A Mixed-Clock Issue Queue Design for Globally Asynchronous, Locally Synchronous Processor Cores [*]

Venkata Syam P. Rapaka
Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh, PA, USA
vp@ece.cmu.edu

Diana Marculescu
Carnegie Mellon University
5000 Forbes Ave,
Pittsburgh, PA, USA
dianam@ece.cmu.edu

ABSTRACT

Ever shrinking device sizes and innovative micro-architectural and circuit design techniques have made it possible to have multi-million transistor systems running at multi-gigahertz speeds. However, such a tremendous computational capability comes at a high price in terms of power consumption and design effort in distributing a global clock signal across the chip. One of the most promising strategies that addresses these issues is the Globally Asynchronous, Locally Synchronous (GALS) design style where multiple domains are governed by different, locally generated clocks. Due to its inherent complexity, a possible driver application for such a design style is the case of superscalar, out-of-order processors. While micro-architectural evaluations for GALS microprocessors have been made available recently, no concrete implementations have been analyzed in a detailed way. In this paper we propose a **mixed-clock issue queue design** for high-end, out-of-order superscalar processors, able to sustain different clock rates and speeds for the incoming and outgoing traffic. We compare and contrast our implementation with existing synchronous versions of issue queues used stand-alone or in conjunction with mixed-clock FIFOs for inter-domain synchronization.

Categories and Subject Descriptors

B.2.2 [**Performance Analysis and Design Aids**]: Simulation; B.4.3 [**Interconnections (Subsystems)**]: Asynchronous/synchronous operation

General Terms

Design, Measurement, and Performance

Keywords

GALS, Issue window design, mixed-clock circuits

1. INTRODUCTION

Moore's Law's, which predicted a trend of exponential growth in transistor density and performance still holds true for current generation systems and it is expected to hold for the next few generations. In fact, shrinking process technology and novel design styles have resulted in highly dense circuits, that are capable of operating at very high frequencies. As a result, present day microprocessors based on such circuits have emerged as powerful computational systems by including innovative micro-architectural and circuit design techniques. For these types of systems, as well as for general ASICs, the synchronous design paradigm has been the popular choice during the past decades. Commercial CAD tools and a well established design flow have made this choice particularly attractive.

In synchronous design, the master clock acts as a timing reference signal for all basic modules of the design. A well-designed global clock distribution network, with additional circuitry for keeping clock skew under reasonable limits is required to make sure that local clock signals reaching different computational blocks are synchronized. However, due to the increasing number of transistors and complexity of today's designs, a continuous reduction in clock skew is possible only by careful design and simultaneous consideration of global interconnect delay increase.

In recent years, the Globally Asynchronous, Locally Synchronous (GALS) [7, 13] approach has been explored to tackle this problem. Such a solution eliminates the requirement of a global reference clock signal by assuming that the system is comprised of several synchronous blocks communicating asynchronously. In both existing approaches, the treatment of the impact of asynchronous communication on overall performance and power consumption is done at micro-architectural level, where underlying circuit details are obscured or even unknown. One of the most important pieces in the overall GALS design of a superscalar, out-of-order processor is the *issue queue*. While in [7],[8] it is assumed that the issue queue is fully synchronous and is interfaced asynchronously with the dispatch clock domain via already existing asynchronous FIFOs, in [13],[12] the issue queue itself is assumed to be mixed-clock, thus allowing for dispatching and issuing instructions asynchronously. However, a physical, detailed implementation of such issue queue has not been presented, nor analyzed.

The contribution of this paper stems in proposing a **mixed-clock issue queue design**, able to sustain different clock rates and speeds for incoming and outgoing traffic. In ad-

dition, we demonstrate via detailed circuit level simulation its role as a communication interface between the dispatch and execute clock domains, and compare and contrast the power consumption and throughput of our implementation with existing synchronous versions of issue queues used in stand alone mode or in conjunction with mixed-clock FIFOs [4] for inter-domain synchronization.

