DYNAMIC - A Java Based Toolset For Integrating Dynamic Logic Circuits Into A Standard VLSI Design FLow

Andreas Wassatsch, Dirk Timmermann

University of Rostock
Department of Electrical Engineering and Information Technology
Institute of Applied Microelectronics and Computer Science
Richard-Wagner-Str. 31, D-18119 Rostock, Germany
Tel./Fax.: ++49+381 498 3534/3601
wassatsch@e-technik.uni-rostock.de

INTERNATIONAL CADENCE USER GROUP CONFERENCE
September 10-13, 2000
San Jose, California

Abstract

Dynamic circuit implementation style is a common technique for high performance applications like microprocessor or DSP design. Due to the different characteristics of the dynamic cells compared with static CMOS circuits we need adapted synthesis methods. Unfortunately, there is no standard CAD support for dynamic CMOS logic until now so the design of these high speed circuits has to be donemainly manually. In this paper we present a solution and the necessary toolset to close this design gap by integrating dynamic circuit technology into the standard CMOS design flow. We will show that the chosen Java based realisation of the necessary toolset can fulfill the requirements of an industrial circuit development process with the advantage of a platform independent implementation. Due to the different requirements of dynamic circuit cells we will give some indications to achieve good results in terms of speed and area in the resulting dynamic circuit design. First results from evaluating the proposed design flow on real design projects are also available and will demonstrate the feasibility of our solution. Furthermore, we will give some indications to achieve good results in terms of speed and area.

1. Introduction

Implementing a design using dynamic circuits is state-of-the-art for high speed data processing applications. For the realization of circuits like the Compaq Alpha microprocessor [1] or the experimental IBM PowerPC integer microprocessor [2] this circuit style technique was used intensively on the way to reach operating frequencies near $1GHz$. However, the productivity of the implementation process is limited by the support given by CAD tools for this circuit technology. Currently, there is no relevant assistance from leading commercial CAD tools for this logic style. Furthermore, due to the limited information available from design teams which have successfully applied dynamic circuit techniques in their projects [3][2] we can only make some assumptions about the methodologies utilized up to now. Usually, publications on high speed design with dynamic circuit techniques cover only details of the cell implementation. The results presented are achieved by using manual design and partitioning of sub-blocks [4][5][6][7]. In this paper we will present a new design methodology for handling dynamic cells for a standard synthesis flow. The objective of this methodology is to shorten design time by utilization of newly developed circuit processing tools in conjunction with commonly used logic synthesis. The organization of this paper is as follows. The principles of dynamic circuits as well as two examples for this technique are described in section 3. Section 4. describes the algorithm of the most important part of the suggested design flow, the Micro Pipeline Reorganizer (MPR), and some special aspects related to it. Representative design examples will be given in section 5. for demonstrating the broad applicability of the presented design methodology. An alternative application field is shown in section 6. Section 7. is dedicated to the conclusion.

2. Why Java ?

The enterprices which develop Electronic Design Automation (EDA) toolsets share a common problem: “How can be minimized the total cost of development of one application for the range of different platforms requested by the consumer ?”. Commonly, a tool will be designed only for one of the most requested enviroments and later, on demand, ported to additional systems. This leads, under the pressure for inovations, to a range of different versions for one tool.

On the other side, a typical consumer company has a heterogeneous computing enviroment so that also different tool versions must be maintained and utilized. The variety of versions and the resulting problems during the interoperation of two versions of a tool highly increase the necessary maintainance efforts in a design company.

2.1. EDA Enviroment

Now, how is a typical design environment structured ? Mostly it is an unix driven system based on a risc microprocessor like ultrasparc, power3 or PA-8200/7300 which represents the so called categorie of workstations. The workstations class is characterized by its very good configuration and high reliability but also by a high value of system cost.

A minority of design enviroments at this time are x86 based systems, the so called desktop class, where the chosen operating system belongs to either of two groups, the windows driven and the unix driven devices. The windows based systems are mostly utilized by companies like Intel. Promoted by the cost factor, the count of unix x86 EDA systems is rapidly growing like in the server and desktop world. The utilized unix system is most often based on linux to get faster access to features like clustering or multiprocessing abilities. An unix system on a x86 combines the favored characteristics of the workstation and desktop categories to an ideal EDA environment of the future. This fact is also supported by the constantly growing amount of design application ported to this platform. This wide range of design enviroments utilized is ,as mentioned above, a challenge for the EDA tool industrie.

2.2. Possible solutions

To satisfy this requests for more or less portable toolset the developer can choose different solutions.

