A Fine-Grain Phased Logic CPU

Robert B. Reese
Mississippi State University
reese@ece.msstate.edu

Mitchell A. Thornton
Southern Methodist University
mitch@engr.smu.edu

Cherrice Traver
Union College
traverc@union.edu

Abstract

A five-stage pipelined CPU based on the MIPs ISA is mapped to a self-timed logic family known as Phased Logic (PL). The mapping is performed automatically from a netlist of D-Flip-Flops and 4-input Lookup Tables (LUT4s) to a netlist of Phased Logic gates. Each PL gate implements a 4-input Lookup Table in addition to control logic required for the PL control scheme. PL offers a speedup technique known as Early Evaluation that can be used to boost performance at the cost of additional PL gates. Several different PL gate-level implementations are produced to explore different architectural tradeoffs using early evaluation. Simulations run for five benchmark programs show an average speedup of 1.48 over the clocked netlist at the cost of 17% additional PL gates.

1. Introduction

Various design communities view global clocking problems substantially differently. For the ASIC community, global clocking issues are still at the stage where a combination of innovative designers and hard work can solve global clocking challenges. Of course, the amount of needed innovation and hard work keeps increasing, but it is has not yet reached the 'broken' stage. A different perspective on global clocking issues exists in the programmable logic community, which can be separated into consumers and vendors. Consumers expect a combination of tool, methodology, and silicon substrate that will input RTL descriptions of complex designs and produce working implementations that run at specification. Consumers expect global clocking issues to be solved by the vendor, usually by vendor provided tools in addition to the programmable logic substrate. Consumers balance many factors in choosing programmable logic such as density, cost, speed, power, and ease of use. The ease of use is important because consumers are dependent upon vendor provided design/verification methodologies for implementation and the effectiveness of these methodologies in directly impacting time to market. Furthermore, programmable logic consumers tend to work in smaller design teams than ASIC designers, but still must deal with designs in the hundreds of thousands to millions of gates. This means that designer productivity must be high, and must continue to increase as programmable logic densities increase. Design reuse for consumers means the ability to reuse RTL blocks in new programmable logic designs, either by the same vendor or a different vendor.

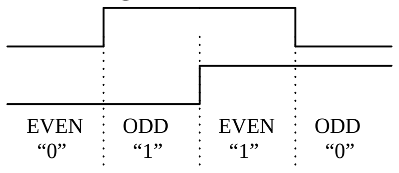
Programmable logic vendors are faced with providing the magical tool, methodology, and silicon substrate combination that programmable logic consumers desire. To deal with global clocking issues, programmable logic vendors are using the techniques of ASIC designers, except they are lagging behind ASICs by anywhere from 2-5 years in terms of the aggressiveness of their solutions. PLLs and DLLs are now common on all high end FPGAs. It is only a matter of time before high-end FPGAs begin including the local clock generation/management/active-deskewing that is now common in high-performance ASIC and CPU design [1]. Because vendors must provide a solution for global clocking issues to users, the vendors face the additional problem of consumer education in whatever methodology they provide that allows consumers to reach timing closure. Complex clocking strategies that cause a reduction in ease-of-use, difficulty in reaching timing closure, or has a steep learning curve that increases time to market may result in consumers seeking an alternate solution.

This paper discusses a self-timed design methodology known as *Phased Logic* that eliminates the need for a global clock and could form the basis for a new family of programmable logic devices. Elimination of the global clock network benefits programmable logic vendors as it provides a scalable architecture that requires no significant architecture or methodology changes as the die size grows. Our methodology satisfies the needs of programmable logic consumers in that it provides automated mapping from a netlist of D-flip-flops and combinational logic to our self-timed architecture. This means that the familiar synchronous RTL design methodology and tools used by programmable logic consumers are compatible with our proposed approach.

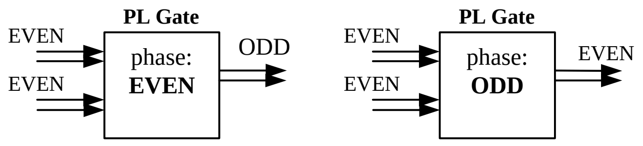
2. Phased Logic

2.1 Background

Sutherland's micropipelining [2] is a self-timed methodology that uses bundled data signaling and Muller C-elements [3] for controlling data movement between pipeline stages. Level Encoded Dual Rail (LEDR) signaling was introduced in [4] as a method for providing delay insensitive signaling for micropipelines. The term *phase* is used in [4] to distinguish successive computation cycles in the LEDR micropipeline, with the data undergoing successive *even* and *odd* phase changes. The systems demonstrated in [2][4] were all linear pipelined datapaths, with some limited fork/join capability also demonstrated, but with no indication of how general digital systems could be mapped to these structures. This problem was solved in [5] via a methodology termed *Phased Logic* (PL), which uses marked graph theory [6] as the basis for an automated method for mapping a clocked netlist composed of D-Flip-Flops, combinational gates, and clocked by a single global clock to a self-timed netlist of PL gates. Logically, a PL gate is simply a Sutherland micropipeline block with the state of the Muller C-element known as the *gate phase*, which can be either even or odd. In this paper, a fine-grain PL gate will be used that has only one output, a compute function composed of a single logic function, and which uses LEDR signaling for data. A PL gate is said to *fire* (the Muller C-element changes state) when the phase of all data inputs match the gate phase. This firing causes the output data to be updated with the result of the computation block of the gate.



a. LEDR encoding



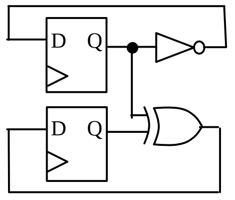
b. Ready to fire

c. After gate fires

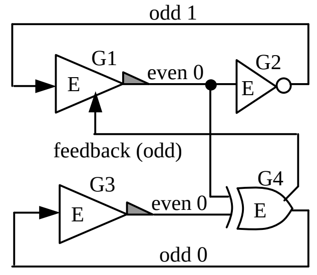
Figure 1. LEDR Encoding and PL Gate Firing

