

# Wave Pipelined Circuits Synthesis

Woo Jin Kim<sup>1</sup>, Yong-Bin Kim<sup>2</sup>

<sup>1</sup>Advanced Micro Devices,  
Boxborough, MA, 01719, USA,  
Phone: 978 795 2640, E-mail: woo.kim@amd.com.

<sup>2</sup>Department of Electrical and Computer Engineering  
Northeastern University,  
Boston, MA, 02115, USA  
Phone: 617 373 2919, E-mail: ybk@ece.neu.edu.

**Abstract** – Wave pipelining is a technique which can be used to speed up the circuit without insertion of storage elements, but because of that fact, needs to be more tightly controlled when being designed. This paper – taking the wave pipeline design constraints into account – looks to automate the generation of wave pipelined design netlists through synthesis and delay balancing scripts. The results show less than 20% delay deviation between maximum and minimum delays which was the target criteria and is comparable to numbers from hand-crafted designs.

**Keywords** – Wave Pipeline, VLSI, Design Automation

## I. INTRODUCTION

Pipelining can be divided into two techniques: conventional pipelining and wave pipelining. While the wave pipeline enjoys several advantages over the conventional technique, it is also more difficult to design requiring much more care than the conventional pipeline. Wave pipeline method does not physically partition the logic and insert registers as do the conventional. Instead it utilizes the fact that, if the logic path is long enough and the data dispersion can be reasonably small, multiple sets of data can be sent through the logic at a faster clock rate and without latching the data on the way [1], [2], [3]. The delay difference between the maximum and minimum delay paths constrains the design in terms of clock frequency. This implies that the designer needs to look at not only the maximum delay but also the minimum delay and try to equalize them. This characteristic makes it challenging to automate the design of wave pipeline circuits, especially the generation of a design netlist.

This paper looks at some of the previous research done in this area and also presents a flow for synthesizing a wave pipelined design.

## II. BACKGROUND

Wave pipelining addresses the issue of logic or component utilization. Since poor logic utilization is due to the fact the conventional pipeline waits for the data signal to arrive at the end of the stage and be latched into a register before sending the next signal, wave pipelining sends in new sets of data before the earlier sets of data are latched at the output registers. This is possible because, temporally, there is a large percent of logic that could be used at the same time without interfering with each other. This has the effect of reducing the clock cycle beyond that of the conventional pipelining as more data will be traveling through the pipeline at the same time and, thus, the latching of the data at the registers will occur more frequently. The requirement that needs to be followed is that the design satisfy the clocking constraint [4], [2], [5], [6].

The clock for wave pipelined design can be expressed as follows:

$$T_{CLK} > T_{DELAY}(Max) - T_{DELAY}(Min) + T_{SETUP} + T_{HOLD} + 2 * T_{SKEW} \quad (1)$$

where,  $T_{CLK}$  is the clock cycle,  $T_{DELAY}(Max)$  is the maximum propagation delay of the design,  $T_{DELAY}(Min)$  is the minimum propagation delay,  $T_{SETUP}$  and  $T_{HOLD}$  are the setup and hold times of the register and  $T_{SKEW}$  is the clock skew. The sum of setup and hold times of the registers and the clock skew constitute the clocking overhead. Since, there are no additional registers and their inherent delays, latency is reduced when compared to conventional pipelining.

As can be seen from Equation 1,  $T_{DELAY}(Max)$  and  $T_{DELAY}(Min)$  are really the only parameters that the designer can control, and the balancing or equalizing of the maximum and minimum delays are where most of the efforts are put into.

### III. RESEARCH CONSIDERATIONS

The focus of this research is to find ways to ease the design effort on wave pipelined circuits. Up to now, most of the works – if not all – have treated wave pipelining in a custom way. This is because the manual handling of the design gives more control to the designer to implement regularity into the design, which makes it easier to balance the maximum and minimum delays. While this is the best approach in term of design quality, it can be extremely time consuming and requires a lot of effort and expertise as well as detailed knowledge of circuit from the designer which the designer might or might not have.

