King Saud University
Journal of King Saud University – Computer and Information Sciences
www.ksu.edu.sa
www.sciencedirect.com

A high level implementation and performance evaluation of level-I asynchronous cache on FPGA

Mansi Jhamb [a,*], **R.K. Sharma** [b], **A.K. Gupta** [b]

[a] *University School of Information and Communication Technology, Guru Gobind Singh Indraprastha University, Dwarka Sector 16C, New Delhi 110078, India*
[b] *Department of Electronics and Communication Engineering, NIT Kurukshetra, India*

Received 19 February 2015; revised 9 June 2015; accepted 27 June 2015
Available online 31 October 2015

KEYWORDS

Asynchronous;
Handshaking;
Cache

Abstract To bridge the ever-increasing performance gap between the processor and the main memory in a cost-effective manner, novel cache designs and implementations are indispensable. Cache is responsible for a major part of energy consumption (approx. 50%) of processors. This paper presents a high level implementation of a micropipelined asynchronous architecture of L1 cache. Due to the fact that each cache memory implementation is time consuming and error-prone process, a synthesizable and a configurable model proves out to be of immense help as it aids in generating a range of caches in a reproducible and quick fashion. The micropipelined cache, implemented using C-Elements acts as a distributed message-passing system. The RTL cache model implemented in this paper, comprising of data and instruction caches has a wide array of configurable parameters. In addition to timing robustness our implementation has high average cache throughput and low latency. The implemented architecture comprises of two direct-mapped, write-through caches for data and instruction. The architecture is implemented in a Field Programmable Gate Array (FPGA) chip using Very High Speed Integrated Circuit Hardware Description Language (VHSIC HDL) along with advanced synthesis and place-and-route tools.

1. Introduction

In technical market the majority of processors are embedded systems (Tennenhouse, 2000). The microcontrollers, DSPs, FPGAs and their combinations as SoCs provide solution for embedded system (George et al., 1999). Since FPGAs have shown significant advances in the terms of speed, density and storage capacity, they have become most commonly used embedded general purpose computing environment. Owing to the diverse functionality of FPGAs, complexity level increases thereby increasing the number of logic gates in the

* Corresponding author.
E-mail addresses: mansi.jhamb@gmail.com (M. Jhamb), mail2drrks@gmail.com (R.K. Sharma), anilg699@rediffmail.com (A.K. Gupta).
Peer review under responsibility of King Saud University.

Production and hosting by Elsevier

circuit. This results in increased energy dissipation in the system (Anderson and Najm, 2004; Shang et al., 2002). Another concern is the increasing difference in the speed of the processor and main memory which creates a bottleneck while accessing the data from the memory. While the programers want unlimited fast memory, it is too expensive to be achieved. Hence, the optimal solution is to provide a memory hierarchy in which each level is faster and more expensive per byte than the immediate higher level (Patterson and Hennessey, 2003). The cache is the first level in this memory hierarchy. Modern multi-core processors support multiple levels of cache to improve performance. In most such systems, caches (last level caches) are often shared by the processors for concurrent applications (Warrier et al., 2013). According to the principle of locality of references the data and the instructions exhibit temporal and spatial locality. This principle forms the basis of cache memory hierarchy, leading to the fact the processor performance depends on the speed of cache rather than slow main memory. 45% of energy consumption of a processor is contributed by cache hierarchy (Segars, 2001).

Research reveals that the adjustment of the cache memory parameters can contribute to reduction in energy consumption to 62% (Ross et al., 2005), leading to the performance enhancement by 30% (Ross et al., 2004). The complex task of determining an appropriate cache configuration for a dedicated application, simulation and analysis takes considerable computational time. For reduction in the area and energy dissipation without compromising the performance, designers need to develop strategies for design space exploration. There are a number of ways to implement the L1 cache and the selected design strategy impacts the complete behavior of processor system (Bahar et al., 1998; Milenkovic et al., 2003). For the implementation parameters to be freely selected, an RTL model is desired. In the logic simulation, the desired cache configuration can be employed after the cache model parameters are set. This work presents a high level implementation of a micropipelined asynchronous architecture of an L-1 cache using FPGAs.

2. Related work

In embedded-system design one has to compromise between energy-dissipation, performance and cost. The critical task of choosing the best cache architecture involves the selection of total cache size, amount of associativity, cache line size and several other architectural options. These characteristics greatly influence the hit rate and the energy consumption in cache access. The power consumption occurs when (i) the cache is accessed. (ii) The data are transferred from/to the next memory level during a cache miss and also by the idle processor when miss occurs.

Associativity segregates a cache into a number of ways, each of which is looked up concurrently during a cache access. For some programs, the cache hit rate is improved on increasing the number of ways to two or four (Patterson and Hennessey, 2003), beyond four the improvement is not significant. More ways imply more concurrent look-ups per access leading to more energy per access: a direct mapped cache uses only 30% of the energy per access as a four way set associated cache (Reinman and Jouppi, 1999). Highest possible associativity is desired in performance-oriented applications. Table 1 summarizes the features of several cache architectures in embedded microprocessors.

Table 1 Cache architectures in embedded microprocessors.

Processor	Instruction cache			Data cache		
	Size	As.	Line	Size	As.	Line
AMD-K6-IIIE	32 K	2	32	32 K	2	32
IBM PPC 750CX	32 K	8	32	32 K	8	32
IBM PPC 7603	16 K	4	32	16 K	4	32
IBM 750 FX	32 K	8	32	32 K	8	32
IBM 403GCX	16 K	2	16	8 K	2	16
IBM Power PC 405CR	16 K	2	32	8 K	2	32
Intel 960IT	16 K	2	N/A	4 K	2	N/A
Motorola MPC8240	16 K	4	32	16 K	4	32
Motorola MPC823E	16 K	4	16	8 K	4	16
Motorola MPC8540	32 K	4	32/64	32 K	4	32/64
Motorola MPC7455	32 K	8	32	32 K	8	32
Xilinx Virtex IIPro	16 K	2	32	8 K	2	32

A novel configurable cache architecture (Zhang et al., 2005) incorporates three configurable cache parameters, these are configured on setting a few bits in the configuration register. The cache can be configured in software as either direct mapped, two-way or four-way set associated while utilizing the full capacity of cache. Such configuration is attained by employing a technique called way concatenation (Zhang et al., 2003a). The technique called line concatenation (Zhang et al., 2003b) is employed to configure the cache line size. Several architectures for reducing the energy consumption in the cache are reported in the literature. One of them is the partitioning of cache into several smaller caches (Racunas and Patt, 2003). This results in the reduction of access time and the power cost per access. Another approach, filter cache (Kin et al., 1997) trades performance for power consumption by the filtration of cache references through an unusually small L1 cache. An L2 cache similar in structure and size to a L1 cache is placed after the filter cache for minimizing the performance loss. A distinct alternative, selective cache ways (Albonesi, 2000) renders the ability to disable a subset of ways in a set associative cache during the intervals of modest cache activity whereas the complete cache is operational for cache-intensive periods. In another approach the conventional unified data cache can be replaced with multi specialized caches. Each one handles different type of memory references as per their specific locality characteristics (Lee et al., 2001). These options make it possible to improve in terms of performance and power efficiency. Lastly Jin and Cho, 2006 achieves power saving in L1 cache by exploiting spacial locality. A useful state of the art design of a pipelined asynchronous cache system is as reported (Nyström et al., 2003). In their work a pipelined cache system is designed for use in an asynchronous MIPS R3000-compatible processor. This design (Nyström et al., 2003) being the first of its kind, represented a QDI (Quasi-delay insensitive) message passing implementation of L1 cache for high speed asynchronous CPU; many assumptions were made for design compatibility with now-obsolete R3000 cache architecture.

Asynchronous paradigm has potential advantages of low power, low noise, modularity, composability and ease of integration to an existing system, proving them to be an interesting alternative in the several applications. The clock distribution and alignment consume a sufficient amount of resources like wiring, power and area (Elrabaa, 2012). The asynchronous design techniques impart modularity, composability, lower power consumption and low noise (Peeters, 1996; Liljeberg et al., 2004). Hence, they can provide a potential alternative. For asynchronous operation in the disk buffer cache memory, performance and reliability can be improved significantly using the algorithms (Hac, 1993). These algorithms permit the files to be written into the buffer cache by the processors, considering the number of active processes in the system and queue length to the disk buffer cache. The lack of CAD tools supporting the design flow of the self-timed circuits turn out to be the biggest disadvantage. Till date their design process is performed using existing tools, meant for synchronous design. This includes a lot of full-custom work which is very time consuming and hence expensive. In this implementation we analyze the functionality of the design flow for self-timed circuits provided by pipelining. The design entry is VHDL, a programing language targeted for the front-end design flow. This paper presents a high level implementation of micropipelined asynchronous architecture of L1 cache memory. Our paper presents an extension to the basic design as reported (Nyström et al., 2003) to treat the high level implementation of micropipelined architecture of asynchronous L1 cache memory. The design has been coded in hardware description language and targeted Xilinx-FPGA. The purpose is to compare the performance of pipeline implementation with the corresponding synchronous implementation. The performance of the self-timed implementation is analyzed and compared with its synchronous counterparts.

This work contributes an RTL model of asynchronous L-1 cache which unlike a specialized cache generator (Putnam et al., 2009), is not customized to a specific processor core (CPU). We have applied two cache constructs: instruction cache (I-Cache) and data cache (D-Cache). In this RTL model, the complicated task of reading and writing from/to L-1 cache is handled by state machines. In this model the option of arbitration between the I-Cache and D-Cache is available for supporting the single bus testing of the processor accommodating L-1 cache. To the best of our information, generic RTL models of cache do not exist in the public domain in the present scenario. The shortcomings of this model: being interface agnostic, this model cannot support deeper pipelining. Once the CPU is selected, the performance co-optimization of cache and CPU must be done.