The algorithm for mapping a clocked netlist to a fine-grain PL netlist was developed in [5] and is summarized below:

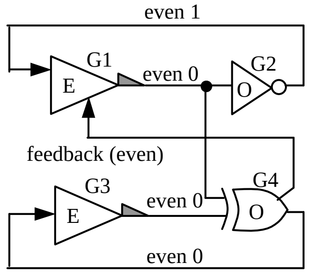
- All DFFs are mapped one-to-one to *barrier* gates in the PL netlist. A barrier gate is a PL gate whose logic function is a buffer function, and whose output phase always matches the gate phase. This means that after reset, all barrier gates will have tokens (active data) on their outputs.
- All combinational gates are mapped one-to-one to *through* gates in the PL netlist. A through gate is a PL gate whose logic function is the same as the original combinational gate, and whose output phase is always opposite the gate phase.
- Single rail signals called feedbacks are added where necessary to ensure *liveness* and *safety* of the resulting marked graph. *Liveness* means that every signal is part of a loop that has at least one gate ready to fire. *Safety* means that a gate cannot fire again until all destination gates have consumed its output data. Feedbacks cannot be added between two barrier gates because this would cause a loop with two tokens on it, violating the safety constraint. If necessary, buffer-function through gates (called *splitter* gates) are inserted between barrier gates to provide a source and termination for feedback.
- Feedbacks that originate from a barrier gate have an initial token on them since all outputs from barrier gates have tokens. This implies that feedbacks from barrier gates must terminate on a through gate.
- A feedback that originates from a through gate and terminates on a through gate must have an initial token since the output of the destination through gate will not have an initial token.
- A feedback that originates from a through gate and terminates on a barrier gate must not have an initial token since the output of the destination barrier gate will have an initial token.



a. Clocked 2-bit Counter



b. Translated PL Counter (reset state)



c. PL Counter after G2,G4 fire

Figure 2. Translation and Fire of a 2-bit Counter

A signal that is part of a loop that is both live and safe is said to be *covered*. All signals in the circuit must be covered to satisfy liveness and safety. Signals that are part of naturally occurring loops that satisfy liveness and safety critera are already covered and do not require feedbacks. It is possible for a single feedback signal to create a loop that covers multiple signals. Figure 1 illustrates the LEDR encoding of the dual rail signals used between PL gates. Notice that the "value signal" (top), is the actual logical value and the two signals together define the phase. A sample gate firing is also shown in Figure 1.

Figure 2 illustrates the translation of a clocked 2-bit counter to a PL netlist and a sample firing of the circuit. The signal between gate G4 and G1 in the PL netlist is a feedback net added to ensure safety.

2.2 A Fine-Grain PL Gate

Figure 3 shows a Phased Logic gate [9] (*PL4gate*) that uses a 4-input Lookup Table (LUT4) for the computation element. The Muller C-element is used to detect operand arrival (*a*,*b*,*c*,*d*) from the LEDR inputs. The *fi* input is a feedback input to the gate; an external C-element is used to concentrate multiple feedbacks if required. The *feedback_out* signal is used to provide feedback to gate sources if needed (the negation of this signal is also provided but not shown). The gating signal (GC) to the output latches is delayed until the *LUT output* value is stable.

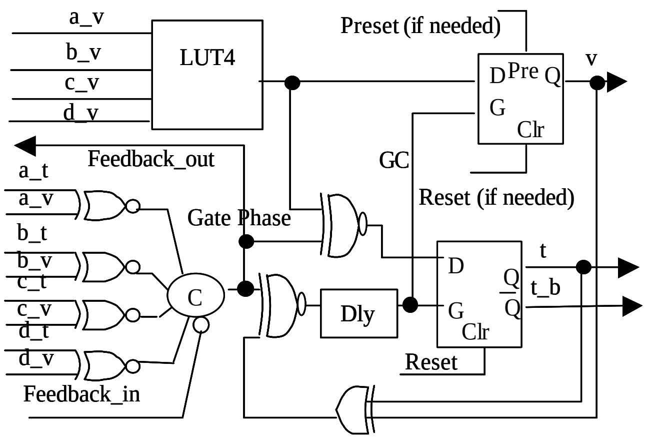


Figure 3: LUT-4 Based Phased Logic Gate

A PL netlist is continually firing even if no data values are changing, just as a clock signal toggles even in the presence of unchanging data. There is a one-to-one mapping of clock cycles in the original clocked netlist to computation cycles in the PL netlist. One advantage of this aspect of a PL system is that it makes it easy to interface a PL system to a clocked system as long as the time for a computation cycle in the PL system is less than the clock period of the clocked system. One disadvantage of continuous token circulation is that the PL control network is always dissipating power. For power efficiency, each PL computation function (i.e., a LUT4) should be large compared to the control logic in order to amortize the control power over a large compute function. A power savings feature in fine-grain PL systems is that latches on the output of each PL gate tend to reduce the number of transient computations in a fine-grain PL system. In [7] it was shown that PL systems using the gate in Figure 3 could be more power efficient than the clocked equivalents.

2.3 Early Evaluation

One disadvantage of micropipelines, and PL systems in general, is that the output latch latency adds to the critical path delay. However, there are other features of PL that can overcome this gate-level performance disadvantage and allow PL systems to outperform their clocked equivalents.

"Early evaluation" [7] is said to occur when a PL gate fires upon arrival of a subset of inputs. This can improve performance as it adds parallelism to the system; gates downstream of the early evaluation gate can be firing in parallel with the gates producing the 'tardy' inputs. Early evaluation gates do not introduce a liveness problem because they do not alter the number of gate firings, only the sequence of gate firings. To maintain safety, all inputs/outputs of early evaluation gates must be in a loop that has a feedback signal that originates from or terminates on the early evaluation gate.

