A Coarse-Grain Phased Logic CPU

Abstract: A five-stage pipelined CPU based on the MIPs ISA is mapped to a self-timed implementation scheme known as Phased Logic (PL). The mapping is performed automatically from a netlist of D-Flip-Flops (DFFs) and 4-input Lookup Tables (LUT4s) to a netlist of PL blocks. Each PL block is composed of control logic wrapped around a collection of DFFs and LUT4s to form a multi-input/output PL gate. PL offers a speedup technique known as early evaluation that can be used to boost performance at the cost of additional logic within each block. In addition to early evaluation, this implementation uses bypass paths in the ALU for shift and logical instructions and buffering stages for increased dataflow to further improve performance. Additional speedup is gained by reordering instructions to provide more opportunity for early evaluation. Simulation results show an average speedup of 41% compared to the clocked netlist over a suite of five benchmark programs.

Keywords: *automatic synthesis, self-timed, asynchronous, pipelined processor, micro pipelines*

1. Introduction

Design challenges related to global clocking are identified throughout the ITRS-2001 Roadmap on Design. It is clear that the engineering effort dedicated solely to clock distribution and clock management issues [1] keeps growing as feature sizes shrink, with no indication of how to halt this trend if traditional clocked methodologies continue to be used.

This paper discusses a self-timed design methodology known as *Phased Logic* (PL) that eliminates the need for a global clock and allows automated mapping from a clocked netlist representation to a self-timed netlist. Previous work on Phased Logic has concerned fine-grain mappings in which gates in the clocked netlist were mapped on a one-to-one basis to PL gates in the self-timed netlist [2]. A natural implementation technology for these fine-grain PL systems would be a new SRAM-based FPGA customized for PL.

By contrast, this paper demonstrates a coarse-grain mapping scheme suitable for ASIC designs in which blocks of logic in the clocked netlist are encapsulated into PL partitions. The logic blocks can be any collection of D-Flip-Flops (DFFs) plus combinational logic or monolithic combinational compute functions such as memories, multipliers, etc. Because the mapping works from the netlist level, design reuse is preserved at the RTL level. Design reuse of hard macros at the physical level such as SRAMs is easier since the PL control scheme involves placing wrapper logic around the compute function with no required modifications to the compute function internals.

2. Phased Logic

Micropipelining [3] is a self-timed methodology that uses bundled data signaling and Muller C-elements [4] for controlling data movement between pipeline stages. Level-Encoded Dual-Rail (LEDR) signaling was introduced in [5] as a method for providing delay insensitive signaling for micropipelines. The term *phase* is used in [5] to distinguish successive computation cycles in the LEDR micropipeline, with the data undergoing successive *even* and *odd* phase changes. The systems demonstrated in [3][5] 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 [6] via a methodology termed *Phased Logic*

(PL), which uses marked graph theory [7] 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. This mapping algorithm performed a one-to-one mapping of gates in the clocked netlist to 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. The computation performed by the gate is the Boolean function of the original gate in the clocked netlist. 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 both the gate phase and output phase to toggle. The phase of the output can either always match the gate phase or always be the opposite of the gate phase. A PL gate can also have multiple outputs with both variations of output phase. Figure 1 illustrates a PL gate firing with an output whose phase is always opposite the gate phase.

The algorithm for mapping a clocked netlist to a PL netlist was developed in [6] and is summarized below:

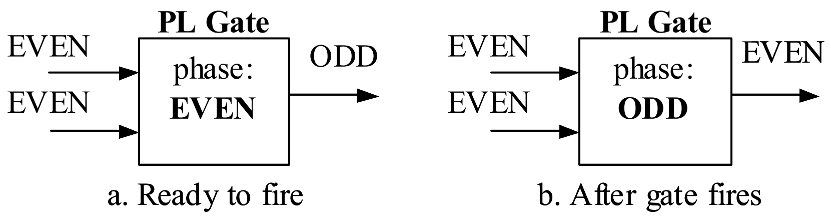


Figure 1. PL gate firing

a. All DFFs are mapped one-to-one to *barrier* gates in the PL netlist. The output phase of a barrier gate 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. The output phase of a through gate is always opposite the gate phase.

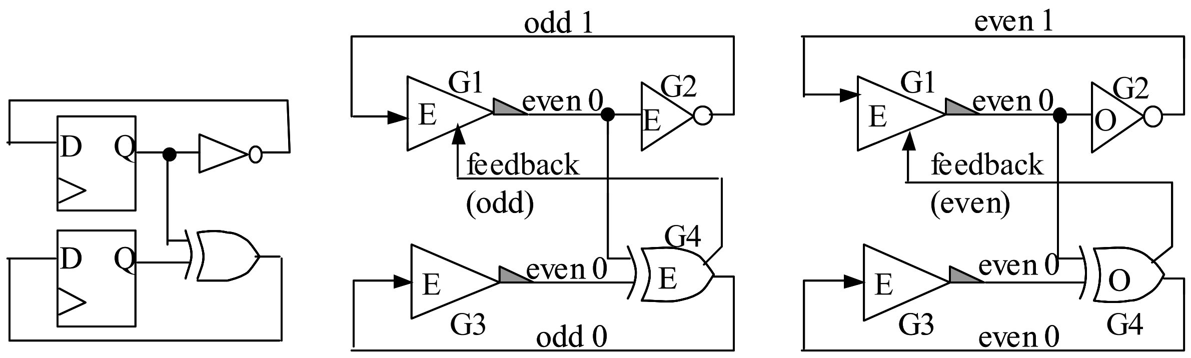
b. Single rail signals called feedbacks are added where necessary to ensure *liveness* and *safety* of the resulting marked graph. *Liveness* means that every signal is part of a loop that has at least one gate ready to fire. *Safety* means that a gate cannot fire again until its output data has been consumed by all destination gates. To ensure safety, all signals must be part of a loop that contains at most one active token. Feedbacks cannot be added between two barrier gates because this would result in a loop with two active tokens, violating the safety constraint. If necessary, buffer-function through gates (called *splitter* gates in [6]) are inserted between barrier gates to provide a source and termination for feedback. From a terminology viewpoint, the term feedback was first used in [6] and is equivalent to an acknowledge signal in a micropipeline. The term feedback will used in this paper to be consistent with the work originally presented in [6].

c. 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.

d. 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.

e. 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 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 criteria are already covered and do not require feedbacks. It is possible for a single feedback signal to create a loop that covers multiple signals.



a. Clocked 2-bit Counter b. Translated PL Counter (reset state) c. PL Counter after G2,G4 fire

Figure 2. Translation and firing of a 2-bit counter

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 added to ensure safety.

2.1 A Coarse-grain Phased Logic block

In [10], a PL gate that used a 4-input Lookup Table (LUT4) with four LEDR-encoded inputs was presented as the basic cell for a proposed FPGA intended for self-timed implementations. Delay-insensitive signaling can be very useful in a programmable logic implementation because of the uncertainty of wire delays through multiple programmable switch elements. However, in ASIC implementations a bundled-data approach can be used because of the ability to match the wire delay of the control wire with its associated data bundle.

