Project 5 Scripts Help

18-765: Digital Systems Testing and Testable Design

For any questions or further information about how the items discussed in this document fit into Project 5, please reference *Project 5 Tutorial: Benchmark Circuit s27*, *Project 5 Cadence Encounter Test Help*, or the help documents within Cadence Encounter Test.

Table of Contents

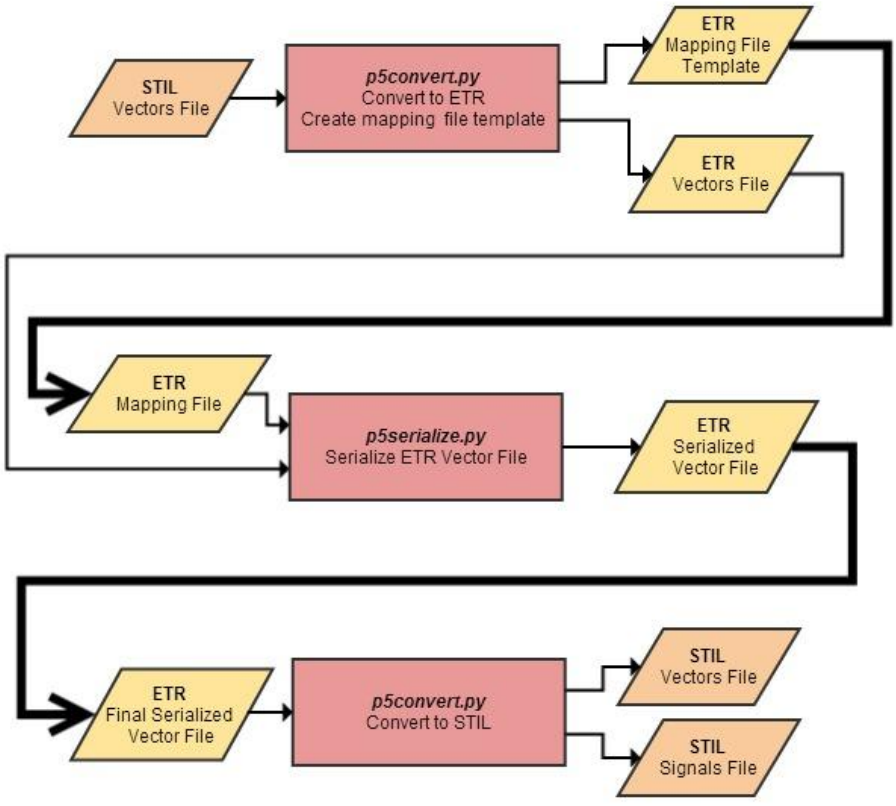
Introduction

Purpose

One of the most important portions of this course's Project 5 comes when you must create the test vectors for your newly-designed DFT circuit. The test vectors will likely be applied in a manner that is different from the original circuit, which requires manipulation of the original test vectors to conform to your design. These scripts are designed to aid in that manipulation.

Process

These scripts were designed to help in one primary process. When working with a scan-based design you will need to perform combinational ATPG using Encounter Test. After this is complete, you must change the application or ordering of the bits in the vectors to match your DFT design. The first script, *p5convert.py* will convert the Encounter Test test vector format to one that is more-easily workable. Then, *p5serialize.py* will serialize those test vectors according to your specifications. Then, after the serialized vectors have been further modified to match your design, they are converted back to the format which is accepted by Encounter Test. This general flow for Project 5's use of these scripts is described graphically below.



Convert Between File Types: *p5convert.py* with *p5fileconvert.py*

When working with Cadence Encounter Test for most applications, all test vectors and testing sequences are handled internally. For this course's Project 5, this is not the case. As a student completing this project, you will need to read, analyze, and manipulate these test vectors. Thus, the vectors must be represented in a way that allows this to occur.
Encounter Test is capable of writing the test vectors it creates to a file, but the formats are not as easily understandable and workable as one would like. You may try using the "Write Vectors …" command in Encounter Test and explore the different file options, and you will find that all of them contain formatting or information that can detract or clutter the important aspects of the file. For this reason it will be very beneficial to convert one of these more complicated file to a format that is much simpler.

This is the primary purpose of the *p5convert.py* script. It can accept a *STIL* file of test vectors created by Encounter Test and convert it to a format that is better for our purposes. This format will be called *Easy-to-Read* throughout this report, and files in the format will be designated with the extension `.etr`. Conversely, *p5convert.py* can also take an *Easy-to-Read* file and create *STIL* files. This setup allows you to take a set of test vectors written by Encounter Test, convert it to a workable file, manipulate those test vectors, convert back to a file for Encounter Test, and finally use the modified test vectors to simulate your circuit.
The script can also convert test vectors to (although not from) a *Verilog TestBench*, allowing you to simulate your Verilog circuit using another piece of software like ModelSim. For troubleshooting, this can often be extremely helpful. Encounter Test does some things very well, but unlike ModelSim it is not easy to find the values of individual pins. The resulting *Verilog TestBench* from *p5convert.py* will require some small additions and alterations, but all of the test vectors will be applied as desired.
This section of the report will describe how the script is used and give other helpful information during its use.