1.1 Prior Work

Issue logic plays an important role in the performance of a superscalar processor. Its functionality is achieved through the use of two important functions namely *wakeup* and *select*. A detailed analysis of complexity-effective superscalar processors has been presented in [11]. Only the tag-matching part of the wakeup logic and a hierarchical position based scheme have been described there. The selection logic is very efficient if a single request signal is to be selected. For selecting two (or more) request signals, the selection blocks have to be used in series. The request signals to the subsequent selection blocks are masked based on the grant signals generated by the previous selection blocks. Such a method can be extremely slow for selecting more than one signal. A possible alternative for a high speed four-way issue queue has been presented in [9]. In the proposed solution, the issue queue is partitioned into smaller parts and one instruction is selected from each part, thus reducing the available parallelism in the pipeline drastically. A moderate speed two-way issue queue design has been presented in [5], but it requires a large number of connections in an asymmetrical manner. This entails the usage of strong buffers for some signals, while the overall structure can lead an inefficient layout. While our developed wake-up logic is based on the scheme described in [11], our selection logic is similar to the one described in [5], but with some significant changes.

Synchronization is crucial for a mixed-clock design and various schemes have been proposed to address this problem. A robust FIFO-based approach has been presented in [4]. In this case, synchronizing latches are used for communication of handshake signals between clock domains. Although it has a low latency in the steady state operation, the worst case latency is two clock cycles. The use of such a FIFO for inter-domain synchronization seems to be feasible, but it increases the number of transistors in the circuit and introduces an additional stage in the pipeline. A stretchable clock approach has been proposed in [14]. Such an interface primarily addresses the interface between asynchronous and synchronous modules. If used for communication between two clock domains, it eliminates the skew between the two clocks and ensures that the rising and/or falling edges are synchronized. However, such a technique cannot be used if the producer and consumer clocks are very different in speed. Our synchronization approach is similar to the one described in [4] in the sense that it is based on synchronizers for inter-domain communication, but we use the issue queue as the interfacing structure, without the need of additional storage structures.

1.2 Organization of this Paper

The rest of the paper is organized as follows:

- In section 2, we describe the basic features of the superscalar pipeline organization considered.

- Our approach for using the issue queue for communication is illustrated in section 3.

- In section 4 we compare our approach with the synchronous case with and without synchronizing FIFOs and present the experimental results.

- Section 5 concludes the paper and discusses possible directions for future research.

2. THE SYNCHRONOUS SUPERSCALAR PIPELINE

Superscalar, out-of-order execution in high-end processors adds another dimension in terms of performance boost by exploiting available instruction level parallelism (ILP). There is a trade off between the complexity of the processor and the operating frequency. A detailed analysis of complexity-effective superscalar processors has been presented in [11]. For the purpose of our study, we consider a superscalar, out-of-order processor, as shown in Figure 1. A brief description of each stage is as follows:

- The *fetch* stage fetches the instructions from the I-Cache and the program counter is updated using the branch prediction logic.

- The *decode and rename* stage is responsible for assigning available physical registers to the current logical destination registers and read the current mappings of the source registers from the map table. It also resolves the RAW dependencies (if any) in the current pool of logical source and destination registers and updates the register status table indicating the non-availability of the assigned physical destination registers.

- The *dispatch* stage is responsible for checking for the availability of physical source registers and entering instructions into the issue queue. This is done by accessing the register status table, and comparing the source registers with destination registers being written back during the current cycle. The available entries of the issue queue are also determined in this stage by maintaining an availability FIFO.
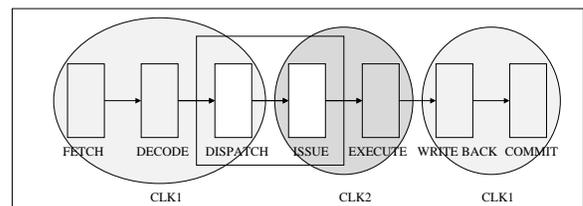

Figure 1: The basic pipeline

- The *issue* stage wakes up the ready instructions and selects instructions based on the availability of functional units using a fixed priority scheme.

