Abstract

In this article we present an architecture for self-timed processing networks. The architecture leads to simple and fast self-timed pipelined processors. We also develop a network analogy that is apt for self-timed networks. [1]

High Performance Self-timed Processing Networks

Willem Mallon
Portland State University
w.c.mallon@gmail.com

Klaus Ullly
Ukay Consultancy
klaus@ukay.eu

1 Introduction

Various groups have built pipelined self-timed general purpose processors over the years[8, 2, 3, 5]. The performance of most, if not all, self-timed processors does not compare favorably with their clocked counterparts. Neither does the complexity of their design, and the complexity of the control-paths in those designs.

In this paper we present an architecture that is self-timed, but is closer to the clocked designs. It can take the best of both worlds. The key differentiator with previous architecture is that we use the synchronization between adjacent stages to send data both forward and backward. The only other architecture that we are aware of that does this is the counterflow pipeline architecture[1], however our architecture requires signficantly less complex control, and has a higher throughput.

We use the architecture as a driver for the design tools that we are developing at the Asynchronous Research Center at Portland State University.

We present a network model that allows us simplified reasoning about pipelines. The model clearly shows that pipelines with backward paths may run slower than the slowest stage. It also shows the commonality of the solutions applied in the Amulet[4] and MiniMips[6] processor.

The pipelined processor architecture that we develop in this paper inherits useful properties from clocked and from self-timed designs. We discuss these in the conclusions section.

2 Pipelined processors

A pipelined processing network performance is usually measured in terms of its throughput, i.e. the rate at which it consumes data tokens at its inputs and the

```
R1 := R2 + R3 (* A *)
R4 := R1 + 6  (* B *)
```

Figure 1: B depends on data computed in A

```
R1 := MEM[ R2 + 4 ] (* A *)
R4 := R1 + 6    (* B *)
```

Figure 2: B depends on data fetched in A

rate at which it produces data tokens at its outputs. For a general purpose processor pipeline, the performance is the rate at which it processes instructions.

The basic idea behind the pipelined processor is that various stages are each responsible for their part of the execution of an instruction. A stage will perform its part, and then hand its instruction to the next stage, and begins performing its part for the next instruction.

The pipelined processor executes multiple instructions at the same time. The instructions are in various different stages of their execution. Processing performance of the pipeline are based on the throughput of the slowest stage.

It is important to realize that a processing network may contain data-dependent cycles. The processing of an instruction may depend on the results of a previously processed instruction. This is illustrated by the partial program of figures 1,2. The figures show instruction sequences where an instruction has a direct data-dependence on the previous instruction. In many clocked pipeline designs, and in the design that we present in the next section, the instruction sequence in (2) would cause the insertion of a `NOP` instruction, thus degrading the performance of the pipeline. In self-timed pipelines, there is no need to insert extra instructions, but the self-timed pipeline still has to wait for the completion of instruction A in order to be able to complete the execution of instruction B.

Sufficiently advanced compilation techniques may allow the rewriting of programs to avoid situations such as those of Figure 1,2. However, if one implements parallel execution slots like in VLIW or Superscalar architectures, these situations return with a vengeance[]. Furthermore, compiler technology for new processors requires time to mature.

In this note we present an architecture that is self-timed, but allows an elegant implementation of bypass network behavior. The performance of the self-timed architecture is similar to that of a clocked architecture, but various self-timed benefits apply.