Usage

The script accepts an input file of designated file type, an output file of designated file type, and other options depending on the output file type.
The script has the ability to automatically detect the input file type and automatically determine the output file type, but it is strongly recommended that you determine the input and output types manually. Information about this automatic detection can be found for *STIL* files and for *Easy-to-Read* files at the corresponding links.

Syntax:

```
./p5convert.py infile [-f <intype>] [-t <outtype>] [options] outfilename
```

Options:

```
--version                    Show program's version number and exit

-h, --help                   Show help message and exit

-f INTYPE, --from=INTYPE     Determine input file type: 'stil' or 'etr'

-t OUTTYPE, --to=OUTTYPE     Determine output file type: 'stil'
                             (default), 'etr', 'verilog', or 'tbd'

--map                        With '-t etr', creates template for
                             p5serialize.py mapping file

--mod=MODNAME                With '-t verilog', uses defined name as
                             name of module (default: "modulename")
```

Further information and details about any of the files discussed in this section can be found toward the end of this document.

Input

The script can accept two possible file types. *STIL* is the type that is created by Encounter Test, which can be converted to a more workable file type. *Easy-to-Read* is a file type that is created solely for reading and manipulating. Typically, the *STIL* file will be converted from after the test vectors have been created within Encounter Test. Conversely, the *Easy-to-Read* file will generally be used as an input to this script when trying to obtain an input for Encounter Test.

STIL Vectors File

```
./p5convert.py <infile.logic.stil> -f stil [-t outtype] <outfilename>
```

There are two types of *STIL* files that are created when writing vectors from Encounter Test. The first will likely have a name similar to: `STIL.FULLSCAN.<expname>.signals.stil`. This file contains signal orders and macro definitions. It is essential for Encounter Test, but is not used by this script. More information about the *STIL* Signals file can be found here.

The name of the second *STIL* file will likely be similar to: `STIL.FULLSCAN.<expname>.logic.ex1.ts1.stil`. This is the file which contains the test vectors of the experiment, and it will be used as the input to this script. When converting from type *STIL*, use this file as the input. More information about the *STIL* Signals file can be found here.

Easy-to-Read Vector File

```
./p5convert.py <infile.etr> -f etr [-t outtype] <outfilename>
```

Converting from the *Easy-to-Read* file generally occurs after the user has performed manipulation on the file. Select this file as the input file for the command. The file type itself is described here.

Output

The *p5convert.py* is capable of writing test vectors to three different file types. The *STIL* file type is intended for use with Encounter Test, the *Easy-to-Read* file type is meant for readability and workability, and the *Verilog TestBench* is intended for use with a simulation tool like ModelSim.

STIL Signals and Vectors File

```
./p5convert.py <infile> [-f intype] -t stil <outfilename>
```

If the output type *STIL* has been selected, the script will produce two output files. The first will be the *STIL* Signals file with name `<outfilename>.signals.stil`. As mentioned previously, this file contains pin and macro definitions.

The script will also produce the *STIL* Vectors file with name `<outfilename>.vectors.stil`. This is the file which contains the test vectors. For use with Encounter Test, both files are required. Reference this course's *Cadence Encounter Test Help* document to find out how to use these files for circuit simulation.

Easy-to-Read Vector File

```
./p5convert.py <infile> [-f intype] -t etr [--map] <outfilename>
```

Selecting the *Easy-to-Read* type as the output creates a file with name `<outfilename>.etr` in the current directory.

If the `--map` option is selected, a template mapping file with name `<outfilename>.Tp5map` will be created for use with the script *p5serialize.py*. This file will contain some information taken from the circuit, but for proper operation the user must make some additions and alterations. More information about the template can be found here.

Verilog TestBench

```
./p5convert.py <infile> [-f intype] -t verilog --mod <modname>
      <outfilename>
```

The *Verilog TestBench* output creates a file with name `TB_<outfilename>.v` which is intended for use with a circuit simulation software like ModelSim. This file will require some additions in the form of wiring definitions. The `--mod` option allows the user to give the name of the top-level module under test, which will be printed inside the file. An example of the *Verilog TestBench* file is described here.