At first he can port his preferred system dependent programming interface from one platform to others and recompile for each of them the common programming code of the toolset. Unfortunately, this solution depends on the quality of portation of the platform depending interface. Most often the ported interface shows additional bugs compared to the implementation of his parent. The effort for the portation is merely shifted from the actual toolset to a programming library component so that we also need to maintain the different platform dependen libraries. One example of such a solution is Synopsyst' mfc-emulation in their simulation userinterface.

Another way is to define or utilize a platform independend programming environment. The portation effort for the runtime libraries is done by the software environment developer. Due to the broad range of possible applications of the independend programming environment the assigned efforts in optimizing such a product are by a much higher order of magnitude. Furthermore, due to special benchmarks the platform independend execution of the software will be intensively tested by the environment developer. Also, the environment version is a minor problem because normally a new, stable version will be offered for the full platform range. With this way the EDA toolkit developer has the chance to write once and run everywhere. Beside tcl
tk also the programming language java is a possible solution for such a portable tool development. The Xilinx coregenerator, written in java, is an successful example for this . Portability has also unwanted side effects, in fact the performance of the non optimized program code isn't very satisfying due to the mostly interpreting program processing. But this problem can be partially solved through the utilization of technologies like just-in-time (jit) compiling and optimization within the runtime system.

The decision for the utilization of a portable runtime environment is also influenced by the demands on the performance of the application. For instance, a mostly interactive application like a waveform viewer, buildingblock generator or the user interface for a synthesis application can easily be transformed into a portable implementation without any performance penalities. In the case of preformance sensitive applications like logic synthesis or automatical placement and routing the decision is quite difficult. For small and medium sized design problems the platform independend implementation already at present time can fullfill the runtime requierements supported by features like jit. Only at high end, very large computation problems the additional delay in runtime through the portable description ammounts to an unacceptable value. This kind of problems can only be solved on very platform depending optimized implementations, but mostly even they need much more computation time as we would like.

2.3. Performanceanalysis

How is, at present time, the relation between the execution times of different kinds of implementation. As shown with our exemplary results at Tab. 1, not only the utilized implementation technology determines the attainable performance. Also, the quality of the sources is an important factor. In the example the C-prototype was designed using the open-source gnu-toolset with only few optimizations for the most utilized internal functions like sort and searching. Unfortunately, due to incompatiblilities in the utilized function libraries the tool was not executable stable on other platforms.

In respect to the demands of the java environment we utilize some more optimized routines for the highly frequented internal functions. The results from booth plain java runsets shows the equality of the java runtime environment (jre 1.2.2/nojit), also the scaling with the cpu speed is recognizable. In the case oft the jit runsets the difference between the quality of the professional implementation (jre 1.2.2/jit) and the opensource product (kaffe3) is still visible at present time.

As the presented results show, a careful implementation of the algorithm together with a quality runtime environment can be suitable for a portable toolset.

3. Dynamic Circuit Technologies

The main difference between standard static CMOS and dynamic circuit techniques from the designers point of view is the requirement of dynamic implementations to clock not only the (pipeline) registers but also logic elements in order to work properly. The dynamic character of this design style is established in two operation phases. During the precharge phase a capacitance, mainly consisting of parasitic elements like wiring and input capacitancies of the following logic, must be charged to a specific voltage value. In the following evaluation phase a logic network made up of transistors determines whether the capacitance is discharged or not. This leads to a continuous power consumption in the case of discharging during the evaluation phase independent of a change of the logical output value. The logic network is normally implemented using nMOS transistors only due to their faster switching behavior. Thus, cell area is reduced by nearly a half in dynamic circuits compared with static circuits. Furthermore, this technique often requires only one

