# Design of a Snoop Filter for Snoop Based Cache Coherency Protocols

Rasmus Ulfsnes

Master of Science in Electronics
Submission date: June 2013
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Øystein Gjermundnes, ARM

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

# Abstract (English)

Multi core architectures has become common in mobile SoCs; not only for CPUs, but also for mobile GPUs. With the introduction of OpenCl for mobile GPU architecture, the SoCs are able to become more powerful than before. Because programs that were executed on the CPU before, can now be executed faster on the GPU. Along with this the need for cache coherence protocols has also been introduced. Snoop based cache coherence protocols inherently leads to extensive coherence traffic on the bus in a multi core system. All this traffic leads to tag lookups in remote data caches. However, recent research shows that these lookups and coherency traffic, are by a large extent unnecessary. In other words a lot of power is wasted by transmitting unnecessary snoop requests over a interconnect. This project has explored one possible solution to reducing these requests: Snoop Filters.

Previous research has been done for CPUs with SPLASH and other benchmark suits. This thesis however, will look at coherence transactions and lookups from a GPU perspective. To be able to thoroughly analyze coherence transactions from OpenCl benchmarks, a parameterizable multi core model has been constructed. The model is capable of replaying OpenCl benchmarks after executing on a ARM-MALI T6xx GPU. The results show that similarly to CPU benchmarks, the coherency traffic induced by OpenCl benchmark also end up in cache misses. Recent research also shows that CPU coherency protocols using the MESI states, instead of just MSI, reduces the unwanted coherency traffic. The reduction is so big that much that snoop filters and other coherence limiting approaches were unnecessary. The research done for this thesis has shown that this is not the case for GPUs, as the MESI protocol does not reduce power consumption in a multi core GPU. Because of this, snoop filters based on the CSR(Ranganathan & Charbon 2012) filter were explored.

The analysis in this thesis of the original destination based CSR filter, showed that the filter reduced the unnecessary tag lookups to around 53% for the OpenCl benchmarks. This means a great underlying potential in how the resources are selected according to the address stream. The analysis also showed that a fair deal of the snoop induced transactions also were unnecessary.

Based on the filter analysis two new filters were designed:

- **Source-CSR**
- **Hashed-index CSR**

Although source CSR represents more hardware overhead compared to the destination filter, it is capable of reducing 30% of the snoop transactions. The source filter is also capable of a 53% reduction of the tag lookups. The hashed index filter was inspired by the potential in reducing the tag lookups further than 53%. The filter was capable of a 56% reduction. Although this is only 3% improvement over the normal filter, the filter performed remarkably for a number of benchmarks. Unfortunately this was not the case for all benchmarks. It shows that dynamic allocation of filter resources is capable of further reduction. The best case scenario would have been to use the original resource selection on some of the benchmarks, and the hashed index system on the others.

The source filter was also implemented in Verilog HDL, and formally verified in JasperGold using SystemVerilog. The filter was supposed to be power simulated, but some unknown error in the switching activity conversion halted any further power estimation. A proper conclusion about the power saving potential for the source filter can therefore not be made. This thesis does however include a power estimation methodology in order for the power estimation to be completed in the future.

# Sammendrag (Norsk)

Flerkjerne arkitekturer har blitt standard i mobile SoCer, ikke bare for CPUer, men også for mobile GPUer. Med introduksjonen av OpenCl for mobile GPU arkitekturer har SoCer blitt mer kraftige enn noen gang. Dette fordi programmer som tidligere ble kjørt på CPUen nå kan bli kjørt raskere på GPUen. På grunn av dette kom også behovet for cache coherency protokoller. Snoop baserte coherency protokoller fører til voldsom coherency trafikk på bussen i flerkjerne systemet. All denne trafikken leder til tag oppslag i de andre nodene i systemet. Samtidig har nyere forskning vist at disse oppslagene og derfor trafikken for det meste er unødvendig. Detter fører da også til at systemet bruker unødvendig mye energi. Dette prosjektet har utforsket en mulig løsning på dette: Snoop Filtre.

Tidligere forskning har kun blitt gjort for CPUer med SPLASH og lignende benchmarker. Denne master oppgaven derimot vil studere coherency transaksjoner og oppslag fra et GPU perspektiv. En parameteriserbar model ble konstruert for å være istand til å grundig analysere transaksjonene som oppstår ved kjøring av OpenCl applikasjoner. Modellen er istand til å rekonstruere transaksjoner fra OpenCl applikasjoner som ble kjørt på en ARM Mali-T6xx GPU. Resultatene viser at akkurat som for CPU applikasjoner ender de fleste transaksjonene for OpenCl applikasjonene også opp i en cache-bom. Nyere forskning har også konkludert med at CPUer med MESI coherency protokoller, reduserer denne unødvendige traffiken i forhold til MSI protokollene. Reduksjonen er så stor at snoop filtre og andre systemer for strømreduksjon blir unødvendig. Denne master oppgaven vil derimot vise at dette gjelder ikke for GPUer, og at MESI protokoller ikke har noen positiv effekt på strømforbruket. På bakgrunn av dette ble snoop filtre basert på (Ranganathan & Charbon 2012) undersøkt.

Analyse av det orginale destinasjons baserte CSR filtret viste at filtret var istand til å redusere oppslagene med 53% for OpenCl applikasjonene. Detter betyr at det er et stort potensiale i hvordan filter velger ressurser avhengig av adresse strømmen. Analysen viste også at mange av snoop transaksjonene også var unødvendig.

På bakgrunn av filter analyse ble to nye filtre utviklet:

- **Source-CSR**
- **Hashed-index CSR**

Selv om source CSR representerer mere hardware kostnad sammenlignet med destinasjons filteret, er det istand til å redusere snoop transaksjonene med 30% i

forhold til det orginale filteret. Samtidig er det også istand til å redusere tag oppslagene med 53%. Hashed-index filteret var inspirert av potensialet til å redusere oppslagene med mer enn 53%. Filteret er istand til å redusere oppslagene med 56%. Selv om dette er kun 3% mer enn det normale filteret var filteret utmerket for noen av applikasjonene. Desverre så var det mye dårligere enn det orginale på andre. Dette viser at en dynamisk allokering av ressursene er istand til å oppnå høyere reduksjon. Det beste hadde vært om the orginale filteret system ble brukt på noen av applikasjonene, og at hashed-index filteret blir brukt på de andre.

Source filteret ble også implementert i Verilog, samt formelt verifisert i JasperGold ved hjelp av SystemVerilog. Det var også meningen at filteret skulle effekt simuleres, men en ukjent feil i svitsje aktivitet konversjonen stoppet dette forsøket. På grunn av dette kan ikke en skikkelig konklusjon angående effekt reduksjonen til filteret bli gjort. Forøvrig har denne oppgaven utviklet en metode for å estimere effekt forbruket, slikt at dette kan bli gjort senere.

# Contents

# List of Figures

# List of Tables

# Abbreviations

| | | |
|---|---|---|
| SoC | = | System on Chip |
| SMP | = | Symmetric Multiprocessing |
| DST | = | Destination |
| SRC | = | Source |

# Chapter 1

# Introduction

The number of transistors per chip has increased following Moore's law since the 1960s. This ever increasing transistor count has made it harder for designers to utilize the extra transistors in CPU architectures. Because of the increase in design complexity the single core CPUs has evolved into multi core architectures. By duplicating the single core architecture, the hardware designers are able to utilize more of the available transistors on the silicon. Another reason for creating a multi core CPU was because the *instruction-level-parallelism* (ILP) (Hennessy & Patterson 2006) had been fully exploited. This paved the way for *thread-level-parallelism* (TLP) where the CPUs are designed to utilize the parallelism between program-threads. By using a multi-core desig the TLP can be further exploited, because different threads can run in parallel on different cores instead of sharing one single core.

While multi-core architectures has made it easier for designers to utilize the increasing transistor count, and further exploit the increasing TLP in consumer systems, it has also introduced some challenges in order to maintain *coherence* among the caches in the system. This is also known as the cache coherence problem.

For mobile SoCs another problem has arisen, the solutions to the coherence problem was aimed at maximum performance not conserving power, which is a big problem for mobile SoCs. Recently the interaction between the modules on a single mobile-SoC has increased. By introducing OpenCl for mobile GPUs it has been possible for operations that has been normally executed on the CPU to be run on the GPU. This of course has introduced the same *cache coherence problem* as for the CPUs. The work done in (Ulfsnes 2012) showed that snoop filter was a possible solution to reduce the power consumption in cache coherent systems. Especially the CSR filter by (Ranganathan & Charbon 2012) was especially interesting.This thesis will consist of four parts.

1. Theory:

    - Cache coherence protocols will be presented, and an introduction to snoop filters in chapter 2. The *stream register* and *counting stream* *register* filters will be presented in chapter 4. AMBA/ACE was investigated, but deemed to time intensive to work on in this thesis.

2. Modeling and Evaluation:

    - Chapter 3 presents a model created for evaluating different filters and coherence protocols. The model is able to create synthetic benchmarks as well as using OpenCl benchmarks from MALI-T6xx GPUs.
    - Newly designed filters will be presented in 4.
    - The old and new filters will be compared in 4.

3. Design and Verification:

    - RTL design of a new filter will be presented in chapter 5.
    - The RTL code will be verified in chapter 5.

4. Power Analysis: In addition to the thesis statement chapter 6 will analyze the power consumption of the filters to figure out if the new filter are useful for implementation.

The Python model used in this project can be provided by sending an email to rasmus.ulfsnes@gmail.com.

## 1.1 Thesis Description

Design of a AMBA ACE snoop filter for snoop based cache coherency protocols

Introduction

A multiprocessor is a tightly coupled computer system having two or more processing units each sharing main memory and pheriperals, in order to simultaneously process programs.

The shared memory system in a multiprocessor usually contains a hierarchy of caches where the lowest cache levels are private to each individual processing unit and the last level cache is shared among all the processing units. In such a memory system it is important to ensure the local caches are always in synch so that no core ends up processing old data. In other words, we need some system that can maintain cache coherency.

One popular class of cache coherency protocols is snoop based cache coherency protocols. In short a memory system with a snoop based cache coherency protocol will have infrastructure that allows each cache to monitor or snoop the shared bus for addresses that it has cached. Based on this information the cache will take actions to ensure coherency is maintained.

Each core in a snoop based system with N cores will have to monitor traffic from the N-1 other cores. With N cores in a system the snoop traffic becomes proportional to N(N-1). Much of this traffic is however wasted it is very rare that a given cache line is shared among all the cores. In order to reduce the cache coherency traffic it is possible to insert a snoop filter. A snoop filter has knowledge about which cores that has cached what, and uses this information to filter away requests that is of no interest to the different cores.

Thesis statement

This master project aim to create a snoop filter for integration into an AMBA ACE based memory system. The project can be divided into the following tasks:

- Literature study:
  - Study the AMBA ACE and AMBA AXI4 protocols.
  - Study typical snoop filtering techniques. Understand what the factors are that affect the efficiency of snoop filter as well as well as how different heuristics will affect the size, performance and power consumption of the design.
- Architecture, Model and benchmark:
  - Suggest filter strategy, create a model and measure the performance of the filter.
  - Create a synthetic benchmark that has a lot of shared data, and demonstrates the performance of the filter for different scenarios.
  - Retrieve buslogs from ARM for interesting real life benchmarks that you test on your model.
- Design and test:
  - Identify the core of the filter and create a spec for that.
  - Write RTL code for the filter.
  - Test that the filter works. Preferably using both assertion based verification and creating a lightweight constrained random UVM based testbench.

Chapter 2

# Memory Systems in Multicore SoCs

## 2.1 Cache Coherence

Along with the introduction of the multi-core processer came the notion of shared memory. In any multi-core system the different cores at some time or another will share some memory. In figure 2.1, a typical multi-core system is depicted. There are a number of cores which are all connected to a shared LLC(last-level-cache), through some interconnect. The core consists of a CPU, a cache controller and a L1-Cache.

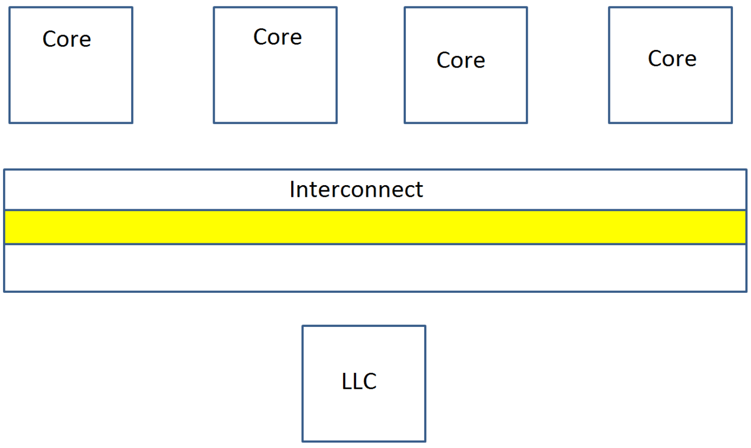
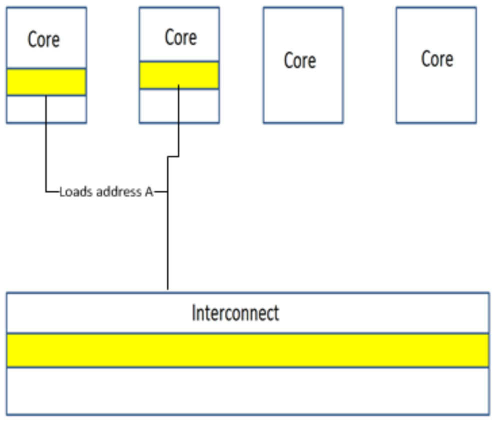
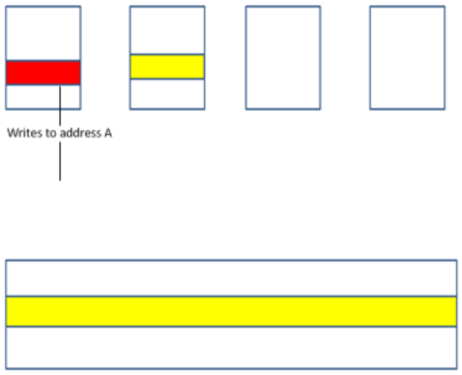


Figure 2.1: Multicore System with 4 cores connected through an interconnect

When two or more cores share cache lines the *coherence problem* occurs when one of the cache controllers wants to change one of the cache lines. If the other controllers are not made aware of that alteration they may end up using stale data.



(a) Time: t=0: A cache line is loaded in two cores



(b) Time: t=1: One of the cores writes to the shared cache line, the system is incoherent

Figure 2.2: Incoherent multi core system

In figure 2.2a, at some time t=0, two of the cores has loaded the same address. The cache line is unaltered, hence the yellow color. Consider that at time t=1, the leftmost core will execute a write operation that particular cache line, figure 2.2b. The write changes the data hence the red color. After the doing the write the two cores has two different versions of the same cache line, leaving the system in an incoherent state. Then if at time t=2 the core with the yellow cache line do some computations with those data it will in fact use old and incorrect data. This is known as the coherence problem.

### 2.1.1 Coherence Protocols

The coherence problem is solved through coherence protocols. There are several different protocols:

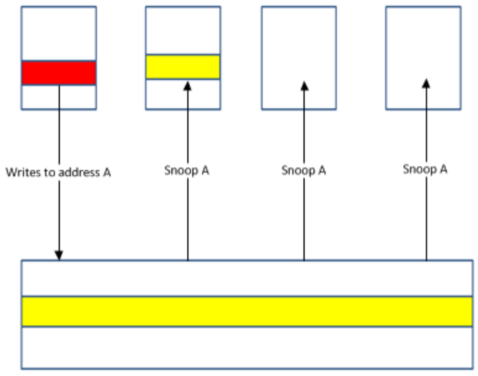
- *Snoop based protocols*
- *Directory based protocols*
- *TokenB*
- *Snarfing*

*Snoop* based protocol is the most popular protocol because of its simplicity and influence in desktop and server solutions, as well as in recent SoCs. It relies on broadcasts, and snooping of those broadcasts, to ensure coherence. *Directory* based protocols employs a directory where information about each cache line is stored. This way the directory has full control over which core has loaded which cache line. *TokenB* is a protocol where each cache line has a number of tokens associated with it. The number of tokens has to be at least as high as the number of cores. Any core which possesses at least one token can read, any core that has all the tokens for the cache line has write privileges. *Snarfing* directly updates local cache data without going through a centralized memory.

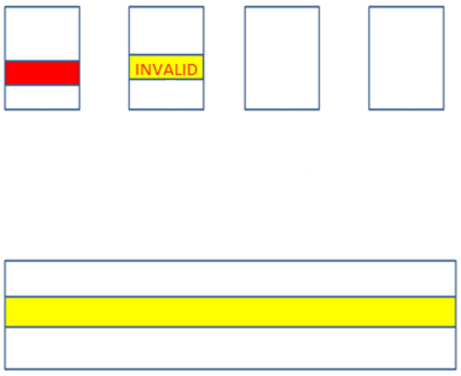
#### 2.1.1.1 Snoop based protocols

Snoop based coherence protocols has because of its simplicity and low overhead become very popular in any multi-core system. The basic idea behind snooping protocols is that every update to a cache line is broadcast to every other controller, so they in turn can check the cache and see if the specific cache line is present in the cache.

Consider the same scenario as in figure 2.2a where two of the cores has address A loaded. Then consider figure 2.3a, which is an alternative version of figure 2.2b. When a snoopy protocol is used, the core will send invalidate requests to all other cores in the system. Then at time t=2 , figure 2.3b, if the core wants to use the data from that particular cache line it will have to request the updated data from the owner of the cache line. Which depending on the protocol will be either the LLC or the controller that updated the line.



(a) Time: t=1: When the write occurs the controller issues invalidate requests to all the other controllers



(b) Time: t=2: If the controller with the yellow cache line wants to read the data it has to ask the owner of the cache line for the updated value.

Figure 2.3: Snoop based protocol execution

The most common protocols use a subset of the MOESI state model (Daniel J. Sorin 2011). Here the MSI are the most basic states, E and O are additions to the MSI state space. All of these states build on some basic characteristics; *validity,dirtiness*, *exclusivity* and *ownership* (Daniel J. Sorin 2011)[p.89].

Table 2.1: MOESI Protocol States

| State: | Description |
|---|---|
| **M(odified)**: | The line is exclusive, owned and valid. It can be read from, or written to, at any time without notifying other controllers. The line is therefore potentially dirty, meaning that this is the only updated version of the line. |
| **O(wned)**: | The line is potentially dirty and valid. The line may be stale in the LLC, meaning that the controller needs to respond to requests. It is not exclusive as other cores may have read-only privileges. |
| **E(xclusive)**: | This line is not dirty, but valid and exclusive. This line is only present in one cache, and the LLC has an up to date version. |
| **S(hared)**: | The line is valid, clean and not exclusive, other caches may have the cache line for reading. |
| **I(nvalid)**: | The line is invalid, the cache line is not present in the cache. |

The motivation for adding the exclusive state was in case a cache first reads to an address, then a immediate write to the same address can be performed without notifying other caches. Thus reducing coherence transactions.

The owned state was motivated by redundant updating of the LLC when a cache needs to respond to a load snoop request. Instead of going from M or E to S, the controller goes to O. The line will not be written to the LLC, and it also avoids future writes to the LLC as the cache controller owns the data.

This project will focus on the basic MSI with the addition of the exclusive state(E). A recent survey (Ulfsnes 2012)(Ulfsnes) offers more information about states, transient states and specification of protocols.

## 2.2 Power Consumption in Multi-Core System

Snoop based protocols are very popular because of its simplicity and low-latency coherence transactions, meaning that every core gets the invalidate request in parallel. This parallel handling of a request is very performance friendly if not very power efficient. In (Ulfsnes 2012) a thorough analysis of power consumption in coherent multicore systems are given. One of the biggest problems with snoop based coherence protocols from an energy perspective, is that even thought the performance is better off with parallel broadcast to every node, the power consumption will also be higher than necessary.

In any multicore system the amount of shared cache lines between cores will be limited, only 10% needs coherence actions (Nilsson et al. 2003). This means that most of the coherence requests will end up wasting power both in the interconnect as well as in the caches. From (Ulfsnes 2012) 2-7% of the total energy consumption can be reduced by removing unnecessary coherence requests.

Several solutions has been proposed to limit the unnecessary tag lookups. The survey in (Ulfsnes 2012) describes the solutions in more detail, but the solutions can be roughly divided into two camps:

1. *Snoop Filters*
2. *Power Saving Protocol Design*

Number 2. requires architectural changes to already designed systems, including relaxed consistencies and different bus interface amongst others. *Power saving protocol design* is therefore not that useful when trying to reduce power in architectures that already exists.

Snoop filters on the other hand requires less architectural changes. That is why this thesis has focused on researching snoop filters for mobile GPU solutions.

## 2.3 Snoop Filters

In order to improve the energy consumption without sacrificing performance snoop filters was introduced by Moshovos et al.(Moshovos et al. 2001). Snoop filters are basically directory based structures that has less lookup costs than a cache lookup. A snoopy system without filters can be seen in figure 2.4.

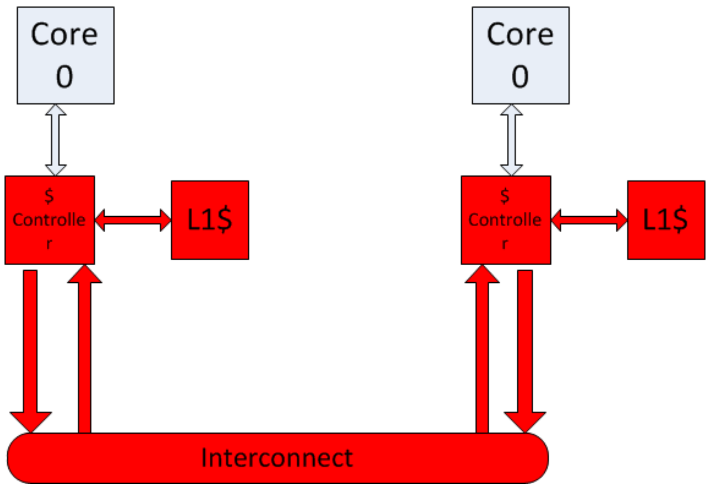


Figure 2.4: What is affected by broadcasting coherence requests

The red areas are the parts of the system that will consume power when subject to any coherence requests. There are two types of filters, *source* and *destination*, which can be subdivided into *inclusive* and *exclusive* as well.

### 2.3.1 Source Filter

Source filter is a type of snoop filter that filters the coherence requests before they reach the interconnect. They will therefore reduce the energy consumption from the interconnect as well as the tag lookups in remote caches.

