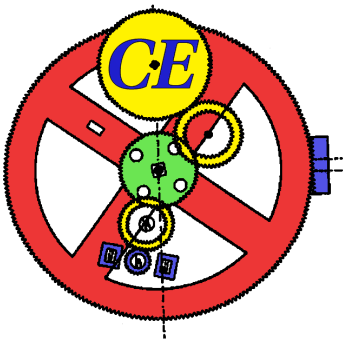


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2010

MSc THESIS

Low Power Evaluation for Arbitration and MPSoC

Ruud Benjaminsen

CE-MS-2010-14

Abstract

This thesis presents a power analysis for various arbitration schemes. We chose variations on the round-robin and time-division multiplexing schemes as our arbiter configurations. The arbiters were implemented with 90 nm low-power standard cell libraries from TSMC, and gate-level power extraction was performed. Clock-gating was optionally introduced during synthesis. We then contrasted the power dissipation for the different arbiters and showed that no single arbitration scheme performs well in terms of power dissipation under all load conditions. We also analyzed why the power dissipation curve of a round-robin arbiter shows a point of maximum inflection. This thesis implements also a multiprocessor system-on-chip design. Such designs can offer significant power savings over traditional uniprocessor designs. We analyzed the power of such a system, and showed how it can be constructed in both hardware and software.

TU Delft
Delft University of Technology

Faculty of Electrical Engineering, Mathematics and Computer Science

Low Power Evaluation for Arbitration and MPSoC

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Ruud Benjaminsen
born in Hulst, The Netherlands

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Low Power Evaluation for Arbitration and MPSoC

by Ruud Benjaminsen

Abstract

This thesis presents a power analysis for various arbitration schemes. We chose variations on the round-robin and time-division multiplexing schemes as our arbiter configurations. The arbiters were implemented with 90 nm low-power standard cell libraries from TSMC, and gate-level power extraction was performed. Clock-gating was optionally introduced during synthesis. We then contrasted the power dissipation for the different arbiters and showed that no single arbitration scheme performs well in terms of power dissipation under all load conditions. We also analyzed why the power dissipation curve of a round-robin arbiter shows a point of maximum inflection. This thesis implements also a multiprocessor system-on-chip design. Such designs can offer significant power savings over traditional uniprocessor designs. We analyzed the power of such a system, and showed how it can be constructed in both hardware and software.

Laboratory : Computer Engineering
Codenumber : CE-MS-2010-14

Committee Members :

Advisor:	Kees Goossens, CE, TU Delft
Advisor:	Filipa Duarte, Holst Centre/IMEC-NL
Chairperson:	Koen Bertels, CE, TU Delft
Member:	Ioannis Sourdis, CE, TU Delft
Member:	Nick van der Meijs, CAS, TU Delft
Member:	Jos Huisken, Holst Centre/IMEC-NL

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would like to thank Holst Centre/IMEC-NL for opening the world of low power to me. This is an interesting field, where real technology improvements can still be made. My thesis could not have been made possible without the keen guidance of Filipa Duarte and Jos Huisken at IMEC-NL, and I am thankful to Kees Goossens for acting as my thesis advisor from TU Delft. Furthermore, I would like to thank everyone in IMEC's ULP-DSP group for giving me advice from-to-time and giving me an enjoyable experience in Eindhoven.

Ruud Benjaminsen
Delft, The Netherlands
June 14, 2010

Motivation

The Ultra Low Power DSP Group at Holst Centre/IMEC-NL carries out active research towards low power signal processing techniques, architectures and devices. This research is necessary to reduce the power dissipation in wireless sensor nodes, and increase their autonomy. There is a focus on targeting applications from the domain of biomedical signal processing and wireless baseband processing.

A typical node consists of sensors, a power manager, radio and digital processing subsystem. The latter includes processors, memories and an on-chip communication infrastructure. Therefore, an effort is required to study power efficient ways to develop on-chip communication infrastructures, which not only includes identifying the basic components required, but also ways to connect them together. The basic components used in on-chip communication infrastructures include first-in first-out queues (FIFOs), arbiters, interrupt controllers, semaphores, etc. Connections between external devices (like sensors, programming or debug interfaces), memories and processors are made using point-to-point links, buses or network-on-chips.

As mentioned, one the components identified for on-chip communication infrastructures is an arbiter, which provides a conflict resolution scheme for when multiple contenders try to access a shared resource. An arbiter is needed, because the on-chip communication infrastructure is likely to be shared by different processing and storage elements, even more so when considering multiprocessor system-on-chip (MPSoC) designs. This is depicted graphically in the figure below, where different processing elements (PEs) are trying to access a shared resource simultaneously through the on-chip communication infrastructure:

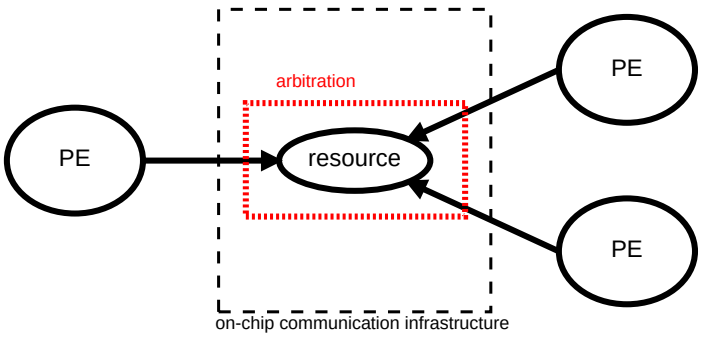


Figure 0.1: On-Chip Communication Infrastructure Arbitration

By analyzing the power characteristics of a variety of arbiters, an insight is gained into the power trade-offs between them, and the consequences this has for on-chip communication infrastructures.

Previous work in the Ultra Low Power DSP Group has also concentrated on application-specific uniprocessor design [9]. For low power applications, (heterogeneous)

MPSoC architectures are however preferred, as each processor can be optimized for a specific task, minimizing communication between them by exploiting data-locality and concurrency. There is a need to understand how the move towards such systems affects the hardware implementation of the on-chip communication infrastructure, and how such infrastructure can be supported in software. Therefore, a MPSoC system has been realized, considering both the hardware and software side of such system, and inferences are made from the extracted power dissipation measurements of the design.

Part of the results of this thesis have been presented at the ProRISC 2009 conference, in Veldhoven, the Netherlands [3].

Thesis Organization

This thesis is organized as follows. The first section (Part I) deals with the design and power analysis of various arbiter configurations. It is composed of an introduction and related work chapter (Chapters 1 and 2), while Chapter 3 present the functional description of arbiters. The hardware implementation of the arbiter configurations chosen for further power analysis is presented in Chapter 4, which includes a description of the test setup for gate-level power extraction. Chapter 5 presents the results from synthesis and power dissipation figures for the arbiters and Chapter 6 has some discussion on these power figures, as well as a comparison. Part I is concluded with conclusions in Chapter 7.

Part II details the implementation of a multiprocessor system-on-chip system. It also includes an introduction and related work section (Chapters 8 and 9). Chapter 10 discusses multiprocessor design, while Chapter 11 provides an overview of the on-chip multiprocessor implementation and test setup. Results are presented in Chapter 12, and conclusions are presented in Chapter 13.

Finally, Part III includes a chapter for future work considerations.

Part I

Arbiters

1 Introduction

In order to design power-efficient on-chip communication infrastructures, there is a need to both look at the components needed for such a communication infrastructure, as well as on ways one can tie those components together. The on-chip communication infrastructure is likely to be shared by various processing and storage elements, even more so when considering multiprocessor system-on-chip (MPSoC) designs. Access to resources within the on-chip communication infrastructure is thus not limited to simply one agent, and as such, a contention resolution scheme is necessary in the form of an arbiter. This is graphically depicted in Figure 1.1, where multiple processing elements (PEs) are contending for a shared resource reachable through an on-chip communication infrastructure.

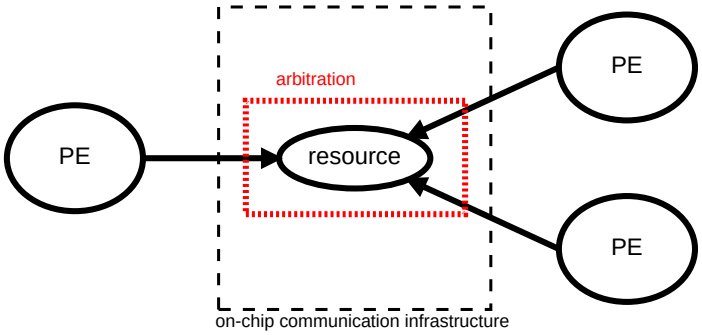


Figure 1.1: On-Chip Communication Infrastructure Arbitration

Traditional small-scale system-on-chip (SoC) designs have a shared-medium architecture. In shared-medium architectures, only single communication instances are supported by the communication infrastructure. Its most common manifestation is in the form of a backplane bus, supporting a couple of master devices (processors) that connect to passive slave elements. Temporary ownership of the shared-medium (bus) can be given to any master willing to communicate with a slave, provided there is contention resolution scheme for when multiple masters try to access the shared-medium at the same time: a bus arbiter. But the presence of the contention resolution indicates performance loss in communication, in the form of extra control actions. And although easily implemented and having low hardware overhead, shared-medium architectures are not scalable. Furthermore, because the shared-medium connects all masters and slaves together in a single transmission medium, these architectures can be power inefficient, because masters and slaves not taking part in a communication instance will still receive the transmission data of those that do take part [2].

Improved scalability over shared-medium architectures comes by using direct and indirect networks. Direct networks are typically utilized in homogeneous SoC designs,

which consist of processing modules being replicated and placed in a regular topology. These modules are connected together with a crossbar switch, which can be reconfigured to offer different connection patterns between modules. Still, arbitration is needed when the connection patterns required by one processing module conflict with those of another module. As an alternative, indirect networks that switch communication packets can be used: a network-on-chip (NoC). These are highly scalable, have well-defined performance metrics and offer separation of communication from computation. The network consists of routers, with some of the routers connected to processing modules. Each router has a crossbar switch that connects input ports with output ports, and arbitration is needed whenever the same output port is requested by a multitude of incoming packets [2] [39].

As such, arbitration takes on an important part of any on-chip communication infrastructure. By analyzing the power characteristics of a variety of arbiters, we gain insight into the power trade-offs between them and the consequences this has for on-chip communication infrastructures.

2 Related Work

Related work on arbiters has mostly focused on their performance and utility in on-chip communication buses and network-on-chip router switches.

The authors of [30] analyzed round-robin, time-division multiplexing and slot reservation arbitration policies in bus arbiters, and showed that different workloads give rise to different optimal contention resolution schemes. By simulation of mutually dependent tasks, independent tasks and pipelined tasks on a multiprocessor platform, they proposed a set of principles for bus-based on-chip communication architectures. Firstly, they show that there is no optimal arbitration policy irrespective of the task to be executed. Tasks that are highly computation dependent benefit from a different arbitration policy than tasks that are more communication centric. Secondly, the authors warn that high-level software primitives can not always be matched efficiently to the underlying hardware platform. Lastly, they note that although commercial bus protocols that offer contention-resolution have some flexibility for the optimization of performance, not all arbitration policies can be integrated easily within such a commercial protocol. In applications where predictability is key (e.g. real-time computing), non-determinant arbitration policies that usually perform well otherwise might become unsuitable for such applications.

Many arbitration schemes cannot meet real-time requirements and bandwidth requirements at the same time. In [5] and [24], the RT_lottery and RB_lottery schemes are presented that meet hard real-time requirements and perform well in bandwidth allocation within an on-chip bus, and use the lottery scheme presented in [22]. In the RT and RB lottery schemes, the weight of each bus contender is continuously finetuned according to bandwidth requirements. Only at really high bus workloads do these schemes fail to meet (hard) real-time requirements.

In [21], the power dissipation of an AMBA on-chip communication bus was estimated in 150 nm low-power standard cell libraries from NEC, where the fixed-priority arbiter contributed 18% to the total power of 12 mW. The authors further looked at possible ways to reduce power in such a commercial communication architecture, and found that their evaluated techniques only helped to reduce the power dissipation in seperate segments (logic, bus lines, bus interfaces, etc.) of the communication architecture.

The authors of [6] simulated the performance of different AMBA bus arbitration schemes and their switching activity, and estimated that the bus switching activity could be reduced by 22% when using a short job first arbitration scheme.

An energy-efficient network-on-chip was designed in [23], which included a crossbar switch based upon mux-tree round-robin arbiters. A 8-input port implementation at 100 MHz in 180 nm CMOS was found to dissipate 136 μW of power at 50% of load.

The Orion power and performance simulator for on-chip interconnects includes models for various arbitration schemes [19]. Version 2.0 of the simulator was validated within

7% and 11% of the total power dissipation in the Intel Teraflops Research Chip and Intel Scalable Communications Core, respectively. The authors claim that the deviations in power dissipation with the Teraflops core could be attributed to the use of different standard cell libraries, memory buffers and arbitration scheme in the model used for Orion. However, version 2.0 significantly improved the accuracy of the power models over the 1.0 version, which had a 85% difference in estimated power dissipation compared with the 80-core Teraflops chip.

Arbiter Functional Description

3

Arbiters are needed in order to resolve conflicts arising from multiple contenders trying to access a shared resource simultaneously. These resources can be storage elements (memories), buses, buffers, data channels, etc. Different conflict resolution schemes are possible, each with its advantages and disadvantages. One import measure for arbiters has to do with *fairness*: how can we make sure that one contender cannot lock out another contender's ability to access the shared resource. Another issue has to do with *timing*, or how long contenders are granted access to a resource before a new arbitration is performed.

3.1 Arbitration Fairness

There are several ways on how to look at the concept of *fairness* for arbiters. The first way has to do with starvation (deadlock): how can we make sure that each contender will eventually have access to the shared resource. An arbitration scheme without starvation has a worst-case time period in which a shared resource can be accessed by a contender.

Secondly, there can be a notion of fairness that stipulates that each contender shall have an equal amount of relative accesses to the shared resource. When looking over a long number of performed arbitrations, each contender will than have obtained access to the shared resource according to the number of requests it has put in.