- The *execute* stage consists of the functional units performing the operation specified by the instruction.

- The *write-back* stage updates the physical destination registers with the values generated after execution.

- The *commit* stage ensures in-order retiring of the instructions.

In this paper, we only concentrate on the interplay between the *dispatch* and *issue* stages where it has been suggested that an asynchronous interface may be introduced in GALS processors [7],[13]. To this end, we consider the underlying microarchitecture organization from Figure 1 and concentrate on the interface between the dispatch and issue logic in three cases: (i) the fully synchronous case, where both dispatch and issue stages run at the same, globally generated clock signal; (ii) a GALS case, in which the dispatch and issue stages are placed in different clock domains and interfaced via an asynchronous FIFO [4]; (iii) a second GALS case, in which the issue queue itself is mixed-clock and supports dispatching from and issuing instructions to different clock domains. We will compare the three designs in terms of power consumption, energy per issue, and worst-case latency.

2.1 The Baseline Issue Logic

Without loss of generality, we have considered a 2-way wide pipeline. However, our implementation is easily extended to a 4-way or a 8-way case. The issue logic has two primary functions, namely *Wakeup* and *Selection*.
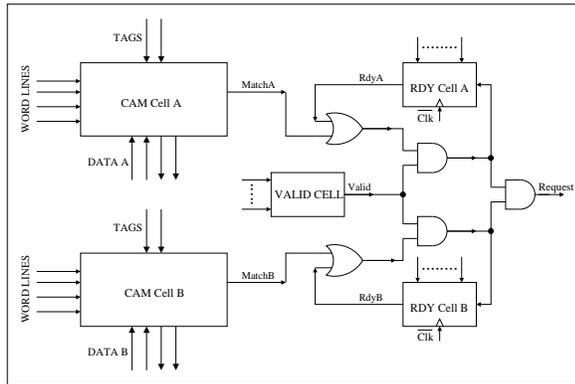

Figure 2: Request generation logic

A typical entry in the issue queue is illustrated in Figure 2. It is made up of five components, the physical mappings of the two source registers, the two *availability* flags for these two registers and a *Valid* bit for the whole entry. As described in the dispatch stage, a source register can become "ready" before it has been entered into the issue queue. In case of instructions having a single operand, the other source register is set to "ready" to avoid deadlock. The source registers are compared with the tags of destination registers being written back during each cycle and they become ready if a match occurs. A request signal is generated when both source operands become ready. The *Valid* bit is set when the instruction enters the issue queue and is reset when it has been selected for execution.

Many entries in the issue queue may generate request signals during a clock cycle, and only a few (two in our case) have to be selected based on the availability of functional units. Several schemes have been proposed previously for instruction selection. In our design, we employ a *position based selection scheme*. An instruction is selected by providing a grant signal by the end of the clock cycle, and the corresponding entry is read in the beginning of the next clock cycle. The corresponding grant signal is used for reading the components of the entry and for resetting the *Valid*

bit, which effectively erases the entry from the issue queue. This freed entry is updated back in the availability FIFO.