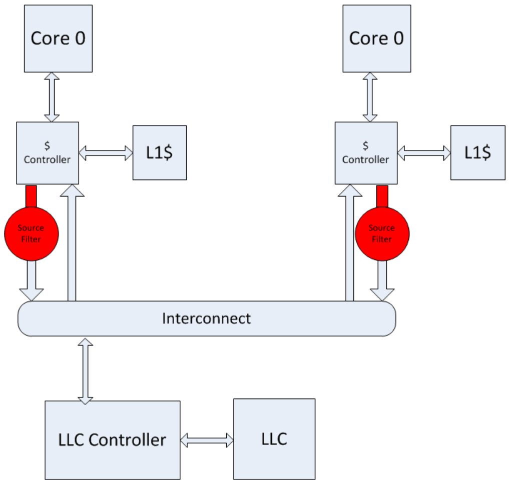


Figure 2.5: Source Filter

Figure 2.5 shows that the source filter will limit the red areas compared to figure 2.4. The introduction of the filter itself will introduce a new source of energy consumption and therefore has to be considered when looking at the net energy reduction.

### 2.3.2 Destination Filter

The destination filter sits on the destination side of the coherence transaction. It will therefore only be able to filter transactions going through the TLB and into the cache. Figure 2.6 shows that the red are expanded to include the interconnect.

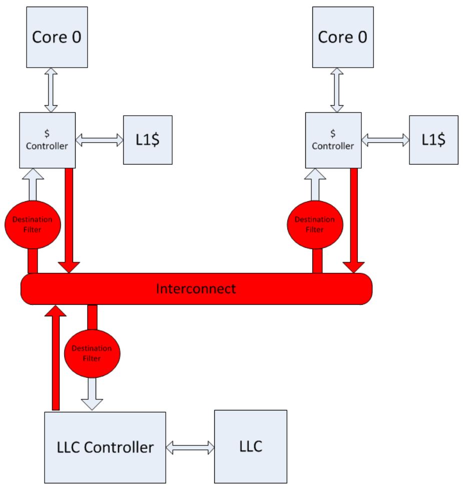


Figure 2.6: Destination Filter

### 2.3.3 Exclusive/Inclusive

The filters can also be divided into *exclusive* and *inclusive*: **Inclusive**: An inclusive filter holds a *superset* of all the addresses that is currently cached. Any miss occurring in the filter will definitely occur in the cache.

**Exclusive**: An exclusive filter is a filter which holds a *subset* of all addresses that is not currently cached, it will filter based on what is not in the cache. Any hit in the filter will definitely not be in the cache.

### 2.3.4 MESI Protocols Make Filters Redundant

Recent research (Ranganathan & Charbon 2012) however has shown that the MESI protocol as compared to the MSI, will to a great extent reduce the number of coherence transaction, acting as a source filter. Making filters redundant. This research was done for embedded CPUs, while this project will consider GPUs with OpenCl application, which has different shared memory footprints. Where MESI protocols does not have the same effect as for CPUs. This will be presented in section 4.2.1.

## 2.4 Summary

The most popular snoopy coherence protocol has limitations when it comes to preserving power, it has been optimized for performance on desktop and server systems. The coherence protocol will broadcast requests to every remote core, without knowing if the cache line associated with the request is in the cache. Recent research into limiting snoop power consumption has introduced snoop filters. Late research has shown that MESI has great effect on snoop induced power reduction in SoCs. This project will focus on GPUs with OpenCl applications, where MESI is not as effective as for CPUs.

Chapter 3

# Model for Evaluating Snooop Filters

In order to find out which filters are effective with GPU benchmarks a model that can use GPU memory traces as stimulus had to be constructed. The model overview can be seen in figure 3.1.

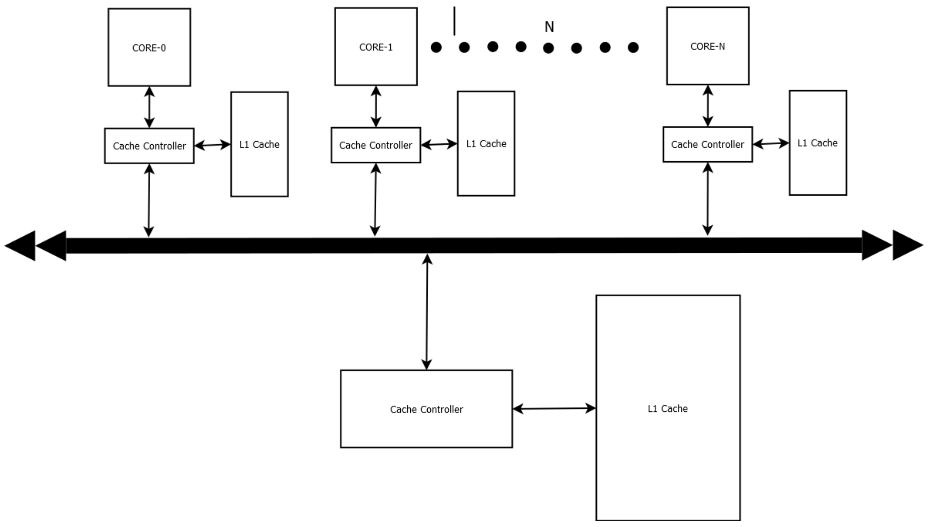


Figure 3.1: Memory System Architecture

The system supports 1 to N number of cores with separate cache-controllers and L1 cache. All of the controllers and interface are interchangeable so that different configurations are supported. The baseline model includes MSI and MESI cache coherence protocols. The L1 cache controller can however be altered to any coherence protocol.There may also be a need to change the interconnect and the L2 cache controllers depending on the wanted protocol. The L1 caches are 32kb 4-way set associative by default.

All of the cores share one L2 cache, and the capacity of the L2 is infinite. Meaning that whenever a L1 cache miss occurs, it will always find the missed cache line in the L2. The reason for this is to avoid modeling higher levels of memory. The stimulus for the model can be created randomly, using Pythons built in random function. It can also replay memory operations from Mali-T600 GPUs.

## 3.1 Evaluation Criteria

The goal of the model is to measure the energy impact of coherence induced traffic and snoop lookups. As stated earlier this can have a significant impact on the overall power consumption.

### 3.1.1 Tag Lookups

There are two types of snoop induced tag lookups, both of which are counted in the L1-controller:

- **Necessary**: These are the tag lookups that are necessary for maintaining system coherence.
- **Unnecessary**: These are the tag lookups that miss in the cache, and therefore waste power.

The model returns both these numbers as:

- **Necessary** = Number of snoop induced hits in the cache.
- **Unnecessary** = Total number of snoops - number of necessary snoops.

### 3.1.2 Snoop Induced Transactions

In order to evaluate source filters, the snoop induced bus traffic also has to be monitored. The model will report the total number of snoop induced transaction. It can not however differentiate between unnecessary/necessary transactions

## 3.2 Architecture

The architecture of the model can then be changed to measure the performance. This can be done with filters, as figure 3.2 illustrates, or without a filter structure as shown in fig 3.1. The system consists of a dynamic number of cores with L1-cache and cache-controllers. The highest level of memory is the L2 cache. All the communication goes through a simple shared interconnect.

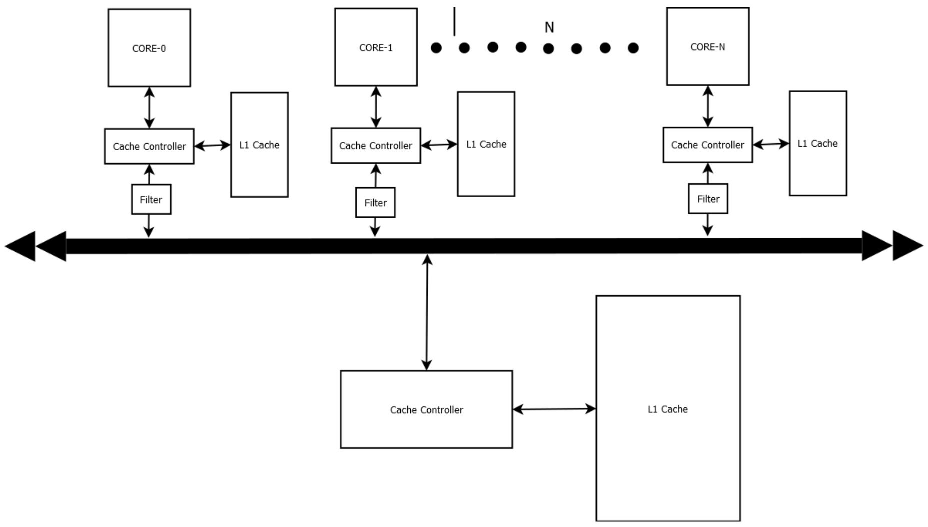


Figure 3.2: Memory System Architecture with Filter

### 3.2.1 Core

The core is basically a transaction pusher. It will have a set of read or write instructions, which it will request the cache controller to perform. Whenever the cache controller is busy, the core will wait until the controller is done before requesting a new operation.

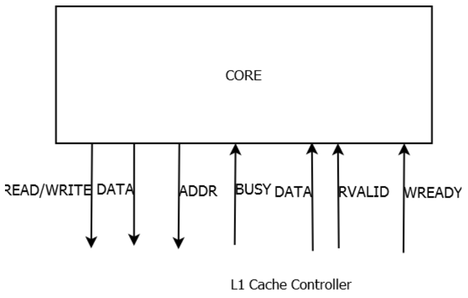


Figure 3.3: Core Module in Model

Figure 3.3 shows the interface of the core module. As long as the busy signal is not high, the core will order a new load/store operation and stall. Whenever WREADY or RVALID goes high a new operation can be started.

At the start of the simulation the core will get its instruction queue filled from three possible sources:

1. **UTLB Logs**: The Mali-T600 GPU consist of a number of cores which all have their own LSC (load-store-cache). The core uses virtual addresses,

while the LSC uses physical addresses. The address going to the LSC therefore needs to be translated using a UTLB(micro-translation-lookaside-buffer). The operations going through the UTLB is captured in a log. The log is then parsed by the model and converted into instructions for the core.

2. **Random Stimuli Generation**: The instructions for the core can also be randomly created. The user can control different parameters of the created stimuli:
   - **Length of Simulation**: The user can define how many operations there are per core.
   - **Address Range**: The range of the address space can be defined.
   - **Sharing**: The user can define the probability for multiple cores sharing the same addresses by altering the ratio: $\frac{nr\_operations}{address\_range}$
   - **Chunk Size**: The number of consecutive addresses can be changed using this variable. If the size is set to 10 the generator will randomly select an address and create 10 consecutive load/stores addresses from the randomly selected address.
3. **Playback**: The randomly created stimuli will be stored in a log file, so that the same stimuli can be tested on different architectural configurations.

### 3.2.2 L1 Cache

The L1 cache is a 32kb 4-way set associative data cache. Each cache line is 64 bytes, and there are 128 sets in the cache. The cache controllers are dynamic in the sense that it does not care about how many sets, or how many cache lines, there are in the cache. The cache architecture can therefore be reconfigured without changing the controller.

### 3.2.3 L1 Cache Controller

The L1 cache controller is the heart of the memory system. It handles requests from the Core as well as snoop requests from remote controllers. The controller is implemented using an FSM that represent the desired coherence protocol.

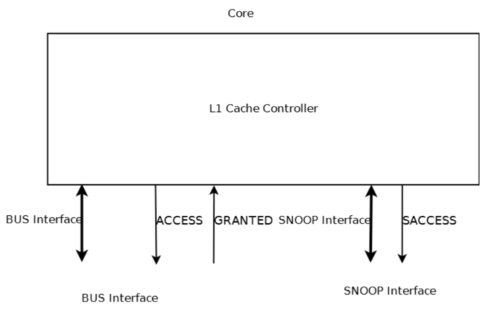


Figure 3.4: L1 Controller Module

Figure 3.4 shows the controller module and its connections. The bottom part is the connections to the bus and snoop interface. The access signals SACCESS and ACCESS are signals sent to the interface asking for access to snoop and bus interfaces respectively.

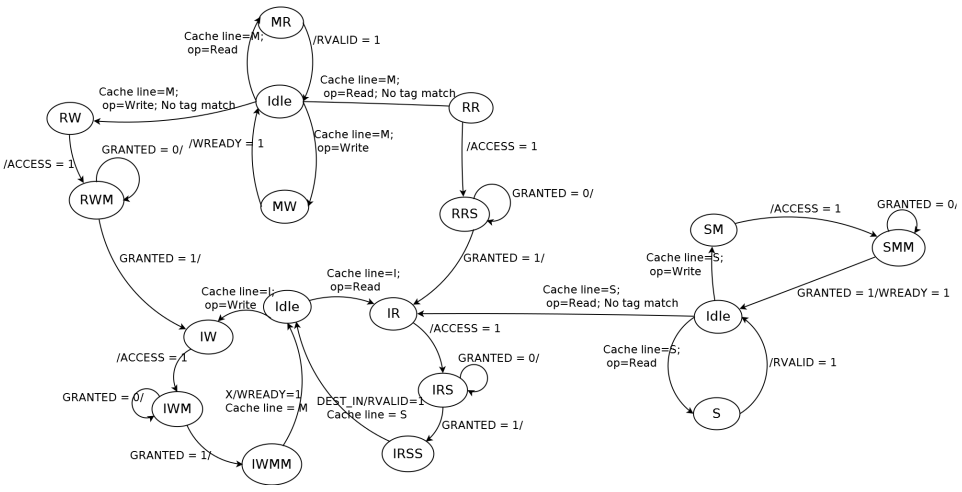


Figure 3.5: FSM for MSI cache controller

The MSI FSM can be seen in figure 3.5. The FSM for the controller was implemented using the specifications given in (Daniel J. Sorin 2011)[p.107]. There are three sub-FSMs in figure 3.5, covering the M, S and I state. Each sub-FSM has two different paths depending on whether the operation is read or write.

Example: Consider that the cache line has not been loaded into the cache yet. In other words the line is invalid. Figure 3.6 shows the sub-FSM for state I.

From an idle state the controller will enter state IW. IW will set the ACCESS and SACCESS(section 3.2.4) signals high and move to IWM where the signals are set low. If the controller uses a source filter ACCESS can be the only signal set high if the filter says the address is not cached in any other node. When access

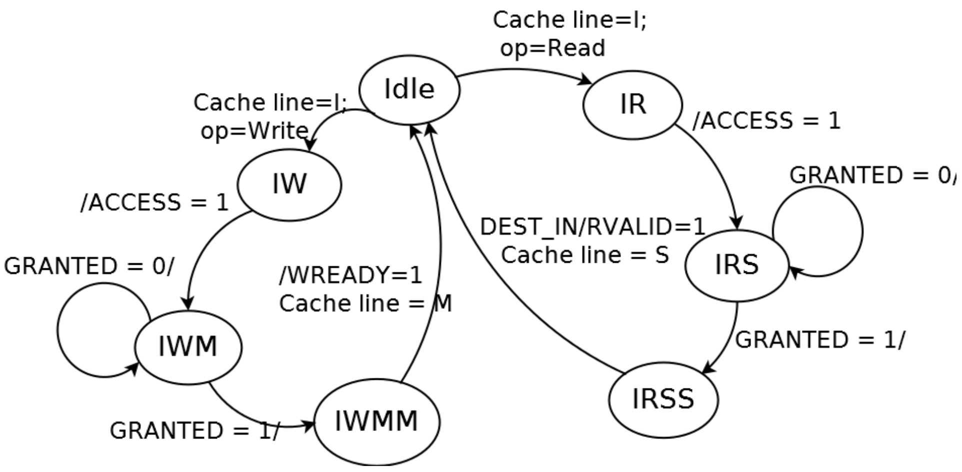


Figure 3.6: Sub-FSM for state I

to the bus has been granted the machine moves to IWMM. IWMM will set the WREADY signal high, inform the core that the controller is now idle, and store the cache line with state I.

The state machine can at any time enter the snoop-state machine(fig 3.7).

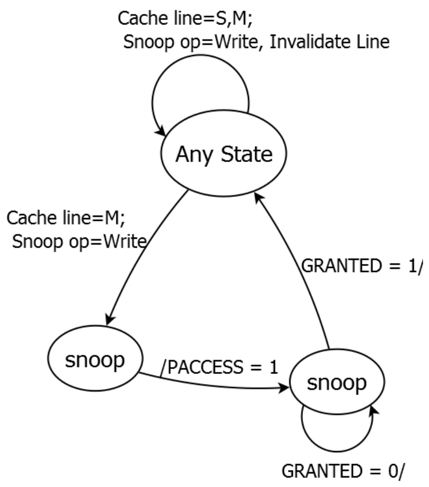


Figure 3.7: FSM for Bus Snooping

If the cache line is in state M or S and the snooped operation is a store, the snoop controller will invalidate the cache line and return to the previous state. If the operation is a load and the cache line is in state M, the controller has to provide the data to the requesting node. It will therefore ask for prioritized access(PACCESS) to the bus interface. As soon as the data has been shipped, the controller will return to the pre-snoop state.

FSM for L2-cache controller and MESI FSM for L1-cache controller can be found in appendix A.

### 3.2.4 Bus Interface

The bus interface in the model is a single shared bus, which accepts *one* atomic request each cycle. Modern CPUs and GPUs use more performance friendly interconnects, which are able to serve one request from each core every clock cycle. These interconnect systems are of course much more complex than a simple bus. But for this project a single shared bus is sufficient. This is because the goal is only to count the number of transactions on the interface and the tag lookups in the cores. A single bus will use four cycles to complete four transactions. A bus which is four times as big will use only one cycle, but the end result for the transaction count is four in either case. Because of this and the reduced complexity a single shared bus was implemented in the model.

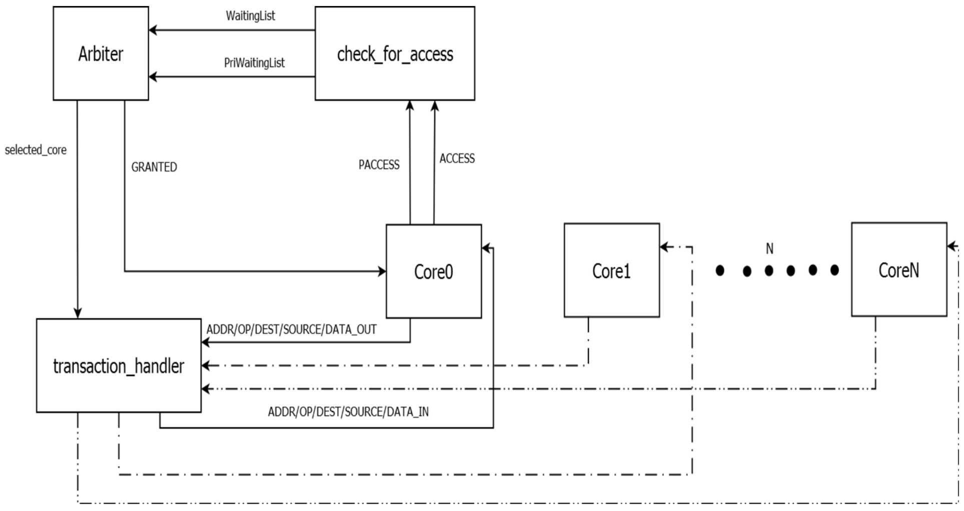


Figure 3.8: Bus Transaction System

Figure 3.8 shows how transactions are handled by the bus interface. Each core is connected to the *check_for_access* module. The *check_for_access* module checks every cycle to see which cores wants to access the bus. The module will then update a waiting list for the bus. The arbiter will then select one of the cores to access the bus. The transaction handler will get information about the selected core from the arbiter and grant the core access. If the destination of the transaction is the L2 cache, it will not grant access until the L2 cache controller is no longer busy. The transaction handler will also check if the core wants to inform the other cores about the transaction. This is done by checking the SACCESS flag. If the system uses a source filter this flag may be low, enabling the system to save power by not informing the other cores.

Every transaction contains these items:

- **OP**: This is a operation specifier:
  - *Read*: If the core wants to load a cache line. If a remote cache has the cache line, it will snoop and respond instead of the L2 cache controller.
  - *Write*: Load operation, other cores will get an invalidate request.
  - *Evict*: The L1 cache controller will evict a cache line if it has to be replaced and the data is in state M.
  - *Snoop Update*: Whenever a cache needs to respond to a load request, it will start a snoop update transaction.
- **ADDR**: The address where the operation is being performed.
- **SOURCE**: Who is the source of the transaction.
- **DESTINATION**: Which node is the destination of the transaction.
- **DATA**: Data needed at the destination.

### 3.2.5 L2 Cache Controller

The controller for the L2 cache will check the bus interface to see whether it is the destination for the transaction. If it is it will capture the address and the operation specifier and do a cache lookup. The cache lookup will give owner information about the cache line. If the cache line is owned by the L2 cache it will in the case of a store operation give ownership to the requesting core, or in the case of a write operation give data back to the requesting core. If the cache line is not owned by the L2 cache it will simply do nothing, just go to an idle state. The L2 cache is given prioritized access to the bus interface in order to speed the simulation time.

## 3.3 Benchmarks

Six different OpenCl benchmarks were tested on the mode, tablel 3.1 has a short description of them. The benchmarks use 4 cores if nothing else is stated.

Table 3.1: Different Benchmarks Run on the Model

| **Benchmark:** | **Description:** |
|---|---|
| **Sobel**: | The Sobel operator is often used in image processing for edge detection. |
| **Atomic-Sum**: | This is a built in function in OpenCl. It will compute the sum of a sequence using atomic operations. |
| **Gauss Filter**: | The Gaussian filter is a filter with an Gaussian function as impulse response. It is often used to reduce noise in images by blurring/smoothing the image. |
| **Atomic-Logical-Xor**: | An atomic **xor** operation. |
| **Histogram**: | OpenCl function that calculates the total number of occurrences for gray-scales in an image. This benchmark uses only 2 cores. |
| **Atomic-Logical-And**: | Atomic logical **and** operation. This benchmarks uses 8 cores. |

GPGPU on mobile devices is not yet very common and the number of available benchmarks and applications are few. That being said, the benchmarks used is a good representation for possible applications that makes sense to run on GPUs

## 3.4 Summary

A dynamic multi core model was designed in order to evaluate different snoop filter structures. The model is able to both randomly create stimul,i and use data from Mali-T6xx executed benchmarks. The next chapter is going to present work related to snoop filters as well comparing the performance of CSR and new proposed filter designs.

# Chapter 4

# Snoop Filters and Performance

With the model presented in the previous chapter as basis, a OpenCl evaluation of snoop filters could be performed. It was also interesting to find out what effect the MESI protocol had on OpenCl benchmarks compared to the basic MSI protocol. This chapter presents recent snoop filters, snoop filter results from the model, a comparison between MESI and MSI and proposed new filters and their design.

## 4.1 Related Work

Based on the literature several filters are interesting. *JETTY* (Moshovos et al. 2001) was the first acknowledged snoop filter. It was design to reduce the L2 power consumption in large SMPs. For SoCs the JETTY is less interesting because of its major hardware overhead. More interesting solutions are the *Stream Registers*(Salapura et al. 2007), and more recently the enhanced version *Counting Stream Registers*(Ranganathan & Charbon 2012). All of the filters are inclusive destination filters, with the exception of the JETTY. The JETTY has an exclusive version, as well as a hybrid exclusive/inclusive version. From the survey in (Ulfsnes 2012) the most interesting solution was the *Counting Stream Registers*. For more information about JETTY and other powers saving solutions, go to (Ulfsnes 2012).

