Advanced High-level HDL Design Techniques for Programmable Logic

Author :
Darron May, Applications Specialist, ALT Technologies Ltd.

Abstract :
Design Methodologies for Programmable Logic focuses on advanced high-level HDL design techniques for programmable logic. Advanced coding and optimisation techniques for designs created in VHDL or Verilog will be discussed. Using HDLs (hardware description languages) for a programmable logic architecture presents a different set of challenges compared to gate array architectures and this session will explore issues related to the area and/or speed optimisation of datapath and control functions (such as complex counters, arithmetic functions, complex state machines, multipliers, etc.) described in VHDL or Verilog and targeted to various programmable logic architectures. The paper will cover tips and tricks of coding in HDL's for PLD's and FPGA's and will demonstrate common pitfalls and styling issues for HDL's when targeting popular architectures. Examples of when to code technology independent HDL's and when to code technology dependent HDL's will be discussed. Architecture features to avoid or exploit (e.g. RAMS, instantiating IO's or other vendor primitives, using carry chains, special routing resources, set/reset flops) will be explored.

Introduction :
The design methods used when writing HDL's to target programmable parts can have a larger impact on design size and timing than when targeting ASIC's. The methods can even effect the results you get for different vendors of programmable logic when targeting through the same synthesis tool. This is largely down to the architectural differences between different vendors technology and ASIC's. The lowest level building block within an ASIC is a two-input logic gate. The ASIC appears as a sea of two-input logic gates where flip-flops are expensive as they take up as much space as 4 or 5 gates. Within the architecture there is an abundance of routing resources and the delay characteristics are highly variable and routing dependent. A PLD is constructed from a fixed AND/OR (sum of product) array which feeds I/O macrocells. The macrocell in a programmable input/output block that can be configured in a number of ways to take signals on and off the device. The routing between the resources is fixed therefore the delays are fixed and well characterised. Flips-flops are very expensive as they tend to use up a macrocell. FPGA's are made up from fixed arrays of logic blocks. In a course grained architecture these logic blocks can represent 10 to 100 two-input logic gates. The logic block is normally implemented as a RAM look up table, plus flip-flops. Flip-flops are plentiful due to the fact that each logic block contains more than one. The logic blocks are connected by specialised routing resources which actually contain active elements which configure the connectivity. The delay introduced by the routing is highly variable and very sensitive to placement and fan-out. Due to the architecture of FPGA's and CPLD's other features have had to be introduced to make the devices implement certain circuit topology more efficiently. These features are detailed in the

following sections. They include carry chains to implement arithmetic functions, wide decode logic, and special routing resources for global signals such as clocks and reset lines. Traditional synthesis tools are based on algorithms that suit the two-input logic gate building block. They utilise Boolean optimisation and library cell substitution based on timing and area parameters stored in the target library. The fact that the architectures of CPLD's and FPGA's are completely different means that new synthesis techniques are required to make full use of the features of a particular CPLD or FPGA architecture.

Architectural Optimisation :
As has already been discussed, matching algorithms to architectures is very important if the synthesis tool is to utilise a certain device comparable to an Engineer handcrafting the design. HDL specifications only define the language for simulation therefore the synthesis tool is free to implement the logic in any way that maintains the specified functionality. There are a number of important techniques that need to be employed to give the synthesis tool the best chance of realising the optimum area or speed possible. The following few paragraphs contain a short summary of the basic building blocks and features for some of the most common FPGA and CPLD architectures.

Altera FLEX 8K and 10K families have an Arithmetic mode that uses dedicated routing to the adjacent cell. The basic cell also has a cascade mode which uses a dedicated AND function for wide fan-in gates. The sequential elements of the cells each have load enables. Figure 1 shows how the basic cell can be configured to achieve different kinds of logic structure.