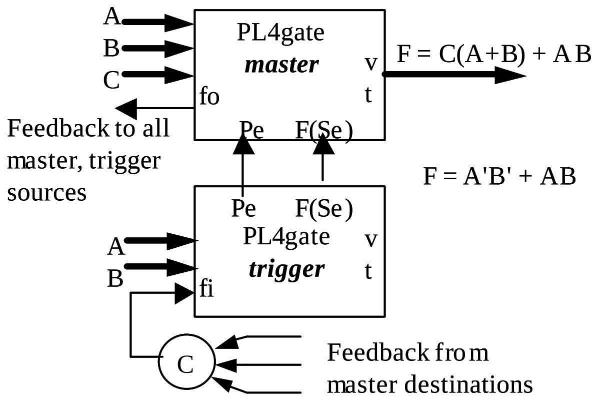


Figure 4: An Evaluation Gate from Two PL Gates

Figure 4 illustrates how an early evaluation gate can be constructed from two PL4gates. The upper gate is termed the *master* and contains the normal logic function. The lower gate is named the *trigger* and contains the early evaluation function based on a subset of the inputs to the master gate. In this example, the master gate implements the carry function for a full adder (A and B are data bits, and C is the carry-in bit). The trigger function is the logical OR of the kill and propagate functions which allows the master gate output to fire upon arrival of only the A and B inputs. This speedup technique for the carry chain of an adder is a well-known self-timed speedup mechanism for addition but PL allows it to be generalized to any logic function [8].

2.4 Loop Delay Averaging

The cycle time of a PL system is bounded by the longest register-to-register delay in the original clocked netlist, but the average cycle time can be less than this value because of the averaging of loop cycle times of different lengths [5]. The circuit in Figure 5 shows a two-stage, unbalanced pipeline. The *DF* block in each circuit represents a D-flip-flop, and the G block a combinational block. The dot shown on particular signals represent the initial tokens (active data) for the PL netlist; the dashed nets are feedback signals added in the PL system for liveness and safety.

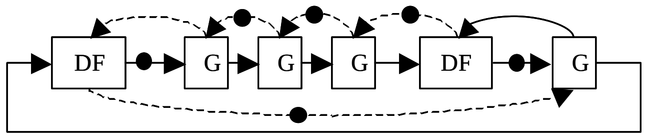


Figure 5: Two-stage, unbalanced Pipeline

If each combinational gate has a delay of 10 units, and the DFF delay plus setup time is also 10 units, then the longest path in the clocked system would be 40, or 4 gate delays. To simplify this particular explanation, we assume that a PL gate has the same delay as its corresponding gate in the clocked netlist. Analysis verified by simulation shows that each gate in the PL system fires in a repeating pattern of 40 time units, 20 time units, for an average delay of 30 time units. Note that if the original clocked system had balanced pipeline stages, then the longest path would be 30 time units. This automatic averaging of loop paths gives more freedom in the placement of logic between DFFs. Even if logic is balanced between pipeline stages in the clocked system, early evaluation firings can create unbalanced loop delay times and delay averaging of these different loop times will still occur.

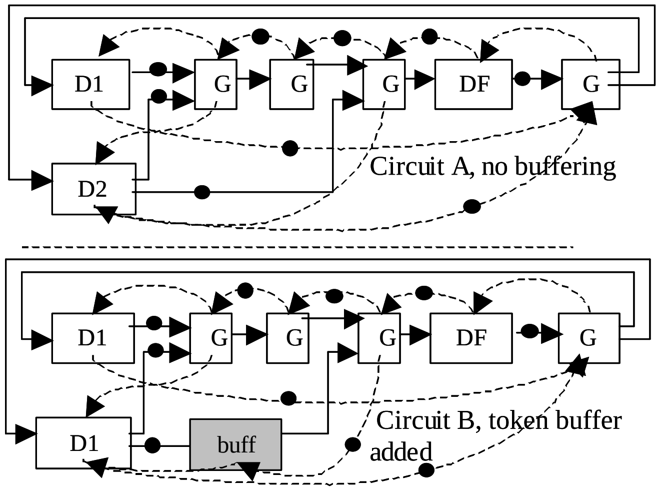


Figure 6: Token buffering to improve performance

2.5 Token Buffering

The flow of data within a PL system can be inhibited if there are not enough gates within a path to take advantage of the available parallelism [11]. Circuit A in Figure 6 is a minor modification of the two-stage unbalanced pipeline of Figure 5. Simulation results show that Circuit A fires in a repeating pattern of 40, 40 time units which is lower performance than Figure 5. However, adding a buffer as shown in Circuit B changes the fire pattern to 40, 20 for an average of 30 time units, the same as Figure 5. We call this buffer a *token* buffer, and it adds no functionality to the circuit, but does increase performance.

3. Comparisons to Other Work

Phased Logic is unique in that it offers an automated mapping from a clocked system to a self-timed system from the *netlist* level. This allows a designer to produce the netlist using familiar design tools and HDLs with the restriction that the clocked netlist has only one global clock. Most asynchronous and self-timed design methodologies [10] use custom synthesis tools and HDLs for design specification and this requires a substantial time investment on the part of the designer to learn the new methodology.