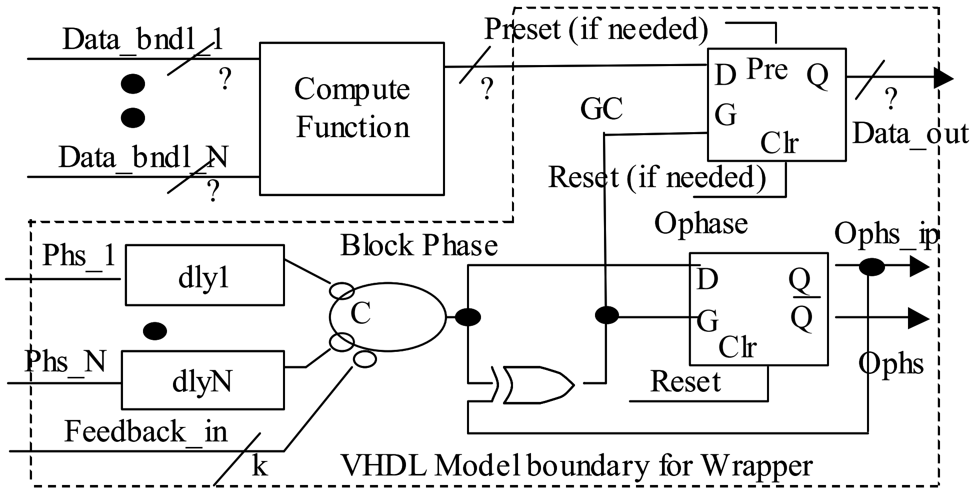


Figure 3. Coarse-grain PL block

Figure 3 shows a coarse grain PL block that is a variation of Sutherland's micropipeline structure [3]. Each signal in a data-bundle is single rail, and each data bundle has a corresponding single-rail phase signal. A change on the phase input signal indicates a new data bundle is available. The mapping from a clocked system to a coarse-grained PL system occurs at the module level, rather than the gate level. The compute function can receive multiple data bundles, and the delay block associated with each phase control wire is the longest delay through the compute block for the associated data bundle. The number of feedbacks terminating on this block is variable depending upon the circuit structure. The firing of the C-element [4] indicates that all inputs have arrived, which causes the GC signal to be asserted, latching new output data. The GC signal is negated when the *block phase* and *ophs_ip* signals match. The *ophs_ip* signal is used as the phase output for barrier-blocks and the *ophs* signal is used for through-blocks. These same signals also function as the feedback output signals. At reset, all block phases are reset to a '0' value which corresponds to a block phase of *even*. The *preset* and *clr* inputs on the data output latches are only used for barrier-blocks and are used to set initial data values corresponding to the initial DFF values in the original clocked netlist.

2.2 Early Evaluation

The block design in Figure 3 can be extended to allow firing upon the arrival of only a subset of inputs. This was first demonstrated on a limited basis in [5] via a two-input AND gate design with LEDR inputs and termed *eager evaluation*. Safety considerations and a general application of this technique were not discussed. This technique was generalized for PL systems in [9] and is referred to as *early evaluation*. This technique can be used to increase the performance of a PL system [8]. As an example, early evaluation can be done within a binary full adder, because the sum output can be determined without waiting for arrival of a carry-in value if the two data operands produce either a kill or generate value. Figure 4 shows the block design of Figure 3 modified to include early evaluation capability.

The PL wrapper control design now has two internal phases, a *trigger* phase and a *master* phase. The compute section is likewise augmented with a logic section that will be used to produce the *EEselect* signal, which is used to select between the trigger and master phases. An early fire occurs when the trigger C-element fires and EEselect is a '1' value. This causes a firing of the outputs based upon the trigger phase value. A *delay kill* signal is generated that short circuits the input delays to the master phase as the output has already been updated and these delays are no longer needed. The delay on the output of the trigger phase C-element is the longest delay of the data bundles through the *EEval* function. If desired, individual delay blocks could be used on each input to the trigger phase C-element as is done for the master phase C-element. Note that the feedback output signals are based upon the master phase C-element. This is important, as feedback cannot be provided until all inputs have been consumed which occurs when the master phase C-element fires. The *dly* block on the output of the master phase C-element should be tuned such that the feedback signals are updated at the same time or after the output phase signals are updated

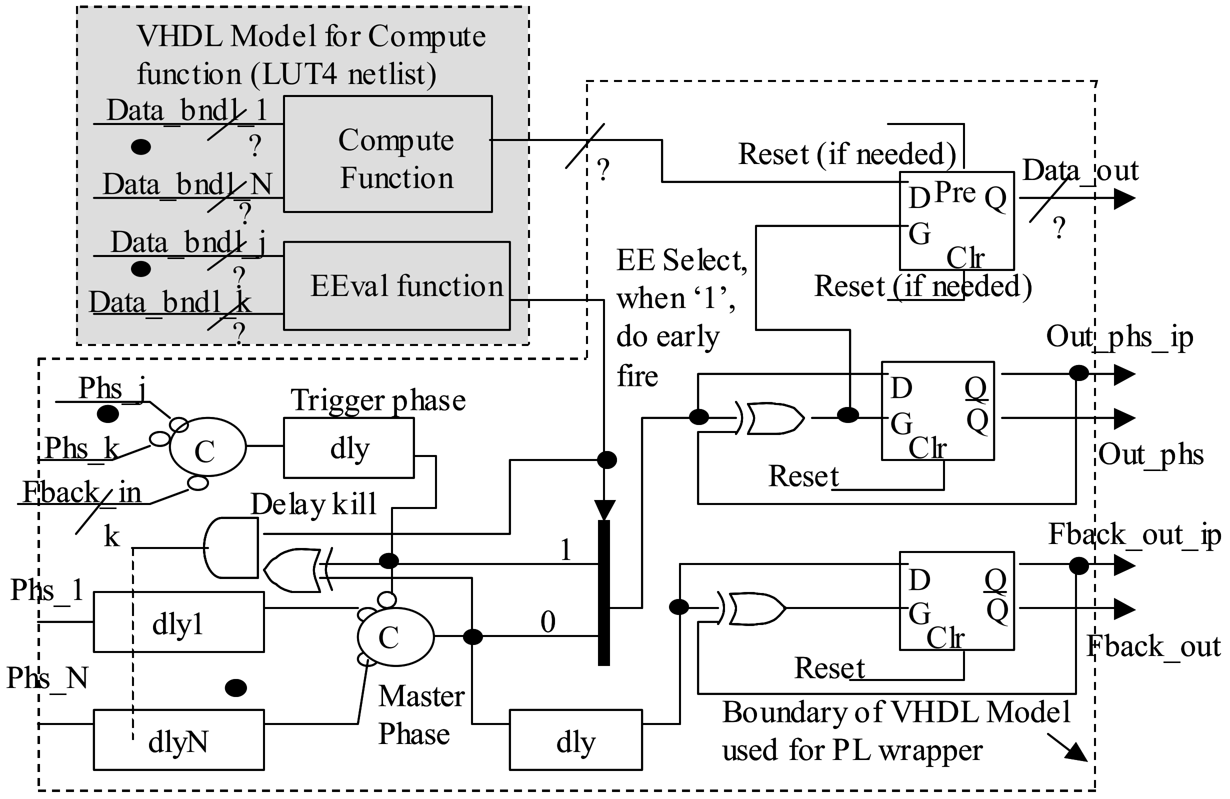


Figure 4. Coarse-grain PL block with early evaluation

. Equally important is the fact that all feedback inputs terminate on the trigger phase. This prevents a second early firing from occurring until all feedback inputs have arrived. A normal firing occurs if EEselect=0 when the trigger phase fires; the output phase will not be updated until all inputs have arrived and the master phase has fired.

A key question is whether safety and liveness are preserved in a PL system with early evaluation gates. Figure 5 and Figure 6 show a simplified two-node model for an early evaluation gate under normal fire and early fire conditions. The two nodes, M and T, correspond to the master and trigger C-elements in the early evaluation block in Figure 4. The Fi and Fo signals are the feedback input and feedback output, respectively. Figure 5 shows a normal fire, where the trigger function evaluates to *false* meaning that the output does not fire until all trigger and master inputs arrive. The master node generates the output token in the case of a normal fire and thus, the output is shown connected to the M node.

