

Beihang University
College of Software
Graduate School of Integrated Circuits Design
Summer Course
Aug. 30 – Sep. 5, 2004

Part 1 : Flow of IC Design

James Yang

```
if (trn_data_period) begin // send normal data
    trn_crc_data <= trn_data[7:4]; // the 1st nibble
    ext_trn <= 1;
end
else begin // send SCR, SD status, etc
    trn_crc_data <= ssw_data[7:4]; // the 1st nibble
    ext_ssw <= 1;
end
```



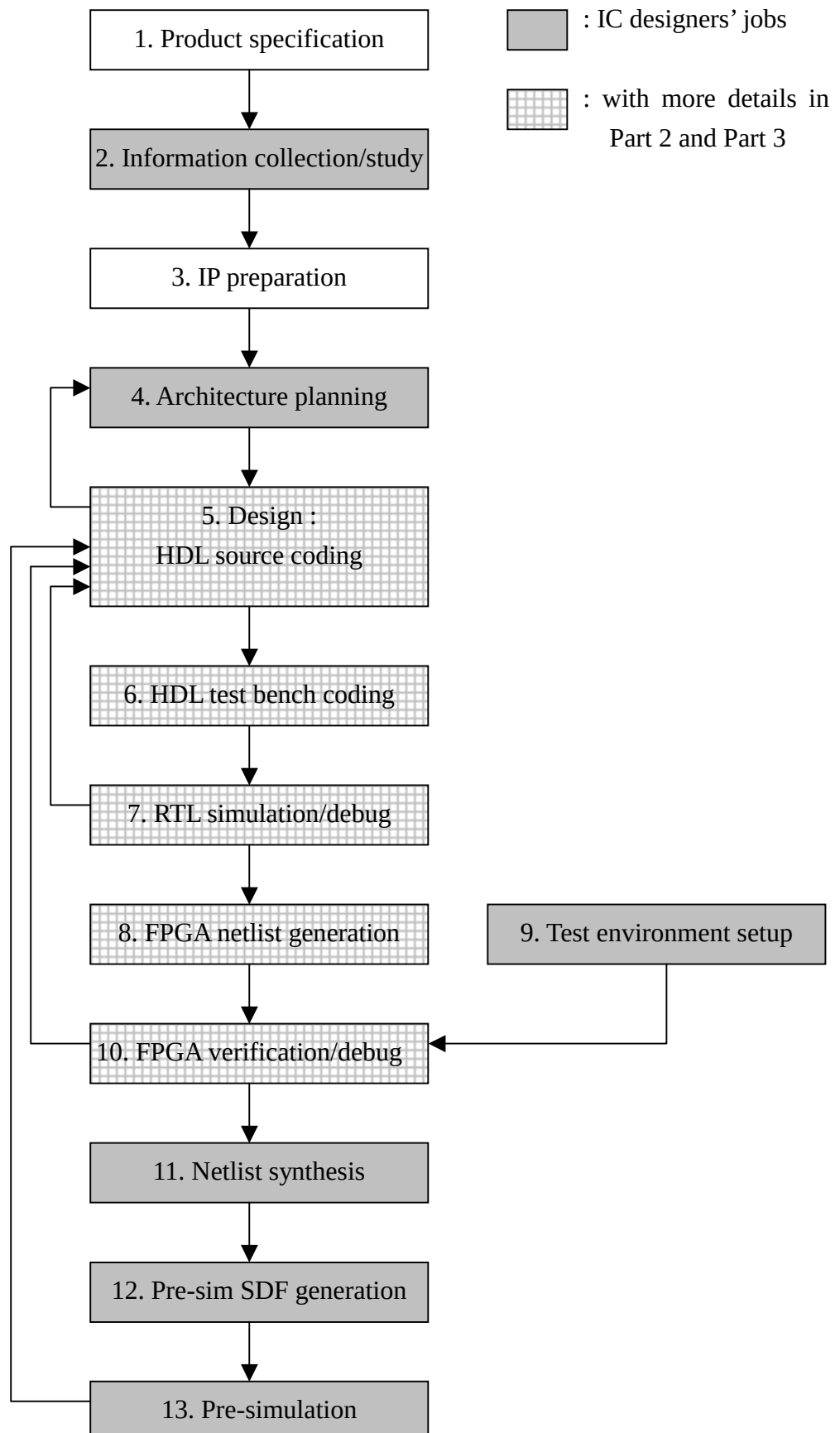
The schedule of 7-day course

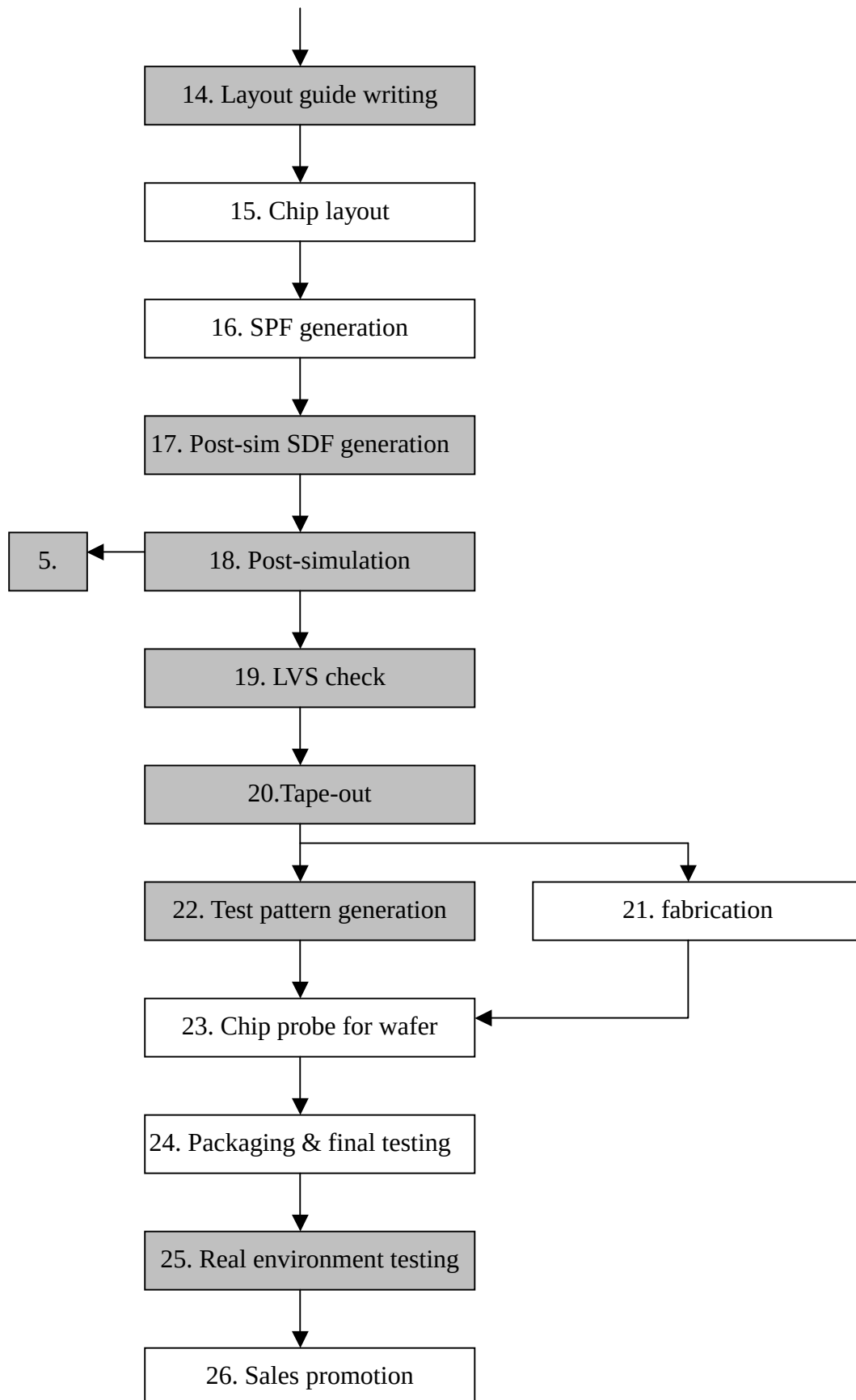
	Aug. 30	Aug. 31	Sep. 1	Sep. 2	Sep. 3	Sep. 4	Sep. 5
8:30 – 10:00	Part I 1-2	Part 2 19-27	Part 2 36-45	Part I 8-10		Part I 16-20	DFT (if we have time)
10:00 – 10:30	Break + Q & A	Break + Q & A	Break + Q & A	Break + Q & A		Break + Q & A	Break + Q & A
10:30 – 12:00	Part I 3-5	Part 2 28-35	Part I 6-7	Part I 11-15		Part I 20-26	DFT (if we have time)
12:00 – 13:00	lunch	lunch	lunch	lunch		lunch	lunch
13:00 – 14:30	Part 2 1-10	Part III Lab	Part III Lab	Part III Lab	Part III Lab	Part III Lab	Part III Lab
14:30 – 15:00	Break + Q & A	Design coding	Test bench coding	RTL sim.	FPGA syn. & routing	FPGA verify.	FPGA verify.
15:30 – 17:00	Part 2 11-18						
After 17:00	Break + Q & A						

