# A BCD Counter with 7 Segment Output using the OCT tools*

Paul B. Cohen
Massachusetts Microelectronics Center
$M^2C$

## 1 Introduction

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 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.)

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

Table 1: Tool list

*©Copyright 1989 Massachusetts Technology Park Corporation.

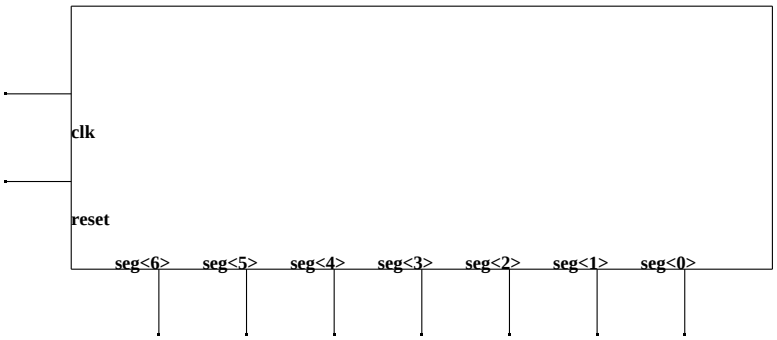


Figure 1: BCD counter / 7 Segment Decoder Chip

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

Table 2: Signal Descriptions

## 2 Specification

Figure 1 shows a symbol for the chip, with the I/O signals labeled. Table 2 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 2. The segment pattern for each number is shown in table 3

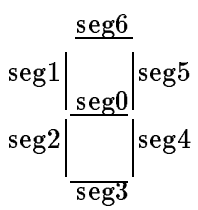


Figure 2: 7 segment display

| 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 3: 7 segment display decoder truth table

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` wil cause the output pattern to return to the "0" pattern.

# 3 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.

# 4 Block Design

# 5 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 (ns GTR 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). **Gem** uses the BLIF description to generate a gate matrix style layout of the logic. Synthesize the layout with the commands:

```
bdsyn count.bds > count.blif
gem count
```

(The **gem** script invokes a number of tools, and will print quite 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:gem -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:gem
```

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 `musa>` to `>` while the macro `in` is being defined, and simulation results are indented 1 space.

```
musa> mv ps ps<3:0>
musa> mv ns ns<3:0>
musa> ma in
> se ps $1
> ev
> sh "%h %h" ps ns
> $end
musa> in h0
 ps:H0   ns:H1
musa> in h1
 ps:H1   ns:H2
musa> in h2
 ps:H2   ns:H3
musa> in h3
 ps:H3   ns:H4
musa> in h4
 ps:H4   ns:H5
musa> in h5
 ps:H5   ns:H6
```

```
musa> in h6
 ps:H6   ns:H7
musa> in h7
 ps:H7   ns:H8
musa> in h8
 ps:H8   ns:H9
musa> in h9
 ps:H9   ns:H0
musa> in ha
 ps:HA   ns:H0
musa> quit
```

The counter control logic is now complete.

## 6 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/lib/technology/scmos/msu/dlm/1580":physical
     1DATA:d<%x>;
     2CLK:clk;
     3RST:reset;
```

```
     Q:q<%x>;
     QB:UNCONNECTED;
     "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, `musa>`. **Musa** uses the letters ``I'' and ``0'' 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

musa: Message: Version 3-00-X11 (made Jul 27 1989)
musa> mv d d<3:0>
musa> mv q q<3:0>
musa> ma cycle
> se clk 1
> ev
> se clk 0
> ev
> sh "%b %b %b" reset d q
> $end
musa> se reset 0 ! reset all 4 ff's
musa> ev
musa> sh "%b %b %b" reset d q
 reset:0   d:xxxx   q:0000
musa> se reset 1 ! won't do much with reset low
musa> se d h5    ! set inputs to 0101
musa> cycle      ! cycle clock 1 time
 reset:I   d:0I0I   q:0101
musa> se reset 0 ! make sure reset is asynch
musa> ev
musa> sh "%b %b %b" reset d q
 reset:0   d:0I0I   q:0000
```

```
musa> se reset 1 ! try a few more values
musa> se d ha
musa> cycle
 reset:I   d:IOIO   q:1010
musa> se d h0
musa> cycle
 reset:I   d:0000   q:0000
musa> se d hf
musa> cycle
 reset:I   d:IIII   q:1111
musa> 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 -o reg:padp 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 command:

```
wolfe -f -r1 -o reg:placed reg:padp
```

When **TimberWolf** is invoked, each step of the simulated annealing process will be indicated by a number. There are 123 steps. (**TimberWolf**) may print a warning such as:

```
WARNING: Unable to redistribute pins in channel 2
```

The logic is correct, so this message can be ignored.

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**.

# 7 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 gate-matrix layout with **gem**. Enter the following 2 commands:

```
bdsyn decode.bds > decode.blif
gem decode
```

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

```
musa decode:gem
```

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.

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

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:gem -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**.

# 8 Assembling the Chip

# 9 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:gem PROMOTE
     data<3>:q<3>;
     data<2>:q<2>;
     data<1>:q<1>;
     data<0>:q<0>;
INSTANCE count:gem 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
c
c
c
c
c
c
c
```

```
c
c
c
se reset 0
ev
sh "%b %b" reset seg
se reset 1
c
c
c
```

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

Since each of the cells in the chip is a module, we should add the OCT property, `CELLTYPE` to each cell. This can be done with `octprop` or `bdnet`, but we will use `octprop` here. Enter the following commands:

```
octprop -pCELLTYPE=MODULE reg:placed:interface
octprop -pCELLTYPE=MODULE decode:gem:interface
octprop -pCELLTYPE=MODULE count:gem:interface
```

(The three command lines above may be combined into a single command line. See the manual for `octprop(1)` for further information.)

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

```
musa -i bcd.sim bcd:logic
```

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 3 to appear on the screen.

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**.

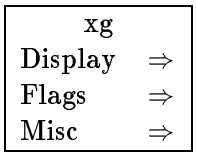


Figure 3: Plot menu

## 10 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 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<0>
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 `_msuFamily`. The alternative is `_2uFamily`. Create a new OCT facet, `bcd:padp` with the pads included by entering:

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

## 11 Place and Route the Blocks

**Mosaico** can now be used to place and route the blocks. **Mosaico** is actually a shell script, which invokes a host of tools to perform the place and route, 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.

There will likely be warnings from various tools, but **mosaico** will continue unless there are fatal errors.

Enter the command:

```
mosaico -P -c -o final bcd:padp
```

When mosaico finishes, the chip is complete. **Vem** can be used to view the resulting 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**.

You may also want to verify that the chip operates properly by re-simulating the final layout of the chip. **Musa** can be used with the command script `bcd.sim` (the same script used earlier) to check the chip. Enter:

```
musa -i bcd.sim bcd:final
```

Remember, this script does not have a `quit` command, so you will need to enter the **musa** command:

```
quit
```

at the `musa>` prompt when you are satisfied with the simulation results.

## 12 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.