Hyper Pipelining of Multicores and SoC Interconnects

Author: Tobias Strauch
tobias@EDAptability.com
October 26th, 2010

1. Introduction

We have seen an enormous rise of multiprocessor usage and its support infrastructure throughout the last years. This trend will most likely continue and is already challenging the community with new hard- and software problems.

The interconnects of multiprocessor SoCs are one potential bottleneck and require additional optimizations to achieve the necessary data throughput. Also for SoCs with cores such as graphic engines, de-/encoders, DMAs and external DRAMs, the interconnects facing tough hurdles as we can see it in the field of video applications for instance.

The instantiation for multiple equal cores such as processors, DSPs and peripherals are also driven by ever increasing challenges of all kind of different applications. We move from 2D to 3D, multiple audio channels, more and more enhanced network switches, multiple channel sensor readout and processing (like we can see it in the CERN project) and last but not least an ever increasing number of instantiation of thousands of equal cores in super-computers.

In this paper, a method is discussed, how the functionality of a core can be multiplied by just adding registers to the core. This results not only in less area usage compared to its individual instantiations, but can also have a serious, beneficial impact on the system performance as a whole. This method is called "hyper pipelining" and is explained in the 2nd chapter. In chapter 3, different approaches and their impact on the system architecture are discussed. Chapter 4 shows the results of a hyper pipelined complex RISC core (OR1200 from OpenCores) in detail.

2. Theory of Hyper Pipelining

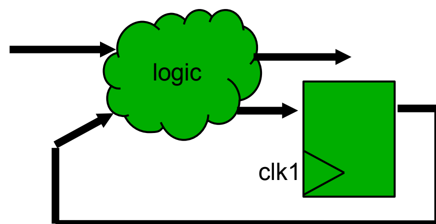

Figure 1. Simplified Sequential Logic

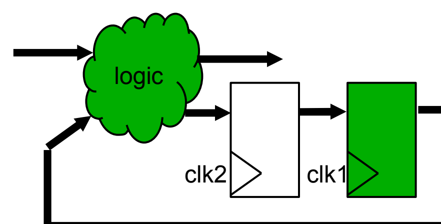

Figure 2. Sequential Logic with Intermediate Register Clocked by clk2

This chapter gives an overview of the theory of hyper pipelining. Figure 1 shows the simplified structure of sequential logic. Inputs and sequential elements clocked by clk1 drive the combinatorial logic. The combinatorial logic drives the outputs and the data inputs of the registers.

In Figure 2 each sequential element is duplicated with an intermediate register clocked by a second clock clk2. If clk2 is synchronous to clk1 but not edge aligned and the timing is right (no setup or hold time violation between clk1 and clk2 registers) the behavior of the sequential logic doesn't change.

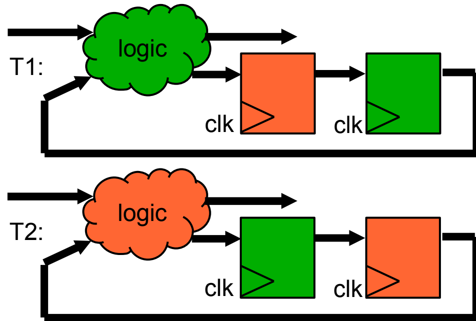

Figure 3. Two Functional Independent Designs

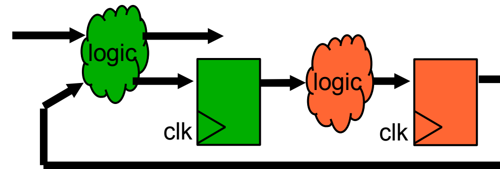

Figure 4. Hyper Pipelined Sequential Logic with Distributed Logic

Assuming clk1 and clk2 of Figure 2 are now identical (clk). This results in 2 functional independent designs in a time sliced fashion. Figure 3 shows how the combinatorial logic is used for one design during T1 and for the second design during T2. The inputs and outputs are valid at the active time slice (T1 or T2). The implemented register set (formally driven by clk2) are called “pipeline stage registers” PSR.

The next step is to distribute the combinatorial logic between the registers without modifying the functionality of the designs. Figure 4 shows one basic rule of hyper pipelining. There are only paths from the PSR to the original register set and from the register to the PSR.

The number of pipes can be increased as shown in Figure 5. The resulting number of independent designs is identical to its multiplication factor, called “core multiplication factor”, CMF.