A self-timed design methodology known as Null Convention Logic (NCL) [13] allows the use of standard HDLs (VHDL) but it places restrictions on how the RTL is written and what gates the RTL is synthesized to. The NCL synthesis methodology requires that the RTL be written in a restrictive manner that separates the combinational logic and storage elements, because the NCL synthesis methodology uses a different synthesis path for registers versus combinational logic. This prevents the use of third party RTL without a significant effort to rewrite the RTL in the NCL style. Designers must also specify the data completion structures and request/acknowledge logic needed at each register, which is an added burden on the designer. The RTL is synthesized to a restricted subset of functions that is then mapped to a set of predefined macros that can be implemented in NCL. NCL uses dual rail signaling where one rail represents TRUE and other FALSE. When both wires are negated this is known as the NULL state; when one wire is high this is the DATA state. A computation asserts one of the rails (the DATA state) followed by the NULL state after an acknowledgement is received. This means that during each computation half of the wires are making a transition from ‘0’ to ‘1’ and back to ‘0’. By contrast, in PL half of the wires make a single transition during a computation; either to ‘1’ or ‘0’. In [14], Theseus logic applies NCL to the Atmel FPGA architecture by replacing the cell internals with a N-of-4 NCL macrocell that can implement 9 different combinational functions: two 2-input functions, three 3-

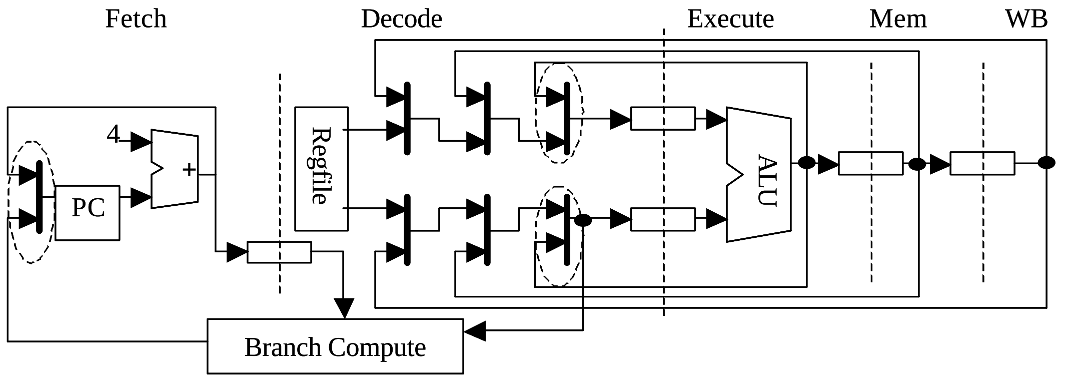


Figure 7: Simplified CPU Datapath Diagram

input functions, and four 4-input functions. The NCL macro cell is inherently state holding so registers are implemented with this same macrocell. Combinational logic must be synthesized to this limited subset of logic functions. By contrast, PL allows any possible LUT4 function. It is difficult to compare NCL performance to Phased Logic performance or to clocked performance because there are no published NCL papers that contain this performance information. Most importantly, there is no concept of early evaluation in NCL so this speedup path is not available to NCL.

A self-timed FPGA based upon LUT3s and using LEDR encoding was presented in [15]. The cell design presented in Figure 3 is a variation of the cell design used in [15]. In [15], three feedback inputs are included in each cell, so the Muller C-element has 6 inputs (3 data, 3 acknowledge). The author uses the cell in the context of Sutherland's micropipelines [2] and self-timed iterative rings [11]. Both methods require a feedback signal for each output destination. The PL methodology removes the need for a feedback for every output signal destination as multiple signals along a path can be covered by the same feedback signal, and some signals need no feedback signal if they are already part of a loop.

An FPGA-based architecture for asynchronous logic is also proposed in [16]. This FPGA architecture was aimed at accommodating a range of asynchronous design styles, and allowed for mixed synchronous and asynchronous designs. All signals were single rail. By contrast, our proposed function block is only intended for supporting the PL design style, and thus implements PL designs more efficiently than [16]. Two other asynchronous FPGA architectures are presented in [17][18][19]. These proposed architectures are for bundled data systems, which are not delay insensitive and thus require programmable delay elements. Neither approach allows automated synthesis to the architectures. Finally, prototyping of asynchronous circuits in commercially available FPGAs is demonstrated in [20]. While it is possible to implement a limited set of asynchronous circuits in current FPGAs, the fact that these FPGAs are optimized for clocked designs means that these mappings are far from optimal and would be much better served via a custom architecture.

4. A Phased Logic CPU

Previously published designs that have been mapped to Phased Logic include an iterative multiplier, a filter datapath, and arithmetic structures [7]. A test case that has been used in the past for other asynchronous methodologies is a CPU implementation such as the MIPs integer subset [21], the ARM processor [22], and the 8051 [23]. However, there are more important reasons for mapping a CPU to PL other than for testing the methodology. Processors have become important features of programmable logic families, either implemented as a soft macro [24] or as a hard macro. Soft macro processors are parameterized by bus width and synthesized from an RTL description. As such, it is important to show that Phased Logic can support this methodology.

In order to show that the Phased Logic methodology is compatible with RTL written by others, we searched the WWW for freely available processors specified in RTL. Our search produced a VHDL specification of a MIPs ISA (integer subset) implemented as a 5-stage pipeline [12]. The documentation provided with the model has synthesis and simulation results that shows the model is functional when targeted to a Xilinx XC4000 device. We found the processor to be functional as both RTL and when synthesized to a netlist of LUT4s and DFFs. The CPU was implemented with the standard fetch, decode, execute, memory and writeback stages. A simplified diagram of the CPU datapath appears in Figure 7 (the memory interface is not shown). Because the design was intended for an FPGA, the register file RTL used edge-triggered devices instead of latches. The ALU did not implement a multiplication operation. Forwarding paths were used to solve data hazards in the pipeline without having to use stalls. The MIPS branch delay slot plus the use of a forwarding path for the branch computation meant that no stalls were needed for branch or jump instructions. The same memory interface was used by both fetch and memory stages, so a stall was generated whenever the memory interface was required by the memory stage.

4.1 Mapping to a PL Netlist

Figure 8 shows the methodology used to go from RTL to a gate-level Phased Logic netlist. Synopsys Design Compiler is used as the synthesis tool for mapping the RTL to an EDIF netlist of D-flip-flops and LUT4s. A DesignWare Library optimized for LUT4s is used to map arithmetic operations.

