressions. In addition, '+', '-', and '*' are supported for variable expressions. '+' and '-' imply a combinational adder in this case, and '*' implies a full combinational multiplier. The '*' operation should be used sparingly on variable expressions since it implies so much logic. In particular, '*' should not be used where a shift would be sufficient.

The comparison operators evaluate to a single bit 1 if the condition is true and a single bit 0 if the condition is false. If an expression in the comparison is a variable, a comparator is implied.

The bitwise boolean operators take two bit vectors and perform the desired operation bit by bit. If one of the expressions is smaller than the other, it will be automatically zero extended. The WIDTH operator returns a constant which is the number of bits in the given expression. ZXT, OXT, and SXT will add more significant bits to the given expression to make it the given width. It is an error to specify a width that is smaller than the given expression.

6.3.10 Macro Definitions and Required Files

BDSYN has the facility for defining macro definitions in the input description. Macro definitions may appear anywhere in the input description and have the following format:

```
MACRO macro_name = macro_body $ENDMACRO
```

Following a macro definition, any use of *macro_name* will be replaced by the *macro_body*. The *macro_body* is expanded exactly as typed (including comments). It may span more than one line and may contain uses of other macro definitions.

BDSYN also allows distinct input files to be included within other files. This is done using the REQUIRE command. A REQUIRE may occur anywhere in a **bdsyn** file and has the syntax:

```
REQUIRE 'filename.ext';
```

The text of the required file will be inserted in place of the REQUIRE command.

6.4 Multiple Assignment

As mentioned in the introduction, **bdsyn**'s input descriptions appear very sequential in nature yet are intended to describe combinational logic. We feel that our form of sequential description gives the user a great deal of power and flexibility in specifying logic. The translation of sequential description to combinational logic is accomplished by correctly handling "multiple assignment."

Our definition of multiple assignment is the correct and consistent handling of many sequential assignments to the same variable. For example:

```
x = default_value;
IF condition THEN x = new_value;
```

describes a multiplexer which is controlled by "condition." The above example could be rewritten in a more conventional way:

```
IF condition THEN
    x = new_value
ELSE
    x = default_value;
```

At first glance, one might think that multiple assignment is not particularly useful. In fact, multiple assignment is the heart of **bdsyn**'s ability to process sequential descriptions. Several examples of this can be found in the "Examples" section at the end of this guide.

6.5 Unspecified Variables

In **bdsyn** input descriptions, as in any programming language, it is possible to have variables which are not always defined. For instance, in the following code:

```
IF cond THEN
    x = expr;
```

the variable x, if it has not previously been assigned to, is unspecified when *cond* is false.

Under normal circumstances **bdsyn** will assume that when variables are unspecified, they should have the value zero. This assumption is probably not a good one. In some cases, the unspecified variables could be caused by a mistake in the user specification. In other cases the user may not care about a variable value under certain conditions. In this case the user should probably specify `DONT_CARE` conditions as discussed in section 6.8.

In general, it is very difficult for **bdsyn** to detect the use of unspecified variables. However, unspecified variables may be detected by using MIS. First, **bdsyn** should be run using the **-n** flag. This will cause **bdsyn** to create a group of variables that end with the characters "**". (Also, the ".inputs" line in the BLIF output will not be printed.) The BLIF file containing the "**" variables should then be read into MIS and minimized. After minimization, if any of the "**" variables fan out to any gates, this corresponds to the use of an unspecified variable.

6.6 Meta-variables

Meta-variables have a very separate use from logic variables. They can be used within constant expressions because they represent constant values. Unlike logic variables, they are guaranteed to always assume the same particular value. The value of a logic variable may change based on the primary inputs to the **bdsyn** combinational block.

Because **bdsyn** is able to statically evaluate the value of meta-variables, it is able to perform some simple logic minimization when meta-variables are involved. In the following example, function1 and function2 are complex functions. (i is a meta-variable.)

```
FOR i FOR 0 TO 5 DO
    IF i MOD 2 EQL 0 THEN
        x = function1(x)
    ELSE
        x = function2(x);
```

After **bdsyn** minimization this becomes:

```
x = function1(x);
x = function2(x);
x = function1(x);
x = function2(x);
x = function1(x);
x = function2(x);
```

Had the condition in the IF statement above depended on a logic variable, no such minimization would have been possible inside of **bdsyn**. This is because the value of a logic variable is generally unpredictable.