The most obvious plan would be to automate parts or all phases of the wave pipeline design. This would be a trade-off of quicker design but less control. The key is how good would the result be? If the design process could be automated, but if the designer had to develop the necessary programs and flows to achieve that certain level of automation, the time and effort might be better spent doing the design custom. In light of this, we looked for areas where commercial tools exist which can be used for the bulk of the automated work.

We deemed that the actual netlist generation would be the area which would require the most design expertise and the most time consuming. This phase of the design was chosen for automation and Synopsys Design-Compiler was chosen as main software as it is the most widely used synthesis engine.

### IV. LIBRARY GENERATION

Once the function and specification are set, we need to come up with a design that will perform the necessary functions. At a higher abstract level this is easy, but as it comes down to gate and circuit level it gets much more difficult. The idea here is to synthesize the netlist from an abstract level to the gate level using Synopsys Design-Compiler. This netlist will be take as the starting point and edited as necessary to achieve the desired the specification.

The first thing that must be done is to create a library of cells which can be expected to give good results. It has been noted that standard CMOS gates are not very balanced in terms of delay due to the topology. And the unbalance is made worse by each increase in the number of inputs. Wave pipelined designs using CMOS gates usually limit the input number to two [4], [2]. Number of researches have put forth results using different type of logic cells including passgate and stacked structures [7], [8], [9].

For ease of design, we felt that the CMOS logic family was best. However, instead of limiting the cell usage to 2-input nand gate, we expanded the stack concept to 2-input nor gate and inverter as well to give the synthesis tool more blocks to play with.

For his wave pipeline design, Klass used only the stacked nand gate. The nand gate was restricted to two inputs to minimize the delay variance of the gate. To increase the func-

tions that could be used by Synopsys to generate the netlist, the stacking concept was expanded to inverter and nor gate while keeping the two-input restriction. Figure 1 illustrates the expanded set of stacked CMOS logic family.

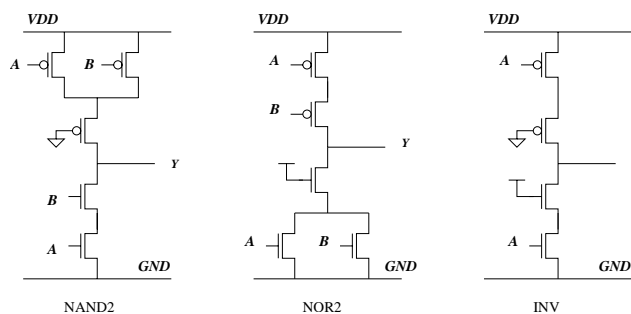


Fig. 1. Expanded Stacked CMOS Logic

One thing that stands out from Figure 1 when looking at the cells is that the depth of the transistors are the same for P and N network for all cells. One of the difficulties with regular static CMOS was that the depth is different for P and the N network and for different gates. This was a major reason Klass stacked the nand2 gate and did not make use of nor2. This configuration sets the gate depth to the output pin to two for all three cells matching the delay in the first order across the board.

The stacked CMOS gates do have slightly longer delay overall due to the depth of the transistors. But the difference between rise and fall delays is significantly better than that of regular CMOS gates. Figure 2 and Figure 3 shows the beta versus delay trend of the regular and stacked configurations.

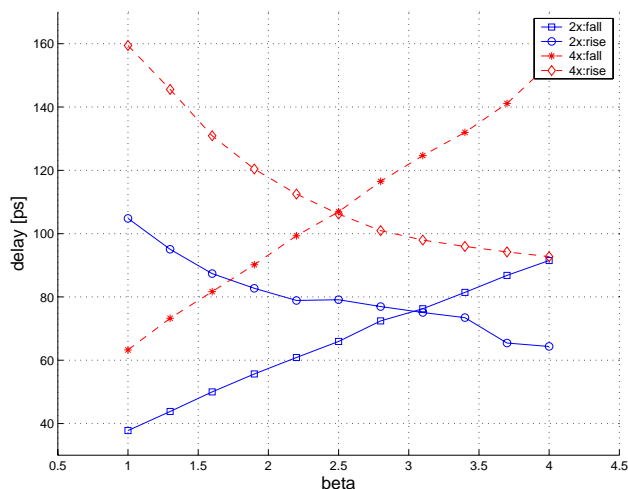


Fig. 2. Regular CMOS: Beta vs Delay

### V. DESIGN SYNTHESIS

Once the library is ready, next phase is the design synthesis. Adders and multipliers were chosen as the target designs.