Serialize Test Vectors: *p5serialize.py*

Project 5 will require you to implement scan in some form. However, the test vectors created by Encounter Test will not be generated for this type of circuit. ATPG must be performed on the purely combinational circuit at the core of your updated DFT design. In order to properly apply these test vectors they must be serialized and placed in correct order to ensure that the correct nets within the circuit are obtaining the correct values placed on the input scan pins.

The purpose of the *p5serialize.py* script is to perform this serialization. This is accomplished by accepting two separate files. The first containing the test vectors to be serialized, and the second containing information about the manner in which the serialization should occur.

NOTE: In order to perform serialization, the script must make an assumption about the manner in which bits are applied to your design. This assumption applies to the most common use of scan in a design. Your circuit may be different. In this case the resulting serialized *Easy-to-Read* file will not match your circuit.
In this instance there are two possible options. You may use the *p5serialize.py* script as it functions now, and make manual modifications to the obtained serialized file. Instead of this, you may also copy the script to your own workspace and alter it to print the test vectors in accordance with your circuit.

If you are looking for more information about how the serialization is performed, view the examples in the file *p5serialize_ex.zip*. This file contains one non-serialized *Easy-to-Read* vector file, three different mapping files, and the three resulting serialized *Easy-to-Read* vector files. Viewing these files should help clear up any confusion in the serialization process.

Usage

This script accepts an *Easy-to-Read* file (not serialized), and an *Easy-to-Read* mapping file, and creates a serialized version of the file according to the defined mapping. Note that the inputs must be in the correct order for the script to function as desired.

Syntax:

```
./p5serialize.py <inputetrfile> <inputmapfile> <outfilename>
```

Options:

```
--version                    Show program's version number and exit

-h, --help                   Show help message and exit
```

Further information and details about any of the files discussed in this section can be found toward the end of this document.

Inputs

Easy-to-Read Unserialized Vector File

```
./p5serialize.py <infile.etr> <inputmapfile> <outfilename>
```

In order to serialize a set of test vectors, the script first reads in the set of test vectors through the *Easy-to-Read* vector file used as an input. As mentioned earlier, this vectors in this file should be created for a purely combinational circuit. Thus, in the signal definitions at the top of the file there should be only input and output signals stated. If clocks signals are designated they will be ignored.

Easy-to-Read Mapping File

```
./p5serialize.py <inputetrfile> <inmapfile.p5map> <outfilename>
```

The mapping file used as an input to this file will essentially act as an outline for some of the significant DFT structures within your design. You will define the primary inputs and outputs of your DFT design, the controls and clocks, and the scan chains. This information will all be used to create file of serialized vectors, which you will then modify to fit your circuit and its function.
A completed version of this mapping file can be found here.

Output

Easy-to-Read Serialized Vector File

```
./p5serialize.py <inputetrfile> <inputmapfile> <outfilename>
```

The serialized vector file is the result of this script. As it is created, this file may be used as an input to *p5convert.py* and converted directly back to a *STIL*file for use in Encounter Test. However, many files will require some alteration before being applied to an actual circuit. You should generally view the file and ensure that the vectors are printed and aligned as desired before converting them to a less-easily readable file type.
An example of this file can be found here.

File Type Descriptions

STIL

STIL files are present in this project to allow the user to work with Encounter Test in the writing, reading, and simulation of test vectors. The *STIL* files are capable of being both written out and read in by Encounter Test. This makes *STIL* files the obvious choice for test vector communication with Encounter Test.
While this aspect of the file type is very appealing, there are a couple of drawbacks. The vectors file contains a large amount of extra information which is not important for the purposes of this project, resulting in cluttering. The *STIL* files you will see displayed below were created by *p5convert.py*, so much of this extraneous information is not included. You can write out test vectors from Encounter Test (as described in *Project 5 Cadence Encounter Test Help*) in order to see what a *STIL* file from Encounter Test looks like.
Additionally, in order to simulate test vectors in Encounter Test using the *STIL* format, two separate files are required. The signals file contains information about the pins in the circuit to be simulated, and the vectors file contains the actual test vectors of the circuit.

STIL Signals File

Over the course of this project the *STIL* Signals file is used solely by Encounter Test. The file will be created by Encounter Test when writing test vectors and will be required when reading in user-created test vectors.

