Model Based Design needs High Level Synthesis

A collection of high level synthesis techniques to improve productivity and quality of results for model based electronic design

Steve Perry
European Technology Centre
Altera Corporation
High Wycombe, UK
Steven.Perry@altera.com

Abstract— **Model Based Design tools based around Simulink from The MathWorks are a popular technology for the creation of streaming DSP designs for FPGAs, since they offer the promise of rapid design exploration through immediate quantitative feedback of algorithm performance. Current tools typically use a library of components that reflect an explicit representation of the underlying FPGA device features. This is undesirable since the designer is forced to mix implementation and architecture, and leads to long design cycles and non-portable results. This paper shows that introducing techniques of high level synthesis allows a more elegant design at a higher level of abstraction. This results in fewer components needed for a design which translates into a faster design cycle, more portable designs and fewer defects. Pushbutton clock frequencies of up to 500 MHz are achieved without detailed knowledge of FPGA architectures.**

Although the capabilities described are embodied in the DSP Builder tool from Altera, this paper describes the technology involved rather than the details of the tools. Four major technologies are described: a latency-insensitive system representation, the module level internal representation with associated transformations, hardware retiming, and lastly a FIR filter design tool layered on top.

Keywords- Model Based Design; High Level Synthesis; FPGAs; Technology Mapping; Retiming; FIR Filter Design.

I. INTRODUCTION

One significant segment of electronic design relates to digital signal processing (DSP) designs where a sequential chain of processing blocks such as filters can work on a data-stream as a deep pipeline. Such applications include digital front-ends for wireless systems, radar applications and increasingly sophisticated medical instrumentation.

The first generation of design tools such as DSP Builder from Altera and System Generator from Xilinx are based around Simulink from The MathWorks. The tools offer the advantage of integrated simulation, rapid RTL creation and productivity features such as hardware-in-the-loop simulation.

The component libraries are typically literal representations of the underlying hardware capabilities of the target device. To achieve high performance designs necessitates understanding FPGA device details such as the delay characteristics of adders, the number of pipeline stages required and modes available for a multiplier to achieve a certain clock frequency. The designer must then take care of balancing delays in the circuit. During the FPGA place and route cycle timing may be an issue, and one of the best ways to improve timing is to introduce register stages. This then has the consequence that the delays in other parts of the design need to be rebalanced.

In a typical design flow a clean architectural idea cannot be captured as the engineer might hope, since the implementation intrudes into the design. This mixture of architecture and implementation has a particularly undesirable impact on portability, since the implementation detailed for one FPGA family will differ from those of another family. Even within speed-grades of the same FPGA family different pipelining will be required to meet the same clock frequency. Maintainability is reduced too, and this can dilute the benefits of an integrated toolflow.

To address these concerns, and others, a new approach is required. The goals of this new tool are to permit the designer to capture only the architectural information and automate the implementation. The idea is to design a function using a set of abstract functional units, such as multipliers, adders, counters, delays, memories etc. These functional units are untimed and allow the designer to capture just the desired architectural intent. This internal representation (IR) is essentially a data flow graph that can then be transformed into a lower level version that targets the capabilities of the particular FPGA target. This parallels the flow within software compilers, such as gcc. Transformations range from the simple, such as constant propagation, technology mapping to specific FPGA hardware features such as hard multipliers, through to novel carry-chain length reductions and to sophisticated retiming techniques using an Integer Linear Programming (ILP) formulation.

II. TECHNOLOGIES

The remainder of the paper covers four major areas, and shows the applicability of each. The four areas are latency-insensitive system representation, the module level internal representation, hardware retiming, and lastly a FIR filter design tool that is layered on top.

978-3-9810801-5-5/DATE09 © 2009 EDAA

A. *Latency Insensitive System Representation*

In this design environment pipeline delays are automatically inserted for the user. So, at design time the exact delays are not known. Moreover, since we want to design once and retarget that module to any one of several FPGA device families (including ones not yet developed) we can never know at design time what the pipeline delays are. Therefore we encourage the designer to avoid thinking about explicit delay balancing and cycle counting, and instead use a simple signaling protocol to communicate between modules. This protocol consists of three elements – data, valid and channel. This collection of signals conveniently permit us to build systems that respond to their inputs and produce outputs that other blocks can use as inputs in a consistent way.