### 4.1.1 Stream Register Based Snoop Filters

This section is recited from (Ulfsnes 2012).

Stream register(SR) based snoop filters is a snoop filtering technique that is actually being used in the Blue Gene/P supercomputer. It was introduced by IBM in (Salapura et al. 2007), the filter is inclusive and uses stream registers to keep track of what is in the cache. The BlueGene uses single port caches in its L1. The system also uses a write-through policy, which leads to a lot of broadcasting on the interconnect. Because the L1 caches are single-port, IBM use SRs in order to avoid performance degradation in L1 accesses. The SR is a destination filter, meaning that it will reduce snoop induced tag lookups. The basic architecture for the SR filter can be seen in figure 4.1.

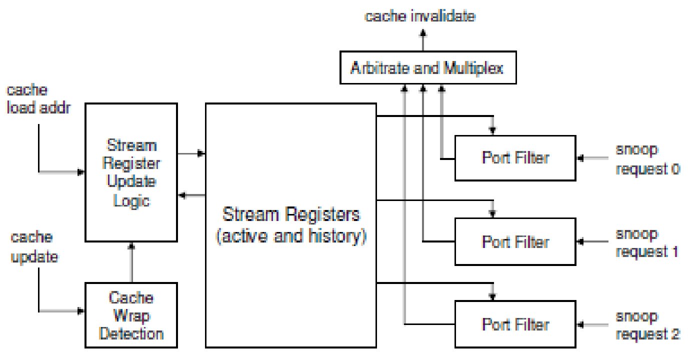


Figure 4.1: Basic architecture for *Stream Register* (Salapura et al. 2007)

The register is updated every time the cache loads a new line. The SR update logic selects the appropriate register based on the state of the stream register and the address that is being loaded. Remote snoop accesses can be run in parallel, and in this architecture there are 4 nodes in the system. The filter therefore has 3 port filters which can handle 3 remote requests simultaneously. The incoming requests are compared to the current state of the registers, and notifies the cache invalidate logic if the address is in the register.

The SR consists of two registers and a valid bit. The base register has information about the address bits that are shared among all the cache lines, and the mask knows which of these bits are shared. The behavior of the SR can be shown with a simple example from (Salapura et al. 2007):

Consider two addresses being loaded into the cache, 0x1708FB1 and 0x1708FB2. After these addresses has been loaded the stream register will be updated as such:

The mask register is initially initialized to all 1's, while the base register loads the first address:

```
base <= 0x1708FB1
mask <= 0x7FFFFFF
```

When the second address is loaded the registers are updated as such:

```
base <= 0x1708FB2
mask <= 0x7FFFFFc
```

Since the two addresses have two different LSBs, the last two bits are cleared in the mask register. The SR will now indicate that addresses 0x1708FB0, 0x1708FB1, 0x1708FB2 and 0x1708FB3 can be in the cache. When a new address is loaded into an empty SR the valid bit is set to 1, indicating that the register is in use. The selection of which SR to choose when a new address is being loaded can be selected by using one of two polices:

- Choose the SR with the minimum Hamming distance. I.e. choosing the SR that will change the minimum number of bits in the mask register.

- Choose the SR where the the highest number of MSB's match the address being loaded.

It is also necessary to set some fictitious distance for empty SRs; if the hamming distance for a used SR is higher than the ”distance” for the empty SR, the empty will be used. This implies that two small registers can filter a lot of addresses. Though after a while, depending on how many SRs you have, they will eventually be all 0's. This makes the SR in turn unable to function any more.

Another issue with SR is that it is not possible to remove specific addresses from the filter, for instance when a line is evicted. The SR will be reset when the cache has been totally replaced since some initial state. This is also known as a cache wrap. This forces the need of a history SR, where the contents of the active SR are copied to a second register bank called history SR. The history SR is treated as a second SR, and is updated the next time a cache wrap occurs.

The lack of functionality for removing addresses from the SR is the biggest drawback with the filtering technique. The filter can be made very small depending of how big your cache is, leading to minimal hardware and power overhead. Being able to detect cache wraps is a non trivial challenge, which requires a lot of extra hardware and storage. The SR can also be reset after a given time interval, but the entire cache needs to be flushed to ensure that the SR does not filter blocks that are present in the cache. This is very power intensive and will take away all the power savings from filtering snoops.

### 4.1.2 Counting Stream Register Snoop Filter

Because of the cache wrap problem for the *Streaming Registers*, (Ranganathan & Charbon 2012) proposes a new type of SR called the *counting stream register*(CSR) filter. The motivation behind this filter is to remove the need for cache wrap logic from the SR filter, making it usable for more power critical applications. This because it does not need any cache wrap logic, and there is no need for a cache flush.

The functionality of the snoop hit mechanism in the CSR can be seen in figure 4.2. The address is split into page tag index(idx) and offset. The index is used to access the SR table. The MSBs of the address is used as a tag for the base register. Initially when a new address is loaded into the CSR, the base register is set to the new address and the mask is set to all 1's. Simultaneously the counter is incremented indicating that the CSR is used to filter one address, eliminating the CSR valid bit. Each time a new address is loaded into the CSR the counter is incremented. Any replacement, eviction or invalidation will decrement the counter. It will not however alter the base and mask registers. When the counter reaches 0 the CSR will be reset, thus eliminating the need for cache wrap logic or cache flushing.

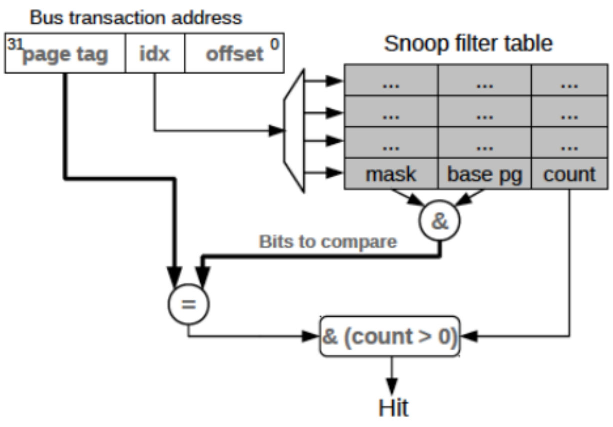


Figure 4.2: Basic architecture for *Counting Stream Register* (Ranganathan & Charbon 2012)

## 4.2 Simulation and Results

The simulation were run using the model described in chapter 3. Every result depicted in this report is the average of five simulation runs. Total number of unnecessary tag lookups, total number of transactions on the snoop interface, and total number of necessary tag lookups are given below:

- Total number of unnecessary tag lookups = $N_{UTag}$
- Total number of transactions on snoop interface = $N_{ST}$
- Total number of necessary tag lookups = $N_{NTag}$

Table 4.1 shows the necessary tag lookups for each benchmark.

Table 4.1: Necessary Tag Lookups

| **Benchmarks** | $N_{NTag}$ |
|---|---|
| ***Sobel*** | 3067.2 |
| ***Atomic Sum*** | 17.4 |
| ***Gauss*** | 16.8 |
| ***Atomic Xor*** | 136 |
| ***Histogram*** | 1 |
| ***Atomic And*** | 2384.4 |

The improvement are given as percentages by using this formulas:

- Improvement tag lookups = $100 - \frac{N_{UTag}(filter)}{N_{UTag}(nofilter)}100$

- Improvement transactions $= 100 - \frac{N_{ST}(filter)}{N_{ST}(nofilter)}100$

While the tag lookup improvement can actually reach 100%, the improvement of the transactions can only go as low as the number of necessary tag lookups divided by the number of cores minus one: $(\frac{N_{NTag}}{N-1})$.

### 4.2.1 Comparing MESI and MSI

All the benchmarks were simulated without filter structures with MESI and MSI. The reason was to investigate whether or not there is any effect by adding the exclusive state to the basic protocol for OpenCl programs on GPUs. Table 4.2 contains the total number of unnecessary snoop induced tag lookups,the total number of transactions on the interface, and any improvements on the benchmarks for both MSI and MESI.

Table 4.2: MSI vs MESI

| | **MSI** | | **MESI** | | **Improvement** | |
|---|---|---|---|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}(\%)$ | $N_{ST}(\%)$ |
| ***Sobel*** | 11545.8 | 4871 | 11555.2 | 4875 | -0.081 | -0.082 |
| ***Atomic Sum*** | 10603.2 | 3540.2 | 10590.6 | 3537 | 0.118 | 0.090 |
| ***Gauss*** | 4615.8 | 1544.2 | 4615.8 | 1544.2 | 0 | 0 |
| ***Atomic Xor*** | 2771.6 | 920 | 2763.8 | 920 | 0.281 | 0 |
| **Average** | | | | | 0.063 | 0.0016 |

The last two benchmarks, **histogram** and **atomic and** (2 and 8 core systems), were not simulated with MESI because of a bug in the L2-cache controller when simulating with a different setup than 4 cores.

The results in table 4.2 shows that extending MSI with the exclusive state has close to no effect on the coherence induced power consumption. A reason for this migth be that when the cores execute the OpenCl benchmarks, cache lines gets loaded into multiple L1-caches. In other words the E state rarely gets used, and therefore becomes redundant. Because of this, and a possible bug in the MESI supported L2- controller, the snoop filters are only going to be compared to the MSI protocol.

### 4.2.2 Destination CSR

The simulation show results for 16, 32, 64 and 128 stream registers per core.The results can be seen in table 4.3

There was also done a random chunk test for the 32-register configuration of the CSR filter. The goal was to figure out the sensitivity of the filter towards the size of the address stream, and the probability of shared core addresses.

Table 4.3: CSR Results- Different Sizes

| | **16-CSR** | | **32-CSR** | | **64-CSR** | | **128-CSR** | |
|---|---|---|---|---|---|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ |
| ***Sobel*** | 8993.8 | 4869 | 5822.4 | 4869.2 | 5124.4 | 4870.4 | 3758 | 4875.2 |
| ***Atomic Sum*** | 8803.6 | 3537.4 | 7642.2 | 3535.8 | 4732 | 3536 | 2806.4 | 3535.6 |
| ***Gauss*** | 3557.8 | 1543.6 | 2473.2 | 1543.8 | 2473 | 1544.4 | 908 | 1544.4 |
| ***Atomic Xor*** | 1861.2 | 920 | 1258 | 920 | 691.2 | 920 | 163.8 | 920 |
| ***Histogram*** | 48.6 | 197 | 2 | 197 | 2 | 197 | 0 | 197 |
| ***Atomic And*** | 15792.4 | 3110.2 | 12650.8 | 3161 | 6804 | 3189 | 2801.8 | 3146.6 |

Table 4.4: Improvement vs. No-Filter: Destination-CSR

| | **16-CSR**(%) | | **32-CSR(%)** | | **64-CSR(%)** | | **128-CSR(%)** | |
|---|---|---|---|---|---|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ |
| ***Sobel*** | 22.103 | 0.041 | 49.57 | 0.037 | 55.616 | 0.012 | 67.451 | -0.086 |
| ***Atomic Sum*** | 16.98 | 0.079 | 27.93 | 0.124 | 55.371 | 0.112 | 73.533 | 0.13 |
| ***Gauss*** | 22.921 | 0.038 | 46.42 | 0.026 | 46.23 | -0.013 | 80.328 | -0.013 |
| ***Atomic Xor*** | 32.85 | 0 | 54.61 | 0 | 75.061 | 0 | 94.09 | 0 |
| ***Histogram*** | 75.479 | 0 | 98.99 | 0 | 98.99 | 0 | 100 | 0 |
| ***Atomic And*** | 27.66 | 2.5 | 42.05 | 0.909 | 68.83 | 0.031 | 87.166 | 1.36 |
| ***Average*** | 32.99 | 0.44 | 53.262 | 0.183 | 66.72 | 0.025 | 83.761 | 0.23 |

Figure 4.3 shows that the CSR are much more sensitive to the size of the streams than increasing the probability of cores sharing addresses. That means that the CSR are prone to variations in how the compiler allocates memory at compile time. This means that there might be a potential in how the CSR selects a specific register to store the new address.

The model also has the capability of creating heat-maps for benchmarks, for both unnecessary and necessary snoops. The point is to look at which address range is shared most often among the cores.

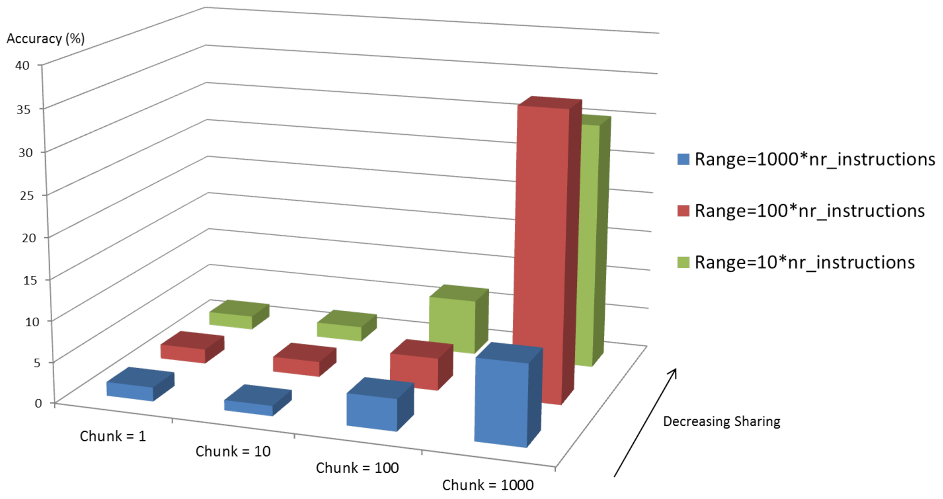


Figure 4.3: Chunk test for CSR destination filter

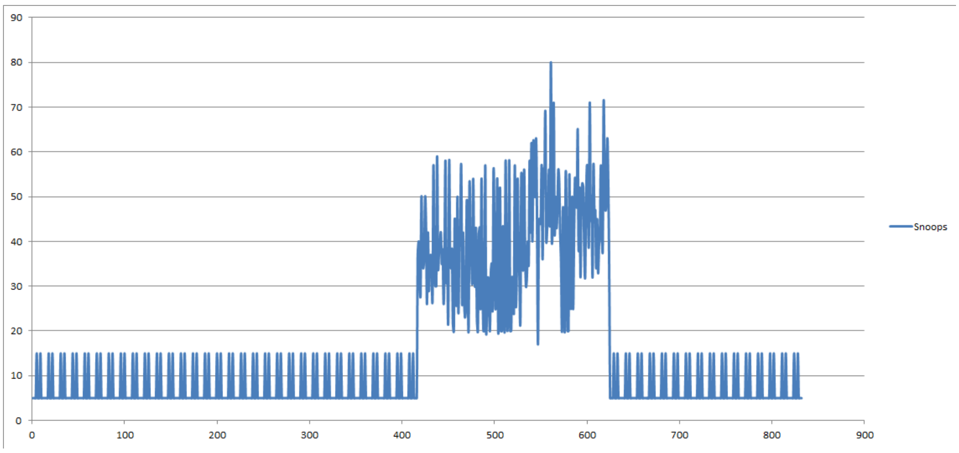


Figure 4.4: Heat map for necessary snoops, **Atomic And**($x = N_{UTag}$, y= address)

Figure 4.4 shows which addresses are snooped by all the nodes necessary. The address range is offset to 0, and the snoop count is summed for all the cores.

Figure 4.5 shows the unnecessary snoops for the same benchmark. The address ranges that has the most unnecessary and necessary snoops are within the same range. The target of any inclusive filter is to only include the addresses that are in the cache, or in other words addresses that are necessary to snoop. In the case of

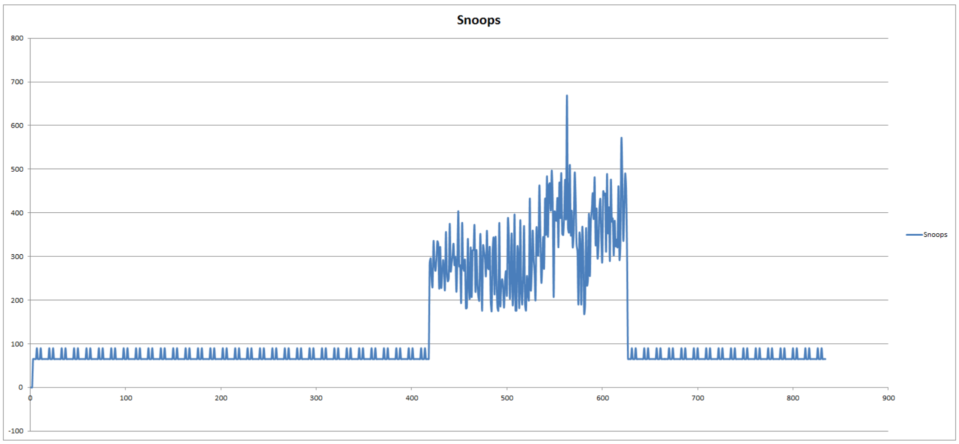


Figure 4.5: Heat map for unnecessary snoops, no filter, **Atomic And**($x = N_{UTag}$, y= address)

a benchmark like the one in 4.4, the filter would be better of by using more of the resources (registers) in the range with the highest count of necessary snoops, and less resources for the rest of the address-space.

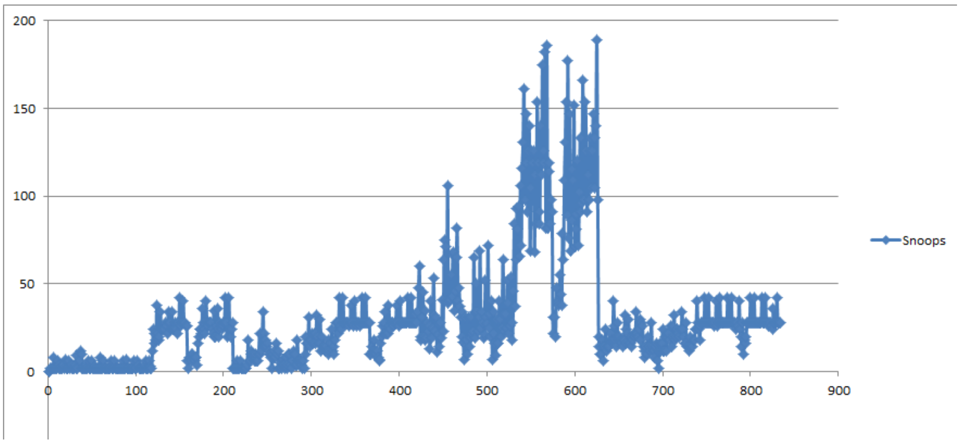


Figure 4.6: Heat map for unnecessary snoops, with CSR-32 destination filter, **Atomic And**($x = N_{UTag}$, y= address)

Figure 4.6 shows how well a CSR-32 reduces the unnecessary snoops for the **atomic and** benchmark. It copes relatively well before and after the 500-600 range. If more resources would have been used on that range, or if the resource had been selected more carefully so that the CSR are filled with continues streams, maybe the filter would be more accurate. The problem is that one cannot know how the heat-maps will be for a specific benchmark without running it first. Some kind of run-time adaptive filter structure might be a good solution, although constructing a dynamic filter will add even more hardware to the system. Besides this is not a trivial task. A system where the chosen CSR is not decided by the lowest bits of the address but some other function. This might also improve the accuracy of the filter, without adding the same amount of hardware as a dynamic filter.

## 4.3 New CSR Filters

Counting stream registers with 128 SR reaches 84% accuracy on the OpenCl benchmarks, the CSR with 32 registers got 53% 4.4. They also do no reduce the number of transactions on the snoop interface. Because of this, two new filters based on the CSR was invented: *Source-CSR* and *Hashed-index CSR*. The goal of *Source-CSR* is to reduce snoop induced transactions as well as snoop induced tag lookups. The *Hashed-index CSR* will try to improve the accuracy of the CSR by hashing the address instead of just using the lower bits of the address to find the index.

### 4.3.1 Source CSR

The results in table 4.3 shows potential for reducing the $N_{ST}$ for OpenCl benchmarks. The source filter has more hardware overhead than the destination filter, but it also has more power reducing potential. The structure in figure 4.2 is enhanced by adding a similar structure for each remote node in the multi-core system. For a system with N-cores the filter will be (N-1) times as big as a destination filter.

The basic concept of the source type filter is that it will observe the transactions performed by the other cores on the interconnect. Based on the type of transaction the filter will decide if an address should be added or removed from the registers. This type of filter also needs some kind of notification line for invalidations caused by store transactions. If a core wants to store some value to an address, other cores might snoop and invalidate the same address. The filters in all of the cores needs to be made aware of the invalidation. A solution to this is to construct 1-bit invalidate lines going to and from all cores. At any time when a core needs to invalidate a cache-line because of a store, it will set its invalidate line high; the source-filters will observe the store, check the invalidate line for each core, and remove the address from the corresponding register.

Whenever a controller wants to do a load/store from the L2-cache it will first inquire the source filter about the address in hand. The source filter will then respond with a hit or miss. This type of filter does not have any information about which cores has the specific address, only that one or more cores has cached the address. If just one core has cached the line in a 4-core system, two of the cores will snoop the address unnecessarily. There are two methods of reducing these unnecessary tag lookups:

1. *Hybrid-Source-Destination*
2. *No-Broadcast-Source*

The hybrid solution employs a destination CSR filter that works in parallel with the source filter. It will however introduce even more hardware overhead, making the total overhead N times the destination filter. The second solution requires the same hardware as a normal source filter, but with the added functionality it will inform the controller about which remote cores that might have cached the address.

### 4.3.2 Source CSR Results

The source CSR was simulated for the same size configurations as the destination filter, table 4.5. The hybrid and the broadcasting-source filter was only simulated for a 32-CSR configuration. Only the results are presented in this section, while the comparison with the destination CSR comes in section 4.4.