Information

At the very top of the file you see a line with `STIL 1.0;`. Do not remove or alter this line, as it is essential in proper file identification in Encounter Test. The remaining lines in this section contain comments giving information about the circuit to which the file corresponds.

```
STIL 1.0;

// --- STIL SIGNAL FILE ---
// s27_scan.signals.stil
// Input file: s27_scan.etr
// Converted from etr to STIL

// 46 vectors in file
// 1 clock signals present
// 6 input signals present (not including clocks)
// 2 output signals preset

// CLOCKS: CK
// INPUTS: reset, scan_en, scan_in, G0, G1, G2, G3
// OUTPUTS: scan_out, G17
```

Signal Definitions

The signal definitions follow, with the name of the signal in quotation marks followed by whether it is an input or output. For the purposes of this file clocks are defined as inputs.

```
// --- DEFINE SIGNALS ---

  Signals {
    "CK" In;
    "reset" In;
    "scan_en" In;
    "scan_in" In;
    "G0" In;
    "G1" In;
    "G2" In;
    "G3" In;
    "scan_out" Out;
    "G17" Out;
  }
```

Signal Group Definitions

In this section the signals are sorted into groups as being either primary inputs or primary outputs. The signals must be in order of how their bit values will appear in the test vectors printed in the *STIL* Vectors file.

```
// --- DEFINE SIGNAL GROUPS ---

  SignalGroups {
    "ALLPIs" = '"CK"+"reset"+"scan_en"+"scan_in"+"G0"+"G1"+"G2"+"G3"';
    "ALLPOs" = '"scan_out"+"G17"';
  }
```

Macro Definition

Below is the definition for the macro that will be used in the *STIL* Vectors file to write test vectors to primary inputs and observe the values at the outputs.

```
// --- DEFINE MACROS ---

  MacroDefs {
    "TEST" { WaveformTable "test_cycle";
      Vector {
        "ALLPOs" = %;
        "ALLPIs" = %; } }
}
```

STIL Vectors File

The *STIL* Vectors file contains the actual test vectors that are written from Encounter Test and will be used for simulation with Encounter Test.

Information and Include statement

At the very top of the file you see a line with `STIL 1.0;`. Do not remove or alter this line, as it is essential in proper file identification in Encounter Test. Likewise, the line containing the string "STIL VECTOR FILE" should be left alone, as it can be used by the script *p5convert.py* to automatically detect the file type if no type is designated.The remaining lines in the section contain comments giving information about the circuit to which the file corresponds.
There is also an `Include` statement referencing the *STIL* Signals file shown above. Ensure that the file pointed to by this line matches the name of the corresponding signals file.

```
STIL 1.0;

// --- STIL VECTOR FILE --- // Do not remove, file type detection
// s27_scan.vectors.stil
// Input file: s27_scan.etr
// Converted from etr to STIL

// 46 vectors in file
// 1 clock signals present
// 6 input signals present (not including clocks)
// 2 output signals preset

// CLOCKS: CK
// INPUTS: reset, scan_en, scan_in, G0, G1, G2, G3
// OUTPUTS: scan_out, G17


  Include "s27_scan.signals.stil";
```

Timing Definitions

The next section contains the timing definitions for the signals in the circuit. Although this section is essential for the file, there will not necessarily be any timing concerns when simulating your circuit, as delay faults are not considered. The most important part of this section for our purposes is the string of three or four characters that are present after the name of the signal. These are the different possible bit values that can be assigned to the corresponding pin. If you see ‘01Z’ next to a signal, that signal is a normal primary input. ‘01ZP’ is present next to clocks, with the ‘P’ added for pulsing ability. ‘LHTX’ will be present next to primary output pins.

```
// ----TIMING DEFINITIONS---- //

  Timing {
    WaveformTable "test_cycle" { Period '80.000000ns';
      Waveforms {
        "CK" { 01ZP { '0ns' P/P/P/P; '8.000000ns' D/U/Z/U; '16.000000ns' D/U/Z/D; } }
        "reset" { 01Z  { '0.000000ns' D/U/Z; } }
        "scan_en" { 01Z  { '0.000000ns' D/U/Z; } }
        "scan_in" { 01Z  { '0.000000ns' D/U/Z; } }
        "G0" { 01Z  { '0.000000ns' D/U/Z; } }
        "G1" { 01Z  { '0.000000ns' D/U/Z; } }
        "G2" { 01Z  { '0.000000ns' D/U/Z; } }
        "G3" { 01Z  { '0.000000ns' D/U/Z; } }
        "scan_out" { LHTX { '0ns' X/X/X/X; '72.000000ns' L/H/T/X; } }
        "G17" { LHTX { '0ns' X/X/X/X; '72.000000ns' L/H/T/X; } }
  } }
}
```

Test Vectors

The test vectors in the *STIL* Vectors file are written using the macro defined in the *STIL* Signals file. The bits within the test vectors will be applied to the signals in the order in which they are listed in the Vectors file Timing Definitions and the Signals file Signal Group Definitions. Thus, the correct ordering in both of these locations is absolutely essential. The *p5convert.py* script will be written to ensure that this ordering is correct, so these areas of the files should not be altered unless necessary.

