Johanna Tuominen | Tero Säntti | Juha Plosila

Feasibility Report on Asynchronous Synthesis

Turku Centre *for* Computer Science

TUCS Technical Report
No 765, April 2006

Feasibility Report on Asynchronous Synthesis

Johanna Tuominen
Turku Centre for Computer Science (TUCS)
Lemminkäisenkatu 14 A, 20520 Turku, Finland
joeltu@utu.fi
Tero Säntti
Dept. of Information Techology, University
Lemminkäisenkatu 14-18 A, 20520 Turku, Finland
teansa@utu.fi
Juha Plosila
Dept. of Information Technology, University of Turku
Lemminkäisenkatu 14-18 A, 20520 Turku, Finland
juplos@utu.fi

TUCS Technical Report
No 765, April 2006

Abstract

The asynchronous design approach is an interesting alternative to the commonly used synchronous approach because of its several benefits. Self-timed circuit have potential for low-power and low-noise design. Moreover, the modularity and the composability of asynchronous systems are favorable properties. This is partly due to the chips getting larger and denser, resulting in serious difficulties in the clock tree design. One of disadvantages has been the lack of commercial computer aided design (CAD) tools. This paper presents synthesis flow targeted for self-timed VLSI circuits provided by Handshake Solutions. The performance of the synthesis tool is compared with its synchronous counterpart in terms of area and speed. We have chosen to use cache controllers as case study.

Keywords: Asynchronous, Synthesis, Haste, Power Consumption

TUCS Laboratory
Communication Systems

1 Introduction

Self-timed circuit have potential advantages making them an interesting option in various application areas. With the advent of wireless and mobile high performance computing platforms, and the limited operational lifetime of batteries, low-power designs are required. Moreover, in many designs the synchronous operation of the circuit is the main source of on-chip noise. The self-timed design techniques have potential for low-power [11] and low-noise design [7]. As the VLSI circuits get larger, maintaining the synchrony gets more and more difficult. Therefore, the modularity and composability of asynchronous systems is a definite advantage. For instance, the integration of an asynchronous module to an existing system is easy, as no clock limitations need to be considered. One of the disadvantages has been the lack of computer aided design (CAD) tools, which support the design flow of the self-timed circuits. So far the design process have been carried out using existing tools of which are mostly targeted to synchronous system design. This includes a lot of full custom work, which is very time consuming and therefore expensive. All together the use of, for instance, VHDL in asynchronous system design is somewhat problematic, and in some cases even impossible.

In this study, we analyze the functionality of the design flow for self-timed circuit provided by Handshake Solutions [5]. The design entry is Haste, a programming language targeted for this design flow. The purpose is to compare the performance of Haste designs with the corresponding synchronous implementation. The synchronous designs are described using VHDL. The case study for the comparison is cache controller.

The pioneers of computing predicted in [3], that programmers would want unlimited amount of fast memory. Since fast memory is expensive, the most economical solution is to apply memory hierarchy where each level is faster and more expensive per byte than the next lower level [9]. Cache is denoted as the first level of the memory hierarchy once the address leaves the CPU. When the CPU finds a requested data item in the cache, it is called $cache\ hit$. When the CPU does not find the data item in the cache, a $cache\ miss$ occurs. Then the data is retrieved from lower level of the memory hierarchy (main memory), and placed into the cache.

The performance of the self-timed synthesis tool is analyzed and compared with its synchronous counterpart. As a case study we apply two cache constructs: instruction cache and data cache. These caches are meant to be used in the REAL-Java Project, which aims to design and implement a low-power Java co-processor. Even though some design flows are made according to the Java processors specification, the most of them are well suited for modern Network-on-Chip applications as well. Both caches are implemented using Haste and VHDL, after which they are synthesized, and finally the results are compared.

Overview of the paper We proceed as follows. In section 2 we give an

overview of Handshake Technology, and shortly describe the design flow applied in the asynchronous cache design. Section 3 compares general properties of the cache architectures, and then presents the chosen one. Analysis and comparation between synchronous and asynchronous design flows are presented in Section 4. The results are presented in Section 5. Finally, in section 6 we draw some conclusions and describe the future work related to this paper.

2 Overview of the Applied Haste Design Flow

The purpose of this section is to give a short overview of the VLSI programming language $Haste$, which is the input format of the Handshake Technology design flow. This design flow provides a tool set to design and synthesize self-timed asynchronous VLSI-circuits [5]. More information on handshake circuits can be found in [11]. The syntax of Haste is strongly influenced by Hoare's CSP [6] and Dijkstra's guarded commands [4]. In year 1998, by the request of designers, non-handshake channels were introduced into Haste language. Non-handshake signals are used to design interfaces (for instance to synchronous domains). The language was further refined and extended in 2004. However, the basic idea behind Haste has not been affected. That is, to provide means to VLSI designer to design self-timed circuit at abstract level without bothering about the details of self-timed circuit operation [10].

The main focus in this paper is to analyze the functionality of the self-timed design flow using tools provided by the Handshake Solutions. The full Handshake Technology design flow is presented in [2], and the parts of the flow applied in this work are shown in Figure 1.

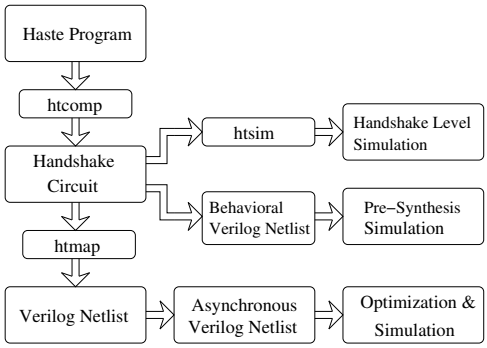


Figure 1: Design flow.

The design entry is Haste, a programming language specifically defined for the design of VLSI circuits. The Haste program is then compiled using the compiler $htcomp$, which translates the source code into handshake circuit. Hence, the connections between components are called handshake channels. At this point it is possible to verify the functionality of the Haste source by using the $htsim$,

which generates abstract models of the handshake components, allowing simulations at the handshake level. Simulations can be conducted at behavioral level also. This is done by compiling the Haste program to behavioral Verilog and using standard simulation software, in our case the Mentor Modelsim. The benefit of this approach is that we can use the same testbench for pre- and post synthesis simulations. The $htsim$ simulator requires separate testbench, which is not applicable with standard Verilog simulators. The $htmap$ tool is used to optimize the control and communication of each handshake channel and perform technology mapping from Handshake circuit to asynchronous Verilog netlist. In this paper, we have chosen to use UMC 0.13 μm technology. Next scripts for optimizing the logic and recalibrating the delay elements to match the new optimized logic are generated using the $htlog$ and $htpost$ tools. The Verilog netlist is subsequently optimized using Cadence PKS shell synthesis tool with the scripts generated in the previous step. Finally, the correctness of the design is validated using Mentor Modelsim simulator.

3 Cache Architectures

3.1 Design space limitations

The caches designed here are meant to be used in the REALJava project, which aims to design and implement a low-power Java co-processor. Asynchronous techniques are chosen to achieve good performance with low power consumption and very easy integration with existing systems, as no clock limitations need to be considered. Asynchronous self-timed circuit technology [12], where timing is based on local handshakes between circuit blocks instead of a global clock signal, provides a promising platform for obtaining highly modular low-power and low-noise implementation. Even though the choices are made with the REALJava in mind, most of them are well suited for generic Network-on-Chip (NoC) applications as well. The common characteristics would include high level of parallelism, shared memories and moderate functional unit size (typically 32 bit processing units).

3.2 General properties of the cache implementation

Two essential questions arise when we are sketching the cache design: How do we know if data item is in the cache? And if it is, how do we find it? If each block has only one place where it can appear in the cache, the cache is said to be $direct\ mapped$. The mapping is usually defined by: $(Blockaddress) mod (Numberof\ blocksinthecache)$ [9]. Another extreme is when data item is placed on any location in the cache, called $fully\ associative$ because a block in memory can be associated with any entry in the cache, and thus to find given memory block from

fully associative cache, all entries are searched. In between these two extremes are $set\ associative$ caches, which have a fixed number of locations (sets) where each block can be placed. For instance, a n-way set associative cache consist of a number of sets, each of which consist of n blocks. Thus, one can consider the set associative placement as combination of the direct mapped and fully associative placements: a block is directly mapped into a set, and then all blocks in that set are searched for a match. In general, the advantage of increasing the degree of associativity is that it usually decreases the miss rate [9]. However, this comes with the expense of increased hit time. By adopting the two-way set associativity the decrease in miss rate is most significant [9], and after that there is no significant improvement. Thus, we have chosen the two-way set associative placement for our implementation. The structure of one cache set is shown in Figure 2.

