High Performance Asynchronous Circuit Design Method and Application

Charlie Brej
School of Computer Science, The University of Manchester,
Oxford Road, Manchester, M13 9PL, UK.
cbrej@cs.man.ac.uk

Abstract

Asynchronous circuits have a number of performance advantages over their synchronous equivalents, yet these advantages are rarely realised due to an overhead in the asynchronous design method. This paper investigates the nature of this overhead and proposes a new approach to overcome its effect. The method is then demonstrated on a processor design. The process demonstrates the ability of the method to implement: register locking, instruction reordering, super-scaling, hyper-threading, cache-banking and other complex techniques, easily and often without any effort from the designer.

1 Asynchronous Circuit Performance

Progress in single threaded performance of modern processors is becoming stagnant due to the design complexity of very large scale systems, the design challenges presented by globally distributed clock nets and increasing variation in component delays. Asynchronous circuits have the potential of achieving substantially higher performance targets than synchronous equivalents. Most advantages have been demonstrated and exploited [1][2][3][4], yet these fail to yield the expected performance boost.

1.1 Advantages

The advantages of asynchronous circuits can be broken down into two groups, those brought in by using local delays rather than a global clock and those achieved by using implicit timing when computing (shown in figure 1). The clock overheads consists of unbalanced stages and clock skew. Although unbalanced stages can cause a degradation in performance in synchronous systems by running at the speed of the slowest stage, the bundled data approach does not automatically solve this problem as the bundled data design will still have the critical cycle time equal to the slowest executed stage. So design effort is still required to lessen this effect and it will never be fully removed by selectively executing/bypassing stages. Matched delay overhead is a much greater component of the clock overheads. These generally cannot be overcome by using bundled data approaches, (although data dependant delays can alleviate some of the worst-average delay effect). The basis of these overheads is the worst-case delay assumptions made by estimating the delay of a computational stage. The worst case delay is calculated

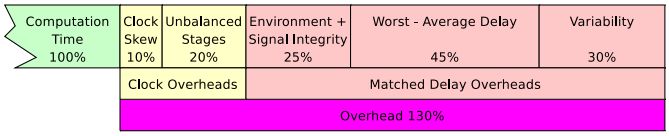

Figure 1: Clock Overheards

by taking the slowest operation within a stage, assuming the worst case voltage and temperature and the most pessimistic distribution of poor transistors and wires.

1.2 Disadvantages

Judging by the overheads of synchronous systems, it should be easy to implement asynchronous designs with 130% higher performance. Yet rarely do asynchronous implementations come close to the speeds of synchronous designs. There are a number of minor effects which contribute to this (area overhead, overly conservative delays, no speed binning...), yet these do not account for the performance decrease removing all advantage gained by the shift to the asynchronous methodology in the first place.

The one major overhead is the inability of data to progress to a stage which is going through the reset phases due to the prevous value passing through it [5]. Many designers seriously under-estimate the granularity of pipelining required to allow tokens to flow freely without colliding with the reset phase of the previous operation. The length of the reset phase is dependant on the protocol and arrangement of the stage.

In the four-phase protocol, the computation happens in parallel with the request assert phase (as shown in figure 2. The protocol then goes though three more phases: acknowledge assert, request release and acknowledge release. In early output designs [5], the output is generated even before the request collection has completed.

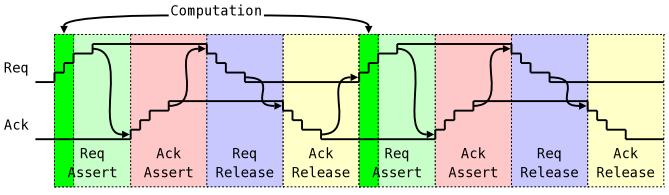

Figure 2: Four Phase Handshake

Not all the phases are of equal length, for example, bundled data circuits have large data bundles which require little synchronisation and well constructed asymmetric delays mean the request release phase is much shorter than the computation

phase. In early output designs the throughput delay is more than six times greater than latency delay (depending on the logic arrangement and data). This ratio can be much greater in stages where the average case computation is much shorter than the acknowledge and request propagation delay (which are always worst case). One example of this is a carry ripple adder where the result is often generated with little or no carry propagation, yet the request/acknowledge collection/release travel the full length of the carry path. In such cases it is common to see stage cycle times 20 times greater than the latency delay.