Table 1. Synthesis time and results for MPR

Design name	Cells /Pipes/ CellsP	Sparc 400MHz/2G Solaris C/Java/JIT	Athlon 500MHz/256M Linux Java/JIT
add_rpl4	11/8/74	2/2/18	1/1
add_cla4	55/7/99	3/3/18	1/1
add_clf4	58/7/103	3/4/18	1/2
add_bk4	26/6/55	1/2/18	1/1
add_rpl8	24/17/331	58/10/18	5/4
add_cla8	88/9/193	14/6/19	3/3
add_clf8	136/9/193	27/11/25	6/5
add_bk8	78/8/170	10/5/19	2/3
add_rpl12	34/23/664	237/25/18	14/10
add_cla12	135/10/328	44/11/24	5/4
add_clf12	230/10/481	79/20/20	11/9
add_bk12	107/10/295	35/9/19	5/4
add_rpl16	44/29/1115	673/52/19	29/17
add_cla16	216/10/479	90/17/20	9/8
add_clf16	353/11/799	231/35/21	21/15
add_bk16	140/11/444	84/15/19	8/6
add_rpl24	64/41/2380	3120/159/19	95/56
add_cla24	329/12/853	322/34/21	20/14
add_clf24	474/12/1304	680/67/24	41/26
add_bk24	218/12/709	218/27/18	16/11
add_rpl32	83/ /4029	/439/20	262/136
add_cla32	440/16/1591	1248/85/24	50/32
add_clf32	590/13/1429	750/75/31	42/30
add_bk32	336/13/1080	508/48/21	26/19
mult_cs4	77/14/272	35/8/3	5/4
mult_cs8	288/23/1173	713/57/11	37/25
mult_cs12	628/32/2684	/191/26	137/91
mult_w4	146/21/465	93/16/5	10/7
mult_w8	574/23/1390	758/68/13	45/31
mult_w12	1141/30/2905	3494/196/28	140/96
des_slice	2046/27/6680	/896/132	/435
des_pipe	32920/241/	/ /22h13m	/
key_56	1680/9/4802	/551/89	416/364

clock signal which reduces the effort in distributing the clock across the whole die. Unfortunately, the clock signal drives a higher fan-in and the clocking network has to be designed very carefully. For these reasons the typical application area of dynamic circuits is more often high-performance than low-power. As examples for dynamic circuit technology two representative techniques are described in the following.

3.1. True Single Phase Clock Logic

True Single Phase Clock (TSPC) technique first presented in [8] can be implemented as shown in the circuit of Fig. 1. The logic function is implemented in the n-logic block (the same is possible for the p-block with decreased speed) by a custom network of nMOS transistors. These are limited with respect to the number of transistors in series by technology and speed requirements. There are also some minor limitations with respect to the number of parallel transistors. Due to the even number of inverter stages only noninverting functions like and and or are possible in one dynamic block. In this paper we do not consider mixed implementations with standard CMOS logic or a variable number of stage registers per cell [4] as speed is our primary design goal. This results in the requirement for a modified netlist which has signal inverters at the input or/and the output of the whole circuit only. The modification of the netlist, mainly the separate inspection of the netlist for each output signal, usually results in a higher gate count. This is partly compensated by a following search for collapsing terms in the resulting netlist trees.

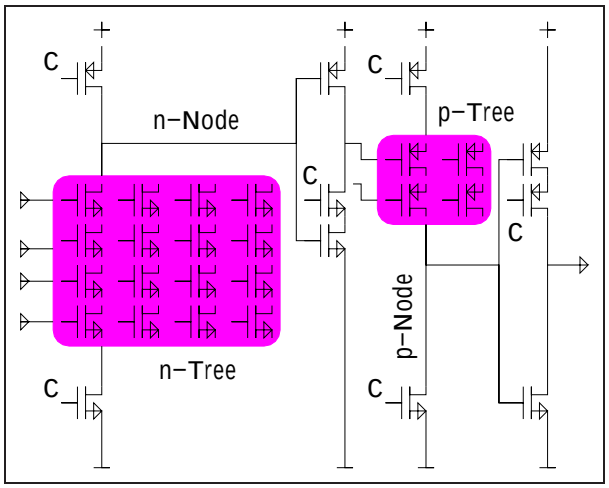

Figure 1. TSPC circuit

3.2. DTSPC

One possible solution to prevent this undesirable increase of the netlist is the use of Differential True Single Phase Clock logic. It is similar to Differential NORA logic (DNORA) [9] but like TSPC it lacks the feedback circuit found in NORA. A DTSPC-cell consists of two TSPC cells with complementary logic networks to generate differential output signals which eliminates the above mentioned preparation of the netlist. An AND2/NAND2 example is shown in Fig. 2. On the other hand the number of interconnections and, consequently, cell area increases as well for this kind of circuit style.

The application of other dynamic circuit techniques like domino logic or differential cascade voltage switch (DCVS) logic [10] is also possible in our approach due to its circuit style independence and modular structure.

4. Design methodology

For our design methodology we do not utilize the common way for development of circuits with dynamic logic, i.e. the manual preparation of building blocks and a final arrangement at the top level. Instead we will use standard logic synthesis and a modified mapping method. This approach enables dynamic circuit style techniques to be integrated in a standard design flow as shown in Fig. 3.