$INDEX_i$	TAG0	DATA0
	TAG1	DATA1

Figure 2: Structure of a cache set.

In our implementation the number of these caches sets is 16, and therefore the number of cache blocks is 32. The cache address is 32 bits in length and consists of index and tag parts. The data part of the cache line is 32 bits as well. The caches are designed so that word lengths and cache sizes can be easily changed. The structure of the cache line with bit positions is illustrated in Figure 3. Observe that, the value of the index is the same for blocks that are located in the same set.

64 ... 43	42 ... 32	31 ... 0
TAG	INDEX	DATA

Figure 3: Cache line (showing bit positions).

The read operation from two-way set associative cache proceeds as follows: First, we search a match for the index part of the received address from the cache. If found, the two tags in the selected set are compared with the tag of the address. In case of match, data from that location is returned. This situation is denoted as $cache\ hit$. If the requested data is not located in the cache, the situation is called a $cache\ miss$. In this case the data is then requested from the lower level of the memory hierarchy, namely the main memory. The received data from the main memory is written into the cache and given to the CPU.

The cache write operation is needed in two cases: the requested data is fetched from the main memory, or the CPU writes data into the memory. However, the cache write operation is similar in both cases. Since we have adopted the set associative placement, we have to choose which block under which set is replaced. The $Least\ Recently\ Used$ (LRU) method relies on a corollary of locality: Recently used blocks are likely to be used again, and therefore a good candidate for disposal is the one that has been unused for the longest time [9]. However, the

LRU can be complicated to calculate on some occasions, and therefore we will adopt the $first\ in,\ first\ out$ (FIFO) method, which is an approximation of the LRU. The FIFO method approximates the LRU method by determining the oldest set rather than the least recently used. Moreover, the difference in accuracy in terms of cache misses is small, and because the expenses of LRU increases as the block size increases, we will adopt the $FIFO$ method. Thus, data is written into the oldest cache set, whereas the block to be replaced is chosen randomly. If data is written to the cache from the CPU side then the data is written in both to the cache and to the main memory immediately. This approach is known as $write\ through\ policy$, and we have adopted it because it is well suited for NoC applications and other environments with multithreading and/or parallelism in the presence of shared memories.

The REALJava co-processor, specified in [13], has two separate caches, namely instruction cache and data cache. The instruction cache is read-only, where as the data cache can be written or read. The interface of the data cache is shown in Figure 4. The instruction cache has similar interface except the write operation from the CPU's side is eliminated. Therefore, for the rest of this section, we will illustrate the architectural aspects using data cache.

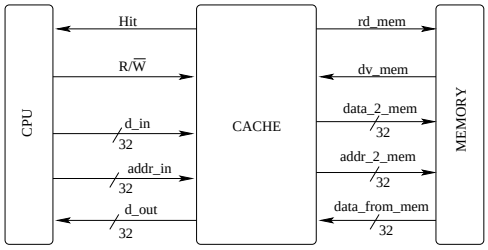


Figure 4: Simplified cache interface.

The interface, shown in Figure 4, is applied to both asynchronous and synchronous cache implementations, in the CPU to cache interface there are five signals. Read not write ($r/\overline{w}$) is used to indicate whether the CPU wants to read or write to the memory. There are two data channels, one for reading (d_{out}) and one for writing data (d_{in}), and one address channel ($addr_{in}$). The hit is set $'1'$ by the cache in case of a cache hit, otherwise it remains $'0'$. The data out and hit signals are both used to indicate to the CPU that the cache is ready to accept new requests. The cache to memory interface also contains five signals. Data valid to memory (dv_{mem}) and read memory (rd_{mem}) are control signals, which are used to issue read access from the cache side and write access from the memory side, respectively. Data and address channels are similar to the CPU interface. All of the address and data channels are 32 bits wide.

3.3 Asynchronous cache design

The timing diagram of the asynchronous cache read operation is shown in Figure 5. The read operation consist of four sequential handshakes: At first the CPU request read access ($HS1$) to the cache. When the access is granted, the address is loaded to the cache. If the result of the cache search is a $miss$, the read request is forwarded to the main memory ($HS2$). Then, the cache waits until there is valid data from the main memory. The main memory has the initiative on the third handshake ($HS3$), which signals that the memory read is completed. Once the cache has received the data from the memory it performs the cache write operation and outputs the requested data to the CPU ($HS4$). After the fourth handshake is completed the cache is ready to accept new requests.

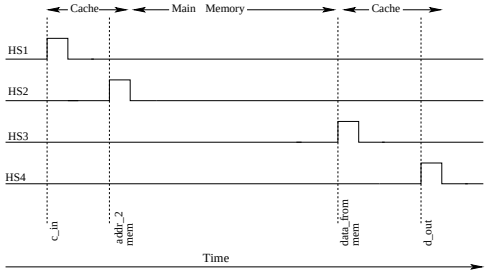


Figure 5: Timing diagram for asynchronous data cache.

If the result of the read operation is hit, the handshakes two and three are skipped, and the result is directly returned to the CPU $(HS4)$, and a new operation can be requested. Thus, the fourth handshake serves as an cache valid signal in read operation, and therefore when it is completed the CPU knows that the cache is ready accept new operation request.

The write operation consists of two handshakes. In the first phase the CPU requests write access to the cache $(HS1)$, and when the access is granted, the data and address are transferred to the cache. In the second phase the cache updates the data in the cache memory and writes it to the main memory $(HS2)$. After the second handshake the cache is ready to accept new commands but the main memory may still be processing the write command. Therefore the next command may be stalled until the previous write command is completed. This happens only if the next command requires access to the main memory, reads from already cached memory locations can be completed. The haste code for data cache and the test environment can be found in Appendix A.

3.4 Synchronous cache design

In order to compare the performances of the caches designed using Haste, we implemented synchronous versions of these two caches using VHDL. The interfaces are similar than in Figure 4, and the operation is consistent with the asynchronous

implementation. Thus, the timing diagram for the synchronous cache read operation (read miss assumed) is shown in Figure 6.

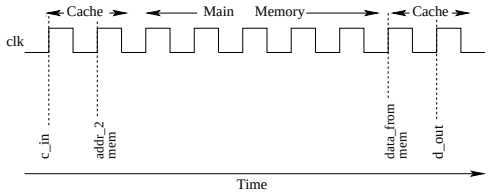


Figure 6: Timing diagram for synchronous data cache.

The address request is read in, and the cache search operation is performed during one clock period. During the second clock period the data is sent to the main memory. Then the cache waits as long as the main memory sets the data valid signal high. Then one clock cycle is used to store the received data in the cache and the fourth clock period is used to output the data to the CPU. Notice the similarity that synchronous cache uses four clock cycles to complete the read operation whereas the asynchronous uses four handshakes. If the read operation results cache hit, then this is indicated to the CPU using hit signal, then only two clock cycles are needed for this operation. The VHDL code for the synchronous data cache and the applied test environment can be found in Appendix B.

4 Implementation Issues

The storage structure of a two way set associative cache was described using $record$ statement [1]. $Record\ types$ are heterogeneous, that is elements may have different types. By adopting the record statement we define one cache set, as shown in Figure 2. Then we can define an array of these records. Now each cell in the array has the properties of the record defined for the cache set. This turned out to be very effective and flexible method to describe a generic cache memory. Moreover, the read and write operations to the cache were easy to implement using a for loop. Overall, the VHDL description follows the simplified timing diagram, shown in Figure 6.

4.1 Haste descriptions

The storage structure for asynchronous caches were implemented using Haste type $tuple$, which has same kind of properties with the record type in VHDL. For instance, the cache line, shown in Figure 3, can be defined to form a tuple type:

$cacheData$ = [[bool22, bool10, bool32]]

and now we can define variables using the tuple type, for instance:

$$cacheLine = cacheData$$

where the size of the tuple is the sum of the sizes of the constituent types [10], in this case it is 64. We are using the $[[]]$ brackets to define the tuple where the contents are indexed MSB first, and therefore in our example tuple the contents are indexed by $[[2, 1, 0]]$. For instance, the cache index can be accessed by following notation: $cacheLine.1$. By adopting the tuple construct, we were able to design the cache memory which resembles the corresponding VHDL implementation. The asynchronous cache implementations are consistent with the timing diagram shown in Figure 5.