One non-obvious problem with meta-variables is the following. A meta-variable is defined within a branching structure (i.e. IF, SELECT, etc.). If the condition of the branching structure depends on a logic variable, the meta-variable is not correctly defined upon exit from that structure. For example:

```
meta = 3;
IF x EQL 1 THEN BEGIN
    meta = 4;
    a<0> = w<meta>;      ! This works correctly
END
a<1> = w<meta>;          ! This uses meta = 4 even if ( x NEQ 1 )
```

This problem stems from indirectly trying to define the meta-variable *meta* in terms of the logic variable x. For similar reasons, meta-variables that are used for loop counts are not necessarily correct upon exit from the loop. For example:

```
loop: FOR i FROM 1 TO 10 DO BEGIN
    x = function(x);
    IF x EQL 1 THEN LEAVE loop;
END;
y = i;              ! y gets the value 10 regardless of the function
```

6.7 Complex Operators

BDSYN supports several operations which cannot be implemented by just one or two gates. These include '+', '-', '*', LEQ, GEQ, LSS, GTR, shifts, and single bit selection when they are operating on non-constants. The operations imply complex structures such as adders, multipliers, comparators, barrel shifters, and multiplexors. The complex operators are implemented through a set of library routines found in the file "bdsyn.lib". The macros in the library are used to create routines of an appropriate size (a 3 bit adder, or a 4 bit comparator, etc) which are then inserted into the code.

6.8 Don't Care Conditions

It is often the case that it makes no difference what the value of an output is. For example, in a finite state machine with seven states encoded in three state bits, state 7 never occurs:

```
SELECT state<2:0> FROM
    [0,1,2,3]: output = 0;
    [4,5,6]: output = 1;
ENDSELECT;
```

Here, the output for state 7 would be unspecified. To resolve this, a default value for output might be added.

```
output = 0;
SELECT state<2:0> FROM
    [0,1,2,3]: output = 0;
    [4,5,6]: output = 1;
ENDSELECT;
```

The problem is that this is unnecessarily restrictive, and does not allow for all of the potential logic simplification. In actuality, it may make no difference whether "`output = 0`" or "`output = 1`" in state 7, because state 7 never occurs. For this reason, **bdsyn** has the special variable `DONT_CARE`, which will allow logic optimization tools to choose the optimal value for output. `DONT_CARE` is used as any other variable:

```
output = DONT_CARE;
SELECT state<2:0> FROM
    [0,1,2,3]: output = 0;
    [4,5,6]: output = 1;
ENDSELECT;
```

Inside of **bdsyn**, assignments from this variable will automatically be extended to fit the size of the destination. When `DONT_CARE` is used, it will automatically be added to the input port list in **bdsyn**'s BLIF output. There is no need to declare the `DONT_CARE` variable.

The current version of MIS cannot process the don't care conditions produced by **bdsyn**. This is because don't care specifications in multiple level descriptions can be ambiguous in their meaning. Until a uniform method for handling multiple level don't cares is implemented, it is still possible to use **espresso** to make partial use of the don't care specification. This can be accomplished by the following (in UNIX):

```
bdsyn input_descr | mis -c clp -T pla | \
        espresso -Dmapdc | espresso > output.pla
```

The resulting PLA can then be read into MIS.

If a description has been written that includes a don't care specification and the use of the don't care specification is not desired, the **-z** option should be used. The **-z** option maps every use of `DONT_CARE` to zero. The effect will be that in places where `DONT_CARE` is specified, the result will be forced to the value zero.

6.9 Examples

Figures 6.1 through 6.5 contain five examples of various **bdsyn** descriptions. The first is a generic finite state machine. It merely generates the next state and control signals based on the present state. The next three descriptions implement the exact same piece of logic. They are included to demonstrate the flexibility of **bdsyn**. Note that the second implementation (COUNT2) uses intermediate variables *seenZero* and *seenTrailing* as state holders. The variables do not really imply latches to hold the state. They are simply a convenient way to describe an idea.

The fifth example is a complicated decoder circuit. Particular attention should be given to the last statement of the main routine. It illustrates an important aspect of **bdsyn**. The decoder is described as a barrel shifter which shifts a single bit 1 onto the correct word line. This is obviously a very inefficient way to build a decoder. It is important to note, however, that **bdsyn** and MIS are capable of reducing the hardware to a more sensible implementation. There is generally no penalty associated with a quick and dirty shortcut description. The logic that is generated is the same.

