Modeling and design of asynchronous priority arbiters for on-chip communication systems

J-B. Rigaud^(†), J. Quartana^(*), L. Fesquet^(†), M. Renaudin^(†)

^(†)*TIMA Laboratory, 46 av. Felix Viallet 38031 Grenoble, France*

^(*)*Ph.D. work in co. with STMicroelectronics, AST Grenoble Lab, France*

*jean-baptiste.rigaud@imag.fr, jerome.quartana@st.com, laurent.fesquet@imag.fr,
marc.renaudin@imag.fr*

Abstract: This paper addresses the design of complex arbitration modules, like those required in SoC communication systems. Clock-less, delay-insensitive arbiters are studied in the perspective of making easier and more practical the design of future GALS or GALA SoCs. The paper focuses on high-level modeling and delay-insensitive implementations of fixed and dynamic priority arbiter. Pre-layout simulations show that arbiters which are able to process several hundreds mega requests per second can be designed using the 0.18 μm CMOS process of STMicroelectronics.

Key words: Asynchronous systems, priority arbiters, high-level modeling, CHP language, QDI circuits.

1. INTRODUCTION

One of the critical components of a SoC is the communication system, commonly named on-chip bus. Such an on-chip communication system has to be very flexible to interface in-house and external virtual components, providing high bandwidth, low latency, low power, arbitration mechanisms and routing capabilities.

In a SoC the on-chip bus connects the components to each other and dynamically allocates a path from one block to another. Several blocks running concurrently may require accessing the same resource leading to contentions. In this case, an arbiter is needed to solve the conflicts and to

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: [10.1007/978-0-387-35597-9_40](https://doi.org/10.1007/978-0-387-35597-9_40)

M. Robert et al. (eds.), *SoC Design Methodologies*

© IFIP International Federation for Information Processing 2002

ensure that only one block is accessing the resource. The choice is done with the help of priorities affected to each request.

Several arbitration algorithms were proposed in the past to solve the problem of accessing a unique resource from an arbitrary number of blocks. These algorithms can be classified according to the characteristics of their corresponding hardware implementation. To mention a few, arbitration structures can be distributed or centralized, can be linear like daisy-chain arbiters, or ring-based like token-ring and round-robin arbiters.

Most on-chip communication systems and the arbitration modules they include are today designed with synchronous circuits. In this paper, delay-insensitive asynchronous arbiters are considered, to be part of future on-chip busses of GALA (globally asynchronous locally asynchronous) or GALS (globally asynchronous locally synchronous) SoCs.

In the SoC design perspectives, delay-insensitive arbiters have this main advantage of being hundred-percent reliable (enough time is given to resolve metastability). Today, reliability of on-chip communication systems is a major issue since the increase transaction rates is drastically reducing the so-called Mean Time Before Failure characterizing clocked synchronizers.

As far as power consumption is concerned, such event driven communication/arbitration structures have a minimal electrical activity. Indeed, unlike clocked circuits, power consumption of delay-insensitive asynchronous arbiters is proportional to access rates [6]. Furthermore, delay insensitivity enables the design of fast “long distance” communication busses [5]. Finally yet importantly, such delay-insensitive communication systems are fully autonomous blocks, which can easily be reused in complex SoC architectures as soft, firm or hardware virtual components, hence decreasing design time and complexity.

Based on these motivations, the paper contributes to two fundamental issues: arbitration algorithms are modeled using a high-level description language called CHP (Communicating Hardware Processes), and the delay-insensitive arbiter architectures are derived from these CHP specifications.

Section 2 of the paper describes the global design flow from algorithm modeling to circuit synthesis and electrical simulation. Section 3 and section 4 present both the modeling and the circuit design of respectively fixed and dynamic-priority arbiters. Section 5 reports pre-layout simulation results. The last section concludes this work and mentions the main prospects.

