

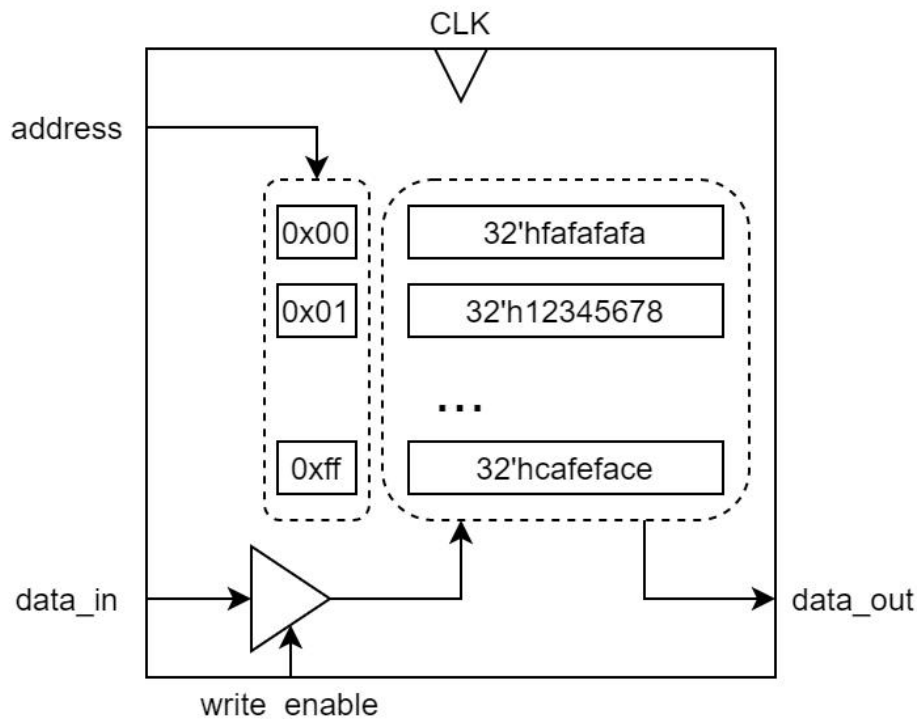
Bài thực hành-thí nghiệm 4: Thiết kế bộ nhớ

Bài thực hành-thí nghiệm này sẽ hướng dẫn cơ bản cách thức tạo, cũng như sử dụng bộ nhớ trong thiết kế vi mạch số.

6.1 Lý thuyết

6.1.1 Bộ nhớ

Bộ nhớ là tên gọi cho một bộ phận các thiết bị, phần tử với nhiệm vụ lưu trữ thông tin cho nhiều mục đích khác nhau. Thời gian lưu trữ của bộ nhớ có thể thay đổi từ ngắn hạn đến dài hạn, kích thước lưu trữ và các cách định địa chỉ cũng khác nhau tùy theo từng loại bộ nhớ. Tuy nhiên, nhìn một cách tổng quát thì các bộ nhớ đều có chung một cấu trúc sau: truy xuất dữ liệu bên trong bộ nhớ thông qua các ô nhớ, với mỗi ô nhớ có một hoặc một vài địa chỉ để xác định vị trí (hình 6.1).



Hình 6.1. Ví dụ cho bộ nhớ thông thường.

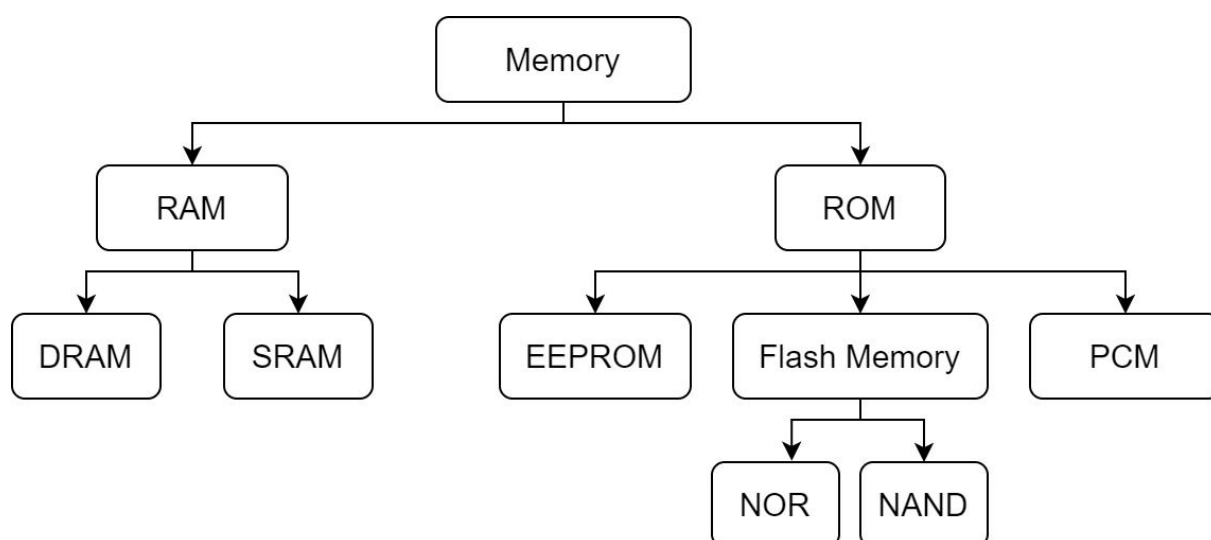
6.1.2 Các loại bộ nhớ

Nếu phân loại dựa trên cách thức vận hành, bộ nhớ được chia làm hai nhóm chính: bộ nhớ truy xuất ngẫu nhiên (Random Access Memory – RAM) và bộ nhớ chỉ đọc (Read Only Memory – ROM). Trong đó, RAM cho phép người dùng đọc và ghi tùy theo nhu cầu truy xuất dữ liệu. RAM thường được dùng khi vi mạch hoạt động như một vùng nhớ đệm, khi mà nhu cầu sử dụng dữ liệu tăng cao và cần truy xuất theo tình huống. Ngược lại, ROM là bộ nhớ chỉ cho phép đọc, với kích thước lớn, thường được dùng cho khởi tạo vận hành các thiết bị lớn và được ghi một lần duy nhất vào lúc đầu. ROM thông thường vẫn có thể lưu trữ dữ liệu khi ngắt nguồn điện, trong khi RAM thường sẽ mất hết dữ liệu trong trường hợp này.

Bên cạnh đó, bộ nhớ còn được phân loại theo rất nhiều cách thức khác nhau, từ cấu tạo linh kiện, tốc độ truy xuất cho đến khả năng tái sử dụng (hình 6.2). Người thiết kế vi mạch cần lựa chọn loại bộ nhớ thích hợp với thiết kế của mình và lựa chọn cách thức điều khiển cho phù hợp. Khi truy cập bộ nhớ, người dùng cần chú ý các tín hiệu cho phép ghi/đọc, tín hiệu lựa chọn bộ nhớ trong trường hợp sử dụng nhiều bộ nhớ, cũng như các bộ định địa chỉ nhằm truy xuất đúng dữ liệu cần thiết.

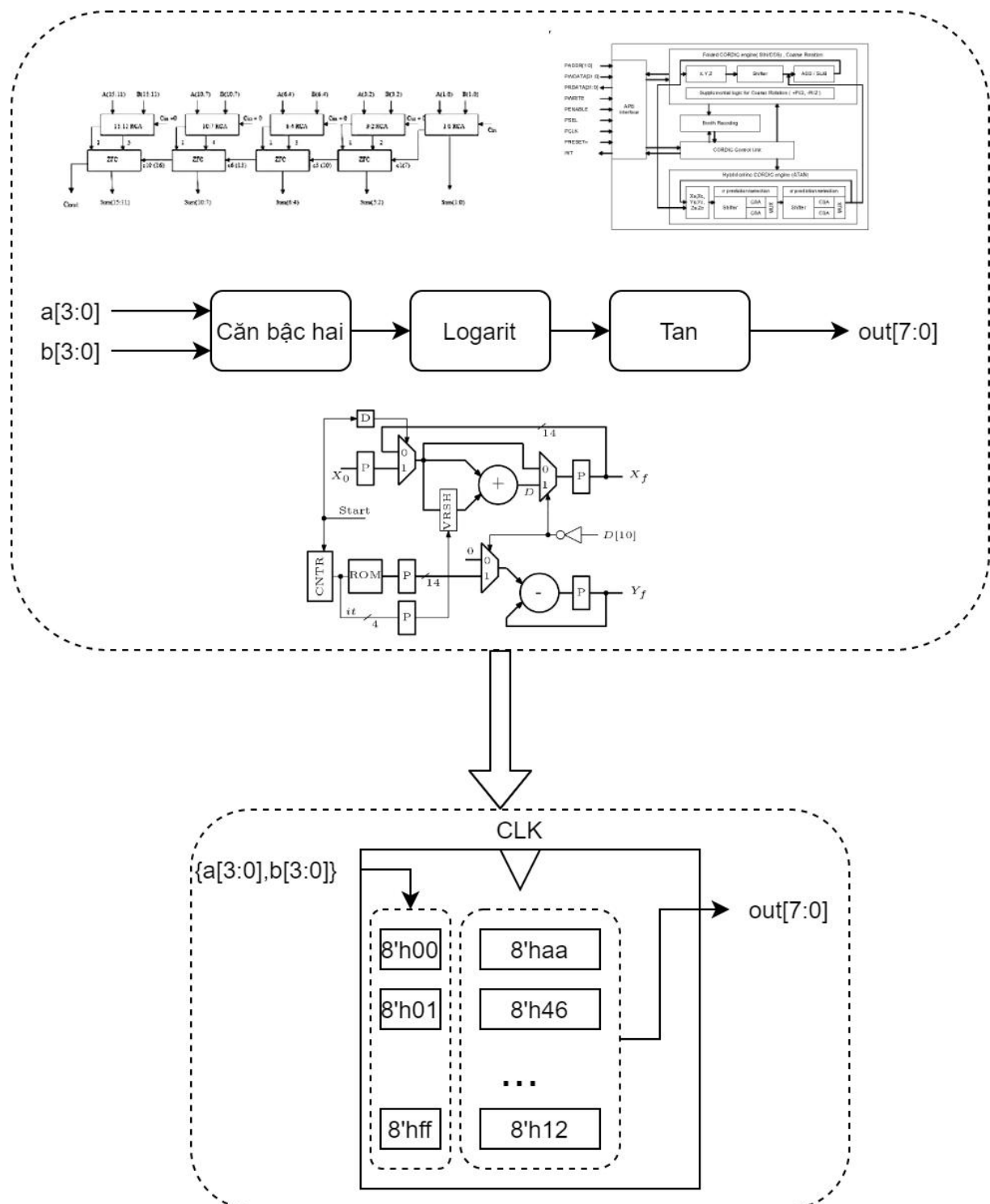
Trong thiết kế vi mạch, bộ nhớ thường được tích hợp sẵn dưới dạng các IP. Người dùng sẽ mua lại các IP này và kết hợp vào trong thiết kế của mình. Tuy nhiên, trong một số trường hợp bộ nhớ có kích thước nhỏ hay chức năng đặc thù, người thiết kế vi mạch vẫn hoàn toàn có khả năng tự lập trình nên chúng.