6.10 Acknowledgments

The parser, inline routine expansion, and macro expansion inside of bdsyn were implemented by Richard Rudell. The FOR loops, RETURN and LEAVE processing, multiple assignment, and mapping to BLIF were implemented by Russell Segal. To do logic collapsing, bdsyn uses MIS, which was implemented by Albert Wang and Richard Rudell. We would like to thank the students and industrial visitors who participated in the Spring 1986 synthesis class for running bdsyn through its paces. We also would like to thank Professors Richard Newton, Alberto Sangiovanni-Vincentelli, and Carlo Sequin for organizing the synthesis project and giving bdsyn a reason to exist.

6.11 Figures

```
! The classic traffic light controller finite state machine from
! Mead and Conway, Introduction to VLSI Technology, page 85
MODEL traffic_light
    hl<1:0>,            ! control for highway light
    fl<1:0>,            ! control for farm light
    st<0>,              ! to start the interval timer
    nextState<1:0> =
    c<0>,               ! indicating a car on the farm road
    ts<0>,              ! timeout of short interval timer
    tl<0>,              ! timeout of long interval timer
    presentState<1:0> ;
! state assignments
CONSTANT HG = 0, HY = 2, FG = 3, FY = 1;
! symbolic output assignments
CONSTANT GREEN = 0, RED = 1, YELLOW = 2;

ROUTINE traffic_light_controller;
    ! set up default outputs (use of multiple assignment)
    nextState = presentState;
    st = 0;

    SELECT presentState FROM
        [HG]: BEGIN
            hl = GREEN;   fl = RED;
            IF c AND tl THEN BEGIN
                nextState = HY;
                st = 1;
            END;
          END;
        [HY]: BEGIN
            hl = YELLOW;   fl = RED;
            IF ts THEN BEGIN
                nextState = FG;
                st = 1;
            END;
          END;
        [FG]: BEGIN
            hl = RED;   fl = GREEN;
            IF NOT c or tl THEN BEGIN
                nextState = FY;
                st = 1;
            END;
          END;
        [FY]: BEGIN
            hl = RED;   fl = YELLOW;
            IF ts THEN BEGIN
                nextState = HG;
                st = 1;
            END;
          END;
    ENDSELECT;
ENDROUTINE;
ENDMODEL;
```

Figure 6.1: Traffic Light Controller

```
! COUNT1 (Count Zeros):

! The input is an 8 bit binary string expected to consist of
! a string of 1's, a string of 0's, and a string of 1's.  Any
! of these strings may be zero length.
! For example, `11000111', `10000001', and `11110000' are all
! valid strings, and `10001100' is a malformed string.

! The problem is to design a circuit which, given the 8 bit
! string, returns the count of number of consecutive zeros in
! the string, or returns an error condition if the string is
! malformed.  The output is undefined if the input string is
! malformed.

MODEL count1 error<0>, out<3:0> = in<7:0>;

! find first bit matching 'val' (return index of the bit)
ROUTINE ff<3:0>(x<7:0>, val<0>);
    STATE i<>;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL val THEN
            RETURN i;
    RETURN 8;
ENDROUTINE;

ROUTINE main;
    STATE x<7:0>;

    ! Shift off the first string of 1's
    x = in SR1 ff(in, 0);

    ! Count is where the first 1 is
    out = ff(x, 1);

    ! Check for a malformed string:
    ! shift off 0's, and check for any more 0's
    x = x SR1 out;
    error = ff(x, 0) NEQ 8;
    IF error THEN
        out = DONT_CARE;

ENDROUTINE;
ENDMODEL;
```

Figure 6.2: Count1

```
! COUNT2 (Count Zeros)
! A different implementation for the last example.

MODEL count2 error<0>, out<3:0> = in<7:0>;
CONSTANT TRUE = 1, FALSE = 0;

ROUTINE legal<0>(x<7:0>);
    STATE i<>;
    STATE seenZero;         ! there has been a zero
    STATE seenTrailing;     ! we have hit the trailing ones field

    seenZero = FALSE;
    seenTrailing = FALSE;
    FOR i FROM 0 TO 7 DO
        IF seenTrailing AND (x<i> EQL 0) THEN
            RETURN FALSE                        ! Illegal string
        ELSE IF seenZero AND (x<i> EQL 1) THEN
            seenTrailing = TRUE
        ELSE IF x<i> EQL 0 THEN
            seenZero = TRUE;

    ! If we made it to here then the input is legal
    RETURN TRUE;
ENDROUTINE;

ROUTINE zeros<3:0>(x<7:0>);
    STATE i<>, count<3:0>;
    count = 0;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL 0 THEN
            count = count + 1;
    RETURN count;
ENDROUTINE;

ROUTINE main;
    IF legal(in) THEN BEGIN
        error = FALSE;
        out = zeros(in);
    END ELSE BEGIN
        error = TRUE;
        out = DONT_CARE;
    END;
ENDROUTINE;
ENDMODEL;
```