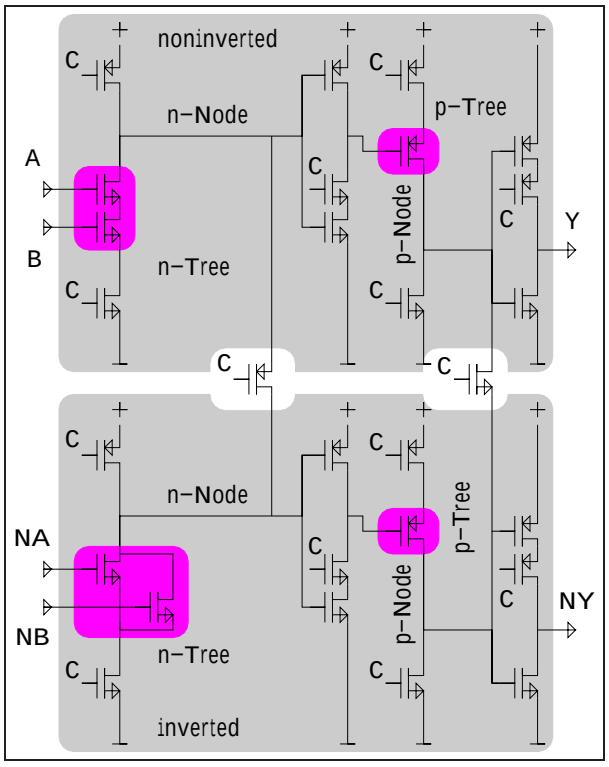

Figure 2. DTSPC circuit

4.1. The DYNAMIC toolset

According to the developed design flow we have now implemented the necessary components for the DYNAMIC toolset like shown in Fig. 4. Due to the modular design each component can be used standalone via a command line interface or in co-operation with the other tools through the gui-interface.

4.2. The Micro Pipeline Reorganizer

The main component in the described design methodology is the Micro Pipeline Reorganizer (MPR), an algorithm to restructure a netlist accord-

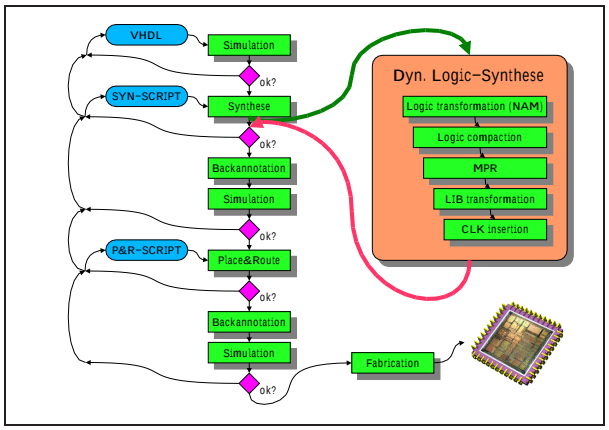

Figure 3. Design flow for dynamic circuit technique

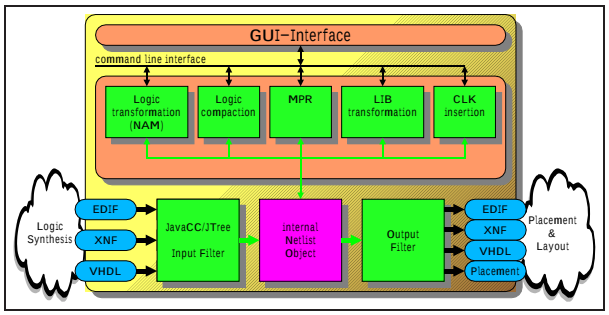

Figure 4. The DYNAMIC components

ing to the requirements of dynamic circuits. Basically, the algorithm is based on the assumption that the processed netlist represents a pipeline structure which is suitable for complete dynamic implementation. Because of the limited logic depth of one cell in each stage we choose the term micro pipeline to reflect this fine granular structure. The task of the MPR is to optimize this pipeline structure with respect to throughput. During restructuring the latency of the pipeline will be unchanged which is essential for predicting the timing behavior of the resulting circuit. The latency of the pipeline is well defined by the maximum number of cascading cells in all parallel signal paths. Starting with the output signals the algorithm iteratively processes the netlist to find and eliminate circuit bypass structures as shown in Fig. 5.