Trong trường hợp ngược lại, người thiết kế cũng cần tìm hiểu kỹ cách thức sử dụng IP, bởi nhu cầu sử dụng dữ liệu cho thiết kế và IP có thể không hoàn toàn tương thích với nhau, về tốc độ truy xuất, kích thước dữ liệu cần truy xuất,... và do đó cần điều chỉnh cho hợp lý.



Hình 6.2. Phân loại bộ nhớ.

Một trong những ứng dụng quan trọng của bộ nhớ là bảng tra (Look-Up Table hay LUT). Bảng tra là một hay nhiều mảng bộ nhớ lưu trữ sẵn kết quả tính toán của một tác vụ nào đó. Thay vì thiết kế phần cứng phức tạp, trong trường hợp số lượng kết quả cần dùng không quá lớn, người thiết kế hoàn toàn có thể dùng bảng tra nhằm giảm đi mức độ phức tạp cũng như thời gian thực thi của thiết kế. Ví dụ (hình 6.3): người thiết kế cần một khối tính toán từ hai ngõ vào 4 bit, trong đó các khâu tính toán thông qua các hàm toán phức tạp như logarit, căn bậc hai, hàm lượng giác,... Thông qua bảng tra, người thiết kế có thể gói gọn lại thành một bộ nhớ có kích thước địa chỉ là 8 bit (kết hợp hai ngõ vào), cùng kích thước ô nhớ là kích thước mong muốn của kết quả khối tính toán. Như vậy, khối lượng một lần tính toán được giảm xuống thành một lần truy cập bộ nhớ, tăng tốc độ đáng kể. Bảng tra trong một số trường hợp lại tỏ ra thiếu hiệu quả, như khi khối lượng tính toán thấp không cần thiết rút gọn, hay khi số lượng khả năng cần tra (kích thước địa chỉ) quá lớn; cho nên người thiết kế cần lựa chọn sao cho phù hợp nhất với thiết kế của mình.



Hình 6.3. Ví dụ về sử dụng bảng tra trong thiết kế vi mạch.

6.1.4. Cài đặt TCL về máy ảo

Ngôn ngữ TCL sử dụng trong bài lab này là phiên bản 8.4.20, ta có thể tải miễn phí ở trang chủ của TCL. Sau khi tải về, ta giải nén thư mục .zip ở bên trong máy ảo và để ở ngoài Desktop.

Sau khi giải nén, di chuyển vào thư mục TCL và cấu hình quá trình cài đặt bằng cách chạy lệnh **cd TCL8.4.20/unix** và **./configure** (như hình 6.4 và kết quả trả về như hình 6.5).

```
[albert@localhost Desktop]$ cd tcl8.4.20/unix
[albert@localhost unix]$ ./configure
```

Hình 6.4. Chạy các lệnh.

```
checking for size_t... (cached) yes
checking for uid_t in sys/types.h... (cached) yes
checking for socklen_t... (cached) yes
checking for opendir... (cached) yes
checking union wait... (cached) yes
checking for strncasecmp... (cached) yes
checking for gettimeofday... (cached) yes
checking for gettimeofday declaration... (cached) present
checking whether char is unsigned... (cached) no
checking signed char declarations... (cached) yes
checking for a putenv() that copies the buffer... (cached) no
checking for langinfo.h... (cached) yes
checking whether to use nl_langinfo... (cached) yes
checking for fts... (cached) no
checking for sys/ioctl.h... (cached) yes
checking for sys/filio.h... (cached) no
checking system version... (cached) Linux-2.6.32-754.27.1.el6.x86_64
checking FIONBIO vs. O_NONBLOCK for nonblocking I/O... O_NONBLOCK
checking whether to enable DTrace support... no
checking whether the cpuid instruction is usable... (cached) yes
creating ./config.status
creating Makefile
creating dltest/Makefile
creating tclConfig.sh
[albert@localhost unix]$ █
```

Hình 6.5. Kết quả sau khi chạy lệnh ./configure

Sau khi cấu hình xong, ta biên dịch TCL bằng lệnh **make**, kết quả thể hiện trong hình 6.6:

```
[albert@localhost unix]$ make
gcc -c -O2 -pipe -DTCL_DBGX= -Wall -fno-strict-aliasing -fPIC -I. -I/home/albert/Desktop/tcl8.4.20/unix -DHAVE_LIMITS_H=1 -DHAVE_UNISTD_H=1 -DHAVE_SYS_PARAM_H=1 -DPEEK_XCLOSEIM=1 -DHAVE_CAST_T
_INT_IS_LONG=1 -DHAVE_GETCWD=1 -DHAVE_OPENDIR=1 -DHAVE_STRSTR=1 -DHAVE_STRTOL=1 -DHAVE_STRTOLL=1 -DHAVE_TERMIOS=1 -DHAVE_SYS_TIME_H=1 -DTIME_WITH_SYS_TIME=1 -DHAVE_TM_ZONE=1 -DHAVE_GMTIME_R=1 -I
VAR=1 -DHAVE_ST_BLKSIZE=1 -DSTDC_HEADERS=1 -DHAVE_SIGNED_CHAR=1 -DHAVE_LANGINFO=1 -DHAVE_SYS_I
-DTCL_LIBRARY="/usr/local/lib/tcl8.4\" \
-DTCL_PACKAGE_PATH="/usr/local/lib \"\" \
/home/albert/Desktop/tcl8.4.20/unix/./unix/tclUnixInit.c
rm -f libtcl8.4.so
gcc -shared -O2 -pipe -DTCL_DBGX= -Wl,--export-dynamic -o libtcl8.4.so regcomp.o regex.o
tclBinary.o tclCkalloc.o tclClock.o tclCmdAH.o tclCmdIL.o tclCmdMZ.o tclCompCmds.o tclCompExpr
nt.o tclExecute.o tclFCmd.o tclFileName.o tclGet.o tclHash.o tclHistory.o tclIndexObj.o tclInte
l.o tclLink.o tclListObj.o tclLiteral.o tclLoad.o tclMain.o tclNamesp.o tclNotify.o tclObj.o t
tclPosixStr.o tclPreserve.o tclProc.o tclRegexp.o tclResolve.o tclResult.o tclScan.o tclString
tubInit.o tclStubLib.o tclTimer.o tclUtf.o tclUtil.o tclVar.o tclUnixChan.o tclUnixEvent.o tcl
lUnixTime.o tclUnixInit.o tclUnixThrd.o tclUnixCompat.o tclUnixNotfy.o tclLoadDl.o -ldl -l:
gcc -O2 -pipe -DTCL_DBGX= -Wl,--export-dynamic tclAppInit.o -L/home/albert/Desktop/tcl8.4.20
-Wl,-rpath,/usr/local/lib -o tclsh
[albert@localhost unix]$
```

Hình 6.6. Kết quả sau khi chạy lệnh **make**

Sau đó ta cài đặt TCL vào hệ thống bằng lệnh **sudo make install** với [sudo] password là hoangtrang. Lúc này TCL sẽ mặc định nằm trong thư mục **/usr/local/bin/**

```
[albert@localhost unix]$ sudo make install
[sudo] password for albert:
Installing libtcl8.4.so to /usr/local/lib/
Installing tclsh as /usr/local/bin/tclsh8.4
Installing tclConfig.sh to /usr/local/lib/
Installing libtclstub8.4.a to /usr/local/lib/
Installing header files
Installing library files to /usr/local/lib/tcl8.4
Installing library platform directory
Installing library http1.0 directory
Installing library http2.5 directory
Installing library opt0.4 directory
Installing library msgcat1.3 directory
Installing library tcltest2.2 directory
Installing library encoding directory
Installing and cross-linking top-level (.1) docs
Installing and cross-linking C API (.3) docs
Installing and cross-linking command (.n) docs
[albert@localhost unix]$
```

Hình 6.7. Kết quả sau khi **sudo make install**

Để sử dụng TCLsh từ bất kỳ đâu, ta cần thêm đường dẫn tới tệp thực thi TCLsh vào biến môi trường PATH. Ta tiếp tục truy cập bằng câu lệnh **gedit ~/.bashrc** và bổ sung dòng **export PATH=\$PATH:/usr/local/bin** vào tệp **.bashrc**. Sau đó dùng lệnh **source ~/.bashrc** để áp dụng thay đổi

```

export VCS_HOME=/home/albert/MyPrograms/synopsys/M-2017.03-SP2
export VCS_TARGET_ARCH=amd64
export VCS_ARCH_OVERRIDE=linux

export PATH=$PATH:/home/albert/MyPrograms/synopsys/M-2016.12/bin
export PATH=$PATH:/home/albert/MyPrograms/synopsys/M-2016.12/admin/install/lc/bin
export LC_HOME=/home/albert/MyPrograms/synopsys/M-2016.12
export PATH=$PATH:/home/albert/MyPrograms/synopsys/M-2017.03-SP2/bin/
# verdi
export PATH=$PATH:/home/albert/MyPrograms/synopsys/Verdi3_L-2016.06-1/bin

alias LoadDC='lmgrd -c $LM_LICENSE_FILE'

alias vsim='vsim -64'
alias vlog='vlog -64'
alias vcom='vcom -64'
alias vcs='vcs -full64'
alias dve='dve -full64'
export PATH=$PATH:/usr/local/bin

```

Hình 6.8. Nội dung file .bashrc

Sau đó, ngoài terminal ta sẽ tạo một liên kết tượng trưng (symlink) để dễ dàng gọi nó

```
sudo ln -s /usr/local/bin/TCLsh8.4 /usr/local/bin/TCLsh
```

Vậy là hoàn tất cài đặt TCL vào máy ảo.