2. DESIGN FLOW

An overview of the global design methodology used to synthesize asynchronous circuits is presented in figure 1. The flow starts from a high-

level modeling using the CHP language (Communicating Hardware Processes). The CHP language, initially proposed by Martin [2], is naturally adopted in this work because i) it includes non deterministic choice structures required to model arbitration, and ii) it is very well suited to model and synthesize delay-insensitive circuits [2][3]. In [4], some new features were added to the CHP language to satisfy simulation requirements.

Delay-insensitive gate-level implementations are derived from the CHP specifications following a method that is not fully automated so far. However, the schematics are obtained following a formal procedure that is beyond the scope of this paper and will be detailed in future communications. It is based on the formal methodology presented in [2].

The design flow used is similar to the one presented in [4]. Gate-level implementations of CHP programs are all described using VHDL gate netlists that are verified by back-annotable logic simulations with timing. Two kinds of libraries are targeted: a standard-cell library and a specific asynchronous-cell library including Muller-gates. For each asynchronous cell, a VHDL functional view (including timing information), a schematic view and a layout view were developed. Then, circuit netlists are imported into the Cadence[®] framework for electrical simulations, placement and routing. The technology used for this study is the 0.18 μm CMOS process from STMicroelectronics.

Figure 1. Design flow overview

All implementations use a four-phase handshaking protocol. Single-rail input channels are used for the fixed-priority arbiter, whereas dual-rail input channels are used for the dynamic-priority arbiter. The request signal associated to the single-rail channel R is denoted R_i . Signals R_{i_0} and R_{i_1} denote the two rails of the dual-rail input channel R . Signal R_o denotes the acknowledge signal of input channel R . Finally, the shared resource is assumed to be accessed through a single-rail channel S .

In the schematics, gates marked with a C are Muller C elements. An R (respectively S) is added when the gate output needs to be set to zero (respectively one) at reset. A "+" is added to specify asymmetric behavior. Input marked with a + only contributes to drive the output to one.

3. FIXED-PRIORITY ARBITER

Fixed-priority arbiters have to choose among input requests with predefined hardware-coded priority values. This section considers several arbitration algorithms: a daisy chain, a particular binary-tree structure and a recent priority-arbiter architecture proposed in [1].

3.1 Fixed-priority arbiter modeling

3.1.1 Daisy-chain arbiter

The CHP program of figure 2 describes a 4-input daisy-chain arbiter implementing a sequential priority scheme. The corresponding design is shown in figure 5. The circuit is by default in sleep mode, and wakes up each time an activity is detected on at least one of the input requests. The probe CHP operator (denoted #) is used to watch activity on inputs and trigger processing (Figure 2, line 1). Priority is simply modeled by sequentially testing the requests. Highest priority input R3 is analyzed first. It is tested active, #R3 statement in figure 2 (line 2). When active, the shared resource S is attributed to R3 by writing to S using the CHP write statement S !. Concurrently (“,” CHP notation), request R3 is granted using the CHP read statement denoted R3 ?. The channel R3 is also tested not active, statement $\neg$ #R3 in figure 2 (line 3). In this case, next channel R2 is analyzed the same way, and so on for R1 and R0.

Note that the stability of guards #R3 and $\neg$ #R3 is not guaranteed [2]. In fact, the request signal level R3 may change while it is evaluated. In this case, a non-deterministic choice denoted "@@" is used in CHP. At the hardware level, it corresponds to using a synchronizer in charge of solving metastability that may occur while deciding whether R3 is zero or one.

```

* [ #( R0 ∨ R1 ∨ R2 ∨ R3 )
→ [ #R3 → S!, R3?
  @@ #R3 → [ #R2 → S!, R2?
             @@ #R2 → [ #R1 → S!, R1?
                       @@ #R1 → S!, R0?
  ] ] ] ]