4.2 Discussion on design environments

In general it is not fair to compare directly these two languages, because VHDL has been developed for a long time by a large community, whereas the Haste is fairly new language with a smaller group behind it. There are a lot of ready made source code libraries for VHDL. Also VHDL is a bit easier to use (tools are older, resulting in more sophisticated user interfaces etc.). However the design of asynchronous circuits using VHDL is cumbersome, even impossible in some cases. Haste on the other hand is developed for the design of asynchronous circuits. One major difference between these two languages is the level of control the designer has. In Haste the way the code is written heavily effects the resulting synthesized structures. This is due to the fact that Haste is based on the idea of $syntax\ driven$ synthesis.

After the experiences gained during design of the caches for this paper we would like to see future versions of Haste with a few improvements. Currently Haste implements all of the communication channels using four-phase handshaking. An option to choose between 2 and 4 phase handshakes would give the designer more control. Also a language structure similar to the “for - generate” statement in VHDL would provide a flexible way of creating large parameterizable structures. Finally a possibility to define data validity scheme (early vs. late) would simplify integration to surrounding environment.

5 Synthesis Results

5.1 Test environment

All the designs were optimized using Cadence PKS shell synthesis tool with UMC $0.13\mu m$ technology, after which VHDL netlists for synchronous caches and Verilog netlists for asynchronous caches were written. The delays for each cache circuit were written in the standard delay format (SDF). The post synthesis simulation is done using the Modelsim simulator for all designs. The design under test (DUT) is surrounded by the test environment as shown in Figure 7.

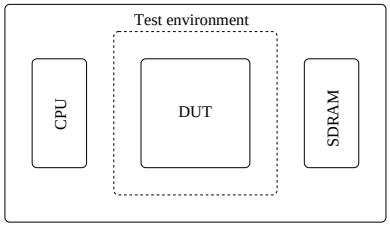


Figure 7: Test environment for the caches.

In all four cases the test environment is generated using VHDL. The technical properties of the SDRAM were chosen from the datasheets of Maxwell Technologies [8]. The adopted memory is 72SD3232 1 Gbit SDRAM, which operates on $100MHz$ clock frequency. The $\overline{CAS}$ latency was chosen to be three, and therefore the memory access time for single read or single write is 90 ns.

5.2 Performance evaluation

The operation of the asynchronous and synchronous caches is cyclic, due to the similar design approach. For instance, compare the timing diagrams for synchronous and asynchronous read operation (miss assumed), shown in Figure 6 and 5, respectively. In both cases the operation is divided either four handshakes (asynchronous) or four operative clock cycles (synchronous). Notice that, the synchronous cache runs on quite slow clock frequency, $f_{clk} = 100MHz$, because of this architecture.

The read access times are shown in Table 1.

Table 1: Read access times.

	Instruction caches		Data caches	
	Sync.	Async.	Sync.	Async.
Hit [ns]	20	28	30	28
Miss [ns]	110	125	120	133

At first, lets consider the synchronous cache designs. The difference in access times between the instruction cache and data cache implementations are $10ns$ for both $read\ hit$ and $read\ miss$ cases. Next, we carry on similar comparison for the access times of the asynchronous caches. Observe that, in case of a $read\ hit$, both the data and instruction cache have equally good performance. Moreover the difference in $read\,miss$ operation is relatively small. Finally, we compare the four cache implementations together. Instruction caches are faster than data caches due to their simpler structure. Furthermore, synchronous designs are slightly faster

than the asynchronous ones, but the difference decreases when the comparison is carried out between the data caches. Especially in the case of a hit in the data cache the asynchronous version is actually a bit faster.

The write access time comparison between the data cache implementations is shown in Table 2. In this case the asynchronous data cache performs cache write faster than the corresponding synchronous one. However, these access times are for the cache write operation, so for instance, if the CPU requests read operation, which results as cache miss, the cache might not get direct read access to the main memory, because the duration of the write operation in main memory is 90 ns.

Table 2: Write access times

	Sync.	Async.
Cache write [ns]	30	20

Conventionally, the asynchronous systems have some amount area overhead comparing with their synchronous counterparts, and therefore one goal in the development of Handshake technology design flow has been to reduce this overhead [10]. The areas of the caches are shown in Table 3.

Table 3: Areas

	Instruction caches		Data caches	
	Sync.	Async.	Sync.	Async.
Total Area [μm^2]	153570.82	95162.69	157004.35	99460.23
Relative Areas	1.6	1	1.6	1

The results are stunning, the area overhead is now problem of synchronous systems. In both cases the synchronous cache has 60% larger area than the corresponding asynchronous one. In the time domain both design styles provide similar performance with only marginal differences. Moreover, asynchronous designs have potential for low-noise and low-power behavior [7]. In many designs noise is caused to a large extent by the synchronous operation of the circuit. As chips grow larger and the energy consumption increases, the portion of the noise induced by the synchronous operation will increase. The clock dictated operation forces a great deal of gates and flip-flops in the chip to change their states at the same moment. This behavior can be demonstrated by measuring the current profile of the circuit. For synchronous designs the current profile is dominated clock induced high peaks. Unfortunately, we were not able to measure the current profiles for

Haste designs due to the technology related problems, and issues related to this new Haste design flow. However, it is fair to assume that the asynchronous design should have better performance in terms of on-chip noise and power consumption, which makes the asynchronous implementation more attractive.

6 Conclusions and Future Work

Considering the speed of operation, the caches achieved comparable performance for both the asynchronous and the synchronous design styles. However the asynchronous style using Haste was significantly smaller in terms of area. The VHDL coded synchronous caches were actually as much as 60% larger. This was a surprising result, since asynchronous designs have been discredited on grounds of area overhead for a long time. Taking in to account the performance in time domain and the area required for the designs, we can say that Haste seems to provide an attractive alternative for logic synthesis. This is accentuated even further by the well known facts that asynchronous systems provide smoother current profiles, resulting in lesser noise, and generally lower power consumption. Also asynchronous subcomponents can be composed to a complete system without any problems in finding an optimal clock frequency for the whole system. This allows components of a given system to be updated without the need to update all the other components as well. NoC system designers are already starting to favor asynchronous communication structures for the network, so it would be only logical to remove the clock from the processing elements also.

We plan to further investigate the possibilities of the Haste tool set. As the next step, regarding these cache designs, we plan to implement an image manipulation algorithm on an FPGA demonstration board and test it with both data caches created for this paper. Haste provides an option to create a synchronized version of the design, so it can be easily programmed to an FPGA. The performance of the synchronized version is of course degraded, but the functionality will be correct. By analyzing the relative performance metrics of the FPGA implementations and the results obtained for this paper we can draw some indicative relations for the performance of a given actual asynchronous design and its synchronized FPGA prototype.

References

[1] J. R. Amstrong and F. Gail Gray, *VHDL Design Representation and Synthesis*, 2nd ed. New Jersey, United States of America: Prentice Hall, 2000.

[2] F. te Beest, M. Verra, A. Peeters, M. de Wit, and E. Woutersen, *Handshake Technology Design Flow Manual*, ver. 4.3, Handshake Solutions, Koninklijke Philips Electronics N.V., October 2005, The Netherlands.

[3] A. W. Burks, H. H. Goldstine, J. von Neumann, *Preliminary discussion of the logical design of an electronic computing instrument*, Report to the U.S. Army Ordinance Department, 1946.

[4] E. W. Dijkstra, *A Discipline of Programming*, Prentice-Hall International, 1976.

[5] Handshake Solutions, *http://www.handshakesolutions.com*.

[6] C. A. R. Hoare, *Communicating Sequential Processes*. Series in Computer Science, Prentice-Hall Int. 1985.

[7] P. Liljeberg, J. Tuominen, S. Tuuna, J. Plosila, and J. Isoaho, *Self-Timed Approach for Noise Reduction in NoC*, In Interconnect-Centric Design for Advanced SoC and NoC, chapter 11, Kluwer Academic Publishers, April 2004.

[8] Maxwell Technologies, *http://www.maxwell.com*

[9] D. A. Patterson and J. L. Hennessy, *Computer Architecture A Quantitative Approach*, 3rd ed. San Fransisco, CA, United States of America: Morgan Kaufmann Publishers, Inc, 2003.

[10] A. Peeters and M. de Wit, *Haste Manual*, ver. 2.9, Handshake Solutions, Koninklijke Philips Electronics N.V., 2005, The Netherlands.

[11] A. Peeters, *Single-Rail Handshake Circuits*, Ph.D Thesis, Eindhoven University of Technology, 1996.

[12] J. Sparsø, and S. Furber, *Principles of Asynchronous Circuit Design - A System Perspective*, Kluwer Academic Publishers, 2001.