3.2 Arbitration Timing

The requirements for arbitrating *timing* is largely determined by external factors. Some software applications have real-time requirements that need access to a shared resource ever so often, or require that once access to a resource has been granted, a fixed time period (a number of clock cycles perhaps) is needed before the resource can be accessed again by other contenders, which results in grants with variable time duration.

3.3 Arbitration Schemes

We will look at synchronous arbitration schemes that perform arbitration every clock cycle, although there is no reason why they cannot be extended to grant access to shared resources for longer time periods, if application or external requirements require it. Since the round-robin scheme and time-division multiplexing schemes both have a regular structure and provide each contender with equal opportunity to access a shared resource, we will use these schemes later on for a more detailed study in order to understand their power dissipation characteristics. For a short overview of the power dissipation in

CMOS digital circuits and techniques for low-power design, such as power-gating and clock-gating, the reader is referred to Appendix A.

The functionality of an arbiter can be described by a number of contender request lines (R), the corresponding priorities for each contender (P), and a couple of output grant lines (G) which denote which contender was granted access [8].

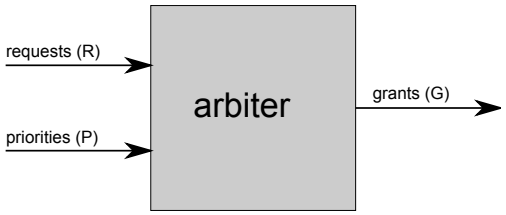


Figure 3.1: Arbiter Functionality

3.3.1 Fixed Priority

The simplest arbitration scheme is one where each contender has a fixed priority (FP) for accessing the shared resource, which is consequently also it biggest drawback. If a high priority contender is continously trying to access the resource, all lower priority contenders are blocked and are not be able to access the resource at all, a situation described previously as *starvation*. Only when priorities are changed between arbitration cycles one can expect a resolution scheme which is fair, although this is not a minimum requirement. Because the hardware implementation of a fixed priority arbiter is quite straightforward, and contender priorities do not have to updated between arbitration cycles, it will also be the scheme with the lowest power dissipation, although there might be few occassions when such an arbiter can actually be used in an on-chip communication infrastructure. Because of this specificity, we will not take the fixed priority arbiter into account for further analysis. For completeness, we do present a descripion of the fixed priority arbiter below (Algorithm 1).

3.3.2 Round-Robin

A round-robin (RR) arbitration scheme operates on the principle that the contender which was granted access to the shared resource, should have the lowest priority in the next round of arbitration. The contenders can be pictured as being placed in a ring, where the priority of each contender decreases linearly from the contender with highest priority (Algorithm 2). A large part of the dynamic power dissipation of this arbiter will be due to the updating of contender priorities between arbitration cycles.

In terms of performance (latency) - in the best case - a contender which puts in a request will be granted access immediately. The worst case situation happens when the lowest priority contender puts in a request, but all other contenders as well. The lowest priority contender will then have to wait before the higher priority contenders have been served, which can take as long as $n - 1$ arbitration cycles.

Algorithm 1 Fixed Priority Arbiter

Require: A sequence of contender requests, $R_i^{(t)}$; A corresponding initial set of contender priorities, $P_i^{(1)} = \{n-1, n-2, \ldots, 0\}$; $0 \leq i \leq n-1$, where n is the number of contenders, and $1 \leq t < \infty$, where t is a time index

Ensure: A corresponding sequence of contender grants, $G_i^{(t)}$

```
t ← 1
loop
  {assume no grant initially}
  for i = 0 to n - 1 do
    G_i^(t) ← false
  end for
  {start scanning for a contender request from contender with highest priority}
  i ← 0, j ← 0
  while j = 0 and i < n do
    if R_i^(t) = true then
      j ← i
    end if
    i ← i + 1
  end while
  {update grants}
  if R_j^(t) = true then
    G_j^(t) ← true
  end if
  {update priorities}
  for i = 0 to n − 1 do
    P_i^(t+1) ← P_i^(t)
  end for
  t ← t + 1
end loop
```

3.3.3 Time-Division Multiplexing

A time-division multiplexing (TDM) scheme can also be used to resolve access conflicts: a fixed cycle - or time slot - is assigned to each contender in which it can try to access the shared resource. If a contender does not put in a request in its time slot, the slot is wasted for other contenders trying to access the resource (Algorithm 3). A large part of the dynamic power dissipation of this arbiter will be due to the updating of contender priorities between arbitration cycles, but less so than for the RR arbiter. In a TDM arbiter, the priority is only shifted to the next contender, although this happens even for arbitration cycles where a contender does not put in a request in its assigned time slot.

In terms of performance, the TDM arbiter will do as well (or badly) as the RR arbiter in the worst-case and best-case. In the best-case for TDM, a contender puts in a request in the same cycle as it has been assigned a time slot. The worst-case happens when a

Algorithm 2 Round-Robin Arbiter

Require: A sequence of contender requests, $R_i^{(t)}$; A corresponding initial set of contender priorities, $P_i^{(1)} = \{n-1, n-2, \ldots, 0\}$; $0 \leq i \leq n-1$, where n is the number of contenders, and $1 \leq t < \infty$, where t is a time index

Ensure: A corresponding sequence of contender grants, $G_i^{(t)}$

```
t ← 1
loop
  {assume no grant initially}
  for i = 0 to n − 1 do
    G_i^(t) ← false
  end for
  {find contender with highest priority}
  j ← 0
  for i = 1 to n − 1 do
    if P_i^(t) > P_j^(t) then
      j ← i
    end if
  end for
  {start scanning for contender request from contender with highest priority}
  i ← 0, k ← j
  while R_j^(t) ≠ true and i < n do
    i ← i + 1
    j ← (j + i) mod n
  end while
  {update grants and priorities}
  if R_j^(t) = true then
    G_j^(t) ← true
    for i = 0 to n − 1 do
      P_i^(t+1) ← (P_i^(t) + 1 + j − k) mod n
    end for
  end if
  t ← t + 1
end loop
```

contender puts in a request in the arbitration cycle just after it was assigned a time slot, but it is independent of the requests put in by the remainder of the contenders.

3.3.4 Mixed RR / TDM Configurations

We also looked at mixed configurations of the RR and TDM arbiter, to see if these arbiters can perform better in terms of power dissipation over the regular schemes, and under which conditions.

Algorithm 3 Time-Division Multiplexing Arbiter

Require: A sequence of contender requests, $R_i^{(t)}$; A corresponding initial set of contender priorities, $P_i^{(1)} = \{n-1, n-2, \ldots, 0\}$; $0 \leq i \leq n-1$, where n is the number of contenders, and $1 \leq t < \infty$, where t is a time index

Ensure: A corresponding sequence of contender grants, $G_i^{(t)}$

```
t ← 1
loop
  {assume no grant initially}
  for i = 0 to n − 1 do
    G_i^(t) ← false
  end for
  {find contender with highest priority}
  j ← 0
  for i = 1 to n − 1 do
    if P_i^(t) > P_j^(t) then
      j ← i
    end if
  end for
  {start scanning for contender request from contender with highest priority}
  i ← 0, k ← j
  while R_j^(t) ≠ true and i < n do
    i ← i + 1
    j ← (j + i) mod n
  end while
  {update grants and priorities}
  if R_j^(t) = true then
    G_j^(t) ← true
    for i = 0 to n − 1 do
      P_i^(t+1) ← (P_i^(t) + 1) mod n
    end for
  end if
  t ← t + 1
end loop
```

3.3.5 TDM + RR Arbiter

The TDM + RR arbiter configuration uses a two-step arbitration scheme. Since in TDM a time slot is wasted when no contender puts in a request in its assigned time slot, we can still do RR arbitration amongst the other contenders. When the contenders are putting in a lot of requests, the arbitration will then be largely limited to the first step (TDM), while at a reduced number of contender requests, the contribution of the second step (RR) will increase.

3.3.6 TDM + subset(RR) Arbiter

The average-case performance of the TDM arbiter can be improved by not fixed a time slot for a single contender, but for multiple contenders (a frame of contenders). This should give the arbiter more opportunities to serve contender requests. Within a frame, we can choose to perform RR arbitration between the contenders assigned to the frame, which will reduce the number of wasted time slots we have seen in the TDM arbiter. In terms of best-case and worst-case performance, this configuration still has the same latency characteristics of the TMD and RR arbiters.

3.3.7 Least-Recently Used

In a least-recently used (LRU) scheme the contender which waited the longest before trying to access the shared resource will win the arbitration. Because this scheme requires the arbiter to keep track of relative priorities between each contender, it is only practical for a small number of contenders. The updating of priorities in each round of arbitration will incur a large power dissipation overhead. As such, we will not further analyze this arbiter configuration.

Hardware Implementation & Test Setup

4

The different variations of the RR & TDM arbitration schemes are implemented in hardware and used for further power analysis.

4.1 VHDL implementation

We describe the different arbiter configurations hardware description language VHDL.

For the RR arbiter, we keep track of the contender priorities with a one-hot encoded state vector, where the contender with highest priority is one corresponding with the sole bit set in the state vector. We follow the architecture given in [8], which works as follows: Each arbitration cycle, the state vector will be updated according to the contender which received a grant from the arbiter. If no grant is given (because there are no contender input requests), the priorities remain the same. This also denotes the condition necessary for succesful clock-gating of the design. The arbiter issues grants in the cycle next to the one in which it arbitrated between the contenders, but the contender which received the grant is free to issue another request in the cycle in was granted its previous request.

For the TDM arbiter, there is no need to keep track of priorities, instead a counter is used to keep track of which contender is able to use the time slot, although a shift-register implementation is also possible.

In the mixed TDM and RR arbitration combinations, additional multiplexing work has to be done so that the right set of contenders is arbitrated upon in the each step of the arbitration.

The tool flow used for the physical implementation of the arbiter configurations is described in Appendix B.

For the hardware implementation of the arbiters we used 90 nm low-power high-threshold voltage standard cell libraries from TSMC [36]. We synthesized the designs with a target clock frequency of 100 MHz, with optional clock-gating insertion. The high-threshold voltage standard cell libraries will reduce the leakage power of the designs, although the effect of clock-gating on the power dissipation is highly data dependent. The generated hardware for a RR arbiter of 6 contenders after RTL synthesis with optional clock-gating can be found in Figures 4.1 and 4.2, respectively.

4.2 Test Setup

After we have synthesized the arbiter hardware implementations, we use gate-level power extraction and gate-level power simulation to obtained comparable power dissipation numbers for the different arbiter configurations.

We made sure that that simulation time was long enough in order for the power dissipation numbers to converge (on the order of 2000 clock cycles). Random requests

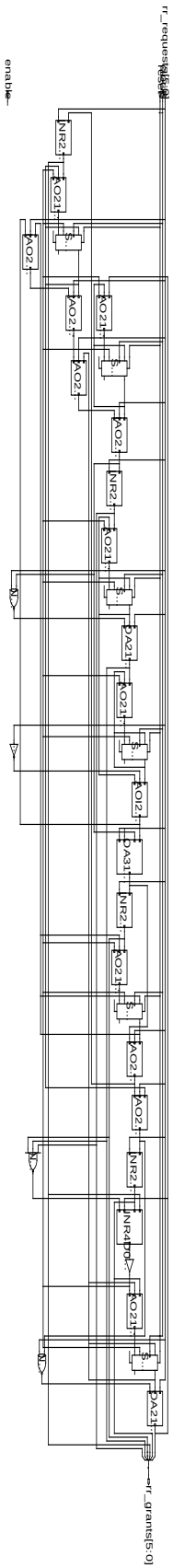


Figure 4.1: Non-Gated Round-Robin Arbiter

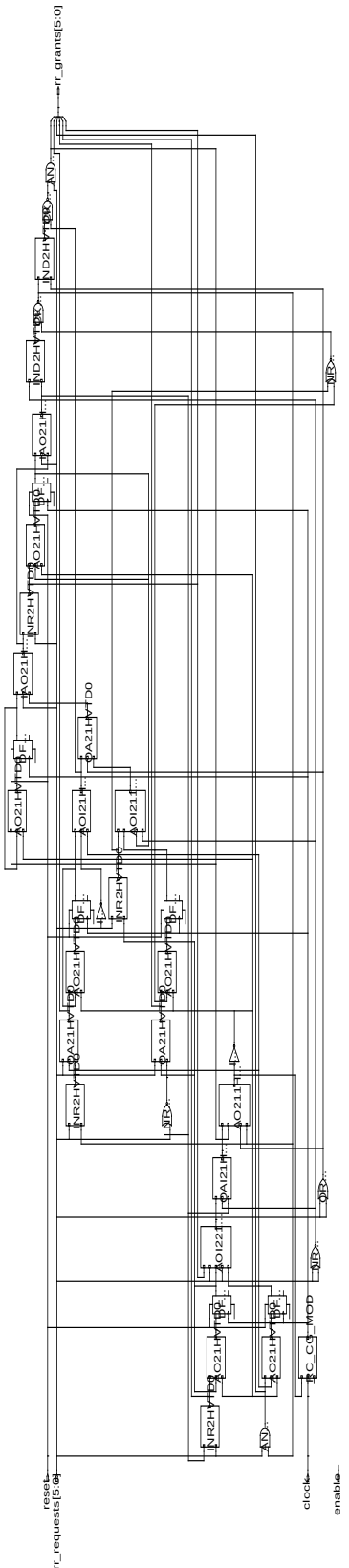


Figure 4.2: Clock-Gated Round-Robin Arbiter

were generated for each contender. By varying the request rate, the behavior of the arbiters under different load conditions becomes apparent.

4.2.1 Test Bench Description

In the simulation test bench, we generate random requests for each contender separately. In VHDL, we can make use of *GENERATE statement* to instantiate a *PROCESS* for each contender. Within each contender *PROCESS*, we then implement the contender request functionality based upon a random number generator (RNG). The RNG generates a number between 0 and 1, which gets converted to a number of cycles between 0 and *Maximum Contender Wait Time*, which is basically the time out period for each contender between the points where it was last granted a request and when it puts in a new request. See Algorithm 4 for the pseudocode of contender *PROCESS*.

