Examples of Tool Use in Octtools 3.5

Rick L. Spickelmier

Electronics Research Laboratory
University of California, Berkeley

Paul Cohen

Massachusetts Microelectronics Center

March 5, 1990

Contents

1 Synthesis Example Using Different Layout Strategies **2**
- 1.1 Specification . . . 2
- 1.2 Standard Cell Implementation . . . 2
- 1.3 PLA Implementation . . . 9
- 1.4 Gate Matrix Implementation . . . 10

2 Simple Finite State Machine **13**
- 2.1 Specification . . . 13

3 A 4-bit Up/Down Counter, Paul Cohen (M2C) **18**
- 3.1 Specification . . . 18
- 3.2 Design . . . 18
- 3.3 Simulation . . . 20
- 3.4 Layout . . . 20

4 Synthesis and Optimization **22**

5 Tool Roadmap **29**

List of Figures

1.1 Road Map for the WOLFE Synthesis Path 7
1.2 OCTPLA roadmap 11
1.3 GEM roadmap 12

2.1 FSM Roadmap 14

Chapter 1

Synthesis Example Using Different Layout Strategies

The following section shows using 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 in the *Octtools Users Guide* for detailed information on each tool.

All of the input files are included in this section, you should take a look at these as you go through the examples.

1.1 Specification

1.2 Standard Cell Implementation

This section shows how to use the Octtool set 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 -z count.bds > count.blif
  ```

  Translate the `bds` functional description into a set of logic equations in `blif` format. The '-z' maps all don't cares in the input file to zero.

- 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.

The program can 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**.

```
 !!!! this input deck is stored in 'count.musa'
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 00011100    ! set the input vector
musa> ev                 ! evaluate the circuit
musa> show err out       ! display the output vectors
 err:1  out:H0            ! error flag is set
musa> set in 11100011    ! 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 10000001    ! 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
```

- 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
  ```

  Th program **padplace** is used for assigning 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.

  Note that **wolfe** only guarentees that the terminals with be on the specified edge, not necessarily in the same relative order requested.

- wolfe (standard cell place and route)

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

  This runs **wolfe** reading the logic view of the cell and creating the symbolic view of the count cell. Note that logic and symbolic 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, and 'symbolic' to refer to the placed cell.

- mizer (minimize vias)

  ```
  % mizer -o count:final count:symbolic
  ```

  Minimize the number of vias in the layout. Minimizing the number of vias in a design will improve the yield.

- musa

  ```
  % musa count:final

      musa> source count.musa
  ```

Verify that the logic is still the same.

- chipstats (statistics about the layout)

  ```
  % chipstats count:symbolic
  ```

  ```
  % chipstats count:final
  ```

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

```
Tue Oct 18 10:32:12 PDT 1988
chipstats: Version 1.0 : Chip  count:symbolic

Number of instances    = 86
Number of nets         = 94
Number of connectors   = 436
Number of vias         = 317

Average instance area  =        3292.28
St.dev. instance area  =        1079.72
Chip Area W x H        =    688 x   724
Chip Area              =      498112.00 (100.00%)
Total Instance Area    =      283136.00 ( 56.84%)
Routing + Empty Area   =      214976.00 ( 43.16%)
Total   net area       =      195052.50
Total   net length     =       50061.50
Average net length     =         532.57
St.dev. net length     =         494.85

***** 6 LONGEST NETS
(---) NAME                  LENGTH (lambda)  AREA (lambda^2)
(  1) GROUND                        3126.00            31956.00
(  2) SUPPLY                        3078.00            31524.00
(  3) [691]                         1509.00             4527.00
(  4) [722]                         1375.00             4125.00
(  5) [865]                         1190.00             3570.00
(  6) [848]                         1184.00             3552.00
```

- vulcan (protection frame generator)

  ```
  % vulcan -k count:final
  ```

  Create an *interface* facet for 'count:final'. The interface facet is an abstract version of the contents facet with the hierarchy cut off and the detailed geometry replaced with *protection frames*[**?**]. The `-k` flag has to be used with caution; it destroys any existing properties in the interface facet.