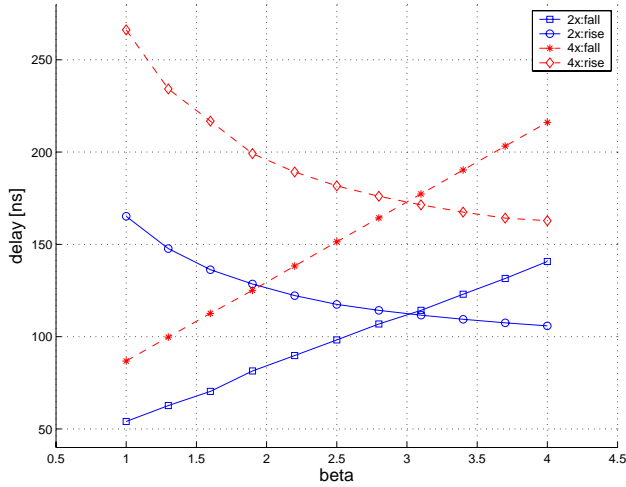


Fig. 3. Stacked CMOS: Beta vs Delay

Design	Delay		Delta	%	Max Depth
	Max [ns]	Min [ns]			
ADD4	0.72	0.60	0.12	16.7	10
ADD16	3.36	2.68	0.68	20.0	43
ADD32	6.52	3.02	2.50	53.7	85
MUL8	4.00	2.78	1.22	31.0	48
MUL16	8.55	1.60	6.95	81.0	103

TABLE I  
SYNTHESIS RESULT USING REGULAR CMOS

For adders 4, 16 and 32 bit designs were run through and for multipliers, 8 and 16 bits were done.

Table I shows the results using the regular CMOS library on the designs and Table II shows the results using the stacked CMOS library. The numbers are the final numbers after the MIN delay compilation was done for the delay matching.

For larger designs such as 32-bit adder and 16-bit multiplier, Synopsys does not do a very good job at trying to buffer up to meet the MIN constraint. For smaller designs such as 4-bit and 16-bit adder and 8-bit multiplier, the delay difference gets reduced to below 31% for both the regular and stacked CMOS which is not too bad compared to some number from an earlier research paper [7] where the delay difference for an 8-bit adder

Design	Delay		Delta	%	Max Depth
	Max [ns]	Min [ns]			
ADD4	0.84	0.72	0.12	14.3	9
ADD16	3.46	2.06	1.40	40.5	36
ADD32	6.97	4.87	2.10	30.1	65
MUL8	4.07	3.48	0.59	14.0	42
MUL16	9.40	2.18	7.22	76.8	84

TABLE II  
SYNTHESIS RESULT USING STACKED CMOS

was 21.6%. This number came from a custom design with fine tuning applied.

From the results, the stacked CMOS seem to handle the more complex circuitry better than the regular CMOS which was what we expected and the overall delay matching was better for the stacked CMOS as well except for the one hiccup with ADD16. It was noted that with more loose timing constraint, stacked CMOS got better deviation number than regular CMOS for ADD16 as well.

## VI. BALANCING ALGORITHM

It was seen from the results in Section V that Design-Compiler by itself does not handle the designs well – especially the larger designs. To better balance the delay, a balancing algorithm is applied to the netlist.

In our balancing scheme, we partition the netlist into levels but force the levels to have a gate depth of one in all cases. In this way, only one delay element per level may be inserted into any path thus limiting the overall delay element number which may be inserted. This guarantees that the gate depth of the whole design will stay the same – matched to the worst gate depth path of the design – and that the latency of the design will not change as a consequence of delay element insertion. And as less delay components will be added, the likelihood of adding delay difference due to the inherent delay variant nature of the components go down. One important underline is that the gates must be well-balanced.