2.2 Request Logic Implementation

The request generation logic shown in Figure 2 has three important components namely: CAM cells, the *Valid* bit and the *Ready* flag. The CAM Cells consist of two four-ported SRAMs with a *matching* circuitry similar to the one presented in [11]. The first two ports as configured as write ports and the other two are configured as read ports. The two write ports are used by the dispatch logic for writing entries into the issue queue and the read ports are used by the issue logic to send the selected instructions to the execution units. Each entry in the issue queue generates a request signal and receives a grant signal from the selection logic. This grant signal is used as the word signal and the positive phase of the clock signal is used as the read signal while reading the data. All the grant signals are reset to LOW in the negative edge of the cycle to allow pre-charge of the bit lines. The *Ready* flag for a source register can be set either before the instruction enters the issue queue or it can happen after it enters. The implementation of the *Ready* flag is shown in Figure 3. A 2-port SRAM is used for writing the status of the source register just before it enters the issue queue. The source register becomes ready if it matches one of the tags and it should remain ready even after the tag value changes. We use a negative edge triggered flip flop for storing this value. This value should be erased when a new entry is written, so the rising edge of one of the two word lines is used to reset the stored value.
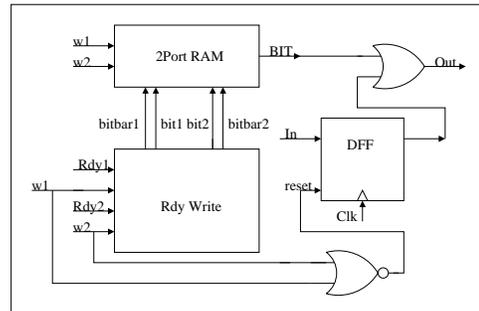

Figure 3: The Ready flag

2.3 Selection Logic

The 32-entry issue queue generates 32 request signals during the positive phase of each cycle. In our two-way wide pipeline, the selection logic has to select two out of the activated request signals. A block level illustration of the selection unit is shown in Figure 4. A 32-bit wide bus, *Request* is used for asserting the 32 request signals and the grant signals are asserted on the two 32-bit wide output buses, *Grant_1* and *Grant_2*. Only one of the signals of these two buses will be asserted high. We have implemented a fixed priority scheme for selecting the instructions based on their position in the issue queue, i.e out of the all the asserted *Request* signals the signals with the two lowest indices will be selected. This can be easily extended to an oldest first scheme ([9]).

In the first stage all *Grant_1* signals are precharged, and all the asserted *Request* signals will discharge the *Grant_1* of all the signals below them. This will leave the *Grant_1* signal
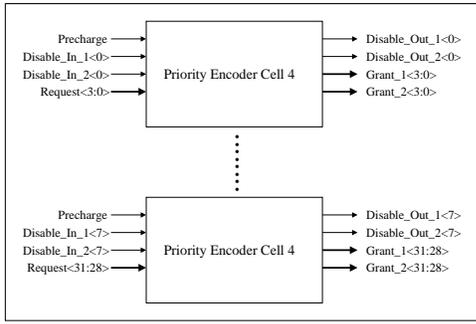
Figure 4: The selection unit

of the *Request* signal with the highest priority un-discharged. In the second stage all the *Grant_2* signals are precharged and all the asserted *Request* signals whose *Grant_1* was discharged will discharge all the *Grant_2* signals below them. Thus, this scheme will leave the *Grant_2* of the top two *Request* signals un-discharged. Static logic gates are used to de-assert the *Grant_2* signal if the *Grant_1* signal remains un-discharged. The circuit level implementation for two grant signals is shown in Figure 5. We have used dynamic circuits for disabling the lower level grant signals as opposed to static gates used in [5] for speed considerations.
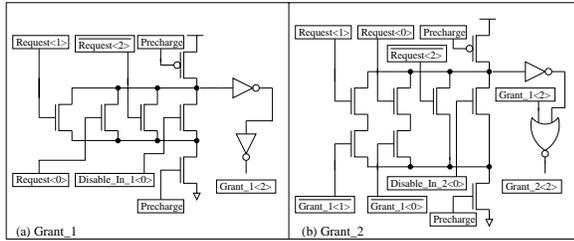

Figure 5: The Grant circuits

The *Disable* signals (or the *Kill* signals) from each signal are connected directly to all the *Grant* signals below. For instance the *Request<0>* signal has to be present in discharge circuitry of all the grant signals below. Thus, this scheme entails a strong buffer circuitry for all the top order request signals and can also lead to an inefficient layout. We made a minor change to scheme by grouping four *Request* signals and generating a single disable signal for the whole group. Thus, in the worst case the disable signal from the top-most group will have to be connected to only seven groups below, thus reducing the number of interconnected lines considerably. It is to be noted that this scheme can also improve the efficiency of the layout of the selection unit. We found this hierarchical scheme to perform better both in terms of power and performance when compared to the brute force scheme. The final circuit level implementation is illustrated in Figure 6