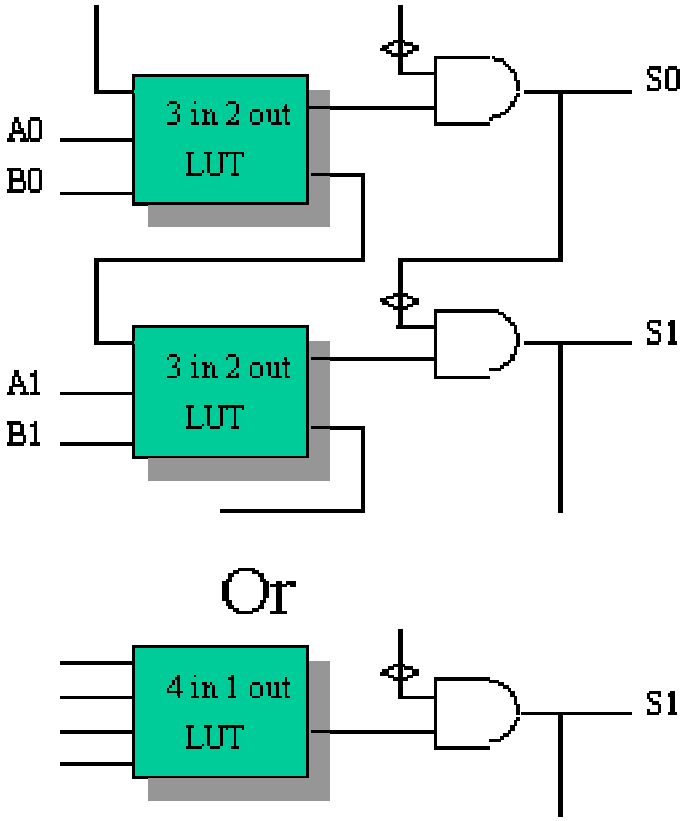

Figure 1 - Altera Building Block

Lucent Technologies ORCA devices have wide LUTs (Look Up Tables) that can be used to reduce delay at the expense of area. There is an Arithmetic mode that speeds up datapath functions and saves area. Each PFU includes special function gates to implement wide elements. There is also a complex flip-flop with multiplexers to allow synchronous reset and load enables. These features are shown in figure 2.

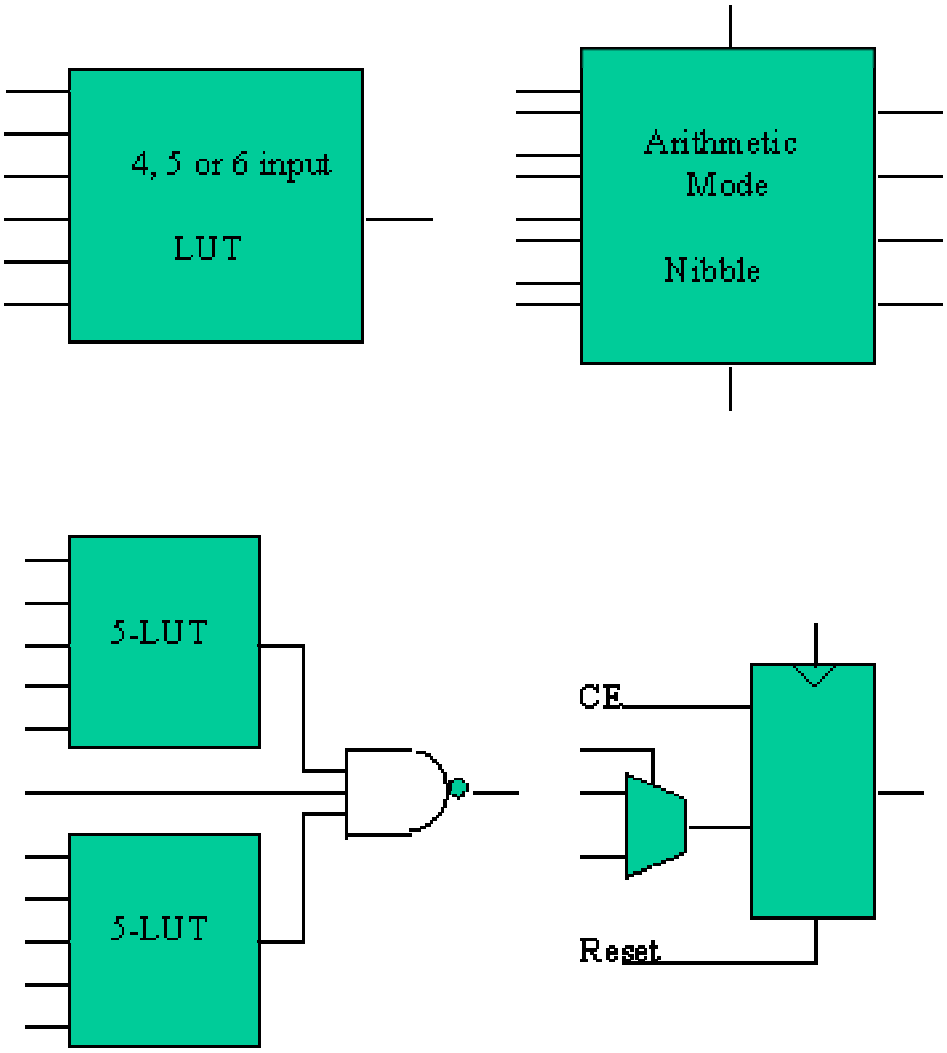

Figure 2 - ORCA PFU Features

Xilinx XC4000[E,EX] devices are made up of two 4-input look up tables and one 3-input look up table with dedicated routing between them. Again this architecture includes special carry logic for arithmetic functions and each flip-flop has a load enable. The structure of the CLB (Configurable Logic Block) is shown below in