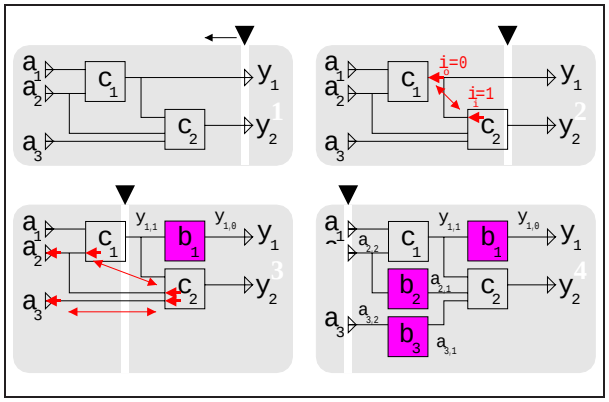

Figure 5. Substituting circuit bypasses by buffers

The elimination is achieved by inserting additional cells with a functionality equal to simple registers. Corresponding to the pipelining algorithm presented in [11] the index value will be initialized at first for all nets in the actual signal layer. Afterwards, for all cells which drive one of this indicated nets the index of the net connected to the input pins will be checked. Based on the value of the input pin net the tool can decide who has to handle this connection. An uninitialized index is a new net which does not need special attention. A net index with the same

value like the corresponding net on the output pin of this cell represents a net bypass on this layer. This bypass must be fixed by the insertion of a buffer cell named b in Fig. 5. To reduce the amount of inserted buffers only one for each net will be inserted. In case of an index value lower than the value of the cell output network a circuit loop has been detected and should be reported to the designer for manual fixing. Input net index values higher then the output pin index plus one are not possible by design of the algorithm. After the completion of the calculation on this network layer the algorithm will be continued on the next layer level until the input connectors of the circuit will be reached. During the processing of the netlist a continuous netlist integrity check will be done to detect unexpected circuit constructions like the exception described in subsection 4.6.

4.3. Netlist adjustment for preparation of noninverting gates

As already mentioned, our design flow requires a transformed netlist without any logical negations in order to use TSPC library cells. This task is executed by the Netlist Adjustment Module (NAM). A recursive algorithm eliminates inverting elements by application of the De Morgan theorem. Due to the definition of the theorem composite equations must be separated into basic operations before the algorithm can be started. The algorithm starts at the end point of a signal path which is equal to an output signal in the given netlist. In case of the detection of a negated output signal the driving cell will be changed to the corresponding cell according to De Morgan's theorem. If a signal drives both an inverted and a noninverted input the remaining path to the input ports of the netlist must be processed separately for both cases. Each output signal path must be observed separately which leads to a higher amount of terms for the complete netlist. This can be slightly compensated by a search for matching terms over the whole netlist after finishing of the NAM algorithm. Finally, due to the capability of dynamic circuits to integrate more logic into one cell clusters of terms can be concentrated in one cell. This is similar to the approach employed in complex static gates.

4.4. Clk tree generation

The transformation of a static CMOS netlist to one which utilizes dynamic circuit style implies the generation of a clock network for the dynamic cells. Commercial tools for the back end in the design flow, i.e. for placement and routing, usually provide integrated functionality for handling clock networks. Because of the precise knowledge on the demands of clocking for the sub-circuits in this early stage of the design flow we decided to implement clock networks right now. Depending on the design requirements we are able to implement different clocking strategies in this module like a clock tree or an antiparallel clocking distribution as shown in Fig. 6.

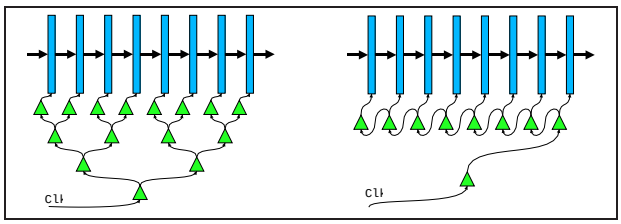

Figure 6. Available clocking strategies

4.5. Implementation

One of the goals during the development of this design flow was the possibility to integrate tools from different vendors for tasks like pure logic synthesis, layout generation, and verification. We used a publicly available C/C++ compiler on our primary target platform SUN SPARC for first prototype implementation. The implementation was verified also on other Unix platforms. During testing we faced some trouble with platform dependent incompatibility of libraries. Subsequently, we restricted the verification to a small number of supported systems. For that reasons we have decided to switch to a platform independent environment for further developments. Current work is on re-implementation of the tool-set in JAVA to provide a fully portable solution for Sun Solaris as well as Linux or Windows NT considering product announcements from different CAD companies for these systems. Above all, the main problem here is to find a solution for the insufficient tool performance on realistic designs. We are currently investigating the acceleration of our basic search algorithms.

The integration of the tools in the design flow is done by providing different netlist formats like structured VHDL or XNF for circuit database interchange. The compiler-compiler technology employed for the development of the netlist reader enables fast implementation of other netlist formats as well.

After transformation of the netlist into a dynamic design we can further validate the functionality of the circuit by simulation with VHDL cell models before we start to generate the final layout.