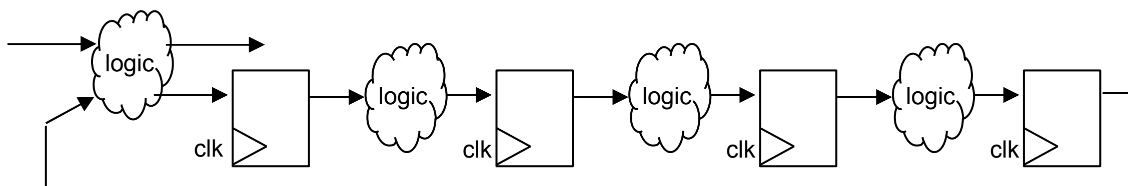

Figure 5. Hyper Pipelined Core with CMF = 4

This hyper pipelining technique is different to the pipelining of instruction decoding known from RISC processors. The point is, that you can use hyper pipelining on top of any sequential logic, for example a RISC processor, independent of its underlying functionality. The RISC processor with pipelined instruction set decoding can automatically be hyper pipelined to generate CMF individual RISC processors. For more information see the documentation of the C252 semester project of the University of Berkeley [1]. Hyper pipelining can also be used on any processor, DSP, peripheral, graphic engine, de-/encoder, de-/encrypter or system bus etc..

Hyper pipelining is also different to register balancing. Register balancing does not modify the overall behavior of the design by moving registers, the hyper pipelining intentionally dramatically "changes" the behavior by multiplying it. Hyper pipelining implements additional registers and can use register balancing for fine grain timing optimizations.

The method hyper pipelining is also called "C-slow Retiming".

The main benefit is the multiplication of the core's functionality by only implementing registers. This is a great advantage for ASICs but obviously very attractive for FPGAs with their already existing registers. It leads to a reduced size and system costs compared to the individual instantiation of the cores. It also changes the overall system architecture and can have a very beneficial impact on the system performance, as described in the next chapter.

Another issue is the performance of the resulting hyper pipelined design. Assuming the formally critical path is now "partitioned" into equal parts, the hyper pipelined design can theoretically run as many times faster as the number of the resulting segments reduced by the additional setup and hold time for the PSR on the critical path. This results in almost the same performance as their individual instantiations, if the critical path is relatively slow compared to the timing arcs of the register. If the critical path is only 4 LUT or 4 gates (which is an extreme example), the timing arcs of the PSR dominate the critical path and a CMF-times performance is hard to reach. For larger cores with a critical path of 16 LUT or 20 gates, the multiplied performance can be reached easier. In some applications the performance of a hyper pipelined core is still by far fast enough to achieve application specific goals. For instance peripherals which use paralleled data from a serial data streams (ethernet).

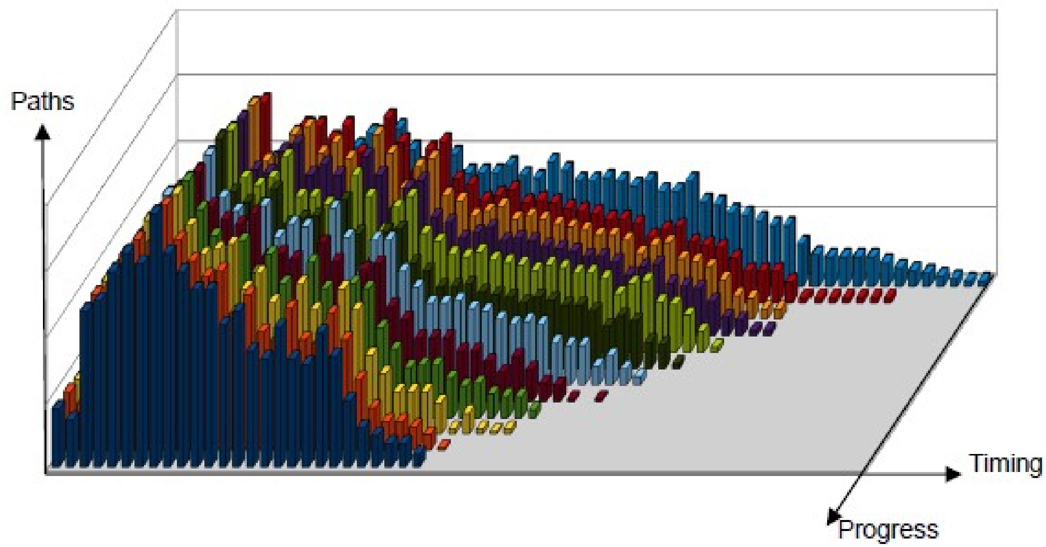

Figure 6. STA Histogram of Timing Optimization

In order to achieve the CMF-times faster clock speed, the PSR must be introduced at the right places in the design. For that a simple algorithm can be used. It starts with placing the PSR (pipeline stage registers) at the outputs of each original register (Figure 2). The PSR are then moved through the combinatorial logic until the critical path is partitioned into equal elements. The passing must follow certain rules, so that the overall functionality of the hyper pipelined core is not broken. Figure 6 shows the individual STA histograms which are taken from the optimization process of the OR1200 core [2]. It starts with the original STA results in the back and shows how the STA results change by passing the critical PSR through the combinatorial logic.