6.1.5 Thiết kế bộ nhớ thông qua script TCL

Tài liệu này sẽ không mô tả một cách chi tiết và đầy đủ cách sử dụng TCL trong quá trình viết script mà chỉ ứng dụng nó trong việc xây dựng một script tạo ra bộ nhớ dưới dạng ngôn ngữ Verilog.

Script mà tài liệu nhắc đến sẽ tương tác với hai tệp khác: một tệp định dạng .txt chứa dữ liệu của bộ nhớ mong muốn tạo, và một tệp định dạng .v (Verilog) mô tả phần cứng của bộ nhớ có nội dung như trên, và được tạo ra tự động bởi script TCL.

Hình 6.9 mô tả script TCL thực hiện toàn bộ quá trình trên. Biến dùng trong script TCL được truyền vào thông qua tham số khi gọi thủ tục **main {input_file output_file}**. Ví dụ, trong trường hợp này, khi người dùng nhập lệnh

```
TCLsh memory_gen.TCL mem.txt memory_mult.v
```

trên Terminal, script **memory_gen.TCL** sẽ được thực thi, và biến **input_file** sẽ nhận giá trị **mem.txt**, còn biến **output_file** sẽ nhận giá trị **memory_mult.v**.

Các biến **input_path** và **output_path** được thiết lập để chứa đường dẫn đến các file **input** và **output** tương ứng trong thư mục **/home/albert/Desktop/Lab4_Mem/02_rtl/**. Các lệnh **puts** được sử dụng để

ghi nội dung vào file output, bao gồm cả việc viết header của module Verilog và các dòng mã cần thiết để mô tả phần cứng của bộ nhớ. Vòng lặp **while** giúp đọc từng dòng dữ liệu trong file input và gán vào vị trí bộ nhớ tương ứng thông qua biến **count**. Như vậy, khi xây dựng file input, người thiết kế cần tổ chức dữ liệu theo từng dòng, mỗi dòng chứa dữ liệu của ô nhớ có thứ tự tương ứng (tính từ 0), thứ tự tăng dần với bước nhảy là 1.

Ví dụ cụ thể về việc sử dụng script này sẽ tạo ra một module Verilog dựa trên dữ liệu từ file **mem.txt** và ghi vào file **memory_mult.v** như đã mô tả.

```

memory_gen.tcl x
proc main {input_file output_file} {
    # Set the file paths
    set input_path "/home/albert/Desktop/Lab4_Mem/02_rtl/$input_file"
    set output_path "/home/albert/Desktop/Lab4_Mem/02_rtl/$output_file"

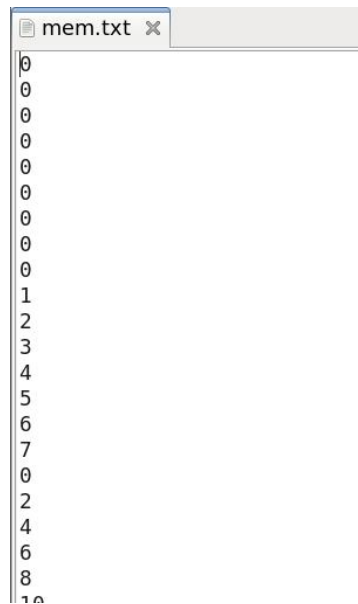
    # Open the input and output files
    set in_mem [open $input_path r]
    set out_mem [open $output_path w]
    # Write the Verilog module header
    puts $out_mem "module memory_mult(clk, rst, addr, data);"
    puts $out_mem "parameter DATA_WIDTH = 32;"
    puts $out_mem "input clk, rst;"
    puts $out_mem "input \[5:0\] addr;"
    puts $out_mem "output reg \[DATA_WIDTH-1:0\] data;"
    puts $out_mem "reg \[DATA_WIDTH-1:0\] data_mem \[0:100\];"
    puts $out_mem "always @ (posedge clk) begin"
    puts $out_mem "\tif (rst) begin"
    set count 0
    while {[gets $in_mem line] >= 0} {
        # Trim the newline character
        set line [string trim $line]

        # Handle empty lines and write the memory initialization line to the output file
        if {[string length $line] > 0} {
            # Directly output the line without special character escaping
            puts $out_mem "\t\tdata_mem\[ $count\] = $line;"
            incr count
        }
    }

    puts $out_mem "\tend"
    puts $out_mem "\telse"
    puts $out_mem "\t\tdata <= data_mem\[addr\];"
    puts $out_mem "end"
    puts $out_mem "endmodule"
    # Close the files
    close $in_mem
    close $out_mem
}
# Execute the main procedure with input and output file arguments
main "mem.txt" "memory_mult.v"

```

Hình 6.9. Nội dung script TCL tạo bộ nhớ



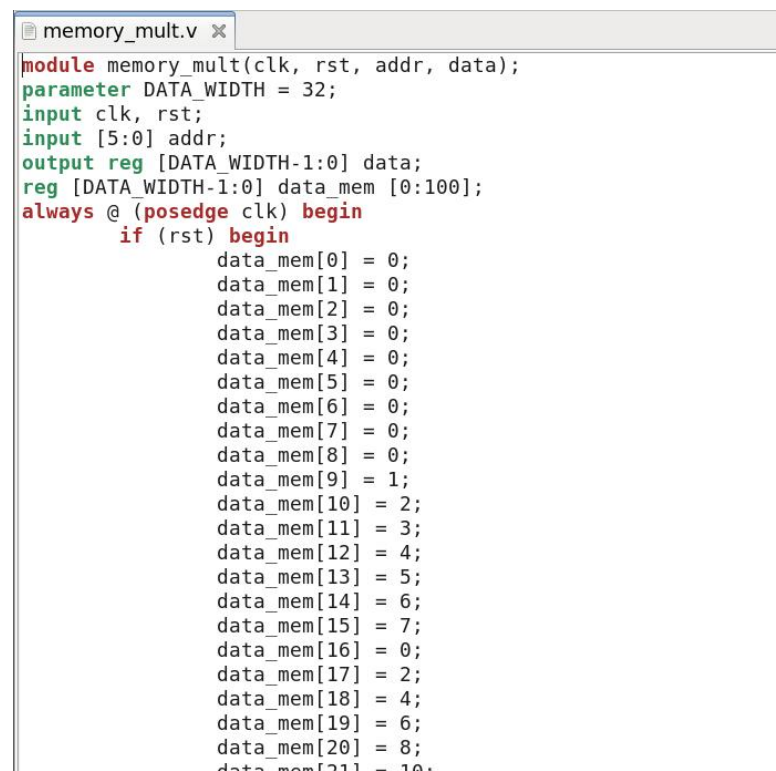
```

0
0
0
0
0
0
0
0
0
0
1
2
3
4
5
6
7
0
2
4
6
8
10

```

Hình 6.10. Ví dụ file input cho script TCL.

Kết quả chạy script Perl được mô tả trong hình 6.10. Các dòng lệnh được tạo ra hoàn toàn giống theo mô tả trong script Perl, trong đó nội dung bộ nhớ được thiết lập theo tín hiệu reset.



```

module memory_mult(clk, rst, addr, data);
parameter DATA_WIDTH = 32;
input clk, rst;
input [5:0] addr;
output reg [DATA_WIDTH-1:0] data;
reg [DATA_WIDTH-1:0] data_mem [0:100];
always @ (posedge clk) begin
    if (rst) begin
        data_mem[0] = 0;
        data_mem[1] = 0;
        data_mem[2] = 0;
        data_mem[3] = 0;
        data_mem[4] = 0;
        data_mem[5] = 0;
        data_mem[6] = 0;
        data_mem[7] = 0;
        data_mem[8] = 0;
        data_mem[9] = 1;
        data_mem[10] = 2;
        data_mem[11] = 3;
        data_mem[12] = 4;
        data_mem[13] = 5;
        data_mem[14] = 6;
        data_mem[15] = 7;
        data_mem[16] = 0;
        data_mem[17] = 2;
        data_mem[18] = 4;
        data_mem[19] = 6;
        data_mem[20] = 8;
        data_mem[21] = 10;

```

Hình 6.11. Ví dụ file output cho script TCL.

6.2 Thực hiện thiết kế

Trong bài thực hành - thí nghiệm này, mục tiêu người đọc hướng đến là

thiết kế hai bộ nhân hai số 3 bit, trong đó bộ thứ nhất (mult_1) nhân theo cách truyền thống (nhân từng bit sau đó dịch và cộng lại), bộ thứ hai dùng bảng tra (tra kết quả theo bộ nhớ được tạo ra bằng script TCL). Cụ thể:

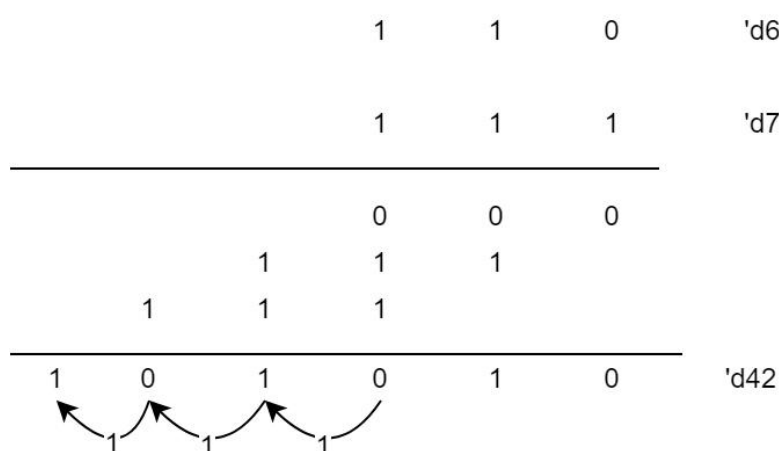
Ngõ vào:

- Tín hiệu clock tên clk, độ rộng 1 bit.
- Tín hiệu reset tên rst, độ rộng 1 bit.
- Các toán hạng vào tên lần lượt là in1 và in2, là hai số 3 bit cần tính tích.