The application of each test vector is contained within a test sequence, which is named by the string of numbers and periods above their definition. For files created by the *p5convert.py* script the test sequence name is primarily used for labeling, but these test sequences are important for test vector separation when the information is present inside Encounter Test. More information about this can be found through examination of different *STIL* files created by Encounter Test or through searching the Encounter Test *Help* files.

```
// ----TEST VECTORS---- //

PatternBurst
  MAIN_BRST { Termination { "ALLPOs" TerminateOff; }
    PatList { MAIN_TEST; } }

PatternExec
  MAIN_EXEC { PatternBurst MAIN_BRST; }

Pattern
  MAIN_TEST {

  // -- Pattern 1 --
  "1.1.1.1.2.1.2":
  Macro "TEST" {
    "ALLPIs" = 0ZZZZZZZ;
    "ALLPOs" = XX; }

  // -- Pattern 2 --
  "1.1.1.1.2.2.2":
  Macro "TEST" {
    "ALLPIs" = P01ZZZZZ;
    "ALLPOs" = XX; }

  // -- Pattern 3 --
  "1.1.1.1.2.3.2":
  Macro "TEST" {
    "ALLPIs" = P110ZZZZ;
    "ALLPOs" = XX; }

  // -- Pattern 4 --
  "1.1.1.1.2.4.2":
  Macro "TEST" {
    "ALLPIs" = P110ZZZZ;
    "ALLPOs" = XX; }
```

Easy-to-Read

The *Easy-to-Read* vector format is was created to represent a file of test vectors in a way that is more easily understandable and workable. This is likely the format in which most of your time viewing and manipulating test vectors will be spent.

Easy-to-Read Unserialized Vector File

Information
The top of the *Easy-to-Read* file created by the *p5convert.py* script contains information about the circuit, about the test vectors, and about the conversion process. At the very top of the file is a string which should not be removed. This is used in the *p5convert.py* script when no input file type has been designated. The script will look for this string and will determine that it should be read as an *Easy-to-Read* vector file.

```
EASY-TO-READ ** do not remove, file type detection
-- s27_comb.etr --
Initial input: p5example/final/STIL.FULLSCAN.exp1.logic.ex1.ts1.stil
Converted from stil to easy-to-read
10 vectors in file
0 clock signals present
7 input signals present (not including clocks)
4 output signals preset
```

Input/Output signal definitions
The input and output signals are then given in lists separated by commas. The ordering of these signals corresponds to the ordering of the bits in the test vectors printed below, so be careful when making any changes to these lists.

```
INPUTS: G0, G1, G2, G3, G5, G6, G7
OUTPUTS: G10, G11, G13, G17
```

Vectors

The rest of the file contains the test vectors. As stated above, the ordering of the bits is such to match the ordering of the signals listed above. The test vectors are represented in this fashion to make for a neat display and to increase the number of test vectors that can be viewed on the screen.

```
#       IN      OUT
1:      1000100 HLLH
2:      0011000 LHLL
3:      0000011 LHHL
4:      0100000 LLHH
5:      0000000 LLLH
6:      0111001 LLLH
7:      1000010 HLLH
8:      0000110 LLLH
9:      1001000 LHLL
10:     1001000 XXXX
```

Easy-to-Read Mapping File Template

When converting to the *Easy-to-Read* file format the `--map` option exists, which creates a template for the *Easy-to-Read* mapping file used with *p5serialize.py*. This template file is meant to make the creation of the *Easy-to-Read* mapping file easier and more straightforward.

Information and Input/Output Signals
The top of this file is a little bit of information about the file. The inputs and outputs from the original test vector file are then printed.

```
 -- s27_comb.Tp5map --

Project 5 mapping file template
File used to define scan chains for use with 'p5serialize.py' script


** Input/output definitions do not need modification
INPUTS: G0, G1, G2, G3, G5, G6, G7
OUTPUTS: G10, G11, G13, G17
```

Signal Assignment
In this portion of the mapping file the user must decide how the serialization should treat each of the input and output signals printed above. For this project, the serialization will be performed on a set of test vectors which have been created for a purely combinational circuit. This combinational circuit is a modified version of the original sequential circuit in which the inputs and outputs of internal flip flops become primary outputs and inputs respectively.
In the DFT circuit that will be built in this project, many of the signals corresponding to the primary inputs and outputs of the combinational circuit are no longer input or output pins. Some may now be boundary scan cells, and others may be internal scan flip flops. In this section of the mapping file, the user will define how the signals are treated so that the serialization can remain consistent with their DFT circuit. For each signal group, every signal must be listed on the same line.
A completed example of this section can be seen here.

```
** EACH OF THE ABOVE SIGNALS MUST BE ASSIGNED TO ONE OF THE FOLLOWING GROUPS **

** Which of the inputs/outputs are to remain primary I/Os?
PRIMARY_INPUTS:
PRIMARY_OUTPUTS:

** Which of the inputs/outputs are being used in boundary scan cells?
INPUT_BS_SIGNALS:
OUTPUT_BS_SIGNALS:

** Which of the inputs/outputs correspond to internal scan flip flops?
** If 'A_in' (PO of comb circuit) is the serial input to the scan flip flop and
**    'A_out' (PI of comb circuit) is the serial output of the scan flip flop
** Use form:    'A_in+A_out, ...'
SCAN_PAIRS:
```