- vem

  `% vem`

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

  `vem> "count:symbolic" o`

  `vem> "count:final" o`

  Notice that now the cells have been placed and routed and that in the second window, the number of vias has been reduced.

- net-length (remote application)

  While still in `vem`, give the command:

  `vem> : net-length`

  "net-length initialized" will appear in the **vem** console window. **net-length** is a remote application for determining the length of nets in a symbolic layout and displaying them in length sorted order. Once **net-length** is initialized, the middle mouse button (while holding down the shift key) will bring up a new menu containing `net-length` specific commands.

- oct2ps (convert Oct to PostScript for printing)

  `% oct2ps -f bb count:final > 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.

Behavioral Description (count.bds)

```
! 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>);
```

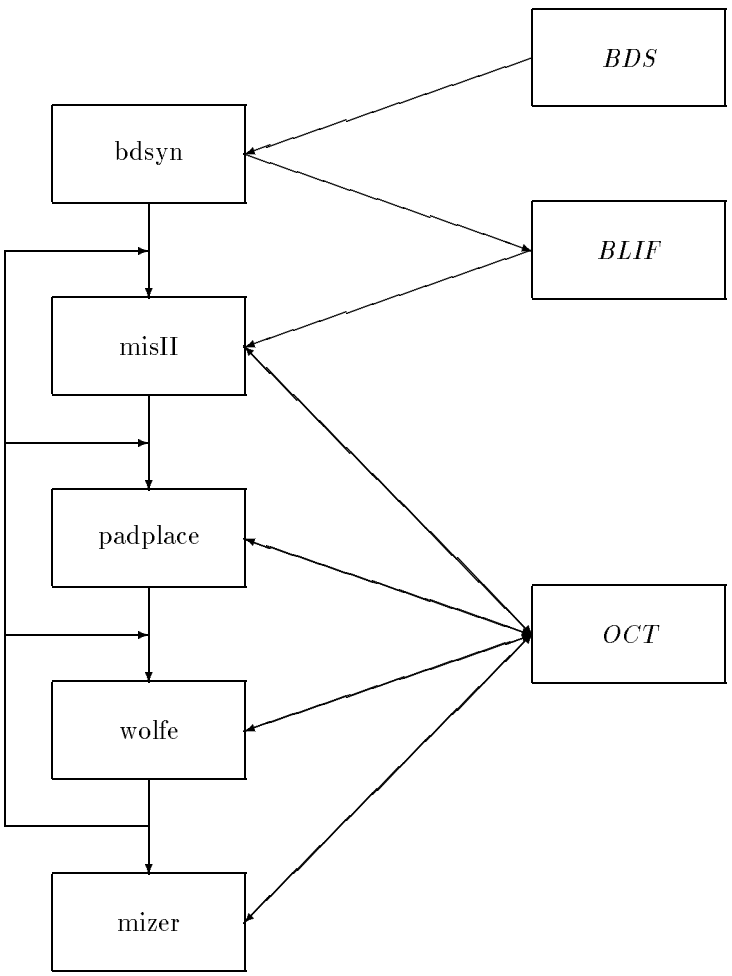


Figure 1.1: Road Map for the WOLFE Synthesis Path

```
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;
```

Simulation Input (count.musa)

```
mv in in<7:0>     ! create an input vector
mv out out<2:0>   ! create output vectors
mv err error<0>
set in 00011100   ! set the input vector
ev                ! evaluate the circuit
show err out      ! display the output vectors
verify err 1
verify out 000
set in 11100011   ! change the input vector
ev                ! re-evaluate the circuit
show err out      ! display the new output vector
verify err 0
verify out 011
set in 10000001   ! and so on....
ev
show err out
verify err 0
verify out 110
exit
```