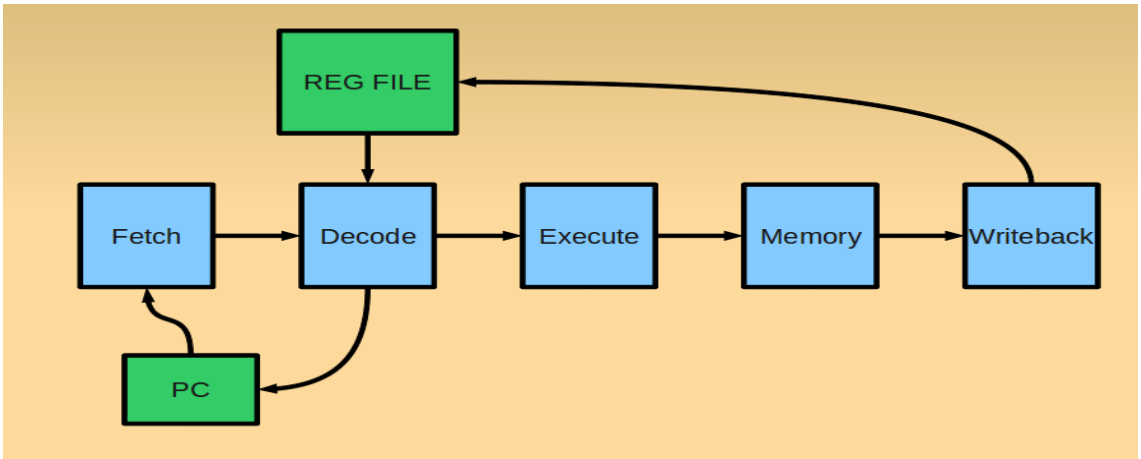

Figure 3: Simple 5-stage pipeline processor

3 Basic clocked pipeline

A basic processor pipeline looks like Figure 3. The "Fetch" stage, fetches instructions from an instruction memory. The Fetch stage than passes instructions to the "decode" stage. The Decode stage collects bit-patterns describing the instruction operation, such as the ALU-codes, and the operands from the register-file. The "execute" stages applies the operation generated in the decode stage to the data-operands. The memory stage interfaces with the RAM and reads or writes memory. Finally the "Writeback" stage writes the instruction result back into the register-file.
Usually, there is a program counter that tells the "Fetch" stage from which location to fetch the next instruction. In normal operation this program counter is incremented by one instruction unit. If the instruction is a jump, however, the new address must be fetched from the Registerfile.

The "decode" stage is the stage that detects the jump. This means that the instruction after the jump has already entered the pipeline. Some architectures disable this instructions. Others consider it a valid (delay slot) instruction.

Consider the two data-dependent instructions of figure (1). Instruction B cannot leave the decode stage until the result of instruction A has been written back to the register file. It takes 3 cycles for A to write its result into the registerfile, and hence instruction B will be delayed at least 3 cycles.

Since the result of A is available earlier, a *forwarding* network is used to write the result to the register file when they become available. The resulting architecture looks like Figure 4
In Figure 4, instruction B may still have to wait for A, depending on the implemen-

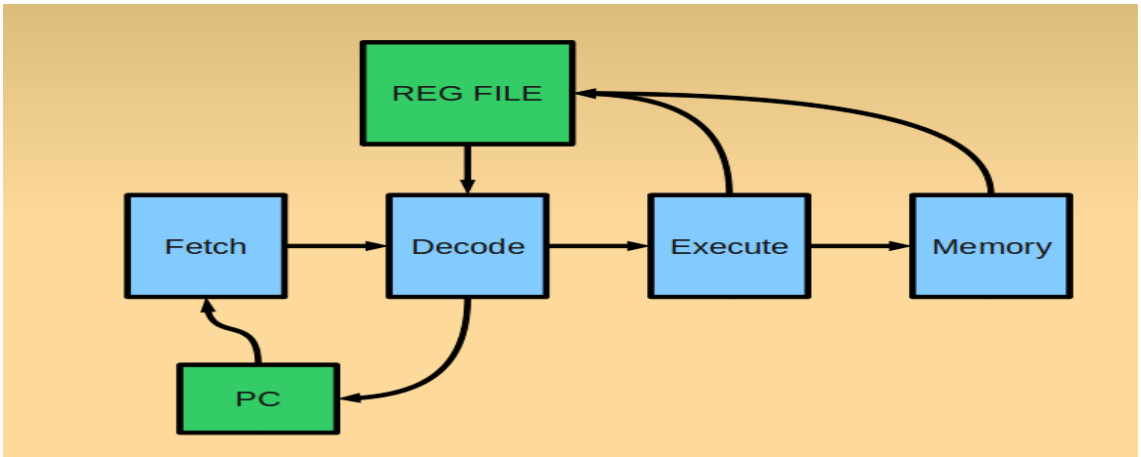

Figure 4: Pipeline processor with forwarding and bypass

tation of the register-file. The path from the result of the execution stage, through the registerfile, through the decode stage logic may be too long for one cycle.

This can solved by introducing a bypass path; a path from the execute stage to itself, so that the next instruction has direct access to the previous result. Similarly the memory stage can make its result directly available to itself and the execution stage. In such an implementation, the instruction decode stage detects that instruction B depends on the previous result, and codes the execute operations accordingly.
The resulting paths are shown in Figure 5.

4 The train model

In this section we present a train model that helps to reason about self-timed pipeline behavior. The knowledgeable reader may appreciate the analogy with self-timed pipelines as it develops. We will refrain from linking the analogy with self-timed pipeline behavior until the end of this section.

Consider stations A and B that are connected by railroad track in Figure 6. A single train is present and runs in circles.