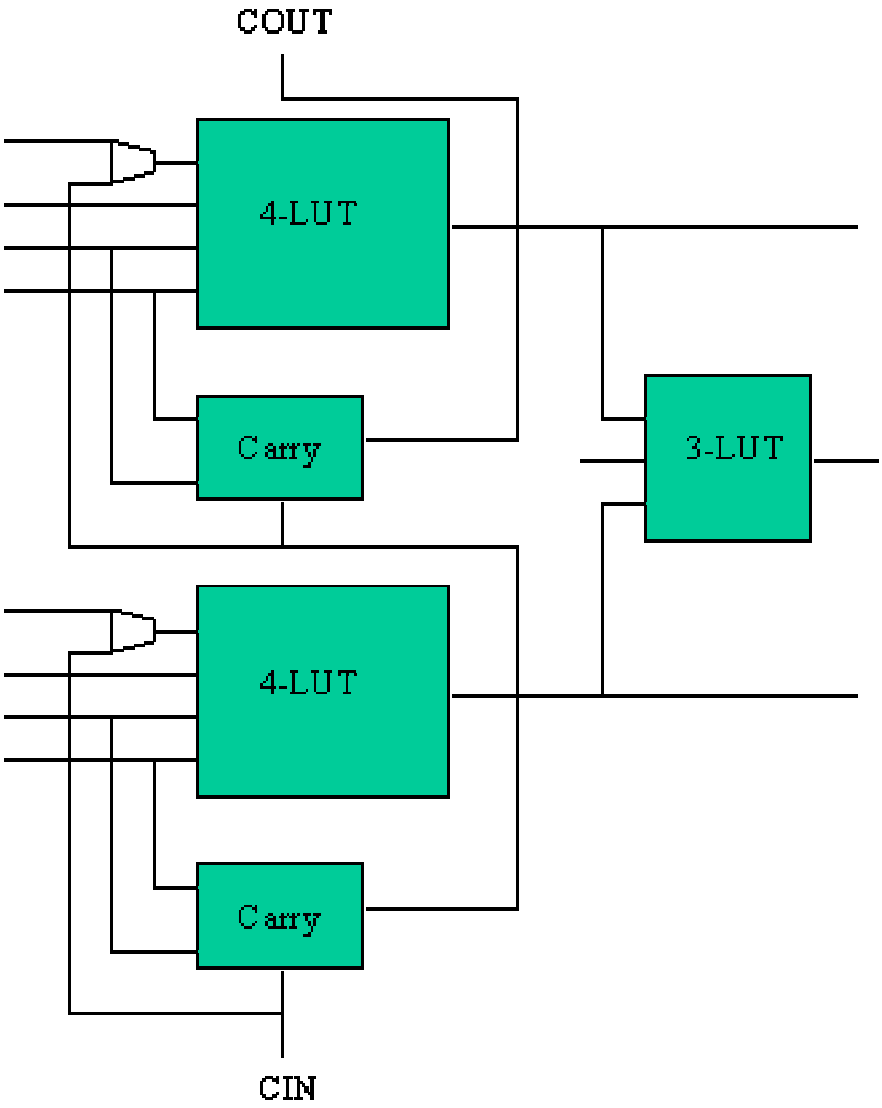

Figure 3 - Xilinx CLB

Implementing datapath logic (adders, multipliers, counters, etc.) efficiently has posed significant problems for FPGA synthesis tools. Synthesis vendors started to develop various solutions which allowed for direct instantiation of macrofunctions in HDL. This is simple to implement but requires effort on the part of the user and is not retargetable. What is necessary is the ability to recognise these functions directly from the HDL and then generate a module for the target architecture. Module generation is a very important factor for good FPGA synthesis as it can make use of the carry chains and other architectural features. Many synthesis tools today do module generation, however module generation on its own does not mean the best result for a certain application. What is required is the ability to fine tune the generated module to the application - what is known as context sensitive module generation. With context sensitive module generation the module is optimised into the given application producing the best fit in terms of both area and speed. If we take for

example the implementation of an adder, in some architectures the smallest adder is not always the fastest. In a fine grain architecture it is possible to produce a carry ripple adder that would produce a small area implementation or a carry look ahead adder which would produce a fast implementation. Context sensitive module generation would use the designers input to decide whether a fast speed or low area implementation would best fit the application. It would also go a bit further than this by producing some kind of hybrid adder depending on the use of the outputs. For example an eight-bit adder may have the four least significant bits connected to logic paths that are speed critical whereas the four most significant bits are connected to lower non-critical logic paths. In this case the module generator would produce a fast implementation for the bottom half of the adder and a slower smaller implementation for the top half of the adder. Some module generators within synthesis tools produce black boxes for the function recognised in the HDL. This black box is used every time the function is found in the HDL and is simply instantiated into the finished netlist. Another example of context sensitive module generation is where the logic at the boundaries of the module is modified by merging and optimising some of the logic that is part of the module into the application. For example there may be some kind of extra clock enable logic connected to a counter with clock enable. It would be possible to optimise this extra control logic into the logic function that is already implemented in front of the sequential elements reducing the levels of logic and therefore increasing the speed of operation.

Resource sharing is another area in which the synthesis tool can make large savings in device area. Automatic resource sharing is done by analysing if-then-else and case statements to find branches that have arithmetic functions that can be shared. These branches are mutually exclusive therefore variables can be multiplexed to use the same arithmetic functions. It is possible to write HDL to share resources however the coder has to think carefully while architecting the design. The simple example below shows two code segments, one written with resource sharing, the other without. The code on the left will produce two adders followed by a multiplexer controlled by the signal sel to switch between the answers from each adder. The code of the right produces multiplexers to switch between vector b and vector c followed by one adder. A synthesis tool with resource sharing will recognise that it can use just one adder with the style of code on the left, and will use it if instructed or if a better synthesis result will be gained. The advantage of having a synthesis tool that does automatic resource sharing is that often it can find more opportunities within the design to share logic than the designer would.