4.6. Recommended design style

The presented design flow works well on common datapath circuits with one exception. Implementation of sequential elements like a RS-flipflop with few single gates is not possible due to the working principle of dynamic circuit. Furthermore, it is mandatory to reduce the number of gates (logical depth) in the longest path to keep latency low. The longest path determines the overall latency. While throughput is increased by enforced pipeling the latency remains

nearly the same after the transformation to dynamic logic. It is highly recommended to take into account the target cell technology during system modeling. For example, one has to choose an adequate architecture for the implementation of arithmetic operations which allows for all required specifications. Using a ripple carry adder for word lengths higher than four is possible only for low speed requirements. Also other fast adder concepts like carry-select or conditional-sum should be used only if latency is uncritical as the maximum number of cascading gates depends on the wordlength. Redundant adder implementations should be favoured for low latency applications. Carry-save as well as signed-digit adders have a logic depth independent of word length thus achieving very low latencies.

In general, the application field of this design flow is the implementation of pure datapath structures in a pipeline. For realization of non-unfoldable iterative algorithms a reduction of the depth of the loop is necessary to achieve a high throughput. A simple counter can be used to demonstrate this. The upper bound of possible counting frequency is determined by equation 1.

$$f_{CntClk} = \frac{f_{Clk}}{n_{LoopGates}} \tag{1}$$

$$with \quad n \in \{1, 2, 3, \ldots\} \tag{2}$$

A logical depth of 2 inside the loop will half the counting frequency. For only one logic level in the loop the counting frequency will be equal to the clock frequency.

To simplify tool development we have further imposed the rule that each cell can drive only one signal. For example you have to prevent the utilization of a full-adder cell with two output signals carry and sum. However, this is not a severe restriction. In this example we can use a separated implementation for both signals either constructed manually or by the synthesis tool. The validity of this rule is checked during processing and an exception is generated if necessary. For the logic synthesis itself we use either a generic cell library or a specialized library which reflects the implemented cell function without the sequential behavior. The former way is faster but generates also slightly slower results compared with a logic compaction in the MPR.

5. Examples

Due to the limited space of this paper only the results of different implementations of two example operations will be presented to show the successful application of this approach.

5.1. Adder

As mentioned above dynamic logic circuits can not always be sucessfully employed in all architectures to realize a given functionality. To exploit the full potential of dynamic logic the maximum depth of the logic path should be two for each operation. However, not all commonly used variants of circuit representation meet this requirement. Redundant operator implementations like carry-save or signed-digit fulfill this requirement. Due to the greater logical depth possibly with an n-tree of dynamic logic cells we can realize the basic building block for this circuit style in one cell.

As an example for a disadvantageous architecture a ripple carry adder (RCA) with a word length of 8 is shown in Fig. 7 . In almost each pipeline stage only one useful gate can be found. The rest of the cells serves only for skewing interim signals. Carry propagate adder structures should be used only if latency is uncritical for the whole implemented algorithm .

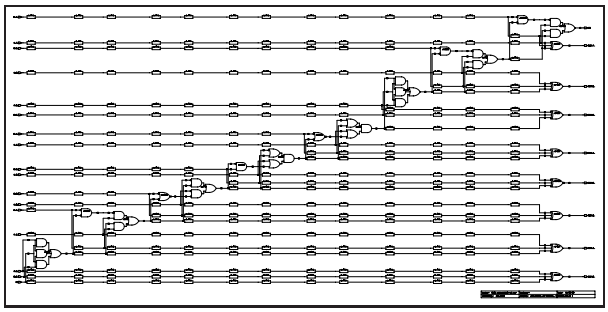

Figure 7. RCA architecture - high latency for dynamic adder

A detailed analysis of latency for different adder architectures over the word length is shown in Fig. 8. Each carry propagate adder exhibits a latency depending on word length. Both redundant adders have a fixed latency of two clocks.

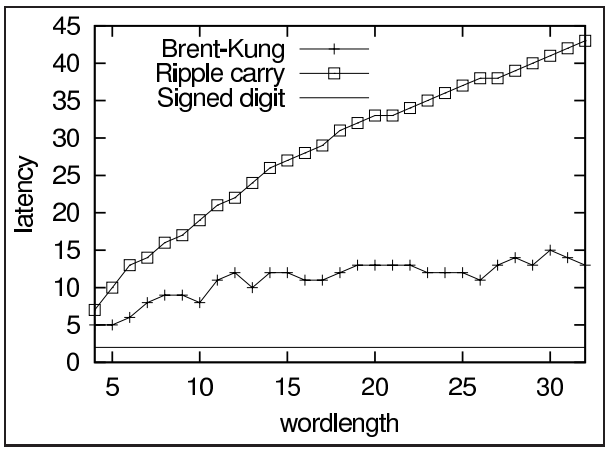

Figure 8. Latency vs. word length for dynamic adders

Like shown in Fig. 9 for an example technology the clocking frequency in a full pipelined RCA are constant over the wordlength where for the common non pipelined version the possible frequency extremely decrease.