Pad Placement Description (count.padlist)

```
#
```

```
# Formal Terminal description for count:logic
# padplace recognizes the following keywords:
# TERMTYPE DIRECTION
# TERM_FLOATING_EDGES TERM_EDGE
# TERM_RELATIVE_POSITION TERM_RELATIVE_POSITION_STEP

# This file uses a 'REVERSE' logic:
# First you specify the properties
# then the terminal for which the props apply
TERMTYPE SIGNAL
DIRECTION OUTPUT
TERM_ROW 0
TERM_EDGE RIGHT
TERM_RELATIVE_POSITION 0
TERM_RELATIVE_POSITION_STEP 0.2
FORMAL_TERMINAL out<0>
FORMAL_TERMINAL out<1>
FORMAL_TERMINAL out<2>
FORMAL_TERMINAL error<0>
DIRECTION INPUT
TERM_EDGE LEFT
TERM_RELATIVE_POSITION 0
TERM_RELATIVE_POSITION_STEP 0.1
FORMAL_TERMINAL in<5>
FORMAL_TERMINAL in<4>
FORMAL_TERMINAL in<7>
FORMAL_TERMINAL in<6>
FORMAL_TERMINAL in<3>
FORMAL_TERMINAL in<2>
FORMAL_TERMINAL in<1>
FORMAL_TERMINAL in<0>
```

1.3 PLA Implementation

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

- bdsyn (convert `BDS` to logic equations)

  ```
  % bdsyn -z count.bds > count.blif
  ```

- octpla (generate PLA from `blif`)

  ```
  % octpla -o pla count
  ```

  Read in the blif file, run **misII** to convert to to *pla* format, run **espresso** to minimize the logic, run **genie** to generate the pla array, run **octopus** to connect up the cells, and run **sparcs** to compact the layout. Generates `count:pla`. 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.

- musa (simulate the PLA)

  ```
  % musa count:pla

      musa> source count.musa
  ```

  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**.

- chipstats

  ```
  % chipstats count:pla
  ```

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

1.4 Gate Matrix Implementation

This section shows the same example as above, but using **gem** to generate the layout (see Figure 1.3).

- bdsyn (convert BDS to logic equations)

  ```
  % bdsyn -z count.bds > count.blif
  ```

- gem (generate oct array from blif)

  ```
  % gem count
  ```

  Read in the `blif` file, run **misII** to convert to to `pla` format, run **espresso** to minimize the logic, run **genie** to generate the array, run **electra** to connect up the cells, and run **sparcs** to compact the layout. Generates `count:gem`.

- musa

  ```
  % musa count:gem

      musa> source count.musa
  ```

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

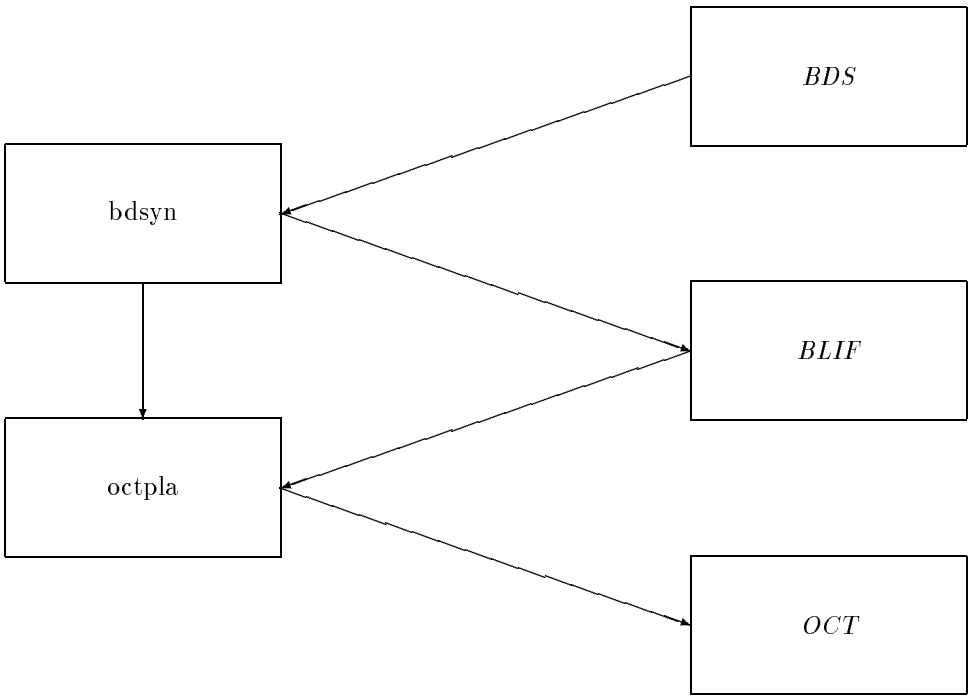


Figure 1.2: OCTPLA roadmap

[1]