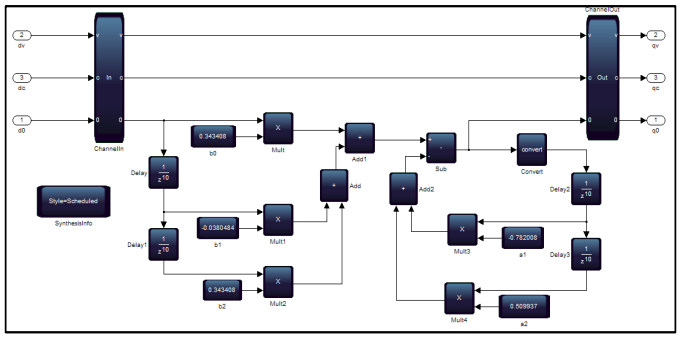

Figure 1. Simulink Block diagram of Infinite Impulse Response (IIR) circuit using DSP Builder Primitive Blocks

In Figure 1 the ChannelIn and ChannelOut blocks gather these signals together to indicate that they will be aligned in the same clock cycle. This is the only guarantee that the tool makes about any timing relationship. The channel signal provides meta-data about the content of the data wire in any cycle, indicating the contents of the data on a time-division multiplexed wire. An example of how this is used to label the data for a 4-channel filter is shown in Figure 2. By preserving this information through the design it makes it easy to strip out certain channels for display, or inject new data during debug. Also real and imaginary components of complex channels can easily be identified by examining a single bit if they are labeled consistently.

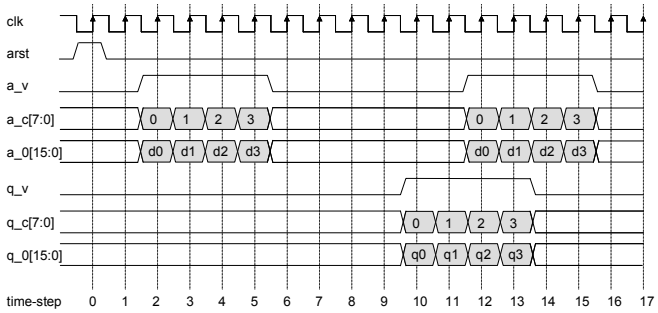

Figure 2. Inputs, a, and outputs, q, for a 4-channel filter

If a set of blocks are chained together the second and subsequent blocks will receive their input signals an unknown number of cycles after system reset, depending on the latency of the upstream blocks. This will not impact on the correct functioning of the system since there is no top-level controller attempting to count cycles. This is an essential characteristic of latency insensitive systems and allows the tool to automate the pipelining of designs in a very aggressive way and yet maintain correctness over a wide range of parameters.

B. *Module Level Internal Representation (IR)*

The module-level IR is the key data structure that represents each module in a system. Each module is logically bounded by ports that implement the <data, valid, channel> protocol. Within the module, the cyclic data flow graph (DFG) is represented as a netlist of functional units. There are around twenty types of functional units, each parameterized by port numbers, bit widths, latencies and so on. Each functional unit type has a fast C++ simulation model, knows how to write itself out in RTL, and is able to give a resource estimate for itself.

There is some subtlety in the choice of components for this library since we want to provide a useful abstraction above the hardware for entities that are difficult to design, be small enough to be manageable, and yet complete enough to build useful designs. Many functional units are obvious – adders, multipliers, etc. Some blocks are needed for efficiency in RTL such as counters. Note that a counter can be built from adders and comparators, but it turns out that to build efficient counters and sequencers some care needs to be taken. Techniques such as comparing to the end count needs to be performed in the preceding cycle. This is a detail that is best hidden, since it is all too easy to get wrong if performed by a casual user. Additionally, these components contain feedback loops within them. If these feedback loops are made visible to the schedulers then this would introduce unnecessary complexities.

Using the IIR example in Figure 1, the visual representation in the Simulink library has a one-to-one mapping to the underlying functional units (though not to the underlying FPGA hardware target), and so provides an insight into how simple the initial representation is. In particular, note that lumped delays are used in the design. Each of the adders and multipliers has zero latency. This makes debugging the design very simple, since all the relevant signals can be viewed at the same instant of time, and the pipelining does not intrude into the logical propagation of signals as in conventional RTL simulation. We will follow this example through the phases of the toolflow.