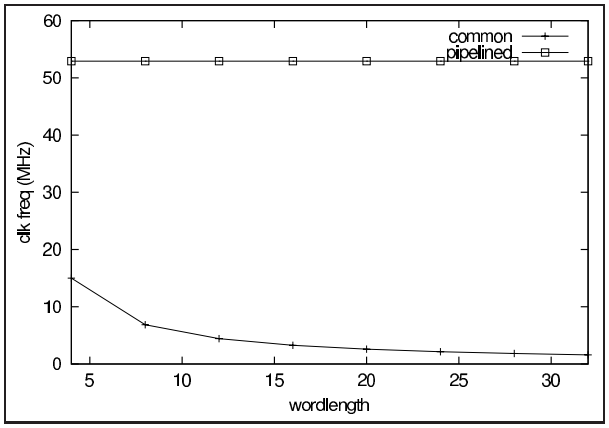

Figure 9. Comparison of clocking frequency

As for standard logic synthesis results depend heavily on the synthesis library. To achieve a latency of two clock cycles for the redundant implementations we must provide optimized logic cells comparable to the complex logic gates approach in conventional CMOS technology. In Fig. 10, for example, the circuit for the d-Digit generation of a signed-digit (SD) adder is shown. Due to the large amount of logic we must divide the implementation into parts which leads to the two clock-cycles latency.

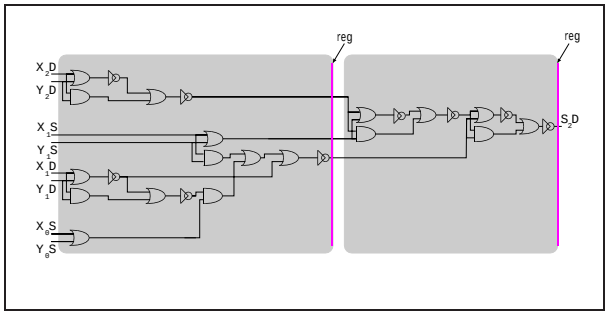

Figure 10. data-digit generation on a SD-adder

As only the n-tree has to be realized the implementation effort for this dynamic complex logic cell is smaller compared with standard CMOS.

5.2. Multiplier

A more complex but also generic example is the multiplier module. Built up from adder components the previously mentioned topics can also be applied to this example. Furthermore, it extends the degree of design parameters by the topic of the main architecture of the module. Due to the pipeline architecture the influence of the carry propagate adder structures on latency is slightly reduced.

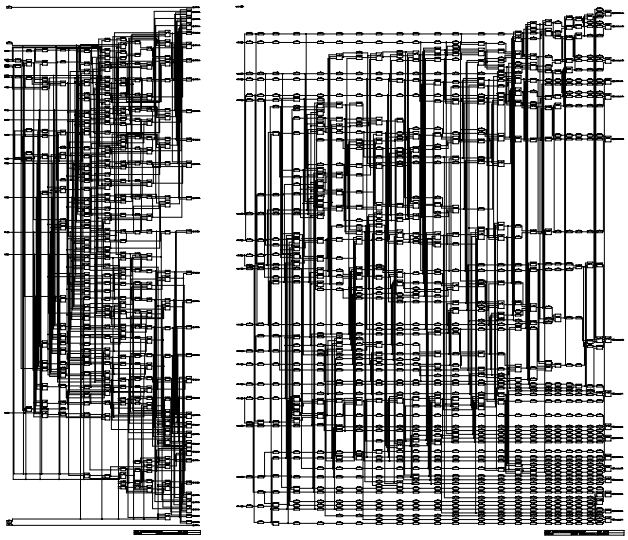

Figure 11. 8x8 CSA multiplier synth. and MPR

Both examples shown above represent a comparatively small architecture only. Currently, larger designs like a redundant CORDIC implementation [12] are processed with this design flow. Further algorithmic optimizations should yield even faster latencies.

6. Alternative applications

The tools developed for the new design flow can be applied to some other fields beyond dynamic logic. Because of the reorganization of the netlist we can also benefit from the calculated position assignment. We can exploit the column allocation for the efficient preparation of cell placement in the layout process which leads to a general pipeline-intensive datapath.