Ngõ ra: Tích của hai toán hạng ngõ vào tên out, có độ rộng 6 bit.

6.2.1 Thiết kế cấp độ hệ thống

Bộ nhân thứ nhất được thiết kế theo nguyên tắc nhân hai số đã học trong các bậc giáo dục cơ sở. Trong đó thừa số thứ nhất được tách thành các chữ số riêng biệt, các chữ số này được đem đi nhân lần lượt với thừa số thứ hai. Chữ số có vị trí càng cao thì sẽ được dịch trái càng nhiều. Đối với số nhị phân, phép nhân hai số 1 bit chính là phép logic AND. Hình 6.7 là ví dụ của nhân 2 số 3 bit theo mô tả như trên.



Hình 6.12. Ví dụ nhân hai số 3 bit thông qua dịch bit và cộng logic.

Bộ nhân thứ hai sẽ sử dụng bảng tra, trong đó kết quả phép nhân được lưu sẵn trong một bộ nhớ. Để truy cập ô nhớ chính xác tương ứng, địa chỉ từng ô nhớ sẽ được xây dựng bằng cách nối tiếp các toán hạng ngõ vào. Ví dụ: Khi cần tính tích của hai số 3'b110 và 3'b111, người thiết kế sẽ hướng dẫn bộ nhân truy cập vào bộ nhớ với địa chỉ ô nhớ là 6'b110111, và trong ô nhớ này cần lưu trữ giá trị 6'b101010 (=d42). Như vậy, bộ nhớ cần xây dựng phải có kích thước tối thiểu là 64 ô nhớ, mỗi ô nhớ có kích thước tối thiểu là 6 bit. Thông qua script TCL ở trên, người thiết kế có thể dễ dàng sửa đổi các thông số này thông qua nội dung các lệnh puts.

Có thể thấy rằng, sử dụng bảng tra trong thiết kế có vẻ rất tối ưu khi mà chỉ cần một lần truy cập bộ nhớ là có được kết quả mong muốn mà không cần thông qua tính toán phức tạp. Dù vậy, tại sao con người vẫn cần thiết kế các khối tính toán. Câu trả lời sẽ được rút ra ở cuối bài thực hành - thí nghiệm này.

6.2.2 Tạo dựng môi trường và các chuẩn bị khác

Phần này được thực hiện tương tự các bài thực hành-thí nghiệm trước, chỉ thêm phần sửa đổi script TCL cho phù hợp theo yêu cầu của thiết kế.

6.2.3 Mô tả thiết kế bằng Verilog

Để chuẩn bị cho bộ nhân thứ nhất, người thiết kế cần chuẩn bị thiết kế bộ cộng Full Adder trước, dành cho các phép cộng sau khi AND logic các bit. Bộ cộng này đã được thực hiện trong bài thực hành - thí nghiệm đầu tiên nên tài liệu xin phép không nhắc lại. Hình 6.13 là mô tả bộ nhân thứ nhất. Khi cần cộng nhiều hơn 3 số 1 bit, người thiết kế tách thành nhiều bộ cộng Full Adder cộng riêng lẻ, và lưu ý ứng với mỗi một bộ cộng như vậy, cần đem 1 bit nhớ lên bậc bit tiếp theo để cộng

```
module mult_1(clk, rst, in1, in2, out);
input clk, rst;
input [2:0] in1, in2;
output reg [5:0] out;
wire [5:0] S;
wire C1,C2_1,C2_2,C3_1,C3_2;

assign S[0] = in1[0]&in2[0];
full_adder FA_1(in1[1]&in2[0],in1[0]&in2[1],1'b0,S[1],C1);
full_adder FA_21(in1[2]&in2[0],in1[1]&in2[1],in1[0]&in2[2],S2_1,C2_1);
full_adder FA_22(S2_1,C1,1'b0,S[2],C2_2);
full_adder FA_31(in1[2]&in2[1],in1[1]&in2[2],C2_1,S3_1,C3_1);
full_adder FA_32(S3_1,C2_2,1'b0,S[3],C3_2);
full_adder FA_4(in1[2]&in2[2],C3_1,C3_2,S[4],S[5]);

always @(posedge clk) begin
if (rst)
    out <= 6'b0;
else
    out <= S;
end

endmodule
```

Hình 6.13. Mô tả phần cứng của bộ nhân thứ nhất.

Đối với bộ nhân thứ hai, người thiết kế cần chuẩn bị trước nội dung bảng tra, theo đúng thứ tự từ 6'b000000 đến 6'b111111. Trong phần cứng của bộ nhân thứ hai, bộ nhớ sẽ được instance, với ngõ vào địa chỉ được gán từ hai toán hạng ngõ vào theo mô tả ở phần trên. Hình 6.14 là mô tả bộ nhân thứ hai.

```

module mult_2(clk, rst, in1, in2, out);
input clk, rst;
input [2:0] in1, in2;
output [5:0] out;
wire [5:0] addr;

assign addr = {in1,in2};
memory_mult #6 LUT(clk, rst, addr, out);

endmodule

```

Hình 6.14. Mô tả phần cứng của bộ nhân thứ hai.

6.2.4 Thực hiện viết testbench

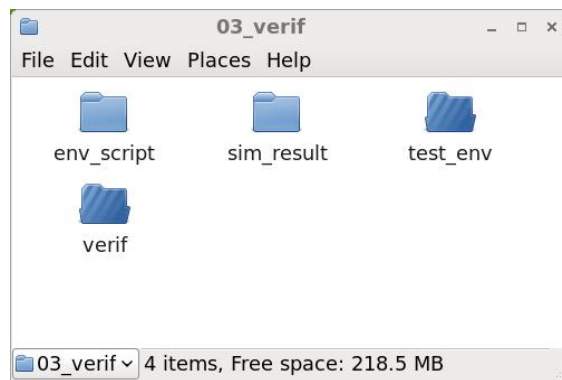
Bảng 6.1: Mô tả bước kiểm tra thiết kế cấp độ RTL (Verification).

Đầu vào	Đầu ra	Các công đoạn
<ul style="list-style-type: none"> - Các file mã lập trình đuôi .v - Các file testbench cũng đuôi .v hay .sv (verilog) - File đầu vào (input.txt) chứa các giá trị dùng để kiểm tra 	<ul style="list-style-type: none"> - Các báo cáo (report) về quá trình mô phỏng - Các file dạng sóng (waveform) nhằm kiểm tra mô phỏng thiết kế - File đầu ra (output.txt) chứa các giá trị đầu vào và kết quả của hệ thống - Hệ thống sẽ hiện “1” nếu các file đầu ra từ các môi trường kiểm tra giống nhau, ngược lại sẽ hiện “0” 	<ul style="list-style-type: none"> - Tạo file chứa các giá trị đầu vào (input.txt) - Xây dựng môi trường cho việc kiểm tra trong verilog và Python - Kiểm tra lỗi cú pháp của RTL (các file .v) và testbench - Chạy mô phỏng và xuất ra dạng sóng - Kiểm tra các file báo cáo (report) và dạng sóng nhằm đảm bảo mô phỏng RTL sạch lỗi - So sánh kết quả của các file đầu ra của verilog và Python

Bảng 6.1 là mô tả khái quát của bước kiểm tra thiết kế cấp độ RTL (Verification).

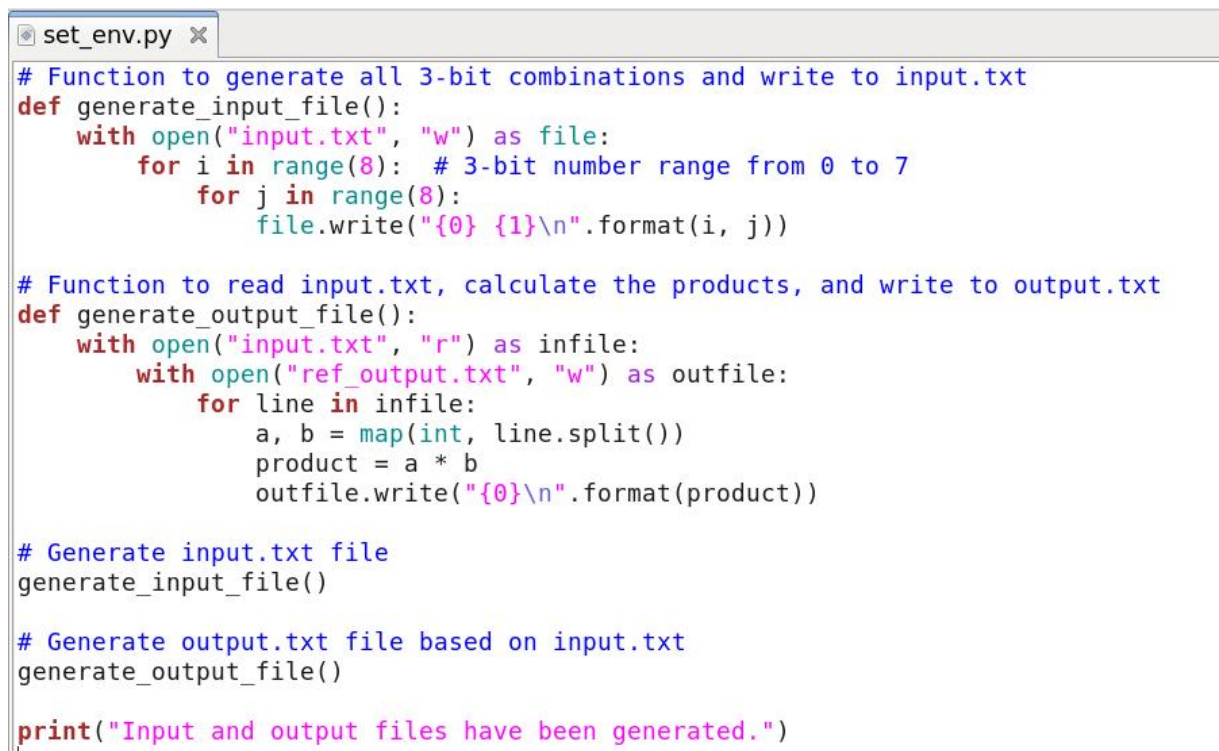
Việc tiếp theo cần làm là xây dựng môi trường để kiểm tra tính đúng đắn của các file RTL đã thiết kế ở trên. Các file cần thiết cho việc thực hiện kiểm định bao gồm một file đầu vào (input.txt), một file đầu ra dùng để tham chiếu (ref_output.txt) và một file đầu ra từ code verilog (output.txt). File input.txt và ref_output.txt sẽ được tạo bằng ngôn ngữ (thông qua C/C++, python,..). Còn file output.txt sẽ được tạo ra sau khi chạy testbench.