The structural optimization phases work by examining each function unit in turn. For each unit a set of rule based optimizations are performed. These can take the form of a simple logical or arithmetic reduction, a technology mapping rule, or a structural transformation such as adder-tree to a reducer tree.

For example in the IIR example there are two pairs of multipliers each feeding an adder. On an Altera Cyclone FPGA these will be translated literally into separate multipliers and adders. On Stratix II FPGAs these can be efficiently mapped into a single DSP block.

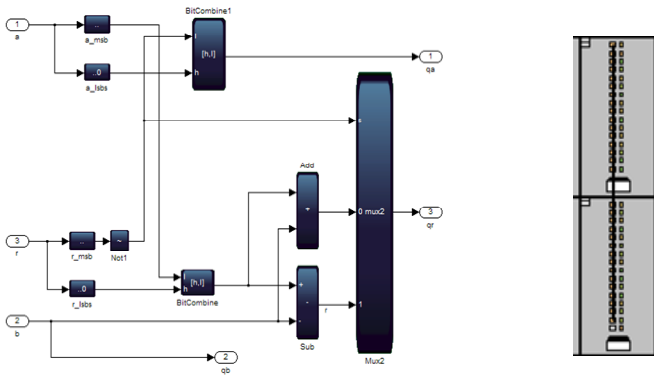

Figure 3. Technology mapping of (a) add-subtract-multiplexer components into physical add-subtractor

Another example in Figure 3 performs the mapping of an adder and subtractor and a multiplexer into a single physical add-subtract block. The advantage of this is that some devices may not have add-subtract blocks and we want to maintain portability.

Compared to RTL level technology mapping we have more scope for performing these transformations since, at this stage, the design is not registered which can often prevent certain mapping rules matching.

In order to meet the required timing goals the tool creates a design whose critical path is estimated to be longer than the required period. Typical critical paths in FPGAs consist of a routing component and a fixed function-dependent delay.

Firstly, let us consider the routing delay. Since we are executing this tool flow before exact placement details are known we use an empirical estimate for the likely routing delay on a particular FPGA family and speed-grade. We have little control over this, except to ensure that in any register to register path there is only a single routing hop. This is achieved by ensuring that each functional unit is registered at its output.

Secondly, we need to control the delay through each functional unit. Delays through multipliers and memory blocks are fairly uniform and can be deterministically pipelined to the required clock frequency. The only significant varying functional unit delay is that of adders. Note that subtractors, comparators, rounding blocks are all implemented as adders, and so this set includes a significant proportion of the units in a module. The delay is a function of the number of bits in the adder. The tool pipelines long adders into smaller sections that will meet the desired clock frequency. For example a 42-bit adder, as typically found in a FIR filter will be split into two 22-bit adders for the Stratix II FPGA at 250MHz. At 500 MHz it will be split into 6, 7-bit adders. Each adder section is a ripple-adder and the carryout from one section is registered before being used to feed into the next most significant adder section. Note that 2 extra bits are required on each adder section for the carry-in and carry-out bits.

To determine how many stages to create we derive an empirical rule from timings extracted from the timing database in the place and route tools. The chief components are the delay to get onto a carry chain and the delay for each step along the carry chain. Note that where logic elements are grouped into blocks, the delay for each step varies at the block boundaries.

By then finding the maximum number of adder-bits for a given device and speed-grade we can determine the maximum length of a carry chain for our desired clock period.

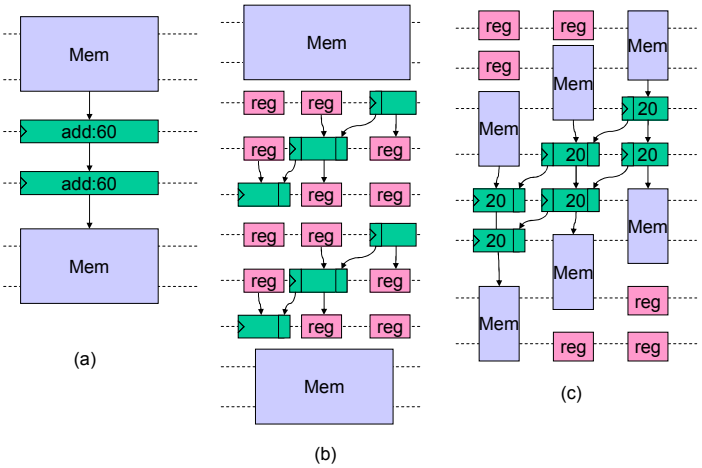

Figure 4. Pipelining a series of 60-bit adders (a) simplistically (b) and more optimally (c).