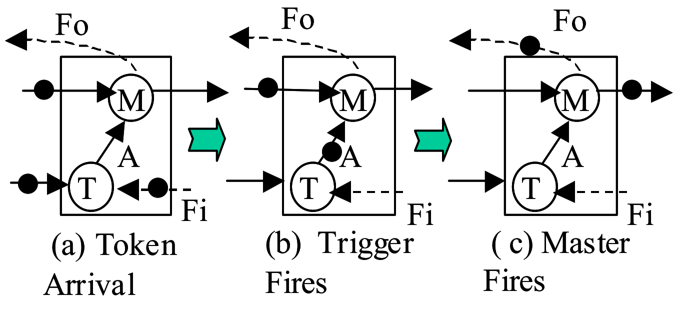


Figure 5. Early evaluation gate -- normal fire

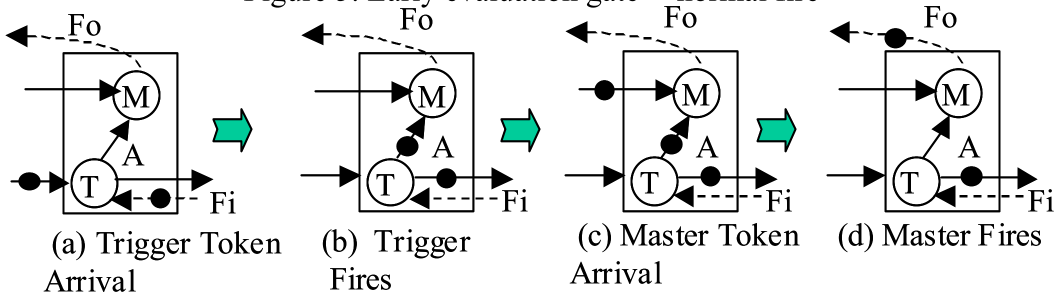


Figure 6. Early evaluation gate -- early fire

Figure 6 shows an early fire in which the trigger function evaluates to *true* causing a token to appear on the output after only the trigger inputs arrive. For an early fire the trigger node generates the output token so the output is shown connected to the T-node. For safety, signals must be covered for the firing cases shown in Figure 5 and Figure 6. However, Figure 7 shows that a normal fire topology can be transformed into an early fire topology by simply inserting a large enough delay on the early fire inputs. This can be done because marked graphs have a bounded cycle time, so there exists a delay that can be inserted on the early fire inputs that will guarantee that the T-node will not fire until the M-node inputs have arrived.

This means that if the graph is made live and safe assuming that all early evaluation nodes have the topology shown in Figure 6 (early fire), then the graph will also be live and safe if an early evaluation node has the topology of Figure 5 (normal fire) because delays do not affect the liveness and safety of a marked graph.

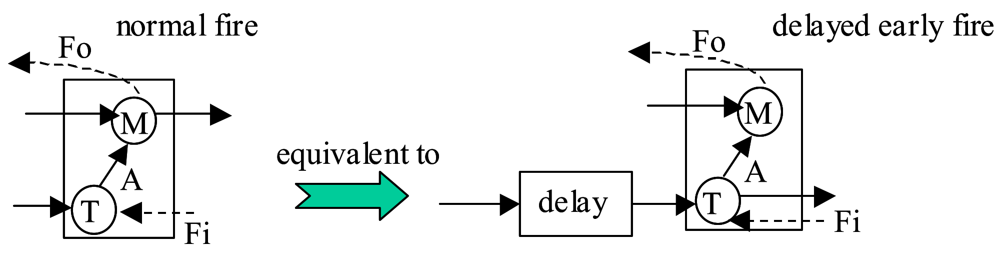


Figure 7. Normal fire is equivalent to a delayed early fire

The current mapping algorithm forces all non-feedback inputs that are connected to an early evaluation gate to be part of a loop that contains the feedback output of that gate. In addition to covering the input signals, this also covers the internal signal ‘A’ since the feedback output originates from the M-node. Also, the current mapping algorithm covers the output signal via a loop containing the feedback input signal. This means that the current mapping algorithm produces PL netlists in which early evaluation blocks are always both a source and destination of feedback. For some topologies, these rules add more feedbacks than the minimum required in exchange for reducing the complexity of feedback generation.

The initial token marking rules in the beginning of Section 2 ensure liveness by enabling at least one gate in each loop to fire after reset. These rules do not require altering in the presence of early evaluation gates, and early evaluation gates can function as either barrier or through gates.

2.3 Loop Delay Averaging

The cycle time of a PL system is bounded by the longest register-to-register delay in the original clocked netlist, although the average cycle time can be less than this value because of the averaging of loop cycle times of different lengths [6]. The circuit in Figure 8 shows a two-stage, unbalanced pipeline. The *DF* block in each circuit represents a barrier gate (a D-flip-flop in the original clocked netlist), 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.

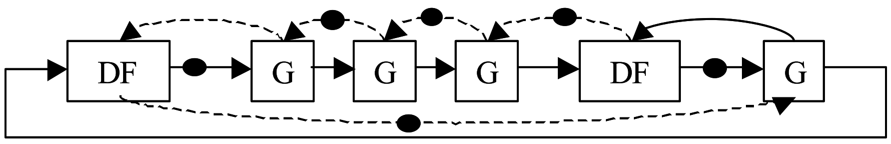


Figure 8: An unbalanced PL pipeline

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. Simple 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 token delay of 30 time units. Note that if the original clocked system had balanced pipeline stages, then the longest path would have been 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.

It should be noted that the system in Figure 8 has more than the minimum number of feedbacks and that feedback placement can also affect system performance. In general, keeping feedbacks short in terms of the number of gates between source and destination of the feedback will improve system performance if the path is part of the critical loops that determine the system's throughput. However, feedback can skip over multiple gates and not affect system performance if the path is part of a non-critical loop.

2.4 Slack Matching Buffers

A register-to-register path in a clocked circuit corresponds to a barrier gate to barrier gate path in the PL circuit. In a clocked circuit, only one computation can be in progress on the register-to-register path at any given time during a clock cycle, unless an asynchronous technique such as wave pipelining is used. However, in a PL circuit, it is possible to have more than one token in flight between barrier gates, which would correspond to more than one computation in progress between the barrier gates. This behavior is a form of loop delay averaging and can reduce the cycle time of a PL circuit. However, if there is more than one path between the barrier gates, extra buffers called slack matching buffers [14] may have to be added in order to take advantage of this loop delay averaging. Circuit A in Figure 9 shows a two-stage pipeline. Gates D1, D2 and DF are barrier gates; the remaining gates are through gates. The dashed lines are feedbacks of length 1. If each gate has a delay of 10 time units, simulation (or simple analysis) shows that each gate firing is spaced by 40 time units. However, by adding the extra buffer shown in Circuit B, the firing pattern of each gate is changed to a repeating pattern of 40,20,40,20, etc. This gives an average cycle time of 30 time units, an improvement of 10 time units. The reason for this improvement is that the extra buffer in the path from D2 to DF allows two tokens to be in flight along this path instead of one.

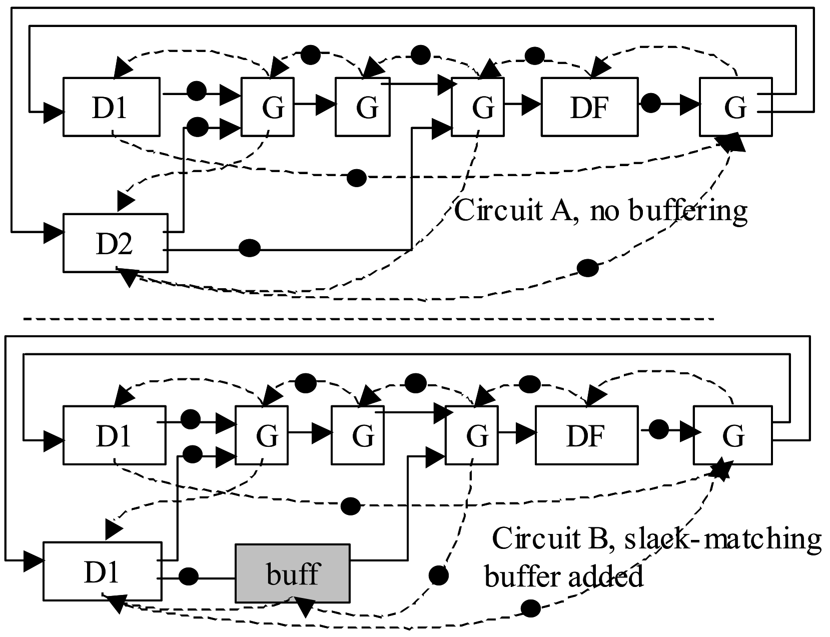


Figure 9. Adding slack matching buffer to improve performance

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 [11] 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) [12] allows the use of standard HDLs but 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. Dual-rail signaling is an inherent feature of the NCL computation style. This makes NCL wiring-delay insensitive at a 2X wiring cost. Bundled data signaling has not been demonstrated within NCL as an option that would exchange the wiring overhead of dual rail signaling for delay-matched signaling paths.