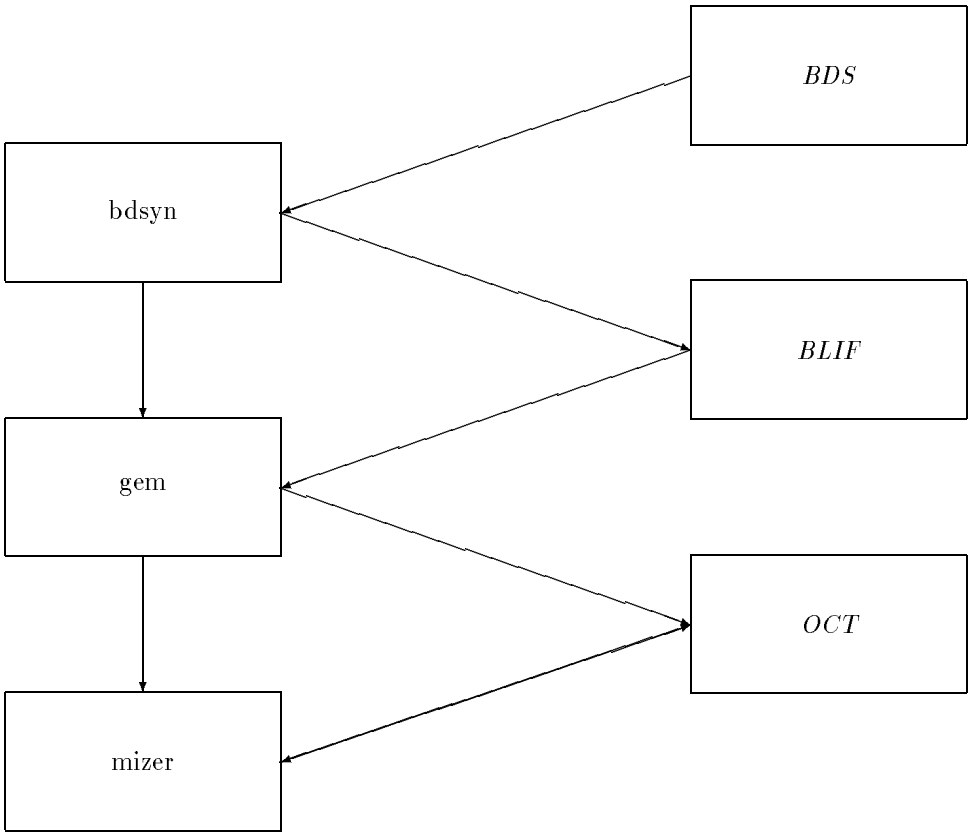


Figure 1.3: GEM roadmap

[2]

- vem

  `% vem`

  `vem> "count:gem" o`

  Take a look at the result.

- chipstats

  `% chipstats count:gem`

  Find out the area.

Chapter 2

Simple Finite State Machine

This section shows how to build a simple finite-state machine, in this case, a traffic light controller (see Figure 2.1).

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 -z fsm.bds > fsm.blif
  ```

- 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.

- octpla (minimize and generate a PLA)

  ```
  % octpla -o pla fsm
  ```

- bdnet (wire the latches)