We might expect this to increase the latency of the design considerably. In practice, this is mitigated by continuations of the bit-level pipelining splits through other functional units. In Figure 4 (a) an original pipeline is shown created by a designer. Two 60-bit adders are each split into three 20-bit adders to achieve a higher clock frequency. If performed locally many pipeline registers need to be inserted as shown in (b). More optimally, the new adder sections, shown in (c) tessellate with each other in time, so that additional pipeline register requirements are reduced. Furthermore, by continuing the bit-boundary splitting through components such as delays, memories, multiplexers, and logic functions it turns out that additional register cost is reduced further.

C. *Hardware Retiming*

At this stage of the mapping process we now have a DFG that has the designer's delay blocks still preserved, and each functional unit has zero delay. The next stage of the automation distributes the delays around the circuit so as to satisfy the conditions that each functional unit must have sufficient delay for itself. For example each adder stage must have one delay, and each multiplier must have a number of cycles that is dependent on the FPGA, but is typically two to four cycles. These delays have zero implementation cost since the registers are coupled with the logical function, and so consume no extra logic cells. Any additional delay is lumped into as few additional delay blocks as possible to reduce the implementation cost.

Cyclic paths must be implemented in such a way that the original delay around the loop is preserved in the final implementation. All paths must satisfy the relationship that any paths originating from the same function unit and are incident

on different inputs of a functional unit must be equal, ensuring correct behavior of each operation.

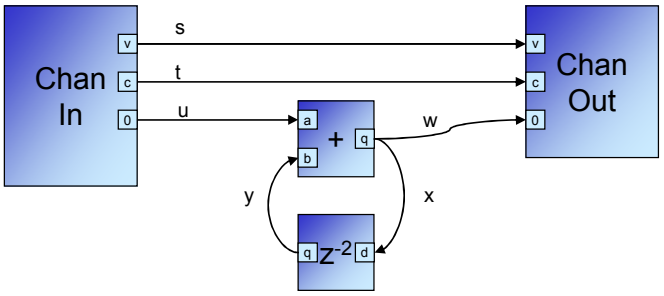

Figure 5. Simple primitive design with delay

Two special-case paths are those paths that arrive from inputs through a ChannelIn block. The latency insensitive protocol guarantees that these signals arrive during the same clock cycle. Similarly, signals arriving at the ChannelOut block, and thus be presented at the output of our module, must be presented at the same cycle. Constraints are added to ensure this. Note that we don't mind which particular cycle.

In general, the implementation may require delays inserted at each input of each functional unit. So we create delay elements at each input, and try to solve all the constraints at the lowest cost by minimizing the number of non-zero latencies assigned to these delays.

These equations are generated by enumerating each loop and path in the circuit. For each path traversed an equality that ensures the delays added plus the functional unit latencies match the sum of any original user-generated delays in that path. These are then passed to *glpsol* to generate the delay latencies using Integer Linear Programming techniques.

For example, in Figure 5, if L is the latency that the channelOut block will eventually be given, then we have the following equalities

$$s = t = L \quad (1)$$
$$u + 1 + w = L \quad (2)$$
$$x + y + 1 = 2 \quad (3)$$

where s, t, u, w, x, y are delays on the respective edges of the DFG.

The implementation cost of each delay block is the product of the delay depth times its bit-width. The sum of this implementation cost over all possible delays is the total implementation cost. These can be solved for total minimal delay implementation cost by setting L = s = t = y = 1, and x=0.

Any zero-latency delays are removed (for example x), and we now have the final technology mapped and pipelined DFG.

Very accurate resource estimates can be provided at this point since we have a structural netlist that subsequent RTL synthesis must respect closely since each functional unit has a tightly controlled implementation. The DFG can be simulated using the internal simulator. This provides bit and cycle accurate simulation that can provide a faithful simulation model for use within Simulink, or externally as a 'C' model in a System C flow for example.

D. FIR Filter Synthesis

The toolflow described above is accessible from Simulink as shown, but also as a set of C++ libraries. A netlist is created by allocating a series of net and functional unit objects and connecting them programmatically. Using this method of module creation it is possible to build domain specific module generators. In this section we describe a novel technique for creating a wide range of efficient FIR filters.