Many self-timed CPUs have been designed in the past including the MIPs integer subset [14], the ARM processor [15], and the 8051 [16]. The distinguishing features of our design are the automated mapping from the clocked netlist to a self-timed netlist and the use of early evaluation to achieve speedup over the clocked design. Previous self-timed CPUs such as the Amulet3 have used bypass paths to speed execution. The Amulet3 execution unit had an iterative multiplier and barrel shifter in series with the ALU; these two components were bypassed when instructions did not require them. Bypass operations are essentially a degenerative case of early evaluation in which all phase inputs are part of the trigger phase and the early evaluation function has a smaller delay than the normal compute function. The bypass operation is used when all signals arrive at the same time, but different delays are desired depending upon the block operation for that particular compute cycle (i.e., within an ALU, shift versus addition).

As such, the PL block in Figure 4 supports bypass operations and our design makes use of bypass in much the same way as the Amulet3. However, our design also uses early evaluation, which proves to be crucial to a significant portion of our obtained speedup.

4. A Phased Logic CPU

Our primary goal for this work was to demonstrate a PL methodology compatible with an ASIC implementation for a non-trivial design example. A CPU was chosen, as it is a well-understood example that has been used as a test case for other self-timed methodologies. A secondary goal was to demonstrate that the PL methodology could take advantage of design re-use at the RTL level. As such, we searched the WWW for freely available processors specified in RTL. Our search produced a VHDL specification of a 32-bit, MIPs CPU (integer subset) implemented as a 5-stage pipeline [13]. 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 standard fetch, decode, execute, memory and writeback stages. Because the design was intended for an FPGA, the register file RTL used positive edge-triggered devices instead of latches. For this design, the register file was altered to use level sensitive latches where the read was done first during the high phase of the clock and a write was done last, during the low phase of the clock. This is actually opposite of most register files, which are write-through, but this preserved the semantics of the read/write operations used in the original model that assumed edge-triggered devices. 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. The original RTL had a tri-state data bus interface to memory. This was changed to use dedicated input/output data busses as we have not yet investigated tri-state interfaces for busses in PL. The RTL operators for addition/subtraction in the ALU, for branch computation, and for the PC+4 increment were replaced with Synopsys DesignWare components that were optimized for LUT4s. This was done to produce more efficient netlist implementations for these operations; a carry-lookahead structure was used for all addition operations.

4.1 Mapping to a PL Netlist

The methodology we used for previous fine grain designs had to be substantially altered to accommodate the new goal of using coarse-grained compute functions with multiple inputs/outputs and that could contain a mixture of DFFs + combinational logic. The term *partition* will be used to refer to a block of logic that is encapsulated by PL control logic. Initially it was envisioned that partitions could be automatically created from a flattened netlist. However, it quickly became apparent that automated partitioning involves complex issues that will have to be addressed in future efforts. Instead, our coarse-grain methodology requires the designer to partition the logic at the VHDL level such that an EDIF netlist is synthesized with two levels of hierarchy – the instances at the top level and a lower gate level. Each instance can contain any number of DFFs plus combinational logic in the form of LUT4s. A LUT4 is used as the basic element for combinational logic because it offers a method of comparison with our fine grain mapping efforts. In a physical implementation, the LUT4 netlists would be mapped to a standard cell library. In our tool flow (Figure 10), we use Synopsys as the logic synthesis tool that produces the hierarchical DFF+LUT4 EDIF netlist from a VHDL description that has one level of hierarchy, with the leaf instances in the VHDL description containing RTL.

The process that converts the EDIF netlist to a self-timed netlist involves two stages: partitioning and mapping. The partitioning tool examines each of the top-level instances to see if they contain only combinational logic, or a mixture of DFFs and combinational logic. If the latter, then the partitioner splits this instance into two partitions; one that contains only combinational logic, and one that will contain the DFFs plus a limited amount of optional combinational logic. A partition with DFFs will become a barrier-block in the PL netlist; a partition with only combinational logic will become a through-block.

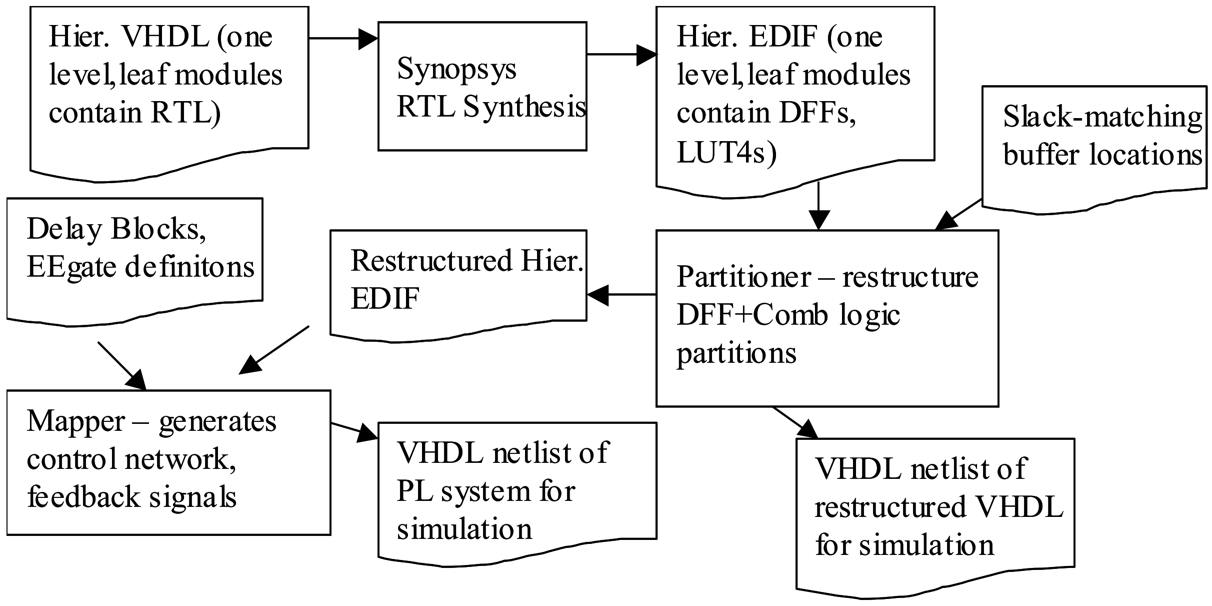

Hier. VHDL (one level, leaf modules contain RTL)
Synopsys RTL Synthesis
Hier. EDIF (one level, leaf modules contain DFFs, LUT4s)
Slack-matching buffer locations
Delay Blocks, EEgate definitons
Restructured Hier. EDIF
Partitioner – restructure DFF+Comb logic partitions
Mapper – generates control network, feedback signals
VHDL netlist of PL system for simulation
VHDL netlist of restructured VHDL for simulation

Figure 10. Clocked to PL Netlist Transformation

Figure 11 shows how an instance with DFF and combinational logic is restructured into two partitions that form a through-block and a barrier-block in the PL netlist. This example shows several important aspects of this partitioning process:

a. Inputs to the instance that go directly to a DFF must first pass through the combinational partition. This is required so that a barrier-block output does not drive a barrier-block input in the PL netlist. This would require feedback between two barrier-blocks, which violates the feedback generation rules discussed in Section 2. The barrier-block partition can only receive inputs from the through-block partition.