3. MIXED-CLOCK ISSUE WINDOW DESIGN

We have assumed a simplified mixed clock pipeline as illustrated in Figure 1. In the synchronous version of the pipeline Clk1 and Clk2 are the same. In the GALS version of the pipeline, Clk1 and Clk2 are different. The interface between the two clock domains is assumed to be done via a mixed-clock issue queue. For comparison, we also consider

the case when the entire issue queue is fully synchronous with Clk2, and the case in which synchronization with Clk1 is done via a generic mixed-clock FIFO [4]. In all three cases, we assume a 32-entry, 2-way wide issue queue.
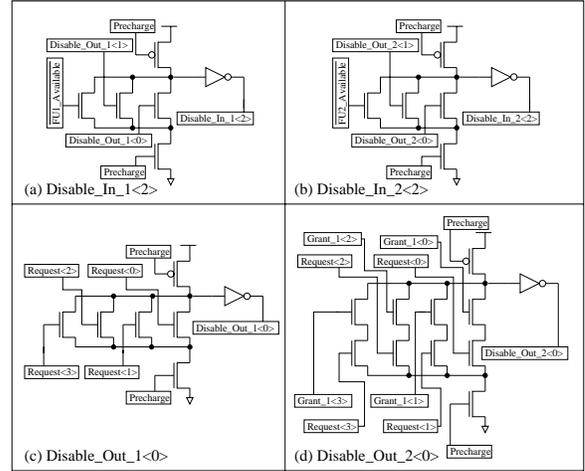

Figure 6: The Disable circuits

3.1 The Valid Bit

The mixed-clock issue queue differs from its synchronous counterpart in the implementation of the *Valid* bit. The *Valid* bit plays the crucial role for synchronization between the two clock domains. Since the *Valid* bit is written by the dispatch unit and read by the issue unit, there can be a timing violation if it is directly used in the issue domain. To cope with this problem, we use two synchronizers to increase the *Mean Time-to-Failure (MTF)* of the *Valid* bit while using it in the clock domain of issue logic. The *Valid* bit implementation is illustrated in Figure 7. Similar to the case of CAM Cells, it also has four ports and four word signals. No explicit data signals are required as the *Valid* bit is set when an entry is written into the issue queue and it is reset when the entry is read. Hence, if one of w1 or w2 is high a logic high is written into the *Valid* bit and a logic low is written when either w3 or w4 goes high. Since the *Valid* bit has to be initialized to zero during reset, the negative pulse of the reset is also used to set the *Valid* bit to zero.
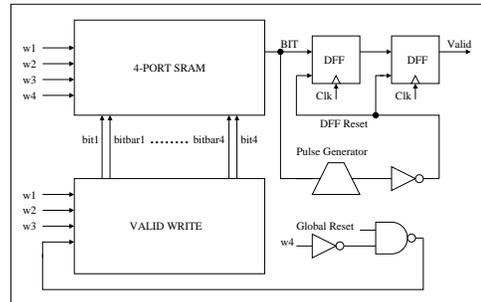

Figure 7: The Valid bit

Since its value is set to zero in the issue clock domain, it need not be synchronized again. So, the falling edge of the *Valid* bit is used for generating a negative pulse which resets the two synchronizers to zero. The pulse generator circuit is illustrated in Figure 8.
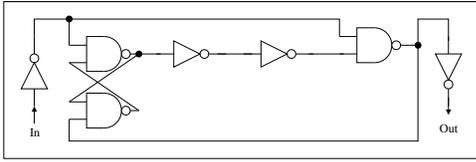
Figure 8: Pulse generator