Đầu tiên cần tạo thư mục cho quá trình test



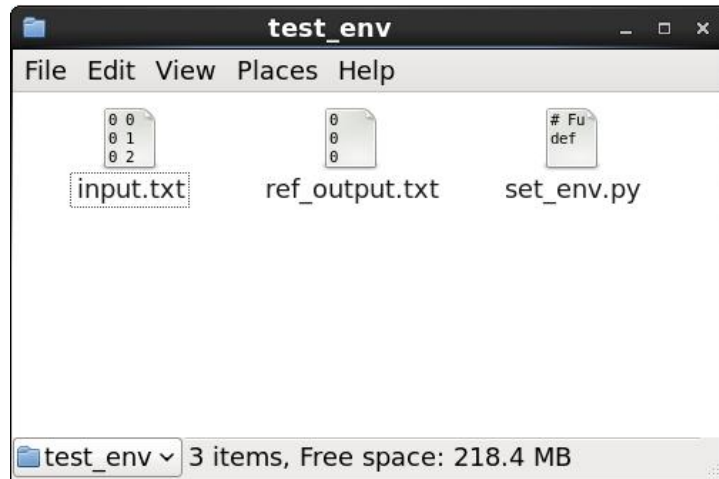
Hình 6.15. Cấu trúc thư mục **03_verif**.

Trong thư mục **test_env** ta sẽ có một file code python để tạo tất cả các giá trị đầu vào của bộ nhân 3-bit.



Hình 6.16. Lập trình tạo file **input.txt**, **ref_output.txt** bằng Python

Sau đó, tiến hành chạy file Python bằng câu lệnh **python set_env.py**. Hình 6.16 là kết quả sau khi chạy.



Hình 6.17. Kết quả sau khi chạy *set_env.py*.

Trong đó, file *input.txt* là các giá trị đầu vào với mỗi hàng là một cặp giá trị (hình 6.18) và *ref_output.txt* là các kết quả đầu ra sẽ được dùng làm tham chiếu (hình 6.19)

0 0	3 0	
0 1	3 1	
0 2	3 2	
0 3	3 3	
0 4	3 4	
0 5	3 5	
0 6	3 6	
0 7	3 7	6 0
1 0	4 0	6 1
1 1	4 1	6 2
1 2	4 2	6 3
1 3	4 3	6 4
1 4	4 4	6 5
1 5	4 5	6 6
1 6	4 6	6 7
1 7	4 7	7 0
2 0	5 0	7 1
2 1	5 1	7 2
2 2	5 2	7 3
2 3	5 3	7 4
2 4	5 4	7 5
2 5	5 5	7 6
2 6	5 6	7 7
2 7	5 7	

Hình 6.18. Nội dung file *input.txt*

0	0	
0	3	
0	6	
0	9	
0	12	
0	15	
0	18	
0	21	
0	0	
1	4	0
2	8	6
3	12	12
4	16	18
5	20	24
6	24	30
7	28	36
0	0	0
2	5	7
4	10	14
6	15	21
8	20	28
10	25	35
12	30	42
14	35	49

Hình 6.19. Nội dung file *ref_output.txt*

Tiếp theo, ta sẽ tạo môi trường test ngay trong máy ảo (trong file testtop.v hình 6.20). Trong file testtop.v này, ta sẽ tạo file output.txt từ file input.txt ở trên, đồng thời xuất ra kết quả so sánh giữa hai file output.txt và ref_output.txt.

```

module combined_testbench;
    reg clk;
    reg rst;
    reg [2:0] in1, in2;
    wire [5:0] out;
    reg [5:0] out_read; // Register to read output file data

    integer in_file, out_file, ref_file, report_file;
    integer scan_status_in, scan_status_out, scan_status_ref;
    reg [5:0] ref_out;
    reg all_match;
    reg [7:0] line_number;

    // Instantiate the multiplier
    mult_2| UUT (
        .clk(clk),
        .rst(rst),
        .in1(in1),
        .in2(in2),
        .out(out)S
    );

    // Clock generation
    initial begin
        clk = 0;
        forever #5 clk = ~clk; // 100 MHz clock
    end

```



```

// Test procedure
initial begin
    // Open the input file
    in_file = $fopen("/home/albert/Desktop/Lab4_Mem/03_verif/test_env/input.txt", "r");
    if (in_file == 0) begin
        $display("Error: Could not open input file.");
        $finish;
    end

    // Open the output file
    out_file = $fopen("/home/albert/Desktop/Lab4_Mem/03_verif/test_env/output.txt", "w");
    if (out_file == 0) begin
        $display("Error: Could not open output file.");
        $finish;
    end

    // Open the reference output file
    ref_file = $fopen("/home/albert/Desktop/Lab4_Mem/03_verif/test_env/ref_output.txt", "r");
    if (ref_file == 0) begin
        $display("Error: Could not open reference output file.");
        $finish;
    end

    // Open the report file
    report_file = $fopen("/home/albert/Desktop/Lab4_Mem/03_verif/test_env/report.txt", "w");
    if (report_file == 0) begin
        $display("Error: Could not open report file.");
        $finish;
    end

    // Reset the DUT
    rst = 1;
    #10;
    rst = 0;

    line_number = 1;

    // Initialize all_match to true
    all_match = 1;

    // Read input values and apply them to the DUT, write outputs
    while (!$feof(in_file)) begin
        scan_status_in = $fscanf(in_file, "%d %d\n", in1, in2);
        if (scan_status_in != 2) begin
            $display("Error: Failed to read input values at line %d.", line_number);
            $finish;
        end

        #10; // wait for a few clock cycles for the output to settle

        $fwrite(out_file, "%d\n", out);
        line_number = line_number + 1;
    end

    // Close the input and output files
    $fclose(in_file);
    $fclose(out_file);

```

```

// Reset line_number for comparison
line_number = 1;

// Re-open the output file for reading
out_file = $fopen("/home/albert/Desktop/Lab4_Mem/03_verif/test_env/output.txt", "r");
if (out_file == 0) begin
    $display("Error: Could not re-open generated output file.");
    $finish;
end

// Read and compare outputs
while (!$feof(out_file) && !$feof(ref_file)) begin
    scan_status_out = $fscanf(out_file, "%d\n", out_read);
    scan_status_ref = $fscanf(ref_file, "%d\n", ref_out);
    if (scan_status_out != 1 || scan_status_ref != 1) begin
        $display("Error: Failed to read output values at line %d.", line_number);
        all_match = 0;
        $finish;
    end

    // Compare the output with the reference output
    if (out_read != ref_out) begin
        $display("Mismatch at line %d: expected %d, got %d", line_number, ref_out, out_read);
        $fwrite(report_file, "Mismatch at line %d: expected %d, got %d\n", line_number, ref_out, out_read);
        all_match = 0;
    end
    line_number = line_number + 1;
end

// Write the test environment report
if (all_match)
    $fwrite(report_file, "1\n");
else
    $fwrite(report_file, "0\n");

// Close files
$fclose(out_file);
$fclose(ref_file);
$fclose(report_file);

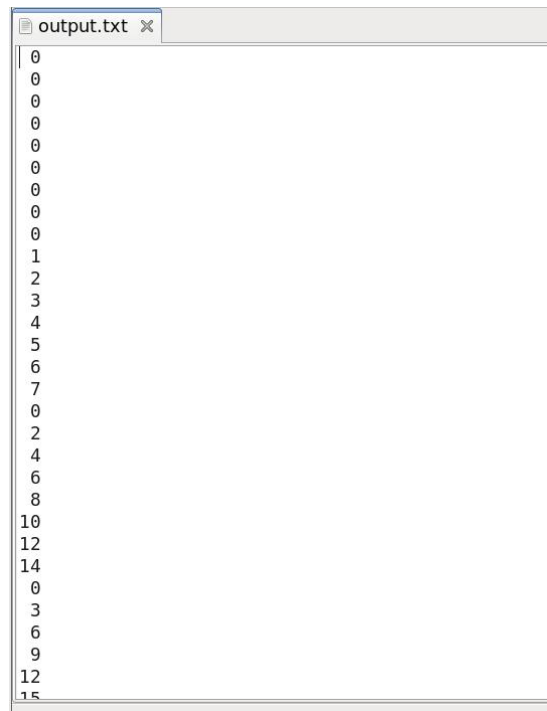
$display("Test and comparison completed. Outputs written to output.txt and report written to report.txt");
$finish;
end
endmodule

```

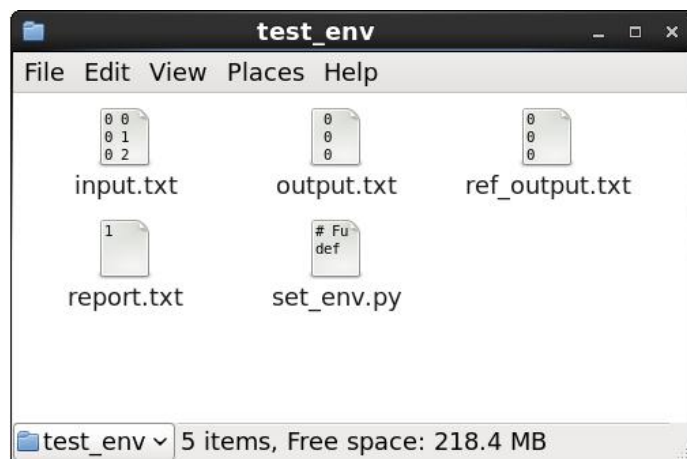
Hình 6.20. Nội dung file *testtop.v*

Ở file *testtop.v*, ta chọn bộ nhân thứ nhất *mult_1* để kiểm tra. Thực hiện chạy phân tích và mô phỏng RTL, kết quả xuất ra là file *output.txt* (hình 6.21) và kết quả so sánh (hình 6.22).