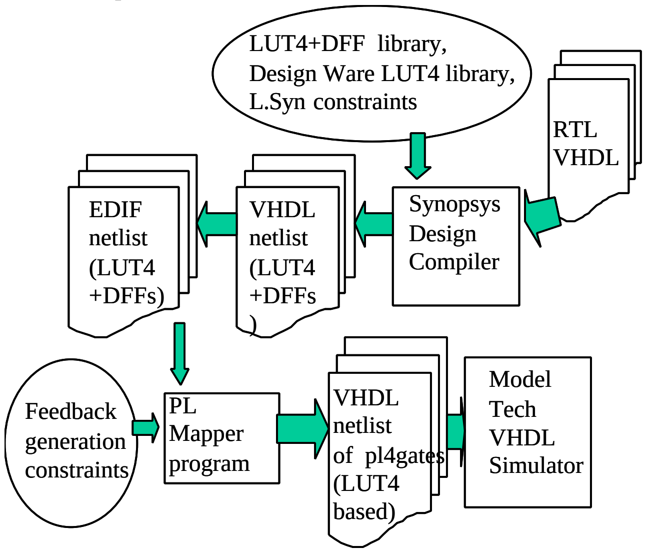

LUT4+DFF library,
Design Ware LUT4 library,
L.Syn constraints
RTL VHDL
EDIF netlist (LUT4 +DFFs)
VHDL netlist (LUT4 +DFFs)
Synopsys Design Compiler
Feedback generation constraints
PL Mapper program
VHDL netlist of pl4gates (LUT4 based)
Model Tech VHDL Simulator

Figure 8: Clocked netlist to PL netlist methodology

The mapping program reads the EDIF netlist and creates a VHDL netlist of PL4gates and 4-input Muller C-elements. The Muller C-elements are used for feedback concentration when multiple feedbacks converge on one PL4gate. The PL4gate used in the VHDL netlist is the same gate as described in Figure 3. The model was modified to replace the bidirectional data bus with a separate I/O databus as our mapping program does not yet support tri-state or bi-directional gates. The RTL operators for addition/subtraction in the ALU, for branch computation, and for PC+4 increment were replaced with Synopsys DesignWare components that were optimized for LUT4s. This was done to improve the quality of the resulting implementation. The Mentor Graphics Modelsim environment was used for simulating the original RTL and the PL netlist. The clock period for the RTL simulation was set based upon the longest timing paths reported by Synopsys after the design was synthesized to a netlist of D-flip-flops and LUT4s. The gate delay of the PL4gate was set to 40% higher than a LUT4 delay to account for the output latch latency. This value was derived from delay values for LUT4s and D-flip-flops from Altera and Xilinx datasheets. This means that the PL netlist begins with a 40% performance penalty when compared to the clocked netlist.

Our mapping program produced several versions of the PL CPU; these versions are summarized in Table 1.

Table 1: Fine-Grain CPU Implementations

Version	% Extra LUT4s
a) No EE gates, no token buffering (4855 LUT4s)	0%
b) Manually inserted EE gates, no token buffering	1.5%
c) Version (b) + token buffering	2.1%
d) Version (c) + automated insertion of EE gates, with trigger gates chosen by a cost function that weights signal arrival times with a trigger function coverage of 50% or better	16.9%
e) Version (c) + automated insertion of EE gates on all LUTs with input signal arrival time differences of one LUT delay or better	38.4%

Version (a) used no Early Evaluation (EE) gates and did not have any token buffers. Version (b) used manually inserted EE gates in the form of multiplexers in datapaths where one operand had a significantly earlier arrival time than the other operand. One of these datatpaths was the ALU result forwarding path to the idecode stage and branchpc computation. An example of ALU forwarding is shown below in which the result of the first addition is required as an operand in the second instruction.

```
Add    r5, r6, r9
Add    r4, r5, r10
```

A second datapath that used these manually inserted EE gates was the branchpc computation; if the instruction was not a jump or branch, then the PC+4 value required for the next instruction fetch could be produced faster. The multiplexers that interfaced the external input databus to the rest of the CPU were also replaced with EE gates. If the instruction was not a load word (*lw*), this allowed the rest of the CPU to proceed without having to wait for the memory interface to fire. The circled multiplexers in Figure 7 show where early evaluation gates were added in the ALU forwarding path and the branchpc computation.

Examination of the simulation results for version (b) indicated that performance was limited by the lack of token buffers in the execute stage where the ALU opcode fanned out to several levels of logic without buffering. Version (c) included buffers on these control lines that were inserted manually by modifying the RTL VHDL.

We plan on adding automated insertion of token buffers in the next version of our mapping program.

Our mapping program supports automated insertion of EE gates [8] by traversing the netlist and searching for trigger functions based upon signal arrival times. Version (d) augmented version (c) with automated insertion of EE gates chosen by a cost function that weighted signal arrival times with a trigger function coverage of 50% or better. Version (e) inserted the maximum number of EE gates by inserting an EE gate for any LUT4 that had signal arrival differences of one LUT4 delay or better.

4.2 Simulation Results

The VHDL PL fine-grain netlists were simulated using the Mentor Modelsim environment. The output latches of Figure 3 were assigned a 0.4 LUT4 delay. Four input C-elements that were used for feedback concentration were assigned a 0.6 LUT4 delay. Five benchmark programs were used for performance measurement: (a) fibonnaci (fib), a value of 7 was used, (b) bubblesort, an array size of 10 was used, (c) crc, calculate a CRC table with 256 entries, (d) sieve – find prime numbers, stopping point set to 40 (e) matrix transpose - a 20x30 matrix was used.