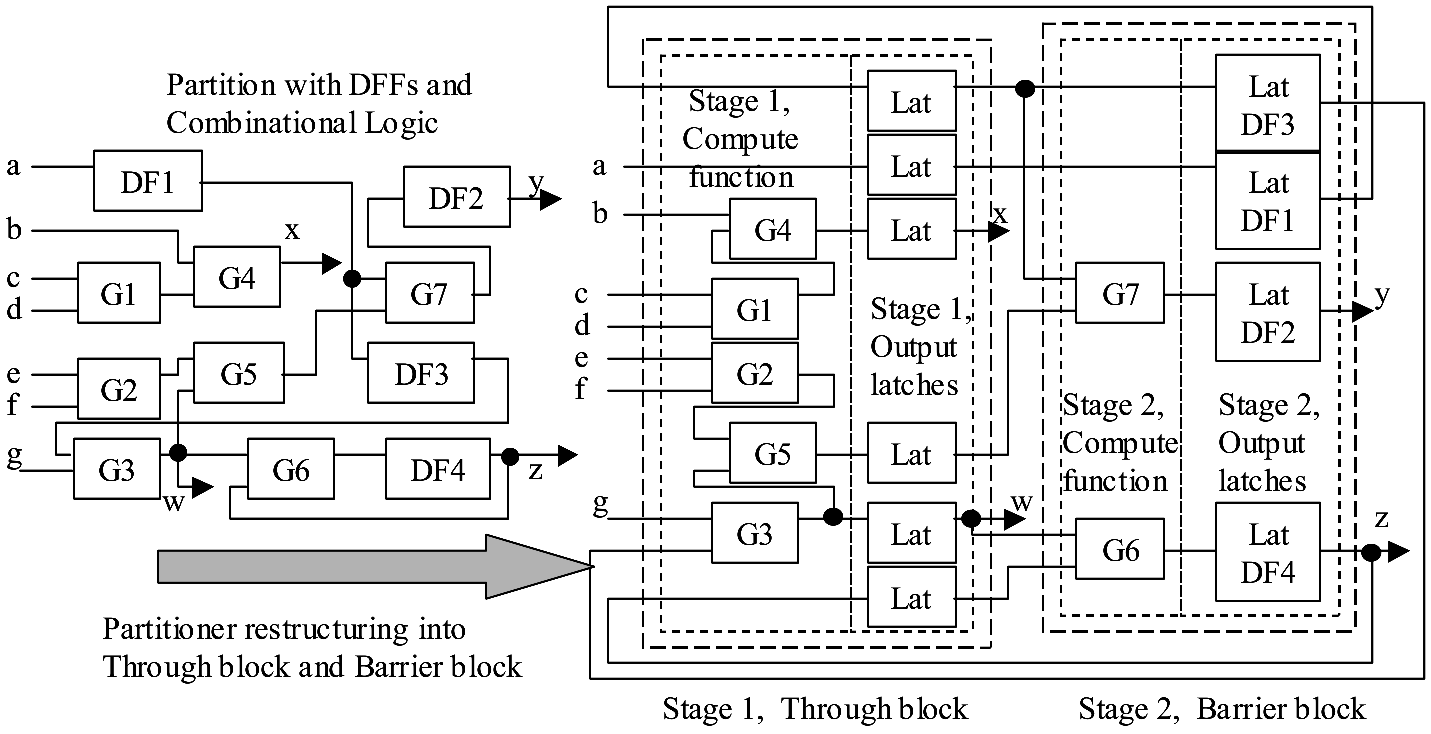


Figure 11. Instance restructuring into Through, Barrier blocks

b. Similarly, if a DFF output goes directly to a DFF input, it must be rerouted via the through-block partition.

c. Combinational logic is allowed within the barrier-block partition as long as the barrier-block partition still only receives inputs from the through-block partition, and the combinational paths terminate on DFFs. The current version of the partitioner only pushes one level of logic into the barrier-block. This was found to be beneficial as this last level of logic before the DFFs usually serves as the conditional load for a register. Note that all latches are in the PL wrapper logic placed around the compute blocks.

The mapping tool reads the EDIF netlist produced by the partitioner and treats each instance as either a barrier-block or through-block as indicated by the partitioner. A separate control netlist is generated in which each instance in the datapath netlist has a corresponding control instance created for it. If a datapath instance receives an input from another datapath instance, this is considered a data bundle and a control wire is created for it.

After the control netlist is generated, the mapping tool adds feedback nets to ensure liveness and safety of the control network. Figure 4 shows the boundaries of the VHDL models of the compute function and the PL control wrapper. Finally, a VHDL netlist of the PL system that contains both compute block instances and PL control instances is created for simulation purposes.

4.2 Control for the PL CPU

Figure 12 shows the blocks present in the PL CPU netlist. Each net connection indicates both a data bundle and its associated phase wire. We make no claims as to the optimality of this partitioning; this particular partitioning was arrived at only after many design iterations through the PL mapping process and subsequent CPU simulations. Important aspects of this partitioning are:

4.2.1 Barrier-blocks. The blocks marked as ‘BB’ are barrier-block partitions produced by the partitioner. The ifetch, idecode, execute and memstages were all specified as one instance going into the partitioner but were split into two partitions because they contained both combinational logic and DFFs.