Sau khi chạy mô phỏng trong máy ảo, ta sẽ thu được kết quả như sau:



Hình 6.21. Kết quả sau khi mô phỏng RTL



Hình 6.22. Kết quả so sánh (*report.txt*)

6.2.5 Tổng hợp (Synthesis)

Bảng 6.2. là mô tả khái quát của bước Synthesis (tổng hợp) thiết kế.

Bảng 6.2: Mô tả bước Synthesis (tổng hợp) thiết kế.

Đầu vào	Đầu ra	Các công đoạn
<ul style="list-style-type: none"> - Các file mã lập trình đuôi .v - Các file thư viện đuôi .db nhằm cung cấp các cell tham khảo cho quá trình chuyển RTL sang các cell này. - Một số ràng buộc về timing, area hay power. 	<ul style="list-style-type: none"> - Các báo cáo (report) về quá trình tổng hợp. - Các file dùng cho các công đoạn tiếp theo trong quy trình thiết kế (.sdc, .sdf, .ddc,...) - File RTL (.v) đã tổng hợp (đã chuyển về các cell trong thư viện tham khảo), thường gọi là netlist. 	<ul style="list-style-type: none"> - Viết script cho quá trình tổng hợp, bao gồm việc lựa chọn thư viện, lựa chọn RTL, thêm các ràng buộc, tổng hợp và xuất kết quả. - Chạy script tổng hợp - Kiểm tra và sửa lỗi sau khi tổng hợp (nếu có).

Quá trình Synthesis diễn ra tương tự các bài thực hành-thí nghiệm trước (hình 6.23.)

```

dc_command x
#!/bin/bash

#===== SET DIRECTORY =====
set search_path "../lib"
#set search_path "/home/albert/Desktop/icc_lab/logical_lib"
set osearch_path [ concat $search_path \
]
#===== ADD THE LIBRARY =====
set target_library "NangateOpenCellLibrary_typical.db"
set link_library "* $target_library"
set synthesis_library standard.sldb

#===== ANALYSE DESIGN =====
analyze -format verilog "../rtl/full_adder.v"
analyze -format verilog "../rtl/mult_1.v"
#analyze -format verilog "../rtl/memory_mult.v"
#analyze -format verilog "../rtl/mult_2.v"
elaborate mult_1
current_design mult_1

#===== CONSTRAINT FOR DESIGN =====
create_clock -name clk -period 1000 {clk}
set_input_delay -max 10 -clock clk [all_inputs]
set_input_delay -min 1 -clock clk [all_inputs]
set_output_delay -max 10 -clock clk [all_outputs]
set_output_delay -min 1 -clock clk [all_outputs]
set_fanout_load 8 [all_outputs]

#===== SYNTHESIZE=====
compile_ultra

```

Hình 6.23. Chỉnh sửa thư viện tham khảo cho file dc_command.src.

```

lab_synth.netlist.v x
module mult_1 ( clk, rst, in1, in2, out );
  input [2:0] in1;
  input [2:0] in2;
  output [5:0] out;
  input clk, rst;
  wire N3, N4, N5, N6, N7, N8, n21, n22, n23, n24, n25, n26, n27, n28, n29,
        n30, n31, n32, n33, n34, n35, n36, n37, n38, n39, n40;

  DFF_X1 \out_reg[5] ( .D(N8), .CK(clk), .Q(out[5]) );
  DFF_X1 \out_reg[4] ( .D(N7), .CK(clk), .Q(out[4]) );
  DFF_X1 \out_reg[3] ( .D(N6), .CK(clk), .Q(out[3]) );
  DFF_X1 \out_reg[2] ( .D(N5), .CK(clk), .Q(out[2]) );
  DFF_X1 \out_reg[1] ( .D(N4), .CK(clk), .Q(out[1]) );
  DFF_X1 \out_reg[0] ( .D(N3), .CK(clk), .Q(out[0]) );
  NAND2_X1 U26 ( .A1(in2[0]), .A2(in1[0]), .ZN(n21) );
  NOR2_X1 U27 ( .A1(rst), .A2(n21), .ZN(N3) );
  AND2_X1 U28 ( .A1(in2[1]), .A2(in1[1]), .ZN(n25) );
  NAND3_X1 U29 ( .A1(in2[0]), .A2(in1[0]), .A3(n25), .ZN(n24) );
  INV_X1 U30 ( .A(n24), .ZN(n35) );
  AOI22_X1 U31 ( .A1(in2[0]), .A2(in1[1]), .B1(in1[0]), .B2(in2[1]), .ZN(n22)
  );
  NOR3_X1 U32 ( .A1(rst), .A2(n35), .A3(n22), .ZN(N4) );
  AND2_X1 U33 ( .A1(in1[2]), .A2(in2[0]), .ZN(n27) );
  AND2_X1 U34 ( .A1(in2[2]), .A2(in1[0]), .ZN(n26) );
  INV_X1 U35 ( .A(n28), .ZN(n23) );
  AOI221_X1 U36 ( .B1(n35), .B2(n28), .C1(n24), .C2(n23), .A(rst), .ZN(N5) );
  FA_X1 U37 ( .A(n27), .B(n26), .CI(n25), .CO(n34), .S(n28) );
  AND2_X1 U38 ( .A1(in2[2]), .A2(in1[1]), .ZN(n33) );
  AND2_X1 U39 ( .A1(in1[2]), .A2(in2[1]), .ZN(n32) );

```

Hình 6.24. Kết quả RTL sau khi Synthesis

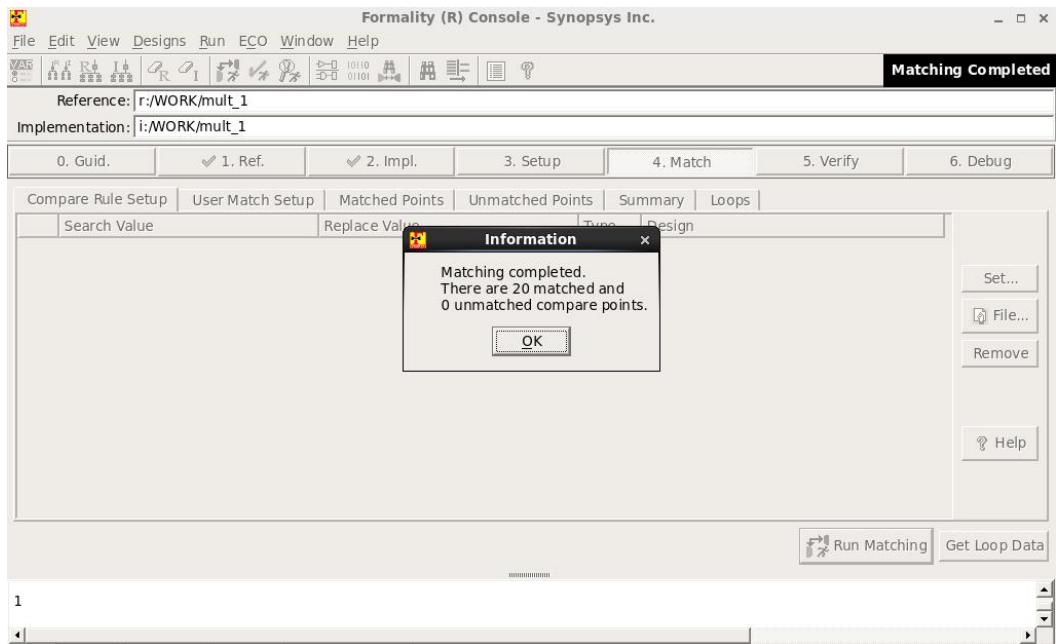
6.2.6 Kiểm tra netlist

Bảng 6.3. Mô tả bước kiểm tra netlist.

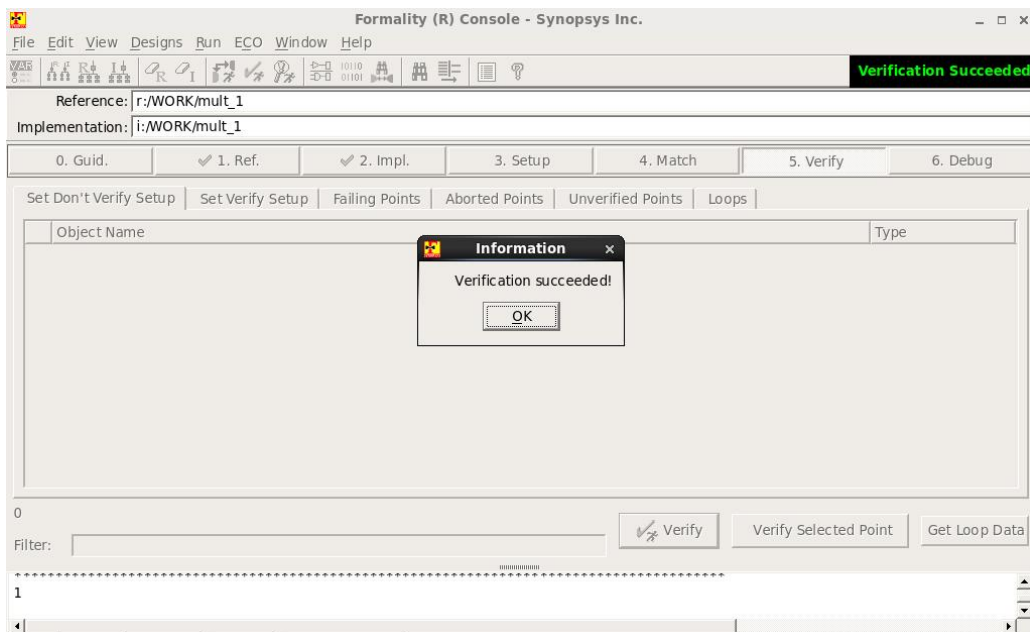
Đầu vào	Đầu ra	Các công đoạn
<ul style="list-style-type: none"> - Các file mã lập trình RTL đuôi .v - Các file RTL đã tổng hợp (netlist) đuôi .v 	<ul style="list-style-type: none"> - Các báo cáo (report) về quá trình Matching. - Các báo cáo (report) về quá trình Verify. 	<ul style="list-style-type: none"> - Lựa chọn các file thiết kế RTL. - Lựa chọn file RTL đã tổng hợp cùng thư viện tham khảo đi kèm. - Thực hiện kiểm tra Matching và sửa lỗi (nếu có). - Thực hiện kiểm tra Verify và sửa lỗi (nếu có).