The used tool does the hyper pipelining automatically within seconds, because all estimations (STA) and modifications are done on RTL. If the timing needs further optimizations it reads intermediate "real" ASIC or FPGA STA results to squeeze out the last picoseconds for a particular implementation (fine grain register balancing). The main benefit of doing the modifications (PSR insertion) on RTL results from the fact, that the new system must be verified on RTL. The hyper pipelined core will be used in a system with quite a different architecture compared to a system which uses the multiple instantiations of the core. Although CMF-times individual cores exist as before, the surrounding logic must be adapted to the new core and most likely other parts of the system architecture are affected. The complete system verification must/can be done on RTL.

3. Impact on the System Architecture

In this chapter, the impact of hyper pipelining on the system architecture is discussed. Examples are given for the hyper pipelining of DSPs, processors, system buses and peripherals.

3.1 DSPs

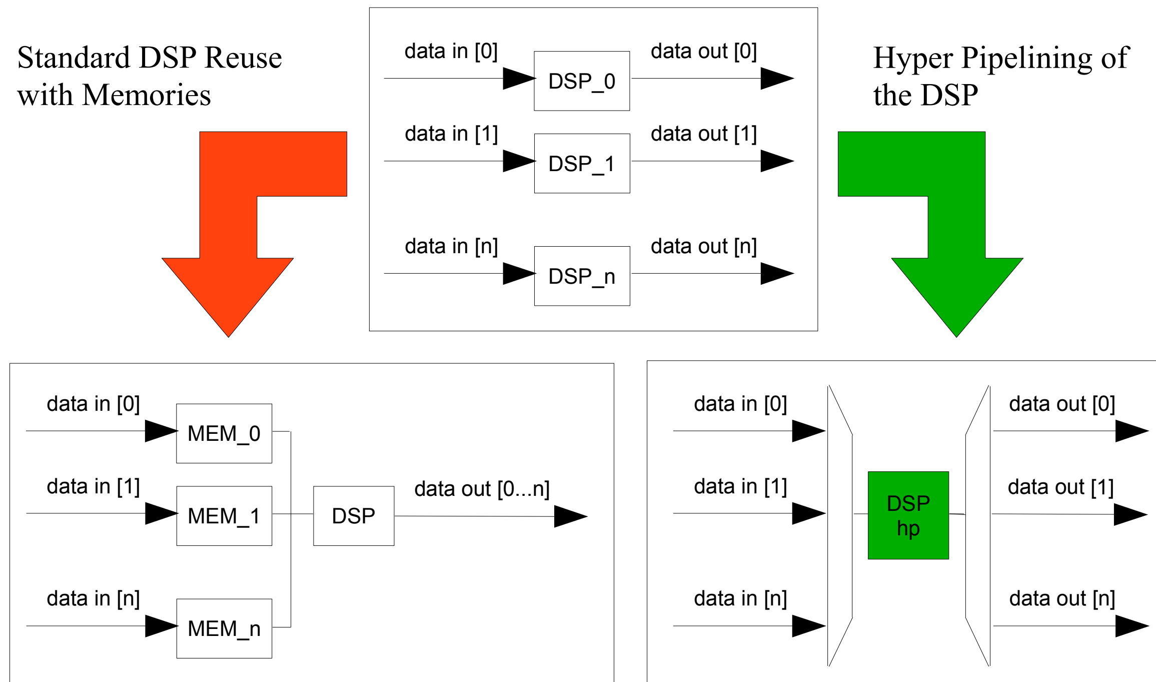

Figure 7: Standard DSP Reuse with Memories and Hyper Pipelining of the DSP

DSPs usually have one or more area critical hot spots, which result in huge area usage for ASICs or a critical usage of dedicated logic (e.g. DSP slices) in FPGAs. This leads to a reuse of the DSP cores in a time sliced fashion and an intermediate storage of the incoming data if multiple data channels must be processed (Figure 7). The additional memory is usually a memory overhead for ASICs, the new scenario results in more design and verification effort (memory access) and the system adds a certain latency of multiple cycles (n-times the memory depth) to the processed data. If the DSP is hyper pipelined, the independent DSP cores handle the incoming data stream in real time in a time sliced fashion, because the timing of the hyper pipelined DSP core is now CMF times faster. The additional memories become obsolete, no additional design and verification effort of the memory read and write mechanism must be done and the overall system has the same latency as using one DSP per channel.