  ```
  % bdnet fsm.bdnet
  ```

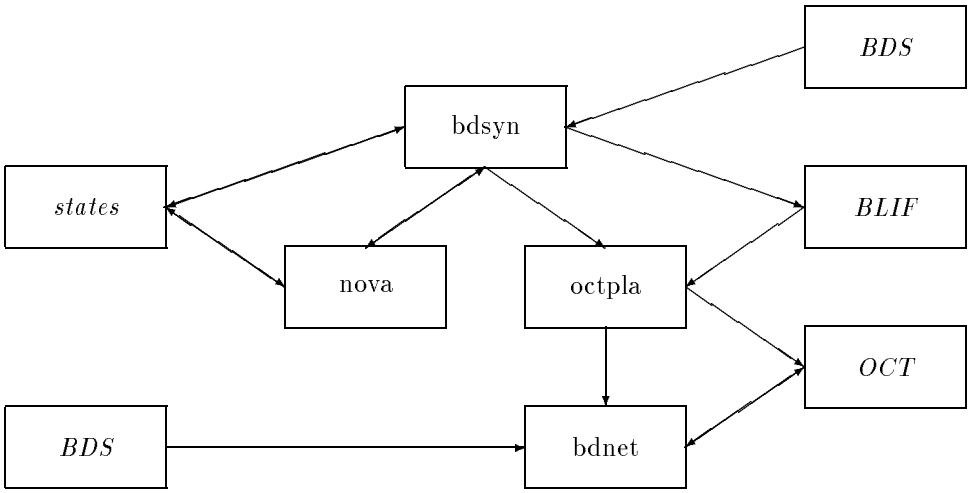


Figure 2.1: FSM Roadmap

- musa (simulate the FSM)

  ```
  % musa fsm:logic < fsm.musa
  ```

Behavioral Description (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;
```

State Table (fsm.st)

```
.encoding_algo i_exact
-0- 11 10 00000
-1- 01 10 00000
0-- 10 10 00000
--1 10 10 00000
1-0 10 10 00000
-0- 11 01 00000
-0- 01 01 00000
0-- 10 01 00000
--1 10 01 00000
1-1 00 01 00000
--- 01 00 10000
--- 11 00 01000
--- 10 00 01000
--- 11 00 00100
--- 01 00 00010
--- 00 00 00010
-1- 11 00 00001
-1- 01 00 00001
0-- 10 00 00001
--1 10 00 00001
1-1 00 00 00001
```

Wiring Description (fsm.bdnet)

```
!
! the purpose of this BDNET file is to wire
! the nextState output to the prevState input
!
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>: state<1:0>;
```

```
presentState<1:0>: state<1:0>;
Vdd# Vdd;
GND# GND;
outPhi# clk;

ENDMODEL;
```

Simulation Input (fsm.musa)

```
makevector a a<0>
```

Chapter 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.

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

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 updown.bds > updown.blif
```

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;
TECHNOLOGY scmos;
VIEWTYPE SYMBOLIC;
EDITSTYLE SYMBOLIC;

INPUT clk, ud<0>, clr<0>;
OUTPUT q<3:0>;
SUPPLY Vdd;
GROUND GND;

INSTANCE updown:logic PROMOTE;

ARRAY %i FROM 0 TO 3 OF
    INSTANCE "~octtools/lib/technology/scmos/msu/stdcell2_0/dfnf311":physical
        DATA1:d<%i>;
        CLK2:clk;
        Q:q<%i>;
        Q_b:UNCONNECTED;
        "Vdd!":Vdd;
        "GND!":GND;
ENDMODEL;
```

The command:

```
bdnet updown.bdnet
```

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

3.3 Simulation

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

```
mv q q<3:0>
ma c
se clk 1
ev
se clk 0
ev
sh ud<0> clr<0> q
$end
se clr<0> 0
c
se clr<0> 1
se ud<0> 1
c
c
c
se ud<0> 0
c
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.

Typing:

```
musa updown:unplaced
```

will invoke the simulator. When the circuit is read into **musa**, the command:

```
source updown.sim
```

will read the simulation command file. The above commands can also be entered directly into **musa**.

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
```

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

```
wolfe -f -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`).

Chapter 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:

- minimize the logic via three different techniques

  ```
  % 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

  ```
  % wolfe -f -o :zero-pandr ex:zero
  % wolfe -f -o :algebraic-pandr ex:algebraic
  % wolfe -f -o :boolean-pandr ex:boolean
  ```