Bảng 5.6. là mô tả khái quát của bước kiểm tra netlist.

Tương tự bài thực hành-thí nghiệm trước, quá trình kiểm tra netlist (bao gồm kiểm tra Matching và Verifying) cũng theo các bước đã mô tả thông qua phần mềm **Formality**. Lưu ý, khi chọn thư viện tham khảo cho netlist, cần chọn đúng thư viện dùng trong Synthesis. Kết quả kiểm tra trong hình 6.25 và 6.26.



Hình 6.25. Kiểm tra tương đồng giữa RTL trước và sau Synthesize.



Hình 6.26. Kiểm tra chức năng RTL sau Synthesize.

6.2.7 Phân tích thời gian tĩnh (STA)

Bảng 6.4. là mô tả khái quát của bước phân tích thời gian tĩnh (STA).

Bảng 6.4. Mô tả bước phân tích thời gian tĩnh (STA).

Đầu vào	Đầu ra	Các công đoạn
<ul style="list-style-type: none"> - Các file RTL đã tổng hợp (netlist) đuôi .v - Các ràng buộc đã sử dụng trong quá trình tổng hợp (file .sdc) - Các ràng buộc mới thêm vào. 	<ul style="list-style-type: none"> - Các báo cáo (report) về kiểm tra Design. - Các báo cáo (report) về kiểm tra STA. 	<ul style="list-style-type: none"> - Lựa chọn thư viện tham khảo đuôi .db (thư viện dùng trong Synthesis). - Liên kết thiết kế (file RTL đã tổng hợp). - Thêm các ràng buộc cho STA. - Thiết lập chế độ hoạt động. - Chạy kiểm tra STA và xuất các báo cáo (report). - Kiểm tra các báo cáo và sửa lỗi (nếu có).

Việc thực hiện STA ta thực hiện tương tự bài thực hành-thí nghiệm trước. File **sta_command.src** và kết quả kiểm tra timing được thể hiện trong hình 6.27 và hình 6.28.

```

sta_command
#!/bin/bash

##### SET DIRECTORY #####
set search_path "/home/hoangtrang/Desktop/icc_lab/logical_lib"
set osearch_path [ concat $search_path \
]

##### ADD THE LIBRARY #####
set target_library "NangateOpenCellLibrary_typical.db"
set link_library "* $target_library"

##### LINK DESIGN #####
read_verilog "../04_synth/report/lab_synth.netlist.v"
current_design mult_1
link

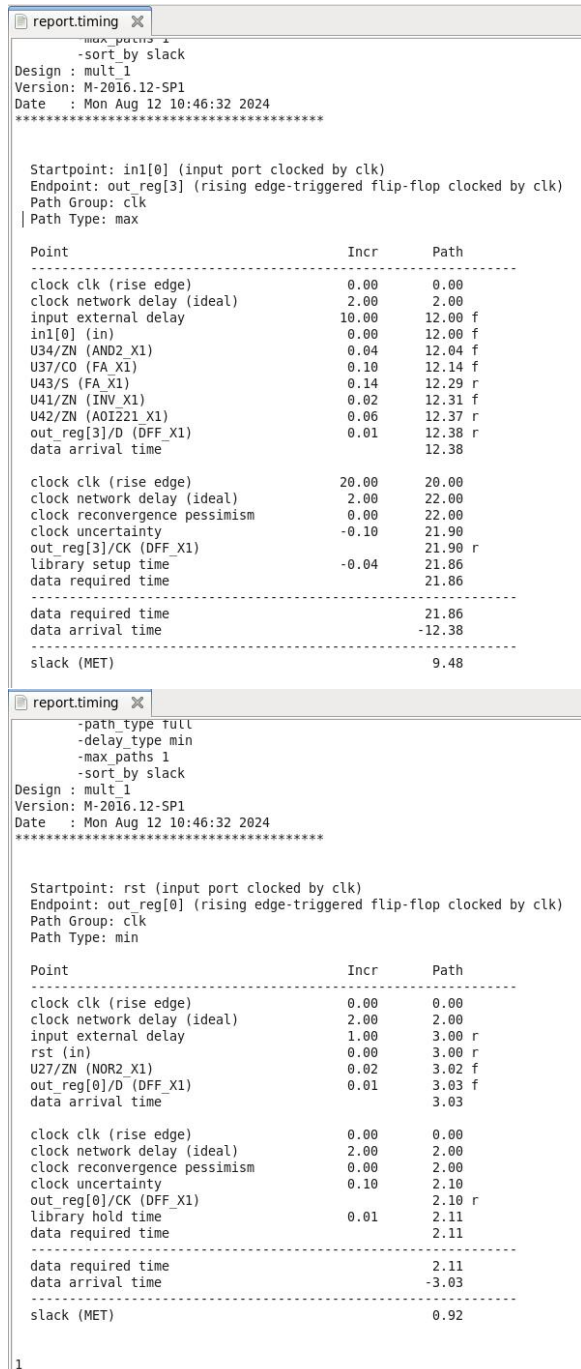
##### CONSTRAINT FOR STA #####
set_units -time ns -resistance kohm -capacitance pF -voltage V -current mA
set_max_area 0
create_clock -name clk -period 20 {clk}
set_clock_uncertainty 0.1 [get_clocks clk]
set_clock_latency 2 [get_clocks clk]
set_clock_transition -max -rise 0.1 [get_clocks clk]
set_clock_transition -max -fall 0.1 [get_clocks clk]
set_clock_transition -min -rise 0.1 [get_clocks clk]
set_clock_transition -min -fall 0.1 [get_clocks clk]
set_input_delay -max 10 -clock clk [all_inputs]
set_input_delay -min 1 -clock clk [all_inputs]
set_output_delay -max 10 -clock clk [all_outputs]
set_output_delay -min 1 -clock clk [all_outputs]
set_fanout_load 8 [all_outputs]

##### OPERATING CONDITIONS=====
set_operating_conditions -analysis_type on_chip_variation

##### REPORT DESIGN=====
report_design > report/report_design/report.design
report_net > report/report_design/report.net
report_cell > report/report_design/report.cell
report_hierarchy > report/report_design/report.hierarchy
report_clock > report/report_design/report.clock

```

Hình 6.27. Nội dung file sta_command.src.



Hình 6.28. Kết quả kiểm tra timing trong bước STA.

6.2.8 Place & Route

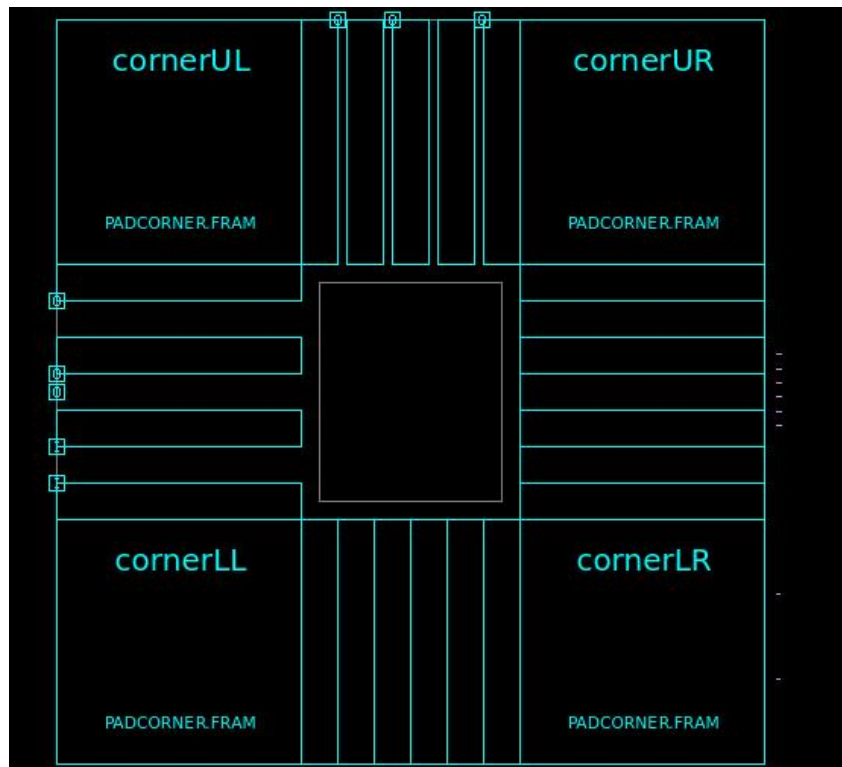
Bảng 6.5. là mô tả khái quát của bước Place & Route.

Bảng 6.5. Mô tả bước Place & Route.

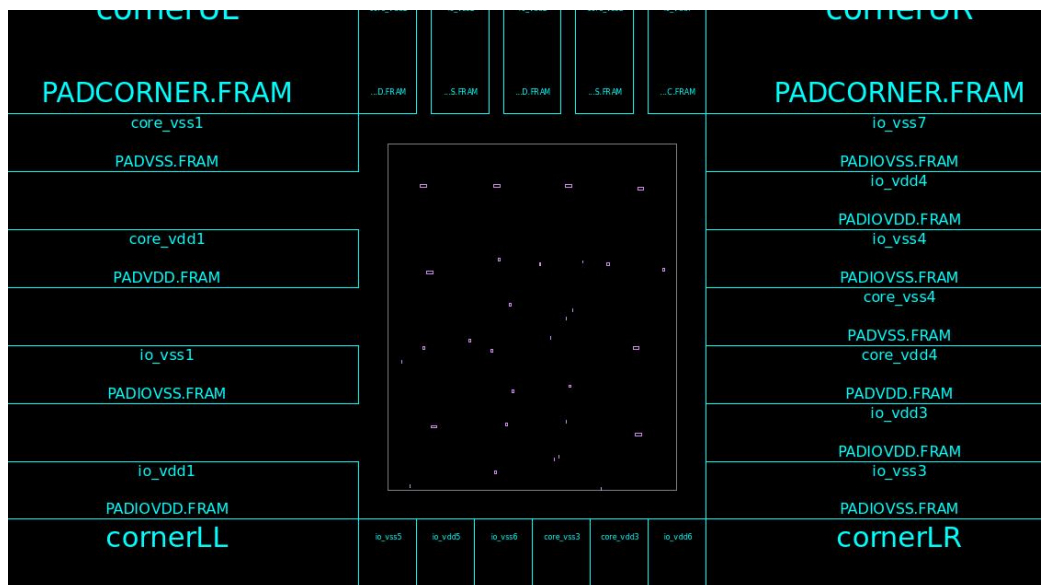
Đầu vào	Đầu ra	Các công đoạn
- Các file RTL đã tổng hợp (netlist) đuôi .v	- Các báo cáo (report) về kiểm tra DRC.	- Tạo thư viện Milkyway và import thiết kế.