[13] T. Säntti, and J.Plosila, *Architecture for an Advanced Java Co-Processor*, In Proc. of International Symposium on Signals, Circuits and Systems, ISSCS 2005, July 2005, Iasi, Romania.

7 Appendix A: Haste code for data cache

7.1 Asynchronous data cache

```
// Asynchronous data cache
// Port              Purpose
// c_in              consist of input data, address, and a read/write bit.
// d_out             requested data out to the CPU
// addr_2_mem        address request to the main memory
// data_from_mem     requested data from memory
// data_2_mem        CPU writes to cache, data is written throug to
//                   the main memory
// Type declarations

int16= type [0..16]

&int22= type[0..22]
&int32= type[0..32]
```

```
&byte= type[0..1023]

&bool10= type [[bool,bool,bool,bool,bool,bool,bool,bool,bool,bool]]

&bool22= type
[[bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,
bool,bool]]

&bool32= type
[[bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,
bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool,bool]]

&data= type [[bool,bool10,bool22,bool32]]
&cache_data= type [[bool22, bool32]]
&addr= type [[bool10,bool22]]

// Global variable declarations

&tmp_c_in: var data narb: ff
&memory_data: var bool32 narb: ff
&tmp_addr: var addr narb: ff
&tmp_data_2_mem: var bool32 narb: ff

// Main procedure, which is divided into two sub procedures: cm and mmu
// The cm procedure handles write, and read operations into the cache,
// and controls the cache memory.
// The mmu procedure is mainly an interface between cache and main memory.

& dcache: main proc(c_in?chan data pas & d_out!chan bool32 &
addr_2_mem!chan addr & data_from_mem?chan bool32 pas \\ & data_2_mem!chan bool32).

begin

&c: chan bool
&d: chan bool
&e: chan bool
|

        cm(c_in, c, d, d_out,e)||mmu(data_from_mem, c, d, addr_2_mem,e, data_2_mem)

end


&cm: proc(c_in?chan data pas & mmu_dv?chan bool & enable_mmu!chan bool & d_out!chan
bool32 & enable_data_2_mem!chan bool).

begin

//variable declarations

&m_indx: var bool10  narb:
&m_tag: var bool22 narb:
&m_data: var bool32 narb:
&tmp_enable_mmu: var bool
&tmp_mmu_dv: var bool
&rd_mode: var bool ff := true
&dec_number: var byte ff:=0
&cache_hit: var bool ff
&fifo_out: var bool32 arb!
&tmp_enable_data: var bool ff
&hit0: var bool ff := false
&hit1: var bool ff := false
&hit2: var bool ff := false
&hit3: var bool ff := false
&hit4: var bool ff := false
&hit5: var bool ff := false
&hit6: var bool ff := false
&hit7: var bool ff := false
&hit8: var bool ff := false
&hit9: var bool ff := false
&hit10: var bool ff := false
&hit11: var bool ff := false
&hit12: var bool ff := false
&hit13: var bool ff := false
&hit14: var bool ff := false
&hit15: var bool ff := false
&cache_contents0: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents1: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents2: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents3: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents4: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents5: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents6: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents7: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents8: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents9: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents10: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents11: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
```

```
&cache_contents12: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents13: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents14: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents15: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents16: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents17: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents18: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents19: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents20: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents21: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents22: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents23: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents24: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents25: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents26: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents27: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents28: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents29: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents30: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_contents31: var cache_data narb: := [[0 fit int22 cast bool22, 0 fit int32 cast bool32]]
&cache_index0: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index1: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index2: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index3: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index4: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index5: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index6: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index7: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index8: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index9: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index10: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index11: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index12: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index13: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index14: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&cache_index15: var bool10 narb! := [[false,false,false,false,false,false,false,false,false,false]]
&i: var int16 ff := 0



|

forever do

c_in?tmp_c_in;
rd_mode:= tmp_c_in.3;

(if (tmp_c_in.2 = cache_index0) * (tmp_c_in.1 =cache_contents0.1) then
    fifo_out:= cache_contents0.0 || hit0 := true
or  (tmp_c_in.2 = cache_index0) * (tmp_c_in.1 = cache_contents1.1) then
    fifo_out:= cache_contents1.0 || hit0 := true
else
hit0:=false
fi)


||

(if (tmp_c_in.2= cache_index1) * (tmp_c_in.1=cache_contents2.1) then
    fifo_out:= cache_contents2.0 || hit1 := true
or  (tmp_c_in.2= cache_index1) * (tmp_c_in.1=cache_contents3.1) then
    fifo_out:= cache_contents3.0 || hit1 := true
else
hit1:=false
fi)

||

(if (tmp_c_in.2= cache_index2) * (tmp_c_in.1=cache_contents4.1) then
    fifo_out:= cache_contents4.0 || hit2 := true
or  (tmp_c_in.2= cache_index2) * (tmp_c_in.1=cache_contents5.1) then
    fifo_out:= cache_contents5.0 || hit2 := true
else
hit2:=false
fi)

||

(if (tmp_c_in.2= cache_index3) * (tmp_c_in.1=cache_contents6.1) then
    fifo_out:= cache_contents6.0 || hit3 := true
or  (tmp_c_in.2= cache_index3) * (tmp_c_in.1=cache_contents7.1) then
    fifo_out:= cache_contents7.0 || hit3 := true
else
hit3:=false
fi)

||
```

```
(if (tmp_c_in.2= cache_index4) * (tmp_c_in.1=cache_contents8.1) then
    fifo_out:= cache_contents8.0 || hit4 := true
or  (tmp_c_in.2= cache_index4) * (tmp_c_in.1=cache_contents9.1) then
    fifo_out:= cache_contents9.0 || hit4 := true
else
hit4:=false
fi)

||

(if (tmp_c_in.2= cache_index5) * (tmp_c_in.1=cache_contents10.1) then
    fifo_out:= cache_contents10.0 || hit5 := true
or  (tmp_c_in.2= cache_index5) * (tmp_c_in.1=cache_contents11.1) then
    fifo_out:= cache_contents11.0 || hit5 := true
else
hit5:=false
fi)

||

(if (tmp_c_in.2= cache_index6) * (tmp_c_in.1=cache_contents12.1) then
    fifo_out:= cache_contents12.0 || hit6 := true
or  (tmp_c_in.2= cache_index6) * (tmp_c_in.1=cache_contents13.1) then
    fifo_out:= cache_contents13.0 || hit6 := true
else
hit6:=false
fi)

||

(if (tmp_c_in.2= cache_index7) * (tmp_c_in.1=cache_contents14.1) then
    fifo_out:= cache_contents14.0 || hit7 := true
or  (tmp_c_in.2= cache_index7) * (tmp_c_in.1=cache_contents15.1) then
    fifo_out:= cache_contents15.0 || hit7 := true
else
hit7:=false
fi)

||

(if (tmp_c_in.2= cache_index8) * (tmp_c_in.1=cache_contents16.1) then
    fifo_out:= cache_contents16.0 || hit8 := true
or  (tmp_c_in.2= cache_index8) * (tmp_c_in.1=cache_contents17.1) then
    fifo_out:= cache_contents17.0 || hit8 := true
else
hit8:=false
fi)

||

(if (tmp_c_in.2= cache_index9) * (tmp_c_in.1=cache_contents18.1) then
    fifo_out:= cache_contents18.0 || hit9 := true
or  (tmp_c_in.2= cache_index9) * (tmp_c_in.1=cache_contents19.1) then
    fifo_out:= cache_contents19.0 || hit9 := true
else
hit9:=false
fi)

||


(if (tmp_c_in.2= cache_index10) * (tmp_c_in.1=cache_contents20.1) then
    fifo_out:= cache_contents20.0 || hit10 := true
or  (tmp_c_in.2= cache_index10) * (tmp_c_in.1=cache_contents21.1) then
    fifo_out:= cache_contents21.0 || hit10 := true
else
hit10:=false
fi)

||

(if (tmp_c_in.2= cache_index11) * (tmp_c_in.1=cache_contents22.1) then
    fifo_out:= cache_contents22.0 || hit11 := true
or  (tmp_c_in.2= cache_index11) * (tmp_c_in.1=cache_contents23.1) then
    fifo_out:= cache_contents23.0 || hit11 := true
else
hit11:=false
fi)

||

(if (tmp_c_in.2= cache_index12) * (tmp_c_in.1=cache_contents24.1) then
    fifo_out:= cache_contents24.0 || hit12 := true
or  (tmp_c_in.2= cache_index12) * (tmp_c_in.1=cache_contents25.1) then
    fifo_out:= cache_contents25.0 || hit12 := true
else
hit12:=false
fi)
```

```
||

(if (tmp_c_in.2= cache_index13) * (tmp_c_in.1=cache_contents26.1) then
    fifo_out:= cache_contents26.0 || hit13 := true
or  (tmp_c_in.2= cache_index13) * (tmp_c_in.1=cache_contents27.1) then
    fifo_out:= cache_contents27.0 || hit13 := true
else
hit13:=false
fi)

||


(if (tmp_c_in.2= cache_index14) * (tmp_c_in.1=cache_contents28.1) then
    fifo_out:= cache_contents28.0 || hit14 := true
or  (tmp_c_in.2= cache_index14) * (tmp_c_in.1=cache_contents29.1) then
    fifo_out:= cache_contents29.0 || hit14 := true
else
hit14:=false
fi)

||


(if (tmp_c_in.2= cache_index15) * (tmp_c_in.1=cache_contents30.1) then
    fifo_out:= cache_contents30.0 || hit15 := true
or  (tmp_c_in.2= cache_index15) * (tmp_c_in.1=cache_contents31.1) then
    fifo_out:= cache_contents31.0 || hit15 := true
else
hit15:=false
fi);

cache_hit := (hit0 + hit1 + hit2 + hit3 + hit4 + hit5 + hit6 + hit7 +
hit8 + hit9 + hit10 + hit11 + hit12 + hit13 + hit14 + hit15);


if cache_hit = true then
d_out!fifo_out
else
(if (rd_mode=true) then

    (tmp_enable_mmu := true ||
     tmp_enable_data := false);
     tmp_addr:=[[tmp_c_in.2, tmp_c_in.1]]
else
    (tmp_enable_mmu := false ||
     tmp_enable_data:= true);
     tmp_data_2_mem:=tmp_c_in.0
fi);

(enable_mmu!tmp_enable_mmu ||
enable_data_2_mem!tmp_enable_data);


mmu_dv?tmp_mmu_dv;

(if (tmp_mmu_dv = true) then

m_indx:=tmp_c_in.2;
m_tag:=tmp_c_in.1;
m_data:=memory_data;
d_out!m_data

or (tmp_mmu_dv = false) then
m_indx:=tmp_c_in.2;
m_tag:=tmp_c_in.1;
m_data:=tmp_c_in.0
else
skip
fi);


(if i < 15 then i:=i+1
 or i = 15 then i:=0
 else
   skip
 fi) || (dec_number:= m_indx cast byte);

(if i=0 then

(cache_index0 := m_indx) ||
(if dec_number <= 510 then (cache_contents0 := [[m_tag, m_data]])
else (cache_contents1 := [[m_tag, m_data]])
fi)
else
skip
fi)
```

```
||

(if i=1 then
(cache_index1 := m_indx) ||
(if dec_number <= 510 then (cache_contents2 := [[m_tag, m_data]])
else (cache_contents3 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=2 then

(cache_index2 := m_indx) ||
(if dec_number <= 510 then (cache_contents4 := [[m_tag, m_data]])
else (cache_contents5 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=3 then
(cache_index3 := m_indx) ||
(if dec_number <= 510 then (cache_contents6 := [[m_tag, m_data]])
else (cache_contents7 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=4 then

(cache_index4 := m_indx) ||
(if dec_number <= 510 then (cache_contents8 := [[m_tag, m_data]])
else (cache_contents9 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=5 then

(cache_index5 := m_indx) ||
(if dec_number <= 510 then (cache_contents10 := [[m_tag, m_data]])
else (cache_contents11 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=6 then

(cache_index6 := m_indx) ||
(if dec_number <= 510 then (cache_contents12 := [[m_tag, m_data]])
else (cache_contents13 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=7 then
(cache_index7 := m_indx) ||
(if dec_number <= 510 then (cache_contents14 := [[m_tag, m_data]])
else (cache_contents15 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=8 then (cache_index8 := m_indx) ||
(if dec_number <= 510 then (cache_contents16 := [[m_tag, m_data]])
else (cache_contents17 := [[m_tag, m_data]])
fi)
```

```
else
skip
fi)

||

(if i=9 then
(cache_index9 := m_indx)||
(if dec_number <= 510 then (cache_contents18 := [[m_tag, m_data]])
else (cache_contents19 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=10 then
(cache_index10 := m_indx) ||
(if dec_number <= 510 then (cache_contents20 := [[m_tag, m_data]])
else (cache_contents21 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=11 then
(cache_index11 := m_indx) ||
(if dec_number <= 510 then (cache_contents22 := [[m_tag, m_data]])
else (cache_contents23 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=12 then
(cache_index12 := m_indx) ||
(if dec_number <= 510 then (cache_contents24 := [[m_tag, m_data]])
else (cache_contents25 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=13 then
(cache_index13 := m_indx) ||
(if dec_number <= 510 then (cache_contents26 := [[m_tag, m_data]])
else (cache_contents27 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=14 then (cache_index14 := m_indx) ||
(if dec_number <= 510 then (cache_contents28 := [[m_tag, m_data]])
else (cache_contents29 := [[m_tag, m_data]])
fi)
else
skip
fi)

||

(if i=15 then (cache_index15 := m_indx)||
(if dec_number <= 510 then (cache_contents30 := [[m_tag, m_data]])
else (cache_contents31 := [[m_tag, m_data]])
fi)
else
skip
fi)
fi
od
end



&mmu: proc(data_from_mem?chan bool32 pas & mmu_dv!chan bool &
enable_mmu?chan bool & addr_2_mem!chan addr & enable_data_2_mem?chan bool & data_2_mem!chan bool32).
```

```
begin

&tmp_enable: var bool
&tmp_dv: var bool
&tmp_data: var bool
|

forever do

(enable_mmu?tmp_enable ||
enable_data_2_mem?tmp_data);

(if (tmp_enable =true) then
addr_2_mem!tmp_addr;
data_from_mem?memory_data;
tmp_dv:=true
else
tmp_dv:=false
fi)

||

(if (tmp_data = true) then
data_2_mem!tmp_data_2_mem
else
skip
fi);

mmu_dv!tmp_dv
od
end
```