Timing Driven Synthesis is another very important factor in gaining good quality of results. A synthesis tool driven by timing has the ability to take in constraints set by the user to allow it to make decisions during the optimisation stage. Timing can be taken into consideration during the synthesis process by reducing the levels of logic but this in itself does not alleviate the need for the designers input.

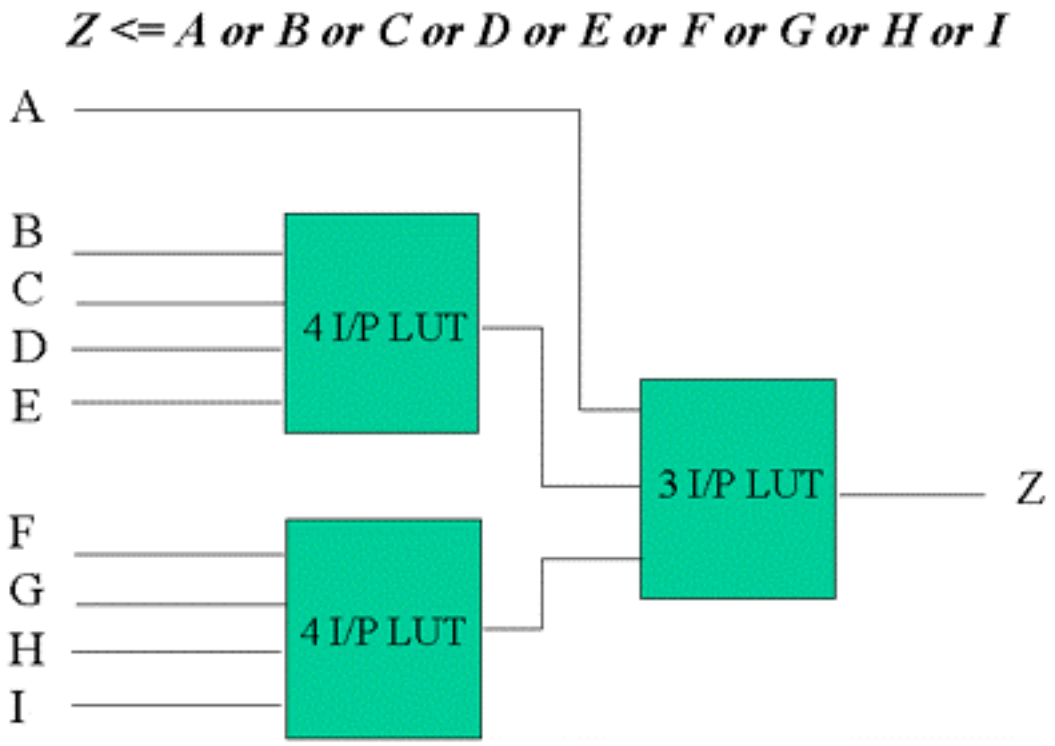

Figure 4 - Timing Driven

ing good quality of results. A synthesis tool driven by timing has the ability to take in constraints set by the user to allow it to make decisions during the optimisation stage. Timing can be taken into consideration during the synthesis process by reducing the levels of logic but this in itself does not alleviate the need for the designers input.

There may be times when certain paths have to be weighted so that the synthesis tool can concentrate on them during the optimization process. If the user has no way to feed the synthesis tool constraints then there would be no control on the final implementation. This is shown in the simple example above in figure 4. The synthesis tool has produced the best fit for this nine-input logic function in an XC4000 device. However input H may be more time critical than input A which has the fastest path to Z. Timing driven synthesis allows us to make H more critical than the other inputs to ensure that during the synthesis process H becomes the input to the 3-input LUT giving it the fastest path to Z.

As has been said previously, most synthesis tools use the same algorithms for implementing FPGA's and CPLD's as they do for ASIC's. This normally involves a standard approach of matching library elements onto a generic form of the circuit. With ASIC's this approach is fine due to the fact that the target library is made up of macros that are larger than the building blocks of the device. For example implementing a two-input OR gate would be done with a few transistors. With an FPGA or CPLD the library used would still have the same kind of macros as the ASIC library, however the way they are implemented is completely different. The two-input OR gate would be implemented

in a course grain FPGA in a look up table that would have the ability to implement more than just the OR gate depending on the topology of the circuit. This means that once the circuit has been implemented a further stage has to be done to map this logic into the building block of the target. Some synthesis tools try to do this second stage, others rely on the vendors tools to do this mapping. Either way the method is not as efficient as a tool that can directly map to the target architecture using the building blocks of the technology. Direct synthesis uses the building blocks of the technology to tile the circuit. This can be seen in figure 5 where the inverter and NAND function in the top left hand corner end up in both look up tables. Replication of theses functions have no impact on area due to the nature of the look up tables and end up making the implementation faster and more efficient.