3. Cache

For the description of the structure and parameters of the model, a brief introduction to cache along with its functions and terminologies is presented.

3.1. Basic terminologies and functions

An example of cache organization is demonstrated in Fig. 1. In our implementation, there is one dedicated cache for instructions and one for data. The data and tag bits constitute each cache line and these cache lines constitute the cache. Tag bits indicate the main memory address associated with the corresponding cache line. The program counter is used for addressing the instruction cache and the data cache takes address from store and load instruction. The address is divided into 3 fields: tag (first field), index (second field), line offset (third field). The first field indicates MSB of the requested main memory address, the second identifies the cache line, and the third selects the appropriate word in the cache line. This applies to both instruction cache and data cache.

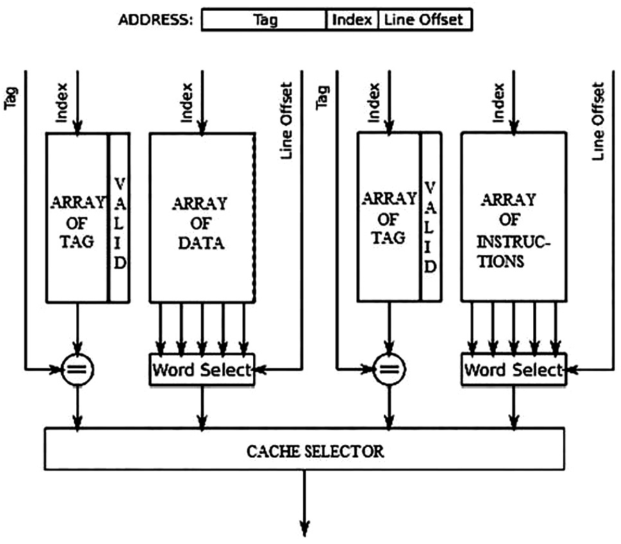


Figure 1 Basic organization of cache memory.

Whenever the processor wants to read data or instruction from the cache or write data into the cache, a basic sequence of steps is followed in any cache memory.

During the completion of the transfer from or to the next level of hierarchy, the CPU is stalled. On the start of the program when the first set of data and instruction is sent to the cache, CPU is being stalled. The main memory addresses are mapped to the cache entries by dividing the address space into the block similar to the size of a cache line. The blocks are combined together in a page. The page size is equal to the size of the cache memory array so that the complete page can be loaded into the cache in one go. When the CPU accesses the cache, the tag from the memory address is compared with the tag field stored in the cache line along with the word. It is said to be a cache hit when the tags match and valid bit is set. Else, a cache miss is reported pointing that the required data are not present in the cache.

3.2. Cache parameters

For the implementation of highly optimized cache memories, several issues have to be resolved. Several techniques are available to enable the cache to have a good cache hit ratio and at the same time, be accessed quickly (Peir et al., 1998). Various parameters associated with caches have to be decided during the design of the cache. These parameters are listed in the remaining part of the section.

o Address for Cache – Cache can work with physical address or with virtual address (Peir et al., 1998).
o Size of cache and the size of page.
o Address mapping schemes – This is the mechanism used to assign the cache line to each higher level memory entry where it is to be stored. The three common mapping schemes are direct, fully associative and set-associative mapping.

o Jouppi (1993) has shown write policy – In order to store data, the cache can use write-through mechanism or write-back mechanism. Moreover, in the case of a cache miss, the cache can follow a write-allocate or a no-write-allocate mechanism.

3.3. Cache operation

The Fig. 2 shows the operations of the I-Cache in the form of state transitions consisting of only the essential signals. The I-Cache is a Mealy FSM composed of three states. It has to perform only read operations as the processor will only request for instructions to be executed from the memory and will not store instructions into the memory.

- When the cache powers up, it enters the *Flush* state where it is brought to a known initial state. In order to avoid fake hits, all the valid bits are made zero.
- On the reception of request for instruction, cache switches to *tag_compare* state which involves the comparison of tags and checking of the valid bits to ensure whether the requested instruction is present in the cache or not.
- In case a cache hit is reported (valid bit is set and the tags match), the CPU is supplied with instruction while the cache sustains in the present state.
- If there is a cache miss, the state changes to *Memory_Read* in which the requested instruction is loaded from the main memory. Once the cache line is updated and the corresponding valid bit is set, the cache returns to the *Tag_Compare* state and supplies the instruction to the processor.

The state transition diagram for Data cache is given in Fig. 3. It is represents a Mealy Finite State Machine. D-Cache is not as frequently accessed as I-Cache.

- After the initial *Flush* state (similar to the I-Cache), the cache enters the *Idle* state where it waits for the request from the CPU.
- When a request from CPU arrives, the cache enters the *Tag_Compare* state where it compares the tags and checks the valid bit to determine if there is a cache hit or miss.
- If the instruction received is "read" and there is a cache hit, the cache goes to
- *Supply_Data* state where it sends the required data to the CPU. Then, it returns to the *Idle* state.
- If the instruction received is "read" and there is a cache miss, the cache goes to
- *Memory_Read* state. The data are transferred from the higher level memory into the cache. Once the cache line is updated and the corresponding valid bit is set, the cache enters the *Supply_Data* state.

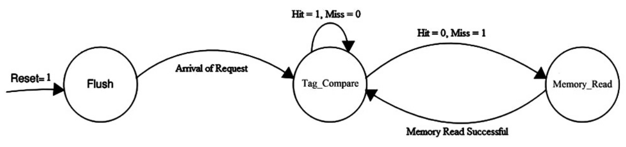


Figure 2 State transition diagram for instruction cache.

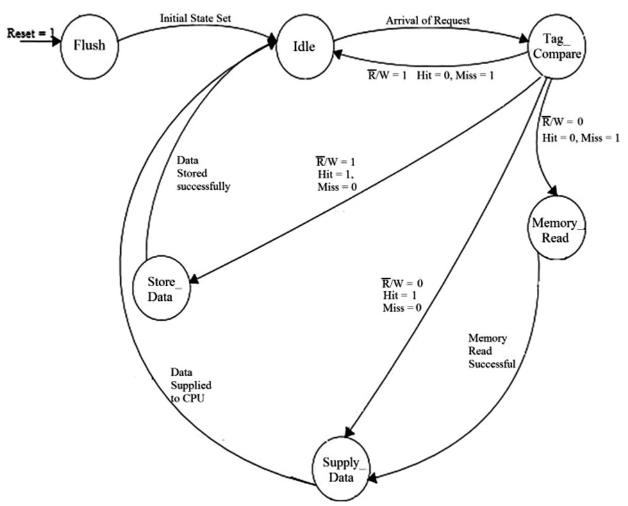


Figure 3 State transition diagram for data cache.

- If the instruction received is "write" and there is a cache hit, the cache goes to
- *Store_Data* state. The data provided by the CPU are stored in the specific location and the cache enters *Idle* state once again after storing the data.

If the instruction received is "write" and there is a cache miss, the cache directly goes to *Idle* state.

4. Design

4.1. Design of asynchronous interface

The asynchronous interface facilitates the data movement from one module to another module using the concept of handshaking as shown in Fig. 4.

The process of Handshaking is explained as following steps:

Step 1. Initially all four handshake signals are low.
Step 2. The block A sends out data and in the meantime pulls req_A high.
Step 3. Since req_B is low, ack_A will go high. This positive edge of ack_A is also the clock signal for data register. Thus, data_A is loaded to the stage register.
Step 4. The ack_A signal is also the input of the next C-Element. When ack_A is asserted high, it pulls req_B high.
Step 5. When req_A goes low, ack_A will also become low.
Step 6. The second C-Element waits for the ack_B signal from next stage to go high. When ack_B is asserted high, it conveys that next stage has read the output data of the register. This causes the req_B to go low.
Step 7. Finally, corresponding to req_B going low, the block B pulls the ack_B signal low.

4.2. Cache architecture

Each cache is decomposed into two modules: control and core. The operations of the cache are monitored by the control section and it is a component of the main data path of the

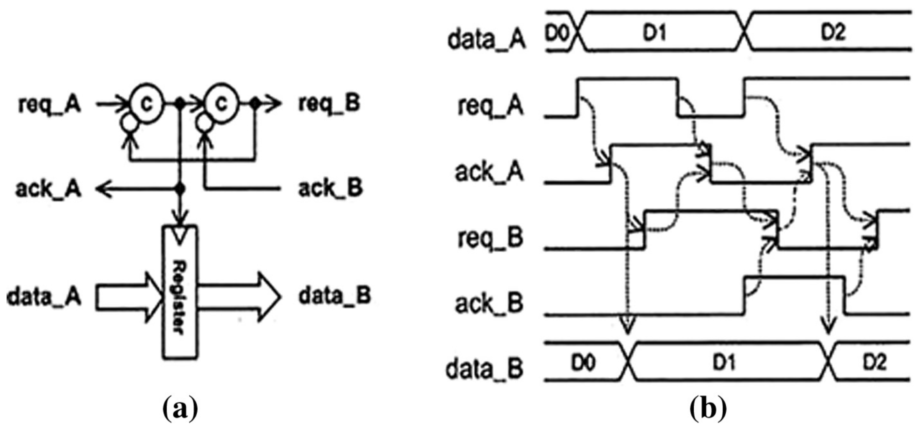


Figure 4 Muller pipeline using C-Elements (a) structure, (b) 4-Phase bundled data handshaking protocol.

processor shown in Fig. 5. It is responsible for decoding addresses, scheduling refills and giving instructions to the core. The core contains a 256 line memory array where the data (D-Cache) or instructions (I-Cache) along with the tag bits are stored. Four bits of tag are associated with each cache line. Core also contains the necessary circuitry to perform the operations directed by the control.