Algorithm 4 Arbiter Simulation Test Bench Pseudocode

```
Request(i) ← low
Wait for reset to go low
Randomly wait for upto Maximum Contender Wait Time cycles
Request(i) ← high
loop
  loop
    Wait for Grant(i) to go high
    Request(i) ← low
  end loop
  Randomly wait for upto Maximum Contender Wait Time cycles
  Request(i) ← high
end loop
```

5 Results

5.1 Results from Synthesis

The area results from synthesis for the RR, TDM+RR and TDM arbiters are summarized in Tables 5.1, 5.2 and 5.3 respectively, while Figure 5.1 shows a graphical depiction of the results. For the combined TDM+RR arbiter, the results are shown for a shift-registered version of the TDM arbiter.

Clock-gating the RR arbiter reduced the area for arbiters of size 6 and larger, because the clock-gating circuitry could optimize some equivalent combinational logic away in the clock-gating circuitry, which also occurs in the TDM+RR arbiter. Comparatively speaking, the TDM arbiter configuration has the lowest cell area, for all arbiter sizes. We also see that the cell area for the arbiters largely scales linearly with the arbiter size, irrespective of the arbiter configuration used. Finally, for the TDM arbiter, a counter implementation has a small area advantage over the shift-registered version, but its area jumps irregularly between arbiter sizes.

Table 5.1: RR Arbiter Cell Area (equivalent gates)

Arbiter Size	**Non-Gated Area**	**Clock-Gated Area**
2	106	116
4	229	254
6	320	300
8	430	397
10	543	500
12	665	612

5.2 Power Results

Next, the power dissipation results are summarized for the different arbiter configurations, with and without clock-gating. As well as changing the arbiter size (number of contenders), we also vary arbiter load conditions by changing the *Maximum Contender Wait Time*. A high arbiter load occurs when the wait time is small, while low load has a wait time that is large. At full load, each contender will immediately put in a new request after its last request was served by the arbiter.

Table 5.2: TDM Arbiter Cell Area (equivalent gates)

Arbiter Size	Counter Area	Shift-Register Area
2	54	66
4	113	133
6	193	199
8	226	265
10	318	332
12	339	398

Table 5.3: TDM+RR Arbiter Cell Area (equivalent gates)

Arbiter Size	Non-Gated Area	Clock-Gated Area
2	152	179
4	344	375
6	541	510
8	711	676
10	916	876
12	1021	986

5.2.1 RR Arbiter

For the RR arbiter, we see that except for an arbiter with two contenders, the power dissipation first increases when going from full load to lower loads, reaches a maximum at a certain point, and then starts decreasing until it reaches a asymptotic boundary (Figure 5.2).

The effect of clock-gating is different for when load conditions are changed. At high loads, the number of grants is sufficiently large so that the clock-gating circuitry serves no purpose at all. By decreasing the load, the clock-gating circuitry has more opportunities to disable the clock to the registers holding the contender priorities. Because the number of arbiter grants decreases, the number of times the contender priorities need to be updated decreases as well.

The point at which clock-gating becomes useful varies for different arbiter sizes, which is shown in Figure 5.3. The synthesis tool implemented a wasteful implementation of clock-gating for arbiters of size 2 and 4, which therefore have a relative large positive power difference at high load, whilst the clock-gated arbiter of size 6 was actually performing better under all load conditions, because the synthesis tool found it necessary to insert a clock tree buffer in the non-gated version. If we look at even larger arbiters, the benefits of clock-gating can appear at higher loads for an arbiter of size 8 than for arbiters of size 10 and 12, respectively. However, when load conditions are reduced even more, eventually, the higher sized arbiter will enjoy the largest benefits from clock-gating.

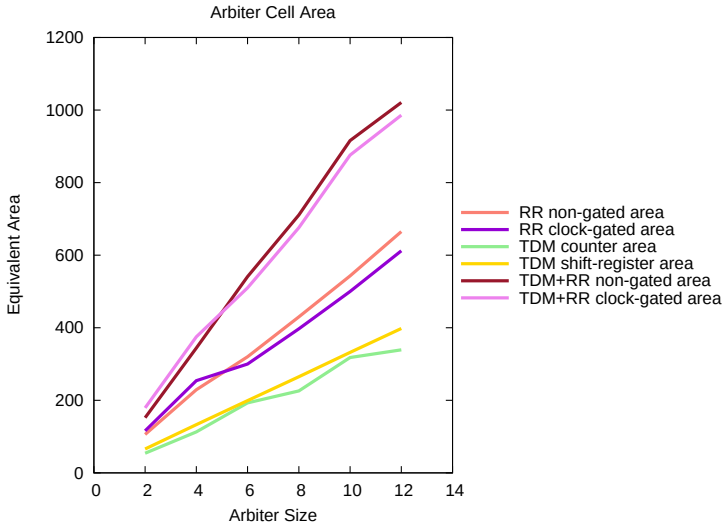


Figure 5.1: Arbiter Cell Area

For a further discussion on why there is point of maximum inflection in the power dissipation curve for a RR arbiter, the reader is referred to section 6.1.1.

5.2.2 TDM Arbiter

The results for a non-gated version of doing TDM exclusively is shown in Figure 5.4. Because initially a binary counter was used for the TDM implementation, the power dissipation jumps irregularly between different arbiter sizes, because the optimal binary counter implementation occur when the arbiter size is a power of two. We can overcome this behaviour by using a shift-register implementation (ring counter), which result is also shown. See section 6.1.2 for a comparison of binary and ring counters.

No TDM arbiters were implemented using clock-gating, because after each arbitration cycle, the contender priority is simply shifted to the next contender.

5.2.3 TDM + RR Arbiter

From the results for doing a two-step arbitration of TDM and RR (Figure 5.5), there is a clear benefit in doing a clock-gated design at high loads, because then there is almost no RR arbitration. The dramatic jump in power dissipation is the point at which the RR arbiter kicks in. This jump is larger for the clock-gated design because of the power benefit of clock-gating at higher loads, but overall, clock-gating remains beneficial.

5.2.4 TDM+subset(RR) Arbiter

When a time slot is fixed in TDM for a multitude of contenders, and RR arbitration is performed between them, the average-case latency of the arbiter can be improved with respect to plain TDM arbitration. The power dissipation for different frame size

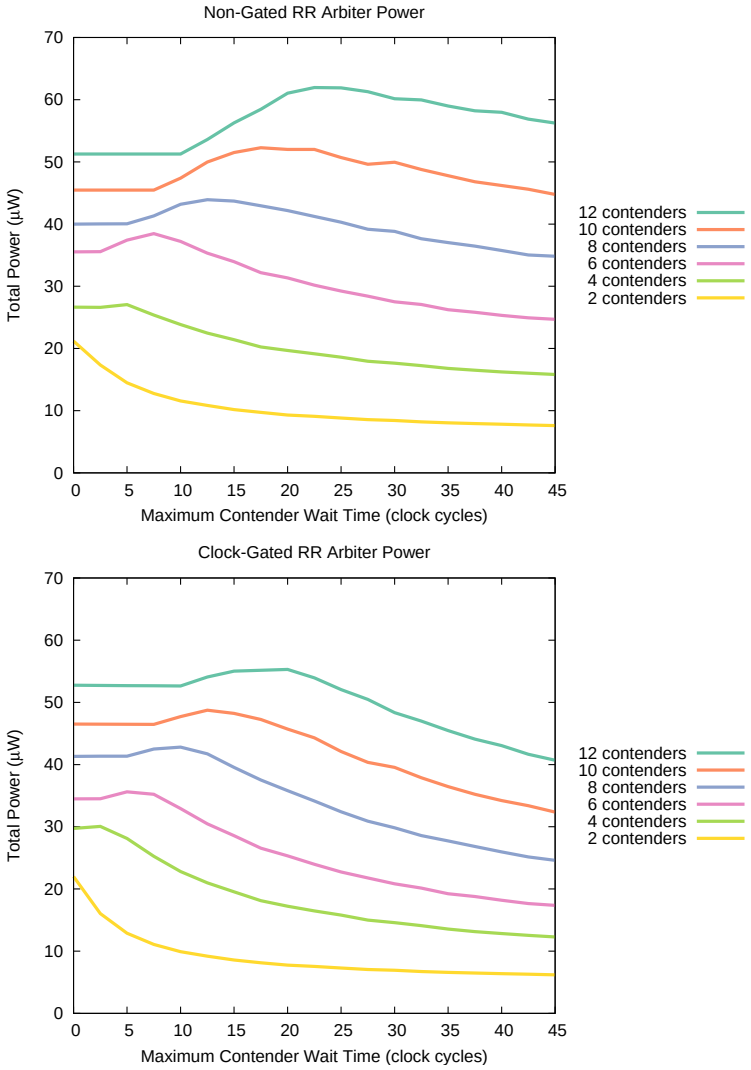


Figure 5.2: RR Arbiter Power

under changing load conditions for arbiters of size 6, 8, 10 and 12 are shown in Figures 5.6, 5.7, 5.8 and 5.9 respectively. With a frame size of 1, we get back to doing plain TDM arbitration, while a frame size corresponding to the arbiter size corresponds to plain RR arbitration. In between, both TDM and RR is performed every arbitration cycle.

Although the average-case latency might be improved, the trade-off comes in terms of higher power dissipation.

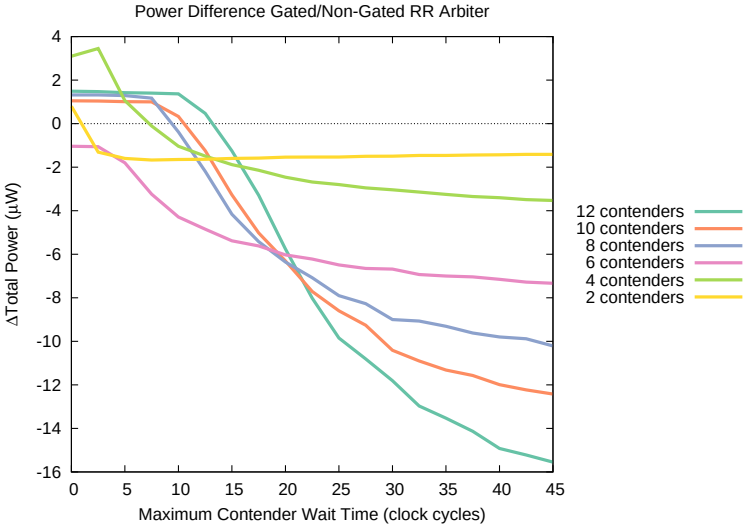
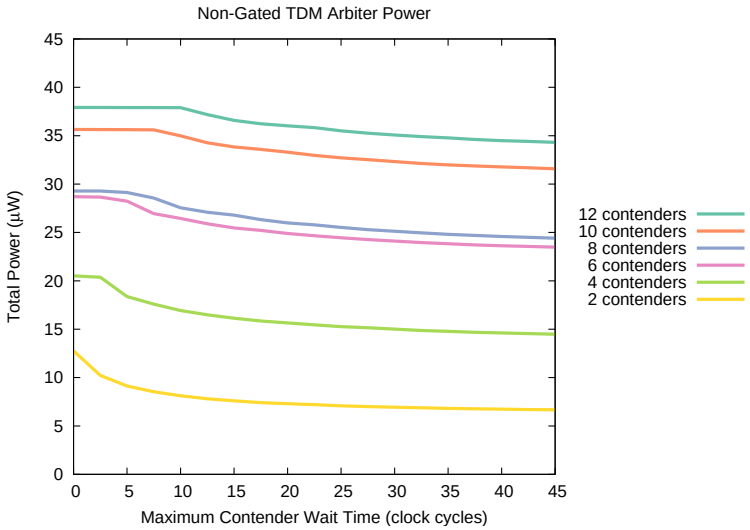
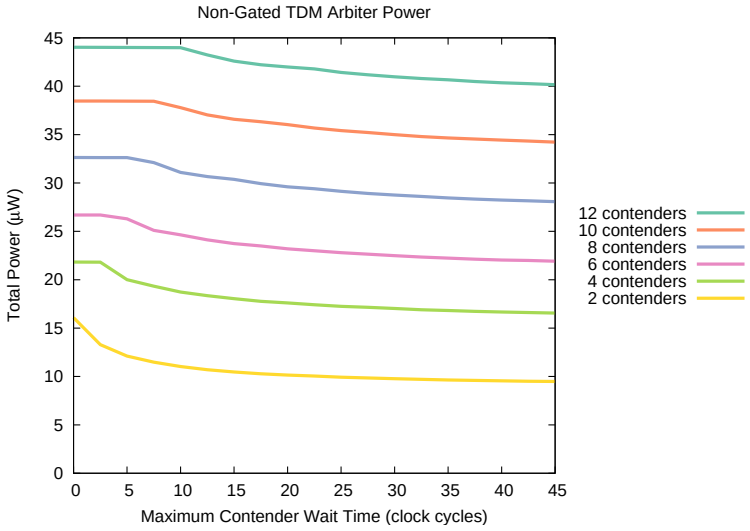