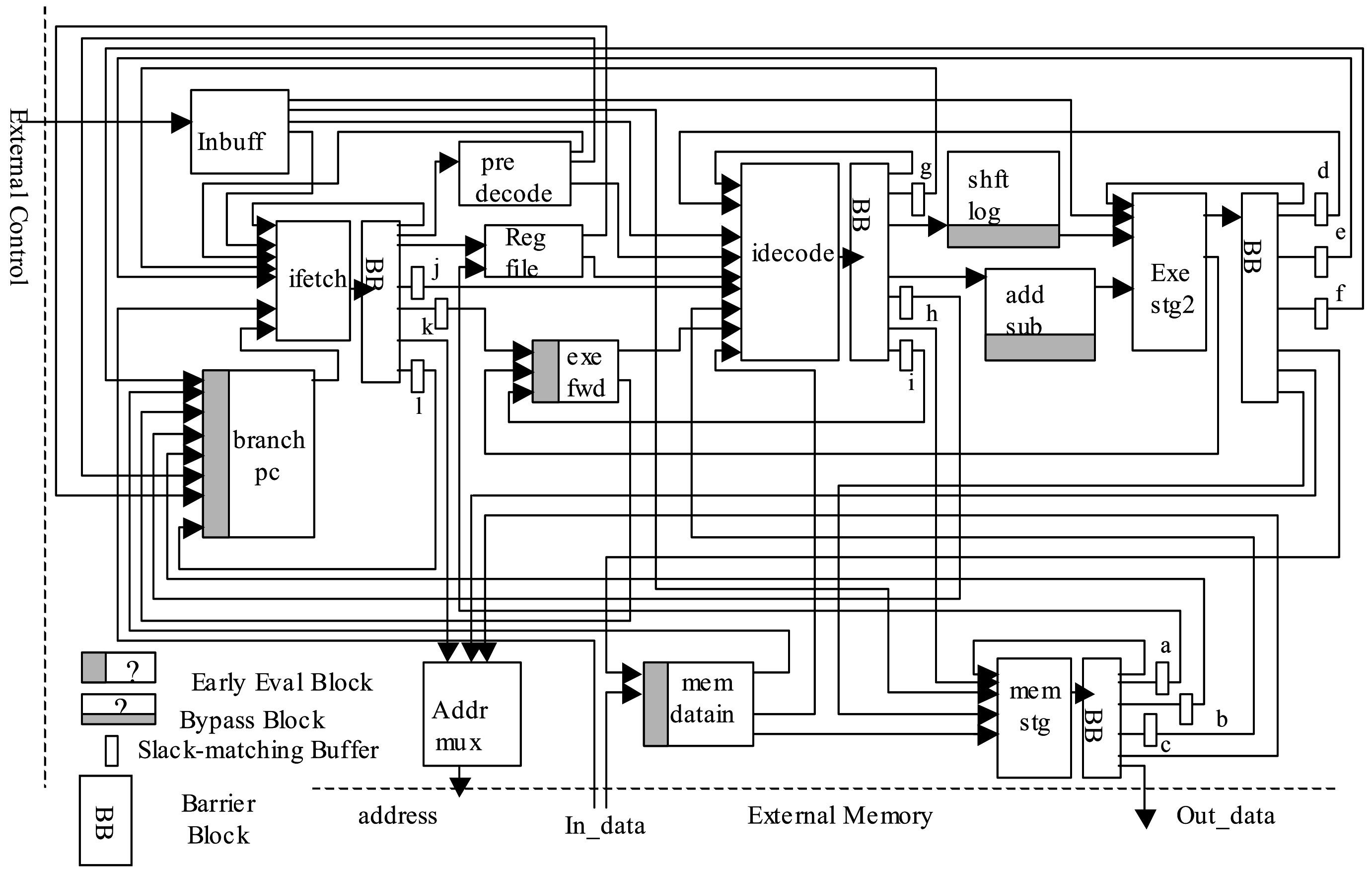


Figure 12. Control network for PL CPU

4.2.2 Slack-matching Buffers. The blocks marked as *slack-matching buffers* have empty compute functions; they only contain the PL wrapper logic. These were added when it became apparent that feedback paths were limiting performance in some loops. Time spent waiting for a feedback signal to arrive is dead time; if a gate is waiting on feedback then it cannot fire its output and send feedback to its fanin components and begin the next computation. Adding a buffer on the output allows the buffer to consume the output, freeing the gate to fire and begin the next computation when new inputs arrive. Of course, a slack-matching buffer adds forward latency to the path, so the extra latency must be offset by the time gained from not waiting on feedback. These buffers are placed in the netlist by the partitioner; an external data file read by the partitioner specifies buffer locations. This frees the designer from having to pollute the input VHDL netlist with buffers

4.2.3 Early evaluation gates. Early evaluation was used in three places as seen from Figure 12. The exefwd block in the ALU forwarding path to the idecode and branchpc gates fires early if the ALU result does not have to be forwarded to these blocks. This early fire decision is based upon values from the ifetch and idecode barrier-blocks whose values are immediately available. As an example, in the code stream below, there is no ALU forwarding needed from the first add instruction to the second add instruction, so the exefwd gate fires early. This causes the idecode gate to fire faster because it does not have to wait for the ALU result to become ready.

```
Add    r5, r6,  r9
Add    r4, r8, r10
```

As a counterexample, the code stream below does require ALU forwarding from the first add to the second add instruction due to register value r5 being the destination in the first addition and an operand source in the second addition.

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

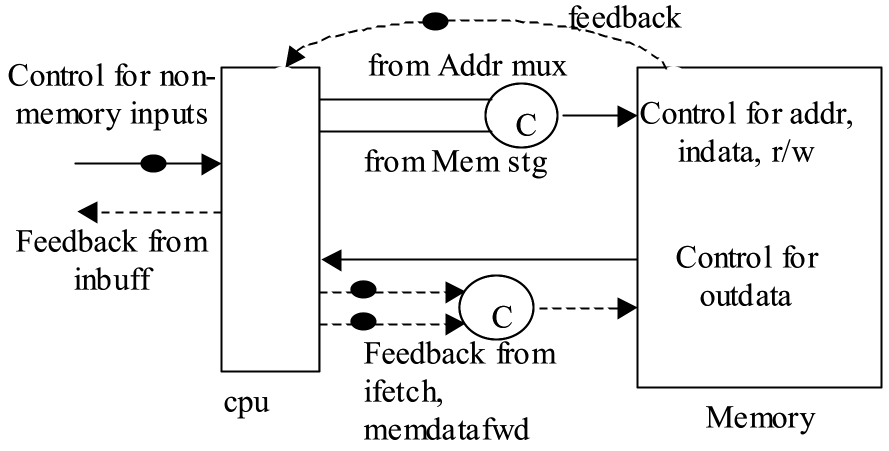
The branchpc gate is responsible for computing the next PC address for all branch and jump instructions. The branchpc gate fires early if a new PC value does not have to be computed. This early fire decision is based only upon the data value from the predecode gate which is available a short delay after the ifetch barrier-block fires. Note that the branchpc gate has as one of its inputs the output of another early evaluation gate, the exefwd block.

The ifetch block contains the incrementer required for computing PC+4 if the instruction is not a branch. This value can be produced faster if the branchpc block fires early supplying its control input to the ifetch block sooner. The mem_datain block fires early if the instruction is not a load word (*lw*) instruction. This frees the branchpc, idecode, and memstage blocks from having to wait on the external memory interface to fire. This early fire decision is based upon a pipeline register value from the execute barrier-block.

In general, the early evaluation stages allow blocks to fire in parallel where they would otherwise fire sequentially. For example, when the exefwd block fires early, the idecode and branchpc blocks can compute in parallel with the execute stage instead of having to wait for the execute stage value to be ready.

4.2.4 Bypass paths. The ALU was split into three partitions. Two partitions were combinational-only blocks; the shft_log block contained the logic for the shift and bit-wise logical instructions, the addsub block contained the add/subtract logic. The third partition contained the mux for the outputs of the first two blocks as well as all of the registers contained in the execute stage. The addsub block was bypassed if either a logical or shift instruction was being executed. The shft_log block was bypassed if a logical or add/subtract instruction was executed. The bypass delay in the shft_log block is long enough to produce the result of a logical instruction.

4.2.5 Feedback nets. For figure clarity, feedback nets are not shown in Figure 12. Most fanouts in the CPU have associated feedback nets. Even though feedbacks can skip back over multiple through-blocks as discussed in Section 2, the length of a feedback net was kept to one block level to ensure that feedback signals had the earliest possible arrival time. Control signals that are safe because they are already part of a loop that involves their destination barrier-block did not require feedbacks. As an example, all barrier-blocks had control signals that looped back to their corresponding through-block partition so these signals are safe and do not need feedback. Another example of a loop that results in safe signals is the loop formed by the ifetch barrier-block to the predecode through-block to the ifetch through-block back to the ifetch barrier-block.


feedback
Control for non-memory inputs
from Addr mux
C
Control for addr, indata, r/w
from Mem stg
Feedback from inbuff
Control for outdata
C
Feedback from ifetch, memdatafwd
cpu
Memory
Figure 13. PL CPU interface to external memory, control

4.2.5 Memory Interface. Figure 13 shows how an external memory and control inputs were connected to the PL CPU. A PL wrapper was placed around a VHDL model for an external memory with one control line used for all data inputs and one control line for data outputs. The memory model had separate datain/dataout busses. Memory was treated as a through-block in terms of mapping as seen from the initial token marking in

Figure 13. The only external control signals present in the original MIPS VHDL model that were not directly connected to the memory model were *rdy* and *init*. The *rdy* signal was an external stall line for the processor, while the *init* signal functioned as a synchronous reset. The *inbuff* block shown in Figure 12 acted as a slack-matching buffer between the external control line and the destination blocks for these signals.