As an overview, the flow takes a filter specification as input via a Simulink GUI that specifies high level specifications such as number of taps, number of channels, bit widths, interpolating, decimating or fractional rate changes, symmetry or anti-symmetry, and L-band optimizations. A set of equations for the calculation can then be derived. Then, by tracing the data flow through the filter, logic can be created that will perform the required function. We can create a high level DFG and use this as input to the technology mapping and pipelining functionality.

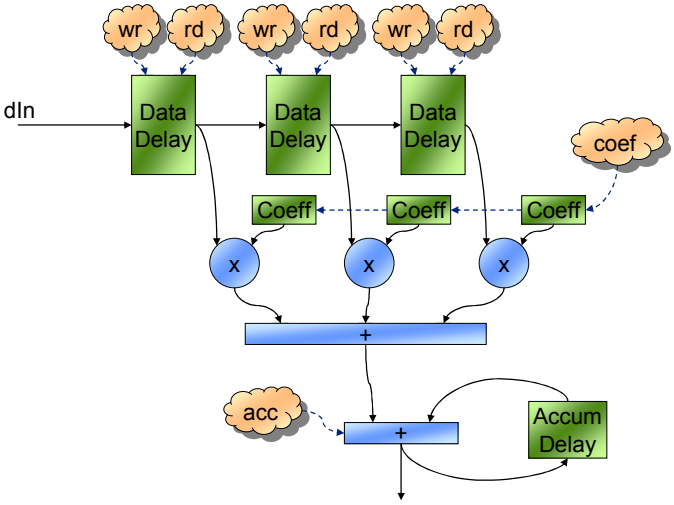

Figure 6. Basic FIR filter template

The complex part of the FIR design is the delay line. As shown in Figure 6, the delay line must pass data to the multiplier-add circuitry, but also to the next element of the delay line. Data is often presented several times if the input sample rate is low enough and folding is used. Data is sometimes skipped if the FIR implements a rate change. Existing module generators typically configure a parameterized template design that has been created by an expert designer. Creating these templates is a difficult and very time-consuming task. Accounting for the many corner-cases that occur leads to many special cases that prove to be difficult to maintain for the IP vendor. This often leads to sub-optimal implementations.

We take a different approach and use a form of trace analysis to build a specialized filter at configuration time. By examining the data input timing we can plan when the data is written to each memory in the delay line, and to which address. Also, by looking at the FIR data requirements at each cycle we know which sample must be read from each memory each

cycle. So we can build up a cycle by cycle view of all the data movements. Since we know the contents of each memory, we can now infer the addresses that must be applied to the memory in each cycle. From this we can identify the pattern of repeating addresses that are required.

Then we decide how to generate the addresses for the memories. Sometimes a set of summed counters will be the smallest solution, and sometime a lookup-table driven by a counter will be smaller.

An example of a typical low level irregularity that makes the template style difficult is when one delay line stage cannot read the data from the previous delay line stage early enough. This is FPGA-dependent because it depends on the latencies of the memory elements. In this case our tool detects the situation and reads its data from the preceding delay line stage.

Some high level decisions are made too. For example an interpolating FIR can be built from multiple phases (essentially a set of smaller FIRs multiplexed together), or from a single filter with a more complex addressing pattern. Which one is more efficient is difficult to decide, so the tool builds both, and uses the accurate resource estimations to decide which will be best.

III. RESULTS

We use the FIR filter design tool to provide some experimental results since it exercises all of the technologies described. The following graphs summarize the hardware generated for a 47-tap, 16-channel, interpolate by 2 FIR filter targeted at the Stratix II family from Altera. The sample rate is fixed at 1.2288 Million Samples Per Second (MSPS) per channel, and the system clock frequency is swept from 24.57 MHz through to 491MHz. This is achieved by changing a single clock frequency specification number in the Simulink design. Figure 7 shows that as the clock frequency is increased, the number of multipliers falls. This is expected since a higher clock frequency allows each physical multiplier to perform more multiplications within the fixed sample time.