Figure 5.3: RR Arbiter Power Difference



(a) Counter Version



(b) Shift-Register Version

Figure 5.4: TDM Arbiter Power

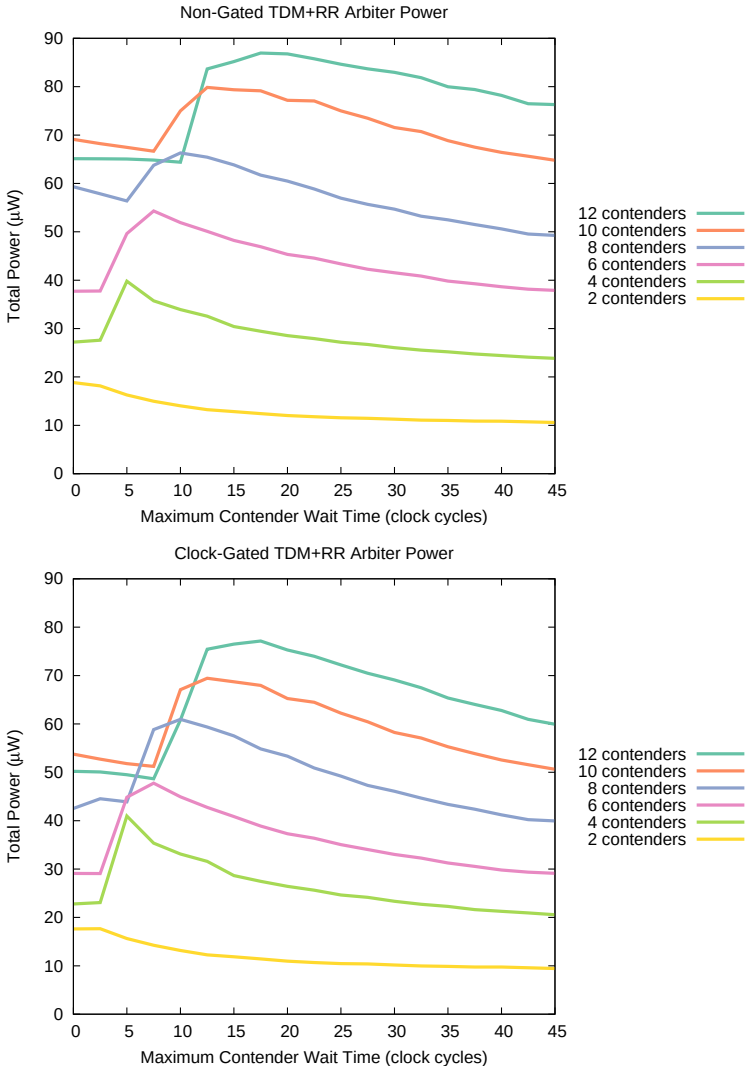


Figure 5.5: TDM+RR Arbiter Power

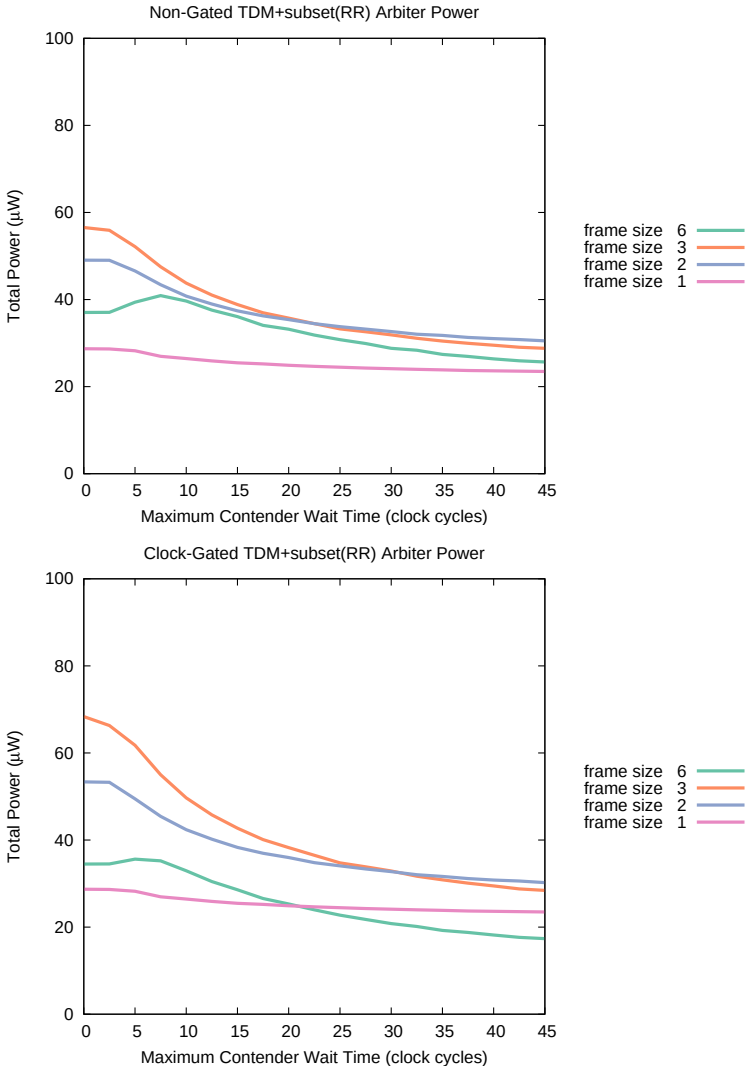


Figure 5.6: TDM+subset(RR) Arbiter Power (size 6)

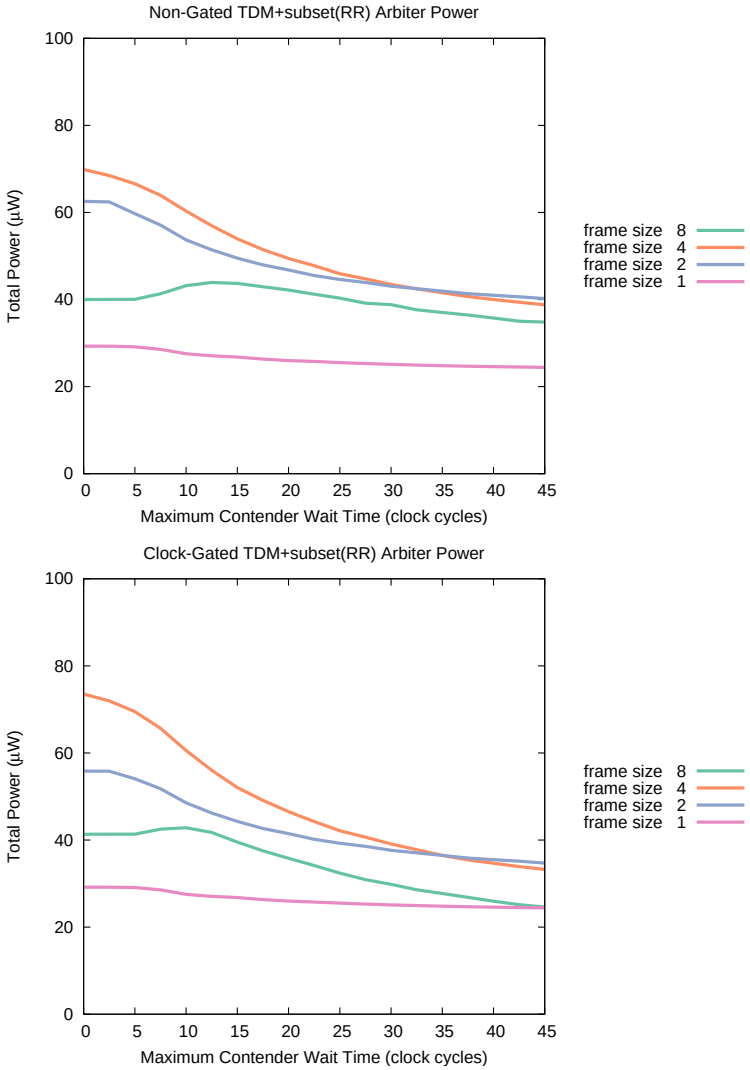


Figure 5.7: TDM+subset(RR) Arbiter Power (size 8)

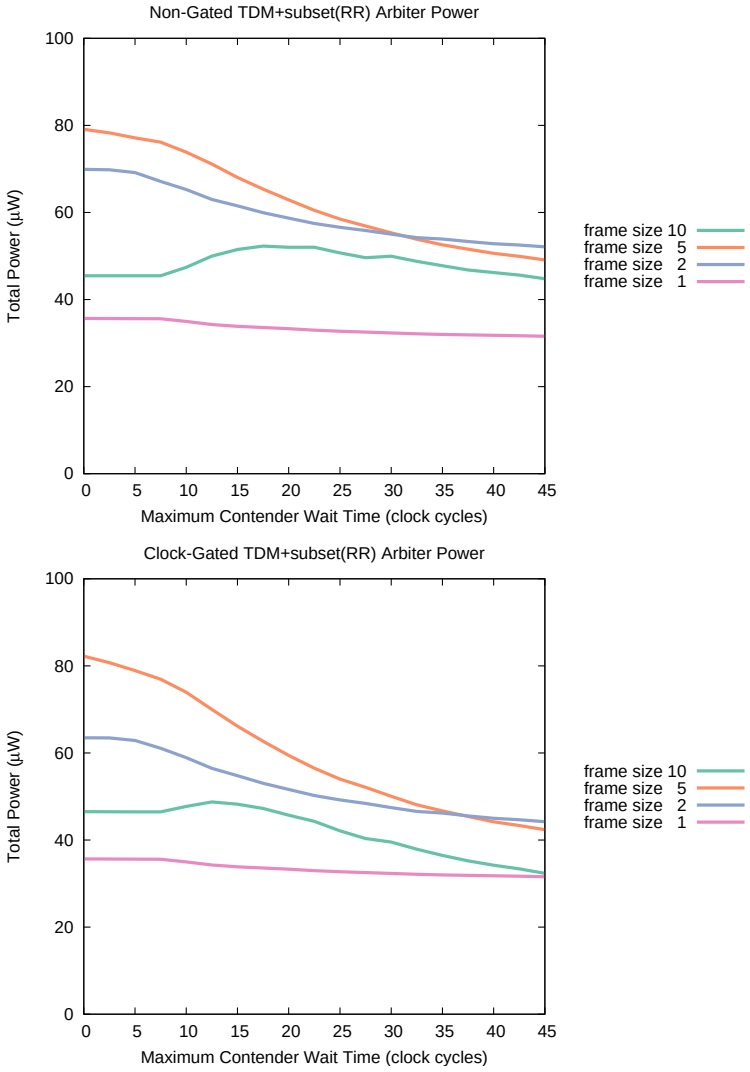


Figure 5.8: TDM+subset(RR) Arbiter Power (size 10)

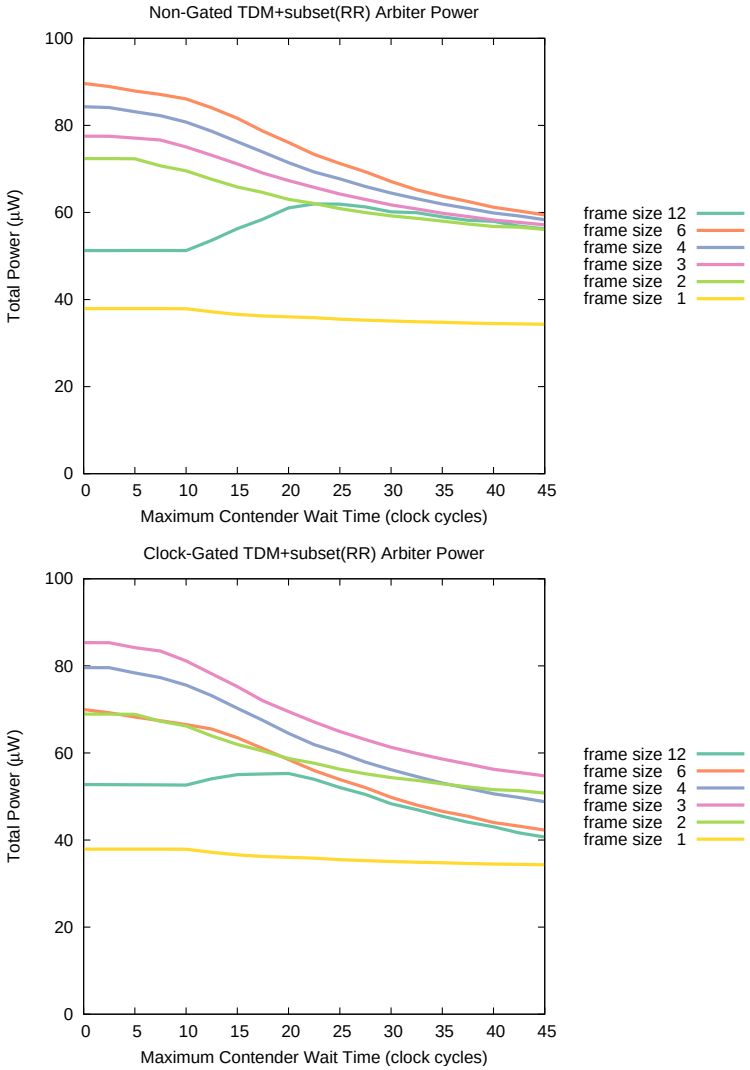


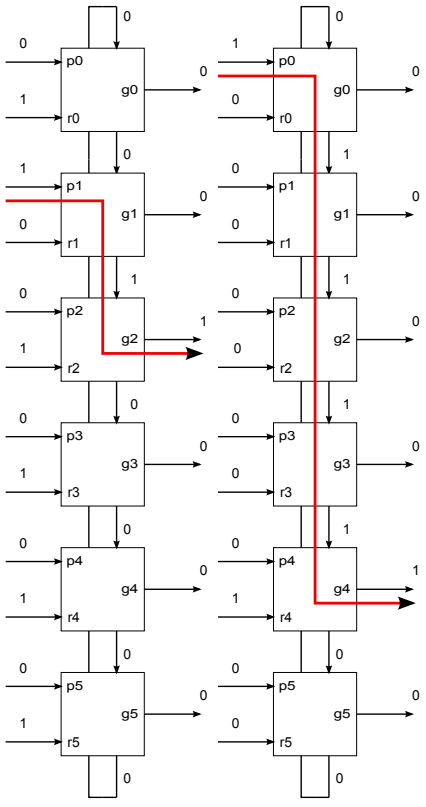
Figure 5.9: TDM+subset(RR) Arbiter Power (size 12)

6 Discussion & Comparison

Some space is provided here to further discuss some peculiarities to the arbiter power dissipation results, as well as a brief comparison between the arbiter configurations.

6.1 Discussion

6.1.1 Carry Chain Analysis



(a) high load (b) low load

Figure 6.1: Carry Chain for High and Low Load

To understand why the power dissipation of a RR arbiter first increases when going from full load to lower loads, peaks at a certain midpoint and then starts to decrease, it is necessary to look more closely to the hardware implementation. The architecture of the carry chain is like the one given in [8], but we had to replicate the carry chain in

order to avoid a combinational loop. The arbiter starts scanning at the contender with highest priority, until it finds a contender with a request. The distance of this scan varies with the load condition. At high loads, the arbiter is likely to find a contender with a request close to the contender which currently has highest priority (if not the contender with highest priority itself). At lower loads, the arbiter is less likely to find a contender next to the one which currently has highest priority, and thus the distance the carry propagates through the chain increases (Figure 6.1). At the same time, decreasing the load also means the number of contenders that put in requests decreases. Larger carry propagations suffer from more power dissipation, but a decreasing number of contenders putting in requests lowers the number of requests the arbiter has to service, and thus its power dissipation. The peak in the power dissipation for a RR arbiter then appears at the load where the power dissipation contributed by all carry propagations is largest relative to the power dissipation contributed by each contender request.