The forward latency is the time it takes for a train to travel from A to B. The same system can be used to ship loads from B to A, the backward latency is the time it takes to ship a load from B to A. The throughput in the forward direction is the number of loads that can be shipped from A to B over a fixed time. This is the number of times the ring is traversed in that fixed amount of time.
Consider a system as in figure 6 running at full speed. We will now connect a

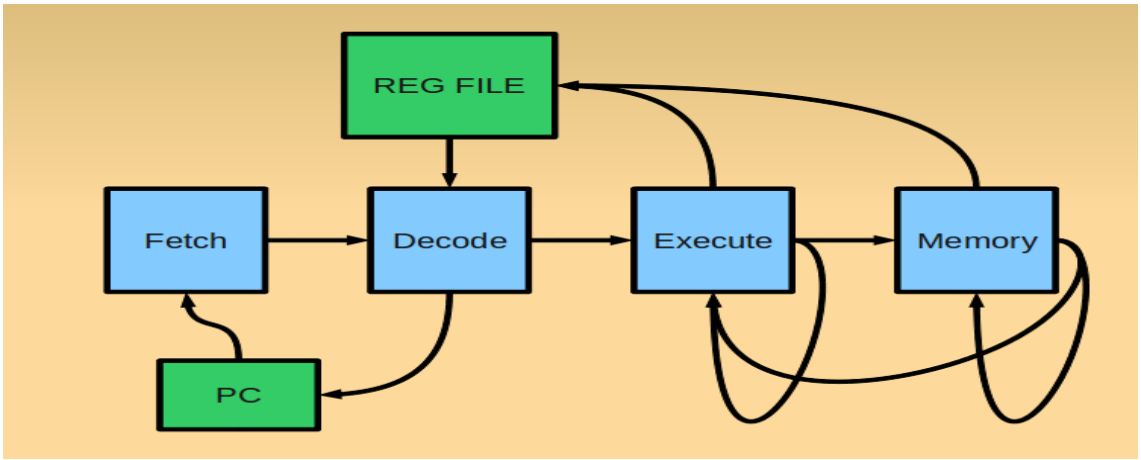

Figure 5: Pipeline processor with forwarding

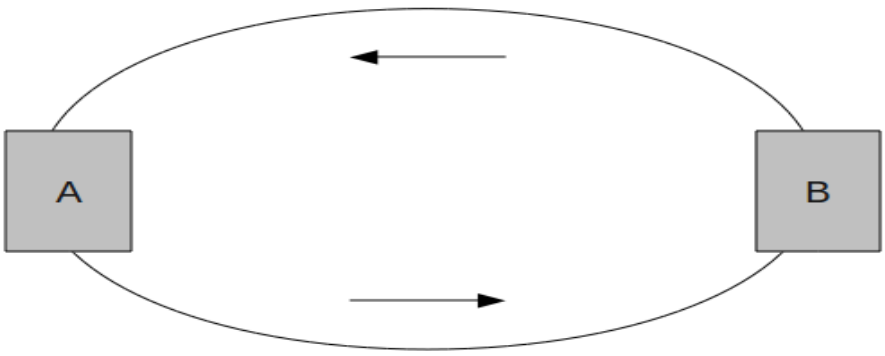

Figure 6: 2 connected train stations

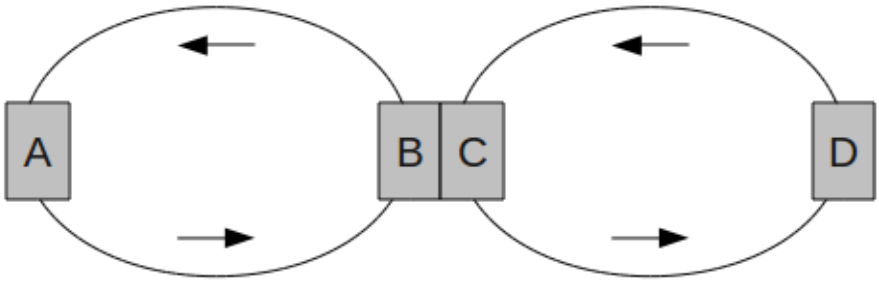

Figure 7: 2 connected rings

new circle to this system, as in Figure 7. Our interpretation is that there are two trains running in their own circles. The tracks are not connected, but the trains can exchange loads at combined station BC. In order for the trains to exchange loads, trains that arrive at the BC station must wait until the opposite trains has arrived, after which both trains will exchange loads and depart. The trains *synchronize* at station BC. A train requires a minimal time to traverse its ring once. The time for a traversal can only increase, and only if a train has to wait at a station.

4.0.1 performance

Suppose that the two circles have different throughput, and that going around CD takes longer than it takes to go around AB. Then after the first 'synchronization' at station BC, the AB train will arrive earlier than the CD train, and will have to wait. This pattern repeats. Since the AB train now always has to wait for the CD train, its throughput is reduced, and the time it takes to go around AB will be equal to the time it takes to go around CD.

There is no lower bound for the number of stations that we can connect. Nor is there a specific reason why we would use only two stations per circle. More complex networks can easily be envisioned. Consider for example the network in Figure 8.