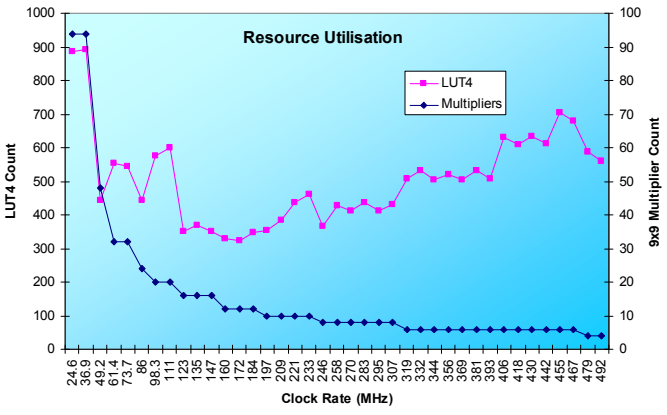

Figure 7. Resource utilisation (LUT4 and 9x9 multipliers) for a FIR filter as a function of system clock frequency

The number of logic cells, measured as 4-input LUTs, falls initially as the decrease in multiplier count results in fewer logic cells to act as adders and address generators. But the logic cell count then increases as more pipelining is required at high clock frequencies. To combine these two components into one metric we approximate the actual silicon area consumed by multiplying the multiply count by 50 and adding the LUT4 count.

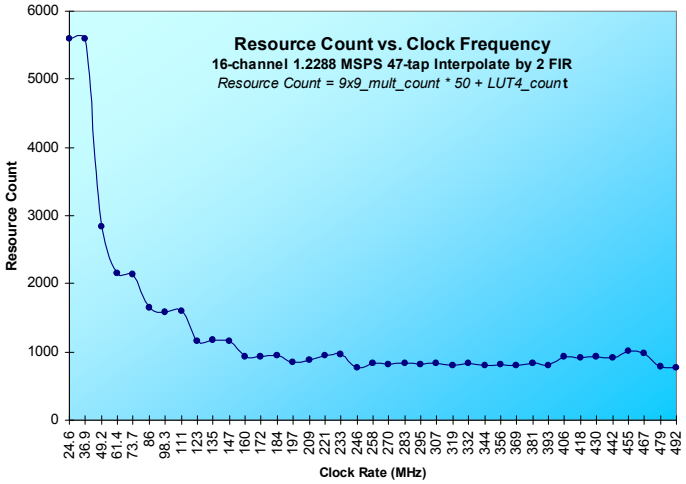

Figure 8. Combined resource metric for FIR filter as a function of system clock frequency

Figure 8 shows that whilst it certainly eases timing closure issues by running at a higher frequency, the overall gain from running at a frequency over 250MHz are limited to changing the ratio of multipliers to logic cells used.

IV. CONCLUSIONS AND FUTURE WORK

Creating designs using an abstract set of components separates out architectural design from detailed implementation. The designer can concentrate on building a portable model using fewer components. The detailed implementation can be created automatically improving defect rates and removing one significant factor in timing closure. One significant contribution of the work is that clock frequencies exceeding those of typical hand-coded designs are reached.

This tool is analogous to a compiler for a microprocessor, since the additional abstract layer of behavioral C removes the need for assembler writing. Here the untimed abstract design input removes the necessity to code in cycle-level RTL. By analogy, since there are cases where programmers may resort to assembler, there are still cases where hardware designers may resort to RTL. One future area of work is to eliminate some of these areas by looking at latency constraints and more specialization for the features of different FPGAs.

ACKNOWLEDGMENT

The majority of the innovations described in this paper were the result of exceptional insight and hugely productive work performed by Andy Hall and Jonah Graham whilst working for Aion Systems Ltd. I feel privileged to have worked with them. I'd also like to thank the many excellent engineers

at Altera who have continued to develop the technology into a widely released product.

REFERENCES

[1] Altera Corporation, Stratix II Device Handbook, 2006.

[2] Altera Corporation, DSP Builder User Guide, 2008

[3] Xilinx Corporation, System Generator User Guide, 2008.

[4] The MathWorks Corporation. Simulink. 7 Reference Guide, October 2008.

[5] A. Ling, D. Singh, and S. Brown, "FPGA technology mapping: a study of optimality," DAC, 2005.

[6] K. N. Lalgudi, M. C. Papaefthymiou, "Delay: an efficient tool for retiming with realistic delay modeling", DAC, 1995.

[7] J. Cortadella, M. Kishinevsky, B. Grundmann, "Synthesis of synchronous elastic architectures", DAC, 2006

[8] Free Software Foundation , GNU Linear Programming Kit, 2008.