Wait Time	Carry Distance Totals											Accumulated Total
	1	2	3	4	5	6	7	8	9	10	11	
0cc	1999	0	0	0	0	0	0	0	0	0	0	1999
5cc	1994	0	0	0	0	0	0	0	0	0	0	1994
10cc	1959	0	0	0	0	0	0	0	0	0	0	1959
11cc	1785	141	38	9	9	2	1	1	1	0	0	1987
12cc	1532	329	85	24	6	3	1	3	0	0	2	1985
13cc	1425	363	118	32	20	12	2	3	2	3	1	1981
14cc	1264	418	153	57	31	13	12	6	5	4	8	1971
15cc	1109	416	203	83	54	35	16	8	12	11	6	1953
20cc	617	385	264	136	110	87	64	31	38	24	26	1782
25cc	412	285	203	142	120	88	78	53	58	55	39	1533
30cc	343	235	173	121	82	79	86	61	61	49	39	1329
35cc	238	188	156	107	97	81	80	63	58	50	46	1164
40cc	196	130	136	106	97	90	70	62	59	50	36	1032
45cc	167	124	119	80	87	70	63	60	51	56	40	917

Table 6.1: RR-12 Accumulated Carry Totals

During simulation of the RR arbiter, we checked how long the distance of the carry propagation was for every round of arbitration. The carry distance totals for a RR arbiter of size 12 is given in Table 6.1, where you can see a gradual increase in the carry distance but a concurrent decrease in the accumulated total (or number of arbitrations) when the arbiter load decreases. Figure 6.2 shows the comparable carry distance totals for arbiters of size 2, 4, 6, 8, 10 and 12, where the increase in carry distances becomes apparent for increasing arbiter sizes.

By linearly weighing the carry distance totals, we get a rough estimate of the power dissipation contributed by all carry propagations. The results of the weighing is shown in Figure 6.3, which confirms our expectations, although a scaling factor should be applied according to the arbiter size.

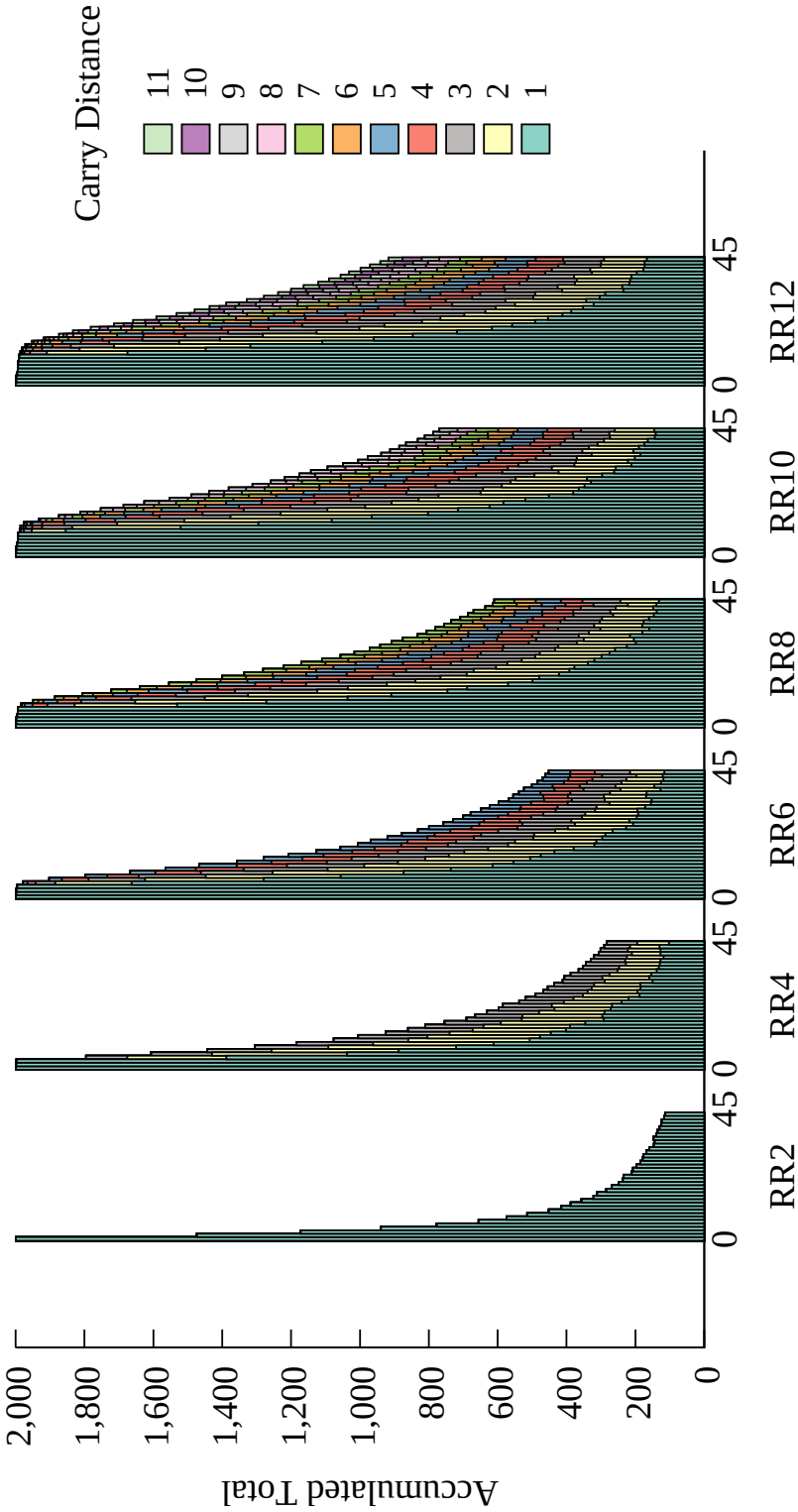


Figure 6.2: Round-Robin Arbiter Accumulated Total

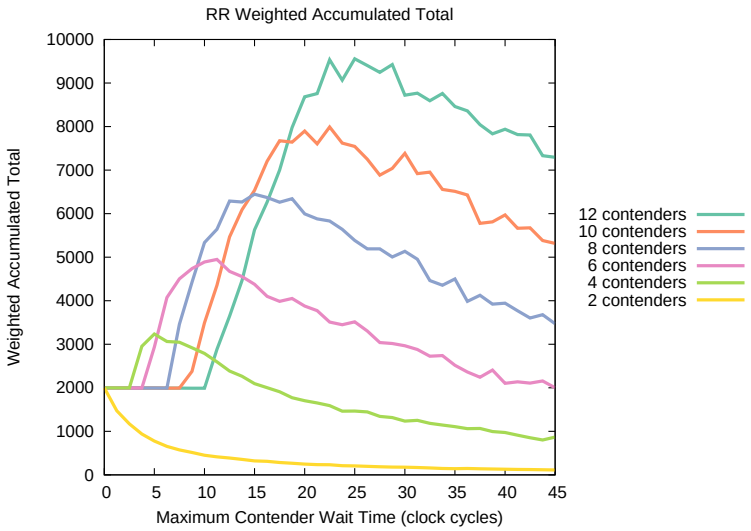
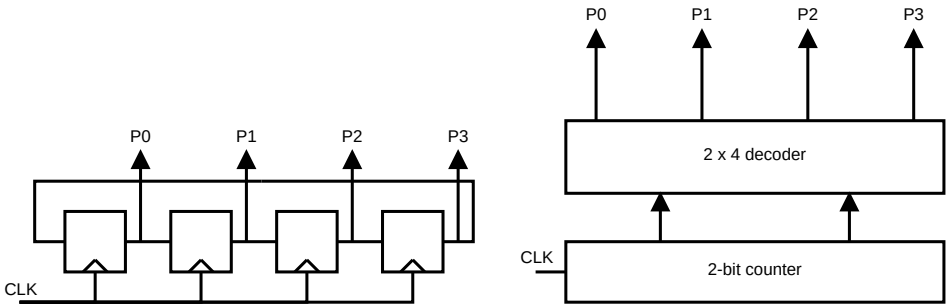


Figure 6.3: Weighted Carry Chain Length

6.1.2 Ring Counter vs. Binary Counter



(a) ring counter (b) binary counter

Figure 6.4: Ring and Binary Counter

There are two main advantages in using a ring counter over a binary counter to generate the TDM priority signals. If we compare a 4-bit ring counter and binary counter (Figure 6.4), we see that a ring counter does not need any extra state decoding logic to generate the priority signals (P0, P1, P2, P3). The binary counter however, needs a 2 x 4 decoder in addition to the 2-bit counter. The binary counter implementation needs even more decoding logic when the number of priority signals to be generated is not a power of two.

6.2 Comparison

To compare the different arbiter configurations (RR, TDM, TDM+RR), we will look at their power dissipation at *full load*, *high load*, *mid load* and *low load*, which correspond to a maximum contender wait time of 0 clock cycles, 5 clock cycles, 25 clock cycles and 45 clock cycles respectively. These results are summarized in Tables 6.2, 6.3, 6.4 and 6.5, where in each individual load table, the arbiter configurations that performed best and worst in terms of power dissipation for each arbiter size are highlighted in green and red respectively.

For full load, we see that the TDM arbiter has the lowest power dissipation, and that its counter-version is more beneficial than a shift-registered one, except for an arbiter of size 6 (which has a wasteful binary counter implementation). For small arbiter sizes, we get the worst power dissipation with a RR arbiter, but starting from an arbiter size of 6, the non-gated TDM+RR arbiter takes over as worst configuration.

When we look at high load conditions, we see that the non-gated TDM-RR arbiter has the worst power dissipation for all arbiter sizes. The TDM configuration again has the lowest power dissipation.

The TDM+RR arbiter also has the worst power dissipation for a mid load and low load. At mid load, a clock-gated RR arbiter nearly edges out a shift-registered version of the TDM arbiter in terms of power dissipation, but for other arbiter sizes, a counter-version of the TDM arbiter has the lowest power dissipation. For low load, a clock-gated RR arbiter has the lowest power dissipation for arbiter sizes 2, 4 and 6. But at higher arbiter sizes, the TDM arbiter will take over as the arbiter with the lowest power dissipation.

Arbiter Size	Arbiter Configurations at Full Load (0cc)					
	RR		TDM		TDM+RR	
	non-gated	clock-gated	counter version	shift-register version	non-gated	clock-gated
2	21.2	22.0	12.8	16.1	18.8	17.6
4	26.6	29.7	20.5	21.8	27.2	22.8
6	35.5	34.5	28.7	26.7	37.7	29.1
8	40.0	41.3	29.3	32.6	59.3	42.5
10	45.5	46.5	35.7	38.5	69.1	53.8
12	51.3	52.8	37.9	44.0	65.1	50.2

Table 6.2: Arbiter Power Dissipation (μW, full load)

Arbiter Size	Arbiter Configurations at High Load (5cc)					
	RR		TDM		TDM+RR	
	non-gated	clock-gated	counter version	shift-register version	non-gated	clock-gated
2	14.5	12.9	9.1	12.1	16.3	15.7
4	27.1	28.1	18.4	20.0	39.8	41.0
6	37.4	35.6	28.2	26.3	49.6	44.9
8	40.1	41.3	29.1	32.6	56.4	43.9
10	45.5	46.5	35.6	38.5	67.5	51.8
12	51.3	52.7	37.9	44.0	65.1	49.5

Table 6.3: Arbiter Power Dissipation (μW, high load)

Arbiter Size	Arbiter Configurations at Mid Load (25cc)					
	RR		TDM		TDM+RR	
	non-gated	clock-gated	counter version	shift-register version	non-gated	clock-gated
2	8.8	7.3	7.1	9.9	11.6	10.5
4	18.6	15.8	15.3	17.3	27.2	24.6
6	29.2	22.7	24.5	22.8	43.4	35.1
8	40.3	32.4	25.5	29.1	57.0	49.2
10	50.7	42.1	32.7	35.4	75.0	62.2
12	61.9	52.1	35.5	41.4	84.6	72.2

Table 6.4: Arbiter Power Dissipation (μW, mid load)

Arbiter Size	Arbiter Configurations at Low Load (45cc)					
	RR		TDM		TDM+RR	
	non-gated	clock-gated	counter version	shift-register version	non-gated	clock-gated
2	7.6	6.2	6.7	9.5	10.6	9.4
4	15.8	12.3	14.4	16.6	23.8	20.6
6	24.7	17.4	23.5	21.9	37.9	29.1
8	34.8	24.6	24.4	28.1	49.2	40.0
10	44.8	32.4	31.6	34.2	64.8	50.6
12	56.2	40.7	34.3	40.2	76.3	59.9

Table 6.5: Arbiter Power Dissipation (μW, low load)

7 Conclusions

Various arbiter configuration have been presented and implemented, for which we characterized their power dissipation behavior, under varying load conditions and arbiter sizes. The effect of the power reduction technique of clock-gating was studied as well.

It has been clearly shown that there is no single arbitration scheme that will perform well in terms of power dissipation under all load conditions. We showed that for a round-robin arbiter, the power dissipation does not vary linearly with load. We studied the effect of applying clock-gating to the designs, and saw that the benefit of clock-gating largely depends on the load conditions.

The power dissipation of performing time-division multiplexing was shown to be lower than in the case of performing round-robin, except when we considered a clock-gated round-robin implementation at very low loads.

A two-step arbitration of time-division multiplexing and round-robin was largely found to be beneficial over plain round-robin arbitration when considering clock-gated implementations, or for small arbiter sizes.

When we tried to improve the average-case latency of time-division multiplexing by arbitrating in each time slot between multiple contenders in round-robin fashion, we saw a clear trade-off in terms of increased power dissipation.