Also if only one DSP is used, the instantiation of a hyper pipelined version can increase the performance. The DSP function is then CMF-times available in a time-sliced fashion. This can possibly lead to a higher sampling rate (CMF-times) or better system efficiency.

Under certain circumstances, the available DSP slices are the limiting factor to select the right FPGA device. By DSP slice reuse (hyper pipelining), an FPGA with a lower number of available DSP slices – and therefore cheaper FPGA - can be selected.

3.2 Processors

Processors such as microcontrollers and RISCs have been enhanced over the last decades, but now the instantiation of multiple processors becomes more and more common for the main stream SoCs, not only for super-computers. Hyper pipelining works on basically all processors (RISC, MIPS, ARM, AVR, CRAYs, …). The next chapter shows one example of a relative complex RISC processor from OpenCores. There are quite a few very interesting aspects, which do have an impact on the system architecture. The processor itself can be optimized for multi-threatening for instance. This is already an ongoing process in the industry. Still the communications, such as mail-boxing and data sharing are mainly done through the system bus.

There is no reason not to functionally enhance the processor for hyper pipelining, before the automatic hyper pipelining task is executed. For example, processor specific special function registers (SFR) can be extended within the processor core for that purpose. With a minor RTL change after the hyper pipelining task, each processor can be enabled to directly read/write particular or all SFR of the other processors and can fire an interrupt at a particular core (SFR-sharing and interrupt-scheme-enhancement). This leads to a faster mail-boxing and data-sharing for multiprocessors.

The hyper pipelining is based on automatic RTL modifications. This allows further manual modifications by the designer after the automatic hyper pipelining task as well. The processor cores can also be enhanced in a way, that they share the same instruction cache (I$) or even more useful the same data cache (D$). This can lead to a performance enhancement of the overall system due to data-sharing in the D$. For that the designer might only need to change a few lines in the RTL where the D$ is instantiated (and possibly the TLB) and adopt the software for the access definition. It is also possible, that not all processors in a hyper pipelined core have the same I$ or D$ sizes. Some even do not need to be implemented at all. Each processor knows its processor index in the hyper pipelined scenario, so that an individual (or even dynamic, on the fly re-) configuration of I$ and D$ is possible.

If one processor has a co-processor, it is accessible to all other processors.