1. Assume James Yang will go to conference in the morning of Sept. 3
2. There are 26 topics in part 1.
3. There are 45 topics in Part 2
4. The schedule is flexible, depending on the students' response. If we have time on the last day, we can study something about Design for Testability (DFT)

The information collected in this topic is based on James Yang's own working experience. The students can use the information as a reference. For some other cases such as more advanced projects, these information and suggestion may not be totally applicable.

1. The Flow of IC Design



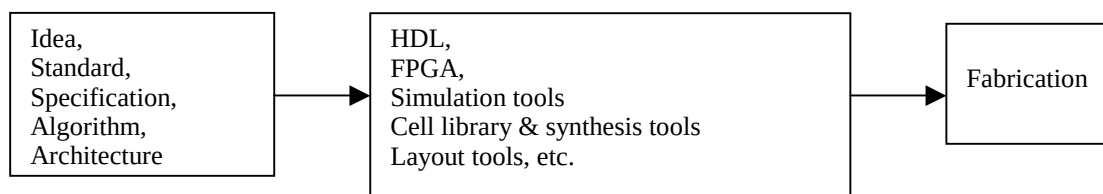


2. Information Collection and Study

There are many different fields in IC industry. The IC designers' specialties also vary a lot. A good IC designer in Area A may take a long time to be familiar with the knowledge in Area B. In the beginning of our career life, we need to ask ourselves that what kind of IC designer we want to be ? Consumer electronics ? PC peripherals ? Communication ? Microprocessor or DSP ? ?

The basic rules and flows of IC design are the same, no matter which kinds of products we are going to join. HDL, FPGA and EDA software, etc. are only the tools to help us in chip realization. The spirit of the IC is the knowledge. So the first challenge we will encounter is to obtain the related information for the design, then understand the information and implement it.

Some information is not free, we need to join the association or buy the documents from their organizations such as ISO or IEEE, etc. The designers should also have strong background so they can understand them quickly, even improve the existing standards or algorithms. Therefore, in addition to polishing our design skill and tool manipulation, the good designers must have sufficient knowledge in their area, and keep on accumulating them.



Ex. 1: 8-port Ethernet switching hub controller

Knowledge :

IEEE 802.3 standards including 10MHz Ethernet and 100MHz Fast Ethernet.

Related area : Asynchronous Transfer Mode (ATM), IEEE 802.11 wireless LAN, IEEE 1394, USB, etc.

Ex. 2 : Card reader + Image displayer

Knowledge :

Joint Picture Expert Group (JPEG) standards.

Image format (RGB, YCbCr, CIF, QCIF, 422, 420, NTSC, PAL, etc.)

Memory card specification (CF, SM, MMC, SD, MS, XD)

File format

Related area : Motion Picture Expert Group (MPEG) -1, MPEG-2, MPEG-4, MP3, etc.

Ex. 3 : Secure Digital (SD) card controller

Knowledge:

SD card specification

Flash memory (a non-volatile memory) specification

Decryption algorithm and flow for SD card

File format

Related are : digital memory cards, thumb drive, non-volatile memory based devices, etc.

3. IP Preparation

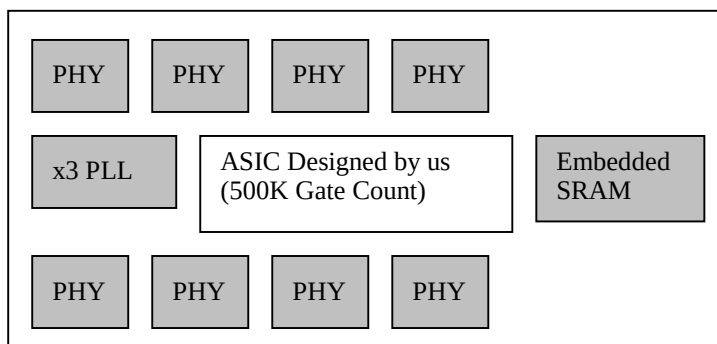
HDL, computer simulation and FPGA verification can only handle the digital part in an ASIC design flow. If there is any analog part in this IC, it relies on the analog IC designers, or it can be purchased from the other companies. Even some pure digital part can also be bought from the other companies to speed up the time to market (TTM).

Those parts that are not designed by us are called IP (intellectual properties). IP may be in the following forms : HDL code, netlist, hard macro which is technology dependent. If the hard macro is made by TSMC 0.35um technology, the users are strongly recommended to use the IP with this technology, or its function and reliability is not guaranteed. So the technology for our design may be determined by the hard macro.

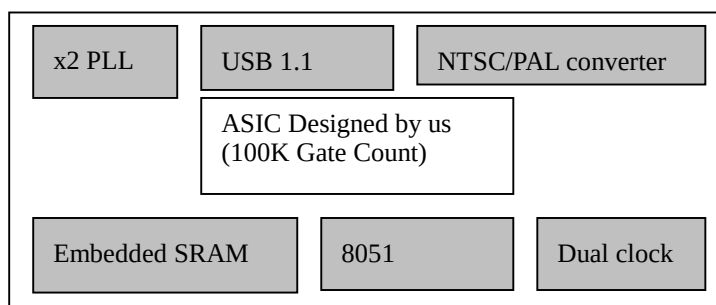
The best case is that all the IP's are provided with their simulation models (such as in HDL). Then we can do the whole chip simulation in computer instead of simulating only part of the design.

Some IP are in very high price such as the PHY in USB 2.0. Some are available from the design service companies. Design service companies are those companies that provide sign-off services to, usually smaller, IC design houses. Sign-off service indicates the jobs from circuits synthesis to wafer out. Some small companies do not have enough man power and software tools for these jobs, even they can not book enough wafer during busy seasons, so the design service companies take care of the jobs for them.