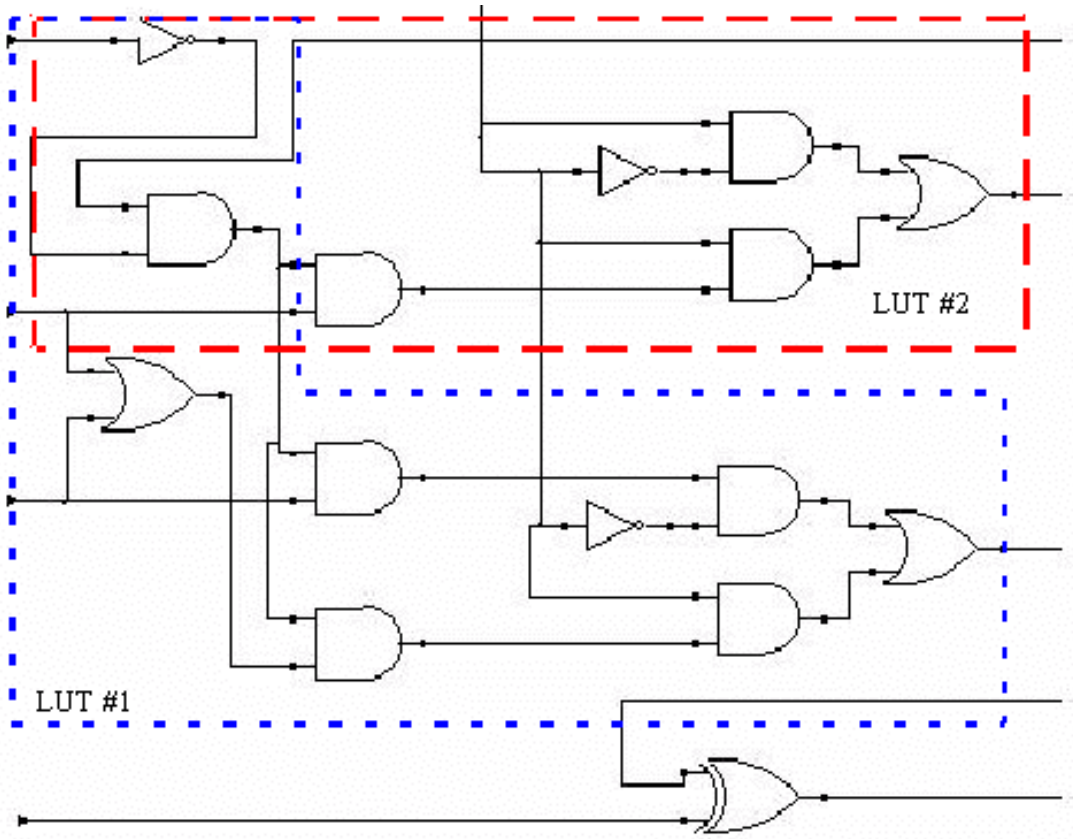

Figure 5 - Tiling Process

this logic into the building block of the target. Some synthesis tools try to do this second stage, others rely on the vendors tools to do this mapping. Either way the method is not as efficient as a tool that can directly map to the target architecture using the building blocks of the technology. Direct synthesis uses the building blocks of the technology to tile the circuit. This can be seen in figure 5 where the inverter and NAND function in the top left hand corner end up in both look up tables. Replication of theses functions have no impact on area due to the nature of the look up tables and end up making the implementation faster and more efficient.

The benefit of this approach over the traditional synthesis approach can be seen in the following example illustrated in figure 6. Boolean optimization reduces the two equations down so that the function of (B or C) NAND NOT(A) is only implemented once. This is perfectly acceptable when implementing the function in ASIC technology because each gate takes area and the only disadvantage is the fact that the fan-out will be increased on the output of the NAND gate. However, in a course grained FPGA, for example, it would be possible to fit each equation into one four-input look up table as would be implemented from a handcrafted design. If this optimization had been carried out in this way then the mapping software is forced to place the common function in one look up table and the other two parts in separate look up tables. This causes extra delay due to the two levels of logic plus an area impact as only two look up tables are required. Whether this replication is done so that the functions are mapped in the most efficient way is down to the vendors backend tools or the mapping algorithms in the synthesis tool. Obviously using a synthesis tool that maps directly to the building blocks of the technology saves runtime and produces better optimized results.