3.3 System Buses

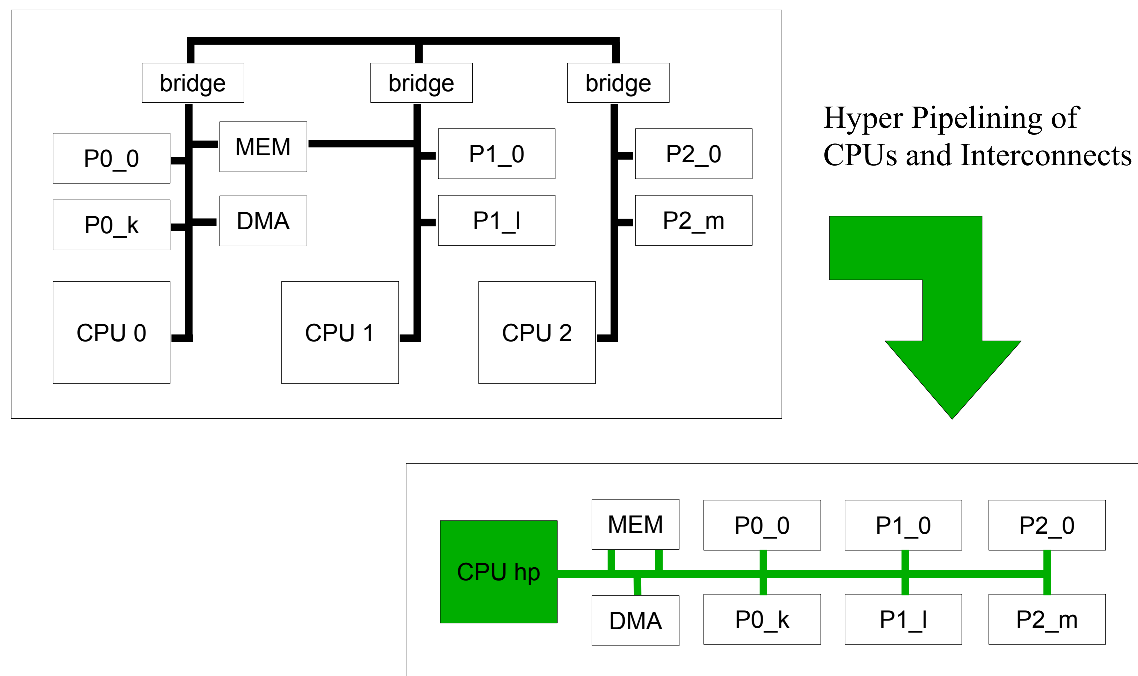

Figure 8: Hyper Pipelining of CPUs and Interconnects

System buses are one major bottleneck of multicore and multiprocessor based SoCs. Especially for interconnect architectures with single cycle read/write requests. The problem is not necessarily the direct data transfer, but instead when multiple data transfers compete with each other at the same time. This leads to the development of multilayer interconnects because an optimization of the timing is not possible. The interconnects tend do be very timing critical in an SoC. If an SoC bus is hyper pipelined, the direct data transfer, single cycle read/write requests, bursts and so on remain the same. The point is, that by multiplying the behavior of the Soc interconnect, multiple data transfers can be done at the same time, for example, multiple bursts can be done at the same time, using different time slices of the hyper pipelined SoC bus (Figure 8). Hyper pipelining can be compared to a multilayer architecture some how, but is works more in a time slice fashion rather than a parallel data routing and can be used on top of a multilayer interconnect scheme, (hyper pipelined multilayer bus) because the automatic hyper pipelining task can be done based on any sequential circuit. The architecture can be supported by an automatically hyper pipelined CPU (as shown in Figure 8) and/or DMA engine.

The hyper pipelined system bus does not necessarily need hyper pipelined bus members. The hyper pipelined CPU hp in Figure 8 is not necessary, the alternative concept with individual CPU instantiation works for a hyper pipelined system bus as

well. Each master can have its own time slice and slaves usually have an asynchronous bus interface or their register set access mechanism is fast enough to handle faster bus interface clocks. On the other hand, a combination of a hyper pipelined processor, a hyper pipelined SoC interconnect and some hyper pipelined peripherals such as a DMA core for instance can be very efficient for the overall system architecture. It can possibly eliminate (some or all of) the original sub system buses and (some or all of) the interconnect bridges in the system, although bridges can be hyper pipelined as well.

The hyper pipelined system bus concept is proven on AXI and OCP-IP buses.

3.4 Peripherals

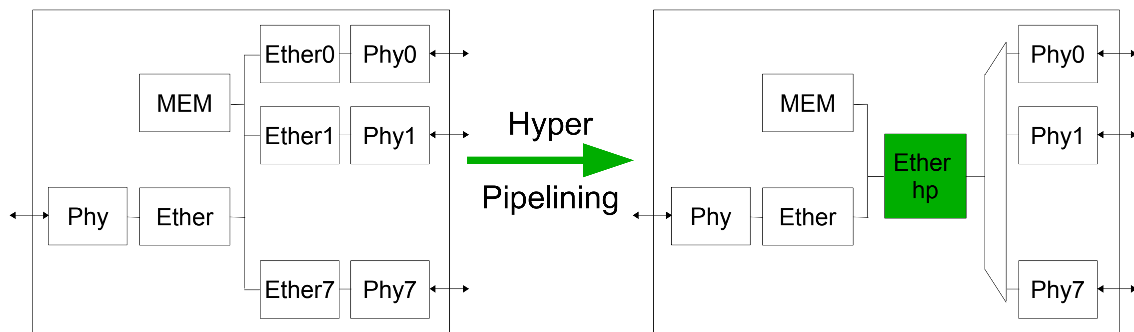

Figure 9: Hyper Pipelining of Ethernet Cores

Peripherals are another example, where hyper pipelining can save a lot of area and therefore costs. Assuming an Ethernet 8x1 switch has nine implemented Ethernet cores, or at least 8 equal ones (Figure 9). They can be grouped together to 1,…, 4 hyper pipelined Ethernet cores, whereas each hyper pipelined Ethernet core (Ether hp) can serve 2,…, 8 independent Ethernet connections .

The Ethernet core is not necessarily timing critical, if the PHY processes parallel data. For 100Mbps usually 4 bits are used, which results in 25MHz and for 1Gbps bytes are used (125MHz). Cores with these frequencies and a more or less simple logic can easily be hyper pipelined. This indicates, that performance is not always the critical factor in hyper pipelining, especially for cores like USB, etc.

4. Hyper Pipelined RISC Example

This section describes the hyper pipelining of the OR1200 core. The original code is taken from the OpenCores' OR1200 project [2]. The OR1200 is a 32-bit scalar RISC with Harvard microarchitecture, 5 stage integer pipeline, virtual memory support (MMU), basic DSP capabilities, TLB, instruction and data cache. For further information, please refer to the OpenCores web site or the web regarding a RISC core in general.

Although it can be misleading, that a RISC is pipelined again, it is selected as a showcase because it is one of the most comprehensive open source cores available. The complete project of the hyper pipelined OR1200 is uploaded on the OpenCores website (project OR1200_hp [3]).