Ex. 1 : 8-port Ethernet switching hub controller



Ex. 2 : Card reader + Image displayer



Not every IP is just fit our requirement. Sometimes, we need to modify it after the purchase. With this consideration, the flexibility (configurable or not) of the IP and the support from its original designers are also criteria for the IP survey.

The all-in-one consideration is very important for the IC design. In Ex. 1, if we can not combine the PHYs into our ASIC, the product can still be promoted with the external PHYs. But, if the other companies can make the all-in-one product, then the system manufactures will not use ours any more for the reason of cost, size, etc. This is why we need to determine the IPs before the starting the design.

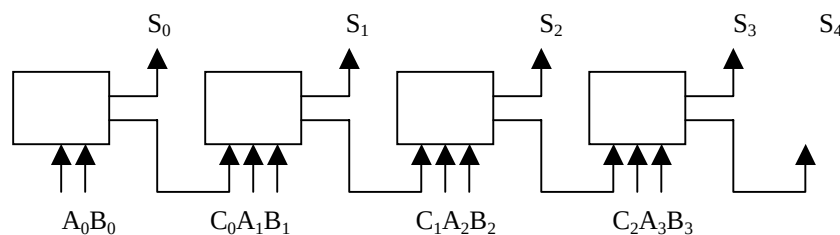
4. Architecture Planning

In the beginning, the designers must understand all the related specifications, standards and algorithms. But there are many ways to implement these specifications and algorithms. The best architecture is the one with the fastest speed and minimum chip size. Unfortunately, the requirement of fast speed is usually contradictory to the requirement of small die size. So, planning an optimal structure with certain trade-off is also an important issue before the job of HDL coding.

Three common examples are given below. They are the adder, the divider and the motion estimator in MPEG.

Ex. 1: The adder

Normal 4-bit Adder : $A[3:0] + B[3:0] = S[4:0]$, one stage by one stage



Look-ahead Adder : expects the carry earlier. It has more gate count, but much faster.

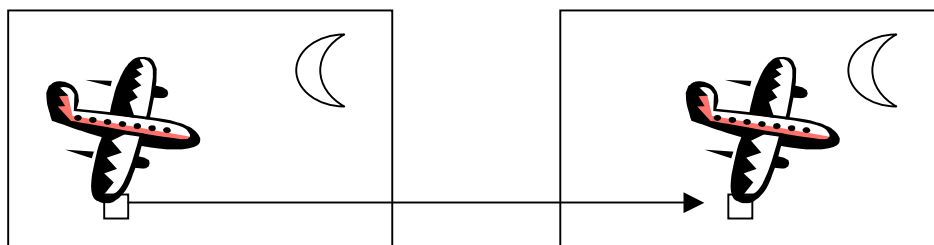
Ex. 2 : The divider

The divisor is fixed. The fastest way is look-up table for this function. But this method needs large memory.

We can subtract the divisor from the dividend repeatedly until the new dividend is less than the divisor. It may take long time but with minimum hardware efforts.

There are still many ways to implement the divider. Each has its own advantages and drawbacks.

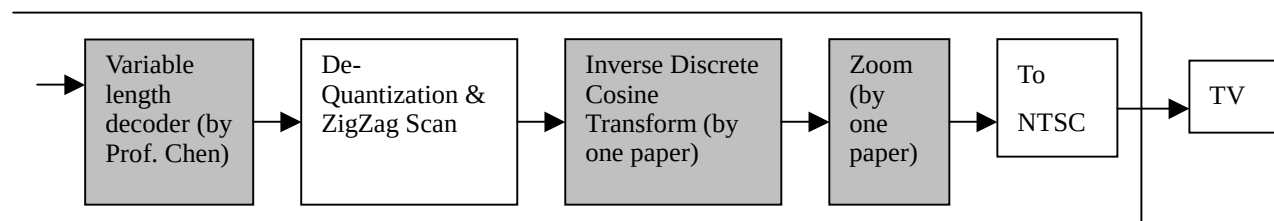
Ex. 3 : Motion Estimator : Find the most similar 8x8 block from the previous picture in one movie.



Basically, there are full search method and three-step search method. Many papers have discussed the architecture to optimize the hardware complexity and speed.

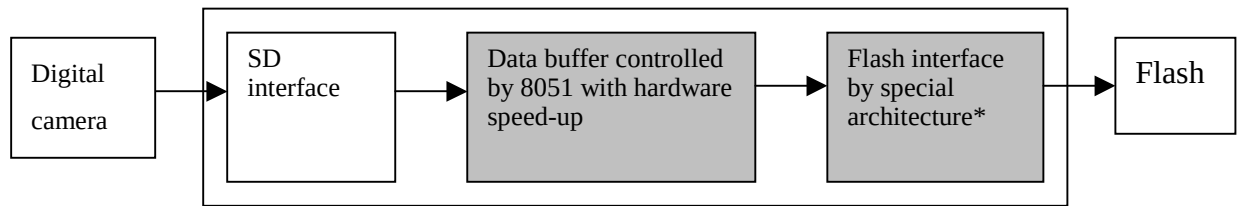
The following two examples are our real cases that the architectures are emphasized.

Ex. 1 : Card reader + Image displayer



Ex. 2 : Secure Digital (SD) card controller

8051 is not fast enough to handle the fast response required by SD card, so many work are share by hardware. The negotiation between hardware and firmware engineers is very important.



* Jan Ti Yang & S.Z. Hsieh, etc. "Optimized Structure for Fast Data Storage to Non-Volatile Memory with Minimum Chip Size", ICSP'04, Beijing, Aug. 2004.

* Jan Ti Yang & S.Z. Hsieh, etc. "Performance Comparison of Logical Address to Physical Address Algorithms for Non-Volatile Memory", ASICON'2003, Beijing, Oct. 2003.

5. Design : HDL Source Coding

A good designer is trained by practical experience and continuous study. We need to be very careful and patient in every design. Because one NRE (explained later) will cost a lot of money and several weeks, the designer will take the responsibility for the money and schedule loss if he/she did some careless mistakes. Experience and carefulness may be the best way to achieve a successful design.

HDL is different from computer language such as C. In HDL, every statement is one circuit. All the statements work at the same time, no matter their order in the code.

Compare the difference :	In VHDL	In C
	$C \leq A + B;$	$C = A + B;$
	$D \leq C + B;$	$D = C + B;$

Assume $A=5$, $B=6$ and $C=7$ initially. In VHDL, $C=11$ and $D=13$ because both additions are executed in the same time. In C, $C=11$ but $D=11+6=17$ because the second statement is executed after the first one is done.

Many students asked : should we learn VHDL at first, or verilog at first ? The answer will be given later.

The following items are some suggestions for a stable and successful design :

Naming style :

1. Don't use the keywords as signal names
2. Don't use the Verilog keyword as the signal names in VHDL because the final netlist is in Verilog.
3. Name the signal with meaning, use x or n as the first letter when this signal is active low.
4. Name the I/O pins as short as possible.
5. Don't name the signal with upper and lower cases mixed.
6. It is the rule that the first letter of a signal name must be a – z.
7. make the module name, instance name and file name the same.

Coding style : remember that a good code is a code that other people can read and understand easily.

1. Add comments as more as possible. Record the modification history.
2. Coding format is fixed and united for all the modules in one design. Follow the format that the project leader assigns.
3. Divide the entire design into different module/entity of suitable number.
4. All the signals in one always/process must be related.
5. Don't use the keywords or syntax those are used not often in order for safe synthesis. In the view point of logic, all the logic circuits can be expressed by IF – THEN – ELSE.
6. Don't use "deep" logic.
7. All the conditions in one IF must be related.