We also analysed the behavior of why the round-robin arbiter power dissipation first increases when going from full load to lower loads, then peaks at a certain midpoint and then starts to decrease. We found that there were two opposing forces at work when the load was reduced: a gradual increase of the carry distance, but a decrease in the total number of accumulated carries. The inflection point in the power dissipation curve was then found by weighing the carry distance totals.

Part II

Multiprocessor Systems-on-Chip

8 Introduction

The use of application-specific instruction-set processors (ASIP)s has been succesfully applied towards the implementation of ultra low power wireless sensor nodes [9]. In ASIP design, the instruction-set architecture (ISA) of the processor is specifically tailored to optimize the performance and related metrics of applications. Those designs offer more flexibility over full-custom integrated circuit designs and are more efficient than digital signal processors. And although power savings can be realized in ASIPs, further gains will have to come from exploiting bit-level, instruction-level and task-level parallelism of applications in the form of (heterogeneous) multiprocessor systems-on-chips (MPSoC)s. Using data-locality and concurrency, highly power efficient systems can then be realized.

The challenges of MPSoC design have been described previously in [25], where it is explained that MPSoC design is motivated by time-to-market considerations, design reuse, the simplification of system verification, and to offer programming flexibility after manufacturing. Finding the right programming models that allow for easy application mapping is considered to be the main challenge for these designs, because of the concurrency present in these applications. Tasks on different processors have varying ways to communicate with each other, and the avoidance of deadlock and starvation in the system is key. In embedded MPSoCs, parts of the system might be shut down or run on a lower clock frequency in order to save energy, in addition to the presence of real-time requirements; but how the right system balance can be found between the requirements of a multitude of tasks running on separate processors remains an open question. The debugging of MPSoC designs is another challenge, and it is suggested that the debugging across different models of computation is a necessary requirement for any debugger. The next challenge comes from the fact that individual processors in a MPSoC design can be heterogeneous to each other, with designs not only limited to a small amount of processors. The design space for finding a correct MPSoC architecture has thereby becoming intractably large, while system design exploration for MPSoC is still in its infancy.

MPSoCs can be classified in two classes. Homogeneous MPSoCs consist of the same processing cores being replicated multiple times, where each processing core offers general-purpose computing. In heterogeneous MPSoC designs on the other hand, a multide of separately optimized processing cores are used to implement the system functionality, and although more complex, these designs offer also more performance gains. Processing and storage elements (that are either locally or globally visible to the processor cores) are connected together using an on-chip communication infrastructure.

In terms of power dissipation, homogenous MPSoC designs are likely to suffer from a mismatch between computation and communication, when a system's functionality is partitioned across processing cores. In heterogeneous designs however, because of utilizing data-locality and concurrency, the power dissipation of the individual processing cores and on-chip interconnect can be separately optimized, and low-power techniques

such as power gating (Appendix A) can be readily applied to the each separate core [11].

In the following chapters, a sample MPSoC system is implemented using a producer and consumer process on separate processors that communicate with each other via a hardware FIFO, while in software, synchronization is considered using polling and interrupt mechanisms, and its power dissipation is analyzed.

9 Related Work

There is a rich literature on multiprocessor systems-on-chips, of which only a selection can be provided here, indicative of current trends in research.

In [31], an overview is given on energy-efficient programming models for multiprocessor systems-on-chips and the coupling thereof on hardware architectures. Both shared memory and message passing are considered as target models. The authors point out that the choice for one programming model over the other is not clearly delineated, and thus analysis is needed to find out if a certain programming model can outperform the other for a given application. Using applications from the domain of multimedia and signal processing, they show that the performance and power dissipation of the message passing and shared memory programming models differ largely when considering the splitting of data across different processor tasks, the degree of data sharing between the processor tasks, data remaining local the processor tasks, and the time spend in computation/communication in processor tasks. Accordingly, a set of MPSoC design guidelines was proposed by the authors for discriminating between message passing and shared-memory programming models.

The authors of [14] implemented an on-chip interconnect and protocol stack that support multiple programming models and communication paradigms for MPSoC systems. In a sample design, the interconnect was only found to occupy 4% of the total chip area, while offering high performancy and low latency. A template for MPSoC platform design has been introduced that allows for the independent development and verification of applications, possibly using different programming models and communication paradigms for each separate application [15]. Previously, a scalable multiprocessor template for applications described by synchronous data flow graphs had been introduced. A network-on-chip was used for the communication between processors, which allows for the inference of the timing behaviour of the MPSoC system [1].

Kahn process networks are regularly used to describe MPSoC applications, whereby processor tasks communicate via unbounded FIFOs. The authors of [16] introduce the concept of windowed FIFOs, which allows for the reordering of data, non-destructive reads and skipping of data in the communication channels between processor tasks, while still keeping the important properties from (determinate) Kahn process networks. Combining windowed FIFOs and Protothreads in a Cell Broadband Engine based MPSoC system, a considerable speedup was measured, without the need to write architecture-specific programming code [13].

10 MPSoC Design

A multiprocessor system-on-chip (MPSoC) is a system-on-chip (SoC) integrated circuit with multiple processing cores working together in parallel. It combines various processing and storage elements on a single chip, connected together with an on-chip communication infrastructure. A *homogeneous* MPSoC architecture replicates the same processing core a multitude of times, meaning that each processor will have the same software system. In *heterogenous* architectures on the other hand, different processor cores are combined together, each with its own software system [32]. And although more complex to design, heterogeneous MPSoCs can offer improvements in performance and show lower power dissipation, because each processor core can be optimized separately [11].

MPSoC architectures can be further divided in two classes, based on the communication abstraction between the processing and storage elements: *shared memory* and *message passing* [32] [7].

In shared memory architectures, a global memory space is visible to all processors, addressed implicitly by using the load and store instructions of the individual processor cores. These instructions are close to the actual hardware, and as such, the benefits of a shared memory architecture are dependent on the latency of these instructions and the amount of data that can be moved in a single load/store operation. By using a memory hierarchy (like caches), (shared) data can be moved closer to the processor that uses the data, however, special care is needed to make sure data remains consistent across all processors.

With message passing, each processor core has its own address space. Communication is made through special input and output operations, typically send and receive instructions, instead of being implicitly available via the memory system. These send and receive instructions also include some form of tag, so that processes on individual processors have a notion of each other. The synchronization of processors executing these send and receive instructions can happen in two ways: either *synchronous* or *asynchronous*. When a processor initiates a send operation towards a particular processor and has to wait until the receiving processor initiates a receive operation, we call the communication synchronous. If a sending processor can immediately continue after initiating a send operation, without having to wait for the receiving processor, the communication is asynchronous, and usually involves buffering the sending data in some storage location close the sending processor. The usage of direct memory access (DMA) transfers allows for non-blocking sends, it is an hardware feature that can access the local storage of a processor independent of the instructions the processor is currently executing. At the end of a DMA transfer, data available at a buffer at the receiving processor, which is then free to initiate a receive instruction to move the data into its own address space. The cost to setup a DMA transfer has to be carefully weighted against the type of data that is to be transmitted: for short messages between processors the overhead of a DMA

transfer can be too large. Alternatively, and simpler, FIFO buffers in hardware can aid with the data transfers. A sending and receiving processor could directly address the hardware buffer using send and receive instructions, however, if the FIFO buffer were too small, it could easily mean that a sending processor would have to block until a receiving processor has removed data from the FIFO [7].

On the software side, traditional application programming interfaces (APIs) for parallel computers that mimic the hardware architecture, are available: OpenMP [28] for shared memory and MPI [27] for message passing. However, with advent of heteregenous MPSoC systems, there is a move towards concurrent hardware and software design. There are two key requirements for programming these systems [18]. Firstly, to reduce the software development cycle in both cost and time, a high-level programming model is needed, so that software developers are not encumbered with lower level architectural details. On the other hand, to optimize the performance and power dissipation of the system, it is necessary to customize the software and hardware layers for a specific application, requiring programming changes on a lower level. Programming hetereogeneous MPSoC platforms can only be successful when the communication overhead between the application tasks running on the different processors is taken into account in the programming model [26]. Although the classical programming interfaces eases the synthesis of software, automatic design space exploration requires the use of a *model of computation* [12].

A model of computation is an abstract representation of a computing system [17]. It captures the essentials of a computation: how one state of a computation moves to the next state, what operations are necessary to make such moves possible, and how information about the state of a computation is used within those operations. By restricting the kind of computations that can take place, the complexity of system design is greatly reduced. Example model of computations are Petri nets, finite state machines, discrete event models and Kahn process networks, the latter being frequently used for signal processing applications. The semantics of the Kahn process network model of computation stipulate that these functional kernels communicate by unbounded FIFO channels, where kernels can only be blocked when a FIFO channel is empty [26]. And although unbounded FIFO channels cannot be realized physically, kernels that use blocking read and write functionality can preserve the semantics of the Kahn process network [13].

For the implementation of our system, we resort to message passing, since a shared memory architecture would implicitly bound the communication model by using load/store instructions. We also want to stay close to the hardware for performance and power dissipation reasons, therefore, no higher level abstraction is necessary at this point in time. If need be, the software side can be extended to support a more layered based approach. We resort to a two processor system, one with a producing process and one with a consumer process. Interprocess communication and event handling in software is explained in detail in [34], we consider both interrupts and polling, as they are readily supported by the processor core we had at our disposal.

Implementation & Test Setup

11

We implemented a producer/consumer system using a hardware FIFO as communication mechanism. The overall system picture is given in Figure 11.1, where the data and program memories of the individual processor cores are depicted as well. A sample 16-bit processor core generated and designed with the Target processor tools [35] were used for both the producer and consumer cores. See Appendix D for a description of the Target tool flow used and of the sample processor core. The hardware design of FIFOs is discussed in Appendix C.

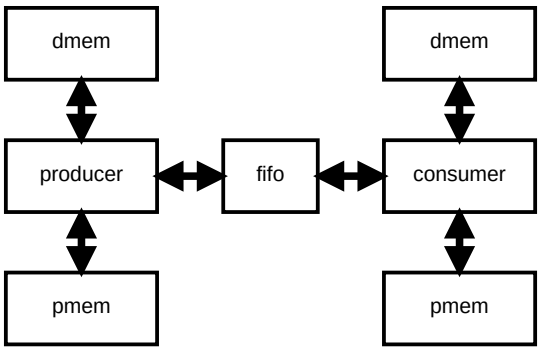


Figure 11.1: Producer/Consumer System

11.1 Software

Using the FIFO as communication mechanism, the synchronization of the producer and consumer processors in software boils downs to using the supplied FIFO primitives, namely that it is either full or empty, but an extra half-full primitive for the FIFO was introduced as well.

To handle external events like a change in the state of the FIFO primitives, there are two software options: *polling* and *interrupts*. With polling, ever so often the processor checks if the FIFO primitives have been changed, and will produce and consume data items if the conditions of the FIFO primitives allow that. With interrupts on the other hand, a change in the FIFO primitives is signaled to a special software routine setup in the processor, which then can execute the production and consumption of data items. If the FIFO primitives are not changing, the processor is left free to do other work.

During instruction set simulation with the Target processor tools it became apparent however that a interrupt based configuration suffered too much overhead from context switching. During context switching, the processor state is saved on the stack (which is located in the data memory) so that a interrupt service routine (ISR) can be called.

After the ISR has finished the handling of an interrupt, the processor state is restored from the information saved on the stack. But both the saving and restoring of this information need quite a few memory accesses, which tend to be costly in terms of power dissipation. And although an interrupt-based system can still prove benefitial in terms of power dissipation for a system that remains largely idle and includes features to turn on and off the power to the individual processor cores, only the polling configuration was ultimately considered for further hardware implementation. Its software description follows next.

The C language code for both the producer and consumer processor cores consists of a loop where we either time out for certain period, or poll the FIFO flags to see if we can produce or consume some data. By changing the timeout period, we can change the overal duty cycle of the producer/consumer system. In software, this timeout period can be controlled by executing a *nop instruction* for a set number of times. The Target design tools include an option to convert this into a zero-overhead hardware loop (Appendix D). The FIFO register is memory-mapped into the IO space of both the producer and consumer core. The producer polls by checking the FIFO *is_empty* or *is_half_full* flags locations in the register, while the consumer polls the *is_half_full* and *is_full* flags. Because we want the number of instruction cycles to be same no matter which flag we are polling, we had to convert a conditional statement into a corresponding arithmetical operation. A normal conditional statement would not evaluate the second expression when the first expression in the statement evaluates to true.

The producer will produce 16 data items when the polling check evaluates to true, reading the values from an area reserved in its data memory and moving them to memory-mapped FIFO, similarly, when the consumer polling check evaluates to true, it will consume 16 data items by moving them from the memory-mapped FIFO to its data memory. The FIFO depth chosen as such was 32 items, with a data-width of 16 bits, and includes a extra *half full* flag that signals when the FIFO is filled with half of possible data values, next to empty and full flags. The area reserved for data items in the producer and consumer data memories had a size of 8192, implemented as circular buffer in software, whereby the producing and consuming of data values wraps around when the end of the buffer is reached. The total program size of the producer and consumer was less than 256 words.

11.2 Hardware

Using the Target design tools, we generate the VHDL description of the producer and consumer processor cores.

For the hardware implementation of the producer/consumer system we used 90 nm low-power high-threshold voltage standard cell libraries from TSMC [36]. We synthesized the design with a target clock frequency of 100 MHz, together with clock-gating insertion. The individual data and program memories were generated with the Virage logic memory compiler [38], but only 16k memories were available for measurement, with a data width of 16 bits.

The tool flow uses for the physical implementation of the system is described in Appendix B.

11.3 Test Setup

After we have synthesized the producer/consumer system, we use gate-level power extraction and gate-level power simulation to obtain power dissipation numbers. We will look at duty cycles with periods of 1024, 2048, 3094 and 4092 clock cycles.

The testbench was simulated through Cadence NC-sim [4]. We made sure that the simulation time was long enough in order for the power dissipation number to converge (on the order of 750,000 clock cycles). In NC-sim, we have the option to initialize the data and program memories of the producer and consumer processor cores. The data memory of the producer was initialized with linearly increasing sample data items.

11.3.1 Test Bench Description

The top entity of the producer/consumer system includes input signals to wake and halt the processors, as well as processor halted output signal (Figure 11.2).