4.1 Verification Trick

After running an RTL modifier tool on the original source code, the resulting hyper pipelined OR1200 core has the same inputs and outputs plus the new clk_i_cml_* clocks, whereas "clk_i" is the original clock name.

A hyper pipelined core is not easy to debug even by its creator, when intermediate signals must be looked at. Fortunately there is a trick to verify the correctness. If all paths from and to each existing clock are constraint, the STA shows if paths between the individual clock domains exists or not. Table 1 shows, that in a hyper pipelined core, there must only exist valid paths from one clock to the "succeeding" clock, or from the last clock index to the original clock. All other paths (e.g. path from one clock domain to itself or "trailing" clocks) are invalid and should not exist.

Table 1: Valid and Invalid Paths for CMF == 4

from\to	orig. clock clk_i	clk_i_cml_1	clk_i_cml_2	clk_i_cml_3
orig. clock clk_i	invalid	valid	invalid	invalid
clk_i_cml_1	invalid	invalid	valid	invalid
clk_i_cml_2	invalid	invalid	invalid	valid
clk_i_cml_3	valid	invalid	invalid	invalid

This is one of the reasons, why all introduced RPS get an individual clock. The hyper pipelined core with all clocks are synthesized and with the right constraining, the STA will reflect potential RTL modification bugs. If no false path is reported, the individual clocks can be merged with the original clock when the hyper pipelined OR1200 core is instantiated.

4.2 Slice Utilization and Performance for FPGAs

The next tables show the area and timing results for a Virtex5 device from Xilinx. In general, ISE 11.1 with the place and route effort option "standard" is used. The Xilinx environment variable XIL_TIMING_ALLOW_IMPOSSIBLE must be set to achieve a better timing. This is done to allow the pushing of registers into the 32x32-bit multiplier (DSP). Memories are regenerated with the multiplied size by Xilinx's Core Generator.

The following results are based on a Virtex5 device (xc5vlx50-1ff676, package FF676, speed grade -1). One implemented OR1200 core reaches 13.853ns (72.2MHz) on this device. The "Achieved Timing" number is the "Data Path Delay" number taken from the first report of the timing report (.twr). The "Theoretical Timing" considers the additional delays introduced by the PSR. The sum of setup time (Tas) and the hold time (Tcko) is 0.400ns. The theoretical achievable timing is:

CMF == 2: (13.853ns + 0.400ns) / 2 = 7.126ns (140MHz = 193% of 72.2MHz)
CMF == 3: (13.853ns + 0.800ns) / 3 = 4.884ns (204MHz = 282% of 72.2MHz)
CMF == 4: (13.853ns + 1.200ns) / 4 = 3.763ns (265MHz = 367% of 72.2MHz)

Table 2 shows the area and timing results of the implemented hyper pipelined OR1200 core.

Table 2. Area and Timing of Virtex5 Device

CMF	FF	Slice LUTs	Occupied Slices	Theoretical Timing	Constraint	Achieved Timing	LUT levels
1 (Orig.)	1.239	3.663 (12%)	1.131 (15%)	n/a	13.5ns (125MHz)	13.853ns (72.2MHz)	12
2	3.048	4.535 (15%)	1.414 (19%)	7.126ns (140MHz)	7.120ns (140MHz)	7.327ns (136MHz)	7
3	4.153	5.602 (19%)	1.594 (22%)	4.884ns (204MHz)	4.880ns (204MHz)	5.398ns (185MHz)	R-Out + 3
4	4.777	6.286 (21%)	1.773 (24%)	3.763ns 265MHz	3.763ns (265MHz)	4.902ns (203MHz)	R-Out + 1

R-Out in Table 2 indicates a RAM output. The number of used DSP48Es for the 32x32-bit multiplier remains 4.

Table 3. Relative Area and Performance of Virtex5 Device

CMF	Slice LUTs	Occupied Slices	Relative Performance	Theoretical vs Achieved Timing	Performance per Slice [kHz]
1 (Orig.)	1	1	1	n/a	63.8
2	1.23	1.25	1.88	0.97	96.1
3	1.52	1.40	2.56	0.90	116
4	1.71	1.56	2.82	0.76	114

Table 3 can be read as follows. With CMF = 2, the number of slice LUT rises by 23% and the number of occupied slices by 25%. The performance increases by 88%, which is 97% of the theoretical achievable timing. The performance per slice is 96.1kHz.

The average delay of a Virtex5 LUT and a net is assumed to be around 1.3ns (after having looked at tons of timing reports). The difference between theoretical timing and achieved timing is within this granularity of 1.3ns for CMF = 2, 3. The increased area of up to 56% has also an impact on the achieved timing.

The device is relative underutilized (15%). Other examples with higher utilization show, that the hyper pipelined core can be better packed (less increase of occupied slices for the hyper pipelined core).