7.2 Test environment for asynchronous data cache

```
-------------------------------------------------------------------------------
-- Title      : asynchronous dcache
-- Project    :
-------------------------------------------------------------------------------
-- File       : t_dcache.vhd
-- Author     : Johanna Tuominen  <joeltu@utu.fi>
-- Company    :
-- Last update: 2006/03/30
-- Platform   : VHDL'93
-------------------------------------------------------------------------------
-- Description: testbench for asynchronous data cache
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2006/02/24  1.0      joeltu  Created
-------------------------------------------------------------------------------
--  channel           purpose
--  z_r               activate request
--  data_from_mem     incoming channel data
--  data_from_mem_a   incoming handshake channel acknowledge
--  data_from_mem_r   incoming handshake channel request
--  addr_2_mem        outgoing handshake channel data
--  addr_2_mem_r      outgoing handshake channel request
--  addr_2_mem_a      incoming handshake channel acknowledge
--  d_out             outgoing handshake channel data
--  d_out_a           incoming handshake channel acknowledge
--  d_out_r           outgoing handshake channel request
--  data_2_mem        outgoing handshake channel data
--  data_2_mem_r      outgoing handshake channel request
--  data_2_mem_a      incoming handshake channel acknowledge
--  c_in              incoming channel data
--  c_in_r            incoming handshake channel request
--  c_in_a            incoming handshake channel acknowledge
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio.all;
use work.all;

entity t_dcache is

end t_dcache;

architecture test of t_dcache is

  component dcache
```

```
    port (
      z_r             : in  std_logic;
      data_from_mem   : in  std_logic_vector(31 downto 0);
      data_from_mem_a : out std_logic;
      data_from_mem_r : in  std_logic;
      addr_2_mem      : out std_logic_vector(31 downto 0);
      addr_2_mem_r    : out std_logic;
      addr_2_mem_a    : in  std_logic;
      d_out           : out std_logic_vector(31 downto 0);
      d_out_a         : in  std_logic;
      d_out_r         : out std_logic;
      data_2_mem      : out std_logic_vector(31 downto 0);
      data_2_mem_r    : out std_logic;
      data_2_mem_a    : in  std_logic;
      c_in            : in  std_logic_vector(64 downto 0);
      c_in_r          : in  std_logic;
      c_in_a          : out std_logic);
  end component;

  for all : dcache
    use entity work.dcache;

  signal z_r, data_from_mem_a, data_from_mem_r, addr_2_mem_r,
         addr_2_mem_a, d_out_a, d_out_r, data_2_mem_a, data_2_mem_r,
         c_in_a, c_in_r : std_logic;
  signal addr_2_mem, d_out : std_logic_vector(31 downto 0);
  signal data_from_mem, data_2_mem: std_logic_vector(31 downto 0);
  signal c_in  : std_logic_vector(64 downto 0);


begin

  i_dcache: dcache
    port map (
      z_r             => z_r,
      data_from_mem   => data_from_mem,
      data_from_mem_a => data_from_mem_a,
      data_from_mem_r => data_from_mem_r,
      addr_2_mem      => addr_2_mem,
      addr_2_mem_r    => addr_2_mem_r,
      addr_2_mem_a    => addr_2_mem_a,
      d_out           => d_out,
      d_out_a         => d_out_a,
      d_out_r         => d_out_r,
      data_2_mem      => data_2_mem,
      data_2_mem_r    => data_2_mem_r,
      data_2_mem_a    => data_2_mem_a,
      c_in            => c_in,
      c_in_r          => c_in_r,
      c_in_a          => c_in_a);


  -- reset

  environment: process
  begin
    z_r <= '0';
    wait for 500 ns;
    z_r <= '1';
    wait for 15 ms;
    assert false report "end simulation" severity failure;
  end process environment;

  -- CPU => cache
  -- purpose: produces input data for data cache

  request_data_from_cache: process

    variable tmp_data : bit_vector(64 downto 0);
    variable l : line;
    file req_cache : text open read_mode is
                     "/export/home/joeltu/haste/projekti/sync/data2c.in";


    begin
      c_in <= (others => '0');
      c_in_r <= '0';
      wait for 1000 ns;
      c_in_r <= '1';                    -- repeat once
      if not (endfile(req_cache)) then
        readline(req_cache,l);
        read(l,tmp_data);
        c_in <= To_StdLogicVector(tmp_data);
        wait until c_in_a = '1';
        wait for 0.1 ns;
        c_in_r <= '0';
        wait until c_in_a = '0';
      end if;
```

```
      wait for 500 ns;
      for i  in 0 to 15 loop            -- repeat 15 times
        c_in_r <= '1';
        wait for 0.1 ns;
        if not (endfile(req_cache)) then
          readline(req_cache,l);
          read(l,tmp_data);
          c_in <= To_StdLogicVector(tmp_data);
          wait until c_in_a = '1';
          wait for 0.1 ns;
          c_in_r <= '0';
          wait until c_in_a = '0';
        end if;
        wait for 500 ns;
       end loop;  -- i
       wait;
      end process request_data_from_cache;

  -- cache => main memory
  -- purpose: request data from main memory when needed


  read_data_from_mem: process
    variable tmp_data : bit_vector(31 downto 0);
    variable l : line;
    file mem_data : text open read_mode is
                    "/export/home/joeltu/haste/projekti/sync/memory.in";
    begin
      for i  in 0 to 11 loop
        data_from_mem <=(others =>'0');
        addr_2_mem_a <= '0';
        data_from_mem_r <= '0';
        wait on addr_2_mem_r until addr_2_mem_r = '1' ;  -- wait request
        addr_2_mem_a <= '1';
        wait until addr_2_mem_r = '0';
        addr_2_mem_a <= '0';
        wait for 90 ns;
        data_from_mem_r <= '1';
        if not (endfile(mem_data)) then
          readline(mem_data,l);
          read(l,tmp_data);
          data_from_mem <= To_StdLogicVector(tmp_data);
          wait until data_from_mem_a ='1';
          data_from_mem_r <= '0';
          wait until data_from_mem_a = '0';
        end if;
      end loop;
      wait;
  end process read_data_from_mem;

  -- cache => CPU
  -- Outputs the requested data

  instruction_out : process
  begin
    loop                                -- repeat forever
      d_out_a <= '0';
      wait on d_out_r until d_out_r ='1';
      d_out_a <= '1';
      wait until d_out_r = '0';
      d_out_a <= '0';
    end loop;
  end process instruction_out ;

  -- cache => main memory
  -- write through operation

  data_write_mem : process
  begin
    loop                                -- repeat forever
      data_2_mem_a <= '0';
      wait on data_2_mem_r until data_2_mem_r ='1';
      data_2_mem_a <= '1';
      wait until data_2_mem_r = '0';
      data_2_mem_a <= '0';
    end loop;
  end process data_write_mem ;

end test;
```