2 Wagging Logic

Early output circuits are designed to propagate the result as soon as possible to the next stage where it can be worked on while the current stage completes and resets ready for the next computation cycle. In a scheme where the number of data tokens is equal to the number of stages, the tokens will spend only a small fraction of time being computed, the remainder of time they wait for the next stage to complete its reset phases. There have been a number of approaches to decrease the time spent waiting, the most common of which is increasing the level of pipelining. This allows parts of the computation stage to begin the reset phase before the full stage has completed. It also reduces the complexity of each stage and thus decreases the distance the completion signals have to travel. Aggressive use of this strategy in slack matching[6] systems yields a pipelining latch for every two gates. This produces optimally pipelined circuits, unfortunately the latency introduced by the additional latching causes tokens to now spend half the time being latched.

Wagging logic attempts to increase the number of stages by replicating the logic of each stage and cycling which copy of the logic the data should go though. This allows one of the stages to work while the others are resetting, yet it does not increase the latency of the stage. Additional latency added is outside the stage where the cyclic data distribution and collection adds delay.

Each stage has a ”level of wagging” which signifies the number of copies of logic the stage contains. Each copy is called a ”slice” and has a slice number associated with it. The inputs and outputs of each slice are connected to ”mixers” which collect the data from the outputs of one wagging logic stage and then distribute the data to the slices of the next stage. The mixers also latch the data to allow the connected slices to work independently. Figure 3 shows an example pipeline with a combination of non-wagging and wagging logic stages. These are connected using a selection of mixers.

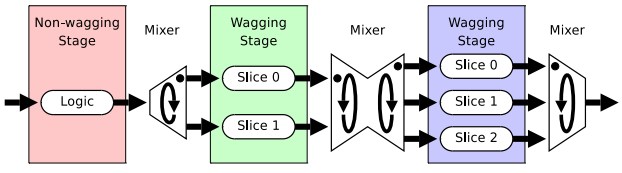

Figure 3: Example Wagging Pipeline

2.1 Wagging Mixers

To connect two wagging stages, it is possible to multiplex the data of the input stage to a single channel and then demultiplex it again to the second stage. The alternative is to demultiplex first to a set of intermediate channels and then multiplex again. The number of intermediate channels is the LCM (lowest common multiple) of the level of wagging of the two stages. These constructions can be seen in figure 4.

When connecting two stages with an equal level of wagging, multiplexing becomes unnecessary as the lowest common multiple of any number and itself is itself. The output of stage X is passed as an input to stage X+1 (mod the wagging level). If a single level of wagging is used in an entire design, the latency overhead is limited to the interfaces between wagging and non-wagging logic. Because non-wagging logic segments are likely to be the bottlenecks of the system performance, it is advisable to place the interfaces on low bandwidth channels.

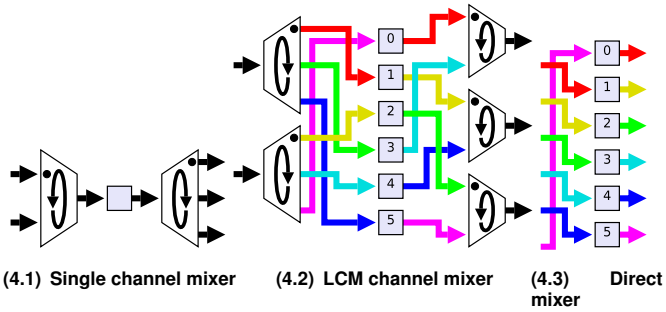

Figure 4: Mixer designs

2.2 Example Wagging Design

Figure 5 shows a design of a simple accumulator circuit. The circuit has two operations: ”Load” reads a new value into the register and ”Accumulate” adds the value in the register to the input value and writes the result back to the register. The type of operation executed is declared in the input token, and it directs the multiplexer to pick the appropriate value. The contents of the register is also passed out each cycle, but the environment often discards the value and just acknowledges. The worst case delay of the stage is the delay of the adder and the multiplexer.