<ul style="list-style-type: none"> - Thư viện tham khảo ở cấp độ vật lý (phải trùng khớp với thư viện dùng cho Synthesis). - File công nghệ đuôi .tf đi theo thư viện trên. - Các ràng buộc cung cấp bởi nhà máy (các file rule về DRC và LVS). 	<ul style="list-style-type: none"> - Các báo cáo (report) về kiểm tra LVS. - File GDSII đuôi .gds để gửi đến nhà máy sản xuất. 	<ul style="list-style-type: none"> - Floor Planning: đặt các thành phần nền tảng lên bề mặt thiết kế (I/O, nguồn, đất,...). - Placement: đặt các cell (trong thư viện tham khảo) ứng với netlist lên Floor vừa tạo. - Clock Tree Synthesis: tổng hợp, đi dây cho toàn bộ hệ thống clock, kiểm tra các vi phạm về timing và sửa lỗi (nếu có). - Route: nối dây cho tất cả các kết nối còn lại trong thiết kế. - Xuất các file GDSII, .sdf nhằm kiểm tra các bước cuối cùng. - Kiểm tra DRC (Design Rule Check) nhằm tìm ra vi phạm với ràng buộc của nhà máy sản xuất (nếu có). - Kiểm tra LVS (Layout vs Schematic) nhằm tìm ra khác biệt giữa Layout (sau khi Place & Route) và netlist trước đó. - Nếu không có vấn đề gì, file GDSII sẽ được gửi đến nhà máy để tiến hành sản xuất.
--	--	--

Các bước thực hiện Place&Route thực hiện tương tự bài thực hành-thí nghiệm trước, với kết quả thể hiện trong các hình ảnh sau (từ hình 6.29. đến hình 6.34).



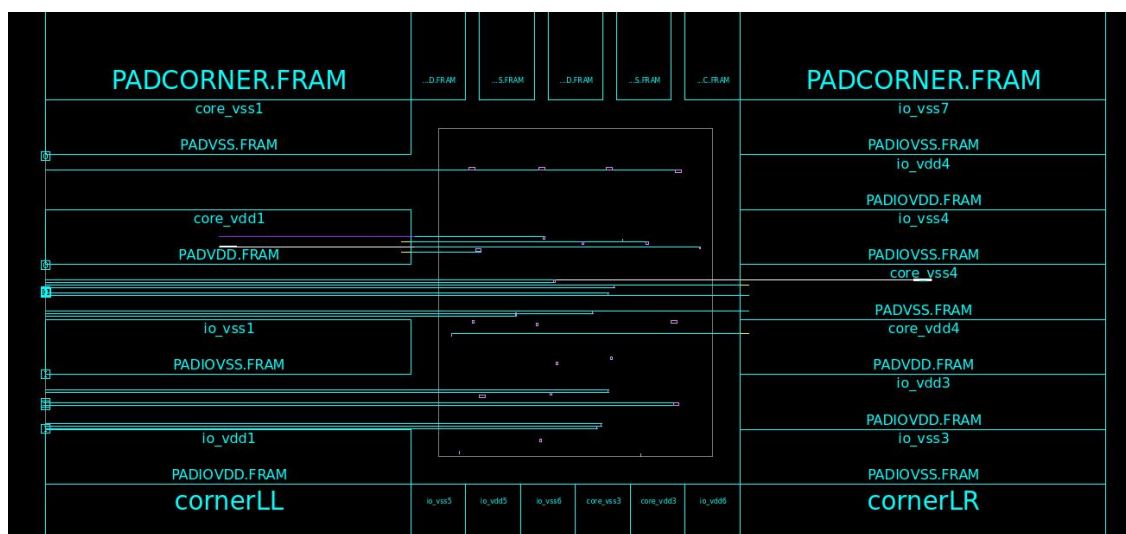
Hình 6.29. Kết quả chạy Floorplanning.



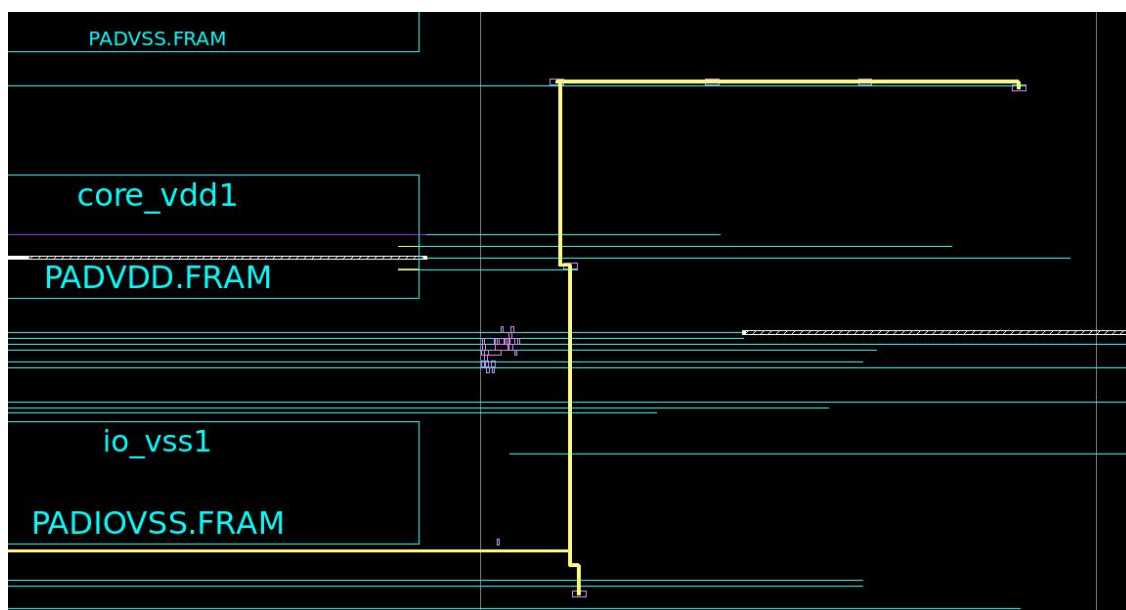
Hình 6.30. Kết quả chạy Placement.

```
icc_shell> derive_pg_connection -power_net {VDD} -ground_net {VSS} -power_pin {VDD} -ground_pin {VSS}
Information: connected 34 power ports and 34 ground ports
```

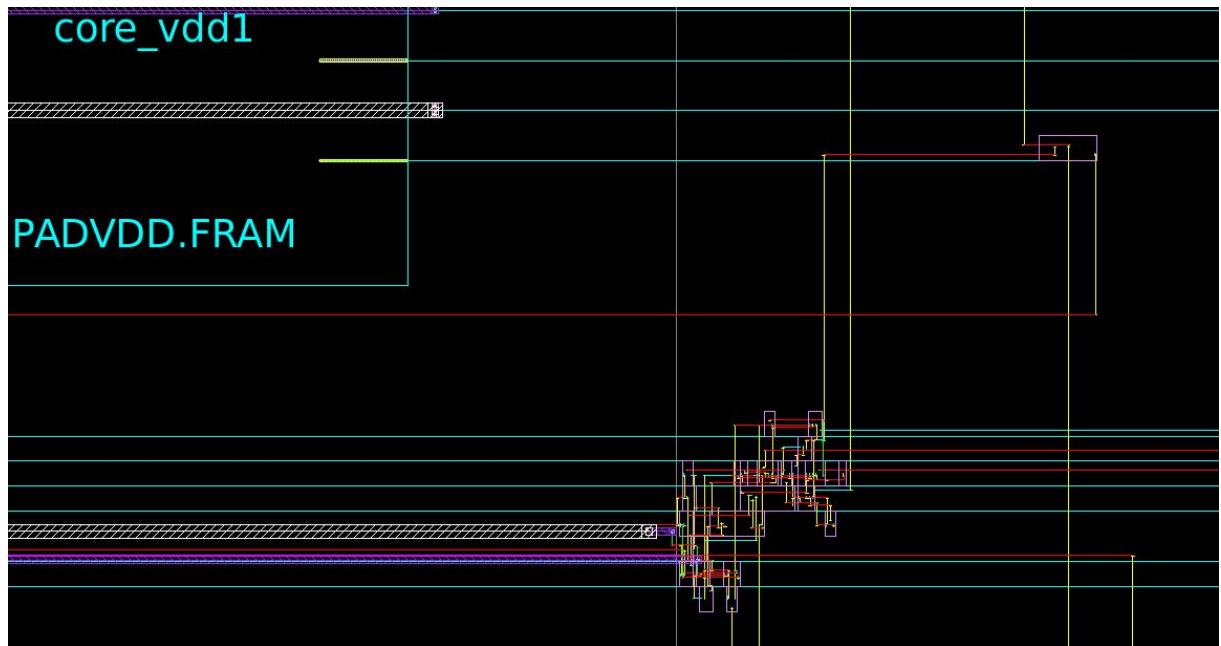
Hình 6.31. Kết quả chạy nhận tín hiệu nguồn/đất.



Hình 6.32. Kết quả nối trước tín hiệu nguồn/đất.



Hình 6.33. Kết quả chạy tổng hợp cây clock.



Hình 6.34. Kết quả chạy quá trình Routing.

Đến đây là kết thúc bài thực hành-thí nghiệm 4. Có thể rút ra được rằng, sử dụng bảng tra có thể tiết kiệm tài nguyên và công sức bỏ ra đáng kể, tuy nhiên không phù hợp cho các thiết kế nhỏ, hoặc cho các thiết kế mà dữ liệu cần tra là quá nhiều. Tối ưu thiết kế dựa trên giải thuật là giải pháp tối ưu cho các trường hợp này, hoặc, người thiết kế cũng có thể kết hợp cả hai nhằm giải quyết bài toán ban đầu, trong đó bảng tra được dùng cho các công đoạn tính toán phức tạp nhưng giới hạn số lượng dữ liệu.