The control part in each cache is further segregated into L-Control and R-Control as shown in Fig. 6 which shows the detailed architecture of the D-Cache. Most parts of the cache use single-rail (1-wire for 1-bit of information) 4-phase bundled data protocol, the feedback from R-control to L-control uses 4-phase dual rail (2 wires for a bit of information) protocol for easier communication.

The core is controlled from only one place, i.e., L-control. This allows the flow of control to be linear and implementation to be more systematic. Also, the R-control is entirely responsible for sending requests to the higher level memory as well as for sending results to the processor since it knows the result of tag comparison before L-control.

The design does not support partial word operations, i.e., data smaller than one word cannot be accessed. The caches use physical addresses and are direct-mapped. The write policy chosen for D-cache is write-through. No refills are made in case of a cache miss for write operations. Data transfer occurs in both caches only after cache hit/miss is determined. Furthermore, the refills are made in groups of four (page size is four), i.e., each refill operation loads four words of data (the desired data and the data from the nearby locations) from the higher level memory into the cache. This is because a refill is a costly operation in terms of time taken and so it is beneficial to reduce the number of refill operations.

In the following sections, the design of each module of the D-Cache has been explained in detail. While the modules of I-Cache are similar to that of the D-Cache, the difference is that the mechanism of write is not present in I-Cache. So there is no input instruction or data from processor to I-Cache.

4.2.1. Design of L-control

This is the first part of the cache which receives the request and message from the CPU and is responsible for directing the operations of the core. Its name is derived from Left-control as it is located to the left of the core in the design. For I-Cache instructions are received from the instruction fetch unit whereas for D-Cache instructions are received from instruction decode. In the design of D-Cache, it has been assumed that the processor issues instructions through a single read/write signal with read given by logic "0" while write is given by logic "1".

L-control decodes the received 16 bit address to generate the 8 bit index using the lower eight bits of the address (bits 0–7) and a 4 bit tag using the next four bits of the address (bits 8–11). It then sends the index to the core and the tag to R-control. This tag is the expected tag that should be present in the data line selected in the core. L-control issues "read" instruction to the core in case of a read operation and a "check" instruction in case of a write operation. The "check" instruction results in a comparison of the tags. The cache will only write the data if the referenced memory location is currently present in the cache and so "check" instruction is necessary to avoid unnecessary writes. Depending on the feedback received from R-control, L-control decides the next course of action. The following four cases can occur after receiving feedback:

- Cache hit in a read operation: L-control takes no action.
- Cache miss in a read operation: L-control issues a "refill" instruction to the core. While R-control schedules the refill operation by sending request to the main memory, L-control instructs the core to store the data from the higher level memory when it arrives and then provide the results to the processor.
- Cache hit on a write operation: L-control gives the "write" instruction to the core to actually store the data from processor in the cache location.
- Cache miss on write operation: L-control takes no action.

The L-Control can issue four instructions to the core. These are summarized in Table 2.

4.2.2. Design of R-control

This is the third part of the D-cache which compares tags, sends data and acknowledgement to the processor and schedules the refill operations. It derives its name from Right-control as it is located to the right of the core. It contains a "compare" unit, "send" unit and a "refill" unit. The "compare" unit compares the tags received from the L-Control and core. It sends the result to the other two units of R-Control and also to L-Control. The "send" unit exists to

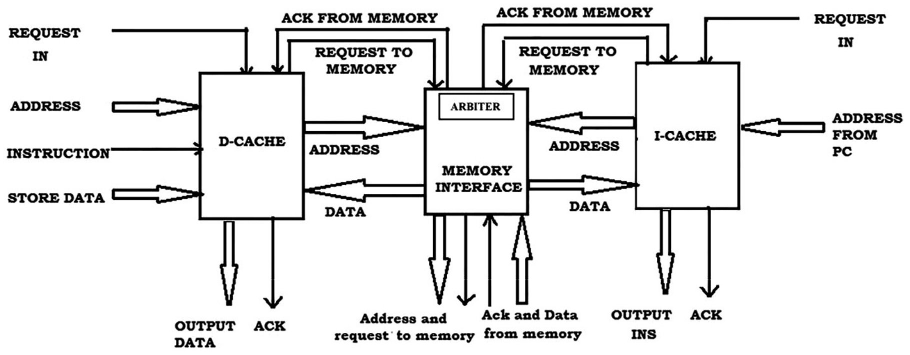


Figure 5 Architectural composition of the implemented cache. (Memory here refers to the next memory level.)

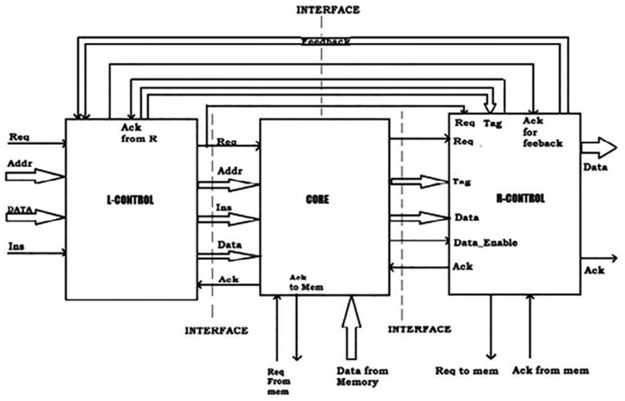


Figure 6 Architectural structure of the D-Cache.

Table 2 Instructions issued to core by L-Control.

CODE	Instruction	Function
00	READ	Send data and Tag to R-Control from the location specified
01	REFILL	Store the data from main memory in the cell specified and then send the data from required line to R-Control
10	CHECK	Send the Tag from line specified to R-control
11	WRITE	Store the data provided in the line specified

send the correct data and also acknowledgement to the processor. During a read operation, the "send" unit checks the result from "compare" unit.

If there is a tag match (cache hit), then it forwards the data from the core to the processor along with an acknowledgement. If there is a tag mismatch (cache miss), then it waits for the data from a refill operation. Once it receives it, it will send that data along with an acknowledgement to the processor. In case of a write operation, the "send" unit simply sends an acknowledgement to the processor once the operation is completed. The "refill" unit is responsible for issuing a refill request to the memory interface during a cache miss in a read operation. Once the data are retrieved from the higher memory level, the "refill" unit sends it to the "send" unit.

When R-control receives a request from L-control, it stores the tag received from it and then acknowledges the request. On receiving a request from core, R-control compares the tag from L-control (stored earlier) with that from core. If they match, then a cache hit is said to occur. Otherwise, it is a cache miss. The result of the comparison is given to the L-control as feedback. The core specifies the operation being performed (read or write) to R-Control using an additional control signal, Data_Enable, which is set to "1"for a read operation and is "0"for a write operation. For the case of a cache hit, if the Data_Enable is high, then the R-Control reads the data from the core. An acknowledgement is sent to the core after tag comparison and reading of data (if Data_Enable is high and there is a cache hit). In case the data are to be sent, R-control sends the data from the core to the CPU along with an acknowledgement. In case data are not to be sent (meaning write operation), R-control waits for the Core to store the data. Once the core has stored the data, it informs the R-control which then sends the acknowledgement to the processor. For a cache miss as well, R-control checks the Data_Enable signal. If the Data_Enable is low indicating a write operation, R-control sends an acknowledgement to the processor as the desired memory element is not in the cache and so no write operation is needed. But if it is high, indicating a read operation, R-control sends a request to the next memory level which is received by the memory interface. The memory interface forwards the request to the next memory level and then acknowledges the request from R-control. When the cache receives the data from the higher level memory, L-control and core complete the refill operation. Then, R-control is informed by the core. R-control gives the data from the core to the CPU along with an acknowledgement.

4.2.3. *Design of core*

This is the middle part of the cache memory. It contains a 256-line array of memory. Each line is capable of holding 8 bits (One word is assumed to be 1 Byte) of data and 4 bits of tag. Thus each line is of 12 bits which results in a cache size of 3 kbits. The 256 lines are divided into four groups having sixty-four lines each. Each group is further divided into sixteen cells with each cell having four lines. This organization of the memory array as a hierarchy helps in faster access of the memory. There is a 3-stage process to access these lines. One of the four groups is identified in the first stage, cell within a group is accessed in second stage and the desired line in that cell is selected in the third stage. The various bits of the index are used for selection in each access stage.

When the core receives a request from the L-control, it finds the correct cache location using the received address. It then checks the instruction received. In case of a "read" or "check" instruction, core sends the tag from the location to the R-control. Furthermore, it sends the data from the location as well in case of a "read" instruction. Finally, it sets the value of Data_Enable signal and then, it sends a request to the R-control and an acknowledgement to L-control. In case of a "write" instruction, the core will store the data provided by L-control in the selected cache location. When the core receives a "refill" instruction, it takes the data from the memory (provided by the memory interface) and stores it sequentially in the entire cell selected in the second stage of access. Once the refill is made, it provides the desired data to R-control. In both these cases, core informs the R-control after performing the operation (storing the data in case of write or loading the required data into the data bus of R-Control in case of a refill). Then it sends an acknowledgement to L-control.

4.3. Flow of operations

While each individual module has been discussed in the above section, the detailed operation of the cache as it executes a read or write instruction is described in this section. The flow chart in Fig. 7 describes the flow of the operations.

4.3.1. Read operation

The read operation takes place in both the I-Cache and the D-Cache. It involves three main asynchronous handshaking communications in case of a cache hit and five in case of a cache miss.