Controls and Clocks

The user's DFT circuit will contain clocks and controls which do not relate to any of the test vectors created for the combinational circuit. In the case of boundary scan, *TMS* and *TCK* will likely be required. In this section of the mapping file the user can list an clock or control signals that are present in their circuit. When the test vectors are serialized, values for the clock and control signals will be included at the beginning of each test vector. For clock signals, a 0 will be printed as the first line and will be pulsed for every subsequent vector.

Control signals can be given a value which will be printed in the serialized vector file. This can be either a singular value which will be printed for every test vector, or a sequence of bits which will be repeated throughout the test vectors. The boundary scan *TMS* pin or a scan enable pin are examples of situations when this sequencing may be useful.

A completed example of this section can be seen here.

```
** Define clock or control signals if desired.
** Clock signals will contain an initial 0 followed by pulses.
** Control signals can be assigned a cycle using the form:
**      'CNTLA=1, CNTLB=1110, CNTLC=ABC'
** If value is not assigned for control signal the value '%' will be given
CLOCKS:
CONTROLS:
```

Padding Vectors

Depending on the design of your circuit, you may wish to apply filler vectors to the circuit. In these filler vectors there are no bits applied to inputs or scan in pins, and there will only be expects on scan out pins if the vector is the last in the cycle. There will still be values applied to the control signals in these vectors.

Filler vectors will primarily be used when you wish to use the control signals to set up the circuit for scanning. For example, when working with boundary scan, the signal *TMS* controls the TAP controller. A specific sequence of bits will need to be applied to *TMS* at the end of each cycle to move the TAP controller through its states and update the circuit. In this case, adding the required number of filler vectors at the end of each cycle will allow the sequence you have defied for *TMS* to operate as desired.

```
** How many filler vectors should print before stimulating PIs or reading POs?
** Use digit, or leave blank if no added vectors are desired
ADD_VECTORS_BEFORE_IO:

** How many filler vectors should print at the end of each cycle?
** Use digit, or leave blank if no added vectors are desired
END_OF_CYCLE_VECTORS:
```

Scan Chains

This section allows the user to define the scan chains of their circuit. Each chain is given a name followed by an ordered list of signals corresponding to boundary scan cells or internal scan cells.

A completed example of this section can be seen here.

```
** User defines scan chains to produce corresponding serialized output
** Scan in port and scan out port will be created automatically
** Definition should be in following form
**  Use form:
**      SCAN_CHAIN <name>: <signal 1>, <signal 2>,  ... , <signal n>
**  For scan chain with order:
**      >|SI_<name>| >> signal 1 > signal 2 >  ... > signal n >> |SO_name|>
** Multiple scan chains can be defined
** For internal scan FFs give only the serial input signal (comb circuit output)
SCAN_CHAIN A:
```

Easy-to-Read Mapping File

This section of the report gives an example of what a completed version of the *Easy-to-Read* Mapping file might look like. Note that this example was used for very small circuit without boundary scan, so the file for your circuit should look very different.
The results of this completed mapping file can be found here.

Signal Assignment

In the signal assignment, you'll note that five of the signals which are primary inputs and outputs remain as such in serialization. There is no boundary scan implemented in this circuit, so none of the original inputs or outputs correspond to boundary scan cells. Finally, there are three internal flip flops in the original sequential version of the circuit which are to become internal scan flip flops in the new version of the circuit. These scan pairs are defined as shown, with the input signal of the scan flip flop and the output signal connected with a "+".
The most important thing to note in this section is that every signal defined as a primary input or output is assigned to one of the three areas. Forgetting to perform this assignment for all of the signals will result in problematic results.

```
** Input/output definitions do not need modification
INPUTS: G0, G1, G2, G3, G5, G6, G7
OUTPUTS: G10, G11, G13, G17


** EACH OF THE ABOVE SIGNALS MUST BE ASSIGNED TO ONE OF THE FOLLOWING GROUPS **

** Which of the inputs/outputs are to remain primary I/Os?
PRIMARY_INPUTS: G0, G1, G2, G3
PRIMARY_OUTPUTS: G17

** Which of the inputs/outputs are being used in boundary scan cells?
INPUT_BS_SIGNALS:
OUTPUT_BS_SIGNALS:

** Which of the inputs/outputs correspond to internal scan flip flops?
** If 'A_in' (PO of comb circuit) is the serial input to the scan flip flop and
**    'A_out' (PI of comb circuit) is the serial output of the scan flip flop
** Use form:    'A_in+A_out, ...'
SCAN_PAIRS: G10+G5, G11+G6, G13+G7
```