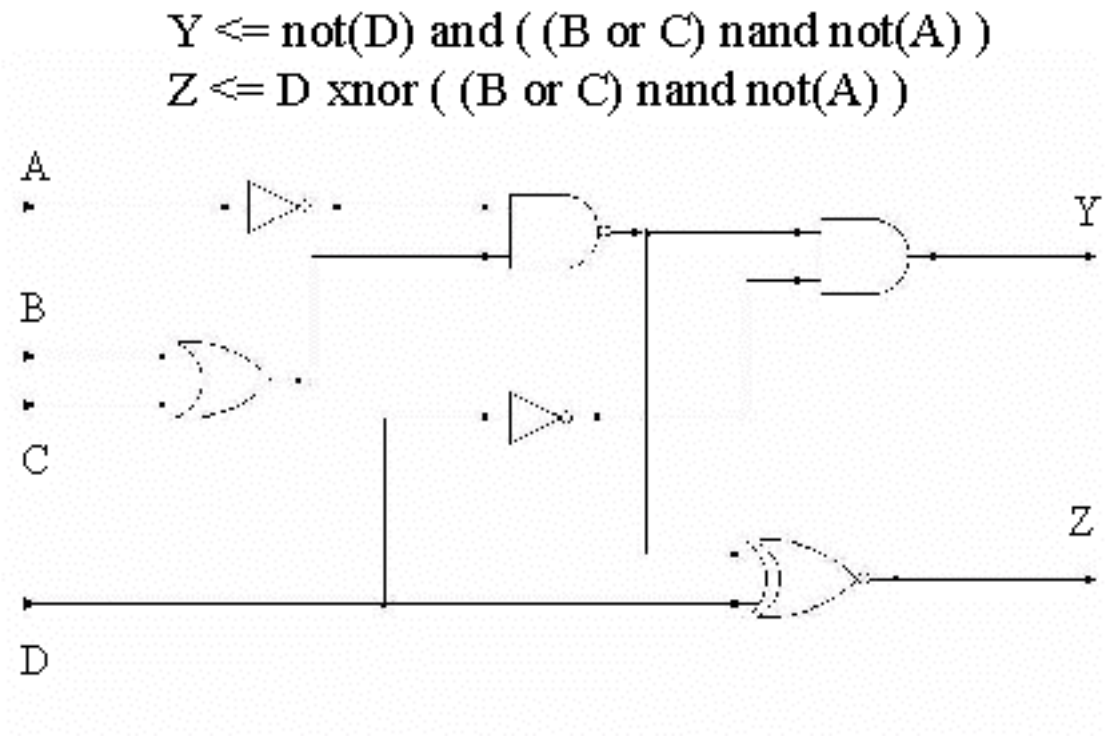

Figure 6 - Direct Synthesis

Using Architecture Features: This section details some of the features within FPGA's and CPLD's that should be considered when coding your HDL and also some general

facts about certain architectures. Many devices have global signal routing resources, to avoid interconnect delay, which need to be utilized to make the most of the technology. They normally have global set and/or reset lines which generally restrict the designer to use a single set or reset throughout the design. These reset lines are asynchronous, therefore if the designer chooses synchronous resets, extra logic is required.
If the code does not follow the restrictions above then the extra high fan-out nets have to be routed via the data routing channels. This will lead to area, performance, and routing problems. Another important global resource is the clock routing. Never gate clocks in your design as this will prohibit the use of dedicated clock routing resources, which may introduce clock skew in devices that are essentially skew free. Most function blocks within FPGA's have sequential control lines that can be used to get the same effect as a gated clock. The two code segments below show examples of both gated clock and clock enable designs. The code on the left hand side produces an AND function that stops the clock reaching the sequential element. This kind of design is not good practice and could result in glitches on Gclk if the enb signal changes asynchronously with respect to Gclk. The code on the right hand size uses a gate in front of the data input to the sequential element that causes the last value to be fed back to the latch if the enb signal is not active. The synthesis tool should recognize this and use the clock enable available in the logic block.

```
Gclk <= clk and enb;
if Gclk'event and Gclk = '1' then
      q <= data;
end if;
```

BAD

```
if Gclk'event and Gclk = '1' then
      if enb = '1' then
      q <= data;
      end if;
end if;
```

GOOD

Be aware that Lucent PFU's contain 4 DFF's and there is one clock enable line per PFU. This has an impact on mapping as DFF's with different enables cannot be packed into the same PFU. Also consider that you may need to help synthesis by instantiating components, for example GSR or STARTUP blocks for Lucent and Xilinx, GLOBAL buffers for Altera, CLKINT for Actel and HIPAD for QuickLogic. If you adhere to the rules for global reset in your HDL code then a good synthesis tool should be able to automatically use the GSR facility in the device. The code below shows a good and bad example for the use of global resets within Xilinx and Lucent devices. Try to use one reset within your device so that the design can benefit from the use of this global facility.

```
always @(posedge clk or posedge reset1)
        if (reset1) then
                q1 = 0;
        else
                q1 = data1
always @(posedge clk or posedge reset2)
        if (reset2) then
                q2 = 0;
        else
                q2 = data2;
```

BAD FOR XILINX & LUCENT

Pad cells are typically programmable cell resources that may be input or output buffers, registers, tristate drivers, or high drive pads. Good HDL synthesis tools automatically select I/O cells to insert where appropriate. If your synthesis tool does not do this, you have to instantiate the cells manually otherwise you will lose out on the use of the registers within the I/O. Not using the flip-flops in the I/O has an impact on the area of the device because the design has to use flip-flops in the core which could be utilised for other functions. The timing is also effected because instead of having a register that is directly on the output pin, an internal flip-flop has to be routed out of an output buffer causing extra delay. This delay will also be dependant on the placement of the flip-flop within the core. Some technologies such as ORCA do not have flip-flops in the I/O as they rely on being able to place the flip-flop close to the boundary of the device. The synthesis tool should have the ability to take input from the designer so that placement information can be passed onto the place and route tools.