4.3 PL Netlist Statitistics

Table 1 gives the netlist statistics for the PL design (the 12 slack matching buffers in the design are not listed). The *max_dly* column gives the maximum delay of the compute function in LUT4 delays. This does not include the output delay of the latch element in the PL control wrapper or the input delay of the C-element. Note that the slack matching buffers have empty compute functions so their compute function delay is zero; the latency of these elements will be determined by the PL wrapper logic.

The *EE_dly* column gives the delay of the early evaluation or bypass function if applicable. The *max_dly* and *EE_dly* values of the *exefwd, memdatain* blocks are equal because these elements were inserted in the netlist solely to provide an early phase input to their destination blocks; their normal compute function is simply a buffer function.

The output column for the compute function is important as it indicates the loading on the G signal that is used to gate the output latches for that block. Besides the global reset signal, these signals will be the most heavily loaded control signals in the system. The control columns give the number of phase and feedback inputs to each block. For early evaluation blocks, the number of phase inputs required for the trigger phase is indicated in parenthesis. The memstg, idecode and ifetch blocks do not require feedback inputs because their outputs are already safe due to an existing loop involving their destination barrier-block.

Table 1. Netlist Statistics for the PL CPU.

	Compute Function					PL Control	
	max dly	EE dly	LUT4	ips	ops	ips	FBs
memstg_bg	1		72	146	71	1	5
exe_bg	1		42	85	41	1	6
ifetch_bg	1		97	196	96	1	5
idecode_bg	1		160	308	159	1	6
memstg	3		186	178	144	5	0
shft_log	9	5	171	140	33	1(1)	1
add_sub	14	3	166	149	50	1(1)	1
exestg2	2		126	125	83	4	1
ifetch	11		244	175	194	7	0
memdatain	1	1	1	33	32	2(1)	3
regfile	7		2489	72	64	2	2
addrmux	1		33	69	34	3	1
idecode	10		652	442	306	9	0
branchpc	12	1	360	287	33	8(1)	1
inbuff	0		0	2	2	1	4
exefwd	6	6	17	48	32	3(2)	2
predecode	5		49	32	19	1	2

4.4 PL CPU Performance

The performance of the PL CPU was measured via simulation of the VHDL netlist produced by the mapper program. Delays were normalized to LUT4 delays. Five benchmark programs were used for performance measurement: fibonnaci (fib), a value of 7 was used; bubblesort, a matrix size of 20 was used; crc, calculate a CRC table with 256 entries; sieve – find prime numbers, stopping point set to 40; matrix transpose - a 20x30 matrix was used.

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 an input file read by the VHDL memory model. Based on previous transistor level simulations, the output latch delay of the non-EE PL wrapper control logic and a four input C-element were each set to 0.6 LUT4 delays (these delay ratios were used in previous fine-grain mapping efforts and were chosen from transistor level simulations and typical FPGA datasheet values). The input delay of the C-element for a PL wrapper block was calculated assuming that a four-input C-element tree would to be used to create C-elements with more than four inputs. The output latch delays of an early evaluation PL block were set to 1.0 LUT4 delays to account for the additional complexity of its control logic. The delay block values in PL gates were set equal to the maximum delay of the associated data bundle through the compute function minus the delay of the C-element. Slack-matching buffer delay is the sum of the C-element delay plus output latch latency. Obviously, adjusting the C-element delay to account for the loading of the G output signal would have produced a more detailed timing delay approximation. In most cases, this extra delay would simply be subtracted from the input delay blocks.

Determining a memory access delay penalty is problematic. If the performance of the system is limited by memory, the issue of performance in the CPU core becomes moot. However, at the same time, the effect of memory access time should not be ignored. As such, the benchmarks were run under two conditions; a slow memory case and a fast memory case. In both cases the memory access time was not the limiting factor in the original clocked netlist. The critical path in the clocked netlist as reported by Synopsys is 24 LUT4 delays and passes through the execute, branchpc and idecode stages. The total memory path delay of the CPU from address out to data in was 13 LUT4 delays, leaving 11 LUT4 delays available for memory access. The slow memory case assumed that the memory interface speed was fixed at the maximum allowable memory latency without limiting performance in the clocked system, or 11 LUT4 delays. The fast memory case assumed that the memory bandwidth could be increased such that it did not limit the speedup of the PL CPU.

Even though the register file model was synthesized to a netlist of latches and combinational logic, this gate level netlist was not used in the PL simulation for this compute block. The normal PL control wrapper assumes the compute function proceeds in parallel with the C-element evaluation. This does not work for the write operation of the register file that can only occur after all inputs have arrived.

As such, the firing of the C-element was used to trigger the write operation in an RTL level model of the register file. Only one delay element was used for this control wrapper and it was placed after the C-element. This means that all inputs paid the full delay penalty of the register file regardless of arrival sequence of the inputs. The delay used for the register file was the maximum delay as reported by Synopsys when synthesized to a latch implementation. This is a conservative estimate for the register file delay.

Table 2 gives the performance results for the PL CPU. The columns marked as *mem_efire*, *branch_efire*, and *exe_efire* show the percentages of early firings for those blocks out of the total instruction cycles. Intuitively, more early firings means higher performance, and these numbers support that hypothesis. The *Spdup* column shows speedup over the clocked netlist where the clocked netlist was simulated using a clock cycle time of 24 LUTs. The speedup value is calculated taking the larger of the two execution times for a benchmark and dividing by the smaller value; a negative sign is used to indicate slowdown. All of the values in Table 2 are positive, indicating that the PL CPU achieved speedup over the clocked netlist for every benchmark.

Table 2. PL CPU Performance Results

			Fast Mem		Slow Mem	
	mem efire	branch efire	exe fire	Speed up	exe efire	Speed up
fib	89%	74%	77%	1.43	73%	1.24
bubbl	85%	83%	42%	1.3	42%	1.21
crc	100%	76%	38%	1.4	31%	1.24
sieve	97%	79%	38%	1.27	37%	1.2
mtpse	92%	92%	53%	1.31	50%	1.23
avg				1.34		1.22
fib	89%	74%	77%	1.43	73%	1.24
bubbl	85%	82%	65%	1.37	58%	1.25
crc	100%	76%	76%	1.52	59%	1.28
sieve	97%	75%	64%	1.35	59%	1.22
mtpse	92%	92%	73%	1.38	70%	1.25
avg				1.41		1.25

The top half of Table 2 shows results for non-reordered instruction streams. In examining the assembly language produced by *gcc*, it became evident that simple reordering of instructions in critical loops would increase early evaluations of the *exefwd* gate, thereby improving performance. For example, a typical code segment produced by *gcc* is shown below:

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

The exefwd gate will not early fire 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 allows the exefwd gate to early fire for the *bne* instruction. Instruction reordering was done manually by examining the critical loops of the assembly code for the benchmarks. Instruction reorderings improved performance in all cases, except for the *fib* benchmark for which no instruction reorderings were found. It can be seen that the increased performance in the lower half of the table is due to the increase in early firings of the exefwd gate due to instruction reordering.