```

Figure 2. Daisy-chain arbiter CHP model

3.1.2 Binary-tree arbiter

Another interesting arbitration structure, a binary-tree arbiter, is described in figure 3 and the corresponding gate level implementation presented in figure 6. The program clearly shows the tree structure of the selection process.

```

* [ # ( R0 ∨ R1 ∨ R2 ∨ R3 )
→ [ [ [ #R3 → s1 := 2          (1)
    @ @ [ #R3 → [ #R2 → s1 := 1
        @ @ [ #R2 → s1 := 0
    ] ] ],
[ [ #R1 → s0 := 2          (2)
  @ @ [ #R1 → [ #R0 → s0 := 1
      @ @ [ #R0 → s0 := 0
  ] ] ] ];
[ s1 = 2 → R3 ? , S!
@ s1 = 1 → R2 ? , S!
@ s1 = 0 → [ s0 = 2 → R1 ? , S!
            @ s0 = 1 → R0 ? , S!
] ] ]

```

Figure 3. Tree arbiter CHP model

Here again, line one of the program is used to sense input activities. Part one and part two of the program (respectively labeled 1 and 2 in figure 3), concurrently solve R3/R2 and R1/R0 contentions in the first stage. Priority of R3 over R2 (respectively R1 over R0) is modeled by a two-stage linear structure first checking R3 (respectively R1) and then R2 (respectively R0). Then, the CHP sequential operator “;” is used to specify that the concurrent parts of the first stage have to complete before stage 2 can process (label 3 in figure 3). In this last stage, deterministic choices are used since variables s0 and s1 are stable (“@” notation). According to s0 and s1 values, the winning input is granted and the shared resource is concurrently accessed.

3.1.3 Parallel-request-sampler priority-arbiter

Recently, Bystrov, Kinniment and Yakovlev introduced in [1] an enhanced version of priority arbiters that decouples request signals sampling (synchronizer module) and contention solving (priority module). This structure outperforms previously proposed structures in terms of complexity and latency. Moreover, unlike the daisy chain and the tree arbiters, this new arbiter structure is strongly modular, and priorities can be modified by only redesigning the priority module. However, in [1] the design of this priority arbiter family was very intuitive and performed by hand.

Figure 4 gives a formal CHP specification of a 4-way parallel-request-sampler fixed-priority arbiter (request-sampler FPA) like those proposed in [1]. Line 1 is used to sense request activities. Then, four identical subparts model the concurrent sampling of the request signals (labeled synchronizer 3 to 0). Variables s0 through s3 are used to store the samples, which are then exploited by the fixed-priority module to figure out which input has to be elected (labeled fixed-priority module).

3.2.2 Binary-tree arbiter

The binary-tree arbiter structure is presented in figure 6. Two pairs of cascaded synchronizers concurrently sample and analyze the input requests according to a priority order. Highest priority is R3 over R2 (respectively R1 over R0).

Then a delay-insensitive logic implements a deterministic comparison in order to select the input to serve.

Figure 6. Tree arbiter architecture

3.2.3 Parallel-request-sampler FPA

The architecture of the 4-way parallel-request-sampler FPA of [1] is described in figure 7. The four synchronizers operate concurrently to sample the input request signals. The fixed-priority module determines the input request to grant according to hardware-coded priorities.

Figure 7. Parallel-request-sampler fixed-priority arbiter architecture of [1]

4. DYNAMIC-PRIORITY ARBITER

In a dynamic-priority arbiter, each input channel is carrying requests as well as priority values. The dynamic-priority arbiter performs priority comparisons of active requests and grants the one with the highest value. The input having the highest index is by convention selected in case of identical priority levels.

According to the performances of the enhanced version of priority arbiters presented in [1], the structure of the parallel-request-sampler arbiter is implemented in this section with a dynamic-priority resolution algorithm. The time overload to resolve dynamic contention is compared in §5 with the fixed-priority version presented in figure 4.