Figure 4 shows the flow that our balancing mechanism will go through.

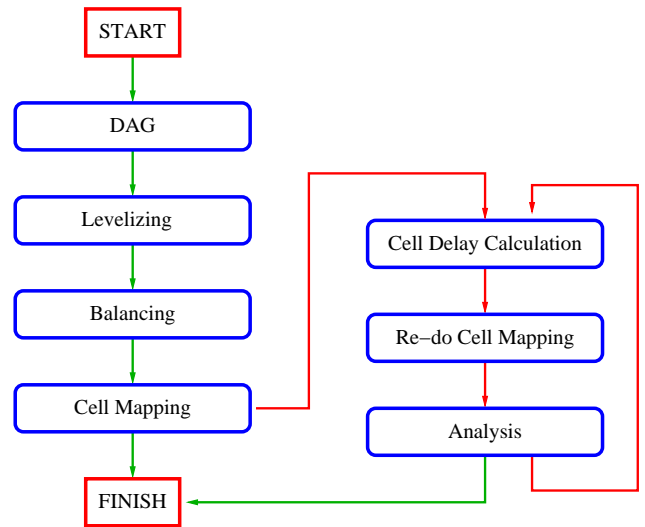


Fig. 4. Delay Balancing Flow

Once the netlist is generated, it goes through "Cell Mapping I" phase. If the netlist was generated with cells with proper drives, only this phase should be needed.

“Cell Mapping I (MapI)” partitions the netlist into levels leaving only one gate per level. If it finds paths which go through levels with no gates, default buffers are inserted for that path on those levels. Working from closest to the output, MapI goes by each inserted buffer and updates the drive of the buffers depending on the loading. MapI, then, calculates the delays for all paths and keeps information on all gate delays as well. Once the calculations are done, path delay numbers and gate delay numbers against other gates in the same level are checked, and if the results are satisfactory, MapI is done.

There is an option to change the buffers to inverters where possible but it is not mandatory.

The following subsection lists the MapI algorithm in more detail.

#### A. Algorithm: Cell Mapping I (MapI)

1. Read netlist.
2. Extract all paths. Sort paths from most gate depth down.
3. Go through all paths and assign levels to all gates. If there are empty levels or gaps in the path, insert 1X drive buffers.
4. Work backwards from the output (highest level to lowest). Find the inserted buffers and update the cell depending on the loading.
5. Calculate the delays for all paths. Keep delays for all gates.
6. Compare the path delay. If okay, done.
7. Else, compare the gate delays with other gates in the same level. If okay, go to step 9.
8. Else, send control to MapII flow.
9. If the change to inverter option is set, search through the paths to find minimum number of 3 inserted buffers in a row without branching.
10. If number is odd, leave the last buffer as is and replace the other buffers with same drive inverters.
11. If number is even, replace all buffers with same drive inverters.

MapI gives us a quick way to balance the delay of the design if all the checks are okay. However, if the checks flag, control is switched to “Cell Mapping II (MapII)” which goes over the problematic gates and resizes them as necessary. Once that is done, it does damage control and if needs be updates the drive of the gates that drive into the updated problematic gate. Again, the delay calculations are done and checks are gone through. The updates through MapII should have fixed the problems due to overdrive or underdrive of the cells and hopefully the checks are clean. If there are still problematic spots, MapII is repeated until max iteration number is reached.

The subsection below lists the MapII algorithm in more detail.

#### B. Algorithm: Cell Mapping II (MapII)

1. Read updated netlist from MapI or MapII.
2. Update the problematic gates working from highest level down. Update drivers of the problematic gates as well if necessary.