It is the sum of the forward and backward latencies that determines the throughput. If we create a network of rings as in Figure 8, and let it run for some time, the time it will take a train in AB to go around depends on the slowest circle in the design.

There is a simple argument to show that after a finite time, all circles will have a throughput that is at most the throughput of the slowest circle.

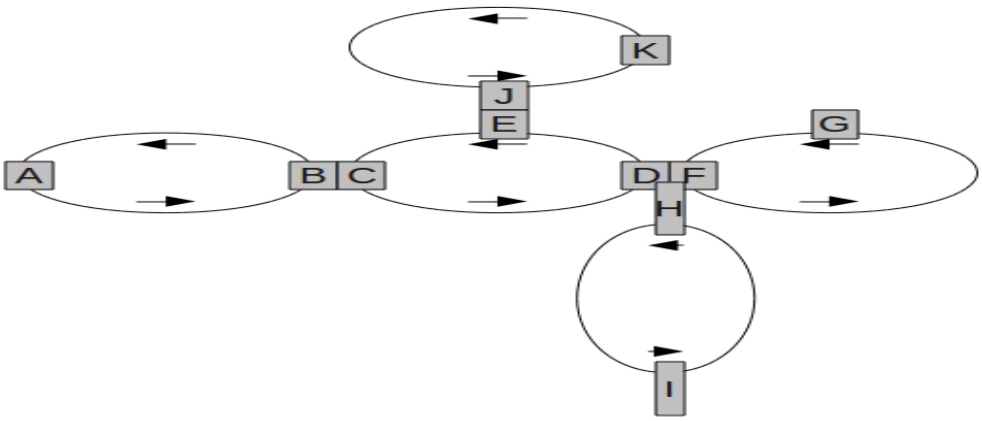

Figure 8: A network of rings

Let s and t be times at which trains in neighboring rings synchronize. The number of times that each train traversed his ring between s and t is equal. It follows that neighboring rings have equal throughput. And hence that the throughput of every ring in the system is at most that of the slowest train.

In designs such as the design in Figure 8. The design will eventually stabilize into running at the speed of the slowest stage. The argument for this is slightly more involved, and depends on structural properties of the network.

A network can be viewed as a graph. The cycles are nodes in the graph, and the stations are edges. If this graph does not contain cycles, we say that the network is an 'acyclic' network. The network in Figure 8 is an acyclic network.

In an acyclic network, all rings will eventually run at the speed of the slowest ring. We give the proof of this claim in the appendix.

The previous theorem holds for networks that contain no cycles. For networks that do contain cycles, we cannot guarantee a performance that matches the slowest ring. In fact, it is possible that these networks are slower.
We will illustrate this with an example. Consider the network of figure 9. It contains numbers that give the delay along the rings. It contains diamonds that give the initial location of trains. We invite the reader to replay the scenario on the figure with tokens. Every ring has a potential speed of 8, if the trains in the rings do not have to wait. Up to time 6, everything is moving along, but at time 6, the AB train has to wait for the KJ train that arrives 2 time units later. Note that no other train has to wait. At time 8, the AB train synchronizes with the KL train. All other trains run and synchronize without delays. At time 10, the CD train has to wait for the AB train that is running with a delay of 2. All other trains run and synchronize without

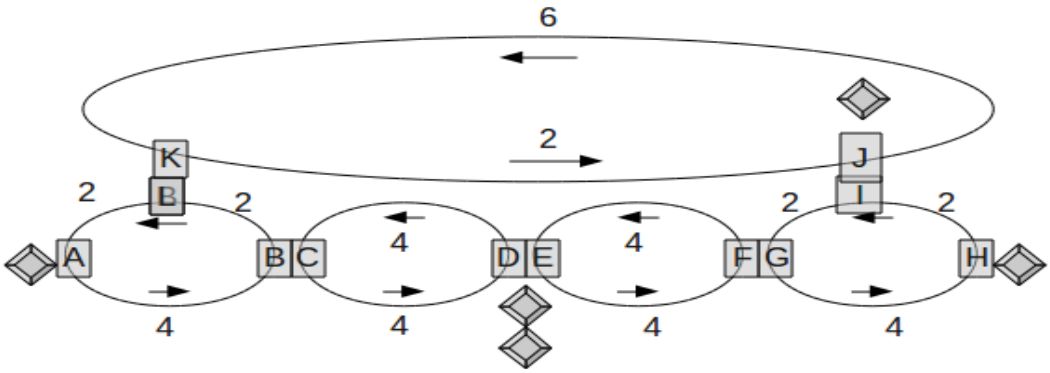

Figure 9: A cyclic network with timing annotation

delay. At time 12, the CD train departs. At time 14 the EF train is delayed. This 'delay bubble' propagates through the system until it arrives at station K, where it wil delay the AB train and bring us around full circle. The delay bubble is not absorbed, and the throughput of the stages is less than that of the slowest stage.