Table 4.5: Source-CSR Results- Different Sizes

| | **16-CSR** | | **32-CSR** | | **64-CSR** | | **128-CSR** | |
|---|---|---|---|---|---|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ |
| ***Sobel*** | 8993.6 | 4654.2 | 5817.8 | 4418.6 | 5126 | 4219.2 | 3754.6 | 3874.2 |
| ***Atomic Sum*** | 8768.4 | 3171.4 | 7668.6 | 3063.8 | 4748.4 | 2574.2 | 2805 | 2002.8 |
| ***Gauss*** | 3551.2 | 1350.4 | 2466.8 | 1228.8 | 2467.4 | 1228.2 | 901.6 | 735.2 |
| ***Atomic Xor*** | 1843.8 | 749.2 | 1247.2 | 625.6 | 687.8 | 625.6 | 164.2 | 260 |
| ***Histogram*** | 47.8 | 48.8 | 2 | 3 | 2 | 3 | 0 | 1 |
| ***Atomic And*** | 16196.4 | 3066.4 | 12643.4 | 2913.8 | 6800.8 | 2765.5 | 2938.6 | 2630.4 |

Table 4.6: Improvement vs. No-Filter: Source-CSR

| | **16-CSR**(%) | | **32-CSR**(%) | | **64-CSR**(%) | | **128-CSR**(%) | |
|---|---|---|---|---|---|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ |
| ***Sobel*** | 22.105 | 4.45 | 49.61 | 9.287 | 55.60 | 13.381 | 67.48 | 20.463 |
| ***Atomic Sum*** | 17.135 | 10.417 | 27.676 | 13.457 | 55.22 | 27.287 | 73.545 | 43.427 |
| ***Gauss*** | 23.06 | 12.55 | 46.56 | 20.437 | 46.54 | 20.464 | 80.47 | 42.389 |
| ***Atomic Xor*** | 33.475 | 18.565 | 55.00 | 32 | 75.18 | 32 | 94.076 | 71.739 |
| ***Histogram*** | 75.88 | 75.228 | 98.99 | 98.477 | 98.99 | 98.477 | 100 | 99.492 |
| ***Atomic And*** | 25.81 | 3.87 | 42.088 | 8.658 | 68.85 | 13.31 | 86.54 | 17.542 |
| ***Average*** | 32.91 | 20.85 | 53.32 | 30.386 | 66.73 | 34.153 | 83.69 | 50.842 |

Table 4.7: Source-CSR Results- Different Setups

| | **CSR-Broadcasting** | | **CSR-Hybrid** | |
|---|---|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ |
| ***Sobel*** | 10185.4 | 4417.6 | 5824.8 | 4421.2 |
| ***Atomic Sum*** | 9120 | 3046.2 | 7661.2 | 3056.6 |
| ***Gauss*** | 3674.4 | 1228 | 2466.8 | 1227.2 |
| ***Atomic Xor*** | 1821.2 | 624.8 | 1258.6 | 625.4 |
| ***Histogram*** | 2.4 | 3 | 2 | 3 |
| ***Atomic And*** | 19671.4 | 2930 | 12748.8 | 2929.6 |

Table 4.8: Improvement vs. No-Filter: Source-CSR, different setups

| | **CSR-Broadcasting**(%) | | **CSR-Hybrid**(%) | |
|---|---|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ | $N_{UTag}$ | $N_{ST}$ |
| ***Sobel*** | 11.78 | 9.308 | 49.55 | 9.234 |
| ***Atomic Sum*** | 13.988 | 13.95 | 27.746 | 13.66 |
| ***Gauss*** | 20.395 | 20.476 | 46.55 | 20.53 |
| ***Atomic Xor*** | 34.29 | 32.09 | 54.59 | 32.021 |
| ***Histogram*** | 98.79 | 98.477 | 98.99 | 98.477 |
| ***Atomic And*** | 9.89 | 8.16 | 41.61 | 8.16 |
| ***Average*** | 31.52 | 30.347 | 53.17 | 30.35 |

### 4.3.3 Hashed-index CSR

Inspired by the heat-maps in section 4.2.2, a different method was tested other than just selecting the 5 lowest bits(32-CSR) for the index. Hashing is a method where variable length data sets are mapped to fixed length data sets. The index will be created by hashing the entire address, then using modulo to create the new index:

$index = hash(address)\%(N_{CSR} - 1)$

Hashing something in hardware will of course cost more power and impose more latency than a filter system that just uses the lower bits to select register. But if the accuracy is sufficiently increased it may be worth the extra cost.

### 4.3.4 Hashed-index CSR Results

The hashed-index CSR was only simulated for 32-CSR configuration, simply because it was implemented quite late in the project and there was not enough time. The model used the MD5 hash-function in python to hash the entire address. Other cryptographic hash functions were tried without any noticeable effect on the accuracy.

Table 4.9: Hashed-index-CSR Results- Different Setups

| | **Hashed-index CSR** | |
|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ |
| ***Sobel*** | 9706.2 | 4872.4 |
| ***Atomic Sum*** | 8281 | 3545.6 |
| ***Gauss*** | 2173.8 | 1542.6 |
| ***Atomic Xor*** | 629.8 | 920 |
| ***Histogram*** | 1 | 197 |
| ***Atomic And*** | 6703 | 3161 |

Table 4.10: Improvement vs. No-Filter: Hashed-index-CSR, different setups

| | **Hash-index CSR**(%) | |
|---|---|---|
| **Benchmark** | $N_{UTag}$ | $N_{ST}$ |
| ***Sobel*** | 15.933 | -0.029 |
| ***Atomic Sum*** | 21.90 | -0.152 |
| ***Gauss*** | 52.905 | 0.104 |
| ***Atomic Xor*** | 77.2766 | 0 |
| ***Histogram*** | 99.49 | 0 |
| ***Atomic And*** | 69.297 | 0.909 |
| ***Average*** | 56.13 | 0.1385 |

## 4.4 Filter Comparisons

This section will thoroughly discuss and compare the old destination filter with the two new filters, both for tag lookups and snoop induced transactions.

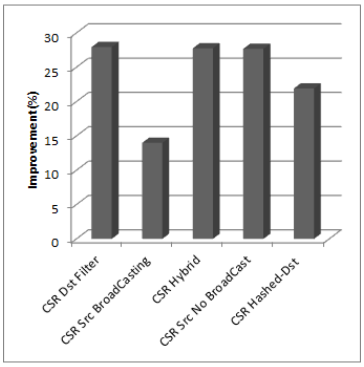
### 4.4.1 Tag Lookups

Tag lookup comparison between the five different filter structures can be seen in figure 4.7. The original source filter, which broadcasts invalidation/updates to every core, performs considerably worse than any other filter for all benchmarks. It has an improvement that is half or less in percentage compared to the other structures. Both hybrid and the non-broadcasting source filter performs equally to the original destination filter for all benchmarks. The hybrid filter does not outperform the non-broadcasting filter however, making the non-broadcast filter a better option as it has less hardware overhead.

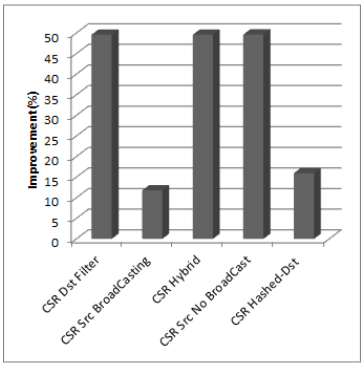
The hashed-index filter shows a little more performance variation. It is almost as bad as the broadcasting source filter for the ***Sobel*** benchmark. For the ***Atomic Sum*** benchmark the performance is a bit better than the broadcasting source. The results are better for all other benchmarks, especially the ***atomic xor/and*** benchmarks, getting 70-80% improvement over a non-filter solution. Not knowing exactly which benchmarks are most representative, it is hard to know if the poor performance in the ***Sobel*** benchmark should be taken more into account than the excellent performance in the atomic benchmarks. The image processing benchmarks are probably more useful than the atomic operations.

As for the original CSR destination filter, heat maps were captured for the hashed index CSR. Figure 4.8 shows the heat map for unnecessary snoops for the hashed filter. Figure 4.6 shows the unnecessary snoops for the regular filter. Comparing the two maps it is easy to see that hashing the index has an incredible effect for this benchmark. The unnecessary spikes are lower in figure 4.8, maxing out at about 100. Whereas figure 4.6 has its maximum at about 150++. The width of the region with the highest number of unnecessary snoops, are about the same for both structures. These experiments show that altering how the resources are allocated across the address space has a huge effect on the performance.

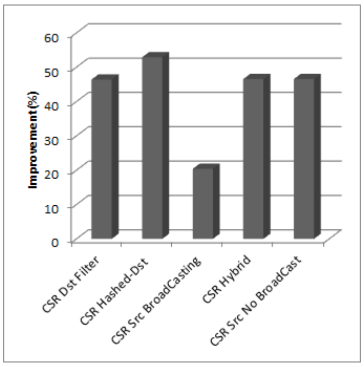
The non-broadcasting source and the destination CSR were also simulated using different size configurations. Figure 4.9 shows that the tag lookup efficiency of the filters are comparable for all configurations. A power analysis, as in chapter 6, has to be performed before any conclusion about the size of the filter can be made. However at this time, the CSR-32 seems like a good compromise.



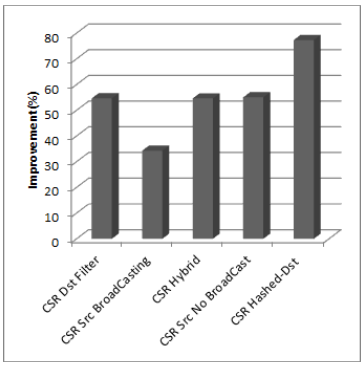
(a) Atomic Global Sum



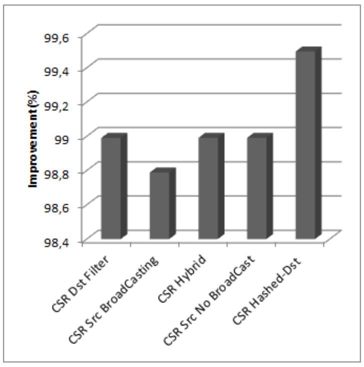
(b) Sobel



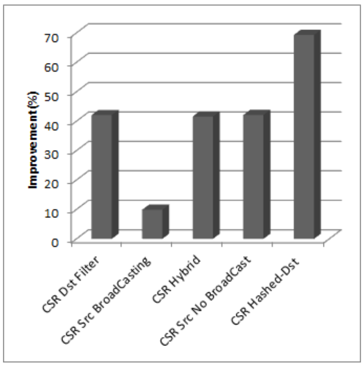
(c) Gauss



(d) Atomic Xor



(e) Histogram



(f) Atomic An

Figure 4.7: Comparison of filters: Tag lookup

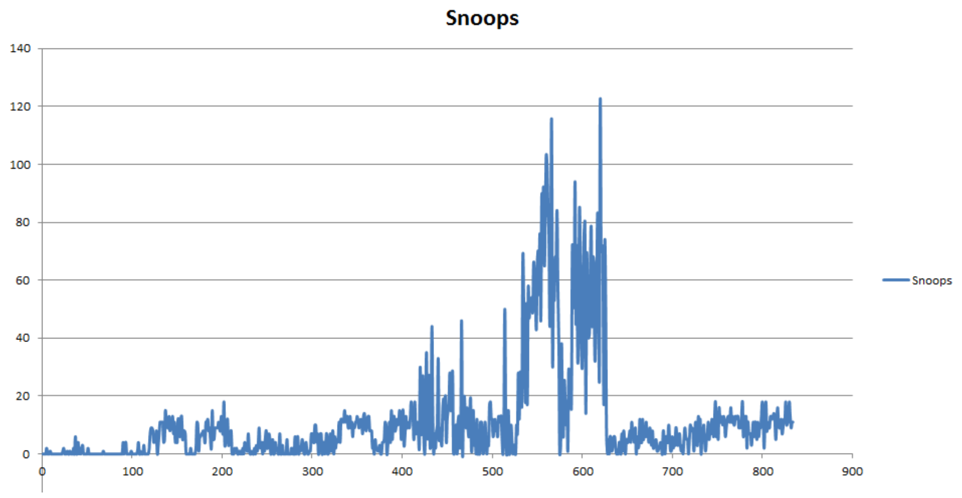


Figure 4.8: Heat map for unnecessary snoops, CSR with hashed index, **Atomic And**($x = N_{UTag}$, y= address)

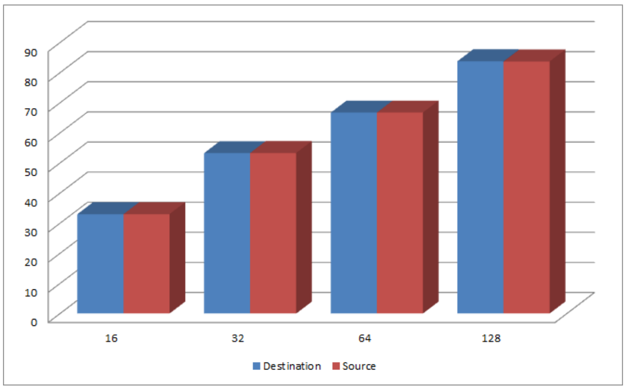


Figure 4.9: Different Filter Sizes: Tag Lookups, Average improvement

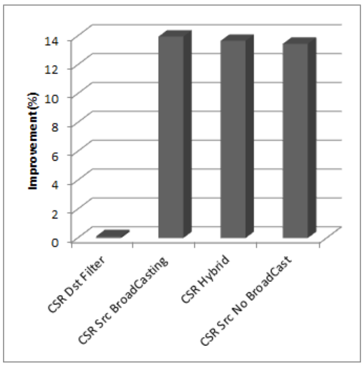
### 4.4.2 Snoop Transactions

Figure 4.10 shows snoop transaction comparisons between four different filter structures. The hashed-index filter is not in this figure because it is a destination filter. As the results in section 4.3.4 shows, it has no effect on the snoop transactions at all. As figure 4.10 shows, all of the source filters performs just about equally. There are no discernible difference except for the ***Atomic And*** benchmark, but that small difference might have been caused by randomness in the model[1].

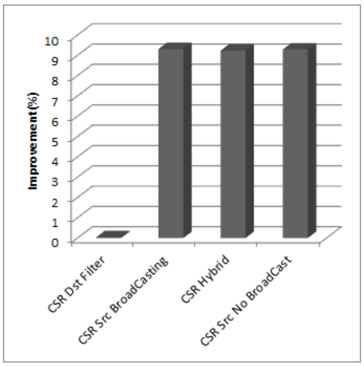
Figure 4.11 shows that for the 32-CSR configuration, about 30% of the snoop transactions are removed. But as mentioned earlier it is hard to say how many snoop transactions are actually needed to maintain coherency. Looking at table 4.5 for the ***Atomic And*** benchmark, the CSR-128 has reduced the number of snoop transactions to 2630. This while the number of necessary snoops required for that benchmark is 2384.4, so the smallest number of transactions possible for that benchmark is 596. This is however a speculative number, and it might very well be somewhat higher.

Again it looks like the best compromising configuration uses CSR-32. The 128 configuration has only 20% less snoop transactions, which may be hard to defend from a power perspective.

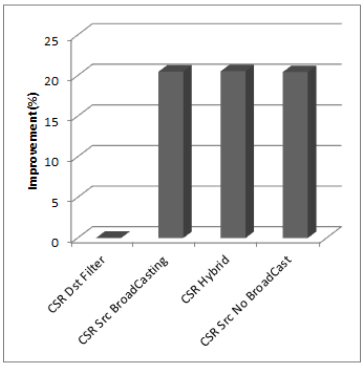
[1]Arbiter selects random nodes for instance.



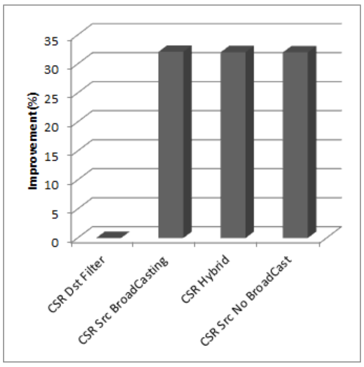
(a) Atomic Global Sum



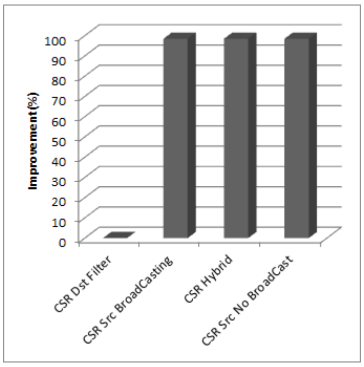
(b) Sobel



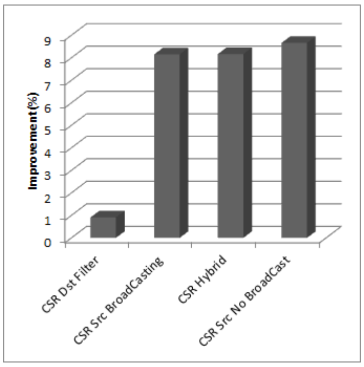
(c) Gauss



(d) Atomic Xor



(e) Histogram



(f) Atomic An

Figure 4.10: Comparison of filters: Snoop Transactions

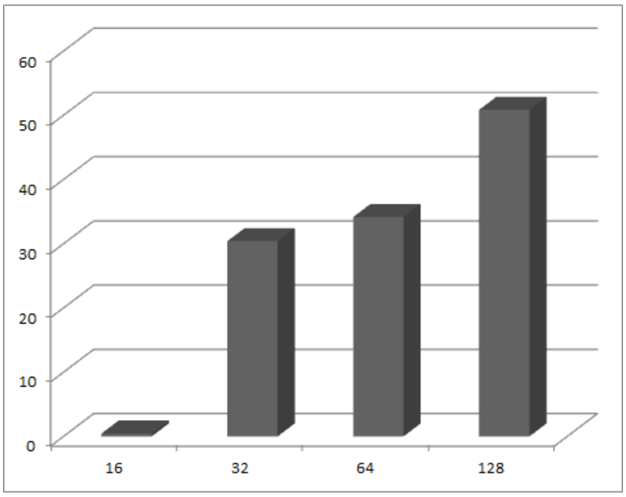


Figure 4.11: Different Source Filter Sizes: Snoop Transaction, avergage improvement

## 4.5 Summary

This chapter has presented the CSR destination filter, along with two newly invented filters: *hashed-index CSR* and *source-CSR*. Simulations on the model presented in chapter 3 using OpenCl benchmarks, has shown that the *hashed-index* filter outperforms the old CSR filter in some benchmarks, while it is worse in others. The take away from this should be that selection of resources has incredible effect on the accuracy of the filter.

The *source-CSR* filters tag lookups just as good as any configuration of the destination CSR. It also has the added bonus of removing on average 30% of snoop induced transactions. Which size configurations to choose, or whether or not the source-CSR is worth implemented, will be presented in the power analysis chapter 6.

Two different cache coherence protocols were also simulated on the model. The results showed no effect of adding an exclusive state in the MESI protocol, compared to the MSI protocol. Even though CPUs have great advantage by using MESI (Ranganathan & Charbon 2012). From a power perspective for GPUs running OpenCl benchmarks the choice of protocol is not that important. The reason for this difference might be that the cores in GPU system loads most of the addresses early, and write after some time, allowing other cores to load the same addresses. This renders the exclusive state useless.

# Chapter 5

# Design and Verification

The analysis in chapter 4, showed that the CSR was fairly effective when it came to limiting the number of snoop induced tag lookups. Hashing the address showed that address selection had a positive effect on some benchmarks. The experiments also showed that source CSRs had a positive effect on snoop transactions needed to maintain coherence. Because of this the non-broadcast-source CSR was chosen to be implemented in RTL. This mainly because it had the best of both worlds; it was able to reduce the snoop induced tag lookups in remote caches just as well as the destination version. It was also able to reduce the use of the snoop induced transactions on the interconnect.

## 5.1 Design and Specification

The source filter compared to the destination requires more additional hardware, as well as some additional communication between the cores, so that the remote caches can update their CSRs. The different filters need to watch the transactions every other core does in order to update the CSR. For a single shared bus this poses no problems as there will only be one transaction on the bus at any given time. For more complicated interconnect architectures, parallel buses, handshaking etc., the problem complexity increases. The net power gain will decrease because the filter and the interconnect needs to be more inter-coupled in order for the CSRs to be updated. The source filter also needs an invalidate line if the system employs a single shared bus. It will consist of N 1-bit lines going to all filter controllers from every cache-controller. If a cache writes to an address, every CSR needs to be made aware of any snoop induced invalidations caused by that write.

By using an update policy to the filters which are not performed every clock cycle, but where addresses are pooled together into packets of addresses, the system can use other interconnect structures. This can possibly be done without increasing the energy consumption to much.

A potential problem with packing address updates is that the memory consistency might be compromised. This because it is possible for a core to use stale data before it is notified about the invalidation. A possible solution to this is to employ an exclusive filter structure.This may again compromise the accuracy of the filter, because an exclusive filter only adds missed addresses.

The source filter features:

- One CSR register file per remote node.
- Size of register file is configurable.
- The system requires invalidate notification on writes.
- Tightly coupled with L1-cache controller for low performance degradation.

The RTL for the source counting stream register filter was implemented using Vcs as compiler and simulator. The filter system consists of 4 modules:

1. **IC_SURV**: Monitors interconnect based on direct monitoring or packets of updated addresses.
2. **Check Addr**: The check-address module gets requests from the L1 cache controller when the controller wants to send a invalidate/update request. Because of the pipeline the module needs to be delayed four cycles compared to the update module.
3. **Update Addr**: The update-module gets orders from the IC_SURV and updates the register file.
4. **CSR Register bank**: The CSR register bank consist of N-1 different register files, where N is the number of cores in the system. Each register file has maps to one remote controller.

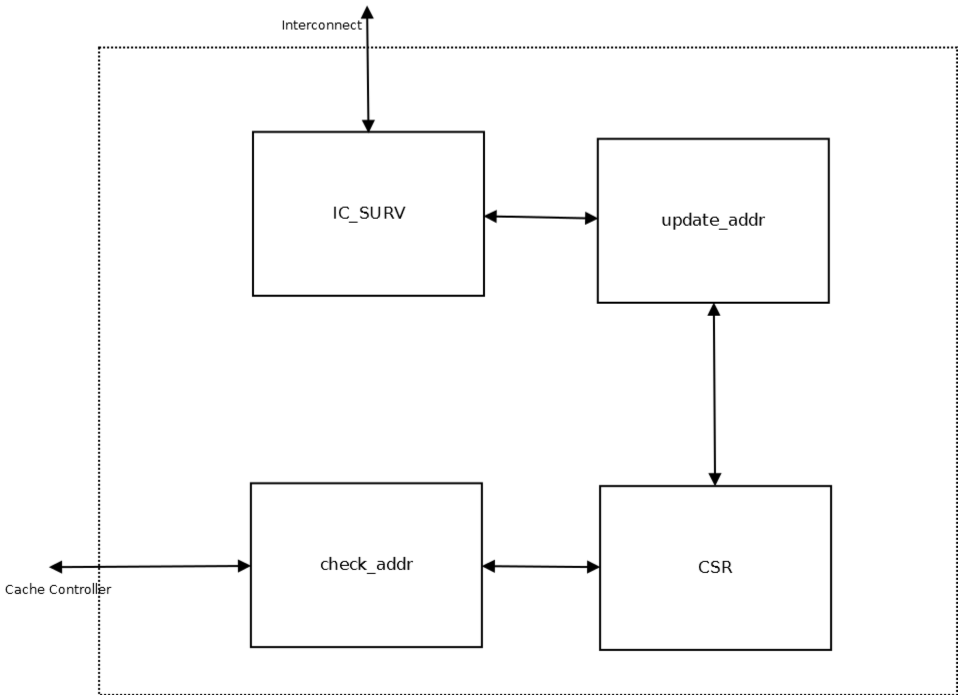


Figure 5.1: Source Filter Toplevel

The toplevel module can be seen in figure 5.1. The interconnect connection is also the invalidate line for writes, in order to simplify the construction of the filter.