4.1 Dynamic-priority arbiter modeling

Figure 8 presents the program of a 4-way parallel-request-sampler dynamic-priority arbiter (request-sampler DPA). Each input R_i is a dual rail channel, encoding the request signal together with a priority value ranging from 0 (lower) to 1 (higher). Only two priority-levels are considered for the sake of simplicity. This can of course easily be extended and adapted to the requirements of real applications.

The first stage is very similar to that of the fixed-priority version of the arbiter (subparts labeled synchronizer 3 to 0). The priority module is much more complicated because it has to perform priority-level comparisons of competing inputs. The program proposed in figure 8 describes a two-stage comparator structure, but any other structure could be modeled leading to different tradeoffs in terms of complexity, speed and power.

The first stage concurrently analyses pairs of inputs and if necessary concurrently performs priority comparisons. Variable $v1_0$ indicates: no request occurred (value 0), request $R0$ occurred (value 1) or request $R1$ occurred (value 2). The request with the highest priority is passed to the following stage. Variable $v3_2$ plays the same role for inputs $R3$ and $R2$. Variables $v1_0$ and $v3_2$ cannot be both zero since priority comparators are triggered on requests' activity. If only one request comes out stage one, the corresponding input is immediately granted (label "1st stage acknowledgment" in figure 8). If two requests come out stage one, then another comparison has to take place. A multiplexer identifies the two requests in competition and sends their priority values to the second stage comparator (subpart labeled "2nd stage: 2-way priority comparator" in figure 8). Finally, the issue of the comparison together with the couple of the inputs in contention is used to figure out which input is the winner (subpart labeled 2nd stage acknowledgment in figure 8).


```

* [ #( R0 ∨ R1 ∨ R2 ∨ R3 )          -- loop control
-- synchronizer 3
→ [ [   #R3 → s3 := 1 , [   #R3 = 0 → r3 := 0
      @ #R3 = 1 → r3 := 1 ]
    @@ #R3 → s3 := 0
  ],
-- synchronizer 2
[   #R2 → s2 := 1 , [   #R2 = 0 → r2 := 0
      @ #R2 = 1 → r2 := 1 ]
  @@ #R2 → s2 := 0
],
-- synchronizer 1
[   #R1 → s1 := 1 , [   #R1 = 0 → r1 := 0
      @ #R1 = 1 → r1 := 1 ]
  @@ #R1 → s1 := 0
],
-- synchronizer 0
[   #R0 → s0 := 1 , [   #R0 = 0 → r0 := 0
      @ #R0 = 1 → r0 := 1 ]
  @@ #R0 → s0 := 0
] ];
-- dynamic-priority module
-- 1st stage: 2-way request analysis and priority
comparators
[ [ (s0, s1) = (0, 0) → v1_0 := 0
  @ (s0, s1) = (1, 0) → v1_0 := 1
  @ (s0, s1) = (0, 1) → v1_0 := 2
  @ (s0, s1) = (1, 1) → [ r1 ≥ r0 → v1_0 := 2
                        @ r1 < r0 → v1_0 := 1 ]
],
[ (s2, s3) = (0, 0) → v3_2 := 0
  @ (s2, s3) = (1, 0) → v3_2 := 1
  @ (s2, s3) = (0, 1) → v3_2 := 2
  @ (s2, s3) = (1, 1) → [ r3 ≥ r2 → v3_2 := 2
                        @ r3 < r2 → v3_2 := 1 ]
] ];

-- 1st stage acknowledgment
[ (v1_0, v3_2) = (1, 0) → comp := 0 , R0? , S!
  @ (v1_0, v3_2) = (2, 0) → comp := 0 , R1? , S!
  @ (v1_0, v3_2) = (0, 1) → comp := 0 , R2? , S!
  @ (v1_0, v3_2) = (0, 2) → comp := 0 , R3? , S!
-- multiplexer
@ (v1_0, v3_2) = (1, 1) → comp := 1 , C0 := r0, C1 := r2
@ (v1_0, v3_2) = (1, 2) → comp := 1 , C0 := r0, C1 := r3
@ (v1_0, v3_2) = (2, 1) → comp := 1 , C0 := r1, C1 := r2
@ (v1_0, v3_2) = (2, 2) → comp := 1 , C0 := r1, C1 := r3
];
-- 2nd stage: 2-way priority comparator
[ comp = 1 → [ C1 ≥ C0 → out := 1
              @ C1 < C0 → out := 0 ]
  @ comp = 0 → skip
];
-- 2nd stage acknowledgment
[ comp = 1 → [ out = 1 → [ (v1_0, v3_2) = (1, 1) → R2? , S!
                          @ (v1_0, v3_2) = (1, 2) → R3? , S!
                          @ (v1_0, v3_2) = (2, 1) → R2? , S!
                          @ (v1_0, v3_2) = (2, 2) → R3? , S!
                        ]
    @ out = 0 → [ (v1_0, v3_2) = (1, 1) → R0? , S!
                  @ (v1_0, v3_2) = (1, 2) → R0? , S!
                  @ (v1_0, v3_2) = (2, 1) → R1? , S!
                  @ (v1_0, v3_2) = (2, 2) → R1? , S!
                ]
  @ comp = 0 → skip
] ];