If one of the stages has a cycle time of less than 8, and can make up for some of the delay of the delay bubble, the delay bubble may be dissolved. For this, the stage must be able to make up the delay in the same direction as the delay bubble is traveling. By creating a ring that is so fast that its train will have to wait at all its stations during normal operation we can create a situation where delay bubbles are guaranteed to be dissolved.

One way of creating such a ring, is by running two trains on the ring. In a two-train ring, the cycle time is effectively halved, and delay bubbles in either direction can easily be dissolved.

4.1 data dependency

We now consider the network in 10. Suppose that there is a stream of passengers $P_0, P_1, ...$ that board trains at station A. The first passenger will board the first train that arrives, the second passenger the second train, and so on. These passengers can collect information (e.g. pictures) on the routes that they traveled, and send that information back with the backward trains in the network. We denote by $I^i_{C,D}$ the information that passenger P_i collected on the track from C to D. When passenger P_i arrives at station D he can send the information back. When the information $I^i_{C,D}$ arrives at station BC, it may be sent further to station A.

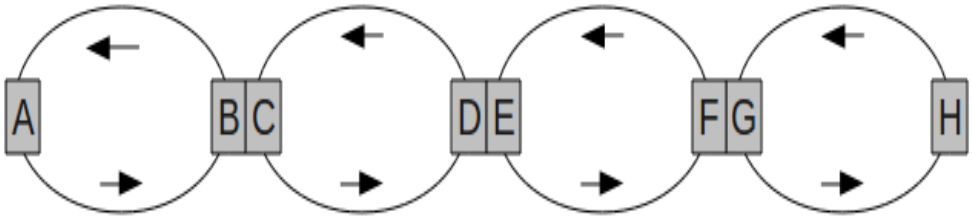

Figure 10: A network that connects H to A

We claim: When the train arrives at station A that will be boarded by passenger P_i, it contains the informations $I_{A,B}^{i-1}$, $I_{C,D}^{i-2}$, $I_{E,F}^{i-3}$ and $I_{G,H}^{i-4}$. This easily generalizes, but we will not do that here.

Suppose that passenger P_i wants to have information packet $I_{G,H}^{i-1}$ before boarding the train. This information is not available on the returning train. If passenger P_i skips 3 trains (wait 3 cycles), the information will be available.

If skipping 3 trains is not acceptable, for instance because this requirement is a common occurrence, other solutions are available.
One such a solution is a new ring connecting the AB and GH rings (Figure 11).
As we saw earlier, cyclic networks require extra care, but assuming that care can be given, this solution may make it possible to skip fewer trains. It is likely that some trains have to be skipped anyway, since it may take more than 1 cycle for passenger P_{i-1} to travel all the way to H.

Rather than skipping trains, we can also introduce a different synchronization style. We could let the AB train remain at station L until the right data arrives. This would save the time that the KL train would otherwise wait at station L, but it introduces a delay bubble. Again, if the requirement is a common occurrence, this is an interesting alternative.

The more complex bypass of Figure 11, with associated passenger protocol is an option if it is often the case that P_i waits for information that is deep in the pipeline. One may consider, however, to restructure the pipeline, such that the information is partially computed on the backward paths. Furthermore, it is not always the case that a bypass will be faster than the path of the returning trains.

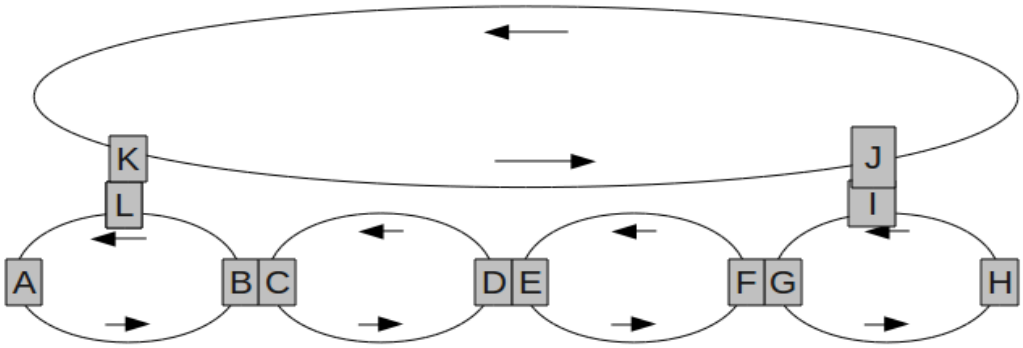

Figure 11: Pipeline with bypass

In general, let the stations be numbered $S_0, S_1, ...$ Let $M_{i,k}$ be the message sent by passenger P_i at station k. The newest messages that are available to passenger P_m at station q are: $P_{m-i,k+i}$. This assumes that passengers move forward and messages are sent backward. If passengers would utilize the backward paths as well, information can be made available more quickly.