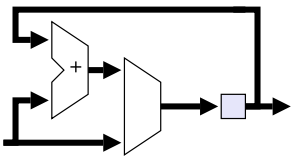

Figure 5: Example Accumulator Design

Figure 6 shows the design and example operation in a level six wagging logic implementation of the accumulator. The sequence of operations passed through it is:

0. Load
1. Accumulate
2. Accumulate
3. Load
4. Accumulate
5. Load

In the figure, the data dependencies can be seen in the black arrows and units (results of gray units were discarded) . Because the two sequences of accumulates have no data dependencies (the black line regions are unconnected), the two sets of accumulations can be executed in parallel. This can be seen in figure 7, where the operations conducted by the circuit are shown flattened. Unless there is a conflict over hardware resources, which in this case there is little due of the high level of wagging, the timing of operations is only dependant on the arrival of inputs. If data of the first self contained sequence is late, or the computation is slow, the second sequence can complete before the first has generated a result. This allows proceeding stages to reach ahead and try conducting as much processing as possible before they strictly require the data from a late arriving input.

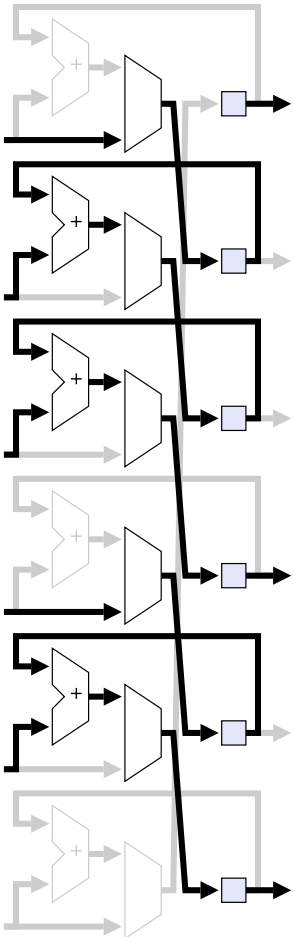

Figure 6: Wagging Accumulator

Because early output logic uses bit level pipelining, each bit of the result progresses to the next stage as soon as it is calculated. This is highly advantageous when using units such as adders which have an ordered sequence of desiring input values and generating output values. This greatly reduces the delay of two adders placed in series as parts of the result generated early by the first adder are the parts desired first by the second adder.

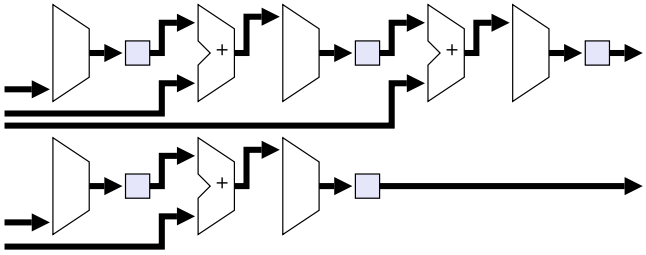

Figure 7: Flattened Accumulator Operation

3 Red Star

"Red Star" is a simple processor designed to explore the possibilities of wagging logic. Using wagging logic, and other techniques outlined below, the design becomes very large and it is not suggested that this is a reasonable alternative to common place practises at this point in time. What is presented is a set of techniques which will become increasingly viable as the price of increasing single threaded performance (in both area and design effort) continue to rise. The target of the design is to implement traditionally complex architectural features with little design effort and combine this with the fastest computation possible. The issues of area and power consumption are not addressed at this point.

3.1 Datapath

The datapath is designed in a synchronous style with a small alteration. The design is fully functional in the synchronous form. The only alteration made is the addition of two stages before committing the results back to the register bank or updating the PC to a branch target. This increases the branch penalty by two instructions (the branch shadow). This penalty is unnecessary in the synchronous version and only reduces performance. In the asynchronous wagging design this increases the number of instructions prefetched. The penalty of prefetching a greater number of instructions is not felt in the wagging version as the resources wasted executing these operations are not shared with the instructions fetched at the branch target address. Nor is there a large delay before the branch target instructions begin to be fetched, as the new PC value quickly progresses through the two additional empty stages and informs the target slice the address to fetch the next instruction from.