### 5.1.1 Interconnect Surveillance Module

The task of the interconnect surveillance is to monitor any traffic from the other cores. That means if a remote cache loads or evicts an address, the surveillance monitor will ask the update logic to remove or add the address from the correct CSR bank. The logic is easily understood by looking at the RTL source code 5.1:

Verilog 5.1: IC_SURV

```
always @(posedge clk or negedge reset_n)begin//IC case
  if(!reset_n)begin //reset
    add_en <= 1'b0;
    remove_en <= 1'b0;
  end else if(valid) begin
    addr_out  <= addr;
    source    <= ic[IC_WIDTH-PAXI_ADDR_BITS-1:IC_OP_WIDTH];
    case (op)
      EVICT: begin
              add_en    <= 1'b0;
              remove_en <= 1'b1;
             end
      READ:  begin
              add_en    <= 1'b1;
              remove_en <= 1'b0;
             end
      WRITE: begin
              add_en    <= 1'b1;
              remove_en <= 1'b0;
             end
      NO_OP: begin
              add_en    <= 1'b0;
              remove_en <= 1'b0;
             end
    endcase
  end else if(!valid) begin
    add_en <= 0;
    remove_en <= 0;
  end
end//IC case
endmodule
```

When a valid transaction is on the bus (and not from the local controller), the surveillance module will decode the ongoing transaction. Either EVICT, READ, WRITE or NO_OP are supported by the module. The surveillance module will have to be altered according to the bus protocol employed on the system.

### 5.1.2 CSR Update Module

The update address logic can be seen in figure 5.2. Basically whenever the IC_SURV wants to update addresses, it will enable the update pipeline. The pipeline has 4-steps, which means that there will be a delay of 4-cycles from the surveillance monitor issues an update, until the change is visible in the CSR register bank. For this reason there will also be two levels of pipeline forwarding. The pipeline control will check if an address that has been received from the IC_SURV is already in the pipeline. It will then enable the forwarding logic, depending on whether or not the address is one or two steps ahead. The forwarding logic will then snatch the most recent data from the output of the update module, instead of asking the register bank for data.

The IC_SURV module will order the update logic to either *remove* or *add* an address from the CSR register bank. The surveillance module will also provide which sources has added or removed an address from its cache. The pipeline is as follows:

- **Stage 0:** In this stage the module will access a CSR register depending on the source which is given by the surveillance module. If the forward logic detects that the index is in the pipeline, the data will not be fetched from the registers but from the output of the module (not shown in the figure).
- **Stage 1:** This stage is a waiting stage, and at the end of the cycle the data will be available from the CSRs. If data_snatch_from_bus_1 is enabled at this point, the data will be fetched from two steps ahead in the pipeline.
- **Stage 2:** This stage calculates the new CSR values based on the operation and the values received from the register. If data_snatch_2 is enabled, the values used for calculation will be fetched from 1 step ahead in the pipeline. If it is a remove operation the counter will be decremented and the old mask and base will be stored unchanged. If it is an add operation however, a new mask will be calculated.
- **Stage 3:**The new mask, base, and count will be stored.

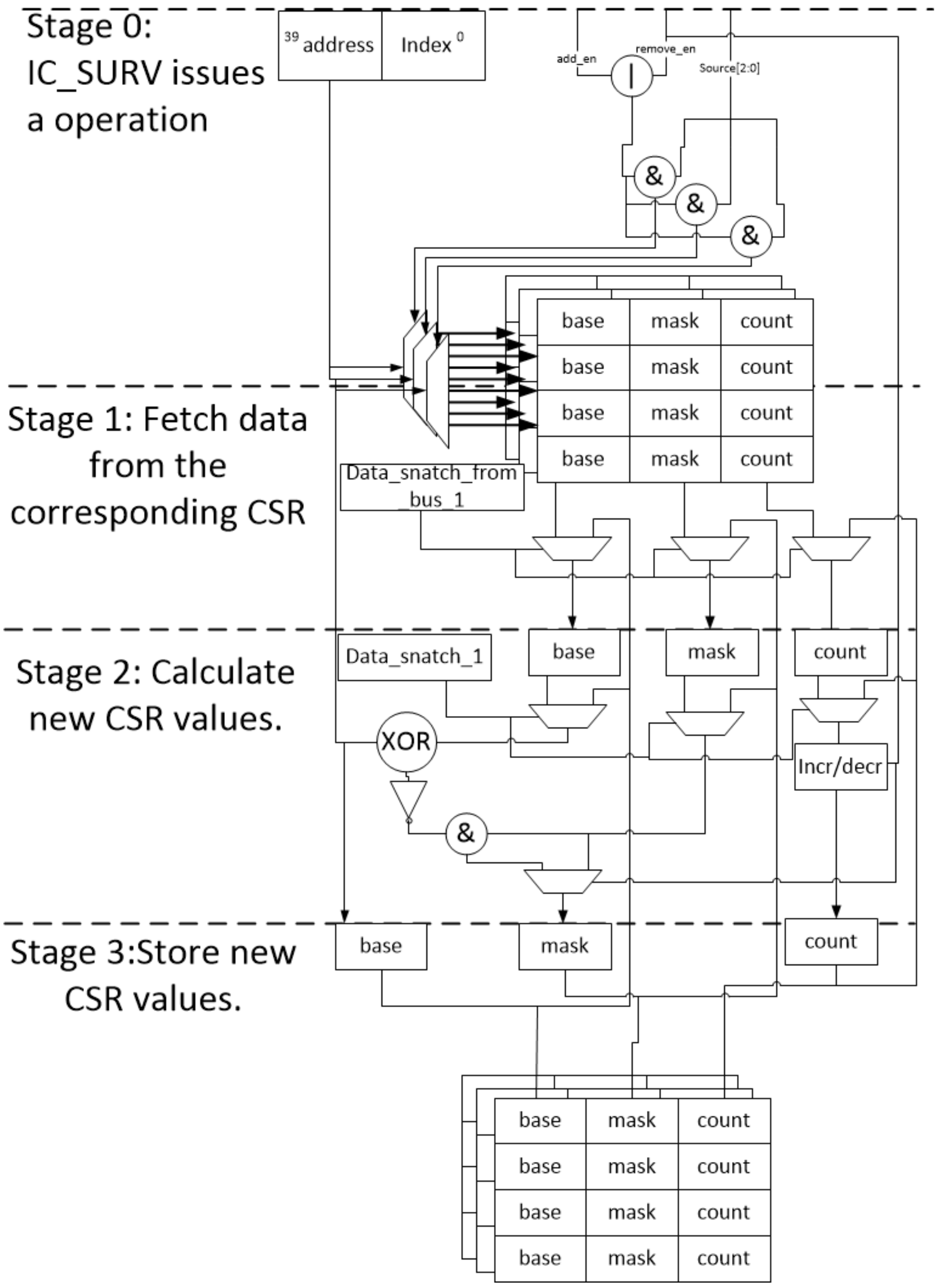


Figure 5.2: Source Filter Update Address Module

### 5.1.3 CSR Check Address Module

The check address gets a request from the snoop controller in the L1-cache. The check address will then read the corresponding data from all CSRs in the bank. It will then compare the calculated address with the incoming address, and decide which remote node has the line cached if any. The functionality can be seen in figure 5.3.

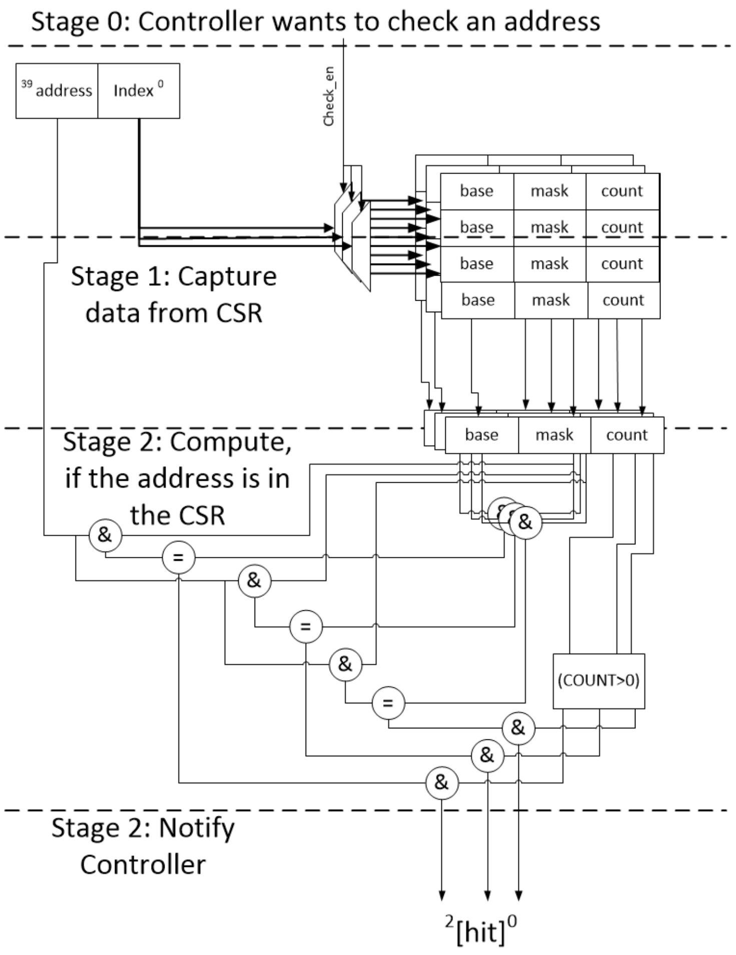

Stage 0: Controller wants to check an address
39 address
Index 0
Check_en
base
mask
count
Stage 1: Capture data from CSR
Stage 2: Compute, if the address is in the CSR
&
=
(COUNT>0)
Stage 2: Notify Controller
2[hit]0

Figure 5.3: Source Filter Check Address Module

The module consists of three stages. Stage 0 is initiated when the controller wants to read or write to an address, and so wants to know whether to inform remote controllers. The CSR register bank then provides the stream data for that index. The stream data is then compared to the incoming address. The last stage informs the controller about any remote controllers that may have cached that particular address.

### 5.1.4 CSR Register Bank

The register bank consists of $N-1$ ($N = Cores$) individual registers. Each register maps to one remote node, and contains information about the node's cache content. Figure 5.4 shows the implementation of one of the register.

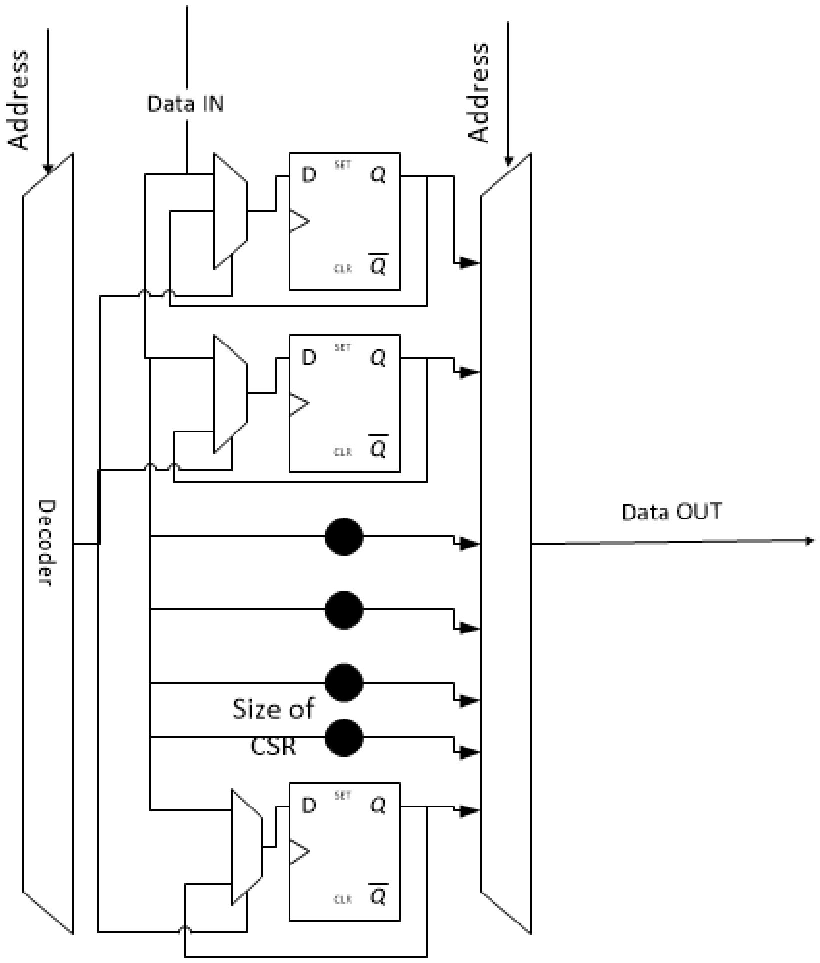


Figure 5.4: CSR Register

One register consists of:

- $CSR_{Size} \times CSR_{Width}$ flip flops.
- The width of the CSR is defined as: $base + mask + count$.
- Where base and mask are both: $Address\_Bits - log_2(CSR_{Size})$. $CSR_{Size}$ is the size of the register.

Each of these registers are then pooled together in a register bank, as figure5.5 depicts.

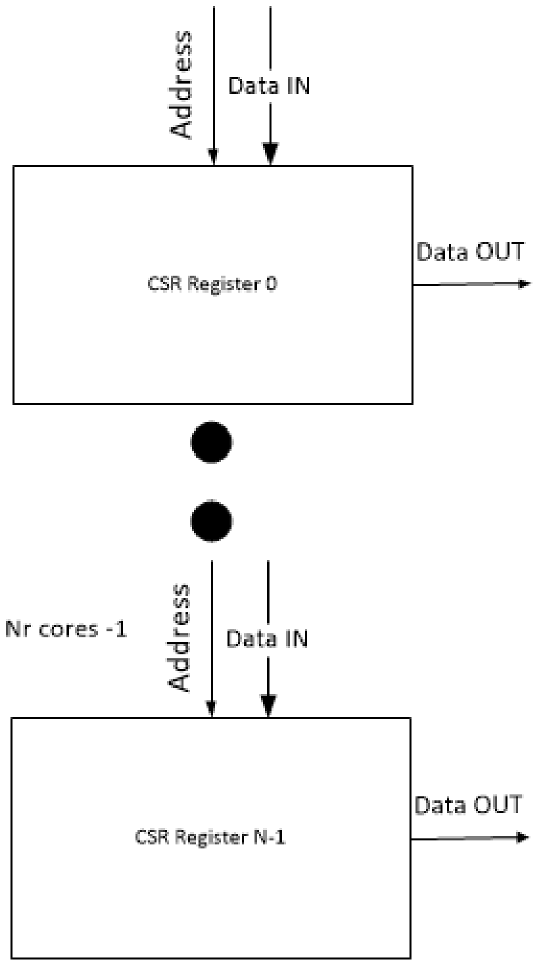


Figure 5.5: CSR Register Bank

Each of the CSR register in that bank maps directly to one remote core. The update logic takes hand of the maintaining the registers.

## 5.2 Verification

In any hardware design process, the verification is the most time consuming and important task. This design has been verified using *formal verification*, *SystemVerilogAssertions*(SVA) and *JasperGold*. Another possible method would have been to use UVM (Universal Verification Methodology). Using two methodologies enables the design to be tested for more cases than just using one. Which means that the probability of the design fail decreases. A third possible verification method could be to actually plug the filter in to a GPU and see how it works. Because of the time limit only formal verification was done. UVM will be described in the next section.

### 5.2.1 UVM

Universal verification methodology is a standardized method for verifying, designed by (Accellera n.d.). UVM provides a framework to achieve coverage-driven verification (CDV), which uses automatic test generation, self-checking testbenches and coverage to reduce the verification time of a design. A UVM testbench consists of reusable verification components. A verification component is a verification environment used to verify your device under test (DUT). The environment has elements for stimulating, checking and coverage collecting for the DUT.

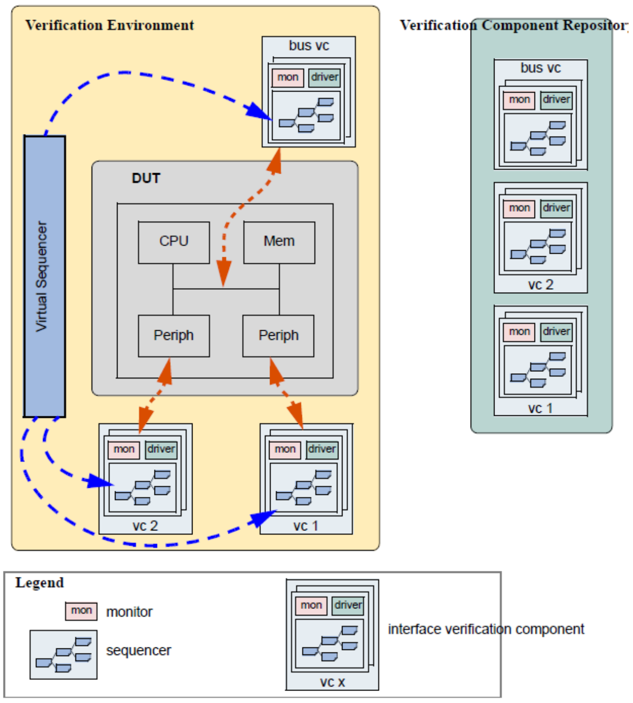


Figure 5.6: UVM: Verification EnvironmentAccellera (n.d.)

Figure 5.6 shows a typical verification environment. Components in a verification component are:

- **Data Item**: Data items are inputs to the DUT, like bus transactions and instructions.

- **Driver**: The driver emulates logic that drives the DUT; it typically receives a data item and drives it to the DUT.
- **Sequencer**: The sequencer is an advanced stimuli generator which controls the data items provided for the driver.
- **Monitor**: The monitor passively samples DUT signals, handles coverage information, and perform checking.

### 5.2.2 Formal Verification

There are three requiremens to formally prove how correct a system is (Raman 2012):

1. A mathematical model of the system with sufficient accuracy.
2. A language that supports property creation.
3. Some method of checking the property against the model.

The methods for formal verification usually involves exhaustive state space exploration. For this reason the memory requirements and verification time increases quickly when the complexity of the design increases. There are several methods for doing this, details can be found in (Raman 2012). SystemVerilog was used to create properties for this design. SystemVerilog is an extension to Verilog; it extends functionality to create properties. The properties can be asserted and formally tested using JasperGold.

### 5.2.3 Design Verification

The verification of the design was done in steps for each sub-module, as well as for the toplevel.

#### 5.2.3.1 Toplevel Properties

1. **Property "Valid transaction leads to execution":** This property checks that if there is a valid transaction on the bus interface, the interconnect surveillance module should set either add_en or remove_en high.

```
property pr_valid_check;
    disable iff(!reset_n)
    (valid & (op != NO_OP)) |=> (add_en | remove_en
        );
endproperty
as_valid_check : assert property(pr_valid_check);
```

2. **Property ”Valid transaction leads to CSR write:** This property checks when there is a valid transaction on the bus interface; the system should write the CSR register bank after 4 clock cycles. Initially JasperGold reported a counter example for this property, implying that there might be a bug in the system. When going through the source code a bug was found in stage 2 and 3 of the pipeline. The chip-enable signal for write (ce1) was set in stage 3, but it used data from stage 2 to calculate which register file to write to.

```
    property pr_check_addr_write_read;
        disable iff(!reset_n)
        (valid & (op != NO_OP) & (source > 0)) |=> ##3
            (ce1 > 0);
    endproperty
    as_check_addr_write_read : assert property(
        pr_check_addr_write_read);

endmodule
```

**Results:**

```
    Total Tasks         : 1
    Total Properties    : 4
          assumptions   : 0
           - approved   : 0
           - temporary  : 0
          assertions    : 2
           - proven     : 2                    ( 100.0% )
           - marked_proven: 0                  ( 0.0% )
           - cex        : 0                    ( 0.0% )
           - ar_cex     : 0                    ( 0.0% )
           - undetermined : 0                  ( 0.0% )
           - unprocessed  : 0                  ( 0.0% )
           - error      : 0                    ( 0.0% )
          covers        : 2
           - unreachable  : 0                  ( 0.0% )
           - covered    : 2                    ( 100.0% )
           - ar_covered : 0                    ( 0.0% )
           - undetermined : 0                  ( 0.0% )
           - unprocessed  : 0                  ( 0.0% )
           - error      : 0                    ( 0.0% )
```

After the bug found by property 2 was fixed, JasperGold reported no counter-examples.

#### 5.2.3.2 Interconnect Surveillance Properties

1. **Property "No valid transaction does not lead to execution":** This property checks that if there is no valid transaction on the bus interface, the interconnect surveillance module should not set add_en or remove_en high.

```
property pr_check_valid_0;
    disable iff(!reset_n)
    (!valid) |=> (!(add_en | remove_en));
endproperty
as_check_valid_0 : assert property(pr_check_valid_0
   );
```

2. **Property "Onehot remove_en and add_en ":** Output ports add_en and remove_en high should never be high simultaneously.

```
property pr_onehot_add_remove;
    disable iff(!reset_n)
    $onehot0({remove_en,add_en});
endproperty
as_onehot_add_remove : assert property(
   pr_onehot_add_remove);
```

3. **Property "Remote node equal transaction source ":** The remote node signal (r_node) should be the same as the source of the transaction.

```
//r_node = source
property pr_source_r_node;
    disable iff(!reset_n)
    (r_node == source);
endproperty
as_source_r_node : assert property(pr_source_r_node
   );
```

4. **Property "Correct address sampling ":** If the transaction is valid the sampled addr_out should be equal to the transaction address (addr) at the time of sampling.

```
property pr_addr_addr_out;
    disable iff(!reset_n)
    (valid) |=>($past(addr) == addr_out);
endproperty
as_addr_addr_out : assert property(pr_addr_addr_out
   );
```

5. **Property "Evict leads to removal of address":** If the bus transaction is an evict, the remove_en should be set. This indicates that the update module should remove the address from the CSRs.

```
property pr_evict;
    disable iff(!reset_n)
    (valid & (op==EVICT)) |=> (remove_en);
endproperty
as_evict : assert property(pr_evict);
```

6. **Property "Read or Write leads to adding of address":** If the bus transaction is read/write the add_en should be set. This indicates that the update module should add that particular address.

```
property pr_write_read;
    disable iff(!reset_n)
    (valid & ((op == READ) | (op == WRITE) )) |=> (
       add_en);
endproperty
as_write_read : assert property(pr_write_read);
```

7. **Property "Reset test":** If reset is asserted no operation should be started.

```
property pr_reset_n;
    (reset_n) |-> (!(remove_en & add_en));
endproperty
as_reset_n : assert property(pr_reset_n);
```

**Results:**

```
    Total  Tasks            :  1
    Total  Properties       :  14
           assumptions      :  0
            -  approved     :  0
            -  temporary    :  0
           assertions       :  8
            -  proven       :  8                    (  100.0%  )
            -  marked_proven:  0                    (  0.0%  )
            -  cex          :  0                    (  0.0%  )
            -  ar_cex       :  0                    (  0.0%  )
            -  undetermined :  0                    (  0.0%  )
            -  unprocessed  :  0                    (  0.0%  )
            -  error        :  0                    (  0.0%  )
           covers           :  6
            -  unreachable  :  0                    (  0.0%  )
            -  covered      :  6                    (  100.0%  )
            -  ar_covered   :  0                    (  0.0%  )
            -  undetermined :  0                    (  0.0%  )
            -  unprocessed  :  0                    (  0.0%  )
            -  error        :  0                    (  0.0%  )
```

The results show that all properties pass and no counter examples (cex) are found.