4.2 Trains and Pipeline controllers

The train model from the previous section is an analogy for a network of self-timed pipeline controllers. The train moving forward from A to B is the signal-wave that carries data. The data are stored by stateholders at station A. The data-wave could be bundled-data or dual-rail data. The train going backwards from B to A indicates that data has arrived at B, and can be released from its state-holders. The train going backward could also hold data that is stored at station BC, but this is not common practice in many self-timed designs. The two-train system corresponds to a wig-wag buffer, or a register-reorder buffer[4], or a dual bus write-back[7].

The structure of Figure 10 can be used to implement the self-timed version of the clocked pipeline of Figure 5. If we use the same 5 stage pipeline distribution as the clocked pipeline, the backward trains take care of dependencies such as the one in Figure 1, but not of the dependency in 2, which requires the P_k passenger at station m to get information from the P_{k-1} passenger but at station $m+2$. In that case, we will insert a NOP instruction Especially since the benefit of the complex bypass network would be less than 1 cycle for an uncommon case.

The consequence of our decisions is a self-timed pipeline architecture that inserts nop-instructions at the same rate as the clocked pipeline architecture. It has the same

data available for instructions as its clocked counterpart, but data and operations are synchronized only locally. There is no need for global synchronization. The performance of this self-timed pipeline is based on the throughput of a stage. The stage controllers can be implemented by well-known self-timed control circuits. The architecture combines timed simplicity and performance with self-timed advantages.

Using this architecture for a self-timed processor allows us to use the same compiler optimizations for eliminating stall cycles in the clocked counterpart. The tools are compatible.

5 Implementation of a general purpose processor

In this section we describe the implementation of a GP-processor based on the architectural considerations above. We choose to keep the control of the design as simple as possible and will utilize backward paths in a network that is not cyclic to allow previous results to flow backward in the network.

We use the `tpdesign`[] tool to implement our processor design.

This leads to figure 12. We see two neighboring stages that communicate data in two directions. A colored box marked X represents the stage. It waits until both input data are available. It then captures all data and produces the data on both its output arrows. Figure 12 is a snapshot during simulation of the design. The colored arrows represent data channels that contain data. The black arrows represent data channels that are empty.
In Figure 12, the left X state has both inputs available, and both outputs acknowledged, and will be able to capture its input data. The `tpdesign` tool allows high-level simulation, where colored arrows denote full channels and black arrows denote empty channels.

The data channels in Figure 12 can be implemented in various ways. Many self-timed protocols exist to implement these channels. Based on the implementation protocol, the controller for the stages in 12 readily reduce to a basic self-timed uni-directional pipeline controller of which many fast implementations exist.

In our case, we use the regular click-implementation[9] for the storage elements. We do not try to use the returning signal of the controller to carry the data, but rather use a full data path. The acknowledging signal on the forward data path will run in parallel with the forward signal on the backward data path. Control logic in the X controllers may be run in parallel with computations.
Based on the two-way synchronization of Figure 12 we now implement our self-timed process pipeline in the style of Figure 13