8 Appendix B: VHDL Code for data cache

```
-------------------------------------------------------------------------------
-- Title      : cache_pkg
-- Project    :
-------------------------------------------------------------------------------
-- File       : cache_pkg.vhd
-- Author     : Johanna Tuominen  <joeltu@utu.fi>
-- Last update: 2006/03/09
-- Platform   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Passes parameters to the synchronous cache controller.
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2006/01/09  1.0      joeltu  Created
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

package cache_pkg is

-------------------------------------------------------------------------------
-- General parameters
-------------------------------------------------------------------------------
-------------------------------------------------------------------------------
-- Structure of the cache block
-- ##############################################################################
-------------------------------------------------------------------------------
--   TAG                    |     INDEX                 | DATA             --
-------------------------------------------------------------------------------
-- Bits:        22           |       10                  | 32               --
-------------------------------------------------------------------------------

-- Width of the data bus.
constant data_bits : natural := 32;

-- Number of bits needed to represent memory address.
constant addr_bits : natural := 32;

-- Number of sets in the cache
constant cache_index : natural := 16;

-- Number of bits in the index
constant index_size : natural := 10;

-- Number of bits the tag
constant tag_size : natural := 22;


end cache_pkg;

package body cache_pkg is


end cache_pkg;
```

```
-------------------------------------------------------------------------------
-- Title      : d_cache.vhd
-- Project    :
-------------------------------------------------------------------------------
-- File       : d_cache.vhd
-- Author     : Johanna Tuominen  <joeltu@utu.fi>
-- Company    :
-- Last update: 2006/03/30
-- Platform   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Synchronous 32 bit data cache.
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2006/01/18  1.0      joeltu  Created
-------------------------------------------------------------------------------
--
--      Port name:         Purpose:
--      clk                clock signal
--      rst                asynchronous reset (active low)
--      d_rd_cache         reading is enabled by the cpu
--      d_wr_cache         writing is enabled by the cpu
--      dv_from_mem        data is valid from main memory
--      d_add_2_cache      cpu request data
--      d_in               data in to cache (cpu side)
```

```
--      mem_wr_dcache      data in to cache (memory side)
--      cache_wr_mem       data out to main memory
--      d_addr_2_mem       cache fetches data from main memory
--      d_rd_mem           enables memory reading.
--      d_hit              cache hit
--      d_out              data out from cache (to cpu side)
--------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use work.cache_pkg.all;


entity d_cache is

  port (
    clk           : in  std_logic;
    rst           : in  std_logic;
    d_rd_cache    : in  std_logic;
    d_wr_cache    : in  std_logic;
    dv_from_mem   : in  std_logic;
    d_add_2_cache : in  std_logic_vector(addr_bits - 1 downto 0);
    d_in          : in  std_logic_vector(2*data_bits-1 downto 0);
    mem_wr_dcache : in  std_logic_vector(data_bits-1 downto 0);
    cache_wr_mem  : out std_logic_vector(data_bits-1 downto 0);
    d_addr_2_mem  : out std_logic_vector(addr_bits - 1 downto 0);
    d_rd_mem      : out std_logic;
    d_hit         : out std_logic;
    d_out         : out std_logic_vector(data_bits - 1 downto 0));

end d_cache;

architecture arch of d_cache is

 -- function declarations

  function replace_block(b_code: integer) return integer is

    variable r_code: integer;
    variable i : integer;

  begin

    i := b_code;
    if i < cache_index - 1 then
      i := i + 1;
    else
      i := 0;
    end if;

    r_code := i;
    return r_code;
  end;



  function choose_tag_2_replace (ctag : integer) return std_logic is
    variable rtag : std_logic;
    variable ptag : integer := 0;

  begin
    if ptag < ctag then
      rtag := '1';
      ptag := ctag;
    else
      rtag := '0';
      ptag := ctag;
    end if;
    return rtag;
  end;

  -- type and signal declarations

  type d_cache_block is record
    index: std_logic_vector(index_size - 1 downto 0);
    tag_1: std_logic_vector(tag_size - 1 downto 0);
    tag_2: std_logic_vector(tag_size - 1 downto 0);
    data_1: std_logic_vector(data_bits - 1 downto 0);
    data_2: std_logic_vector(data_bits - 1 downto 0);
  end record;

  type d_cache_matrix is array (0 to cache_index-1) of d_cache_block;

  signal d_cache_contents : d_cache_matrix;
  signal atag : std_logic_vector(tag_size - 1 downto 0);
  signal aind : std_logic_vector(index_size - 1 downto 0);
  signal adat : std_logic_vector(data_bits - 1 downto 0);
  signal mdat : std_logic_vector(data_bits - 1 downto 0);
```

```
  signal tmp_d_out : std_logic_vector(data_bits - 1 downto 0);
  signal d_miss : std_logic;
  signal wr_ind  : integer;
  signal wr_prev : integer;
  signal wind : integer;
  signal replace : std_logic;
  signal cpu_wr_2_cache : std_logic;
  signal mem_wr_2_cache : std_logic;
  signal write_through : std_logic;

begin  -- arch


  -- cpu request either read or write access from the data cache

  operation_request_from_cpu: process (clk, rst)
  begin  -- process instr_in
    if rst = '0' then                   -- asynchronous reset (active low)
      atag <= (others => '0');
      aind <= (others => '0');
      adat <= (others => '0');
    elsif clk'event and clk = '1' then  -- rising clock edge
      if d_rd_cache = '1' then
        atag <= d_add_2_cache(addr_bits - 1 downto index_size);
        aind <= d_add_2_cache(index_size - 1 downto 0);
      elsif d_wr_cache = '1' then
        atag <= d_in(2*data_bits-1 downto 42);
        aind <= d_in(41 downto 32);
        adat <= d_in(data_bits-1 downto 0);
      else
        null;
      end if;
    end if;
  end process operation_request_from_cpu;

  -- Process to handle writes to cache.
  -- write operation from cpu to memory is implemented as a
  -- write through, that is both cache and memory are updated
  -- in every write operation.

  write_2_cache: process (clk, rst)
  begin
    if rst = '0' then                   -- asynchronous reset (active low)
      write_through <= '0';
      wr_prev <= 0;
      for i  in 0 to (cache_index - 1) loop
       d_cache_contents(i).index <= (others => '1');
       d_cache_contents(i).tag_1 <= (others => '1');
       d_cache_contents(i).tag_2 <= (others => '1');
       d_cache_contents(i).data_1 <= (others => '1');
       d_cache_contents(i).data_2 <= (others => '1');
      end loop;
      write_through <= '0';
    elsif clk'event and clk='1' then    -- rising clock edge
      if cpu_wr_2_cache = '1' then
        if replace = '1' then
          d_cache_contents(wr_ind).index <= aind;
          d_cache_contents(wr_ind).tag_1 <= atag;
          d_cache_contents(wr_ind).data_1 <= adat;
          write_through <= '1';
          wr_prev <= wr_ind;
        elsif replace = '0' then
          d_cache_contents(wr_ind).index <= aind;
          d_cache_contents(wr_ind).tag_2 <= atag;
          d_cache_contents(wr_ind).data_2 <= adat;
          write_through <= '1';
          wr_prev <= wr_ind;
        else
          write_through <= '0';
        end if;
      elsif mem_wr_2_cache = '1' then
        if replace = '1' then
          d_cache_contents(wr_ind).index <= aind;
          d_cache_contents(wr_ind).tag_1 <= atag;
          d_cache_contents(wr_ind).data_1 <= mdat;
          wr_prev <= wr_ind;
        elsif replace = '0' then
          d_cache_contents(wr_ind).index <= aind;
          d_cache_contents(wr_ind).tag_2 <= atag;
          d_cache_contents(wr_ind).data_2 <= mdat;
          wr_prev <= wr_ind;
        else
          null;
        end if;

      else
        null;
      end if;
```

```
  else
  null;
  end if;

end process write_2_cache;


-- controls write operations: (cpu => cache), (cache => memory), (memory => cache).


write_control: process (clk, rst)
begin
  if rst = '0' then                   -- asynchronous reset (active low)
    cpu_wr_2_cache <= '0';
    mem_wr_2_cache <= '0';
  elsif clk'event and clk = '1' then  -- rising clock edge
    if (d_wr_cache = '1' and dv_from_mem = '0') then          -- write request from cpu
      cpu_wr_2_cache <= '1';
      mem_wr_2_cache <= '0';
    elsif (dv_from_mem = '1' and d_wr_cache = '0') then     -- write request from main memory
      mem_wr_2_cache <= '1';
      cpu_wr_2_cache <= '0';
    else
      cpu_wr_2_cache <= '0';
      mem_wr_2_cache <= '0';
    end if;
  end if;

end process write_control;

-- Data is requested from memory in case of cache miss.

rd_from_mem: process (clk, rst)
begin
  if rst = '0' then                     -- asynchronous reset (active low)
    d_addr_2_mem <= (others => '0');
    d_rd_mem <= '0';

  elsif clk'event and clk = '1' then  -- rising clock edge
    if d_miss = '1' then
      d_addr_2_mem(addr_bits - 1 downto index_size)<=atag;
      d_addr_2_mem(index_size - 1 downto 0)<=aind;
      d_rd_mem <= '1';
    else
      d_rd_mem <= '0';
    end if;
  end if;

end process rd_from_mem;

-- Requested data is received from memory.

data_from_mem: process (clk, rst)
begin
  if rst = '0' then                   -- asynchronous reset (active low)
    replace <= '0';
    wr_ind <= 0;
    wind <= 0;
    mdat <= (others => '0');
    d_out <= (others => '0');
  elsif clk'event and clk='1' then    -- rising clock edge
    if dv_from_mem = '1' and d_rd_cache = '0' then
      mdat <= mem_wr_dcache(data_bits-1 downto 0);
      d_out <= mem_wr_dcache(data_bits-1 downto 0);
      wr_ind <= replace_block(wr_prev);
      wind <= conv_integer(aind);
      replace <= choose_tag_2_replace(wind);
    else
      d_out <= tmp_d_out;
    end if;
  else
    null;
  end if;

end process data_from_mem;

-- If cache is written then the same data is written to main memory.

wr_2_mem: process (clk, rst)
begin
  if rst = '0' then                    -- asynchronous reset (active low)
    cache_wr_mem <= (others => '0');
  elsif clk'event and clk = '1' then  -- rising clock edge
    if write_through = '1' then
      cache_wr_mem(data_bits - 1 downto 0)<= adat;
    else
      null;
    end if;
```