When the L-Control receives the request from the CPU, it reads the values in the address bus and the instruction signal from the CPU. It checks the instruction to determine if it is a read or write operation. Once it decides that it is a read operation, it decodes the address to generate the tag and index values. It sends the tag to the R-Control and the index along with the instruction to read to the core. This involves handshaking

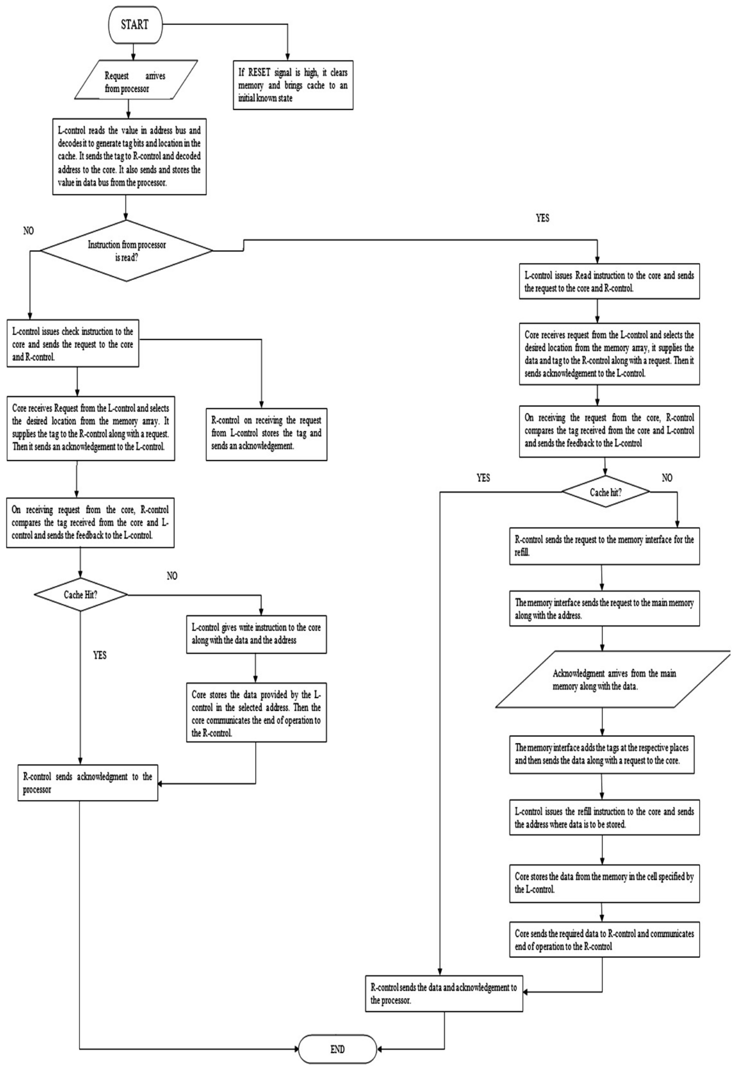

START
Request arrives from processor
If RESET signal is high, it clears memory and brings cache to an initial known state
L-control reads the value in address bus and decodes it to generate tag bits and location in the cache. It sends the tag to R-control and decoded address to the core. It also sends and stores the value in data bus from the processor.
NO
Instruction from processor is read?
YES
L-control issues check instruction to the core and sends the request to the core and R-control.
Core receives Request from the L-control and selects the desired location from the memory array. It supplies the tag to the R-control along with a request. Then it sends an acknowledgement to the L-control.
R-control on receiving the request from L-control stores the tag and sends an acknowledgement.
On receiving request from the core, R-control compares the tag received from the core and L-control and sends the feedback to the L-control.
Cache Hit?
NO
YES
L-control gives write instruction to the core along with the data and the address
Core stores the data provided by the L-control in the selected address. Then the core communicates the end of operation to the R-control.
R-control sends acknowledgment to the processor
L-control issues Read instruction to the core and sends the request to the core and R-control.
Core receives request from the L-control and selects the desired location from the memory array, it supplies the data and tag to the R-control along with a request. Then it sends acknowledgement to the L-control.
On receiving the request from the core, R-control compares the tag received from the core and L-control and sends the feedback to the L-control
YES
Cache hit?
NO
R-control sends the request to the memory interface for the refill.
The memory interface sends the request to the main memory along with the address.
Acknowledgment arrives from the main memory along with the data.
The memory interface adds the tags at the respective places and then sends the data along with a request to the core.
L-control issues the refill instruction to the core and sends the address where data is to be stored.
Core stores the data from the memory in the cell specified by the L-control.
Core sends the required data to R-control and communicates end of operation to the R-control
R-control sends the data and acknowledgement to the processor.
END

Figure 7 Flow of the read and write operations.

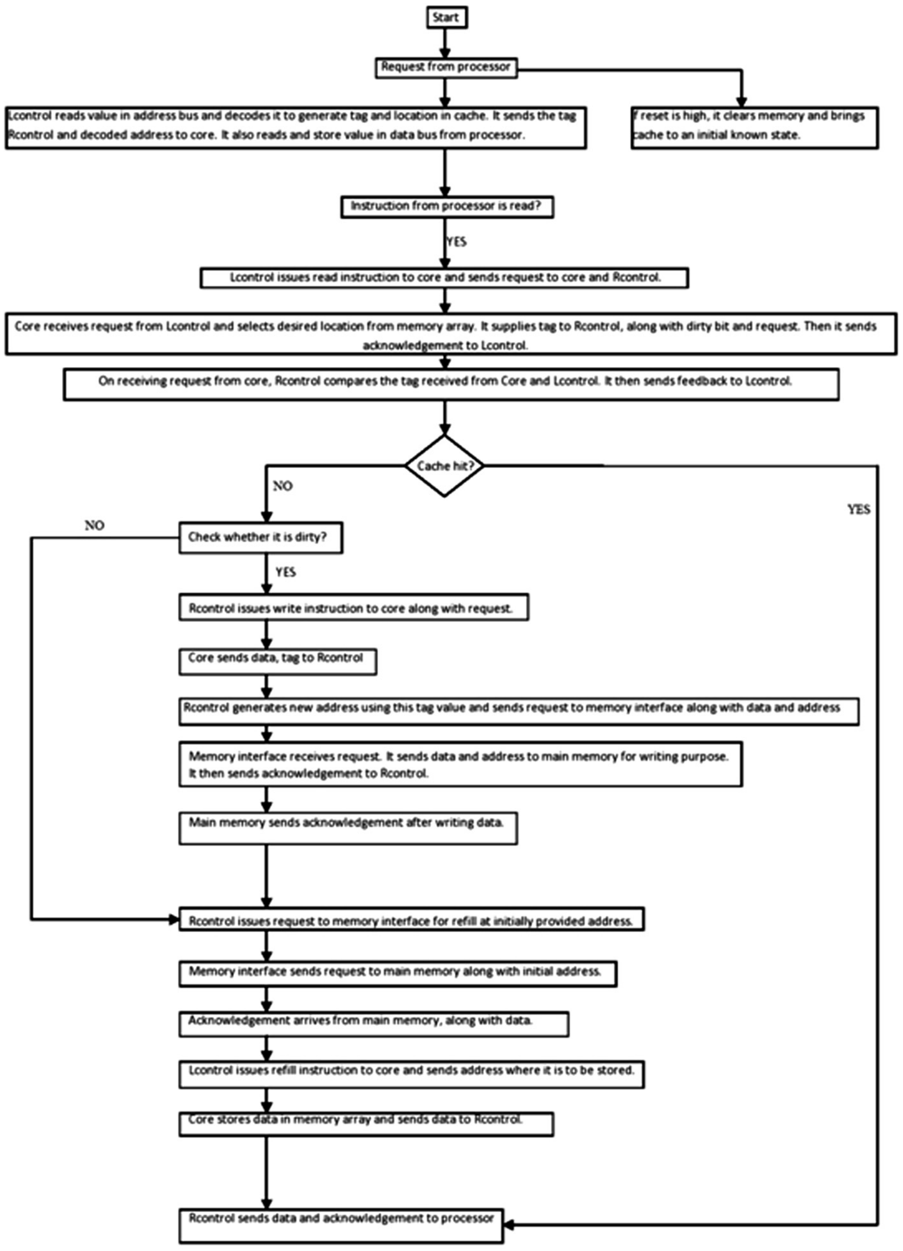

Start
Request from processor
Lcontrol reads value in address bus and decodes it to generate tag and location in cache. It sends the tag Rcontrol and decoded address to core. It also reads and store value in data bus from processor.
f reset is high, it clears memory and brings cache to an initial known state.
Instruction from processor is read?
YES
Lcontrol issues read instruction to core and sends request to core and Rcontrol.
Core receives request from Lcontrol and selects desired location from memory array. It supplies tag to Rcontrol, along with dirty bit and request. Then it sends acknowledgement to Lcontrol.
On receiving request from core, Rcontrol compares the tag received from Core and Lcontrol. It then sends feedback to Lcontrol.
Cache hit?
NO
YES
NO
Check whether it is dirty?
YES
Rcontrol issues write instruction to core along with request.
Core sends data, tag to Rcontrol
Rcontrol generates new address using this tag value and sends request to memory interface along with data and address
Memory interface receives request. It sends data and address to main memory for writing purpose. It then sends acknowledgement to Rcontrol.
Main memory sends acknowledgement after writing data.
Rcontrol issues request to memory interface for refill at initially provided address.
Memory interface sends request to main memory along with initial address.
Acknowledgement arrives from main memory, along with data.
Lcontrol issues refill instruction to core and sends address where it is to be stored.
Core stores data in memory array and sends data to Rcontrol.
Rcontrol sends data and acknowledgement to processor

Figure 8 (a) Read operation and (b) write operation for "write-back" strategy.

through the asynchronous interface between the modules. The R-Control stores the tag for future comparison. Once the core receives the request from the L-Control, it uses the index received to select the cache line from its memory array. Then it sends the tag field from the selected line to R-Control along with the data from that line. The core set the value of Data_Enable to "1". The second handshaking is performed as the core sends a request to R-Control. The R-Control compares the tag from the core with the tag received from L-Control to determine if there is a cache hit or a cache miss. Then it sends a feedback signal to L-Control to inform the results of the comparison. The third handshaking is performed here where the R-Control sends a feedback to the L-Control and L-Control acknowledges it.