Some FPGA's have internal tristate buffers which can be used to implement wide multiplexers. No synthesis tools make use of these feature therefore it is necessary to instantiate these buffers manually in the HDL making it non-portable. Using these buffers can make routing easier and in Xilinx devices will use fewer cell resources. When you do utilise the tristate buffers you need to be careful of bus contention as the select line timing becomes important. Device initialisation behaviour can change and power-up is often not tested during simulation. If you consider a multiplexer whose data lines are all zero, but selects are 'x', the tristate drivers could be driving if the selects are not initialised.

Course grain architectures have special case hardware to improve implementation of datapath elements. It has been stated previously that synthesis tools should use these features for operators. Carry chains and cascade logic implement fast paths that would otherwise be implemented in look up tables. If your synthesis tool does not make use of these features then it is possible to manually access them by the instantiation of the carry or cascade primitives or by using vendor generators, such as LPM (Altera), XBLOX (Xilinx) and ACTGEN (Actel). Remember 'out = a + b' is much easier to specify so bare this in mind when selecting a

synthesis tool. The real challenge for synthesis is to exploit carry and cascade features for random logic. It is important to select an FPGA/CPLD architecture to fit the application you are designing. Antifuse devices are fine grain and do not need dedicated logic to improve speed, they are fast, but at the expense of area.

FPGA and CPLD architectures normally include, in some form, a section of memory that can be configuredas RAM or ROM. These RAMs/ROMs are much more efficient for implementing such functions as tables, fifos, and register files. Even a small register file could end up utilising all the available registers in an FPGA.
Currently no synthesis tools automatically infer RAMs and ROMs from the HDL; access is through LPMs or direct instantiation. It is possible to use two dimensional arrays to produce RAM elements, in some synthesis tools, however this uses the storage elements within the logic blocks and not the RAM available in the technologies. If the synthesis tool did infer these RAM elements then a number of rules would have to be followed. Normally if a designer has chosen a particular architecture for its RAM capabilities then the application would be well understood. Trying to keep the HDL independent so that it could be re-targeted would be difficult and probably not possible due to architectural differences.

Logic and register bits come from different resource pools within the architecture therefore for most designs register bits can be used freely (unlike ASICs) to reduce loading and encode state machines. Finite StateMachines implemented in CPLDs and encoded as one-hot will usually lead to the fastest operational speed, however routing is harder. Other encoding schemes can work well, such as grey encoding. With FPGA architectures, one-hot encoding is normally better, however it is important to have control over the stateencoding when using enumerated types in VHDL. The best encoding scheme for a given application candepend on many different factors; the number of states, the amount of next state logic, speed, area etc.

Remember a synchronous design is always a ‘happy’ design, it is not possible to model asynchronous circuitry and guarantee timing.

Technologies tips/tricks and caveats :
The following section covers various hints and warnings when using different Vendor Architectures. All FPGAs and CPLDs have free resets in their sequential components, therefore they should be used. If these asynchronous resets are not used then area and performance will be lost as extra logic and routing resources will be required. Presets normally cost an input to the look up table, therefore try to incorporate presets in your design by changing the states of signals.

There are many times in an HDL methodology when you end up with simulation mismatches and errors dueto modeling differences. Very often these problems are only down to a few commonly made mistakes. The first is that of initial value assignments made within the RTL code. Synthesis hardware does not honour initial values as the initialization can not be tied to any physical signal. Some sequential circuits will never come out of an un-initialised state if you do not pay careful attention to the reset. Take care when encoding state machines in FPGA’s due to the power-up reset to registers. If you fail to encode the all ‘0’s state then the implemented circuit will never leave this state, however the RTL version will function correctly. The

following two pieces of VHDL code show two methods of ensuring that the all ‘0’s problem never happens. The first method uses an enumerated type and therefore still leaves the encoding scheme up to the synthesis tool. However because the enumerated type has given names to three dummy states so that there are 2Nstates, the others clause will ensure that we move out of the all ‘0’s state. If you chose a one-hot implementation then you need to ensure that the reset takes you away from the all ’0’s condition. The synthesis tool should do this for you as long as it understands FPGA/CPLD architectures. The secondpiece of code uses constants to encode the states. In this example we have ensured that the reset state is the all ‘0’s state therefore we will automatically move into this state on power-up.

Beware of the global reset facility because some translations to simulation netlists struggle to properly connect GSR or STARTUP signals. Remember an "x" mismatch is usually an initialisation problem whereas a ‘1’ vs. ‘0’ mismatch is more likely to be a logic or timing problem.

The semantics of code written for simulation does differ from the semantics of synthesis code. This means that it is possible to get mismatches between RTL code and the implemented circuit. The first example of this is when signals are missing from the sensitivity list of a process in VHDL or an always block in Verilog. In simulation a process can only be triggered by a change in state on any of the signals in the sensitivity list. The logic that is implemented by the synthesis tool has to be based on all inputs. This means that if a signal that is an input to a process is not in the sensitivity list of the RTL code then there may be differences between the two simulations.

```
process (a, sel)
if sel = ‘1’ then
  out <= a;
else
  out <= b;
end if;
end process;
```