3.2 Synchronization of Tag Matching

During the operation of the pipeline, we have to synchronize the tags generated by the execution units with the write-back and dispatch units running on *Clk1*. Additional care has to be taken to avoid deadlock in the pipeline. Consider the worst case situation in the pipeline illustrated in Figure 9. The execution units generate the results of destination register X (say) in the negative phase of clock cycle *a0* of *Clk2*, and the tags are broadcasted in the positive phase of *a1*. The tags are synchronized to *Clk1* by using two synchronizers and they can be used only after two rising edges of *Clk1*. In the worst case the rising edge of clock cycle *b1* just misses the generated tags, i.e the setup time is violated. In this case the tags become available in the positive phase of clock cycle *b3* of *Clk1*. Thus, any instruction entered into the issue queue before *b3*, which requires the result of X marks the corresponding source register as unavailable. In the worst case, an instruction requiring the result of X is entered into the issue queue in the negative phase of clock cycle *b2*. It becomes valid in the domain of *Clk1* in *b3* and in the domain of *Clk2* in *a4*. So, this instruction is waiting in the issue queue for a tag which has been generated four cycles ago. If pipeline operates in a way similar to its synchronous counterpart, this instruction can never be issued and a deadlock occurs. This problem can be solved by postponing the tag matching by three cycles. The tags generated in the cycle *a0* are made available for comparison in cycle *a4*.
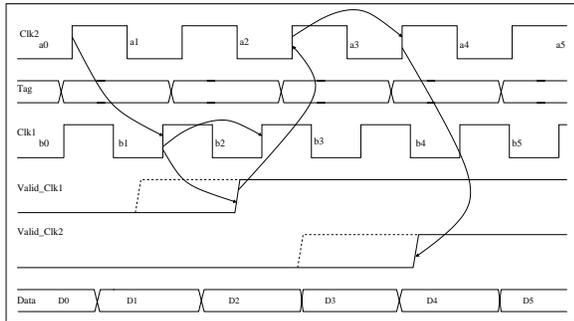

Figure 9: Synchronizing tag matching

3.3 Mixed-Clock FIFO

The implementation details of the FIFO along with considerations to avoid deadlock have been described in [4]. In this section, we will analyze the FIFO based scheme for a mixed-clock superscalar pipeline. Consider the mixed-clock pipeline in which the dispatch unit and the issue unit are running on two different clocks. It is not a trivial task to interface these two units using a mixed-clock FIFO. The data which is passed from the dispatch unit to the issue unit consists of the physical mappings of the source and destination registers along with the availability of the source operands.
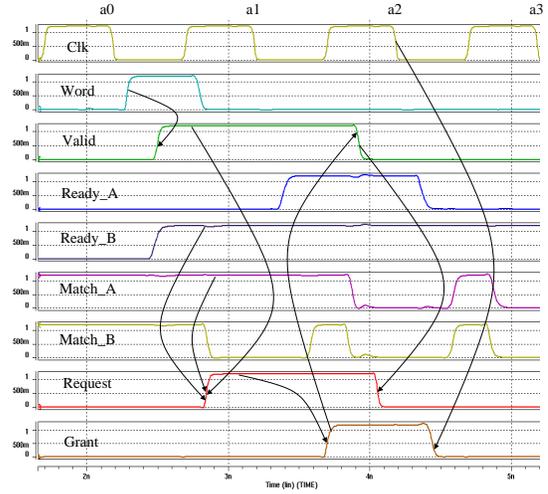

Figure 10: Synchronous issue window

But, the entries to be entered in the issue queue may have to wait in the FIFO for an arbitrary amount of time. We will have to add tag matching capability to each entry in the FIFO. We will also have to address all the issues discussed before in this section for synchronizing tag matching. Once the entries are read from the FIFO they have to be written into the issue queue. Additional care has to be taken to update the status of the operands of the entry being read in the current cycle before it is written into the issue queue. We have to check if any of the tags being broadcasted in the present cycle match with the operands of the entry. It is to be noted that this is exactly what happens in the dispatch stage along with accessing the register status table. Hence, the work done by the dispatch stage before the entry is entered into the FIFO is almost redundant. So, it may be prudent to shift the FIFO before the dispatch stage, and to run the dispatch and issue stages synchronously.