In the case of a cache hit, the L-Control takes no further action. R-Control checks the value of Data_Enable signal and since this signal is set high by the core, it reads the data from the core. Then, it acknowledges the request from the core and sends the data along with an acknowledgement to the CPU.

When a cache miss occurs, R-Control sends a request to the memory interface. This is the fourth Handshaking which takes place in the case of a cache miss. The memory interface then sends a request to the next memory level along with the address. On receiving the block of data from the refill operation, the memory interface gives a signal to L-Control. This is the fifth handshaking which occurs between the cache and the memory interface. L-Control then instructs the core to perform a refill operation wherein the data from the memory bus is loaded into the specific block. Then the desired data are given to R-Control along with appropriate control signals. The R-Control then forwards the data to the CPU along with an acknowledgement.

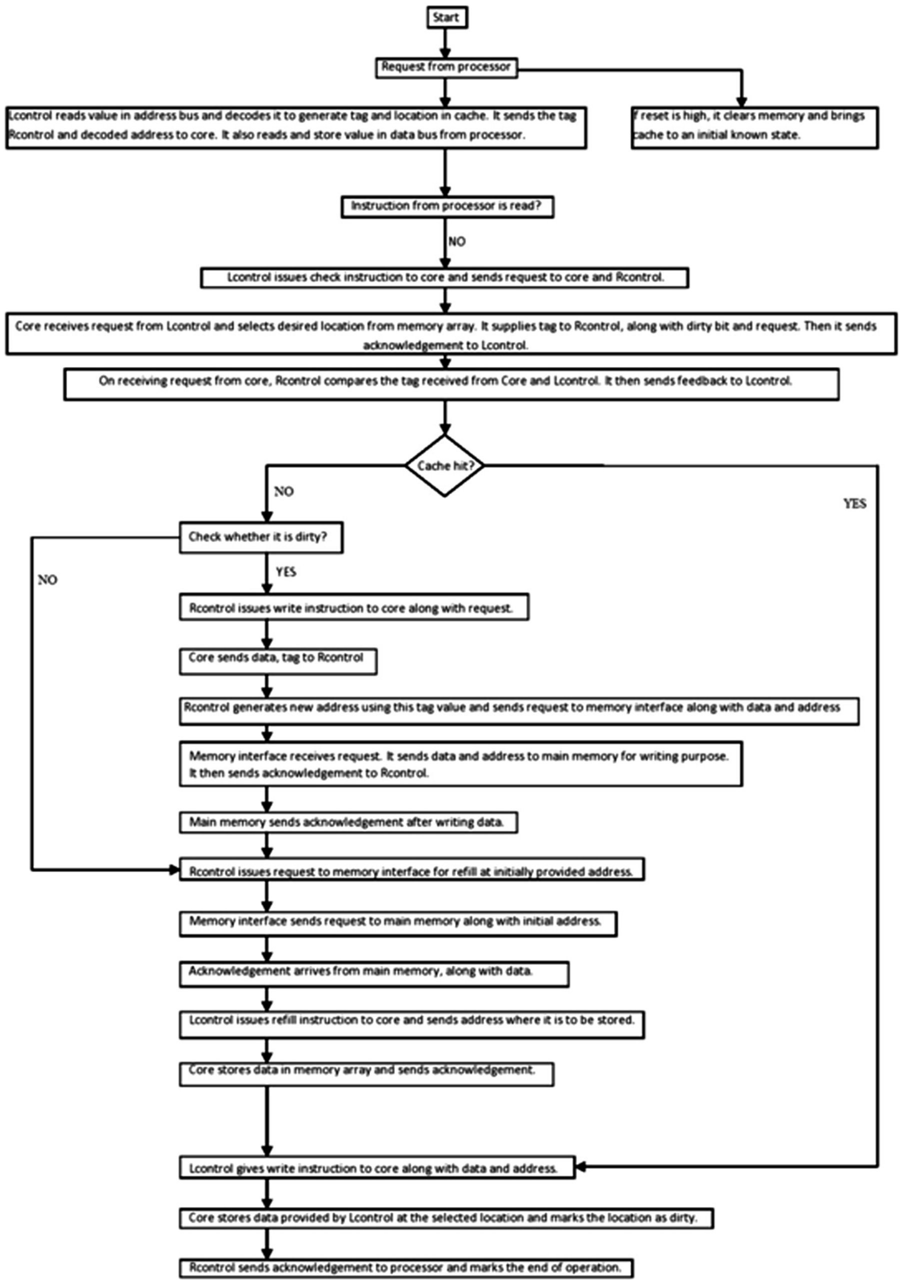

Start
Request from processor
Lcontrol reads value in address bus and decodes it to generate tag and location in cache. It sends the tag Rcontrol and decoded address to core. It also reads and store value in data bus from processor.
If reset is high, it clears memory and brings cache to an initial known state.
Instruction from processor is read?
NO
Lcontrol issues check instruction to core and sends request to core and Rcontrol.
Core receives request from Lcontrol and selects desired location from memory array. It supplies tag to Rcontrol, along with dirty bit and request. Then it sends acknowledgement to Lcontrol.
On receiving request from core, Rcontrol compares the tag received from Core and Lcontrol. It then sends feedback to Lcontrol.
Cache hit?
NO
YES
Check whether it is dirty?
NO
YES
Rcontrol issues write instruction to core along with request.
Core sends data, tag to Rcontrol
Rcontrol generates new address using this tag value and sends request to memory interface along with data and address
Memory interface receives request. It sends data and address to main memory for writing purpose. It then sends acknowledgement to Rcontrol.
Main memory sends acknowledgement after writing data.
Rcontrol issues request to memory interface for refill at initially provided address.
Memory interface sends request to main memory along with initial address.
Acknowledgement arrives from main memory, along with data.
Lcontrol issues refill instruction to core and sends address where it is to be stored.
Core stores data in memory array and sends acknowledgement.
Lcontrol gives write instruction to core along with data and address.
Core stores data provided by Lcontrol at the selected location and marks the location as dirty.
Rcontrol sends acknowledgement to processor and marks the end of operation.

Fig. 8 (*continued*)

In both the cases, the entire operation is completed within one cycle of the Request signal from the CPU. The request signal is reset to its initial state on receiving the acknowledgement from the cache. Corresponding to this, the acknowledgement signal from the cache is also reset to its initial state. The cache is once again ready for the next operation.

4.3.2. Write operation

The write operation is limited to the D-Cache alone. Our cache model employs write-through mechanism with no-write-allocation in case of a cache miss. In this mechanism, the write operation is carried out in the cache and higher memories simultaneously and if the entry is not present in the cache at that instant, it is not loaded from the higher level memory. A write operation involves four main handshaking communications for the case of cache hit and three for the case of cache miss.

The tasks done are the same as that in read operation till the R-Control sends the feedback to L-Control with only three differences. The first is that the data from CPU in the Store_Data bus are also read and passed to the core. The second is that the L-Control issues the check instruction to core. The third is that the core sets the Data_Enable signal to "0".

In case of a cache hit, the L-Control sends a "write" instruction to the core. The core reads the data to be written sent by the core and stores in the correct line of memory array. Once the data write is done, the R-Control is informed which in turn sends an acknowledgement to the processor.

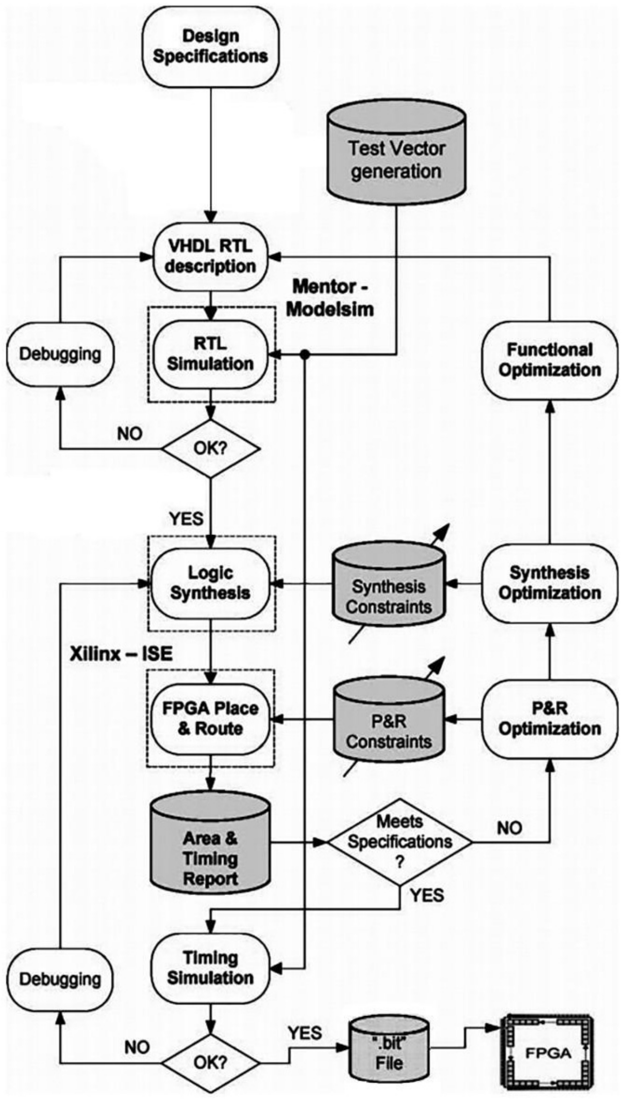


Figure 9 Front-end design flow (Deliparaschos et al., 2008).