#### 5.2.3.3 Update Address Properties

1. **Property "Any operation ends after 3-cycles":** After a operation has started is should be written to the CSR after three cycles.

```
    property pr_finish_test;
        disable iff(!reset_n)
        (p0_enable & (r_node[0] > 0)) |=> ##2 (ce1 > 0)
            ;
    endproperty
    as_finish_test : assert property(pr_finish_test);
```

2. **Property "If read, address should be valid":** If chip-enable for the read-port-0 is enabled the address to read from should be valid.

```
    // If ce0, the read lines to the CSR should not be
       x
    property pr_x_read_line;
        disable iff(!reset_n)
        ce0 |-> (!$isunknown(a0));
    endproperty
```

```
as_x_read_line : assert property(pr_x_read_line);
```

3. **Property "Data forwarding check":** If the pipeline is enabled and the address is similar to the address in stage two, the forwarding data snatch signal should be set high.

```
// If p0_enable and p1, and a1 == a1_1, snatch_data
    ==1;
property pr_snatch_data_en;
    disable iff(!reset_n)
    (p0_enable & p1_enable & (a1 == a1_1)) |->
        snatch_data_en;
endproperty
as_snatch_data_en : assert property(
    pr_snatch_data_en);
```

4. **Property "Pipeline stage 1 enable":** If stage 0 in the pipeline is enabled stage 1 should also be enabled.

```
property pr_p1_enable;
    disable iff(!reset_n)
    p0_enable |=> p1_enable;
endproperty
as_p1_enable : assert property(pr_p1_enable);
```

5. **Property "Stage 1 data forwarding":** When stage 1 is enabled and 2-stage data forwarding is enabled, the data should be taken from the output registers in this state.

```
property pr_snatch_data_bus_base;
    disable iff(!reset_n)
    (p1_enable & snatch_data_from_bus_1) |=> (
        basei_reg == $past(baseo));
endproperty
as_snatch_data_bus_base : assert property(
    pr_snatch_data_bus_base);
```

6. **Property "Stage 1 no data forwarding":** If data forwarding for stage 1 is not set, data should come from the CSR bank.

```
property pr_not_snatch_data_bus_base;
    disable iff(!reset_n)
    (p1_enable & !snatch_data_from_bus_1) |=> (
        basei_reg == $past(basei));
endproperty
as_not_snatch_data_bus_base : assert property(
    pr_not_snatch_data_bus_base);
```

7. **Property "Pipeline stage 2 enable":** If stage 1 in the pipeline is enabled stage 2 should also be enabled.

```
property pr_p2_enable;
    disable iff(!reset_n)
    p1_enable |=> p2_enable;
endproperty
as_p2_enable : assert property(pr_p2_enable);
```

8. **Property "Stage 2 data forwarding":** When data forwarding for stage 2 is enabled the data used for computation should be taken from output registers.

```
property pr_snatch_data;
    disable iff(!reset_n)
    (snatch_data_en) |=> ##1 (base_sel == baseo);
endproperty
as_snatch_data : assert property(pr_snatch_data);
```

9. **Property "Stage 2 no data forwarding":** When data forwarding for stage 2 is not enabled data should be taken from input registers.

```
property pr_not_snatch_data;
    disable iff(!reset_n)
    (!snatch_data_en) |=> ##1 (base_sel ==
        basei_reg);
endproperty
as_not_snatch_data : assert property(
    pr_not_snatch_data);
```

10. **Property "Pipeline stage 3 enable":** If stage 2 in the pipeline is enabled stage 3 should also be enabled.

```
property pr_p3_enable;
    disable iff(!reset_n)
    p2_enable |=> p3_enable;
endproperty
as_p3_enable : assert property(pr_p3_enable);
```

**Results:**

```
    Total  Tasks            :  1
    Total  Properties       :  21
           assumptions      :  1
            - approved      :  0                    (  0.0%  )
            - temporary     :  1                    (  100.0%  )
           assertions       :  10
            - proven        :  10                   (  100.0%  )
            - marked_proven :  0                    (  0.0%  )
            - cex           :  0                    (  0.0%  )
            - ar_cex        :  0                    (  0.0%  )
            - undetermined  :  0                    (  0.0%  )
            - unprocessed   :  0                    (  0.0%  )
            - error         :  0                    (  0.0%  )
           covers           :  10
            - unreachable   :  0                    (  0.0%  )
            - covered       :  10                   (  100.0%  )
            - ar_covered    :  0                    (  0.0%  )
            - undetermined  :  0                    (  0.0%  )
            - unprocessed   :  0                    (  0.0%  )
            - error         :  0                    (  0.0%  )
```

The results show that all properties pass and no counter examples (cex) are found.

#### 5.2.3.4 Check Address Properties

1. **Property "Any operation ends after 2-cycles":** After a operation has started the controller should be notified after two cycles.

```
    property pr_check_finish;
        disable iff(!reset_n)
        check_en |=> ##1 finished;
    endproperty
    as_check_finish : assert property(pr_check_finish);
```

2. **Property "Stage 1 enable":** If the pipeline has started, stage 1 should also be enabled.

```
    property pr_p1_enable;
        disable iff(!reset_n)
        p0_enable |=> p1_enable;
    endproperty
    as_check_p1_enable : assert property(pr_p1_enable);
```

3. **Property "Stage 2 enable":** If stage 1 is enabled, stage 2 should also be enabled.

```
    property pr_p2_enable;
        disable iff(!reset_n)
        p1_enable |=> p2_enable;
    endproperty
    as_check_p2_enable : assert property(pr_p2_enable);
```

4. **Property "Correct calculation of CSR data":** Given the data from each CSR in the register bank, the correct outputs should be set.

```
    property pr_check_comb_logic(i);
        disable iff(!reset_n)
            ( ((basei_reg[i] & maski_reg[i]) ==
                addr_to_compare[i]) & (cnti_reg[i] > 0)
                ) |-> hits_out[i];
    endproperty
    generate
        for (genvar i = 0; i < NR_CORES-1; i = i + 1)
           begin: logic_ass
            as_check_com_logic : assert property(
                pr_check_comb_logic(i));
        end
    endgenerate
```

**Results:**

```
    Total Tasks              : 1
    Total Properties         : 12
          assumptions        : 0
           - approved        : 0
           - temporary       : 0
          assertions         : 6
           - proven          : 6                    ( 100.0% )
           - marked_proven   : 0                    ( 0.0% )
           - cex             : 0                    ( 0.0% )
           - ar_cex          : 0                    ( 0.0% )
           - undetermined    : 0                    ( 0.0% )
           - unprocessed     : 0                    ( 0.0% )
           - error           : 0                    ( 0.0% )
          covers             : 6
           - unreachable     : 0                    ( 0.0% )
           - covered         : 6                    ( 100.0% )
           - ar_covered      : 0                    ( 0.0% )
           - undetermined    : 0                    ( 0.0% )
           - unprocessed     : 0                    ( 0.0% )
           - error           : 0                    ( 0.0% )
```

The results show that all properties pass and no counter examples (cex) are found.

## 5.3 Summary

A non broadcasting-source-CSR filter was designed in Verilog HDL, using Vcs as compiler and simulator. The design was then verified using JasperGold, which is a formal verification tool. The tool takes the RTL source code, and compares it to defined rules written in SystemVerilog. JasperGold reported no rule violation. Even though the tool did not report any rule violations the design is still not completely verified. It has to be tested even more thoroughly using UVM or other methods to be sure that it is working as intended. But the filter is still so far formally verified.

In the next chapter I will discuss power simulations and estimations, in order to see whether the design actually has a net positive power effect on a multi-core system.

# Chapter 6

# Power Consumption Analysis

In order to decide whether the source filter is worth implementing, the power of the structure has to be analyzed. The extra hardware the filter represents costs power because it has to be updated regularly. It also costs power to lookup in the filter every time a transaction is about to be sent across the interconnect. The filter has to use less power than it can save (in interconnect and tag lookups), so that the net power after the filter installation is lower than before the filter implementation. If not the system will actually use more power than without the filter, which is kind of useless. This chapter is unfortunately going to be a lot shorter than intended because the power simulation of the designed filter encountered some unknown error.

## 6.1 Power Consumption

Power is simply the current multiplied with the voltage across a given element: $P(t) = I(t)V(t)$, the energy consumed by the same device is the integral over a given time T: $\int_0^T P(t)dt$. The average power over this interval is:

$$P_{avg} = \frac{E}{T} = \frac{1}{T}\int_0^T P(t)dt \tag{6.1}$$

This is the simple definition of power, but it can also be expanded to include capacitance. Which is very important when we talk about digital circuitry. Digital circuitry is driven by gates, which basically means charging and discharging capacitors.

## 6.2 CMOS Power

There are basically two sources of power dissipation in CMOS:

- *Dynamic dissipation*: Due to discharging the load capacitance of the gates. And short circuiting while the p and nmos are on at the same time

- *Static dissipation*: Due to various leakage.

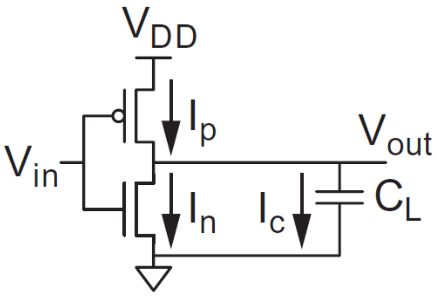


Figure 6.1: CMOS inverter

Figure 6.1 shows a basic digital inverter. The inverter is driven by some $V_{DD}$ and the input is driven by some voltage $V_{IN}$. The energy stored in a capacitor is:

$$E_C = \int_0^{\infty} I(t)V(t)\mathrm{d}t = \int_0^{\infty} C\frac{dV}{dt}V(t)\mathrm{d}t = C\int_0^{V_C} V(t)\mathrm{d}v = \frac{1}{2}CV_C^2 \tag{6.2}$$

While the energy provided from the supply to the same capacitor will be:

$$E_C = \int_0^{\infty} I(t)V_{DD}\mathrm{d}t = \int_0^{\infty} C\frac{dV}{dt}V_{DD}\mathrm{d}t = CV_{DD}\int_0^{V_{DD}} \mathrm{d}v = CV_{DD}^2 \tag{6.3}$$

### 6.2.1 Dynamic Power

As you can see the energy actually stored in the capacitor is half of what the power supply is delivering. The rest is dissipated through heat. When the capacitance is discharged the rest of the energy will also be dissipated into heat. So let us say that the inverter switches at some frequency $f$ over some time $T$ the load will be charged and discharged $Tf$. The average power dissipation will then from (6.1) be :

$$P_{switch} = \frac{E}{T} = \frac{TfCV_{DD}^2}{T} = CV_{DD}^2 f \tag{6.4}$$

This can further be refined using the activity factor $\alpha$, which is multiplied with the frequency. The activity factor takes into account that every gate does not change value every clock cycle.

$$P_{switch} = CV_{DD}^2 f = \alpha CV_{DD}^2 f \tag{6.5}$$

## 6.3 Power Estimation of Tag Lookups and Transactions

In order to see if the filter will lower the power consumption of the system, it has to be compared to the power consumption with and without the filter. The first step is to estimate the cost of a tag lookup and a snoop transaction. The next step is to find the cost of a filter lookup and updating. The third step is to compare the two previous steps with the benchmark results to see if the filter saves power.

### 6.3.1 Tag Lookups

A simple dual-port tag ram was created[1]. Read current and voltage was provided by the synthesis tool. This along with the clock frequency can be used to calculate the read energy:

$$E_{UTag} = \frac{V_{DD}I}{f} \tag{6.6}$$

Using the benchmark results for unnecessary snoops, the total energy spent with and without the filter can be calculated.

### 6.3.2 Transactions

The energy needed to perform one transaction can be estimated using the process library's wire model, which provides the capacitance and resistance per micrometer. By estimating the length between the cores, the total capacitance of the wire needed to connect two cores together can be estimated. Then using (6.3), the energy needed to send one bit can be estimated:

$$E_{ST} = \frac{C_{Wire}}{\mu m} l_{Wire} V_{DD}^2 \tag{6.7}$$

Knowing that you have to send; address, transaction type, source and destination, the total transaction energy can be estimated. This number will be a lower bound of the energy needed for one transaction, as the switching logic involved in the transaction is not estimated. The results from this estimation would have been combined with the snoop induced transaction results to give energy consumption per benchmark with or without filter.

## 6.4 Power Simulation of Filter

The power of the filter is estimated using this method: The filter is simulated using generated stimuli. The stimuli has around 5,000 filter lookups which will give a good average power estimate. These generated stimuli are dumped to a waveform file(.vpd/.vps) using Vcs as simulator. The waveform is then converted to a SAIF format, which contains switching information for all the nodes and nets. This is where the power simulation went wrong; the SAIF creation failed and reported no

[1]Process library28 nm

switching activity. After the SAIF has been created, the design will be synthesized using DesignCompiler, and a net list will be created. The net list together with the SAIF will then be used to estimate average power consumption. Then by using the clock frequency an estimation of one filter lookup can done. The energy for the filter has these symbols:

- Update energy: $E_{FUpdate}$
- Lookup energy: $E_{FLookup}$

## 6.5 Power Estimation of Benchmarks

After the energy consumption of the different parts in the system has been estimated, the total energy consumption with or without filters can be estimated. By looking at the benchmark results in table 4.5 where the results for the different source filter configurations the filter energy can be estimated using equation (6.8). BF means before filter, and AF means after filter.

$$E_{Total}(AF) = (E_{FUpdate} + E_{FLookup})N_{ST}(BF) + E_{ST}N_{ST}(AF) + E_{UTag}N_{UTag}(AF) \tag{6.8}$$

The energy consumption before the filter can be estimated using (6.9)

$$E_{Total}(BF) = E_{ST}N_{ST}(BF) + E_{UTag}N_{UTag}(BF) \tag{6.9}$$

If the filter is successful in reducing the consumption for a benchmark, $E_{Total}(AF) > E_{Total}(BF)$ should hold.

## 6.6 Summary

This section is unfortunately very short since an error occurred when the switching activity of the filter was about to be estimated, but a method for estimating the consumption has been presented. Regardless of the error in the analysis, the filter saves a lot of transactions and tag lookups. This will probably decrease the power consumption of the system. However a more thorough power analysis has to be completed before a proper conclusion can be made.

# Chapter 7

# Conclusion

This project has successfully modeled and designed snoop filters for use in GPUs for mobile SoCs. Two new promising filters has been invented during my work with this thesis. These filters perform better than previous filters based on the SR/CSR filter, at the cost of additional hardware. The *source CSR* has the highest cost, but also the highest power saving potential. It does not only decrease tag lookups in remote nodes, it also decreases the needed transaction count for maintaining coherence. The source filter may involve some architectural changes on systems that does not use a single shared bus. The needed architecture for such a system has to be thoroughly explored, to ensure that the filter will not compromise coherence and consistency while still reducing power in the system.

The *hashed-index CSR* shows that filter resource selection has tremendous effect on the accuracy of the filter. Sadly the hashed filter does not perform well for all benchmarks. An ideal solution could have been to alternate between the new hashed method, and the old method where the index is selected by the lower bits. Then, depending on the benchmark, even higher accuracy could have been reached.

The project also showed that benchmarks for GPUs have different memory footprints than benchmarks used for CPUs. In CPUs running embedded benchmarks, the MESI protocol has a tremendous effect on the power consumption acting as a source filter. For GPUs running OpenCl benchmarks the MESI protocol had virtually no effect on the power consumption of the system.

The source filter design was also implemented in Verilog HDL, then simulated using Vcs as compiler and simulator. The design was then formally verified using JasperGold and SystemVerilog assertions. In order to perform power analysis of the filer, the waveform from the RTL code was dumped to a waveform file, which was then converted to a SAIF file. The conversion however did not go as planned, and consequently did not report any switching activity in the design. Sadly this meant that I did not get any power estimates for the filter, which means that the total power saving is unknown. This thesis does however include a full power estimating methodology in chapter 6. Still the results in this thesis show that the source filter has huge potential in limiting both transactions and tag lookups in GPUs running OpenCl benchmarks.

## 7.1 Future Work

In order for the filter to be implemented in a real system these steps need to be taken.

1. Proper power simulation of the source filter in order to figure out if the filter is actually able to save power.
2. Explore needed architectural changes for the filter to work properly on a modern multi core system.
3. Test that the filter works on GPUs.
4. Explore further on the Hashed-index design. Combining the hashed-index with the source filter for instance.

# Appendix A

# Dynamic Multicore Model

## A.1 L2 Controller: MSI FSM

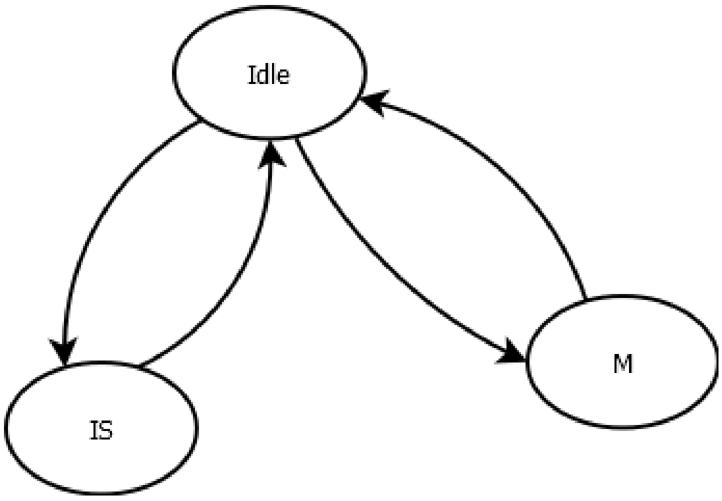


Figure A.1: L2 Controller FSM for MSI Coherence Protocol

## A.2 L1 Controller: MESI FSM

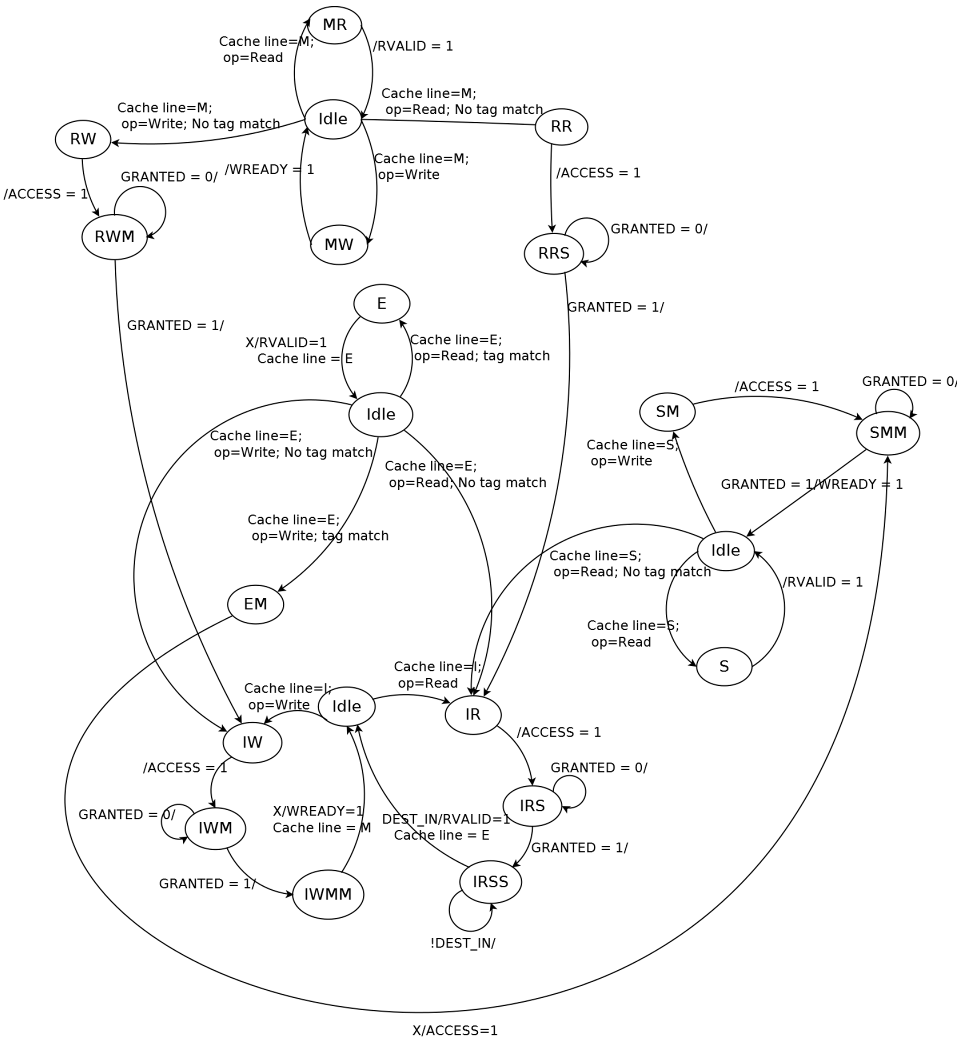

MR
Idle
RR
RW
RWM
MW
RRS
Cache line=M; op=Read
/RVALID = 1
Cache line=M; op=Read; No tag match
Cache line=M; op=Write; No tag match
/WREADY = 1
Cache line=M; op=Write
GRANTED = 0/
/ACCESS = 1
GRANTED = 1/
E
Idle
EM
X/RVALID=1 Cache line = E
Cache line=E; op=Read; tag match
Cache line=E; op=Write; No tag match
Cache line=E; op=Read; No tag match
Cache line=E; op=Write; tag match
SM
SMM
Idle
S
Cache line=S; op=Write
GRANTED = 1/WREADY = 1
Cache line=S; op=Read; No tag match
/RVALID = 1
Cache line=S; op=Read
IW
IWM
IWMM
IR
IRS
IRSS
Cache line=I; op=Write
Cache line=I; op=Read
X/WREADY=1 Cache line = M
DEST_IN/RVALID=1 Cache line = E
!DEST_IN/
X/ACCESS=1