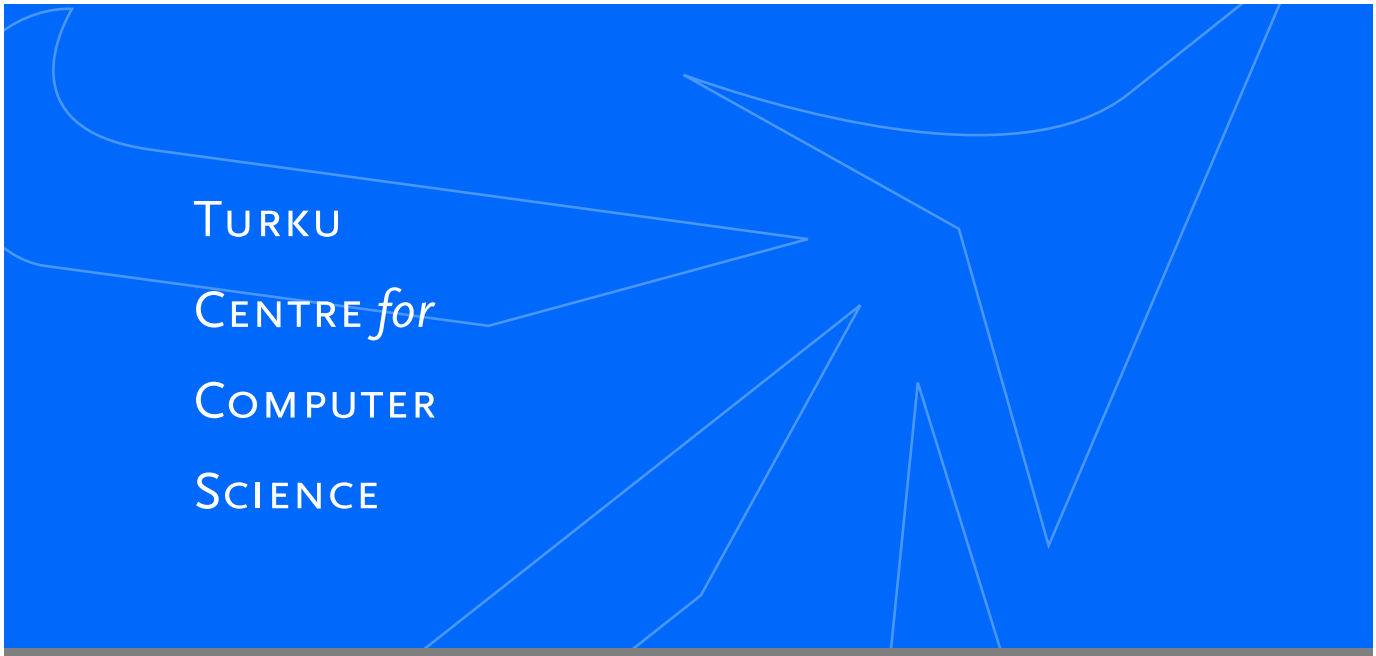
```
    else
      null;
    end if;

  end process wr_2_mem;

  -- read request from cpu.

  rd_data_from_cache: process (clk, rst)
  begin
    if rst = '0' then                   -- asynchronouns reset (active low)
      d_miss <= '0';
      tmp_d_out <= (others => '0');
      d_hit <= '0';
    elsif clk'event and clk = '1' then  -- rising clock edge
      if d_rd_cache = '1' then
        for i in 0 to cache_index - 1 loop
          if (d_cache_contents(i).index = aind) and (d_cache_contents(i).tag_1 = atag) then
            tmp_d_out <= d_cache_contents(i).data_1;
            d_miss <= '0';
            d_hit <= '1';
          elsif d_cache_contents(i).tag_2 = atag then
            tmp_d_out <= d_cache_contents(i).data_2;
            d_miss <= '0';
            d_hit <= '1';
          else
            d_miss <= '1';
            d_hit <= '0';
          end if;
        end loop;       --i
      else
        d_hit <= '0';
        d_miss <= '0';
      end if;
    else
      null;
    end if;

  end process rd_data_from_cache;

end arch;
```