In case of a cache miss, the policy followed is no-fetch-on-miss and no-allocation-on-miss. Thus, no changes are made to the core. The write is made in the higher memory levels simultaneously (write-through policy) and so the cache simply sends an acknowledgement to the processor through the R-control.

Cache coherency during write: The write is performed in the cache and higher memory simultaneously. During a cache hit, the cache does not accept another request until write operation is completed. The cache and higher memory contain the same data before and after the write is done. In the case of a cache miss, the cache does not contain the data and it does not fetch it. The cache is unaffected while the higher memories are updated. Thus, when the written data are fetched, the cache will fetch the recent data. So, cache coherency is maintained at all times.

Similar to the read operation, the entire write operation is carried out within a single request cycle from the CPU. The write operation requires no communication between the D-Cache and the other higher level memories.

The operations are also shown using the flowchart in Fig. 7.

4.3.3. Write-back strategy

To modify our design to write-back with write-allocation technique, we have to add an extra bit in data bus of our cache memory. Earlier there was 8 bit data and 4 bit tag in data bus, but now we have to add one extra bit called *dirty bit*.

High Dirty bit denotes that data in the cache memory have been modified, but not modified in main memory. Low Dirty bit denotes that data are not modified. Overall functionality of our design will be modified as shown in Fig. 8.

5. Design flow

The top-down design approach followed in our design is presented in Fig. 9. In this approach, the primary consideration lies on specification and on the circuit functionality. The design process starts with the system level modeling of architecture. The implemented model is then evaluated and RTL and timing simulations are performed with the test vector values. The design has been coded in Very High-Speed Integrated-Circuits Hardware Description Language (VHDL). In order to construct a fully parameterized code for the design, special care has been taken to encode the different cache blocks incorporated with the handshaking interface. The functionality of the circuit is ensured using RTL simulation. Subsequently, the logic synthesis is performed where on the basis of VHDL code the tool makes a technology-independent schematic. The circuit is then optimized to the selected FPGA specific library (SPARTAN-3A, XC3S50A). At this point, it is important to define specific design requirements, timing constraints and area as they are crucial for synthesis. Next, the input netlist file created by Xilinx ISE (Xilinx 9.1i) synthesis tool is taken by place and route tools. Thereafter following process begins: firstly, the design constraints along with the input netlist file is translated to a Xilinx database file. Subsequently, the mapping of logical design onto a Xilinx FPGA device takes place. In the final stage once place and route is performed on the FPGA, the place and route program produces the output for bitstream generator. On reception of the placed and routed design this program generates a bitstream (.bit) for Xilinx device configuration. With the help of a timing simulation it is to be ensured that the circuit meets the required RTL functionality and timing constraints, before FPGA file is programed.

6. Implementation

This section demonstrates the results of implemented cache architecture. Once the design synthesis is performed, design is translated, mapped, placed and routed onto the FPGA device by Xilinx ISE as shown in Fig. 10. The FPGA utilization given by Xilinx is shown in Table 3. The HDL coding for the implemented cache architecture is fully parameterized. This feature enhances the design flexibility and can be easily used with different specification scenarios.

7. Performance evaluation

In the design implementation, both data cache and instruction cache are made up of 256 lines of 12-bits each, four of them are used as tags. Hence, the total amounts to 3 kbits per cache. To

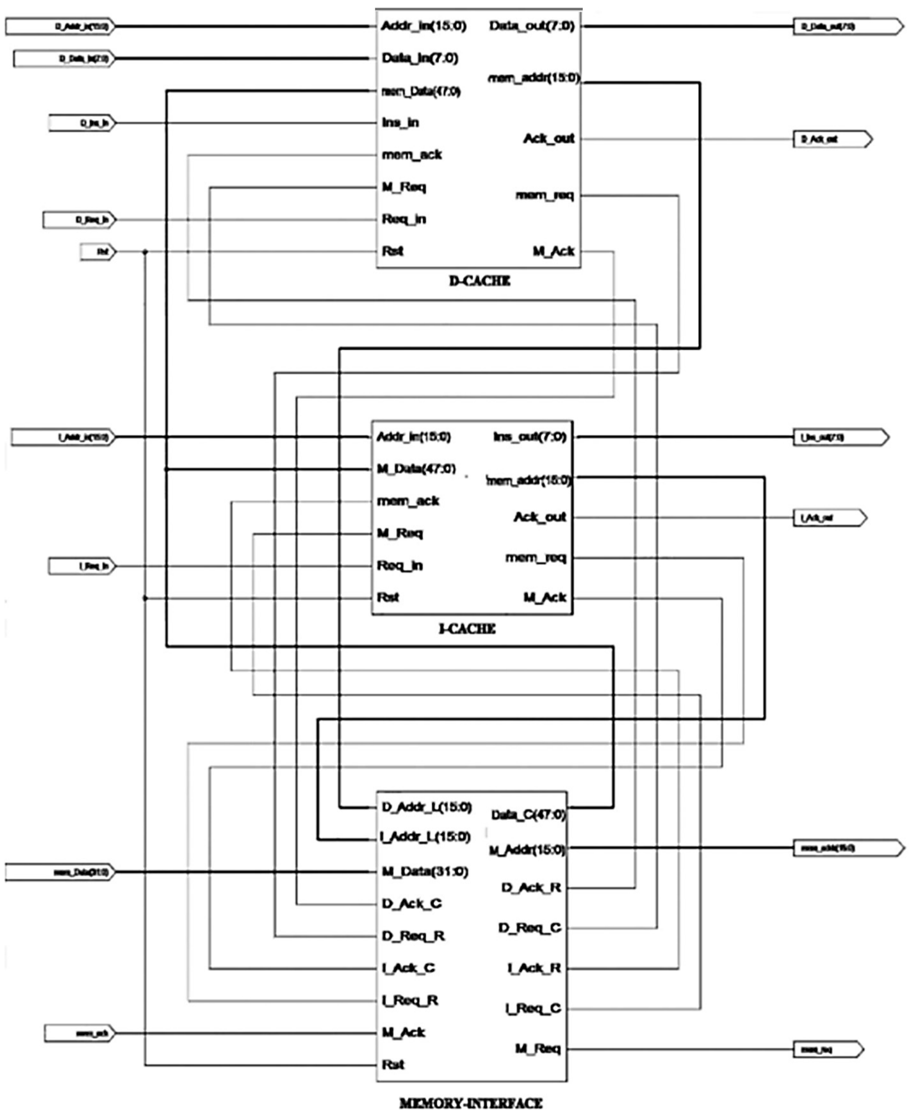


Figure 10 Architecture of (a) complete cache, (b) D-Cache and (c) I-Cache.

verify the functionality of the cache architecture, the gate-level simulations are conducted as shown in Figs. 11 and 12. The data and control signals adopted in the design of D-Cache are described in Table 4. The signals for I-Cache are similar to that of the D-Cache.

7.1. Gate-level simulation results

Fig. 11(a) shows the read operation simulations. The cache is loaded with some initial data to show a variety of scenarios. The first request shows a cache hit. During the first request (req_in goes high), ins_in is low (read operation) and address is $(0045)_{16}$. Since this address is pre-loaded, there is a cache hit and the cache provides the data to the processor. The ack_out signal goes high after data are loaded into the data bus. As req_in goes low, ack_out correspondingly goes low and thus completes the handshake. The time duration between req_in going high and ack_out going high is the read access time. The second request shows the case of a cache miss. When the req_in goes high, the cache reads the address, $(0005)_{16}$, and instruction. Since the address is not present in the cache, D_Cache sends a refill request to the memory interface. Memory interface forwards the request to the higher memory level along with the address. This can be seen as a change in mem_addr and mem_req going high. When data from memory arrive, mem_ack goes high and the handshake is completed. The correct data is then given to the processor and ack_out goes high. When the req_in goes low, then ack_out also goes low. The access time for cache miss is higher than that of cache hit case owing to the refill operations. The third request shows another cache hit case following the cache miss earlier and the last request is again a cache miss.