Table 2: CPU results for CRC program

	Fast Mem		Slow Mem	
Version	CRC	CRC (RO)	CRC	CRC (RO)
(a)	-1.05	-1.05	-1.31	-1.31
(b)	1.08	1.11	1.08	1.11
(c)	1.28	1.38	1.09	1.11
(d)	1.45	1.55	1.10	1.11
(e)	1.46	1.56	1.10	1.11

All programs were written in *C* and compiled with *gcc* using the *–O* option to produce an assembly language file that was then assembled via a *Perl* script to produce an input file read by the VHDL memory model. Table 2 shows the speedup results for the fine-grain PL CPU compared to the clocked netlist for the *crc* program. The clocked netlist was simulated with a clock cycle time of 24 LUT4s which was the critical path as reported by Synopsys, which ran through the execute, branch computation, and idecode stages. The speedup was calculated by dividing the longer simulation time by the smaller simulation time, with a negative sign used to indicate a slowdown of the PL netlist compared to the clocked netlist. The memory access time was set to the maximum slack allowable in the 24 LUT4 cycle time of the clocked netlist so that memory was not the bottleneck for the clocked netlist. The "slow mem" columns uses this memory access time for the PL netlist, while the "fast mem" columns assume that memory bandwidth can be increased such that it is not the bottleneck in the PL netlist. The CRC columns show the speedup for the different fine-grain PL versions for the *crc* benchmark. The 5% slowdown for the non-EE, fast memory case is not as bad as expected given the 40% latency penalty of the output latch for each fine-grain PL gate. Clearly, loop averaging is helping to overcome this delay penalty. The addition of the manual EE gates produced a small speedup, but this speedup was limited by lack of buffering. The large jump in performance between versions (b) and (c) indicates the importance of buffering to take full advantage of available parallelism in the netlist. The automated insertion of EE gates provided another sizeable increase in performance. However, there is diminishing returns on performance as shown with the negligible performance increase between versions (d) and (e). We also do not rule out the possibility that the performance of version (e) is being limited due to inadequate buffering. An area of future work is a performance tool that can be used to identify bottlenecks due to improper buffering.

The CRC RO columns use a version of the benchmark that has the instructions manually reordered in order to decrease the amount of ALU operand forwarding. For example, a typical code segment produced by *gcc* is shown below:

```
      addi   r4,r4,1
      slti   r2, r2, 8
      bne    r2, r0, L10
```

ALU forwarding is required for the *bne* instruction because *r2* is a destination in the *slti* instruction, and a source in the *bne* instruction. However, the instructions can be reordered as shown below:

```
     slti   r2, r2, 8
     addi   r4,r4,1
     bne    r2, r0, L10
```

Functionally, the two code streams are equivalent, but the second code stream does not require ALU forwarding for the *bne* instruction, which increases the number of early evaluation firings and hence the performance. Instruction reordering was done manually by examining the assembly code of the critical loops. Table 3 shows the results for all fine grain CPU versions for the re-ordered instruction benchmarks under the fast memory assumptions.

Table 3: Fast memory, reordered inst. benchmarks

Pgm	Ver (a)	Ver(b)	Ver(c)	Ver(d)	Ver(e)
Fib	-1.05	1.10	1.37	1.50	1.51
Bubbl	-1.05	1.09	1.30	1.44	1.45
CRC	-1.05	1.11	1.38	1.55	1.56
Sieve	-1.05	1.08	1.27	1.43	1.45
Tpose	-1.05	1.09	1.32	1.47	1.50

Avg	-1.05	1.09	1.33	1.48	1.49

Table 4 shows performance results for streams of individual instructions executed on version (d) using the fast memory assumption. The average cycle time is given in LUT4 delays and the fire pattern is the repeating pattern of cycle times for the instruction stream. The jump and branch streams are two instruction streams because of the branch delay slot of the MIPs ISA; a *nop* was placed in the branch delay slot.

Table 4: Individual instruction timings

Instr. Seq. (run on PL(d)	Avg Cycle Time	Fire Pattern
Jump, nop	14.4	15.4,13.4
branch, nop	17.0	19.8,14.0
load	15.6	17.4,12.6,16.2,18.0
store	14.4	14.8,14.0
add (nofwd)	14.4	15.4,13.4
add (fwd)	16.8	16.8
and (nofwd)	14.4	13.0,15.8
and (fwd)	14.8	13.2,15.6
shift (nofwd)	14.4	15.6,13.2
shift(fwd)	16.8	16.8

The descriptors *fwd, nofwd* indicate if ALU operand forwarding was done between instructions. In general, the instructions that required forwarding are slower than those that do not. The logical instructions are the fastest with branches being the slowest. One of the reasons for the complex fire pattern of the load instruction is the stall that is generated during the load operation.

Table 5: Instruction Statistics for Benchmarks

	fib	bubble	crc	sieve	mtpose
brnch	13.4%	17.0%	19.7%	21.0%	7.7%
jmp	12.9%	0.0%	4.7%	0.3%	0.3%
log	10.0%	0.6%	16.3%	0.7%	0.7%
shift	19.9%	25.3%	34.9%	26.8%	26.4%
slt	0.0%	23.0%	11.6%	18.5%	10.5%
addsub	12.2%	17.5%	11.6%	22.0%	32.5%
lw	17.4%	15.3%	0.0%	3.3%	7.5%
sw	14.2%	1.3%	1.2%	7.4%	14.4%

Table 5 shows the dynamic instruction frequency for the benchmarks. The *crc* benchmark has the largest percentage of logical instruction which helps explains its high performance.

Table 6: Early Evaluation Statistics

	mem efire	branch efire	alufwd efire
fib	89%	74%	77%
bubbl	85%	82%	65%
crc	100%	76%	76%
sieve	97%	75%	64%
mtpose	92%	92%	73%

Table 6 shows the percentage of early firings in terms of instruction cycles for CPU version (c) executing the reordered benchmarks under the fast memory assumption. The *crc* benchmark had 100% early firing of the memory databus input multiplexer because this benchmark did not read memory; it started with a fixed seed for the CRC table. Note that the *fib* benchmark had the highest number of ALU operand forwarding early fires, which helps to explain its status as the second highest performing benchmark.

4.3 Netlist Statistics

The netlist statitistics for the designs are shown in Table 7.