4.3 Area Ratio for ASICs

If the hyper pipelined OR1200 core is implemented on an ASIC, the size of the combinatorial logic (gates) remains almost the same, only the number of registers increases. This number should not be simply multiplied, because the registers of the new implemented PSR are located at internal signals. A m+n-adder adds m+n registers, if the registers are placed at the inputs, but only max(m, n)+1, if they are placed at the outputs of the adder logic. Table 4 shows the number of registers implemented on the OR1200 core without the FPGA specific timing optimizations.

Table 4. Area Ratio for ASICs

CMF	Registers	Relative Registers	Area Ratio with 42/58 Ratio
1 (Orig.)	1239	1	1
2	2995	2.42	1.59
3	3769	3.04	1.85
4	4244	3.43	2.01

The number of registers increase by 142%, 204% and 243%. If the ratio of register area and combinatorial logic is set to 42/58 (42% register area and 58% combinatorial logic), which is based on a synthesis report using the lsi_10k library, the area increases by 59%, 85% or 101% of the original area. For ASICs, the performance is much closer to the theoretical timing, because the place and route as well as the timing optimization algorithms can achieve relatively better results in general compared to FPGAs, due to the higher routing capabilities of ASICs.

The hyper pipelined core includes a huge number of shift registers. If an area optimized shift register cell is use, the overall area can be reduced even further. The logic cones of the hyper pipelined core are also fundamentally smaller, which leads to a reduced size of test pattern and test time.

4.4 Simulation

In this section the simulation of a hyper pipelined OR1200 is shown. For the simulation, a random instruction code generator is used, which generates standard register file modification instructions, set flag instructions and conditional forward branches (by a few lines). Once a certain address border is reached, the code generator adds a jump to the content of R0. The advantage is, that if the read address of the instruction fetch cycle is displayed in an analog format, the waveform looks like a chainsaw profile. Since in this case, little forward branches based on calculation with random variables are added, it is more or less the profile of a used chainsaw.

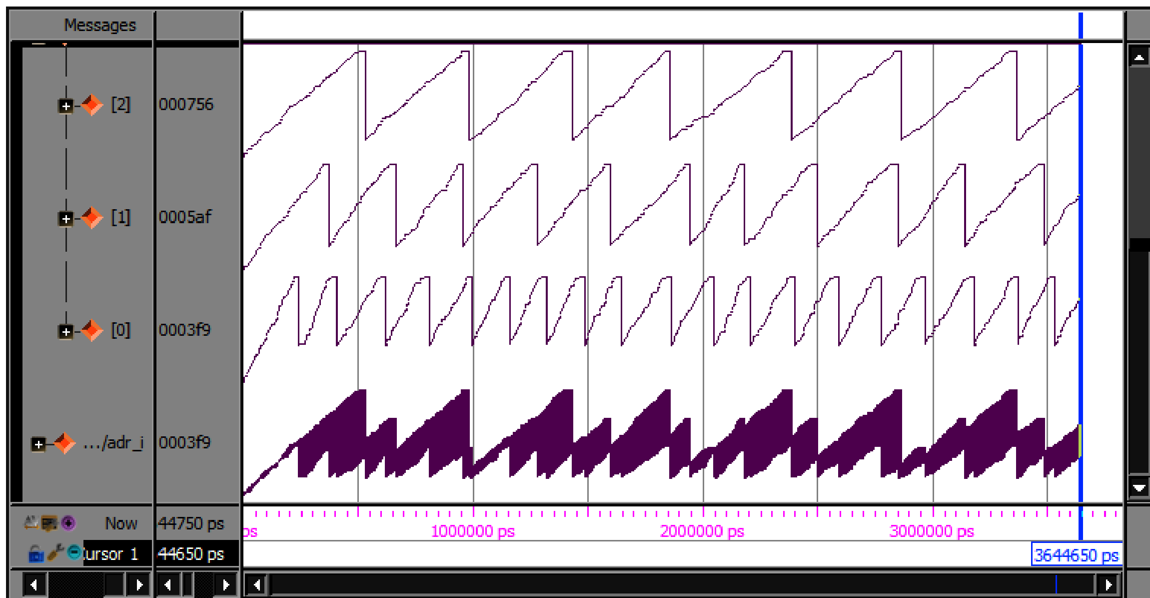

Figure 10. Hyper Pipelined OR1200 Simulation with CMF = 3

The first three lines of Figure 10 show the instruction read address of the three independent OR1200 cores. All three programs run at the same speed, the OR1200 with a higher index in the first line simply has a higher address border to jump back to the content of R0, so it appears in the waveform window to be slower. All signals are saved at the relevant time slice for debugging purpose only, the real instruction read address bus behavior is shown in the fourth line. It can be seen, how the bus signals switch between the three independent OR1200.