- compare the areas and cell counts

  ```
  % chipstats ex:zero-pandr
  ```
  115 instances, 813288 square lambda
  ```
  % chipstats ex:algebraic-pandr
  ```
  67 instances, 430272 square lambda
  ```
  % chipstats ex:boolean-pandr
  ```
  29 instances, 189952 square lambda

ex.blif

```
.model CountZeros
.inputs in<7> in<6> in<5> in<4> in<3> in<2> in<1> in<0> DONT_CARE
.outputs error<0> out<2> out<1> out<0>
```

```
.names in<7> main/$SR1_8_4_1/a<7>.2
1 1
.names in<6> main/$SR1_8_4_1/a<6>.2
1 1
.names in<5> main/$SR1_8_4_1/a<5>.2
1 1
.names in<4> main/$SR1_8_4_1/a<4>.2
1 1
.names in<3> main/$SR1_8_4_1/a<3>.2
1 1
.names in<2> main/$SR1_8_4_1/a<2>.2
1 1
.names main/$SR1_8_4_1/a<7>.2 main/$SR1_8_4_1/a<7>.3
1 1
.names main/$SR1_8_4_1/a<6>.2 main/$SR1_8_4_1/a<6>.3
1 1
.names main/$SR1_8_4_1/a<5>.2 main/$SR1_8_4_1/a<5>.3
1 1
.names main/$SR1_8_4_1/a<4>.2 main/$SR1_8_4_1/a<4>.3
1 1
.names main/$SR1_8_4_1/a<7>.3 main/$SR1_8_4_1/a<7>.5
1 1
.names main/$SR1_8_4_1/a<6>.3 main/$SR1_8_4_1/a<6>.5
1 1
.names main/$SR1_8_4_1/a<5>.3 main/$SR1_8_4_1/a<5>.5
1 1
.names main/$SR1_8_4_1/a<4>.3 main/$SR1_8_4_1/a<4>.5
1 1
.names main/$SR1_8_4_1/a<3>.2 main/$SR1_8_4_1/a<3>.5
1 1
.names main/$SR1_8_4_1/a<2>.2 main/$SR1_8_4_1/a<2>.5
1 1
.names in<1> main/$SR1_8_4_1/a<1>.5
1 1
.names in<0> main/$SR1_8_4_1/a<0>.5
1 1
.names main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$SR1_8_4_1/
01----- 1
0-01--- 1
0-0-01- 1
------1 1
.names main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$SR1_8_4_1/
1-----1- 1
-01---1- 1
-0-01-1- 1
-0-0-01- 1
01-----1 1
0-01---1 1
0-0-0--1 1
-0----11 1
.names main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$SR1_8_4_1/
1----1- 1
-01--1- 1
-0-011- 1
01----1 1
```

```
0-01--1 1
-0---11 1
.names main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$SR1_8_4_1/
1---1- 1
-01-1- 1
-0-01- 1
01---1 1
0-0--1 1
-0--11 1
.names main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$SR1_8_4_1/
1--1- 1
-011- 1
01--1 1
---11 1
.names main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$SR1_8_3_1/
1-1- 1
-01- 1
0--1 1
--11 1
.names main/$SR1_8_3_1/a<7>.2 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/
1-------- 1
-001----- 1
-00-1---- 1
-00--001- 1
-00--00-1 1
.names main/$SR1_8_3_1/a<6>.2 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/
1-------- 1
-001----- 1
-00-1---- 1
-00--001- 1
-00--00-1 1
.names main/$SR1_8_3_1/a<5>.2 main/$SR1_8_3_1/a<7>.2 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/
1-1------- 1
1--1------ 1
-1001----- 1
-100-1---- 1
1---001--- 1
1---00-1-- 1
-100--001- 1