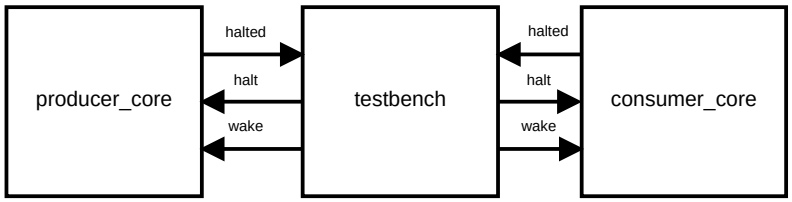


Figure 11.2: Producer/Consumer Testbench

In theory, we could thus externally control when want to wake up and halt the processor cores. However, in the case of the polling configuration, each individual core is really controlled by software, and thus the only thing the testbench needs to do is to initially wake up the processors, as they are initially not clocked. The wake up signal is internally tied to an individual interrupt line of the producer and consumer processor cores, and interrupt service routine will subsequently change the processor states to executing (polling). Afterwards, interrupts are disabled for further use.

12 Results

12.1 Results from Synthesis

The area results from synthesis are summarized in Table 12.1. We see that a large part of the area is composed of the data and program memories, but should note that smaller sized memories could have been used when they were available (a 256x16 memory would have sufficed already for the program memories).

Table 12.1: Producer-Consumer Cell Area (equivalent gates)

Component Instance	**Cell Area**
producer_dmem_cell	488733
consumer_dmem_cell	488733
producer_pmem_cell	488733
consumer_pmem_cell	488733
producer_cell	37965
consumer_cell	37438
fifo_cell	11097
producer_core_gate_cell	12
consumer_core_gate_cell	12
fifo_gate_cell	12
producer_pmem_gate_cell	12
producer_dmem_gate_cell	12
consumer_pmem_gate_cell	12
consumer_dmem_gate_cell	12

12.2 Power Results

Next, the power dissipation result for the producer/consumer system are summarized in Table 12.2 for four duty cycles: of periods 1024, 2048, 3096 and 4092 respectively.

We see a small reduction in power dissipation when the duty cycle is lowered. This is expected, as a lower duty cycle means that the processor cores are polling less when looking over the same (simulation) time period, and thus the associated power dissipation

Table 12.2: Producer-Consumer Power Dissipation

Configuration	**Total Power (mW)**
producer/consumer period 1024	4.05
producer/consumer period 2048	3.88
producer/consumer period 3096	3.82
producer/consumer period 4092	3.78

is reduced. At the same time, the power dissipation for setting up a timeout loop of different lengths is alleviated by using zero-overhead hardware loops.

More interesting is the breakdown of the power dissipation over the various components. In table 12.3, this is summarized for producer/consumer system with period of 4092. We see that a large part of the power dissipation comes from the producer and consumer program memories, about 65%. This while the power dissipation of the data memories is negligible in comparison, because the number of data movements is relatively small to the number of instructions that need to be fetched from the program memories.

Another large contribution to the power dissipation are the clock tree networks for the different components. In total, around 11% of the total power is dissipated in the clock tree buffers. Lastly, we see a reasonable sized contribution of both the producer and consumer processor cores, which is largely due to their respective instruction decoders. The instruction decoder is active anytime an instruction is fetched from program memory, and the decoding of different instructions is by design going to activate different parts of the instruction decoder, so that the operation of the processor between cycles can be controlled. Therefore, a reasonable amount of switching activity is expected in the instruction decoder, which is reflected in the power dissipation numbers.

The contribution of the interconnect (FIFO) is low, at less than 2% of the total power. But we should note that neither the producer and consumer processor cores and memories are in any way optimized, which could make the interconnect power dissipation more relevant for study.

Table 12.3: Producer-Consumer Relative Power Dissipation

Component	**Relative Power (%)**
main clock tree	3.3
consumer clock tree	3.0
fifo clock tree	1.5
producer clock tree	3.1
consumer core	9.9
consumer core gate	0.1
consumer dmem	0.6
consumer dmem gate	0.3
consumer pmem	31.9
consumer pmem gate	0.3
fifo	1.6
fifo gate	0.1
producer core	10.1
producer core gate	0.1
producer dmem	0.6
producer dmem gate	0.3
producer pmem	31.9
producer pmem gate	0.3

13 Conclusions

The design of a producer/consumer MPSoC system has been presented, and implemented in hardware. For hardware synchronization, a FIFO was used, for which we considered supporting in software both by interrupts and polling.

During software simulation it was found however an interrupt-based system would incur too much overhead from context switching, thus only a system based on polling was used for further power analysis.

We analyzed the producer/consumer system for various duty cycles, and looked at the power dissipation breakdown of the various constituent components. It was found that the contribution of the interconnect on the total power dissipation was relatively low, and that most of the power dissipation came from the (non-optimized) program memories, clock-tree buffers and the instruction decoders of the individual processors.

Because, both the memories and processors were largely left unoptimized for power dissipation, it is unclear if the power dissipation in the interconnect becomes more important for study in an optimized system.

Part III
Future Work

14 Future Work

There are several considerations for future work. Currently, the software API for the MPSoC system does not support inter-process communication where processes or processors are tagged, so that communication at this point in time is limited to producer-consumer pairs. Furthermore, the processor design tool used in this thesis has no support to simulate a multiprocessing system on the software level, meaning that an application using multiple threads on a uniprocessor was needed to mimic the behavior of the producer and consumer process that should actually have run on different processors. Likewise, there is no tool support yet for modelling the interconnect between those processors, therefore it is advised that a software-level description of elements used in the interconnect, such as arbiters and FIFOs, is developed so that full simulation and verification is not only possible when performing hardware synthesis, but even earlier in the design process. This would also help early design space exploration, as trade-offs between using different arbitration schemes or FIFO sizes can be taken into account, perhaps using some (mathematical) power model of these elements.

In this thesis, clock-gating was chiefly used a power dissipation reduction method. However, in a MPSoC system, other methods, such as frequency scaling and power gating, could also provide for large benefits in power reduction. Similarly, we did not optimize the processor and memories for our MPSoC design, which might skew the power dissipation results favorably towards the interconnect. Optimizing the processors and memories, even for a producer-consumer type of system, could provide for more definitive conclusions.

Lastly, there is more work to be done on the kind of applications for which (heterogenous) multiprocessing makes sense, and under which circumstances. One interesting application for wireless sensor nodes is electroencephalography (EEG), which registers the electrical activity in the human brain. There are different algorithms for EEG signal analysis, but can they also be effectively mapped to a multiprocessing architecture, using optimized processors, memories and communication infrastructures, as well as power reduction techniques?

Bibliography

[1] Marco Bekooij, Orlando Moreira, Peter Poplavko, Bart Mesman, Milan Pastrnak, and Jef Van Meerbergen, *Predictable embedded multiprocessor system design*, In Proc. Intl Workshop on Software and Compilers for Embedded Systems (SCOPES), LNCS 3199, Springer, 2004.

[2] Luca Benini and Giovanni De Micheli, *Networks on chips: A new SoC paradigm*, Computer **35** (2002), 70–78.

[3] R. Benjaminsen, F. Duarte, J. Huisken, and K. Goossens, *Gate-level power analysis of on-chip communication infrastructures for biomedicial applications*, Proceedings of ProRISC 2009 (Veldhoven, The Netherlands), November 2009.

[4] Cadence Design Systems, www.cadence.com.

[5] Chien-Hua Chen, Geeng-Wei Lee, Juinn-Dar Huang, and Jing-Yang Jou, *A real-time and bandwidth guaranteed arbitration algorithm for SoC bus communication*, Design Automation, 2006. Asia and South Pacific Conference on, Jan. 2006.

[6] Massimo Conti, Marco Caldari, Giovanni B. Vece, Simone Orcioni, and Claudio Turchetti, *Performance analysis of different arbitration algorithms of the AMBA AHB bus*, DAC '04: Proceedings of the 41st annual Design Automation Conference (New York, NY, USA), ACM, 2004, pp. 618–621.

[7] David E. Culler, Anoop Gupta, and Jaswinder Pal Singh, *Parallel Computer Architecture: A Hardware/Software Approach*, Morgan Kaufmann Publishers, 1997.

[8] William Dally and Brian Towles, *Principles and Practices of Interconnection Networks*, Morgan Kaufmann Publishers, 2003.

[9] M. De Nil, L. Yseboodt, F. Bouwens, J. Hulzink, M. Berekovic, J. Huisken, and J. van Meerbergen, *Ultra low power ASIP design for wireless sensor nodes*, Electronics, Circuits and Systems, 2007. ICECS 2007. 14th IEEE International Conference on, Dec. 2007, pp. 1352–1355.

[10] F. Duarte, *Technical Note TN-08-WATS-TP1-016: Low Power FIFO*, Tech. report, "Technology Program 1 - ULP DSP", Holst Centre / IMEC-NL, 2008.

[11] Gert Goossens, Johan Van Praet, Dirk Lanneer, and Werner Geurts, *Ultra-Low Power? Think Multi-ASIP SoC!*, IP 07 Conference, 2007.

[12] Wolfgang Haid, Kai Huang, Iuliana Bacivarov, and Lothar Thiele, *Multiprocessor SoC software design flows*, IEEE Signal Processing Magazine **26** (2009), no. 6, 64–71.

[13] Wolfgang Haid, Lars Schor, Kai Huang, Iuliana Bacivarov, and Lothar Thiele, *Efficient execution of Kahn process networks on multi-processor systems using Protothreads and windowed FIFOs*, ESTImedia, 2009, pp. 35–44.

[14] Andreas Hansson and Kees Goossens, *An on-chip interconnect and protocol stack for multiple communication paradigms and programming models*, CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis (New York, NY, USA), ACM, 2009, pp. 99–108.

[15] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken, *CoMPSoC: A template for composable and predictable multi-processor system on chips*, ACM Trans. Des. Autom. Electron. Syst. **14** (2009), no. 1, 1–24.

[16] Kai Huang, D. Grunert, and Lothar Thiele, *Windowed FIFOs for FPGA-based multiprocessor systems*, ASAP, 2007, pp. 36–41.

[17] Ahmed Jerraya and Wayne Wolf, *Multiprocessor Systems-on-Chips*, Morgan Kaufmann Publishers, 2005.

[18] Ahmed A. Jerraya, Aimen Bouchhima, and Frédéric Pétrot, *Programming models and HW-SW interfaces abstraction for multi-processor SoC*, DAC '06: Proceedings of the 43rd annual Design Automation Conference (New York, NY, USA), ACM, 2006, pp. 280–285.

[19] A.B. Kahng, Bin Li, Li-Shiuan Peh, and K. Samadi, *ORION 2.0: A fast and accurate NoC power and area model for early-stage design space exploration*, Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE '09., April 2009, pp. 423–428.

[20] Michael Keating, David Flynn, Rob Aitken, Alan Gibbons, and Kaijian Shi, *Low Power Methodology Manual: For System-on-Chip Design*, Springer, 2007.

[21] K. Lahiri and A. Raghunathan, *Power analysis of system-level on-chip communication architectures*, Hardware/Software Codesign and System Synthesis, 2004. CODES + ISSS 2004. International Conference on, Sept. 2004, pp. 236–241.

[22] K. Lahiri, A. Raghunathan, and G. Lakshminarayana, *The LOTTERYBUS on-chip communication architecture*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **14** (2006), no. 6, 596–608.

[23] Kangmin Lee, Se-Joong Lee, and Hoi-Jun Yoo, *Low-power network-on-chip for high-performance SoC design*, Very Large Scale Integration (VLSI) Systems, IEEE Transactions on **14** (2006), no. 2, 148–160.

[24] Bu-Ching Lin, Geeng-Wei Lee, Juinn-Dar Huang, and Jing-Yang Jou, *A precise bandwidth control arbitration algorithm for hard real-time SoC buses*, Design Automation Conference, 2007. ASP-DAC '07. Asia and South Pacific, Jan. 2007, pp. 165–170.

[25] Grant Martin, *Overview of the MPSoC design challenge*, DAC '06: Proceedings of the 43rd annual Design Automation Conference (New York, NY, USA), ACM, 2006, pp. 274–279.

[26] Giovanni De Micheli and Luca Benini, *Networks on Chips: Technology and Tools (Systems on Silicon)*, Morgan Kaufmann Publishers, 2006.

[27] MPI Forum, www.mpi-forum.org.

[28] OpenMP, www.openmp.org.

[29] Christian Piguet, *Low-Power CMOS Circuits: Technology, Logic Sesign and CAD Tools*, CRC Press, 2005.

[30] Francesco Poletti, Davide Bertozzi, Luca Benini, and Alessandro Bogliolo, *Performance analysis of arbitration policies for SoC communication architectures*, Design Automation for Embedded Systems **8** (2003), 189–210.

[31] Francesco Poletti, Antonio Poggiali, Davide Bertozzi, Luca Benini, Pol Marchal, Mirko Loghi, and Massimo Poncino, *Energy-efficient multiprocessor systems-on-chip for embedded computing: Exploring programming models and their architectural support*, IEEE Trans. Comput. **56** (2007), no. 5, 606–621.

[32] Katalin Popovici, Frederic Rousseau, Ahmed Jerraya, and Marilyn Wolf, *Embedded Software Design and Programming of Multiprocessor System-on-Chip*, Springer, 2010.

[33] Synopsys, www.synopsys.com.

[34] Andrew S. Tanenbaum, *Modern Operating Systems*, Prentice Hall PTR, 2001.

[35] Target Compiler Technologies, www.retarget.com.

[36] TSMC, www.tsmc.com.

[37] Harry Veendrick, *Nanometer CMOS ICs*, Springer, 2008.

[38] Virage Logic, www.viragelogic.com.

[39] Hoi-Jun Yoo, Kangmin Lee, and Jun Kyoung Kim, *Low-Power NoC for High-Performance SoC Design*, CRC Press, 2008.

Background on Low Power

A

Here, we give a brief summary on low power: the sources of power dissipation in CMOS circuits, and techniques on how power dissipation in CMOS circuits can be reduced. The reader is referred to [37], [29] or [20] for a comprehensive discussion.