Table 3 Individual Instruction Timings

Inst. Seq.(fast mem)	Avg Cyc Time	Fire Pattern
and	13.2	13.2
and (no eefwd)	14.0	14.0
shift	13.2	12.4,14.0
Shift(no eefwd)	17.6	17.6
add	17.0	17.0
add(no eefwd)	22.6	22.6
branch,nop	16.8	20.4, 13.2
jump, nop	16.8	20.4,13.2
load	18.9	10.6,10.6, 22.6
store	17.0	17.0

The slow memory case has a lower speedup than the fast memory case because the memory path now becomes the bottleneck in the system. This is a fairly obvious result in that all aspects of system performance must be increased if maximum speedup is to be achieved. Fortunately, there are many techniques available for increasing memory bandwidth so the fast memory case can be viewed as an achievable speedup

Table 3 shows performance results for streams of individual instructions. 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 'no eefwd' versions mean that each instruction had, as a source register, the destination of the previous instruction so that the exefwd gate did not early fire. Not surprisingly, the logical and shift instructions are the fastest. The logical and shift instructions have the same timings if the exefwd gate fires early because the register file to idecode path becomes the bottleneck in this case. The bypass for the logical operations only increases performance if the exefwd gate does not early fire because this causes the ALU to become the critical path. The jump/branch streams were two-instruction streams where the jump/branch was followed by a *nop* instruction. Speedup in the PL CPU is achieved via early evaluation, bypass operation, slack matching buffering, and instruction reordering. Table 4 shows the speedup contribution for each of these components for the bubblesort benchmark. The slowdown for the PL CPU without any of the performance enhancing features is expected as the PL wrapper latches add latency to the critical paths. It is evident that adding early evaluation provided the largest performance increase.

Table 4. Speedup contributions within the PL CPU

PL CPU Version	Spdup (bubbl)
a. no ALU bypass, no eeval, no slack matching buffers, no instruction reorder	-1.14
b. ALU bypass only	-1.04
c. Version b + early eval	1.24
d. Version c + slack matching buffers	1.30
e. Version d + instruction reorder	1.37

In evaluating these speedup numbers, we offer the caveat that we are comparing PL performance against one particular clocked CPU implementation. Unfortunately, we cannot make the claim that this is the fastest possible clocked implementation of the MIPS ISA via a LUT4 technology. To make this claim, we would have needed to totally rewrite the provided MIPS RTL model in order to test out different approaches for implementing the MIPS ISA. To our credit, we did try to reduce the critical path in the clocked design as much as possible via the use of LUT4-optimized DesignWare components that were created in our fine-grain mapping efforts. Without the use of these DesignWare components, the critical path in the clocked system ballooned to 34 LUT4 delays, and the PL speedup numbers were even higher than those presented in Table 2. There is also the question of timing margins – we assumed no timing margins for either the clocked or PL implementations. Suggested timing margins [16][17] for delay matching in micropipelines range from 10% for regular/tiled layout blocks to 30% for synthesized standard cell blocks. However, delay path matching in micropipelines is equivalent to gated clock/datapath delay matching in high performance microprocessors [1] [18]. These designs regularly use margins of less than 10% of the clock period. It is clear that the speedup numbers in Table 2 depend upon the amount of engineering effort applied to the delay-matching problem as well as the technology chosen for logic implementation. As such, we feel that the important contribution of this work is not in the absolute speedup numbers, but rather in the methodology by which they were obtained.

4.5 PL CPU Design Evolution

At this point, a fair question to ask would be "What design path lead us to the final partitioning, early evaluation/bypass choices, and slack matching buffers shown in Figure 12?" We would like to claim that this partitioning is obvious, but in fact, Figure 12 is the product of many design iterations. The iterative loop for improving performance that was followed can be summarized as:

a. Simulate the processor with an instruction stream of identical instructions so that a repeating firing pattern can be identified.

b. Identify the firing sequence of all blocks starting at block *A* and leading back to block *A* firing again. This is the cycle time.

c. Determine the last input arrival that fires each block.

d. If the last input arrival is a control signal (the forward path), look for opportunities to provide this signal earlier via early evaluation or bypass.

e. If the last input arrival is a feedback signal (the return path), then determine if a token buffer can be used to remove the feedback waiting time. The firing of the barrier-blocks was found to be a good way to determine if the system performance was limited by feedback because they only had one non-feedback input, that of the associated through-block.

Not all feedback waiting time can be eliminated. If an early evaluation gate fires early then the feedback is not provided until all tardy inputs have arrived. This dead time can affect gates that are providing the non-tardy inputs to the early evaluation gate. This is actually a forward path problem in that the only way to eliminate the dead time is to improve the arrival time of the tardy inputs. Also, if a non-critical loop is waiting for feedback, then it is not necessary to place a buffer to eliminate this waiting time. The difficult problem is identifying the critical loop(s) because the interaction of multiple loops determines the cycle time of the PL system.

Once we determined this improvement methodology, we were able to make changes that improved performance rather quickly. The above improvement sequence could be helped via a performance analysis tool that would identify the critical loops(s) and determine if the limiting factor was the forward path or backward path. This is an area for future work.

Partitioning decisions are harder to quantify. Initially, partitions were created along architecturally logical lines such as execute, idecode, ifetch, etc with no thought as to how the PL system would be affected. However, during the evolution of the design, it became evident in some cases that logic could be moved around or new partitions created to provide opportunities for early evaluation or bypass operation. We make no claims as to the optimality of the partitioning or usage of early evaluation. Determination of the optimum partitioning and early evaluation functions is an area of future study.

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 the initial RTL modifications that removed the tri-state memory bus. Thanks also go to David Hemmendinger of Union College for suggestions on improving the clarity of the section on safety in PL netlists with early evaluation gates. This work was funded in part by an award from the National Science Foundation (CCR-0098272).

6. Summary

In this paper we 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 that uses bundled data signaling between multi-input/output computation blocks. The computation blocks support both bypass and early evaluation operation modes, which can be used to improve system performance. This methodology was applied to a publicly available RTL VHDL model of a 5-stage pipelined MIPs processor. The RTL was synthesized via a commercial synthesis tool to a netlist of D flip-flops and 4-input lookup tables before being mapped to a self-timed implementation. Early evaluation was used in the ALU forwarding path, the branch PC computation path, and the memory input data path to improve performance. Bypass was used for shift and logical instructions within the ALU. Buffering stages were added at key points in the architecture to remove bottlenecks due to late arriving feedback signals. Post-compiler instruction reordering was used to increase the percentage of instruction cycles that performed early firing on the ALU forwarding path. Performance results from five benchmark programs demonstrated an average speedup of 41% when compared to the original clocked implementation.

7. References

[1] F.E. Anderson, J. S Wells, E. Z. Berta, “The core clock system on the next generation Itanium1 microprocessor”, Digest of Technical Papers, ISSCC 2002, San Francisco, Vol 1., pp. 146-148.

[2] R.B. Reese, M.A. Thornton and C. Traver, “A Fine-grain Phased Logic CPU”, Proceedings of the *IEEE Computer Society Annual Symposium on VLSI*, February 2003, pp. 70-79.

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

[4] 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.

[5] 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.

[6] 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.

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

[8] R. B. Reese, M. A. Thornton, and C. Traver, “Arithmetic Logic Circuits using Self-timed Bit-Level Dataflow and Early Evaluation”, Proc. ICCCD 2001, Austin, Sept. 2001, pp. 18-23.

[9] 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 2002, pp. 255-259.

[10] C. Traver, R. B. Reese, M. A. Thornton, “Cell Designs for Self-timed FPGAs”, Proc. ASIC 2001, Sept. 2001, Washington, D.C., pp. 175-179.

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

[12] Michiel Ligthart, Karl Fant, Ross Smith, Alexander Taubin, Alex Kondratyev, "Asynchronous Design Using Commercial HDL Synthesis Tools", Proc. Async 2000, Eilat, Israel, April 2000.

[13] 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.

[14] 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.

[15] J. D. Garside, S. B. Furber, and S. B. Chung, “AMULET3 Revealed”, Proc. Async. ’99, Barcelona, April 1999, pp. 51-59.

[16] H. van Gageldonk, K. van Berkel, A. Peeters, D. Baumann, D. Gloor, G. Stegmann, “An Asynchronous Low-Power 80C51 Microcontroller”, Proc. Async ’98, San Diego, March 1998, pp. 96-107.

[17] J. Garside, Private Communication, February 2003.

[18] D. Harris, H. Naffziger, “Statisical Clock Skew Modeling with Data Delay Variations”, IEEE Transactions on VLSI, Vol 9., No 6., December 2001, pp. 888-898.