```

Figure 8. Parallel-request-sampler dynamic-priority arbiter CHP model of [1]

4.2 Dynamic-priority arbiter design

Figure 9 presents the architecture of the 4-way request-sampler DPA specified by the CHP program of figure 8.

The two 2-way request analyzer & priority comparator blocks concurrently compare two pairs of requests. Each of them delivers the priority-holder request. Details of the gate-level structures are given in figure 10. It can be seen that the 2-way priority comparator is only activated when both input requests are active. The first stage of acknowledgment detects if a unique request is active at the output of the two 2-way request analyzers and priority comparators. In that favorable case, the corresponding input is immediately granted and the shared resource accessed, preventing from useless power consumption and latency.

The gate-level detailed blocks for fixed and dynamic parallel-request-sampler arbiters are available in [7].

Figure 9. Parallel-request-sampler dynamic-priority arbiter architecture of [1]

Figure 10. 2-way request analyzer and priority comparator

5. SIMULATION RESULTS

Table 1 summarizes latency and cycle time values obtained by electrical simulations of the gate-level netlists. The arbiters were simulated with two input-sets to measure best and worst case latencies.

For the daisy-chain arbiter, best and worst case latencies are respectively when R3 requests and when only R0 requests among the four inputs. For the tree arbiter, the best case corresponds to R1 active only and the worst case to R2 and R0 active. The parallel-request-sampler FPA best case occurs when R0 is the unique active request. The worst case occurs when only R1 is active. The dynamic-priority arbiter best and worst-case latencies were explained in Section 4.

		Fwd latency	Cycle time	Throughput
Daisy chain arbiter	Best case	1.06 ns	1.66 ns	600 Mhz
	Worst case	1.42 ns	2.28 ns	440 Mhz
Tree arbiter	Best case	1.32 ns	2.15 ns	465 Mhz
	Worst case	1.52 ns	2.51 ns	400 Mhz
Request-sampler FPA	Best case	1.26 ns	2.01 ns	500 Mhz
	Worst case	1.47 ns	2.46 ns	405 Mhz
Request-sampler DPA	Best case	1.58 ns	2.46 ns	405 Mhz
	Worst case	2.53 ns	4.54 ns	220 Mhz

Table 1. Latency and cycle-time of simulated arbiters

Fixed-priority arbiters The daisy-chain arbiter responds here the fastest. It outperforms other structures in the best case of latency. However, the significant decrease of performance in the worst case confirms that the sequential sampling of the input requests strongly penalizes this structure. Furthermore, including wire delays in the simulations would heavily increase the latency. Such a linear architecture is not sufficient to cover the range of real applications, in which the number of peripherals is higher and in which the relative frequencies of the requesters have to be considered.