Figure 6.3: Count2

```
! COUNT3 (Count Zeros)
! Yet another implementation

MODEL count3 error<0>, out<3:0> = in<7:0>;
CONSTANT TRUE = 1, FALSE = 0;

! check for a legal string
ROUTINE legal<0>(x<7:0>);
    STATE i<>, j<>, k<>;

    ! Look for a ...0...1...0...
    FOR i FROM 0 to 5 DO
        FOR j FROM i+1 to 6 DO
            FOR k FROM j+1 to 7 DO
                IF (x<i> EQL 0) AND (x<j> EQL 1) AND
                                    (x<k> EQL 0) THEN
                    ! The input is illegal
                    RETURN FALSE;

    ! If we made it to here then the input is legal
    RETURN TRUE;
ENDROUTINE;


ROUTINE zeros<3:0>(x<7:0>);
    STATE i<>, count<3:0>;
    count = 0;
    FOR i FROM 0 TO 7 DO
        IF x<i> EQL 0 THEN
            count = count + 1;
    RETURN count;
ENDROUTINE;

ROUTINE main;
    IF legal(in) THEN BEGIN
        error = FALSE;
        out = zeros(in);
    END ELSE BEGIN
        error = TRUE;
        out = DONT_CARE;
    END;
ENDROUTINE;
ENDMODEL;
```

Figure 6.4: Count3

```
! Register decoder for the SPUR cpu chip.
! The register file of the SPUR cpu uses a Berkeley RISC
! type register windowing scheme.  The register file is
! divided into 8 overlapping register "windows".  Each
! register window can access 32 of SPUR's 138 registers.
! The first 10 registers of all of the windows point to
! the same 10 "global" registers.  The next 6 registers
! are shared with the top six registers of the previous
! window.  The 6 top registers of each window are similarly
! shared with the next window.  The last window wraps
! around and shares registers with the first window.
MODEL reg_decode
    addr<137:0>     ! one-hot decoded output (word lines)
=
    cwp<2:0>,       ! Current window pointer
    reg<4:0>;       ! Current register in the window

CONSTANT
    NUMREGS = 138,      ! total number of registers
    NUMGLOBALS = 10;    ! number of global registers

ROUTINE REG_DECODE();
    STATE
        sum<7:0>,       ! temporary variable
        decAddr<7:0>;   ! decoded address (number of the word line)

    ! calculate the correct register number
    IF (reg LSS NUMGLOBALS) THEN
        ! point to the global registers
        decAddr = regSpec

    ELSE BEGIN
        sum = (cwp & 0000#2) + regSpec;

        ! Check for wrap around.  If so, need to subtract an offset
        IF (sum GTR (NUMREGS - 1)) THEN
            sum = sum - (NUMREGS - NUMGLOBALS);

        decAddr = sum;
    END;

    ! Do the actual decoding (*** See main text for details ***)
    addr = (ZXT {width=138} 1) SLO decAddr;

ENDROUTINE;
ENDMODEL;
```

Figure 6.5: Register file decoder

Index

Bibliography

[1] Andrea Casotto and Alberto Sangiovanni-Vincentelli. Placement of standard cells using Simulated Annealing on the Connection Machine. In *The Proceedings of the International Conference on Computer-Aided Design*. IEEE, 1987.

[2] U. Choudhury and A. Sangiovanni-Vincentelli. An analytical-model generator for interconnect capacitances. In *Proc. IEEE Custom Integrated Circuit Conference*, 1991.

[3] Bruce Hajek. A tutorial survey of theory and applications of simulated annealing. In *Proc. IEEE Conf. on Decision and control*, Dec 1985.

[4] S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, may 1983.

[5] E. Malavasi, E. Charbon, G. Jusuf, R. Totaro, and A. Sangiovanni-Vincentelli. Virtual symmetry axes for the layout of analog ic's. In *Proc. IEEE Int. Conf. on VLSI and CAD*, 1991.

[6] Carl Sechen. *Placement and Global Routing of Integrated Circuits Using Simulated Annealing*. PhD thesis, University of California, Berkeley, 1987.

[7] Carl Sechen and Alberto Sangiovanni-Vincentelli. The TimberWolf placement and routing package. In *Proc. 1984 Custom Integrated Circuit Conference*, Rochester, may 1984.