Design style :

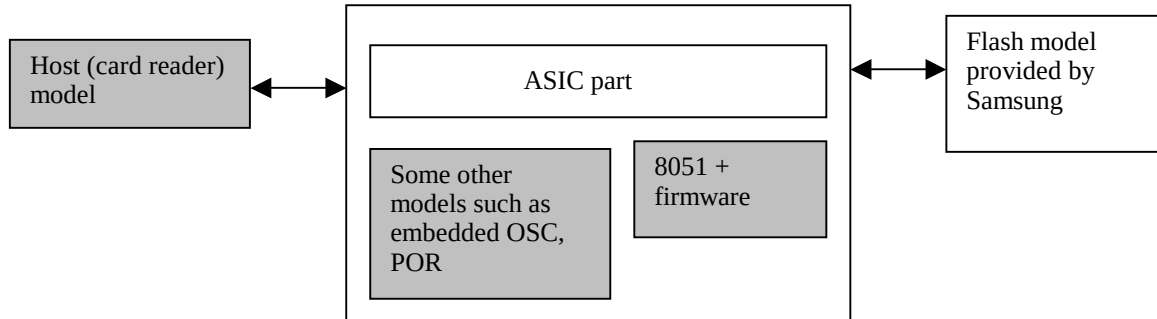
1. It is strongly recommended to do the synchronous design.
2. Always keep the timing issue in mind when doing design. The RTL simulation results may be the same as we thought, but the practical results may be different. Ex. OR two consecutive pulses may not be equal to a two-clock pulse.
3. Determine the low or high reset, synch. or asynch. reset, rising or falling edge trigger in the beginning of the design, and all the modules follow the decision.
4. Use IF and CASE in different situations.
5. Be careful in latching a signal or bus that is generated by the different clock.
6. Make sure that all the signals are reset. FPGA does reset automatically in power-up which may mislead the designer as if they did it.
7. Never read any internal memory (ex. SRAM) before write it. FPGA makes all the bits in the internal SRAM into 0's in power-up.
8. Use data buffer when transfer data from one clock to a different clock, which works as the dual clock FIFO.
9. In VHDL, the 2-dimensional array is allowed, it is very useful. In Verilog, it is allowed in the test bench for simulation only, it can not be synthesized.
10. Follow the register-in register-out rule.
11. Synthesis tool such as Synopsys DesignCompiler, is very stable, any bugs must come from the design code, can not be from synthesis tools.
12. Make the FPGA version and ASIC version as similar as possible, especially the SRAM types. It is ideal that the two versions are the same.
13. Make build-in-self-test (BIST) circuits for embedded memory.
14. Dummy cells and some other remedy circuits are necessary.
15. Some simple test circuits are needed. Usually there are several test modes in one chip.

16. Do gating clock when idle to save power.
17. Don't rely on the script to guarantee the design (Ex. Synopsys full_case). But some good constraint in script can do the better performance (Ex. Look-ahead adder).
18. If time is sufficient, do one more latch by clock instead of using MUX.
19. Don't use internal tri-state. FPGA may not have such circuits. ASIC requires bus holder to handle internal tri-state.
20. Do the pad insertion on the top level. So the output enable signal of every bi-directional pad will be taken out to the top level.
21. Be very careful in selecting pad. (Ex. pull up ability, Schmitt trigger, 5V tolerant, clock pads, etc.)
22. Be aware of the problem caused by clock skew.
23. Don't try to generate the half-cycle signal.
24. If there are many functions to be revised, do it one by one, i.e., revise one function and check one function.
25. It is a good habit to align the bit number of every signal in one computation equation, even though the synthesis tool can handle it.
26. Never use division provided by HDL.
27. Reduce the unnecessary clock. It causes trouble in design and layout. Most of the FPGA has 1-4 dedicate clock paths only.

6. HDL Test Bench Coding

Test bench is also the HDL code that provides the simulation models for the devices connected with the ASIC. The efforts in designing test bench may not be less than that in designing ASIC. For the ASIC with many peripheral devices, it may take much longer time in coding the test bench than coding the ASIC.

Ex. : Secure Digital (SD) card controller

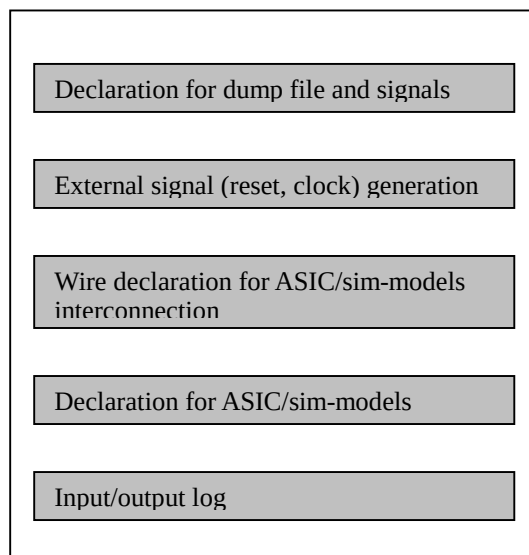


The following items are some requirements for a good test bench

Even though the test bench is not for synthesis and can be coded in a easier way, but we should notice the following items :

1. The test bench can cover as more cases as possible, both the function cases and timing cases. So it is recommended that the designers for test bench are different from the designers for the ASIC.
2. The test bench must be flexible. New testing condition can be added easily.
3. Log the test results for automatic comparison instead of checking the results by eyes.
4. All the results of IC pins are logged as the pattern for the IC testers.
5. If there are some functions that FPGA verification can not afford, the test bench must cover them.
6. Try to make the test bench as close to the real environment as possible (Ex. external memory).
7. It is better to make an integrated test bench. A test bench with too many separate modules is not easy to be maintained.

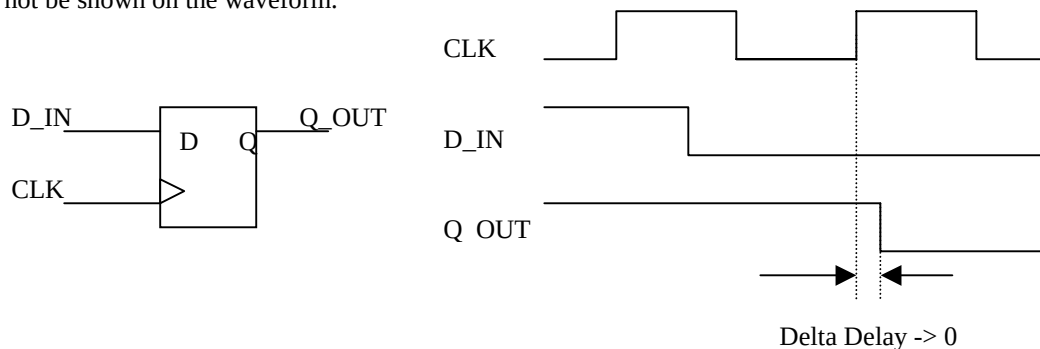
A typical test bench contains the following items :



7. RTL Simulation/Debug

RTL simulation is also called the behavioral simulation or functional simulation. It is only based on the register transfer level (RTL) function, timing is not considered. The delay in the stage is called the delta delay, it really exist but can not be shown on the waveform.

Ex :



Delta delay is effective only for the signals generated by the same clock. If the signal is latched by the other clock, both clocks have the same frequency and phase, then unknown condition may happen. So we need to insert a little delay in this situation.

All the design must pass the RTL simulation with fair test bench at first, then the source code can be synthesized into netlist for FPGA or pre-sim. If not, the debug work on FPGA will be terrible.