The binary-tree and the parallel-request-sampler arbiter present a more balanced structure, which exhibits similar delays for best and worst cases. When the number of inputs increases, the number of synchronizers increases linearly, whereas the fixed-priority logic complexity increases in $O(\log_2)$.

The advantages of the request-sampler FPA over the tree arbiter are: a simpler fixed-priority logic block and a one-level stage request sampler.

Dynamic-priority arbiters The dynamic-priority arbiter can sustain a request rate of about 405 MHz and 220 MHz. The modularity of the dynamic-priority arbiter enables an easy reconfiguration of the priority algorithm. Several priority levels offer a high degree of freedom for system-level designs.

Optimizations Using more aggressive optimizations based on fast handshaking components [8] would increase the speed. Another way to reduce the arbiter latency is to properly assign the arbiter inputs to requesters according to their respective priorities and request frequencies. For example, highest-priority requests should be connected to lowest-latency arbiter inputs in order to answer highest priority requests as quick as possible. Another strategy would be to connect lowest latency arbiter inputs to highest frequency requests in order to optimize average speed and power-consumption simultaneously.

6. CONCLUSION

In this paper, it is shown that arbiters can be modeled using the high-level language CHP and their corresponding delay-insensitive implementations derived. Pre-layout electrical simulations report that delay-insensitive priority arbiters processing several million requests per second can be designed using an up-to-date CMOS technology.

This work actually proves that it is today possible to cleanly and formally model and design delay-insensitive arbiter modules that are reliable, modular and fast. It also defines the fundamentals of an automated synthesis process devoted to arbiters. Finally, it constitutes an enabling factor for the asynchronous technology to be increasingly adopted in the design of SoCs.

Prospective works will be focused on the automation of the synthesis process and the improvement of arbiter architecture and circuit performances. "n initiators to p receivers" fixed or dynamic priority routers will also be investigated to address the design of complex on-chip routing systems.

7. REFERENCES

- [1] A. Bystrov, D. J. Kinniment, A. Yakovlev, "Priority Arbiters", in "International Symposium on Advanced Research in Asynchronous Circuits and Systems" (ASYNC), Eilat, Israel, April 2000, pp. 128-137.
- [2] A.J. Martin, "Synthesis of Asynchronous VLSI Circuits", Internal Report, Caltech-CS_TR-93-28, California Institute of Technology, Pasadena, 1993.
- [3] J-B. Rigaud, M. Renaudin, "Modeling and design/synthesis of arbitration problems", AINT'2000, Proceedings of the Asynchronous Interfaces: Tools, Techniques and Implementations Work-shop, TU Delft, The Netherlands, July 19-20th 2000, pp. 121-128.
- [4] M. Renaudin, P. Vivet, F. Robin, "A Design Framework for Asynchronous/Synchronous Circuits Based on CHP to HDL Translation", in "International Symposium on Advanced Research in Asynchronous. Circuits and Systems", Barcelona, Spain, pp 135-144, April 19-21, 1999.
- [5] W.J. Bainbridge, S.B. Furber, "Delay Insensitive System-On-Chip Interconnect using 1-of-4 Data Encoding", in "International Symposium on Advanced Research in Asynchronous Circuits and Systems" (ASYNC), Salt Lake City, USA, March 11-14, pp 118-126, 2001.
- [6] C. Piguet, M. Renaudin, T. Omnes, "Special Session on Low-power SoCs", Design Automation and Test in Europe (DATE), Munich, Germany, March 2001, pp. 488-494.
- [7] J-B. Rigaud, J. Quartana, L. Fesquet, M. Renaudin, "Modeling and design of asynchronous priority arbiters for on-chip communications systems", Proceedings of VLSI-Soc'01, 11th IFIP International Conference on VLSI, Montpellier, France, December 3-5 2001, pp. 424-429.
- [8] Andrew Lines, "Pipelined Asynchronous Circuits", Master Thesis, Caltech-CS-TR-95-21, 1995, revised June 1998.