-100--00-1 1
1---00--00 1
.names main/$SR1_8_3_1/a<4>.2 main/$SR1_8_3_1/a<6>.2 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/
1-1------- 1
1--1------ 1
-1001----- 1
-100-1---- 1
1---001--- 1
1---00-1-- 1
-100--001- 1
-100--00-1 1
1---00--00 1
.names main/$SR1_8_3_1/a<7>.3 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/
1-------- 1
-00001--- 1
-0000-1-- 1
```

```
-0000--1- 1
-0000---1 1
.names main/$SR1_8_3_1/a<6>.3 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/
1-------- 1
-00001--- 1
-0000-1-- 1
-0000--1- 1
-0000---1 1
.names main/$SR1_8_3_1/a<5>.3 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/
1-------- 1
-00001--- 1
-0000-1-- 1
-0000--1- 1
-0000---1 1
.names main/$SR1_8_3_1/a<4>.3 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/
1-------- 1
-00001--- 1
-0000-1-- 1
-0000--1- 1
-0000---1 1
.names main/$SR1_8_3_1/a<3>.2 main/$SR1_8_3_1/a<5>.2 main/$SR1_8_3_1/a<7>.3 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/
1--1------- 1
1---1------ 1
-1-001----- 1
-1-00-1---- 1
--100001--- 1
--10000-1-- 1
--10000--1- 1
--10000---1 1
1----000000 1
.names main/$SR1_8_3_1/a<2>.2 main/$SR1_8_3_1/a<4>.2 main/$SR1_8_3_1/a<6>.3 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/
1--1------- 1
1---1------ 1
-1-001----- 1
-1-00-1---- 1
--100001--- 1
--10000-1-- 1
--10000--1- 1
--10000---1 1
1----000000 1
.names main/$SR1_8_3_1/a<3>.2 main/$SR1_8_3_1/a<5>.3 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/
--11------ 1
1-0-1----- 1
---11----- 1
1-00-1---- 1
-100001--- 1
-10000-1-- 1
-10000--1- 1
-10000---1 1
.names main/$SR1_8_3_1/a<2>.2 main/$SR1_8_3_1/a<4>.3 main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/
--1------- 1
---1------ 1
1---1----- 1
1----1---- 1
-1--001--- 1
```

```
-1--00-1-- 1
-1--00--1- 1
-1--00---1 1
.names main/$SR1_8_3_1/a<0>.4 main/$SR1_8_3_1/a<1>.4 main/$SR1_8_3_1/a<2>.4 main/$SR1_8_3_1/a<3>.4 main/$SR1_8_3_1/
0------- 1
-0------ 1
--0----- 1
---0---- 1
----0--- 1
-----0-- 1
------0- 1
-------0 1
.names DONT_CARE out<1>.1/1.1
1 1
.names DONT_CARE out<2>.1/1.1
1 1
.names DONT_CARE out<0>.1/1.1
1 1
.names error<0> main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$S
000001---- 1
00000-1--- 1
00000--1-- 1
00000---1- 1
1--------1 1
.names error<0> main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$S
0001------ 1
000-1----- 1
000--001-- 1
000--00-1- 1
1--------1 1
.names error<0> main/$SR1_8_4_1/a<0>.5 main/$SR1_8_4_1/a<1>.5 main/$SR1_8_4_1/a<2>.5 main/$SR1_8_4_1/a<3>.5 main/$S
001------- 1
00-01----- 1
00-0-01--- 1
00-0-0-01- 1
1--------1 1
.end
```