Clocks/Controls

In this section the mapping file the clock and control signals are defined. In this example, the value “1” will be applied to the `reset` signal for every vector in the resulting file. The `scan_en` signal will cycle through the defined sequence “1, 1, 1, 0, 0, 0” in order throughout the test vectors. In this instance, the length of the sequence matches the length of the one serialized test vector, as can be examined in the serialized *Easy-to-Read* file. Because it is only possible to print a repeated value or sequence of values, there may need to be some modifications made at the beginning of the test vectors in order to provide the correct setup for the circuit. For the design of most circuits, these additions should not be relatively easy.

```
** Define clock or control signals if desired.
** Clock signals will contain an initial 0 followed by pulses.
** Control signals can be assigned a cycle using the form:
**      'CNTLA=1, CNTLB=1110, CNTLC=ABC'
** If value is not assigned for control signal the value '%' will be given
CLOCKS: CK
CONTROLS: reset=1, scan_en=111000
```

Padding Vectors

In order to add padding vectors to the serialized *Easy-to-Read* file you must specify how many vectors should be added. In the following example, there will be no padding vectors printed before the inputs and outputs are tested. After the scan chain values have finished, the primary input and outputs values will be printed on the next line.

In this example, there will be two additional filler vectors printed after the primary input and output values have been applied. As mentioned earlier, this effect is generally used when control signals are acting upon the circuit. You can observe the result in the serialized *Easy-to-Read* file in the following section.

```
** How many filler vectors should print before stimulating PIs or reading POs?
** Use digit, or leave blank if no added vectors are desired
ADD_VECTORS_BEFORE_IO: 0

** How many filler vectors should print at the end of each cycle?
** Use digit, or leave blank if no added vectors are desired
END_OF_CYCLE_VECTORS: 2
```

Scan Chain Definitions

In this portion of the file the scan chains are defined. For this circuit there is only one scan chain given the name “chain1” and containing three signals. If you look to the signal assignment, you’ll find that these three signals all correspond to internal scan flip flops. Take note that only the input signal to the scan flip flop is defined in the scan chain.

```
** User defines scan chains to produce corresponding serialized output
** Scan in port and scan out port will be created automatically
** Definition should be in following form
**  Use form:
**      SCAN_CHAIN <name>: <signal 1>, <signal 2>,  ... , <signal n>
**  For scan chain with order:
**      >|SI_<name>| >> signal 1 > signal 2 >  ... > signal n >> |SO_name|>
** Multiple scan chains can be defined
** For internal scan FFs give only the serial input signal (comb circuit output)
** Reference help document 'Project 5 Scripts Help' for details on serialization
SCAN_CHAIN chain1: G10, G11, G13
```

Easy-to-Read Serialized File

Below is an example of what the final *Easy-to-Read* Serialized file might look like. *p5serialize.py* does most of the work required work with test vectors, but some small changes have been made to the test vectors to accomplish the desired function with the DFT circuit.

Information

At the top of the file is the “EASY-TO-READ” string printed for automatic file type detection in *p5convert.py* when no input file type is designated. The remainder of this section contains information about the circuit and test vectors.

```
EASY-TO-READ ** do not remove, file type detection
-- TEST.Setr --
Initial etr input: extra/s27_comb.etr
Mapped using: TEST.p5map

11 vectors in file
1 clock signals present
2 control signals present
7 total input signals present (with controls & scanins, not including clocks)
2 total output signals preset (including scanouts)
3 internal scan flip flops
0 boundary scan cells
1 scan chains present
```

Signal Definitions

This is where the signals in the circuit are defined and printed. Look at this portion in conjunction with the test vectors and take note of two important aspects of the file. Correct ordering of both the signal group definitions and the individual signal definitions is essential. You'll see that the clock signal group is the first to be defined here, and its test bits are the first to be listed in the test vectors. The control signals are defined next and their test bits are placed next in the test vector definition. The scan in signal group is then defined, and its test bits are placed next. Additionally, the ordering within the signal groups is also important to keep in mind. As with the unserialized *Easy-to-Read* vector file, the order of the signal definition within their groups corresponds to the bits in the test vectors.