Design	Timing[ns]					
	Before			After		
	Max	Min	%	Max	Min	%
ADD32	5.54	0.07	98.9	5.84	4.42	24.3
MUL16	8.55	0.07	99.2	8.78	6.28	28.5

TABLE III  
TIMING BEFORE AND AFTER DELAY MATCH (REGULAR)

Design	Timing[ns]					
	Before			After		
	Max	Min	%	Max	Min	%
ADD32	4.31	0.11	97.5	5.07	4.37	13.8
MUL16	9.39	0.11	98.8	10.40	8.76	15.8

TABLE IV  
TIMING BEFORE AND AFTER DELAY MATCH (STACKED)

3. Re-do calculations.
4. Re-do checks done in MapI.
5. If checks are clean go to step 8.
6. If checks are not clean check to see if maximum number of MapII iteration has been reached.
7. If maximum iteration has been reached, the flow stops. Else go to 1.
8. Check if change inverter option is set.
9. If option is set, change to inverters per MapI. If not, flow is done.

## VII. RESULTS

The two larger designs which was poorly balanced using Design-Compiler were put through the balancing flow. These two tables, Table III and Table IV, show the timing numbers for the two designs after MapI was applied.

The designs which failed the balancing through the Design-Compiler was put through the MapI algorithm. Although somewhat rough, MapI did give numbers that were satisfactory. Timing-wise it settled below 20% deviation which we look at to be sort of a reference point as a lot of the earlier work on wave pipelining borders around that number. This is a huge improvement over the numbers which the Design-Compiler showed when ran alone.

The stacked CMOS gave better result 5-10% than the regular CMOS which was expected although the total delay was worse for the stacked.

## VIII. CONCLUSION

This paper attempted to apply automation to wave pipelined design. Deeming the netlist generation phase as the more trying task of the design, effort was made to synthesize the netlist

through Synopsys Design-Compiler. The resulting netlist was not quite at the quality level necessary which we set at delay deviation of 20%. This required an added step of delay balancing through MapI which was able to reduce the delay deviation to below 20% even for the larger circuit of 32-bit adder and 16-by-16 multiplier. This does require that the library of gates are of good delay balance as one of the key attribute of MapI is one delay gate insertion per level which would act as a constraint against too many delay cell insertion. The resulting numbers look promising, indicating that automation of the synthesis phase is quite feasible for wave pipelined designs. The next step on the design automation should be to automate the layout phase with the emphasis on place-and-route for delay balancing.

## REFERENCES

- [1] D. Wong, G. De Micheli and M. Flynn, "Designing High-Performance Digital Circuits Using Wave Pipelining: Algorithms and Practical Experiences", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol.12, No.1, January 1993, pp.25-46.
- [2] F. Klass, "Wave Pipelining: Theoretical and Practical Issues in CMOS", *PhD Dissertation*, Department of Electrical Engineering, Delft University, 1995.
- [3] C. T. Gray, W. Liu, W. Van Noije, T. Hughes and R. K. Cavin III, "A sampling technique and its CMOS implementation with 1 Gb/s bandwidth and 25ps resolution", *IEEE Journal of Solid State Circuits*, March, 1994, pp.340-349.
- [4] K. Nowka, "High-Performance CMOS System Design Using Wave Pipelining", *PhD Dissertation*, Department of Electrical Engineering, Stanford University, 1995.
- [5] D. A. Joy and M. J. Ciesielski, "Clock Period Minimization with Wave Pipelining", *IEEE Transaction on Computer-Aided Design of Integrated Circuits and Systems*, Vol.12, No.4, April 1993, pp.461-472.
- [6] W. Lam, R. Brayton and A. Sangiovanni-Vincentelli, "Valid Clocking in Wavepipelined Circuits", *Proceedings of IEEE Conference on Integrated Circuits Computer Aided Design*, 1992, pp.518-525.
- [7] R. Parthasarathy and R. Sridhar, "Double Pass-Transistor Logic for High Performance Wave Pipeline Circuits", *Proceedings of 11th International Conference on VLSI Design*, 1998, pp.495-500.
- [8] W. H. Lien and W. P. Burlison, "Wave-Domino Logic: Theory and Applications", *Proceedings of TAU'92*, 1992.
- [9] D. Ghosh and S. K. Nandy, "NPCPL: Normal Process Complementary Pass Transistor Logic for Low Latency, High Throughput Application", *Proceedings of VLSI Design*, January 1993, pp.341-346.