The most popular simulation tool is the Cadence Verilog-XL. It is recognized by all the companies that Verilog-XL is the standard to check the simulation results. But it is the workstation version. For the PC version, ModelSim is a good choice. The simulation results are close to those of Verilog-XL.

There are many tools can be used to see the waveform. Debussy is very popular because of its small size of dump file.

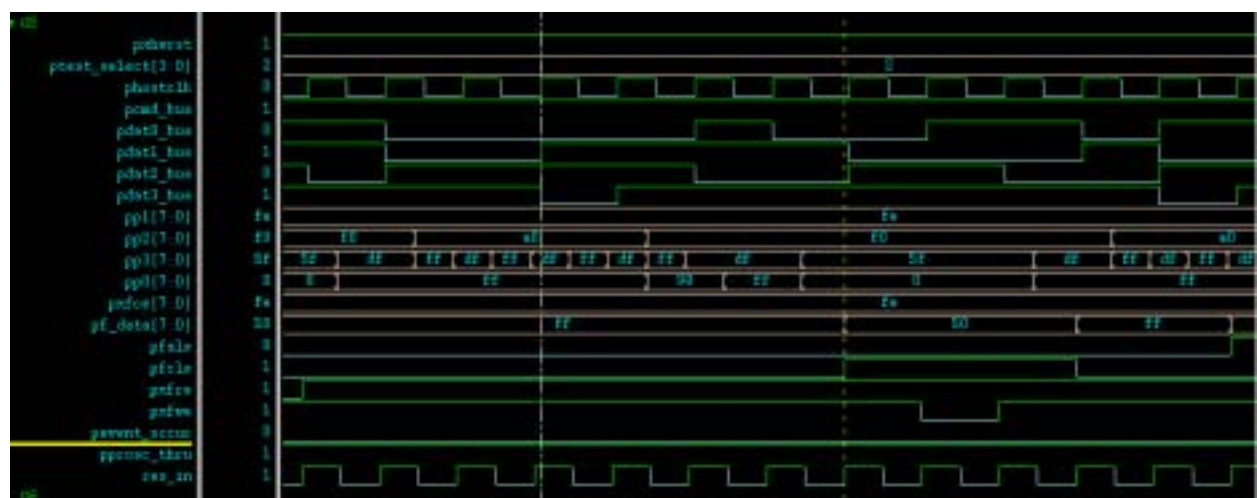
The goal of simulation is to find out the errors before hardware realization. So it takes many iterations to modify the code and do re-simulation until there is no bugs (at least can not be found with the test bench). Then, the results (waveform) will be checked very carefully, some important outputs are logged as the “golden pattern”. From now on, we can compare the output directly to see if the results are correct.

Some functions are very difficult to be tested. To test them requires long pattern. We need to think how to overcome this problem, some circuits may be added to achieve this goal.

Ex . Secure Digital (SD) card controller

There are almost 800K cycles in the test pattern, but toggle rate is about 70%.

An example of the waveform from RTL simulation is shown below :



8. FPGA Netlist Generation

The restriction of FPGA : here the Altera FPGA's are used as the example.

1. Every FPGA has its own capacity limitation. We can not download our design into an FPGA with 90% utility rate or even higher. Take Altera 20K200 as the example, 50K gate count will takes more then 80% utility. It may restrict the performance of the FPGA. So we use Altera 20K600, it takes 30% only.
2. There are only 2 clock paths for clock and 4 dedicate input for reset, etc. If there are more clocks, it may be a trouble. Any clock generated inside the design is suggested to be routed out of the FPGA, the goes into FPGA again through the clock path.
3. There are embedded memory inside the FPGA. If the design contains large embedded SRAM, the FPGA may not have so much. So we need to added external memory outside the FPGA. Also, the memory types may be different from the real embedded ones.
4. Altera's FPGA will reset all the register outputs and SRAM data into 0 at power up, no matter we do it or not. It may be dangerous for the final chip because the FPGA works find and, actually, we did not do the reset in our design.
5. Some FPGA's declare its high speed. These information are based on some special cases. With high utility, the stable speed will not be over 50MHz usually. For some design with high speed and large gate count (Ex. : 8-port Ethernet switching hub controller : the clock is 75MHz, the gate count is 500K), FPGA verification is not practical. Otherwise, the design is separated into several FPGAs and verified with lower clock rate.

After RTL-simulation, our design code can be synthesized into the FPGA netlist. The synthesis tool used here is the Synopsys FPGA Express of PC version. The synthesized netlist is the EDIF file, not the verilog file. The following example is one paragraph of the edif netlist, not easy to be read.

```
edif all_top
  (edifVersion 2 0 0)
  (edifLevel 0)
  (keywordMap
    (keywordLevel 0)
  )
  (status
    (written
      (timeStamp 2004 3 30 8 54 19)
      (program "FPGA Compiler II"
        (version "3.5.1.6112")
      )
      (author "RD")
    )
  )
  (external UnlinkedDesignLibrary
    (edifLevel 0)
    (technology
      (numberDefinition)
    )
    (cell syn_ram_512x8_irou
      (cellType GENERIC)
      (view Netlist_representation
        (viewType NETLIST)
      )
      (interface
        (port
          (array
            (rename Q_7_0 "Q[7:0]") 8
          )
          (direction OUTPUT)
        )
        (port
          (array
            (rename Data_7_0 "Data[7:0]") 8
          )
          (direction INPUT)
        )
        (port WE
          (direction INPUT)
        )
        (port
          (array
            (rename Address_8_0 "Address[8:0]") 9
          )

```

```

    )
    (direction INPUT)
  )
  (port Inclock
    (direction INPUT)
  )
)
)
)
)
)
.....
(instance buf26_reg0
  (viewRef Netlist_representation
    (cellRef APEX20KE_LCELL
      (libraryRef APEX20KE)
    )
  )
  (property operation_mode
    (string "normal")
  )
  (property packed_mode
    (string "true")
  )
  (property lut_mask
    (string "0000")
  )
)
)
.....
(net
  (rename C8937_N33 "C8937/N33")
  (joined
    (portRef DATAA
      (instanceRef C10904)
    )
    (portRef COMBOUT
      (instanceRef C13440)
    )
  )
)
)

```

The next step is to use the netlist files and the other 2 files (LMF and TCL files) derived from FPGA Express to generate the FPGA download file. This step is to place the cells and route the nets according to the netlist into the assigned FPGA. So it is called the auto placement and routing (APR). The tool now is Altera Quartus. Before we do this, a file that lists the pin assignment and target FPGA must be prepared. It is the CSF file with an example shown below (one part of the file) :

```

COMPILER_SETTINGS
{
  RUN_TIMING_ANALYSES = ON;
  MERGE_HEX_FILE = OFF;
  FOCUS_ENTITY_NAME = |all_top;
  FAMILY = APEX20KE;
  DPRAM_DUAL_PORT_MODE_OTHER_SIGNALS_EPXA1 = "DPRAM0 TO 1 DPRAM1 TO 2";
  DPRAM_32BIT_SINGLE_PORT_MODE_OTHER_SIGNALS_EPXA1 = "MEGALAB COLUMN 1";
  DPRAM_8BIT_16BIT_SINGLE_PORT_MODE_OTHER_SIGNALS_EPXA1 = "MEGALAB COLUMN 1";
  DPRAM_DUAL_PORT_MODE_OUTPUT_EPXA1 = "DPRAM0 TO 1 DPRAM1 TO 2";
  DPRAM_32BIT_SINGLE_PORT_MODE_OUTPUT_EPXA1 = "LOWER TO 1ESB UPPER TO 1";
  .....
  COMPILE_LEVEL = FULL;
  SAVE_DISK_SPACE = ON;
  SPEED_DISK_USAGE_TRADEOFF = NORMAL;
  LOGICLOCK_INCREMENTAL_COMPILE_ASSIGNMENT = OFF;
}
CHIP(all_top)
{
  full_buffer[0] : LOCATION = PIN_AF35;
  full_buffer[1] : LOCATION = PIN_AE35;
  full_buffer[2] : LOCATION = PIN_AD35;
  full_buffer[3] : LOCATION = PIN_AC35;
  fr_data_period : LOCATION = PIN_G34;
}