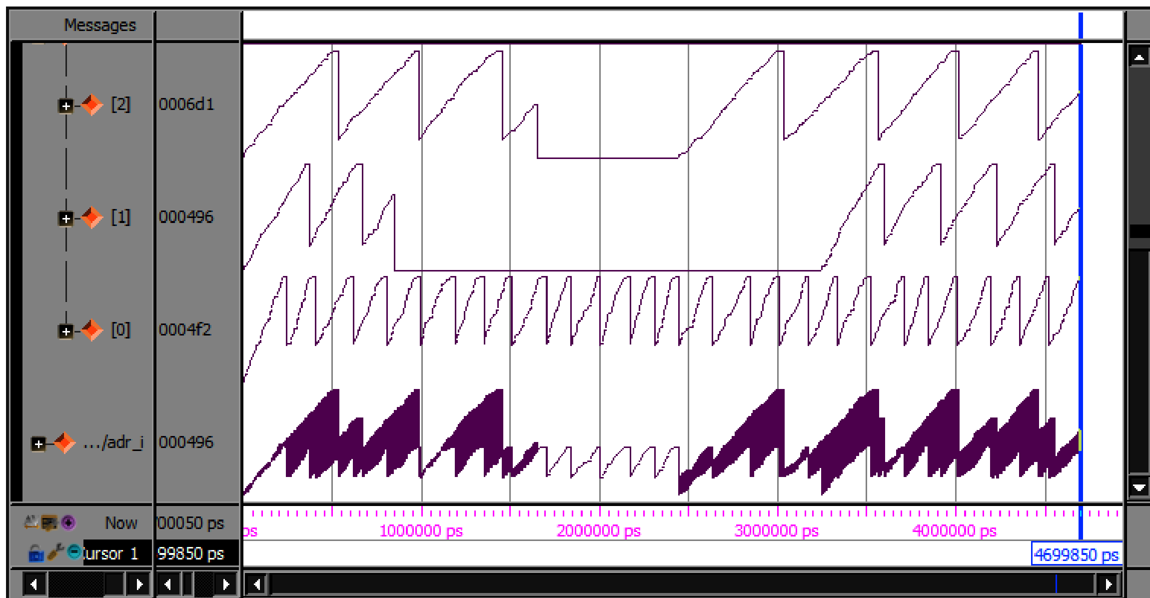

Figure 11. Switching Off and On Cores and Restart

Figure 11 shows, that individual cores can be switched off by disabling the relevant clock at the relevant time slot. This reduces the activity of the design, because the switched off core behaves then exactly as its predecessor (and its outputs should be treated as invalid at the relevant timeslot). By applying a reset impulse at the right timeslot, the core can be individually restarted and continues with the reset sequence (in case of a processor). There is no particular rule in which order the cores can be switched on and off, besides the fact, that the core needs to be switched on, before it can be switched off ;-). The other cores are completely unaffected by this and continue to run at the given speed.

It is important to notice, that the individual core do not store its values when switched off. It always restarts with the reset values, respectively the reset sequence if the core is a processor. If a core is switched off, the clocks can also be gated to reduce activity on the clock line. This is the another reason, why the PSR get their individual clock trees.

Another interesting aspect is, that if no individual reset cycle is added, but the clock is switched on again, a copied version of the preceding processor continues and will only start behave differently, if the inputs change compared to its predecessor at the relevant timeslot.

Tools used:

FPGA Compiler: ISE 11.1, Xilinx, CA, USA
http://www.xilinx.com/tools/webpack.htm

Simulator: Modelsim, Mentor Graphics, CA, USA
http://www.xilinx.com/tools/mxe.htm

Core Multiplier: SynEDA CoreMultiplier, EDAptability, Munich, Germany
http://www.edaptability.com/coremultiplier.htm

References:

[1] Y. Markovskiy, Y. Patel, “C-slow Retiming of a Microprocessor Core”, UC Berkeley, CA, CS252, Semester Project,
http://brass.cs.berkeley.edu/documents/**cslow**fpga.ppt

[2] OpenRISC, OR1200, OpenCores,
www.opencores.org/project,or1k

[3] Hyper Pipelied OpenRISC, OR1200_hp, OpenCores,
www.opencores.org/project,or1200_hp

Author:

Tobias Strauch studied Micro-System-Technology at the University of applied science in Furtwangen, Germany. He did his diploma thesis in '98 on “Wave Pipelining of FSMs”. He is now working for EDAptability in Munich, Germany, as a hard- and software developer.

Please come back if you have comments or questions. Please contact:

tobias@edaptability.com

The author is looking for a constructive and fruitful discussion, wherever it is possible.