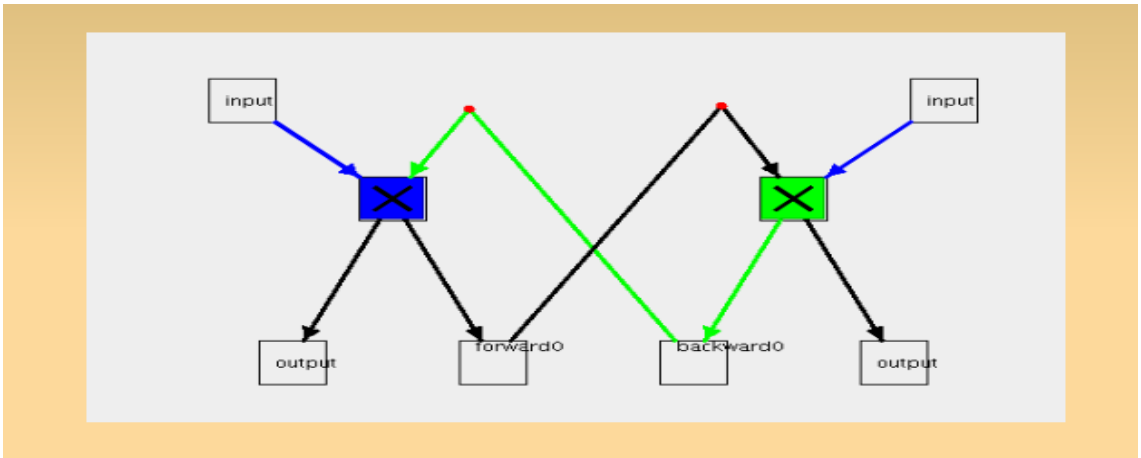

Figure 12: Synchronizing two datastreams at one event

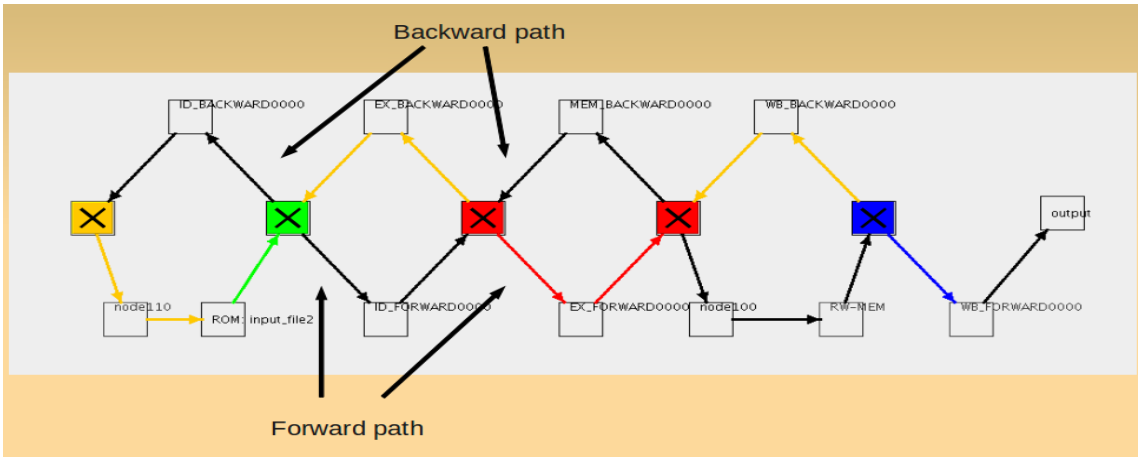

Figure 13: Self-timed processor architecture

Both figures 12,13 are screen-captures from the design tool. The nodes marked X are state-holders. The other nodes either perform computations or access memory (ROM and RW_MEM). The output node at the far right is for monitoring pipeline activity when debugging.

The functions of the node and the types of the channels are implemented in a different file that is dynamically read by the design tool. It is too large to give here, but we will give an abstract that shows how the new address is computed in the ID_BACKWARD computation node. This syntax is the current `tpdesign` syntax. We are still developing the tool, and this syntax will be subject to change.
The types in Figure 14 can be composed of the basic `bool` type, arrays and structs. All types have an associated width (determined by the tool) that is used for the bundled-data buses in the netlist. Functions such as `next_pc` and `if_address` can use all standard operations and are mapped to combinational gates. The function `if_address` is linked to the `ID_BACKWARD` node in design in figure 13. The argument of type `ID_BACKWARD` is obtained from the input channel, and the result of the functions is produced on the output channel.

5.1 Leetl Analysis

- between two stages, either the forward or the backward path is active. A domino logic data-path may reset when the other path is active.
- The control signal that indicates data-availability must be delayed if the combinational logic needs time to settle. The design tool produces tcl scripts that can interface with timing tools to compute these delays. It also make sure that as much of the control as possible executes in parallel with the combinational logic. This reduces the control overhead of 1 ring to 2 flipflops (cp to q) for the click backend, and 2 latches for the future GasP backend. It requires that the backward going paths are utilized, otherwise the backward control cannot run in parallel with combinational logic.
- The current first implementation of the LEETL runs a small program that access memory, iterates and computes a function.
- We can succesfully run the leetl using a verilog gate-model library. We have not yet created a layout.

6 Conclusions

We are implementing the `LEETL` processor in the architecture described in this article and are able to simulate at a gate level. It is an ongoing project and is not finished. Early simulation shows performance that is much better than other self-timed

```
typedef bool[32] Address;

typedef struct
{
JUMP_INFO jump;
EX_RESULT result;
Address next_pc;
} ID_BACKWARD;


function next_pc ( x : JUMP_INFO, ex : EX_RESULT,
                         next_pc : Address ) : Address
{
if (have_to_jump(x,ex))
       then
   if (x.address_select==(2'b00))
   then x.jump_address
   else if (x.address_select==(2'b01))
               then ex.value1
else ex.value2
               fi
          fi
else next_pc
fi
}


function if_address ( x : ID_BACKWARD) : Address
{
next_pc((x.id_backward).jump,(x.id_backward).result,
                                          (x.id_backward).next_pc)
}
```

Figure 14: Partial code for LEETL - next address computation

pipelined processor implementations that are based on standard cell implementations. The design tools allow quick feedback on design decisions. The self-timed `LEETL` implementation is technology agnostic: we are using a Click backend, but we intend to use a GaSP backend in the future.