```

```

.....
DEFAULT_DEVICE_OPTIONS
{
    RESERVE_ALL_UNUSED_PINS = "AS OUTPUT DRIVING GROUND";
    HEXOUT_FILE_COUNT_DIRECTION = UP;
    HEXOUT_FILE_START_ADDRESS = 0;
    GENERATE_HEX_FILE = OFF;
    GENERATE_RBF_FILE = OFF;
    GENERATE_TTF_FILE = OFF;

    .....
    FLEX10K_ENABLE_LOW_VOLTAGE_MODE_ON_CONFIG_DEVICE = ON;
    FLEX6K_ENABLE_LOW_VOLTAGE_MODE_ON_CONFIG_DEVICE = OFF;
    MAX7000_ENABLE_JTAG_BST_SUPPORT = ON;
    ENABLE_JTAG_BST_SUPPORT = OFF;
}
AUTO_SLD_HUB_ENTITY
{
    AUTO_INSERT_SLD_HUB_ENTITY = ENABLE;
    HUB_INSTANCE_NAME = sld_hub_inst;
    HUB_ENTITY_NAME = sld_hub;
}

```

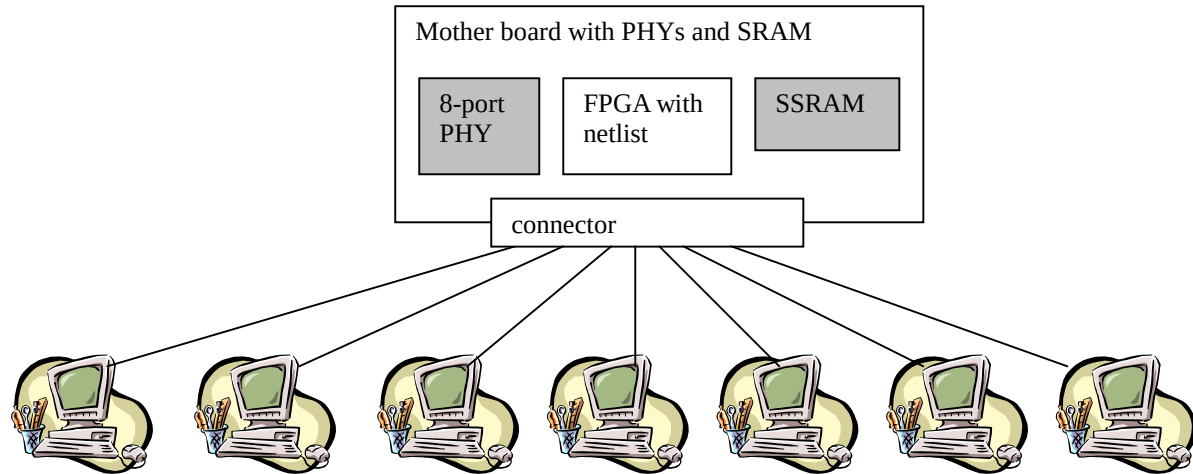
Not every pin of the FPGA can be used as the I/O pins. The available pins are given in their data sheet.

With the 4 files, Quartus can generate the download file for our FPGA. The files are the POF files. With the capacity difference, some FPGA has one POF file and some has more than 4 POF files.

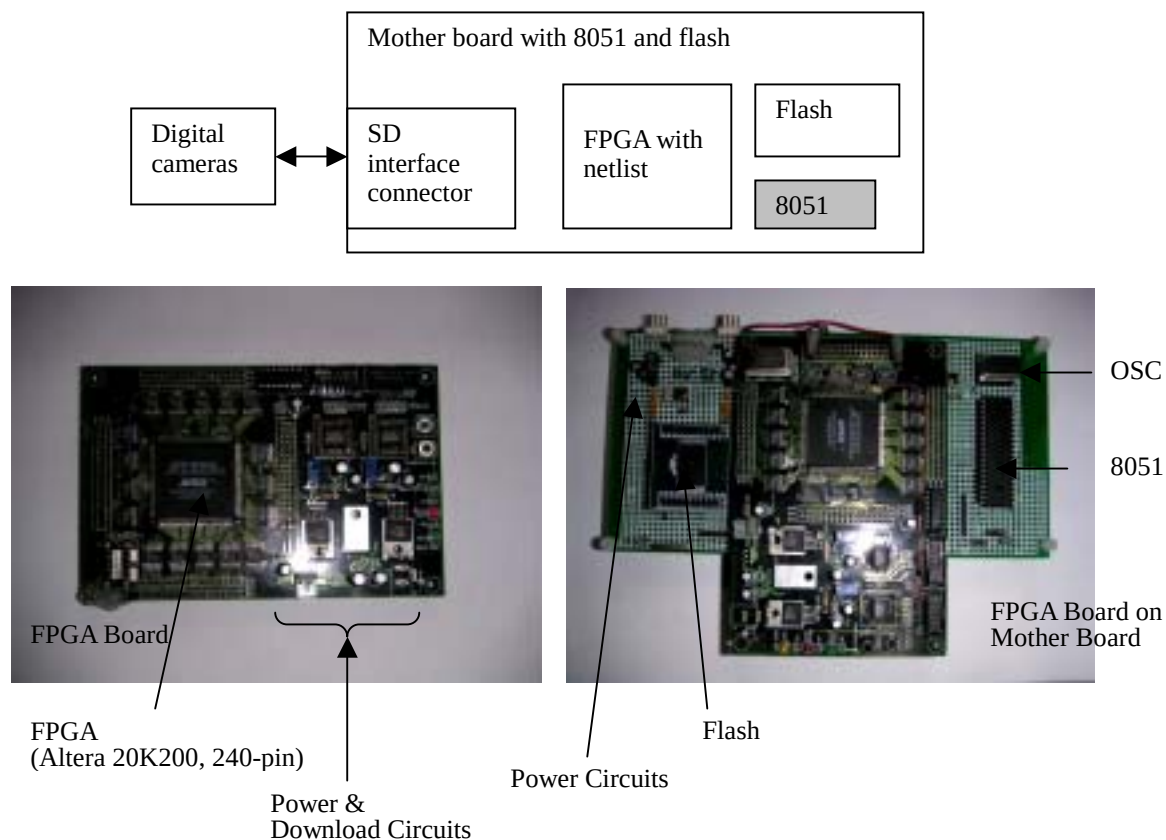
9. Test Environment Setup

FPGA is very similar to the real chip except for the timing and IPs. The FPGA verification with real test environment can find most of the problems that the chip will meet.

Ex. 1: 8-port Ethernet switching hub controller



Ex. 2 : Secure Digital (SD) card controller



For some projects, the test environment is hard to be built. The major problem is it may cost too much. In Ex. 2, one purpose of the verification is the compatibility of the SD card controller. There must be many different types of devices that have SD interface to do the test. But the available devices, such as card reader, digital camera, MP3 player, etc. are over several hundreds. For the complete test report, the cost of buying devices is real high.