Figure A.2: L1 Controller FSM for MESI Coherence Protocol

# Appendix B

# Verilog Code for Source CSR Filter

## B.1 Toplevel for Source CSR Filter

Verilog B.1: Toplevel Module for Source Filter

```
//==============================================================
//   Function: Toplevel for CSR source filter.
//   Coder:   Rasmus Ulfsnes
//   Date:    2013
//==============================================================

module src_csr_top_level(/*AUTOARG*/
  // Outputs
  finished, addr_out, hits_out,
  // Inputs
  clk, reset_n, ic, valid, check_en, addr_in
  );
`include "filter_constants.v"

parameter IC_WIDTH = PAXI_ADDR_BITS + NR_CORES-1 +
   IC_OP_WIDTH;

//GLOBAL INPUTS:
//
   ==========================================================================

input  clk;
input  reset_n;

// Global connections to ic_surv
```

```
//
    ==================================================================
input  [IC_WIDTH-1:0]          ic;
input                          valid;
//OUTPUTS:


//GLOBAL connections to check_addr:
//
    ==================================================================
//Inputs:
input                            check_en;
input  [PAXI_ADDR_BITS-1:0]      addr_in;
//Outputs:
output                           finished;
output  [PAXI_ADDR_BITS-1:0]     addr_out;
output  [NR_CORES-2:0]           hits_out;

// COMM Between ic_surv and update logic:
//
    ==================================================================
wire  add_en;
wire  remove_en;
wire  [NR_CORES-2:0]  r_node;


// COMM between update_addr and register file and ic_surv
//
    ==================================================================
//CSR:
wire  [(NR_CORES-1)*CSR_REG_WIDTH-1:0]  csr_to_add_b;
wire  [(NR_CORES-1)*CSR_O_WIDTH-1:0]    add_to_csr_b;
wire  [CSR_REG_WIDTH-1:0]               csr_to_add[NR_CORES
    -2:0];
wire  [CSR_O_WIDTH-1:0]                 add_to_csr[NR_CORES
    -2:0];

wire  [NR_CORES-2:0]            ce0;
wire  [NR_CORES-2:0]            ce1;
wire  [NR_CORES-2:0]            we1;
wire  [CSR_INDEX_BITS-1:0]      a0[NR_CORES-2:0];
wire [CSR_INDEX_BITS-1:0]      a1[NR_CORES-2:0];
wire [2*CSR_BM_BITS+CSR_CNT_BITS-1:0]    di1[NR_CORES-2:0];

//ic_surv
wire [PAXI_ADDR_BITS-1:0]                addr_ic_update;

//==========================================================================


//Generate connections Between update_and CSR:
genvar i;
generate
  for(i = 0; i < NR_CORES-1;i = i + 1 )begin: i_loop
    assign csr_to_add_b[(i+1)*CSR_REG_WIDTH-1:CSR_REG_WIDTH*i] = csr_to_add[i];
    assign add_to_csr[i] = add_to_csr_b[(i+1)*CSR_O_WIDTH-1:CSR_O_WIDTH*i] ;
    assign di1[i] = add_to_csr[i][CSR_O_WIDTH-1:CSR_O_WIDTH-CSR_REG_WIDTH];
    assign ce0[i] = add_to_csr[i][2*CSR_INDEX_BITS+2];
    assign ce1[i] = add_to_csr[i][2*CSR_INDEX_BITS+1];
    assign we1[i] = add_to_csr[i][2*CSR_INDEX_BITS];
    assign a0[i] = add_to_csr[i][2*CSR_INDEX_BITS-1:CSR_INDEX_BITS];
    assign a1[i] = add_to_csr[i][CSR_INDEX_BITS-1:0];
  end
endgenerate
//=====================================================================

//Communication between check logic and CSRs:
//=====================================================================
wire                                ce2;
wire [CSR_INDEX_BITS-1:0]      a2;
wire [(NR_CORES-1)*CSR_REG_WIDTH-1:0] csr_to_check_b;
wire [CSR_REG_WIDTH-1:0]                csr_to_check[NR_CORES-2:0];
//Generate connection between check_addr and CSR
//=====================================================================

generate
  for(i = 0; i < NR_CORES-1;i = i + 1 )begin: i_loop_1
    assign csr_to_check_b[(i+1)*CSR_REG_WIDTH-1:CSR_REG_WIDTH*i] = csr_to_check[i];
  end
endgenerate
// Instantiate ic_surv:
src_csr_ic_surv ic_surv(
  .clk          (clk),
  .reset_n      (reset_n),
  .valid        (valid),
  .ic           (ic),
  .addr_out     (addr_ic_update),
  .add_en       (add_en),
  .remove_en    (remove_en),
  .r_node       (r_node)
);


// Instantiate add module
src_csr_update_addr update_addr(
  .clk          (clk),
  .reset_n      (reset_n),
  .remove_en    (remove_en),
  .add_en       (add_en),
  .addr_in      (addr_ic_update),
  .r_node       (r_node),
  .csr_i_b      (csr_to_add_b),
  .csr_o_b      (add_to_csr_b)
);

// Instantiate check module
src_csr_check_addr check_addr(
  .clk            (clk),
  .reset_n        (reset_n),
  .check_en       (check_en),
  .addr_in        (addr_in),
  .finished       (finished),
  .addr_out       (addr_out),
  .hits_out       (hits_out),
  .csr_i_b_check(csr_to_check_b),
  .ce2(ce2),
  .a2(a2)
  );
// Instantiate register file
genvar k;
generate
  for(i = 0; i < NR_CORES-1;i = i + 1 )begin:
      instantiate_CSR
```

```

    sim_mem_3p #(.NAME(i)) reg_file (
      .clk(clk),
      .reset_n(reset_n),
      .ce0(ce0[i]),
      .a0(a0[i]),
      .do0(csr_to_add[i]),
      .ce1(ce1[i]),
      .we1(we1[i]),
      .a1(a1[i]),
      .di1(di1[i]),
      .a2(a2),
      .ce2(ce2),
      .do2(csr_to_check[i])
      );
  end
endgenerate







endmodule
```

## B.2 Interconnect Surveillance Module

Verilog B.2: Interconnect Surveillance Module

```
//=====================================================================
//  Function: Verilog module capable of monitoring
    interconnect,
//             deciding which source the operation is from
//  Coder:   Rasmus Ulfsnes
//  Date:    2013
//=====================================================================
//

module src_csr_ic_surv(/*AUTOARG*/
  // Outputs
  addr_out, add_en, remove_en,
  // Inputs
  clk, reset_n, valid, ic, r_node
  );
`include "filter_constants.v"
```

```

parameter IC_WIDTH = PAXI_ADDR_BITS + NR_CORES-1 +
    IC_OP_WIDTH;

input                         clk;
input                         reset_n;
input                         valid;

//  ic                        [IC_WIDTH-1:IC_WIDTH-
    PAXI_ADDR_BITS|IC_WIDTH-PAXI_ADDR_BITS-1:IC_OP_WIDTH|
    IC_OP_WIDTH-1:0]
//                            [            addr
                         |                      source
    |        op          ]
input  [IC_WIDTH-1:0]         ic;

output  [PAXI_ADDR_BITS-1:0]          addr_out;
output                                add_en;
output                                remove_en;
output  [NR_CORES-2:0]                r_node;

//Setting output registers:

wire  [PAXI_ADDR_BITS-1:0]  addr;
reg                         add_en;
reg                         remove_en;

wire  [IC_OP_WIDTH-1:0]     op;
reg  [NR_CORES-2:0]         source;
reg  [PAXI_ADDR_BITS-1:0]   addr_out;

assign  addr        = ic[IC_WIDTH-1:IC_WIDTH-PAXI_ADDR_BITS];
assign  r_node      = source;
assign  op          = ic[IC_OP_WIDTH-1:0];

// Logic check if there is a transaction on the bus
// Operations parameters:

parameter EVICT = 2'b11, WRITE = 2'b01, READ = 2'b10, NO_OP
    = 2'b00;
always @(posedge clk or negedge reset_n)begin//IC case
  if(!reset_n)begin //reset
    add_en <= 1'b0;
    remove_en <= 1'b0;
  end else if(valid) begin
```

```verilog
      addr_out   <= addr;
      source     <= ic[IC_WIDTH-PAXI_ADDR_BITS-1:IC_OP_WIDTH];
      case (op)
        EVICT: begin
                 add_en     <= 1'b0;
                 remove_en <= 1'b1;
               end
        READ:  begin
                 add_en     <= 1'b1;
                 remove_en <= 1'b0;
               end
        WRITE: begin
                 add_en     <= 1'b1;
                 remove_en <= 1'b0;
               end
        NO_OP: begin
                 add_en     <= 1'b0;
                 remove_en <= 1'b0;
               end
      endcase
    end else if(!valid) begin
      add_en <= 0;
      remove_en <= 0;
    end
end//IC case
endmodule
```

## B.3 Update Address Module

Verilog B.3: RTL Code for Updating and Address in the CSR Register Bank Module

```verilog
//=====================================================================
//   Function: Module containing verilog code to update an
     address in
//           the source Counting Stream Registers filter.
//           May need to enable stall function
//   Coder:  Rasmus Ulfsnes
//   Date:   2013
//=====================================================================


module src_csr_update_addr(clk, reset_n,remove_en, add_en,
addr_in, r_node, csr_i_b, csr_o_b);
`include "filter_constants.v"

```

```
input                         clk;
input                         reset_n;
input                         remove_en;
input                         add_en;
input  [PAXI_ADDR_BITS-1:0]   addr_in;
input  [NR_CORES-2:0]         r_node;
//output                        finished;
//output  [PAXI_ADDR_BITS-1:0]  addr_out;
//output                        op_out;

//Output from register files:
//                   [BASE:MASK:COUNT]
input  [(NR_CORES-1)*CSR_REG_WIDTH-1:0]                  csr_i_b;
//csr_o:                        [CSR_O_WIDTH:2*CSR_INDEX_BITS+3|2*CSR_INDEX_BITS+2|2*CSR_INDEX_BITS+1|2*CSR_INDEX_BITS|2*CSR_INDEX_BITS-1:CSR_INDEX_BITS|CSR_INDEX_BITS-1:0]
//Input to register file:[        di1                   |        ce0      |        ce1      |       we1      |       a0                    |      a1             ]
output  [(NR_CORES-1)*CSR_O_WIDTH-1:0]                   csr_o_b;



//Ports
wire  [(NR_CORES-1)*CSR_O_WIDTH-1:0]      csr_o_b;
wire  [(NR_CORES-1)*CSR_REG_WIDTH-1:0]    csr_i_b;

//Output registers:
reg  [CSR_BM_BITS-1:0]          baseo[NR_CORES-2:0];
reg  [CSR_BM_BITS-1:0]          masko[NR_CORES-2:0];
reg  [CSR_CNT_BITS-1:0]         cnto[NR_CORES-2:0];

//Input wires
wire  [CSR_BM_BITS-1:0]         basei[NR_CORES-2:0];
wire  [CSR_BM_BITS-1:0]         maski[NR_CORES-2:0];
wire  [CSR_CNT_BITS-1:0]        cnti[NR_CORES-2:0];

//Input data registers
reg  [CSR_BM_BITS-1:0]        basei_reg[NR_CORES-2:0];
reg  [CSR_BM_BITS-1:0]        maski_reg[NR_CORES-2:0];
reg  [CSR_CNT_BITS-1:0]       cnti_reg[NR_CORES-2:0];

//Combinational wires
```

```
wire  [CSR_BM_BITS-1:0]        base_sel [NR_CORES-2:0];
wire  [CSR_BM_BITS-1:0]        base [NR_CORES-2:0];
wire  [CSR_BM_BITS-1:0]        mask [NR_CORES-2:0];
wire  [CSR_BM_BITS-1:0]        mask_sel [NR_CORES-2:0];
wire  [CSR_BM_BITS-1:0]        mask_bits [NR_CORES-2:0];
wire  [CSR_CNT_BITS-1:0]       cnt_add [NR_CORES-2:0];
wire  [CSR_CNT_BITS-1:0]       cnt_rm [NR_CORES-2:0];
wire  [CSR_CNT_BITS-1:0]       cnt [NR_CORES-2:0];
wire  [CSR_CNT_BITS-1:0]       cnt_sel [NR_CORES-2:0];
//
//Internal registers, used in pipeline, etc.
reg                            p1_enable;
reg                            p2_enable;
reg                            p3_enable;
wire                           p0_enable;
reg                            op_1;
reg                            op_2;
//reg                            op_3;
reg  [NR_CORES-2:0]            r_node_1;
reg  [NR_CORES-2:0]            r_node_2;
reg  [PAXI_ADDR_BITS-1:0]      addr_in_1;
reg  [PAXI_ADDR_BITS-1:0]      addr_in_2;
reg  [PAXI_ADDR_BITS-1:0]      addr_in_3;
wire                           snatch_data_en;
reg                            snatch_data_1;
reg                            snatch_data_2;
wire                           snatch_data_from_bus_en;
reg                            snatch_data_from_bus_1;

wire  [CSR_BM_BITS-1:0]       bits_to_compare [NR_CORES-2:0];

wire  [NR_CORES-2:0]                        ce0;
wire  [NR_CORES-2:0]                        ce1;
wire  [NR_CORES-2:0]                        we1;
wire  [CSR_INDEX_BITS-1:0]                  a0;
wire  [CSR_INDEX_BITS-1:0]                  a1;
reg  [CSR_INDEX_BITS-1:0]                   a1_1;
reg  [CSR_INDEX_BITS-1:0]                   a1_2;
reg  [CSR_INDEX_BITS-1:0]                   a1_3;
wire  [CSR_REG_WIDTH-1:0]                  di1 [NR_CORES-2:0];

wire  [CSR_REG_WIDTH-1:0]      csr_i [NR_CORES-2:0];
wire  [CSR_O_WIDTH-1:0]        csr_o [NR_CORES-2:0];


```

```
// Assign statements:
genvar i;
generate
  for(i = 0; i < NR_CORES-1;i = i + 1 )begin: assign_loop
    //BUSES
    assign csr_i[i] = csr_i_b[(i+1)*CSR_REG_WIDTH-1:
        CSR_REG_WIDTH*i];
    assign csr_o_b[(i+1)*CSR_O_WIDTH-1:CSR_O_WIDTH*i] =
        csr_o[i];
    //Packing of output bus
    assign csr_o[i][2*CSR_INDEX_BITS+2] = ce0[i];
    assign csr_o[i][2*CSR_INDEX_BITS+1] = ce1[i];
    assign csr_o[i][2*CSR_INDEX_BITS] = we1[i];
    assign csr_o[i][2*CSR_INDEX_BITS-1:CSR_INDEX_BITS] = a0
        ;
    assign csr_o[i][CSR_INDEX_BITS-1:0] = a1_3;
    assign csr_o[i][CSR_O_WIDTH-1:CSR_O_WIDTH-CSR_REG_WIDTH
        ] = di1[i];
    assign di1[i][CSR_REG_WIDTH-1:CSR_REG_WIDTH-CSR_BM_BITS
        ] = baseo[i];
    assign di1[i][CSR_REG_WIDTH-CSR_BM_BITS-1:CSR_CNT_BITS
        ]= masko[i];
    assign di1[i][CSR_CNT_BITS-1:0] = cnto[i];

    //Unpacking of input bus
    assign basei[i] = csr_i[i][CSR_REG_WIDTH-1:
        CSR_REG_WIDTH-CSR_BM_BITS];
    assign maski[i] = csr_i[i][CSR_REG_WIDTH-CSR_BM_BITS-1:
        CSR_CNT_BITS];
    assign cnti[i] = csr_i[i][CSR_CNT_BITS-1:0];
  end
endgenerate
//*************************************************
//Stage 0 Sequential:
always @(posedge clk or negedge reset_n)begin
  if ( !reset_n )begin
    p1_enable <= 1'b0;
    op_1 <= 1'b0;
  end else begin
    //Enable pipe
    p1_enable <= p0_enable;
    snatch_data_1 <= snatch_data_en;
    snatch_data_from_bus_1 <= snatch_data_from_bus_en;
    //Data pipe
    //Which nodes to select
    op_1 <= remove_en;
    a1_1 <= a1;
    r_node_1 <= r_node;
    addr_in_1 <= addr_in;
  end
end


//Stage 0, Combinatorics:7 fffffff
assign snatch_data_en = (a1 == a1_1) & p0_enable &
   p1_enable;
assign snatch_data_from_bus_en = (a1 == a1_2) & p0_enable &
    p2_enable;
assign p0_enable = (add_en || remove_en);
assign a0 = a1;// (addr_in & CSR_INDEX_MASK);
assign a1 = addr_in[CSR_INDEX_BITS-1:0];
generate
  for(i = 0; i < NR_CORES-1;i = i + 1 )begin: ce0_loop
    assign ce0[i] = (p0_enable & !snatch_data_en & !
        snatch_data_from_bus_en) ? (r_node[i]) : 1'b0;
  end
endgenerate


//**************************************************
//Stage 1, get data from memory.
//Just waiting, moving the pipe further.
integer l;
always @(posedge clk)begin
  if(!reset_n)begin
    p2_enable <= 1'b0;
    op_2 <= 1'b0;
    a1_2 <= 5'b00000;
    snatch_data_2 <= 1'b0;
  end else begin//Move pipe
    p2_enable <= p1_enable;
    r_node_2 <= r_node_1;
    op_2 <= op_1;
    a1_2 <= a1_1;
    addr_in_2 <= addr_in_1;
    snatch_data_2 <= snatch_data_1;
    if ( p1_enable )begin
      if(snatch_data_from_bus_1)begin
        for(l = 0; l < NR_CORES-1; l = l + 1)begin
          basei_reg[l] <= baseo[l];
          maski_reg[l] <= masko[l];
          cnti_reg[l] <= cnto[l];
        end
      end else if(!snatch_data_from_bus_1)begin
        for(l = 0; l < NR_CORES-1; l = l + 1 )begin
          basei_reg[l] <= basei[l];
          maski_reg[l] <= maski[l];
          cnti_reg[l] <= cnti[l];
        end
      end
    end
  end
end
//***************************************************
//Stage 2, calculate new values:
//Combinational

generate
  for(i = 0; i < NR_CORES-1;i = i + 1 )begin: comb_loop
    //Calculation
    assign base[i] = addr_in_2[PAXI_ADDR_BITS-1:
        CSR_INDEX_BITS];
    assign cnt_sel[i] = (snatch_data_2) ? cnto[i] :
        cnti_reg[i];
    assign mask_sel[i] = (snatch_data_2) ? masko[i] :
        maski_reg[i];
    assign base_sel[i] = (snatch_data_2) ? baseo[i] :
        basei_reg[i];
    assign cnt_add[i] = cnt_sel[i] + 1'b1;
    assign cnt_rm[i] = cnt_sel[i] - 1'b1;
    assign bits_to_compare[i] = ~(base_sel[i] ^ addr_in_2[
        PAXI_ADDR_BITS-1:CSR_INDEX_BITS]);
    assign cnt[i] = (op_2) ? cnt_rm[i]: cnt_add[i];
    assign mask_bits[i] = bits_to_compare[i] & mask_sel[i];
    //Which data to snatch:
    assign mask[i] = (op_2) ? mask_sel[i] : ((cnt_sel[i] ==
         0) ? 35'h7FFFFFFFF : mask_bits[i] & mask_sel[i]);
  end
endgenerate
//Sequential part of Stage 2
integer k;
always @(posedge clk ) begin
  // Stage 2 stores the calculated csr data
  if ( !reset_n )begin
    p3_enable <= 1'b0;
```

```
      a1_3 <= 5'b00000;
    end else begin
      p3_enable <= p2_enable;
      a1_3 <= a1_2;
      //op_3 <= op_2;
      addr_in_3 <= addr_in_2;
      if ( p2_enable ) begin
        //Push completed index into register to allow check.
        for ( k = 0; k < NR_CORES-1; k = k +1)begin
          if ( r_node_2[k] )begin //Store the calculated CSR
              results:
            cnto[k] <= cnt[k];
            masko[k] <= mask[k];
            baseo[k] <= base[k];
          end
        end
      end
    end
end//stage2
//*****************************************************
//Stage 3 tell the CSR Register to store the data:
assign ce1 = (p3_enable) ? r_node_2 : 1'b0;
assign we1 = ce1;
//assign finished = p3_enable;
//assign addr_out = addr_in_3;
//assign op_out = op_3;

//stage3
endmodule//src_csr_update_addr
```

## B.4 Check Address Module

Verilog B.4: Module to check the register bank whether or not a certain address is cached, and in wich remote nodes.

```
//==================================================
//  Function: Verilog module that checks whether an
//            address maybe cached or not. This module will
     check the write/read bus to the CSR
//            to see whether or not someon is writin/
    reading to the same index as the check wants to
//            read. If it matches a read it will simply
    wait for the data to appear. If the
//            data is aleready on the bus, it will
    immidiately fetch the data.
//            else it will stall the update_addr module
    from executing any more updates.
```

```
//   Coder:   Rasmus Ulfsnes
//   Date:    2013
//=====================================================================


module src_csr_check_addr(/*AUTOARG*/
  // Outputs
  finished, addr_out, hits_out, ce2, a2,
  // Inputs
  clk, reset_n, check_en, addr_in, csr_i_b_check
  );
`include "filter_constants.v"

parameter NAME = "src_csr_add_addr";
//Parameter Cores can be 5,13,11,7 for the different
    combinations of remote nodes
parameter CORES=5;

input                            clk;
input                            reset_n;
input                            check_en;
input  [PAXI_ADDR_BITS-1:0]   addr_in;
// Inputs from the update_logic, to enable fetching data
    before it is saved
// into the CSR

output                           finished;
output  [PAXI_ADDR_BITS-1:0]  addr_out;
output  [NR_CORES-2:0]        hits_out;

//Output from register files:
//                        [BASE:MASK:COUNT]
input  [(NR_CORES-1)*CSR_REG_WIDTH-1:0]
    csr_i_b_check;
output  [CSR_INDEX_BITS-1:0]  a2;
output                           ce2;

//Input wires
wire  [CSR_BM_BITS-1:0]         basei[NR_CORES-2:0];
wire  [CSR_BM_BITS-1:0]         maski[NR_CORES-2:0];
wire  [CSR_CNT_BITS-1:0]        cnti[NR_CORES-2:0];

//Input data registers
reg  [CSR_BM_BITS-1:0]         basei_reg[NR_CORES-2:0];
reg  [CSR_BM_BITS-1:0]         maski_reg[NR_CORES-2:0];
reg  [CSR_CNT_BITS-1:0]      cnti_reg[NR_CORES-2:0];

//Internal registers, used in pipeline, etc.
wire                          p0_enable;
reg                           p1_enable;
reg                           p2_enable;

reg  [PAXI_ADDR_BITS-1:0]     addr_in_1;
reg  [PAXI_ADDR_BITS-1:0]     addr_in_2;


wire  [CSR_BM_BITS-1:0]      bits_to_compare[NR_CORES-2:0];
wire  [CSR_BM_BITS-1:0]      addr_to_compare;


wire  [CSR_REG_WIDTH-1:0]      csr_i[NR_CORES-2:0];


// Assign statements:
genvar i;
generate
  for(i = 0; i < NR_CORES-1;i = i + 1 )begin: assign_loop
    //BUSES
    assign csr_i[i] = csr_i_b_check[(i+1)*CSR_REG_WIDTH-1:
        CSR_REG_WIDTH*i];
    //Unpacking of input bus
    assign basei[i] = csr_i[i][CSR_REG_WIDTH-1:
        CSR_REG_WIDTH-CSR_BM_BITS];
    assign maski[i] = csr_i[i][CSR_REG_WIDTH-CSR_BM_BITS-1:
        CSR_CNT_BITS];
    assign cnti[i]  = csr_i[i][CSR_CNT_BITS-1:0];
  end
endgenerate

// Logic

//Stage 0, Combinatorics: Ask the CSR register file for
    data.
assign p0_enable = (check_en);
assign a2 = addr_in[CSR_INDEX_BITS-1:0];
assign ce2 = (p0_enable) ? 1'b1 : 1'b0;



// Stage 0 Sequential: Move the pipe to next stage
```

```
always @(posedge clk or negedge reset_n)begin
  if ( !reset_n )begin
    p1_enable <= 1'b0;
  end else begin
    //Enable pipe
    p1_enable <= p0_enable;
    //Data pipe
    addr_in_1 <= addr_in;
  end
end

// Stage 1: Capture data into input registers
integer l;
always @(posedge clk)begin
  if(!reset_n)begin
    p2_enable <= 1'b0;
  end else begin//Move pipe
    p2_enable <= p1_enable;
    addr_in_2 <= addr_in_1;
    if ( p1_enable )begin
      for (l = 0 ; l < NR_CORES-1; l = l + 1)begin
        basei_reg[l] <= basei[l];
        maski_reg[l] <= maski[l];
        cnti_reg[l] <= cnti[l];
      end
    end
  end
end
// Stage 2: Calculate data and notify requestor about hits:
assign addr_to_compare = addr_in_2[PAXI_ADDR_BITS-1:
   CSR_INDEX_BITS];
assign finished = p2_enable;
assign addr_out = addr_in_2;
generate
  for(i = 0; i < NR_CORES-1; i = i + 1)begin: hit_loop
    assign bits_to_compare[i] = basei_reg[i] & maski_reg[i
       ];
    assign hits_out[i] =(bits_to_compare[i] ==
       addr_to_compare) & (cnti_reg[i] > 0 );
  end
endgenerate


endmodule
```