4. EXPERIMENTAL RESULTS

We have designed all the circuits described in this paper using an STMicro 0.13μm technology. We have done the pre-layout simulations using hspice for the issue logic design for three cases. In the first case the issue logic is synchronous with the dispatch logic and both run at the same speed of 1GHz. The corresponding simulation results are shown in Figure 10. The dispatch unit writes an entry into the issue window in the negative phase of the clock. The entry is set to valid, the source operand A is not available and the source operand B is available. The entry tries to match the source operand A with the tags of the destination registers forwarded from the execution unit. When a tag matches with operand A, a request is asserted and in this case this entry is selected for issue. In the next clock cycle, the source operands are read and the instruction is issued for execution. This entry is also erased from the issue window by resetting the *Valid* bit.

In the GALS pipeline, the mixed clock issue window interfaces between two different clock domains: the dispatch unit is running on Clk1 of 1.1 GHz and the execution unit running on a Clk2 of 1 GHz. The choice of the clock speed for the dispatch unit has been done based on timing analysis of the dispatch and rename logic (that are likely to be

Component	Synchronous	Mixed-Clock	FIFO
Power	21.02 mW	22.08 mW	25.11 mW
EPI	13.13	17.94	18.83
WCL	1	4	4

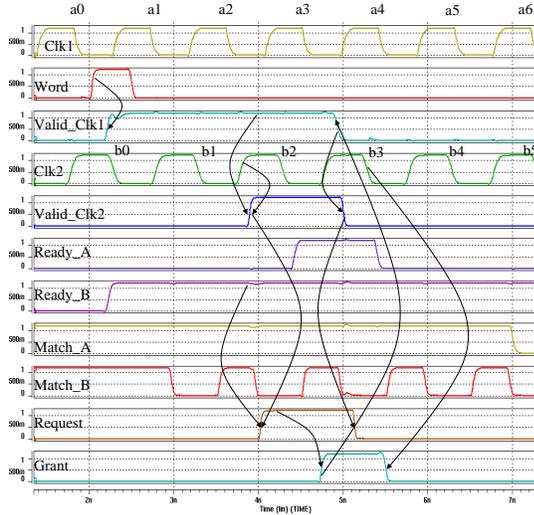**Table 1: Comparison of the three schemes**

Figure 11: Mixed clock issue window

on the critical path in the front-end of the pipeline). The simulation results have been illustrated in Figure 11. An entry is written into the issue window in the negative phase of Clk1. The *Valid* bit is passed through the two synchronizing latches and takes two clock cycles to become active. After this instant, the operation of the issue window is similar to it's synchronous counterpart. It is to be noted that we have to delay the forwarding of destination tags by three clock cycles to avoid deadlock in the issue window. In this case the required register is generated in *b0*, but is broadcast in *b1*. The GALS pipeline with the FIFO acting as the interface adds an additional stage in the pipeline along with the synchronization penalty. The pipeline behaves in the same manner as the synchronous version, but entails a worst case delay of three cycles. We compared the three strategies in terms of power consumption, the energy consumed per issue (EPI), given by Equation 1 (*Power* is power consumed by the circuit in mW, T_n is the time required in ns to issue the n instructions), and the Worst Case Latency (WCL).

$$EPI = \frac{Power \times T_n}{n} \qquad (1)$$

As expected, the mixed-clock issue queue entails a penalty in latency as well as power consumption. Although the mixed-clock issue queue should not be directly compared with the mixed-clock FIFO, it can be observed that the former is better than the latter in terms of power, and is equivalent in terms of latency. In addition, the smaller overhead of extra logic needed for synchronization makes the mixed-clock issue queue comparable to its fully synchronous counterpart. While it does introduce additional latency, this latency can be hidden most of the time as typically instructions written into the issue window are not issued until a few cycles later.