Table 7: Netlist Statistics

	Ver a	Ver b	Ver c	Ver d	Ver e
Signal Fanout	17149	17558	17614	17614	17614
% unsafe	72.7%	83.8%	83.9%	84.1%	84.1%
Signal Nets	6346	6569	6164	6164	6164
Fdbck Nets	4578	4762	4791	4799	4799
Max FB Inputs	239	244	164	164	164
4-input Celem	2596	3153	3168	3204	3225
LUT4s	4855	4928	4957	5674	6719
%increase (LUT4s)		1.5%	2.1%	16.9%	38.4%

The *signal fanout* is interesting as this represents the maximum number of signals that would need to be covered by feedback signals. The worst case in terms of required feedback signals would be one feedback net for each signal fanout. However, some signals are part of naturally occurring loops and do not have to be covered. The *%unsafe* row shows the percentage of signals that must be covered via feedback net insertion. The *max fb inputs* row is the maximum number of feedbacks concentrated at a PL gate. This large concentration was due to a *rdy* signal used in the bus interface that can be used to halt the processor. The *rdy* signal acts as a conditional load on many of the registers within the CPU. Any large fanout, synchronous signal in the clocked netlist will generate a large number of feedbacks terminating at the originating PL gate in the PL netlist. In versions c, d, and e some buffering was inserted on this signal path to reduce the maximum number of required feedback inputs. Trees of 4-input C-elements were inserted for feedback signal concentration. These C-elements would have to be included as a separate resource on the programmable substrate, perhaps one four-input C-element for each PL gate. Figure 9 shows the distribution of inputs for the C-elements in version (d) of the CPU. The large input C-elements are rare as can be seen from the figure. The spikes at 32 and 33 are due to the 32-bit datapaths in the design.

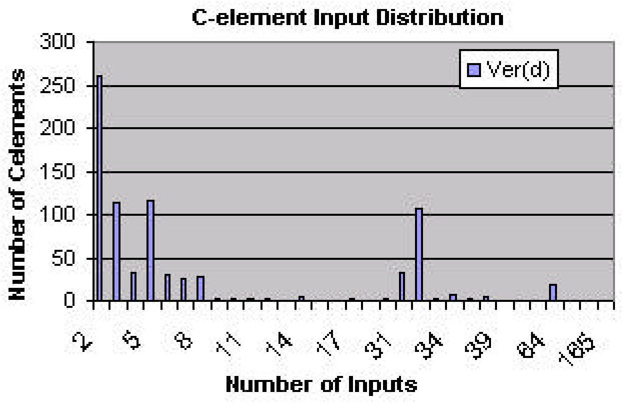


Figure 9: Distribution of Inputs for C-elements

All of the PL designs in Table 5 were generated with feedback signal lengths restricted to the preceding gate level, or a path length of one gate. This means that the maximum number of feedbacks was inserted into all of these designs. Table 8 shows the effect of different feedback arrangements on version (a) (the non-early evaluation gate netlist) on the speedup of the *bubblesort* benchmark.

Table 8: Effect of Feedback on Performance

PLEN	Speedup	Feedback Signals
1	-1.05	4578
2	-1.05	4472
4	-1.05	4397
8	-1.31	4401
∞(13)	-1.05	4401

The PLEN parameter specifies the number of gate levels to trace back from an originating gate to look for feedback termination candidates. The last row shows PLEN set to a large value with the number in parenthesis the maximum feedback length actually used in the netlist. The current feedback generation algorithm cycles through all gates in the netlist, ranking candidate feedback termination gates by the scoring function shown below:

fbscore = (covered_signals) – (fb_inputs)
– (0.1* path_length)
+ (covered_signals – path_length)

The *covered_signals* term is the number of signals made safe by adding this feedback. The *fb_inputs* term is the number of feedbacks already terminating at this node; a negative weight encourages spreading feedbacks among gates. The *path_length* term is the number of gate levels from the feedback source to the destination. The third term in the score assigns a small penalty based on feedback length. The fourth term assigns a penalty that is the difference between the number of signals covered and the feedback length. Ties are broken in favor of shorter feedback signals. The current algorithm does not consider netlist performance when placing feedbacks; its only goal is to reduce the number of feedbacks. No claims are made as to the optimality of this heuristic approach.

In Table 8, the number of feedbacks decreases as PLEN increases, because feedbacks are allowed to traverse multiple gate levels and hence cover multiple signals. However, the decrease in the number of feedbacks is small because the register file has a large number of paths that have only one gate between two D flip-flops. This forces feedbacks with a path length of 1, and increasing the PLEN parameter will not affect these feedbacks. The scoring function does not guarantee the minimum number of feedbacks as evidenced by the fact that the cases of PLEN=8,∞ has more feedbacks than for PLEN=4.

The speedup column illustrates a problem with the current feedback placement algorithm in that the case of PLEN=8 had significantly lower performance. This means that a feedback net critical to performance just happened to be placed correctly for the case of PLEN=∞, and incorrectly for PLEN=8. An area of future work is to incorporate PL system performance considerations into the feedback net placement algorithm.

5. Acknowledgements

The authors would like to thank Lokesh Shivakumaraiah and Pritam Kokate for assistance in running benchmarks, the creation of the C-to-object code path, and for RTL modifications that removed the tri-state memory bus. Thanks also go to Kenneth Fazel for discussions on parameter settings for controlling insertion of early evaluation gates. This work was funded in part by an award from the National Science Foundation (CCR-0098272).