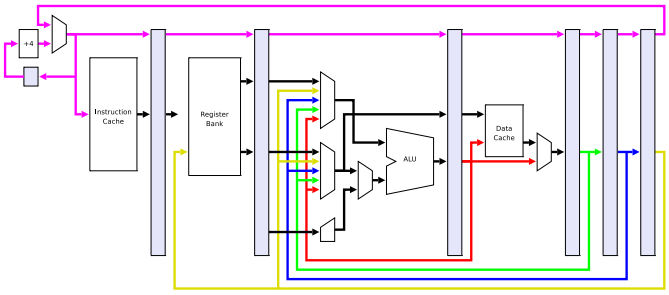

Figure 8: Red Star Datapath

3.2 Register Bank

All storage, in early output logic, is constructed using FIFO latches. Each cycle, the value is removed from the latch and a new value is written to it. Register bank latches have enable inputs which select whether the new data input should be written or the old value should be recycled. This kind of construction is wasteful of both energy and area, but it fits well with wagging logic. In wagging circuits, the contents of the whole register bank is copied to the register bank in the next slice. If the data, a register is to be written with, arrives to the register bank late, the value will arrive at the next slice some time later than the others. Unless this register is read in the next slice, there is no need to wait for its arrival before reading one of the other registers. Once the value is entered into the register bank in one of the previous slices, it can propagate through the slices and quickly catch up the computation wavefront. This enables the register bank to block while waiting for the value to be written. Additionally this allows the register bank reads and writes to be executed in parallel and even out of order. If the value, to be written to a register, is still being computed, there

is nothing stopping the following instructions from writing to the next slice of the register bank (to any register).

3.3 Caches

The two caches (instruction and data) are the only connections between the processor and the environment. The position of the interfaces between wagging and non-wagging logic is likely to be a bottleneck unless a low bandwidth point is chosen. Having a single cache shared between all slices will cause the system to be bound by the performance of the cache. The alternative of treating the cache like a register bank and copying its entire contents to the next slice is impractical. Here, two different approaches are used to allow independent parallel access to a component, yet keep coherence between the instances.

As data passes through slices of a wagged circuit, it may be advantageous to be aware of which slice number it is currently in. This can be easily achieved by placing a component which outputs the slice number to the logic function. In the synchronous design such a component can be made with a mod X counter, where X is the wagging level. Although it is not necessary to set the wagging level until the later stages of the design process, if the slice number component is used, the design must be aware at least of the width of the slice number variable.

In the Red Star design the wagging level is targeted at 16. If the processor starts executing an instruction from address 0 in slice 0, if there are no branches, the next instruction to be executed in slice 0 will be from instruction address 16. If no branches were executed, slice 0 will execute only instructions with the bottom four instruction address bits equal to 0. The instructions in slice X will have come from an address with the lowest 4 bits set to X. If the slices had separate instruction caches with no coherency between them, the cache of each slice would contain only the values of instructions with the slice number at the bottom four bits. This is, of cause, destroyed when branch instructions cause slices to fetch instructions which have addresses not associated with that slice.

By comparing the branch target address with the slice number, it is possible to determine if the slice should fetch this instruction or pass the branch target unchanged to the next slice and execute a NOP. Only once the desired slice is reached does the new code segment begin to be executed.

The advantage of this technique is to break up the large instruction cache into small fast segments, allow their access to be parallelised, ensure the data is not replicated between the caches and allow the removal of the bottom four bits of the address.

The data cache requires coherence between individual caches but the bandwidth to it is much lower than the instruction cache, so the option of having a single data cache is a possibility. The alternative is to implement independent write-through caches with a coherency network between them. This is an attractive option as there is correlation between the data addresses accessed and the instruction addresses they are accessed from. This could lead to increased cache hit rates.

The data cache can be constructed using a set of separate caches, along with a small level zero cache which is propagated the same way as the register bank. The level zero cache stores the write accesses of the previous N instructions. If there is no write access to a cache in a slice, a value from the level zero cache is written to the cache and the committed flag is marked for that slice number in the level zero cache. Once a value has been committed in all caches it can be removed from the level zero cache. This strategy is somewhat complicated and a routine to deal with an overflow of the level zero cache must also be designed. It is also possible to reduce the number of data caches from the number of slices down to a lower number by sharing a single cache between several slices.