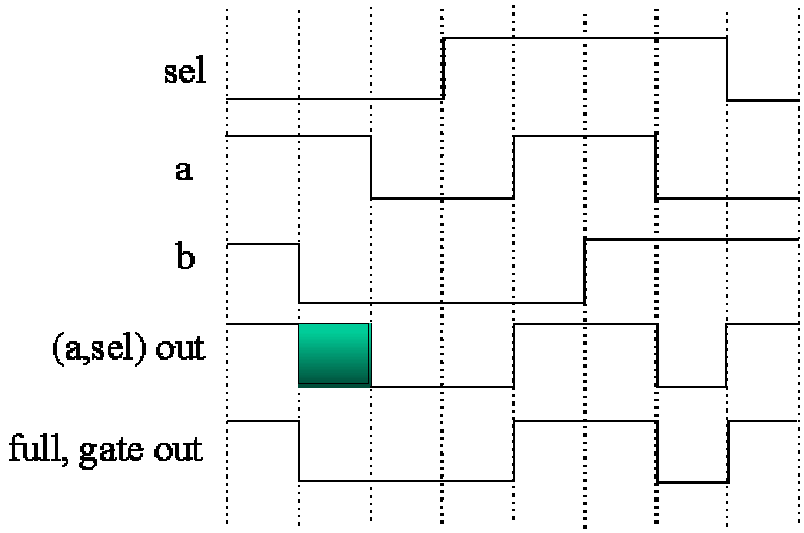

Figure 7 - Missing Sensitivity Variable

The example above, in figure 7, shows the RTL model of a two input multiplexer with signal b missing from the sensitivity list. If the patterns in the simulation shown are input into the model it can be seen that there is a difference in the results. This is because when the falling edge of signal b occurs the process is not triggered therefore the if-then-else statement is not processed and the out signal keeps its value until the falling edge of signal a. This is a very simple example, however there may be times when the

function simulated at the RTL level is what the designer wanted however when the implementation is simulated the results are different. Good synthesis tools will advise the designer that variables are missing from sensitivity lists.

The way some sequential registers are modelled with the HDL's can mean differences in behaviour between RTL simulations and the gate level model in some circumstances. For example the following always block will not behave in the same way as the silicon if reset falls while set remains high.
If set goes high then the block will be triggered and the q output will be set to '1'. If reset goes high while set remains high then the block will be triggered and the q output will be reset to '0'. However, now if the reset line falls, the q output will remain low even though set is high which means that the q output should be set to '1'. In the implemented version the set and reset will function asynchronously therefore the element will set again.

When designing with SRAM architectures it is important to try to match the physical and logic hierarchies. It is very easy to carry out a top-down approach with HDL's, however if there is no balance in the logical hierarchy then a lot of time may be needed in the floorplanning tools to get the best placement. The devices architecture views the world in 4 quadrants thereby splitting the top level module and minimising connection between instances, the routability will be improved and the placement will be made far easier.

Estimated gate counts are calculated in different ways by different Vendors. They are based on an average number of gates implementable by a logic block multiplied by the number of blocks in the device. The number of gates in the I/O ring, for example the flip-flops, are then added to this figure. This estimate is not based on circuit topology therefore care needs to be taken when estimating which device from a family will be required for a design.

Altera's FLEX 10K and 8K are an SRAM based technology. There are no internal tristate drivers within these devices therefore care must be taken when tristate busses are implemented in HDL. It is possible that the devices may blow up when these buses are implemented as multiplexers. Remember fixed pin location assignment can cause routing difficulties within these devices. The best methodology, if possible, is to ignore pin assignments to establish initial area and delay baselines. These devices have EAB's (Extended Array Blocks) which are a separate logic resource for implementing RAM's, ROM's and look up tables. These blocks can be used to help you to fit the device but at the expense of sacrificing some performance. Altera do add the estimated capacity of these blocks to the total estimated gate capacity of their devices therefore, if they are not used, it is not possible to utilise the device to the maximum levels printed in the databooks. The larger CPLD devices have ample cell resource but sometimes suffer from not enough routing resources.

Many Xilinx devices have inadequate routing resources which result in delays being very unpredictable due to very long routes. The newer families have been improved greatly so that this is not such an issue. The use of the internal tristates can help with utilisation, which necessitates having to convert your wide

multiplexers to tristates when routing or fitting becomes a problem.

Lucent Technologies ORCA PFU's do not have local VCC or GND taps which means that they have to be routed - this can cause routing congestion. The mapping program within the Foundry software tools offers another set of options which can compliment or compete with your front end synthesis optimisation. It is best to turn off most of the backend optimisations initially, maybe adding some of them if you have fitting problems.

Actel and QuickLogic devices can be fully utilised and still route. The delay is very predictable, however delay is very sensitive to fanout so lower fanout limit to 6-10 for highest performance. With fine grain architectures there seems to be an under estimation in gate count which means that the devices can normally end up holding more gates than the databook suggests.

Conclusions :

FPGA synthesis requires completely different optimisation techniques than for ASIC's. An important and difficult goal for FPGA synthesis is to let you design in a technology independent way but still utilise all of the special case logic to produce good results. There are a number of techniques, discussed in this paper, that are required to produce results as close to handcrafted as possible. If quality of results are important to your design then you should select a synthesis tool that employs these techniques. When using HDL's to design FPGA's and CPLD's it is important to understand the vendors architecture because the design methodology used during coding can still have a significant effect on the overall quality of results.