6. Summary

This paper has presented a design methodology known as Phased Logic (PL) that allows a netlist of D-flip-flops and combinational logic clocked by a single global clock to be automatically mapped to a self-timed circuit. This methodology was applied to a publicly available RTL VHDL model of a 5-stage pipelined MIPs. The RTL was synthesized via a commercial synthesis tool to a netlist of D flip-flops and 4-input Lookup tables and then mapped to several different self-timed PL netlists that explored various tradeoffs in the automated mapping process. One benefit of PL is averaging of loop cycle times, which means that performance is not necessarily limited to the longest delay path between registers. This fact was clearly illustrated in the measured performance data that showed a PL netlist without early evaluation only had a 5% slowdown instead of the expected 40%. Another benefit of PL is that speedup can be achieved by inserting early evaluation gates which trigger when only a subset of their inputs arrive. The simulation results also demonstrated the importance of buffering in order to take full advantage of the available parallelism in the netlist. Four different PL designs that varied the number of early evaluation gates and buffering gates were simulated using a suite of five benchmark programs and showed average speedups of between 1.09 and 1.49 when compared to the clocked netlist. These results indicate that fine-grained PL systems would be advantageous from a performance viewpoint as the basis for a new family of SRAM-based FPGAs.

7. References

[1] S. Tam, S. Ruso, U. Desai Nagarji, R. Kim, Ji Zhang, I. Young, "Clock generation and distribution for the first IA-64 microprocessor", *IEEE Journal of Solid-State Circuits*, Volume 35, Issue 11, Nov 2000, pp. 1545-1552.

[2] I. Sutherland, "Micropipelines", *Communications of the ACM*, Vol 32, No. 6, June 1989, pp. 720-738.

[3] D.E. Muller and W. S. Bartky, "A Theory of Asynchronous Circuits", in *Proc. Int. Symp. on Theory of Switching*, vol. 29, 1959, pp. 204-243.

[4] M.E. Dean, T.E. Williams, and D.L. Dill, "Efficient Self-Timing with Level-Encoded 2-Phase Dual-Rail (LEDR)," in *Advanced Research in VLSI*, 1991, pp. 55-70.

[5] Daniel H. Linder and James C. Harden, "Phased Logic: Supporting the Synchronous Design Paradigm with Delay-insensitive Circuitry." *IEEE Transactions on Computers*, Vol. 45, No 9, September 1996, pp. 1031-1044.

[6] F. Commoner, A. W. Hol, S. Even, A. Pneuli, "Marked Directed Graphs", *J. Computer and System Sciences*, Vol. 5, 1971, pp. 511-523.

[7] R. B. Reese, M. A. Thornton, and C. Traver, "Arithmetic Logic Circuits using Self-timed Bit-Level Dataflow and Early Evaluation", Proceedings of the 2001 International Conference on Computer Design, September 2001, pp. 18-23.

[8] M. A. Thornton, K. Fazel, R.B. Reese, and C. Traver, "Generalized Early Evaluation in Self-Timed Circuits", *Proc. Design, Automation and Test In Europe (DATE)*, Paris, France, March 4-8, 2002.

[9] C. Traver, R. B. Reese, M. A. Thornton, "Cell Designs for Self-timed FPGAs", Proceedings of the 2001 ASIC/SOC Conference, September 2001, pp. 175-179

[10] Scott Hauck, "Asynchronous Design Methodologies: An Overview", *Proceedings of the IEEE*, Vol. 83, No. 1, January, 1995, pp. 69-93.

[11] M.R. Greenstreet, T.E. Williams, and J. Staunstrup, "Self-Timed Iteration", *VLSI '87*, C. H. Sequin (Ed.), Elsevier Science Publishers, 1988, pp. 309-322.

[12] Anders Wallander, "A VHDL Implementation of a MIPS", Project Report, Dept. of Computer Science and Electrical Engineering, Luleå University of Technology, http://www.ludd.luth.se/~walle/projects/myrisc.

[13] Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, Alex Kondratyev, "Asynchronous Design Using Commercial HDL Synthesis Tools", In *Sixth Int. Symp. on Advanced Research in Asynchronous Circuits and Systems (Async 2000)*, Eilat, Israel, April 2000.

[14] K. Meekins, D. Ferguson, M. Basta, "Delay insensitive NCL reconfigurable logic", Proceedings of the IEEE Aerospace Conference, Vol. 4, 2002, pp. 1961-1967.

[15] Dana L. How, "A Self Clocked FPGA for General Purpose Logic Emulation", in proceedings of *IEEE 1996 Custom Integrated Circuits Conference*, 1996, pp. 148-151.

[16] Scott Hauck, Steven Burns, Gaetano Borriello, Carl Ebeling, "An FPGA for Implementing Asychronous Circuits", *IEEE Design and Test of Computers*, Fall 1994, pp. 60-69.

[17] R. Payne, "Asynchronous FPGA Architectures", IEE Proc.-Comput. Digit. Tech, Vol. 143, No. 5. September 1996, pp. 282-286.

[18] Maheswaran, K., "Implementing self-timed circuits in field programmable gate array", Master's thesis, U.C.Davis, 1995.

[19] R. Payne, "Self-timed FPGA Systems", 5th International workshop on Field programmable logic and applications, LNCS 975, 1995, pp. 21-25.

[20] E. Brunvand, "Using FPGAs to implement self-timed systems', J. VLSI Signal Processing, 1993, Vol. 6, No 2, pp. 173-190.

[21] A. J. Martin, A. Lines, R. Manohar, M. Nystrom, P. Penzes, R. Southworth, U. Cummings, Tak Kwan Lee, "The Design of an Asynchronous MIPS R3000 Microprocessor", *Proceedings of the 17th Conference on Advanced Research in VLSI*, pp. 164-181.

[22] J. D. Garside, S. B. Furber, and S. B. Chung, "AMULET3 Revealed", Proc. Async. '99, Barcelona, April 1999, pp. 51-59.

[23] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, G. Stegmann, "An Asynchronous Low-Power 8C51 Microcontroller", Fourth Intl. Symp. on *Symp. on Advanced Research in Asynchronous Circuits and Systems (Async 1998)*, San Diego, California, March 1998, pp. 96-107.

[24] "Nios 2.1 CPU Datasheet", Altera Corporation, April 2002, Version 1.1. http://www.altera.com/literature/ds/ds_nioscpu.pdf.