The architecture that uses the same synchronization points for the backward paths and forward paths is simple and effective. The performance is as good as its clocked counterpart. It has a number of advantages that we list below.

- A clocked implementation can be ported directly to the self-timed implementation. All compilers and other tools apply.
- The self-timed processor can easily be turned on/off.
- The self-timed processor avoids a global clock.
- The local clock signal for the stages has looser balancing requirements than a global clock.
- Stages can be disabled dynamically when not needed. This can introduce speed-ups if the disabled stage is a bottleneck.
- There is less peak current in the design.
- The control overhead of the `LEETL` design can be reduced to 2 latches per cycle. This is the same control overhead we find in a clocked design.
- The performance of this self-timed architecture matches or improves on the performance of the clocked architecture.

A Proof of attainable speed

We proof that in an acyclic network, all trains will eventually reach the cycle time of the slowest unimpeded train.
proof:
We already showed that rings cannot run faster than the slowest speed. What remains is to show that they attain this slowest speed.

In an acyclic network we can create a tree by simply designating a note as the tree top. Since the graph contains no cycles, every node has a unique path to the tree top. We define the *parent* of node A to be the first node on that path. All other nodes that connect to node A must have node A as their parent, and are children nodes. The *parent station* connects a node to its parent. A *child station* connects a node to one of its children. The root node has no parent node.
If a node has no children, it has level 0. For every other node n we define $level.n = 1 + (max\ c : c \in children.n : level.c)$. We now claim:

Let n be a ring at level k. Let s be the parent station. After a train has run the ring k times, the time it takes for the train to go from departure to arrival at the parent station will be at most the slowest ring in the tree.

We prove this by induction.
basis: Rings at level 0 only have a parent station. Since by themselves they run at least as fast as the slowest cycle-time, the will need at most this much time to reach the parent station again.

step:
Let L be the longest time that any unimpeded train in the whole network can take to run its ring. Consider a node at level $k+1$. When the train in this node has run k rings, so will the trains in its children nodes. The children nodes have at most level k, so the trains in those children will need at most L time from now on.

Consider the train departing the parent station to start it's $k+1$-th cycle. It will meet all the children trains along the way, and these children may be late. However, by induction are guaranteed to require less than L time to run their ring. Thus if our train leaves a station at time s, it is guaranteed to be able to leave the next round at most at time $s+L$.

Now consider the next $(k+2)$ round of our train. We have to show that this round will take at most L. Suppose the train leaves the parent station at time t. If the train is not impeded on any of its stations, by definition, it will require less than L time to run its ring.
Suppose that the train is impeded on some of its stations. Consider the **last** station on which it is impeded. We denote the time at which the train left that station in the previous round was s. We denote the time the train left the parent station with t. We denote the time we can leave this last impeding station with u. Then we have $s < t < u$ but also $s + L >= u$ by property of the child train. Since the train will not be impeded at any other station this round, the time it takes to reach the parent station again will be at most $t - s$ (the time it took the previous round). We find for the arrival time of the train at the parent station : $u+(t-s) <= s+L+(t-s) = t+L$. $\square$.

References

[1] Bill Coates, Jo Ebergen, Jon Lexau, Scott Fairbanks, Ian Jones, Alex Ridgway, David Harris, and Ivan Sutherland. A counterflow pipeline experiment. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 161–172, April 1999.

[2] S. B. Furber, P. Day, J. D. Garside, N. C. Paver, and J. V. Woods. A micropipelined ARM. In T. Yanagawa and P. A. Ivey, editors, *Proceedings of VLSI 93*, pages 5.4.1–5.4.10, September 1993.

[3] Hans van Gageldonk, Daniel Baumann, Kees van Berkel, Daniel Gloor, Ad Peeters, and Gerhard Stegmann. An asynchronous low-power 80c51 microcontroller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 96–107, 1998.

[4] J. D. Garside, S. B. Furber, and S.-H. Chung. AMULET3 revealed. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 51–59, April 1999.

[5] Andrew Lines. The vortex: A superscalar asynchronous processor. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 39–48, March 2007.

[6] Alain J. Martin, Andrew Lines, Rajit Manohar, Mika Nyström, Paul Pénzes, Robert Southworth, and Uri Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[7] Alain J. Martin, Mika Nyström, Paul Pénzes, and Catherine Wong. Speed and energy performance of an asynchronous MIPS R3000 microprocessor. Technical Report CSTR:2001.012, California Institute of Technology, 2001.

[8] Alain J. Martin, Mika Nyström, and Catherine G. Wong. Three generations of asynchronous microprocessors. *IEEE Design & Test of Computers*, 20(6):9–17, 2003.

[9] Ad Peeters, Frank te Beest, Mark de Wit, and Willem Mallon. Click elements: An implementation style for data-driven compilation. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 3–14, May 2010.