3.4 Hyper-Threading

The latch inputs to slice zero are of a different design than the other latches in the system. Most latches are made with a half-buffer design but the level zero latches must start with a token reset time. These latches could start with two tokens at reset time causing two computational wavefronts. Because the data of the two wavefronts is carried with them, the circuit holds no state when the next wavefront reaches it. This allows the two wavefronts to be fully independent yet be executed on the same hardware. This effectively replicates hyper threading where a single set of resources is shared between two (or more) threads. There are still points where the design must be changed for the correct functionality. Elements which do not keep all their data in the wavefront (in this case caches) must be manually protected from one thread accessing or damaging the data of another. This can be done by adding a variable which holds the ID of the thread, carried by the thread. This ID can be used inside the caches to determine which data is accessible.

The second necessary change is the interface with non-wagging logic. The interfaces now, rather than cycling through the slices in order, must interleave accesses from the two threads. This can be done in either deterministic precalculated sequence, or using arbiters to serve the requests in order of their arrival.

The latch controllers of slice zero can dynamically add or remove threads from the processor with the state of the whole thread being moved to or from the slice zero latches. This data can be accessible through the memory interface and be stored in RAM.

4 Conclusions

This paper presented a powerful strategy to overcome the greatest obstacle in generating high performance asynchronous circuits. The application of the method and additional high-performance techniques are demonstrated on an example processor design. Implicit data dependency tracking allows the engineer to concentrate on higher level architectural improvements rather than worrying about blocking stages and sharing resources.

Although this method has the ability to create very high speed circuits, it does come at a large cost and is probably still too expensive to justify its use. As the cost of increasing single thread performance increases, this strategy will become progressively more viable.

4.1 Future Work

Some preliminary experiments have been conducted on wagging circuits, which yielded favourable results, but the method is still largely manual. The target of future work is to implement a complete flow of specification, implementation, optimisation and technology mapping. The final flow should be adequately comprehensive to be used in an industrial setting.

Parts of this flow, such as wagging logic generation, optimisation and dynamic timing analysis, have been implemented before as stand alone components. These are currently being reimplemented into a single modular tool which allows information from simulation based analysis to be used in all stages of the design process.

To demonstrate the technique, Red Star will be synthesised down to layout level, with the possibility of a manufacturing run. It is hoped that the design could compete with current high performance cores, but with a fraction of the design effort.

References

[1] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 290–299. IEEE Computer Society Press, 1997.

[2] J. D. Garside, W. J. Bainbridge, A. Bardsley, D. M. Clark, D. A. Edwards, S. B. Furber, D. W. Lloyd, S. Mohammadi, J. S. Pepper, S. Temple, J. V. Woods, J. Liu, and O. Petlin. Amulet3i - an asynchronous system-on-chip. In *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 162, Washington, DC, USA, 2000. IEEE Computer Society.

[3] Montek Singh, Steven M. Nowick, Jose A. Tierno, Sergey Rylov, and Alexander Rylyakov. An adaptively-pipelined mixed synchronous-asynchronous digital fir filter chip operating at 1.3 gigahertz. In *ASYNC '02: Proceedings of the 8th International Symposium on Asynchronus Circuits and Systems*, page 84, Washington, DC, USA, 2002. IEEE Computer Society.

[4] Montek Singh and Steven M. Nowick. High-throughput asynchronous pipelines for fine-grain dynamic datapaths. In *ASYNC '00: Proceedings of the 6th International Symposium on Advanced Research in Asynchronous Circuits and Systems*, page 198, Washington, DC, USA, 2000. IEEE Computer Society.

[5] Charles Brej. *Early Output Logic and Anti-Tokens*. PhD thesis, 2005.

[6] Peter A. Beerel, Nam-Hoon Kim, Andrew Lines, and Mike Davies. Slack matching asynchronous designs. In *ASYNC '06: Proceedings of the 12th IEEE International Symposium on Asynchronous Circuits and Systems*, page 184, Washington, DC, USA, 2006. IEEE Computer Society.