A.1 Power Dissipation

For digital circuits implemented with CMOS technology, the total power dissipation is calculated as follows:

$$P_{total} = P_{dynamic} + P_{static} + P_{short} + P_{leakage} \tag{A.1}$$

where $P_{dynamic}$ is the dynamic power dissipation contributed by the charging and discharging of the gate capacitances, P_{static} the power associated with the current drawn from supply to ground under non-transient operation of the gates, P_{short} the power dissipation due to the short-circuit currents from supply to ground under transient operation of the gates, and $P_{leakage}$ the power dissipation associated with subthreshold and reverse-bias leakage currents. The contribution of the static power and leakage power has been relatively small for large technology nodes, but they are expected to rise significantly when moving into deep submicron regions.

The dynamic power dissipation for a circuit node is given by:

$$P_{dynamic} = \alpha \cdot C \cdot V^2 \cdot f \tag{A.2}$$

where α is representative of a node's transient activity, C the (effective) load capacitance of a node, V the supply voltage of a node and f the operating clock frequency. Hence, the dynamic power dissipation over time is largely data dependent.

During transient operation, a brief short-circuit current exists when both NMOS and PMOS transistors are on, between the supply voltage and ground. As well, there is a current charging the internal node capacitances. We can then calculate the short-circuit power dissipation as follows:

$$P_{short} = t_{short} \cdot V \cdot I_{peak} \cdot f \tag{A.3}$$

where t_{short} is the time duration of the short-circuit current, V the supply voltage of a node, I_{peak} the combined switching current (short-circuit current and internal charging current) and f the operating clock frequency. Since the short-circuit current is only likely to occur for a brief time during transients, the transient power dissipation will be dominated by $P_{dynamic}$ [20].

The static power dissipation in CMOS circuits is negligible, because ideally, there is no direct path between the supply voltage and ground under non-transient operation.

Thus, P_{static}, like P_{short}, is most of the time eliminated from the total power dissipation calculation for CMOS circuits. However, considerable leakage currents can be present, especially when using deep submicron technology.

We can separate the contribution to $P_{leakage}$ into two main sources. First of all, there is subthreshold leakage current, which flows from drain to source of a transistor operating in subthreshold mode, that is, when the transistor is not completely turned off. There is a exponential dependency of the subthreshold leakage current on the difference between the gate-source voltage V_{GS} and the threshold voltage V_T, so that when the supply voltage of a node and the threshold voltage are decreased to reduce the dynamic power dissipation, the subthreshold leakage power increases rapidly in the other direction. Secondly, there are substrate leakage currents that flow either from the gate or drain of a transistor to the substrate, but these have been relatively small compared to the subthreshold leakage current [20].

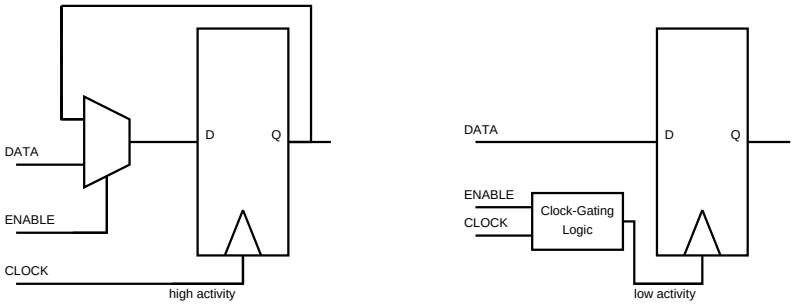
A.2 Low Power Techniques

Both technology choices and design implementations might affect the different components of the total power dissipation. For example, the short-circuit power P_{short} of a buffer can be reduced by using a tapered design, which is a design measure. On the other hand, we can use high threshold voltages – a technology measure – to reduce the leakage power dissipation [37].

Techniques for low-power design can be applied at various levels in the design space: from the system and chip level all the way down to the gate and layout level. The effects of techniques applied at a higher level can have more effect in reducing power. If a section of a chip can be powered off cheaply when not needed, the gains in reduced power dissipation will be larger than if one was to try to minimize the leakage power in that same section at the gate level by using multi-threshold voltage standard cells, for example.

A popular technique to reduce the dynamic power dissipation at the RTL level of a design is *clock-gating* (Figure A.1). The synchronous components in a design continue to be clocked even though the input data to a component might not be changing. With clock-gating, the clock to these components is selectively disabled, thereby reducing the power dissipation associated with unneccesary clock toggling, but at a cost of increased hardware which will also slightly increase leakage power.

With *power-gating* a reduction in leakage power is targetted. The idea is to switch off sections of the chip that are not currently in use. If a system has long period in which it is idle, the power dissipation will be largely dominated by leakage power. But switching on and off chip sections cannot be done in zero time, and thus power-gating introduces additional timing delays in a design, as well as some extra hardware overhead in the form of control circuitry. Figure A.2 shows the power profile of a system that includes sections that are power gated, not taking the hardware overhead into account, and assuming clock-gating is used. In the idle mode, the design will only dissipate leakage power (because of clock-gating), but because power gating is used for certain sections, the leakage power will be less than for the active mode.



(a) Before Clock-Gating (b) After Clock-Gating

Figure A.1: Clock Gating

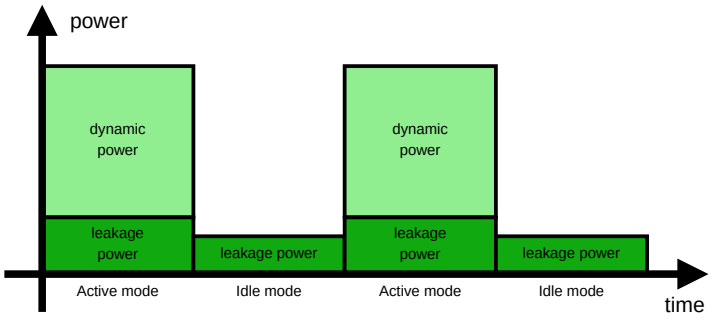


Figure A.2: Power Gating Power Profile

For more power-reduction techniques, the reader is referred to the sources mentioned at the start of this appendix.

B Hardware Design Flow

Here, we briefly report on the tool flow used to implement the designs and to obtain power dissipation numbers. See Figure B.1 for a combined illustration of the individual steps.

B.1 Synthesis

The designs are described in the hardware language VHDL, giving us a register-transfer level description (RTL). After error-checking the VHDL code, the first step of the synthesis consists of elaborating the design and setting timing constraints, as well as the optional insertion of the clock-gating circuitry. Then, the design is mapped to standard cells from a target technology library, assuring that the giving timing constraints are met. When timing closure is possible, an intermediate netlist is generated, as well as a corresponding design constraints (SDC) file that will help in the next step of the flow. We use *RTL Compiler* from *Cadence* [4] to complete this step of the flow.

B.2 Place and Route

In this step of the flow, the design is physically implemented. To the synthesized netlist and SDC file we attach new constraints, such as the size of the design floorplan and the number of metal layers for power routing. The standard cells are placed on the floorplan, whilst assuring that timing requirements are still met. Then the design clock tree is generated, with multiple optimization iterations necessary if timing is not immediately met at this point in time. Finally, everything is connected together, and a new netlist is generated, as well as information on the parasitics present in the design. This step of the flow is completed using *SoC Encounter* from *Cadence.*[4]

B.3 Timing Analysis

We use timing analysis to annotate the netlist created by the place and route step, combining the timing details of the standard cells and interconnect, so that consequently, reliable power dissipation numbers can be generated. *Synopsys Primetime* [33] is used to complete this step, with a resultant SDF (Standard Delay Format) file containing the results from timing analysis.

B.4 Power Extraction

The final step of the flow consists of obtaining power dissipation numbers for the design. We simulate the design using *Cadence NC-Sim* [4] in order to generate switching data, which also requires the timing analysis data from the post layout netlist. Power extraction is then performed, using *Synopsys Primepower.*[33]

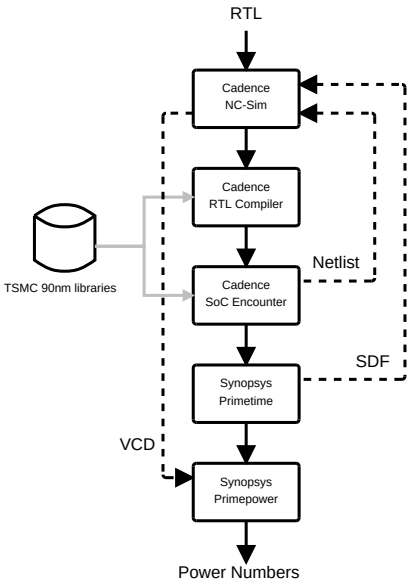


Figure B.1: Hardware Design Flow

C

FIFO Design

A FIFO is an abstract way of data organization and manipulation, both temporally and presendence-wise. Data is processed as in a queue: data that comes in first, is handled first, data that comes in second waits until the first data is processed, and so on. Eventually, every data is processed in the order that it arrived. FIFOs are commonly used for buffering and flow control (synchronization), both in software and hardware applications.

In hardware, FIFOs are customarily implemented using a set of read and write pointers, some storage logic where data values are buffered, and control logic. With the read and write pointers, we can basically implement a circular queue over the storage area. Initially, both read and write pointers locate to the first storage location, denoting that the FIFO is *empty*. The read pointer will be incremented when we write to the FIFO, while the write pointer is incremented when read from. If after we increment the read pointer we find that it again points to the same location as the write pointer, the FIFO can signal it is *empty* again. The FIFO will signal that is *full* when after the write pointer is incremented, it points to same location as the read pointer.

The storage area in a FIFO can be constructed in various ways. Any hardware element that can store some binary information can be used, but typically random access memory (RAM), flip-flops (FF) and latches are used. RAM optionally has dual-port implementations, which allow us to write to one port, and read from the other. For the read and write pointers we also have several options for how they address the storage area: binary encoding, one-hot encoding or Gray encoding, so that internally to the FIFO, a address generation unit (AGU) might be required. See Figure C.1 for a basic outline of a FIFO implementation in hardware.

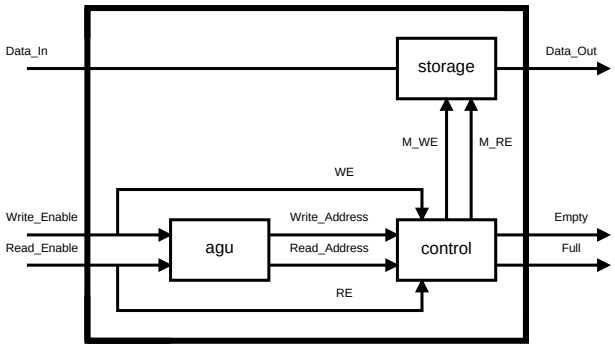


Figure C.1: FIFO Block Diagram

Previous work at IMEC-NL [10] has focused on the power dissipation results for different FIFO implementations. It showed that for the address encoding, there was

no benefit on using one-hot encoding over binary encoding, and that for small FIFO sizes, latch-based implementations have lower active and leakage power dissipation than flip-flop based FIFOs. Furthermore, it was shown that the situation for random-access memory was even worse in terms of power dissipation, both active and leakage. Therefore, for lower power designs (that have no high speed requirements) random-access memory should not be used as FIFO storage area.

Target Processor Design Tools

D

This appendix briefly summarizes the Target processor design tools. The tools from Target Compiler Technologies allow for the design, programming and verification of retargetable application-specific instruction set processors (ASIPs) [35].

The processors are designed with a markup-language, with which one can define for example the processor instruction set, instruction set encoding as well as the mapping of instructions to specific functional units. Software for the processors are described with C programming language. The Target tools can automatically generate the instruction-set simulator for the designed processor, as well as a compiler that allows the generation of machine code for the specific processor. From the processor model, a synthesizable hardware description in VHDL is generated.

D.1 Base Processor Description

Target delivers a sample 16-bit processor core than can be used as starting point for ASIP design, the "base" processor. The base processor features a set of general purpose instructions, as well as 16-bit arithmetical, logical and comparison instructions, integer multiplication and shift operations. It offers load/store instruction to 16-bit data memory, with a 64k address space using indirect addressing. Address computations are executed on a 16-bit address generation unit. Furthermore, the processor can be configured for the use of interrupts, on-chip debugging and zero-overhead loops. The datapath of the processor is depicted in Figure D.1, and includes a register file and data memory (dm), an adress generation unit (agu), an arithmetic logic unit (alu), shifter and finally a multiplier-accumulator.

For the processor implementation of interrupts, several instructions are provided, as well as a *halt* instruction, which can disable program execution on the processor, which then asserts a halted output signal that can notify external components of the processor state.

The zero-overhead loops allow for fast looping over a block of instructions. It is setup by *do* instruction, with no additional execution cycles needed to perform the looping tasks. The loop is executed a predetermined number of iterations, and controlled by a dedicated hardware block.

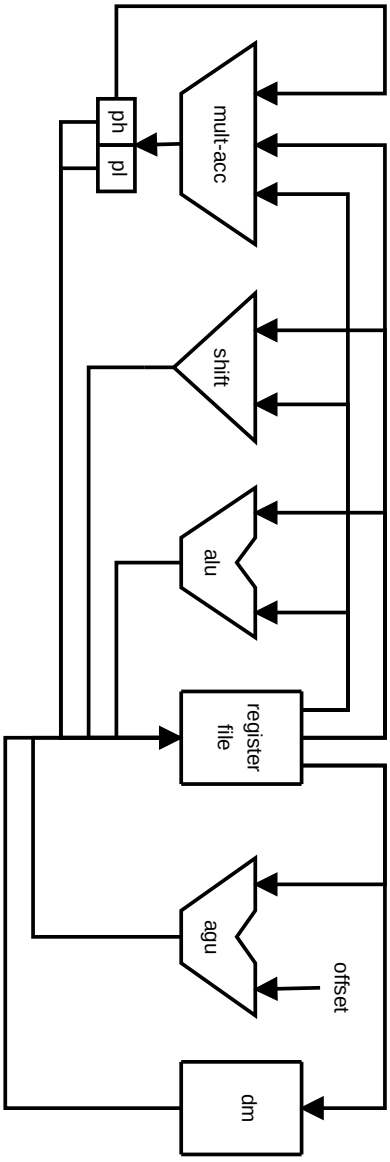


Figure D.1: Base Processor Datapath