# Appendix C

# SystemVerilog properties used to verify CSR Source Filter

## C.1 Toplevel Properties

SystemVerilog C.1: Toplevel verification properties.

```
// ****************************************************
// This file contains properties and such for formal verification
// of update_addr module in source filter
//
//
//
//
//
//

module src_csr_top_level_prop(/*AUTOARG*/
  // Inputs
  clk, reset_n, ic, valid, check_en, addr_in, finished,
     addr_out,
  hits_out, add_en, remove_en, r_node, csr_to_add_b,
     add_to_csr_b,
  csr_to_add, add_to_csr, ce0, ce1, we1, a0, a1, di1, ce2,
     a2,
  csr_to_check_b, csr_to_check
  );

`include "filter_constants.v"
```

```

parameter IC_WIDTH = PAXI_ADDR_BITS + NR_CORES-1 +
    IC_OP_WIDTH;
parameter EVICT = 2'b11, WRITE = 2'b01, READ = 2'b10, NO_OP
     = 2'b00;

//GLOBAL INPUTS:
//
    ==============================================================

input wire  clk;
input wire reset_n;

// Global connections to ic_surv
//
    ==============================================================

input wire [IC_WIDTH-1:0]       ic;
input wire                     valid;
//OUTPUTS:


//GLOBAL connections to check_addr:
//
    ==============================================================

//Inputs:
input wire                      check_en;
//Outputs:
input wire                      finished;
input wire [PAXI_ADDR_BITS-1:0] addr_out;
input wire [NR_CORES-2:0]       hits_out;

// COMM Between ic_surv and update logic:
//
    ==============================================================

input wire add_en;
input wire remove_en;
input wire [PAXI_ADDR_BITS-1:0] addr_in;
input wire [NR_CORES-2:0] r_node;


// COMM between update_addr and register file and ic_surv
```

```
//
    ================================================================================

//CSR:
input wire [(NR_CORES−1)*CSR_REG_WIDTH−1:0] csr_to_add_b;
input wire [(NR_CORES−1)*CSR_O_WIDTH−1:0]   add_to_csr_b;
input wire [CSR_REG_WIDTH−1:0]              csr_to_add[
    NR_CORES−2:0];
input wire [CSR_O_WIDTH−1:0]                add_to_csr[
    NR_CORES−2:0];

input wire [NR_CORES−2:0]          ce0;
input wire [NR_CORES−2:0]          ce1;
input wire [NR_CORES−2:0]          we1;
input wire [CSR_INDEX_BITS−1:0]    a0[NR_CORES−2:0];
input wire [CSR_INDEX_BITS−1:0]    a1[NR_CORES−2:0];
input wire [2*CSR_BM_BITS+CSR_CNT_BITS−1:0]  di1[NR_CORES
    −2:0];

//Communication between check logic and CSRs:
//========================================================================
input wire                                  ce2;
input wire [CSR_INDEX_BITS−1:0]             a2;
input wire [(NR_CORES−1)*CSR_REG_WIDTH−1:0] csr_to_check_b;
input wire [CSR_REG_WIDTH−1:0]              csr_to_check[
    NR_CORES−2:0];
//Generate connection between check_addr and CSR
//========================================================================


wire [IC_OP_WIDTH−1:0]       op;
wire [PAXI_ADDR_BITS−1:0]    addr;
wire [NR_CORES−2:0]          source;

assign op       = ic[IC_OP_WIDTH−1:0];
assign addr     = ic[IC_WIDTH−1:IC_WIDTH−PAXI_ADDR_BITS];
assign source   = ic[IC_WIDTH−PAXI_ADDR_BITS−1:IC_OP_WIDTH
    ];

    default clocking cb @(posedge clk); endclocking

    property pr_valid_check;
        disable iff(!reset_n)
        (valid & (op != NO_OP)) |=> (add_en | remove_en);
    endproperty
    as_valid_check : assert property(pr_valid_check);

    property pr_check_addr_write_read;
        disable iff(!reset_n)
        (valid & (op != NO_OP) & (source > 0)) |=> ##3 (ce1
            > 0);
    endproperty
    as_check_addr_write_read : assert property(
        pr_check_addr_write_read);

endmodule
```

## C.2 Interconnect Surveillance Properties

SystemVerilog C.2: Interconnect surveillance properties.

```
//
    ****************************************************
// This file contains properties and such for formal
    verification
// of update_addr module in source filter
//
//
//
//
//
//

module src_csr_ic_surv_prop(/*AUTOARG*/
   // Inputs
   clk, reset_n, valid, ic, addr_out, add_en, remove_en,
       r_node, op,
   source, addr
   );

`include "filter_constants.v"

parameter IC_WIDTH = PAXI_ADDR_BITS + NR_CORES-1 +
    IC_OP_WIDTH;
parameter EVICT = 2'b11, WRITE = 2'b01, READ = 2'b10, NO_OP
      = 2'b00;

input wire                      clk;
input wire                      reset_n;
input wire                      valid;
```

```

//  ic                          [IC_WIDTH−1:IC_WIDTH−
    PAXI_ADDR_BITS|IC_WIDTH−PAXI_ADDR_BITS−1:IC_OP_WIDTH|
    IC_OP_WIDTH−1:0]
//                              [               addr
                          |                      source
    |         op         ]
input  wire  [IC_WIDTH−1:0]         ic;

input  wire  [PAXI_ADDR_BITS−1:0]          addr;
input  wire  [PAXI_ADDR_BITS−1:0]          addr_out;
input  wire                                add_en;
input  wire                                remove_en;
input  wire  [NR_CORES−2:0]                r_node;

input  wire  [IC_OP_WIDTH−1:0]      op;
input  wire  [NR_CORES−2:0]         source;



    default clocking cb @(posedge clk); endclocking

    // No operation should be started if !valid
    property pr_check_valid_0;
        disable iff(!reset_n)
        (!valid) |=> (!(add_en | remove_en));
    endproperty
    as_check_valid_0 : assert property(pr_check_valid_0);

    // If a valid signal has been set, either remove_en or
       add_en should be set 1 cycle later, if there's an
       operation on the bus
    property  pr_check_valid_1;
        disable iff(!reset_n)
        (valid & (op != NO_OP)) |=> (add_en | remove_en);
    endproperty
    as_check_valid_1 : assert property(pr_check_valid_1);

    // remove_en and add_en should not be high
       simultaneously
    property pr_onehot_add_remove;
        disable iff(!reset_n)
        $onehot0({remove_en,add_en});
    endproperty
    as_onehot_add_remove : assert property(
        pr_onehot_add_remove);

    //r_node = source
    property pr_source_r_node;
        disable iff(!reset_n)
        (r_node == source);
    endproperty
    as_source_r_node : assert property(pr_source_r_node);

    // If valid addr_out should be addr
    property pr_addr_addr_out;
        disable iff(!reset_n)
        (valid) |=>($past(addr) == addr_out);
    endproperty
    as_addr_addr_out : assert property(pr_addr_addr_out);

    //Evict should lead to remove_en<=1;
    property pr_evict;
        disable iff(!reset_n)
        (valid & (op==EVICT)) |=> (remove_en);
    endproperty
    as_evict : assert property(pr_evict);

    // Write or read should lead to add_en <=1;
    property pr_write_read;
        disable iff(!reset_n)
        (valid & ((op == READ) | (op == WRITE) )) |=> (
            add_en);
    endproperty
    as_write_read : assert property(pr_write_read);

    //reset
    property pr_reset_n;
        (reset_n) |-> (!(remove_en & add_en));
    endproperty
    as_reset_n : assert property(pr_reset_n);
endmodule
```

## C.3 Update Address Properties

SystemVerilog C.3: Update address Properties.

```
//
    ***********************************************************
```

```verilog
// This file contains properties and such for formal verification
// of update_addr module in source filter
//
//
//
//
//
//

module src_csr_update_addr_prop(/*AUTOARG*/
  // Inputs
  clk, reset_n, remove_en, add_en, addr_in, r_node,
  csr_i_b, csr_o_b, baseo, masko, cnto, basei,
  maski, cnti, basei_reg, maski_reg, cnti_reg, base_sel, base, mask,
  mask_sel, mask_bits, cnt_add, cnt_rm, cnt, cnt_sel, p1_enable,
  p2_enable,p3_enable, p0_enable, op_1, op_2, r_node_1, r_node_2, addr_in_1,
  addr_in_2, snatch_data_en, snatch_data_1, snatch_data_2,
  snatch_data_from_bus_en, snatch_data_from_bus_1, bits_to_compare,
  ce0, ce1, we1, a0, a1, a1_1, a1_2, a1_3, di1, csr_i, csr_o
  );

`include "filter_constants.v"

parameter NAME = "src_csr_add_addr";
//Parameter Cores can be 5,13,11,7 for the different combinations of remote nodes
parameter CORES=5;

input wire                         clk;
input wire                         reset_n;
input wire                         remove_en;
input wire                         add_en;
input wire [PAXI_ADDR_BITS-1:0]    addr_in;
input wire [NR_CORES-2:0]          r_node;

//Output from register files:
//                        [BASE:MASK:COUNT]
input wire [(NR_CORES-1)*CSR_REG_WIDTH-1:0] csr_i_b;
```

```
//csr_o:                    [CSR_O_WIDTH:2*CSR_INDEX_BITS+3|2*CSR_INDEX_BITS+2|2*CSR_INDEX_BITS+1|2*CSR_INDEX_BITS|2*CSR_INDEX_BITS-1:CSR_INDEX_BITS|CSR_INDEX_BITS-1:0]
//Input to register file:[        di1                  |           ce0       |           ce1        |         we1        |           a0                           |        a1              ]
input wire [(NR_CORES-1)*CSR_O_WIDTH-1:0]
    csr_o_b;



//Ports

//Reg and wires used to pack bus:
input wire [CSR_BM_BITS-1:0]          baseo[NR_CORES-2:0];
input wire [CSR_BM_BITS-1:0]          masko[NR_CORES-2:0];
input wire [CSR_CNT_BITS-1:0]         cnto[NR_CORES-2:0];

input wire [CSR_BM_BITS-1:0]         basei[NR_CORES-2:0];
input wire [CSR_BM_BITS-1:0]         maski[NR_CORES-2:0];
input wire [CSR_CNT_BITS-1:0]        cnti[NR_CORES-2:0];

//Input data registers
input wire [CSR_BM_BITS-1:0]         basei_reg[NR_CORES-2:0];
input wire [CSR_BM_BITS-1:0]         maski_reg[NR_CORES-2:0];
input wire [CSR_CNT_BITS-1:0]        cnti_reg[NR_CORES-2:0];

//Combinational wires
input wire [CSR_BM_BITS-1:0]         base_sel[NR_CORES-2:0];
input wire [CSR_BM_BITS-1:0]         base[NR_CORES-2:0];
input wire [CSR_BM_BITS-1:0]          mask[NR_CORES-2:0];
input wire [CSR_BM_BITS-1:0]         mask_sel[NR_CORES-2:0];
input wire [CSR_BM_BITS-1:0]         mask_bits[NR_CORES-2:0];
input wire [CSR_CNT_BITS-1:0]        cnt_add[NR_CORES-2:0];
input wire [CSR_CNT_BITS-1:0]        cnt_rm[NR_CORES-2:0];
input wire [CSR_CNT_BITS-1:0]        cnt[NR_CORES-2:0];
input wire [CSR_CNT_BITS-1:0]        cnt_sel[NR_CORES-2:0];
//
//Internal registers, used in pipeline, etc.
input wire                            p1_enable;
input wire                            p2_enable;
input wire                            p3_enable;
input wire                            p0_enable;
input wire                            op_1;
input wire                            op_2;
input wire [NR_CORES-2:0]          r_node_1;
input wire [NR_CORES-2:0]          r_node_2;
input wire [PAXI_ADDR_BITS-1:0]      addr_in_1;
input wire [PAXI_ADDR_BITS-1:0]      addr_in_2;
input wire                         snatch_data_en;
input wire                         snatch_data_1;
input wire                         snatch_data_2;
input wire                         snatch_data_from_bus_en;
input wire                         snatch_data_from_bus_1;
input wire [CSR_BM_BITS-1:0]      bits_to_compare[NR_CORES-2:0];

input wire [NR_CORES-2:0]                        ce0;
input wire [NR_CORES-2:0]                        ce1;
input wire [NR_CORES-2:0]                        we1;
input wire [CSR_INDEX_BITS-1:0]                  a0;
input wire [CSR_INDEX_BITS-1:0]                  a1;
input wire [CSR_INDEX_BITS-1:0]                    a1_1;
input wire [CSR_INDEX_BITS-1:0]                    a1_2;
input wire [CSR_INDEX_BITS-1:0]                    a1_3;
input wire [CSR_REG_WIDTH-1:0]                  di1[NR_CORES-2:0];

input wire [CSR_REG_WIDTH-1:0]      csr_i[NR_CORES-2:0];
input wire [CSR_O_WIDTH-1:0]        csr_o[NR_CORES-2:0];



    default clocking cb @(posedge clk); endclocking

    // Three cycles after an operation has started it should end.
    property pr_finish_test;
        disable iff(!reset_n)
        (p0_enable & (r_node[0] > 0)) |=> ##2 (ce1 > 0);
    endproperty
    as_finish_test : assert property(pr_finish_test);

    //Only one of add_en, remove_en should be one.
    property pr_one_hot_add_rm;
        disable iff(!reset_n)
        $onehot0({add_en,remove_en});
    endproperty
    am_one_hot_add_rm : assume property(pr_one_hot_add_rm);

    // If ce0, the read lines to the CSR should not be x
    property pr_x_read_line;
        disable iff(!reset_n)
        ce0 |-> (!$isunknown(a0));
    endproperty
    as_x_read_line : assert property(pr_x_read_line);

    // If p0_enable and p1, and a1 == a1_1, snatch_data==1;
    property pr_snatch_data_en;
        disable iff(!reset_n)
        (p0_enable & p1_enable & (a1 == a1_1)) |->
            snatch_data_en;
    endproperty
    as_snatch_data_en : assert property(pr_snatch_data_en);
//Stage 1 assertions:
    property pr_p1_enable;
        disable iff(!reset_n)
        p0_enable |=> p1_enable;
    endproperty
    as_p1_enable : assert property(pr_p1_enable);

    // If the p1 and snatch_data is enabled the data should
        be snatched from the output
    // registers.
    property pr_snatch_data_bus_base;
        disable iff(!reset_n)
        (p1_enable & snatch_data_from_bus_1) |=> (basei_reg
            == $past(baseo));
    endproperty
    as_snatch_data_bus_base : assert property(
        pr_snatch_data_bus_base);

    //If stage 1 is enabled and the snatch is not set the
        data should come
    //from the register file.
    property pr_not_snatch_data_bus_base;
        disable iff(!reset_n)
        (p1_enable & !snatch_data_from_bus_1) |=> (
            basei_reg == $past(basei));
    endproperty
    as_not_snatch_data_bus_base : assert property(
        pr_not_snatch_data_bus_base);


// Stage 2 assertions
```

```
    property pr_p2_enable;
        disable iff(!reset_n)
        p1_enable |=> p2_enable;
    endproperty
    as_p2_enable : assert property(pr_p2_enable);

    //If data_snatch is enabled, the selected data should
       be taken from output registers
    property pr_snatch_data;
        disable iff(!reset_n)
        (snatch_data_en) |=> ##1 (base_sel == baseo);
    endproperty
    as_snatch_data : assert property(pr_snatch_data);

    //If data_snatch is not enabled, the selected data
       should be taken from input registers
    property pr_not_snatch_data;
        disable iff(!reset_n)
        (!snatch_data_en) |=> ##1 (base_sel == basei_reg);
    endproperty
    as_not_snatch_data : assert property(pr_not_snatch_data
       );

// Stage 3 assertions
    property pr_p3_enable;
        disable iff(!reset_n)
        p2_enable |=> p3_enable;
    endproperty
    as_p3_enable : assert property(pr_p3_enable);
endmodule
```

## C.4 Check Address Properties

SystemVerilog C.4: Properties used to verify check_addr module.

```
//
   ****************************************************

// This file contains properties and such for formal
   verification
// of update_addr module in source filter
//
//
//
//
//
```

```
//

module src_csr_check_addr_prop(/*AUTOARG*/
  // Inputs
  clk, reset_n, check_en, addr_in, finished, addr_out,
     hits_out,
  csr_i_b_check, a2, ce2, basei_reg, maski_reg, cnti_reg,
     p0_enable,
  p1_enable, p2_enable, addr_in_1, addr_in_2,
     bits_to_compare,
  addr_to_compare, csr_i
  );

`include "filter_constants.v"

input wire                              clk;
input wire                              reset_n;
input wire                              check_en;
input wire [PAXI_ADDR_BITS-1:0]         addr_in;
// Inputs from the update_logic, to enable fetching data
   before it is saved
// into the CSR

input wire                              finished;
input wire [PAXI_ADDR_BITS-1:0]         addr_out;
input wire [NR_CORES-2:0]               hits_out;

//Output from register files:
//                        [BASE:MASK:COUNT]
input wire [(NR_CORES-1)*CSR_REG_WIDTH-1:0]
   csr_i_b_check;
input wire [CSR_INDEX_BITS-1:0]         a2;
input wire                              ce2;

//Input wires
wire [CSR_BM_BITS-1:0]          basei[NR_CORES-2:0];
wire [CSR_BM_BITS-1:0]          maski[NR_CORES-2:0];
wire [CSR_CNT_BITS-1:0]         cnti[NR_CORES-2:0];

//Input data registers
input wire [CSR_BM_BITS-1:0]          basei_reg[NR_CORES-2:0];
input wire [CSR_BM_BITS-1:0]          maski_reg[NR_CORES-2:0];
input wire [CSR_CNT_BITS-1:0]         cnti_reg[NR_CORES-2:0];

//Internal registers, used in pipeline, etc.
```

```
input wire                              p0_enable;
input wire                                p1_enable;
input wire                                p2_enable;

input wire [PAXI_ADDR_BITS-1:0]           addr_in_1;
input wire [PAXI_ADDR_BITS-1:0]           addr_in_2;


input wire [CSR_BM_BITS-1:0]          bits_to_compare[NR_CORES
    -2:0];
input wire [CSR_BM_BITS-1:0]          addr_to_compare[NR_CORES
    -2:0];


input wire [CSR_REG_WIDTH-1:0]            csr_i[NR_CORES-2:0];



    default clocking cb @(posedge clk); endclocking

    // Stage 0 assertions
    // Check that module finishes
    property pr_check_finish;
        disable iff(!reset_n)
        check_en |=> ##1 finished;
    endproperty
    as_check_finish : assert property(pr_check_finish);

    // Stage 1 assertions
    // P1 should be set if p0
    property pr_p1_enable;
        disable iff(!reset_n)
        p0_enable |=> p1_enable;
    endproperty
    as_check_p1_enable : assert property(pr_p1_enable);
// Stage 2:

    // P2 should be set if p1
    property pr_p2_enable;
        disable iff(!reset_n)
        p1_enable |=> p2_enable;
    endproperty
    as_check_p2_enable : assert property(pr_p2_enable);

    // Check that for every CSR if its a hit the input wire
        gets set
    property pr_check_comb_logic(i);
        disable iff(!reset_n)
            ( ((basei_reg[i] & maski_reg[i]) ==
                addr_to_compare[i]) & (cnti_reg[i] > 0) )
                |-> hits_out[i];
    endproperty
    generate
        for (genvar i = 0; i < NR_CORES-1; i = i + 1)begin:
            logic_ass
            as_check_com_logic : assert property(
                pr_check_comb_logic(i));
        end
    endgenerate
endmodule
```

# References

Accellera (n.d.), Universal verification methodology (uvm) 1.1 user's guide, Technical report, Accellera.
**URL:** *http://www.accellera.org/downloads/standards/uvm/*

Daniel J. Sorin, Mark D. Hill, D. A. W. (2011), *A Primer on Memory Consistency and Cache Coherence*, Morgan & Claypool Publishers.

Hennessy, J. L. & Patterson, D. A. (2006), *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann.

Moshovos, A., Memik, G., Falsafi, B. & Choudhary, A. (2001), Jetty: filtering snoops for reduced energy consumption in smp servers, *in* 'High-Performance Computer Architecture, 2001. HPCA. The Seventh International Symposium on', pp. 85 –96.

Nilsson, J., Landin, A. & Stenstrom, P. (2003), The coherence predictor cache: a resource-efficient and accurate coherence prediction infrastructure, *in* 'Parallel and Distributed Processing Symposium, 2003. Proceedings. International', pp. 10 –17.

Raman, R. (2012), Cache coherence protocol verifications, Semester project, NTNU.

Ranganathan, Aanjhan, B. A. G. K. T. B. P. I. P. & Charbon, E. (2012), 'Counting stream registers: an efficient and effective snoop filter architecture'.

Salapura, V., Blumrich, M. & Gara, A. (2007), Improving the accuracy of snoop filtering using stream registers, *in* 'Proceedings of the 2007 workshop on MEmory performance: DEaling with Applications, systems and architecture', MEDEA '07, ACM, New York, NY, USA, pp. 25–32.
**URL:** *http://doi.acm.org/10.1145/1327171.1327174*

Ulfsnes, R. (2012), A survey of low power design techniques for cache coherency in multiprocessor memory systems, Semester project, NTNU.