This step should start as early as possible. We also need to be careful at the power of the entire system. Don't let the designers waste too much time on debugging the board, not the source code.

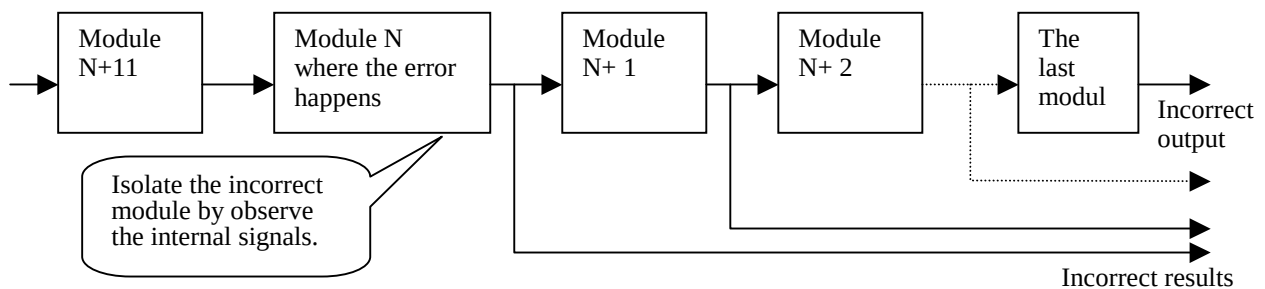
10. FPGA Verification/Debug

This step takes the longest time in the entire design flow. The source code can not be synthesized for tape-out until the verification is passed. Almost all the real problems can be found in this step to make up the insufficiency of our test bench. It is also the hardest step for the junior engineers who know there is an error, but can not find the error.

Basically, the errors encountered can be divided into two categories : logical errors and timing/electrical errors.

The logical errors are those errors made by our HDL design or the associate firmware. These cause the functional problems. The basic debug flow can be :

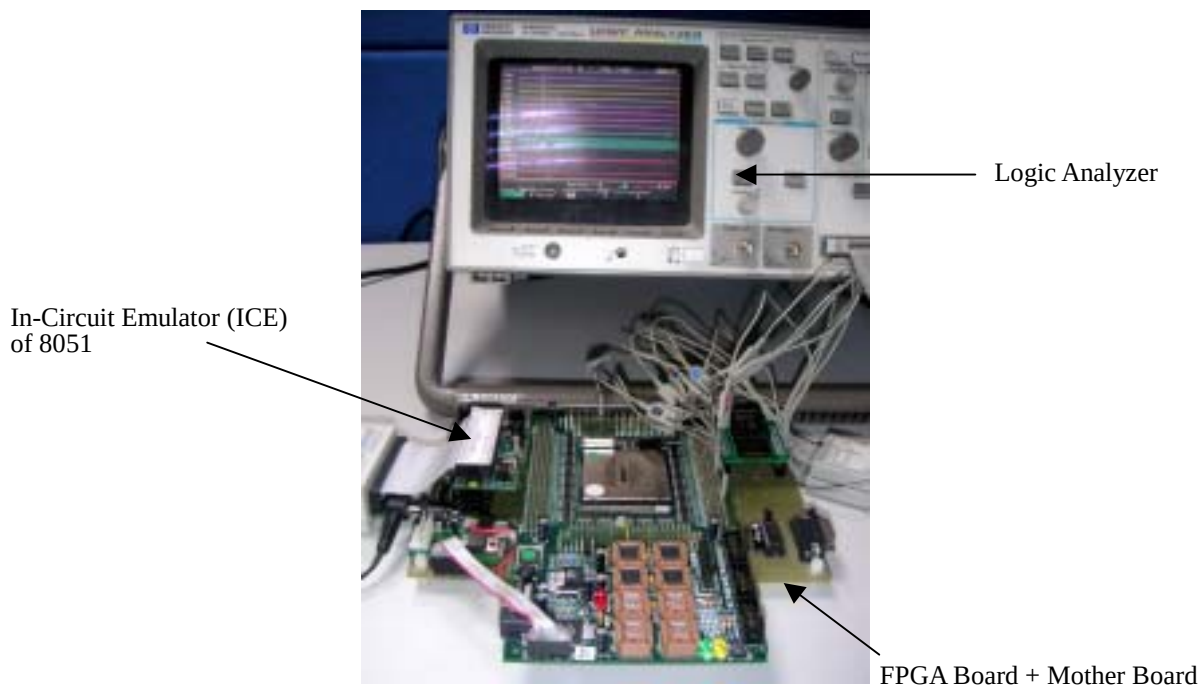
1. Observe the error from the output.
2. Repeat the error. Make sure the initial conditions are the same and try to repeat the same error again.
3. If it is repeatable, re-route the netlist with more observation pins. These observation pins are not the actual I/O pins, but some important internal signals. If the observation shows the incorrect results, then we need to extract more signals for observation. This procedure is repeated until the error is located.



Usually, the debug procedures will not be so simple because there are several errors that may influence each other. So it is hard to isolate the error in this condition. This step relies on the engineer's debug experience very much. An experienced engineer can find the error faster based on the similar problems of those previous projects. Besides, the more the engineers understand the system, the faster they can find the errors.

The equipment used the most often is the logic analyzer. How to set the desired trigger point is a challenge.

Ex. : Secure Digital (SD) card controller



The timing/electrical errors are those problems that may not be repeatable. Sometimes it works well, and sometimes it will not work at all. If this phenomenon happens, we can focus on this type of errors.

Ex. 1 : FPGA consumes much more power than a IC. If the current supply is not sufficient, or some power wires

are too thin, it will result in bad effect to the FPGA verification.

Ex . 2: For the signals from the external devices, they are connected to the FPGA mother board through some long wire and connector. If the driving current of the external devices are not strong enough, the signals will be too weak to be identified.

Ex. 3 : The ground level of mother board is different from that of the external devices.

Ex. 4 : The looseness of mounted elements and the broken wires in the FPGA board and mother board are the most often problems encountered in the verification environment.

Ex. 5 : The clock wire is very critical if the frequency is high. If it is too thin, it may be induced by the other signal. If it is too wide, the capacitance may be strong that the sharp edge will be reduced. Do not route the clock wire into a loop.

Ex. 6 : The power up condition of FPGA may be different from that of real chip. Make sure that there is no problem for the chip power up even if the FPGA verification is OK.

Ex. 7 : The signals generated with asynch. Logic may be unstable.

Ex. 8 : For high utility of FPGA, some paths may be routed long and the timing is not met. If it is routed again with some minor change, these paths may be OK but some other paths may fail.

The equipments used the most often are, besides the logic analyzer, the oscilloscope.

11. Netlist Synthesis

After the FPGA verification, the design code is confirmed to be a good one. The next step is to synthesize the design code into a netlist for chip layout.

The most popular synthesis tool is Synopsys DesignCompiler. It has been used by numerous engineers, so its performance is guaranteed. The synthesized netlist does not include the IP parts. These IP parts are also synthesized from the empty modules with their input/output declaration only. So that the layout engineers can insert the IP's inside the empty places.