8.1 Test enviroment for synchronous data cache

```
-------------------------------------------------------------------------------
-- Title      : t_d_cache.vhd
-- Project    :
-------------------------------------------------------------------------------
-- File       : t_d_cache.vhd
-- Author     : Johanna Tuominen  <joeltu@utu.fi>
-- Company    :
-- Last update: 2006/03/30
-- Platform   : VHDL'93
-------------------------------------------------------------------------------
-- Description: Test bench for data cache
-------------------------------------------------------------------------------
-- Revisions  :
-- Date        Version  Author  Description
-- 2006/01/19  1.0      joeltu  Created
-------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use work.cache_pkg.all;


entity t_d_cache is

end t_d_cache;

architecture test of t_d_cache is

  component d_cache is
    port(
    clk           : in  std_logic;
    rst           : in  std_logic;
    d_rd_cache    : in  std_logic;
    d_wr_cache    : in  std_logic;
    dv_from_mem   : in  std_logic;
    d_add_2_cache : in  std_logic_vector(addr_bits - 1 downto 0);
    d_in          : in  std_logic_vector(data_bits-1 downto 0);
    mem_wr_dcache : in  std_logic_vector(data_bits-1 downto 0);
```

```
    cache_wr_mem  : out std_logic_vector(data_bits-1 downto 0);
    d_addr_2_mem  : out std_logic_vector(addr_bits - 1 downto 0);
    d_rd_mem      : out std_logic;
    d_hit         : out std_logic;
    d_out         : out std_logic_vector(data_bits - 1 downto 0));
  end component;

  signal clk : std_logic := '0';
  signal rst : std_logic := '0';
  signal d_rd_cache : std_logic:='0';
  signal d_wr_cache    :  std_logic:='0';
  signal dv_from_mem   :  std_logic;
  signal d_add_2_cache :  std_logic_vector(addr_bits - 1 downto 0) :=(others => '0');
  signal d_in          :  std_logic_vector(data_bits-1 downto 0) :=(others => '0');
  signal mem_wr_dcache :  std_logic_vector(data_bits-1 downto 0) :=(others => '0');
  signal cache_wr_mem  :  std_logic_vector(data_bits-1 downto 0);
  signal d_addr_2_mem  :  std_logic_vector(addr_bits - 1 downto 0);
  signal d_rd_mem      :  std_logic;
  signal d_hit         :  std_logic;
  signal d_out         :  std_logic_vector(data_bits - 1 downto 0);
  signal wr_data : std_logic:='0';
  signal rd_data : std_logic:='0';

begin  -- test

  dut: d_cache
  port map (
    clk           => clk,
    rst           => rst,
    d_rd_cache    => d_rd_cache,
    d_wr_cache    => d_wr_cache,
    dv_from_mem   => dv_from_mem,
    d_add_2_cache => d_add_2_cache,
    d_in          => d_in,
    mem_wr_dcache => mem_wr_dcache,
    cache_wr_mem  => cache_wr_mem,
    d_addr_2_mem  => d_addr_2_mem,
    d_rd_mem      => d_rd_mem,
    d_hit         => d_hit,
    d_out         => d_out);

  --clock generation

  clock: process
  begin
    wait for 5 ns; clk <= not clk;
  end process clock;

  -- reset and control signal generation

  control: process
  begin
    rst <= '0';
    wait for 50 ns;
    rst <= '1';
    wait for 200 ns;
    rd_data <= '1';                 -- cache read
    wait for 10 ns;
    rd_data <= '0';
    wait for 500 ns;
    rd_data <= '1';                 -- cache read
    wait for 10 ns;
    rd_data <= '0';
    wait for 500 ns;
    rd_data <= '1';                 -- cache read
    wait for 10 ns;
    rd_data <= '0';
    wait for 500 ns;
    rd_data <= '1';                 -- cache read
    wait for 10 ns;
    rd_data <= '0';
    wait for 500 ns;
    rd_data <= '1';                 -- cache read
    wait for 10 ns;
    rd_data <= '0';
    wait for 500 ns;
    rd_data <= '1';                 -- cache read
    wait for 10 ns;
    rd_data <= '0';
    wait for 500 ns;
    rd_data <= '1';                 -- cache read
    wait for 10 ns;
    rd_data <= '0';
    wait for 500 ns;
    rd_data <= '1';                 -- cache read
    wait for 10 ns;
    rd_data <= '0';
    wait for 500 ns;
```

```
    wr_data <= '1';                 -- cache write (cpu)
    wait for 10 ns;
    wr_data <= '0';
    wait for 500 ns;
    wr_data <= '1';                 -- cache write (cpu)
    wait for 10 ns;
    wr_data <= '0';
    wait for 500 ns;
    wr_data <= '1';                 -- cache write (cpu)
    wait for 10 ns;
    wr_data <= '0';
    wait for 2000 ns;
    assert false report "end simulation" severity failure;
  end process control;



  read_data_from_mem: process

    variable tmp_data : bit_vector(data_bits-1 downto 0);
    variable l : line;
    file d_mem_data : text open read_mode is "/export/home/joeltu/haste/projekti/sync/dmemory.in";

  begin
    wait on d_rd_mem until d_rd_mem = '1';
    if not (endfile(d_mem_data)) then
      readline(d_mem_data,l);
      read(l,tmp_data);
      mem_wr_dcache <= To_StdLogicVector(tmp_data);
      dv_from_mem <= '1';
    wait for 90 ns;
    end if;
    dv_from_mem <= '0';
  end process read_data_from_mem;

  read_data_from_cache: process

    variable tmp_data : bit_vector(data_bits-1 downto 0);
    variable l : line;
    file d_req_cache : text open read_mode is "/export/home/joeltu/haste/projekti/sync/dcache.in";

    begin
      wait on rd_data until rd_data = '1';
      d_rd_cache <= '1';
      if not (endfile(d_req_cache)) then
        readline(d_req_cache,l);
        read(l,tmp_data);
        d_add_2_cache <= To_StdLogicVector(tmp_data);
        wait for 5 ns;
        d_rd_cache <= '0';
      end if;
  end process read_data_from_cache;

  write_data_2_cache: process
    variable tmp_data : bit_vector(2*data_bits-1 downto 0);
    variable l : line;
    file d_wr_data : text open read_mode is "/export/home/joeltu/haste/projekti/sync/data2c.in";

  begin
    wait on wr_data until wr_data = '1';
    d_wr_cache <= '1';
    if not (endfile(d_wr_data)) then
      readline(d_wr_data,l);
      read(l,tmp_data);
      d_in <= To_StdLogicVector(tmp_data);
      wait for 5 ns;
    d_wr_cache <= '0';
    end if;

  end process write_data_2_cache;


end test;
```



Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1717-0
ISSN 1239-1891