Fig. 11(b) shows simulations for write operations in D-Cache. First a read operation is performed with address $(4545)_{16}$. The output data is $(E5)_{16}$. Then a write operation is performed in the same location with new data being $(4E)_{16}$. Next a read operation is performed on the same address. The output data change to $(4E)_{16}$. Thus the write operation is performed successfully. The last request is that of a cache

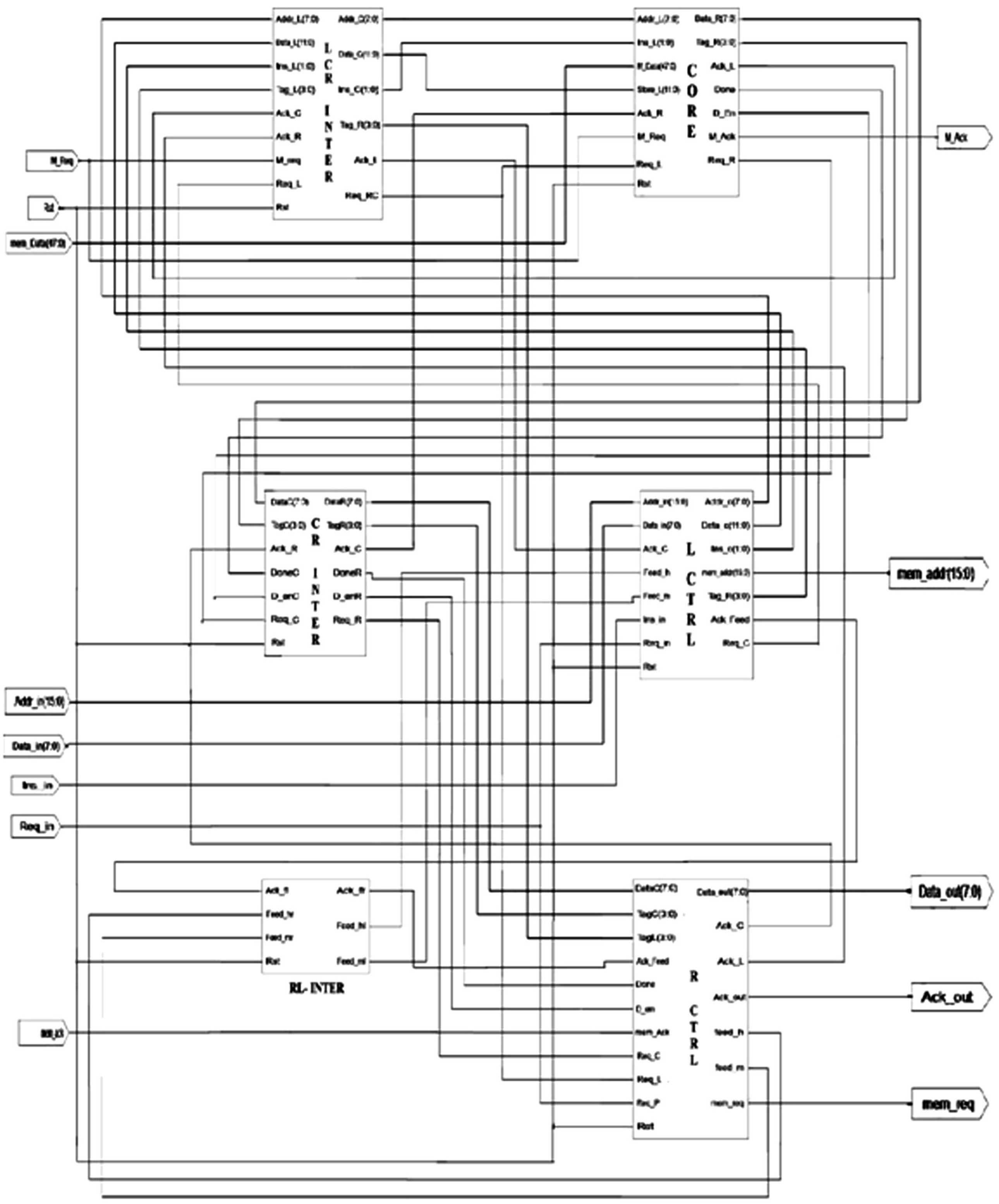


Fig. 10 (*continued*)

miss for which the cache simply provides an acknowledgement to the processor.

Fig. 12 shows the read operation simulations on I-Cache. The scenarios taken are similar to that of the D-Cache.

The read and write access times for instruction cache and data cache are shown in Table 5. It has already been stated that our implementation considers cache only, hence the main memory access time is not included.

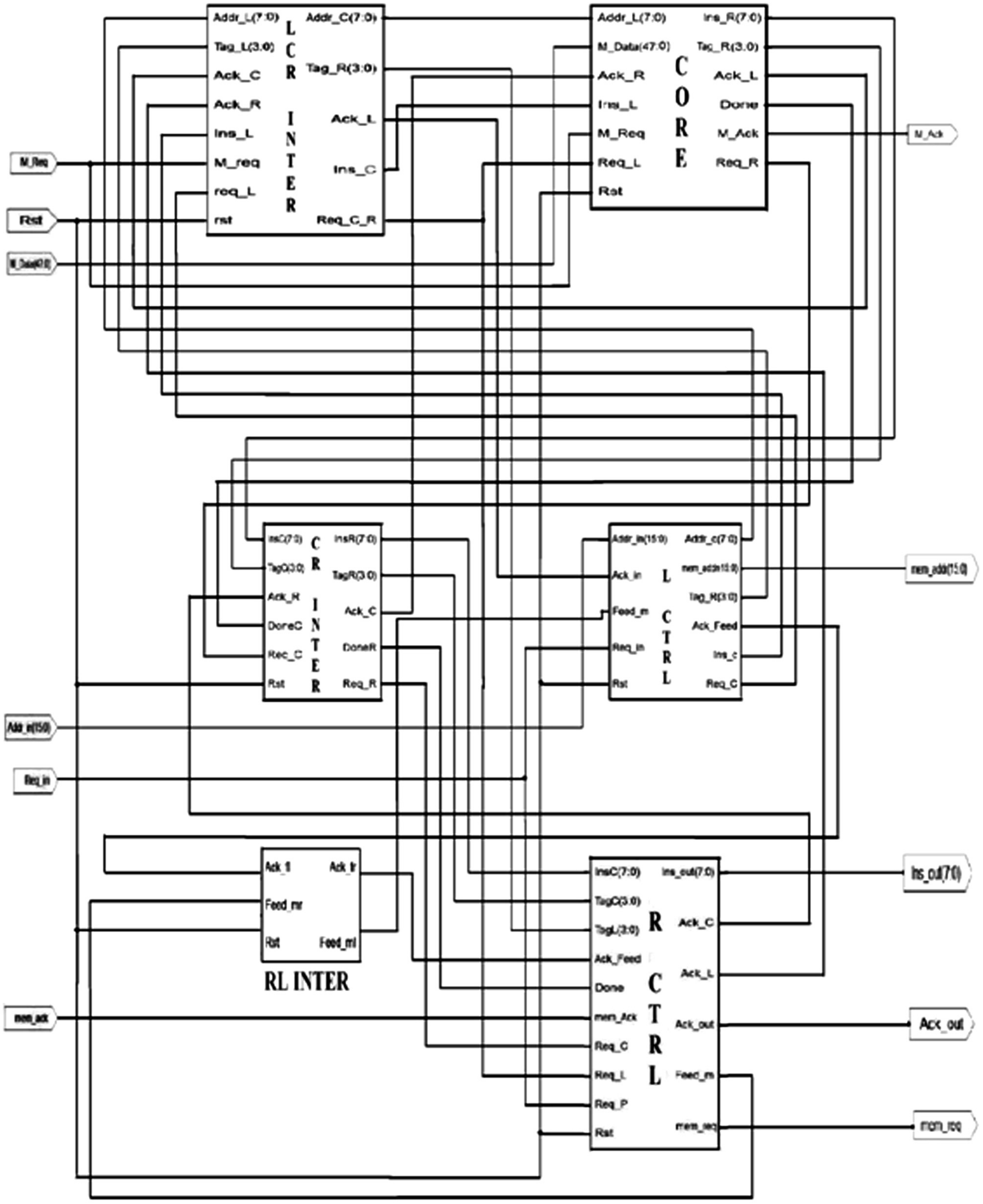


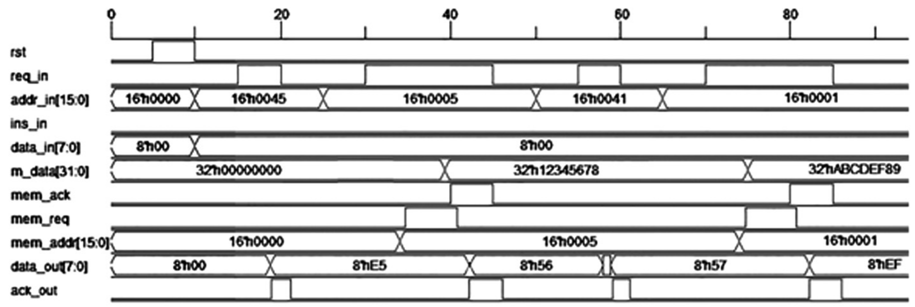
Fig. 10 (*continued*)

An external main memory is also designed which is capable of interfacing with our cache memory using asynchronous pipeline. It is found that the main memory access time is of the order of 10 ns, which is quite high as compared to the cache memory access time.

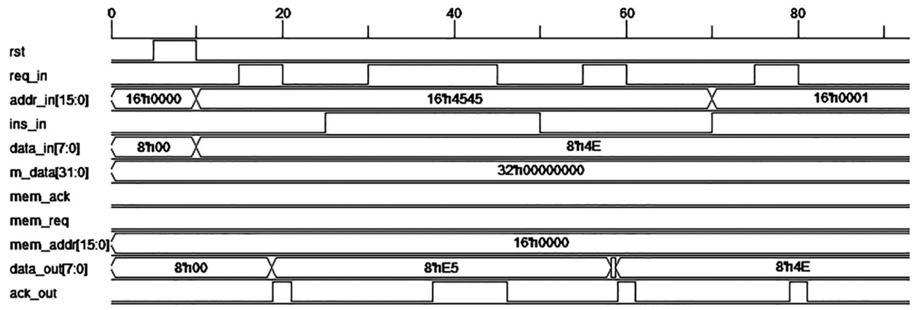
For comparing the performance of implemented micropipelined asynchronous cache architecture with the corresponding synchronous counterpart, the Table 6 presents the read and writes access times of the synchronous version.

Table 3 FPGA utilization for the implemented cache architecture.