The choice of the cell library should be determined in the beginning of the project because some hard macros are technology dependent, i.e., if the IP is designed with 0.35um technology, we need to choose the 0.35um cell library. The cell libraries from UMC and TSMC are very popular. The 0.35um means the length of the transistor in the real chips. From our experience, the history for the popular technologies of the recent 10 years is :

* This is based on our consumer products requiring no high-end technologies

1995	0.7um	
1996	0.6um	
1997	0.5um	3.3V
1999	0.35um	
2001	0.25um	
2004	0.18um	0.18V
	0.15um	
	0.13um	

We need to indicate the directory path of our cell library in the file "synopsys_dc.setup". The following paragraph is an example for synopsys_dc.setup

```
/* setup the Setupe..Defaults menu */
include /diska1/user/ba057/sd_card/syn/TSMC_naming_rule.script

designer = "James Yang"
company = "MEGAWIN"

search_path = { "." "/diska1/user/ba017/KIT/tcb773s/synopsys" }
link_library= { "*" "/diska1/user/ba017/KIT/tcb773s/synopsys/tcb773stc.db" }
target_library= { tcb773stc.db }
symbol_library= { tcb773s.sdb }
/* tcb773s.sdb      Synopsys symbol library */
/* tcb773stc.db     Synopsys technology library (typical case)*/
/* tcb773swc.db     Synopsys technology library (worst case) */
/* tcb773sbc.db     Synopsys technology library (best case) */
hdlin_translate_off_skip_text = TRUE
default_schematic_options = "-size infinite"

ink_library = link_library + target_library ;

define_design_lib TOY_BOX -path ./TOY_BOX

/* Site Specific Variables */
synthetic_library = {}
command_log_file = "./command.log"
view_command_log_file = "./view_command.log"
plot_command = "lpr -Plw"
text_print_command = "lpr -Plw"

/* enable Text Viewer feature */
hdlin_source_to_gates_mode = "high"
write_name_nets_same_as_ports = "true"
compile_fix_multiple_port_nets = "true" /* avoid use "assign" for wire connection */
verilogout_no_tri = "true"
verilogout_single_bit = "true"
compile_no_new_cells_at_top_level = "true"
```

```

view_script_submenu_items = view_script_submenu_items + \
                                {"Clean All" , "remove_design -designs"}
/* Add change_names rule */

bus_naming_style = "%s_%d"
define_name_rules simple_names -allowed "A-Za-z0-9_[]" \
-first_restricted "_" \
-map {{ "\"*cell\"", "U" }, { "\"*-return\"", "RET" }} ;

alias change_names "change_names -rules simple_names"

/* Synopsys to Compass EDIF interface */
edifin_ground_name = "VSS"
edifin_ground_net_name = "VSS"
edifin_ground_net_property_name = "global"
edifin_ground_net_property_value = "VSS"
edifin_ground_pin_name = "VSS"
edifin_ground_port_name = "VSS"
edifin_netlist_only = "true"
edifin_power_name = "VDD"
edifin_power_net_name = "VDD"
edifin_power_net_property_name = "global"
edifin_power_net_property_value = "VDD"
edifin_power_pin_name = "VDD"
edifin_power_port_name = "VDD"
edifin_power_and_ground_representation = "net"
edifout_ground_name = "VSS"
edifout_ground_net_name = "VSS"
edifout_ground_net_property_name = "global"
edifout_ground_net_property_value = "VSS"
edifout_ground_pin_name = "VSS"
edifout_ground_port_name = "VSS"
edifout_netlist_only = "true"
edifout_no_array = "true"
edifout_power_name = "VDD"
edifout_power_net_name = "VDD"
edifout_power_net_property_name = "global"
edifout_power_net_property_value = "VDD"
edifout_power_pin_name = "VDD"
edifout_power_port_name = "VDD"
edifout_power_and_ground_representation = "net"

```

We can operate the synthesis procedures from the window of DesignCompiler. But it is strongly recommended to use the script file for this job in order to get rid of the operation errors. The following paragraph is an example for the script of synthesizing a module :

```

read -format verilog {"diska1/user/ba057/sd_card/data_clk_change.v"}
current_design = "data_clk_change"
auto_wire_load_selection = "true"
set_fix_multiple_port_nets -all -buffer_constants
create_clock -name "hostclk" -period 32 -waveform { "0" "16" } { "hostclk" }
set_dont_touch_network find( clock, "hostclk")
set_clock_skew -uncertainty 0.5 -ideal hostclk
create_clock -name "sysclk" -period 32 -waveform { "0" "16" } { "sysclk" }
set_dont_touch_network find( clock, "sysclk")
set_dont_touch_network find( port, "xhwrst")
set_dont_touch_network find( port, "xswrst")
set_clock_skew -uncertainty 0.5 -ideal sysclk
set_output_delay 5 -max -clock sysclk all_outputs()
set_output_delay 1 -min -clock sysclk all_outputs()
set_max_area 0
compile -map_effort medium

```

```
current_design = "data_clk_change"
change_names
set_dont_touch "data_clk_change"
write -format db -hierarchy -output "./files_db/data_clk_change.db"
write -format verilog -hierarchy -output "./files_ver/data_clk_change.v"
```

The timing constraints used in the synthesis process are based on the timing setting of the clocks. Usually, if the clock is 10MHz in the real case, then it is recommended to use 13MHz as the timing constraints. The clock must be set as “don’t touch” to prohibit the driving buffers in the clock wire. The driving ability is done in the layout stage by clock tree. The other signal that is critical in timing can also be set as “don’t touch”. In this case, the reset signals are set as “don’t touch”.

For the IP part, the following example is an 256x8 SRAM. We use an empty module to declare its inputs and output, then synthesis it into a empty block :

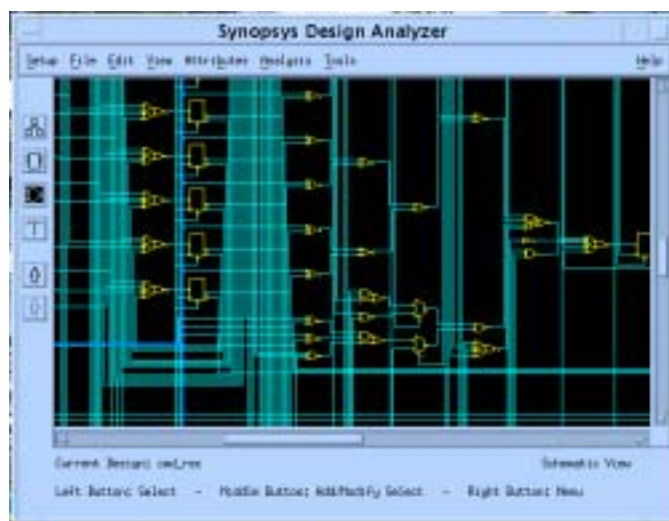
```
`timescale 1 ns/10 ps
module SSRAM256 (DOUT, CP, nCE, nWR, A, DIN);
    output [7:0] DOUT;
    input CP;
    input nCE;
    input nWR;
    input [7:0] A;
    input [7:0] DIN;
endmodule
```

```
read -format verilog {"/SSRAM256.v"}
current_design = "SSRAM256"
auto_wire_load_selection = "true"
set_max_area 0
compile -map_effort medium
current_design = "SSRAM256"
change_names
set_dont_touch "SSRAM256"
write -format db -hierarchy -output "./files_db/SSRAM256.db"
write -format verilog -hierarchy -output "./files_ver/SSRAM256.v"
```

Then we can write all of them in one script file. The DesignCompiler will synthesize all the design for us automatically. But we still need to check the view_command.log to see if there is anything wrong.

Finally, we will get two type of files :

(1) db files for the netlist display in the window of DesignCompiler



(2) the Verilog files which list the cells/registers and wires of all the design as shown below :

以上内容仅为本文档的试下载部分，为可阅读页数的一半内容。如要下载或阅读全文，请访问：<https://d.book118.com/358071143043006031>