ex.setup

```
#
set OCT-CELL-VIEW physical
set OCT-CELL-PATH ~octtools/lib/technology/scmos/msu/stdcell2_0
set OCT-TECHNOLOGY scmos
set OCT-EDITSTYLE SYMBOLIC
set OCT-VIEWTYPE SYMBOLIC
#
# loading standard-cell library description
#
read_library stdcell2_0.genlib
#
```

ex.zero

```
#
sr ex.setup
#
# mapping logic to standard-cells
#
map
phase -g
print_stats
print_gate -s
#
# verify result
#
verify ex.blif
```

ex.algebraic

```
#
sr ex.setup
#
# doing algebraic minimization
#
sw
el 5
sim1 *
asb

gkx -abt 30
asb; sw
gcx -bt 30
asb; sw

gkx -abt 10
asb; sw
gcx -bt 10
asb; sw

gkx -ab
asb; sw
gcx -b
asb; sw

el 0
gd *
#
# mapping logic to standard-cells
#
map
phase -g
print_stats
print_gate -s
#
# verify result
```

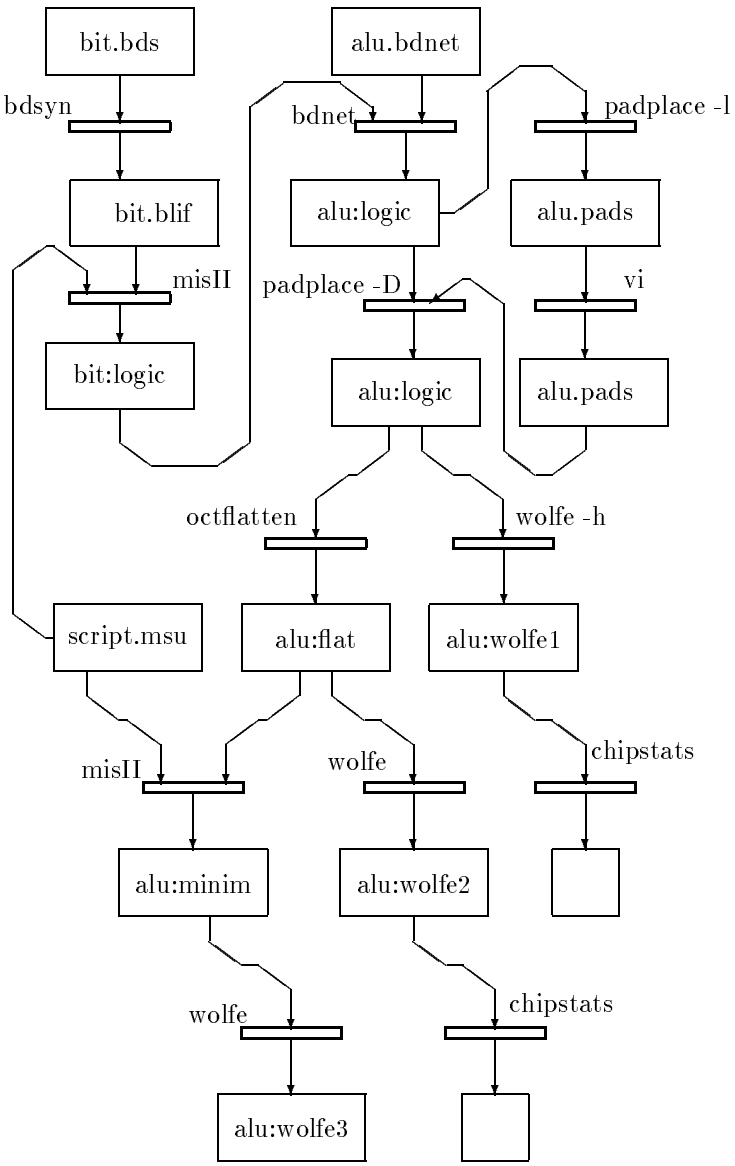
```
#
verify ex.blif
```

ex.boolean

```
#
sr ex.setup
#
# doing boolean minimization
#
el 10
sim1 *
asb
gkx
asb
el 0
asb
gkx
asb
el 0
sim2 *
gd *
asb
el -1
asb
sim2 *
el -1
#
# mapping logic to standard-cells
#
map
phase -g
print_stats
print_gate -s
#
# verify result
#
verify ex.blif
```

Chapter 5

Tool Roadmap



Index

bdnet 19
bds 2
BDS 9, 13
bdsyn 13, 19
blif 2, 9, 10, 13

electra 10
espresso 9, 10

file formats
  BDS, 9, 13
  bds, 2
  blif, 2, 9, 10, 13
  pla, 10
  PostScript, 6
files
  script.msu, 2
  updown.bdnet, 19
  updown.bds, 19

gem 10
genie 9, 10

misII 3, 9, 10, 22
musa 20

net-length 6
nova 13

Oct 6, 18, 19
oct2ps 6, 21
octflatten 20
octopus 9
octpla 9

packages
  Oct, 6, 18, 19
padplace 4, 9
pla 10
PostScript 6

script.msu 2
sparcs 9, 10

TimberWolfSC 21
tools
  bdnet, 19
  bdsyn, 13, 19
  electra, 10
  espresso, 9, 10
  gem, 10
  genie, 9, 10
  misII, 3, 9, 10, 22
  musa, 20
  net-length, 6
  nova, 13
  oct2ps, 6, 21
  octflatten, 20
  octopus, 9
  octpla, 9
  padplace, 4, 9
  sparcs, 9, 10
  TimberWolfSC, 21
  vem, 3, 4, 6, 21
  wolfe, 2, 4, 20, 21

updown.bdnet 19
updown.bds 19

vem 3, 4, 6, 21

wolfe 2, 4, 20, 21