	Used	Available	Utilization
Complete cache			
Logic utilization			
Total number of slice registers	476	1408	33%
Number used as flip flops	44		
Total number of 4-input LUTs	801	1408	56%
Logic distribution			
Number of occupied slices	503	704	71%
Number of slices containing related logic			
Only	503	503	100%
Total equivalent gate count	8378	–	–
Macro statistics			
Registers	102	–	–
Flip flops	102	–	–
Comparators	2	–	–
Multiplexers	30	–	–
D-Cache			
Logic utilization			
Total number of slice registers	244	1408	17%
Number used as flip flops	24	–	–
Total number of 4-input LUTs	439	1408	31%
Logic distribution			
Number of occupied slices	274	704	38%
Number of slices containing related logic			
Only	274	274	100%
Total equivalent gate count	4426	–	–
Macro statistics			
Registers	44	–	–
Flip flops	44	–	–
Comparators	1	–	–
Multiplexers	25	–	–
I-Cache			
Logic utilization			
Total number of slice registers	217	1408	15%
Number used as flip flops	12	–	–
Total number of 4-input LUTs	331	1408	23%
Logic distribution			
Number of occupied slices	205	704	29%
Number of slices containing related logic			
Only	205	205	100%
Total equivalent gate count	3483	–	–
Macro statistics			
Registers	26	–	–
Flip flops	26	–	–
Comparators	1	–	–
Multiplexers	5	–	–
Memory interface			
Logic utilization			
Total number of slice registers	7	1408	1%
Number used as flip flops		–	
Total number of 4-input LUTs	31	1408	2%
Logic distribution			
Number of occupied slices	21	704	2%
Number of slices containing related logic			
Only	21	21	100%
Total equivalent gate count	613	–	–
Macro statistics			
Registers	36	–	–
Flip flops	36	–	–



(a)



(b)

Figure 11 Gate level simulation for 3-kbits D-Cache (a) read operation and (b) write operation.

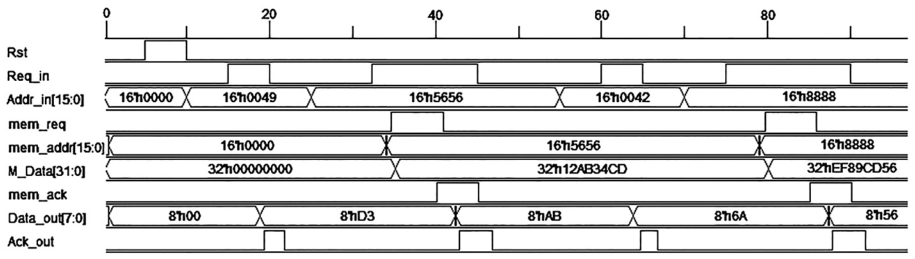


Figure 12 Gate level simulation for read operation of 3-kbits I-Cache.

Table 4 Description of signals in cache architecture pertaining to D-Cache.

Signal	Description
Rst	It is the RESET signal that brings the cache to a known initial state. It is asynchronous and no operations can be performed while this signal is high
Req_in	It is the request from the processor
Addr_in	It is the 16-bit address from the processor
Ins_in	This carries the instruction from the processor. A low value indicates a read operation while a high value indicates a write operation
Data_in	It carries a 8-bit data to be stored in the cache during a write operation
M_Data	It carries four words of data to be stored in the cache from the higher level memory at the time of refill operation
Mem_Ack	It is the acknowledgement from the higher level memory indicating that it has serviced the request from cache for a refill. The data from higher level memory is read after this signal goes high
Mem_Req	This is the request sent from the cache to the higher level memory requesting for a refill. This signal is controlled by the memory interface
Mem_Addr	This is the complete 16-bit address sent by the cache to the higher level memory for a refill
Data_out	This bus contains the 8-bit output data from a read operation for the processor
Ack_out	This is the acknowledgement sent from the cache to the processor once the specified operation has been performed

Table 5 Read and write access times.

	Data cache	Instruction cache
Read-hit (ns)	4.3	4.1
Read-miss (ns)	5.9	5.8
Write-hit (ns)	5.8	–
Write-miss (ns)	4.3	–

Table 6 Read and write access times for synchronous implementation.

	Data cache	Instruction cache
Read-hit (ns)	2	2
Read-miss (ns)	3	3
Write-hit (ns)	3	–
Write-miss (ns)	2	–

8. Conclusion

In this work, we have presented the high level implementation of L1-Cache system based on asynchronous communication oriented design styles. Each of the units of cache architecture is pipelined asynchronously using asynchronous interfaces (Fig. 4). No doubt, the enhanced pipelining increases the area and energy owing to the increased control circuitry required for handling enormous number of handshakes, we have saved considerable area by omission of clock generation, distribution and gating. The implemented cache model is configurable in terms of many parameters, e.g. line count, line size etc. This parameterization permits the implemented model to be adapted to any problem specification without changing the developed VHDL code. The results reveal that the design effectively achieves remarkable average case performance with a relatively small design complexity when compared to the synchronous counterparts. Though the asynchronous design has an inherent advantage of automatic power down, our work primarily focuses on implementing the asynchronous cache with enhanced simplicity from the synchronization perspective.

References

Albonesi, D., 2000. Selective cache ways: on-demand cache resource allocation. J. Instr.-Level Parallelism 2.

Anderson, J., Najm, F., 2004. Power Estimation Techniques for FPGAs. VLSI Syst. 12 (10), 1015–1027.

Bahar, R., Albera, G., Manne, S., 1998. Power and performance tradeoffs using various caching strategies. In: Proceedings international symposium on low power electronics and design, pp. 64–69.

Deliparaschos, K.M., Doyamis, G.C., Tzafestas, S.G., 2008. A parameterised genetic algorithm IP core: FPGA design, implementation and performance evaluation. Int. J. Electr.

Elrabaa, M.E.S., 2012. A new FIFO for transferring data between two unrelated clock domains. Int. J. Electr. 99 (8), 1063–1074.

George, V., Zhang, H., Rabaey J., 1999. The design of low energy FPGA. In: International symposium on low power electronics and design, pp. 183–193.

Hac, A., 1993. Performance and reliability improvement by using asynchronous algorithms in disk buffer cache memory. Acta Inform. 30 (2), 131–146 (Springer-Verlag).

Jin, L., Cho, S., 2006. Reducing cache traffic and energy with macro data load. ISLPED 2006, 147–150.

Jouppi, N.P., 1993. Cache write policies and performance. In: Proceedings of the 20th annual international symposium on computer architecture, pp. 191–201.

Kin, J., Gupta, M., Mangione-Smith, W., 1997. The filter cache: an energy efficient memory structure. MICRO 1997, 184–193.

Lee, H., Smelyanskiy, M., Newburn, C., Tyson, G., 2001. Stack value file: custom microarchitecture for the stack. HPCA 2001, 5–14.

Liljeberg, P., Tuominen, J., Tuuna, S., Plosila, J., Isoaho, j., 2004. Self-timed approach for noise reduction in NoC. In: Interconnect-centric design for advanced SoC and NoC. Kluwer Academic Publishers (Chapter 11).

Milenkovic, A., Milenkovic, M., Barnes, N., 2003. A performance evaluation of memory hierarchy in Embedded systems. In: Proceedings 35th southeastern symposium system theory, pp. 427–431.

Nyström, M., Lines, A.M., Martin, A.J., 2003. A pipelined asynchronous cache system. Technical Report (revised 2005).

Patterson, D.A., Hennessey, J.L., 2003. Computer Architecture A Quantitative Approach, 3rd ed. Morgan Kaufmann Publishers, San Francisco, CA, USA.

Peeters, A., 1996. Single-rail Handshake Circuits. Eindhoven University of Technology, (PhD thesis).

Peir, J.-K., Hsu, W.W., Smith, A.J., 1998. Implementation issues in modern cache memories. IEEE Trans. Comput. 48 (2).

Putnam, A., Eggers, S., Bennett, D., Dellinger, E., Mason, J., Styles, H., Sundararajan, P., Wittig, R., 2009. Performance and power of cache-based reconfigurable computing. In: Proceedings International symposium on computer architecture, pp. 395–405.

Racunas, P., Patt, Y.N., 2003. Partitioned first-level cache design for clustered microarchitectures. ICS 2003, 22–31.

Reinman, G., Jouppi, N.P., 1999. CACTI2.0, An integrated cache timing and power model. COMPAQ Western Research Lab.

Ross, A., Vahid, F., Dutt, N., 2004. Automatic tuning of two-level caches to embedded applications. Des. Autom. Test Eur. 1, 102–108.

Ross, A., Vahid, F., Dutt, N., 2005. Fast configurable-cache tuning with a unified second-level cache. In: IEEE/ACM international symposium on low power electronics and design.

Segars, S., 2001. Low power design techniques for microprocessors. In: ISSCC tutorial note.

Shang, L., Kaviani, A., Bathala, K., 2002. Dynamic power consumption in the Virtex-II FPGA family. In: Proceedings ACM international symposium field-programmable gate arrays, pp. 157–164.

Tennenhouse, D., 2000. Proactive computing. Commun. ACM 43 (5), 43–50.

Warrier, T.S., Anupama, B., Mutyam, M., 2013. An application-aware cache replacement policy for last-level caches, ARCS 2013, LCNS Volume 7767. Springer-Verlag, Berlin Heidelberg, pp. 207–219.

Zhang, Chuanjun, Vahid, Frank, Najjar, Walid, 2003. A highly configurable cache architecture for embedded systems. In: 30th ACM/IEEE International symposium on computer architecture.

Zhang, Chuanjun, Vahid, Frank, Najjar, Walid, 2003. Energy benefits of a configurable line size cache for embedded systems. In: Proceedings of the IEEE computer society annual symposium on VLSI (ISVLSI'03), 87.

Zhang, Chuanjun, Vahid, Frank, Najjar, Walid, 2005. A highly configurable cache for low energy embedded systems. ACM Trans. Embedded Comput. Syst. 4 (2), 363–387.