6.1. Speeding up standard logic designs

The design flow can be easily accomodated to transform standard logic into a heavily pipelined design thus greatly increasing throughput at the cost of additional pipeline registers. In this case the logic transformation module NAM is obsolete. The resulting design can be mapped to ASICs as well as commodity FPGAs.

6.2. Wave-pipeline

Due to the similarity of the task to generate a netlist for dynamic logic and the task to generate a netlist for a wave-pipelined design the program can be adapted with small modifications to this problem. The main difference between both tasks is the absence of the clocking signal. Furthermore, after the micro-pipeline step we must balance the driver

strength according to the length of the wiring and to the fan-in of the following cells. This can be easily done by choosing another cell from a library which offers different driver versions of each cell. Because of the micro-pipeline the possible fan-in of the signal drivers can be reduced which also limits the range of variation of the signal load. Now we can limit our simplified adaptation process on one column by iteration over the whole netlist.

7. Conclusions and further work

The proposed approach for the integration of dynamic logic style into a standard design flow shows one possible way to use the potential of dynamic circuit style regarding speed in a conventional design process without the disadvantage of the commonly needed manual work. Beside that, by use of the presented complementary logic implementation we can reduce the preparation effort. The design flow presented has been successfully verified on selected design projects. Re-design of the pipelineing algorithm promises a reduction of the implementation expenditure. The evaluation of the design-flow for further designs are currently in progress.

[1] P.E. Gronowski, W.J. Bowhill, R.P. Preston, M.K. Gowan, and R.L Allmon, "High-performance microprocessor design," *Journal of Solid State Circuits*, vol. 33, no. 5, pp. 676–686, May 1998.

[2] S. Posluszny, N. Aoki, D. Boerstler, J. Burns, S. Dhong, U. Ghoshal, P. Hofstee, D. LaPotin, K. Lee, D. Meltzer, H. Ngo, K. Nowka, J. Silbermann, O. Takahashi, and I. Vo, "Design methodology for a 1.0 ghz microprocessor," in *Procceedings of the International Conference on Computer Design*, 1998, pp. 17–23.

[3] M. Matson, D. Bailey, S. Bell, L. Biro, S. Butler, J. Clouser, J. Farrel, M. Gowan, D. Priore, and K. Wilcox, "Cicuit implementation of a 600mhz superscalar risc microprocessor," in *Procceedings of the International Conference on Computer Design*, 1998, pp. 104–110.

[4] J. Pihl, *Design Automation of High Speed Digital Signal Processing in VLSI with Application in Speech Recognition Systems Based on Hidden Markov Models*, Ph.D. thesis, Norwegian University of Science and Technology, Dept. of Physical Electronics, 1996.

[5] J.E. Øye, *A High Speed Cell Library in CMOS for Bit-Serial Implementation of DSP Algorithms*, Ph.D. thesis, Norwegian University of Science and Technology, Dept. of Physical Electronics, 1996.

[6] J. Pihl, J.E. Øye, and E. Aas, *Logic Synthesis with High Speed CMOS Circuit Techniques*, Kluwer Academic Puplishers, Jan. 1997.

[7] I. Sundsbø, *Analysis and VLSI design of synthesis filter bank for image subband coding*, Ph.D. thesis, Norwegian University of Science and Technology, Dept. of Physical Electronics, 1997.

[8] J. Yuan, I. Karlsson, and C. Svensson, "A true single phase clock dynamic cmos circuit technique," *IEEE Journal of Solid State Circuits*, vol. SC-22, pp. 899–901, 1987.

[9] P. Ng, P.T. Balsara, and D. Steiss, "Performance of cmos differential circuits," *Journal of Solid State Circuits*, vol. 31, no. 56, pp. 841–846, May 1996.

[10] L.G. Heller, W.R. Griffin, J.W. Davis, and N.G. Thoma, "Cascode voltage switch logic: A differential cmos logic family," in *Procceedings IEEE International Solid-State Circuits Conference*, 1984, pp. 16–17.

[11] A. Wassatsch, "Algorithmen für Umsetzung dynamischer Schaltungstechnologien," Tech. Rep., Universität Rostock, Fachbereich Elektrotechnik und Informationstechnik, Institut für Angewandte Mikroelektronik und Datentechnik, 1998.

[12] A. Wassatsch, S. Dolling, and D. Timmermann, "Area minimization of redundant cordic pipeline architectures," in *Procceedings of the International Conference on Computer Design*, 1998, pp. 136–141.