5. CONCLUSION

In this paper we have proposed a **mixed-clock issue queue design** for high-end, out-of-order superscalar processors, able to sustain different clock rates and speeds for the incoming and outgoing traffic. We have compared and contrasted our implementation with existing synchronous versions of issue queues used stand-alone or in conjunction with mixed-clock FIFOs for inter-domain synchronization. It can be seen that the power consumption of the mixed-clock issue queue is comparable to its synchronous counterpart, and is better than the mixed-clock FIFO. As expected it introduces extra latency when compared with its synchronous counterpart, but it may be hidden by the inherent latency in the pipeline. The presented work can be extended by improving the issue queue by reducing the latency. The features described in this work can be incorporated into an architectural level simulator to compare the different schemes for standard benchmark programs.

6. REFERENCES

[1] R. Canal and A. Gonzalez. A low-complexity issue logic. In *Intl. Conference on SuperComputing (ICS)*, pages 327–335, June 2000.

[2] A. Chandrakasan, W. J. Bowhill, and F. Fox. *Design of High-Performance Microprocessor Circuits*. IEEE Press, New York, 2001.

[3] D. M. Chapiro. *Globally Asynchronous Locally Synchronous Systems*. PhD thesis, Stanford University, 1984.

[4] T. Chelcea and S. M. Nowick. Robust interfaces for mixed timing systems with application to latency-insensitive protocols. In *Design Automation Conference (DAC)*, pages 21–26, 2001.

[5] J. A. Farell and T. C. Fischer. Issue logic for a 600-mhz out-of-order execution microprocessor. *IEEE Journal of Solid-State Circuits.*, 33:707–712, May 1998.

[6] D. Folegnani and A. Gonzalez. Energy-effective issue logic. In *Intl. Symposium on Computer Architecture (ISCA)*, pages 230–239, June 2001.

[7] A. Iyer and D. Marculescu. Power and performance evaluation of globally asynchronous and locally synchronous processors. In *Intl. Symposium on Computer Architecture (ISCA)*, pages 158–168, June 2002.

[8] A. Iyer and D. Marculescu. Power efficiency of multiple clock, multiple voltage cores. In *IEEE/ACM Intl. Conference on Computer-Aided Design (ICCAD)*, pages 379–386, Nov 2002.

[9] J. Leenstra, J. Pille, A. Mueler, W. M. Sauer, and D. F. Wendel. A 1.8 ghz instruction window buffer for an out-of-order microprocessor core. *IEEE Journal of Solid-State Circuits.*, 36:1628–1635, Nov 2001.

[10] J. Muttersbach, T. Villiger, and W. Fitchner. Practical design of globally asynchronous and locally synchronous systems. In *Intl. Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC)*, pages 52–59, 2000.

[11] S. Palacharla, N. Jouppi, and J. E. Smith. Complexity-effective superscalar processors. In *Intl. Symposium on Computer Architecture (ISCA)*, pages 206–218, June 1997.

[12] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Dynamic frequency and voltage control for a multiple clock domain microarchitecture. In *Intl. Symp. on Microarchitecture (MICRO)*, pages 356–367, 2002.

[13] G. Semeraro, G. Magklis, R. Balasubramonian, D. H. Albonesi, S. Dwarkadas, and M. L. Scott. Energy-efficient processor design using multiple clock domains with dynamic voltage and frequency scaling. In *Intl. Symp. on High Performance Computer Architecture (HPCA)*, pages 24–35, 2002.

[14] A. E. Sjogren and C. J. Myers. Interfacing synchronous and asynchronous modules within a high-speed pipeline. *IEEE Tran. on VLSI Systems.*, 8(5):573–583, Oct 2000.

[15] J. E. Smith and G. S. Sohi. The microarchitecture of superscalar processors. In *IEEE*, pages 1609–1624, Dec 1995.

[16] K. Y. Yun and A. E. Dooply. Pausible clocking-based heterogeneous systems. *IEEE Tran. on VLSI Systems.*, 7(4):482–488, Dec 1999.