Be careful when making changes to the ordering of these definitions. Any incorrect modifications will result in incorrect bit application to signals.

```
CLOCKS: CK
CONTROLS: reset, scan_en

SCAN_INS: scan_in
SCAN_OUTS: scan_out

PRIMARY_INPUTS: G0, G1, G2, G3
PRIMARY_OUTPUTS: G17
```

Test Vectors

In the serialized *Easy-to-Read* vector file the test vectors are split into signal groups. When adding a vector be sure to include a digit or "X" designation at the beginning of the line containing the test vector, as shown in this example. In *p5convert.py*, the digit (or "X") denotes that the line is a test vector, but the number of the test vector is not read in. The order of test vector application is performed solely by the order of vector definition within the file.

```
#       CK      CTL     SI      SO      PIN     POUT    FIN     FOUT
1:      0       ZZ      Z       X       ZZZZ    X
X:      P       00      Z       X       ZZZZ    X
2:      P       11      0       X       ZZZZ    X
3:      P       11      0       X       ZZZZ    X
4:      P       11      1       X       ZZZZ    X
5:      P       10      Z       X       1000    H
6:      P       10      Z       X       ZZZZ    X
7:      P       10      Z       L       ZZZZ    X
** END OF VECTOR 1 **
8:      P       11      0       L       ZZZZ    X
9:      P       11      0       H       ZZZZ    X
10:     P       11      0       X       ZZZZ    X
11:     P       10      Z       X       0011    L
12:     P       10      Z       X       ZZZZ    X
13:     P       10      Z       L       ZZZZ    X
** END OF VECTOR 2 **
14:     P       11      1       H       ZZZZ    X
15:     P       11      1       L       ZZZZ    X
16:     P       11      0       X       ZZZZ    X
17:     P       10      Z       X       0000    L
18:     P       10      Z       X       ZZZZ    X
19:     P       10      Z       H       ZZZZ    X
** END OF VECTOR 3 **
20:     P       11      0       H       ZZZZ    X
```

Verilog TestBench

Timescale Definition and Information

At the very top of the Verilog TestBench file created by the script *p5convert.py* there is a timescale definition to allow for timing elements within the file. The remaining lines at the top of the file contain comments with information about the circuit and the vector file.

```
`timescale 1ns/100ps
// VERILOG TESTBENCH// TB_s27_scan.v
// Input file: s27_scan.etr
// Converted from etr to Verilog TestBench

// Module "s27_scan"
// 46 vectors in file
// 1 clock signals present
// 6 input signals present (not including clocks)
// 2 output signals preset

// CLOCKS: CK
// INPUTS: reset, scan_en, scan_in, G0, G1, G2, G3
// OUTPUTS: scan_out, G17
```

Wire/Reg Definitions and Module instantiation

The file then attempts to print some of the required declarations for a TestBench. The module is declared at the beginning of the circuit with the user-designated module name from the `--mod` option. Primary inputs and primary outputs from the original vector file are then declared as regs and wires respectively.

The module being tested (with name from the `--mod` option) is then instantiated. It does not contain the signals declared above within its instantiation, as the wiring of the pins on the module may be different from the ordering of the pins within the original test vector file. The user must place the signals in the instantiation with the correct ordering.

```
module TB_s27_scan();
        wire scan_out, G17;     //output signals
        reg CK, reset, scan_en, scan_in, G0, G1, G2, G3;        //input signals


        //Insert correct wiring below
        s27_scan TOP (CLOCKS, INPUTS, OUTPUTS);
```

Vectors

The test vectors are then printed. There are two different ways in which the file assigns values to the input pins. The first applies to the clock values. Generally, these clock values will be pulsed throughout the application of the test vectors, but there may be times at which the user wishes to set a certain clock at a certain value for a series of vectors. To accomplish this, the values of the clock are assigned alongside the values for the normal inputs instead of pulsing continuously inside an “always” block.

The file applies values to the remaining primary inputs through the use of a repeated bus. For clocks that are pulsing, the inputs are set at the falling edge. This process is followed for the remainder of the file.

```
        initial begin
                CK = 1'b0;
                //Test vector 1
                {reset, scan_en, scan_in, G0, G1, G2, G3} = 7'bZZZZZZZ;
                #5;
                #5;
                CK = 1'b0;
                //Test vector 2
                {reset, scan_en, scan_in, G0, G1, G2, G3} = 7'b01ZZZZZ;
                #5;
                CK = 1'b1;
                #5;
                CK = 1'b0;
                //Test vector 3
                {reset, scan_en, scan_in, G0, G1, G2, G3} = 7'b110ZZZZ;
                #5;
                CK = 1'b1;
                #5;
                CK = 1'b0;
                //Test vector 4
                {reset, scan_en, scan_in, G0, G1, G2, G3} = 7'b110ZZZZ;
                #5;
                CK = 1'b1;
```