

Synchronisation barriers on the Anemone processor

April 2011

Paralant Ltd. (www.paralant.com)

1 Abstract

Barrier functionality is required for a generic MIMD programming model to ensure the independent programs running on the separate cores synchronise at critical points in the application. This paper describes a multi-core barrier implementation for BittWare's Anemone floating-point FPGA co-processor that has 16-core Epiphany cores on an eMesh. The barrier is implemented by creating software dependencies between the multiple cores in a tree structure. A tree structure minimises the latency of the barrier by minimising the number of layers the barrier signals have to pass through.

2 Introduction

Barrier synchronisation provides essential functionality for multi-threaded, multi-core and multi-processor applications. Barriers are used to ensure that the threads are in a known state between sections of the application; and only continue when it is safe to do so. One such critical section of multi-threaded systems is the initialisation section. It is common to see one system thread tasked with initialising external interfaces and configuring shared resources such as the external memory and IO devices. It can be unsafe to allow the other threads to proceed until the initialisation has completed. A barrier provides a method for synchronisation across all or just some of the threads.

Programming models, such as OpenMP, rely on barriers to provide synchronisation over the threads participating in a common task. OpenMP segments a task across the available threads, each thread performs its given operations then waits at a barrier to prevent race conditions where there are inter-dependencies between the threads.

The Insight library implements the barrier functionality using software logic and the global memory map to signal the core barrier states through a virtual tree interconnect structure. There are two important factors to consider when implementing a multi-core barrier using a shared memory architecture; (1) signalling latency, and (2) the bandwidth used. This paper considers both of these key factors for the Anemone processor.

3 Processor architecture

3.1 Epiphany mesh architecture

The Epiphany architecture provides a grid of compute cores, as shown in Figure 1. These cores are interconnected via an IO mesh, which allows each core to access every other core's local memory and any off-chip memories.

The mesh provides 8 GBytes per second full duplex bandwidth between each core router; that is 8 GBytes per second in both directions. The data is passed around the mesh according to the destination coreIDx and coreIDy coordinate, where each router resolves the coreIDx coordinate first, then the coreIDy by passing the data out through the most relevant port. The router has a round-robin scheduler so that each input port will not be blocked for longer than 4 cycles.

The off-chip data transfers pass through one of the four external bridges; which of the four bridges (north, south, east or west) is determined by the destination coreIDx and coreIDy address coordinates. The external bridge has multiple cores accessing a one 1 GByte per second port. There is an understandable bottleneck in the off-chip bandwidth; therefore a tight control over the off-chip data transfers will provide better performance. Off-chip data is best read once, then shared between the multiple DSP cores as appropriate.

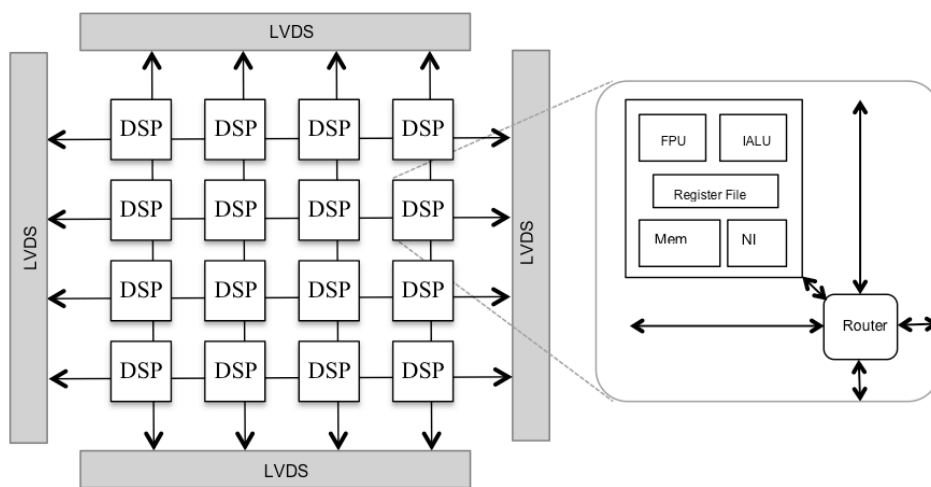


Figure 1 Epiphany processor mesh of cores

4 Barrier implementation

4.1 Barrier tree structure

The Insight barrier implementation operates using a software tree interconnection. One core acts as the barrier master and this core lives at the root of the tree. The barrier master can synchronise directly with up to four other cores. Each of these cores can also synchronise with a further four cores; this provides us with a tree-like structure. Once a core has hit the barrier, it will stop until all of its dependant cores are in a waiting-state before entering that state itself. When the barrier master has all of its dependants in the waiting-state, all cores must be in the waiting-state. The barrier master can send a go signal that propagates down to all other cores.

To achieve best bandwidth utilisation, the dependent cores should be neighbouring cores. Any signalling between cores is performed without any mesh router hops. The barrier tree depicted in Figure 2 shows how to interconnect the cores to achieve the minimum depth whilst keeping all dependents as neighbouring cores.

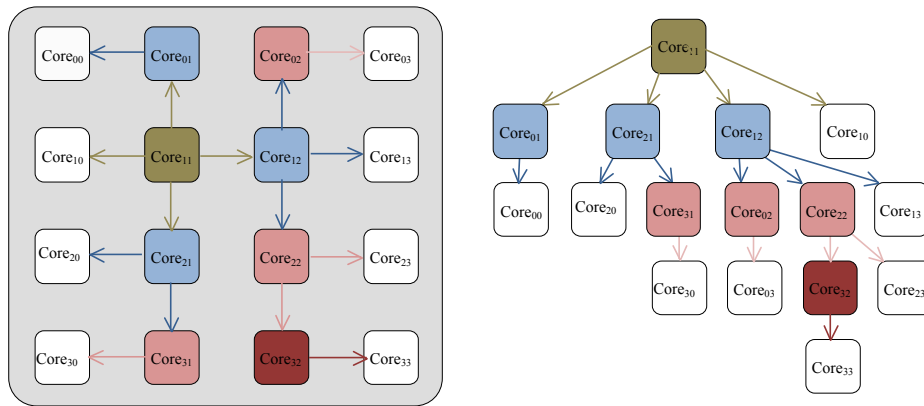


Figure 2 Barrier tree interconnections with core₁₁ acting as the barrier master. The barrier tree has a depth of 5 and a core dependent is always a neighbouring core to reduce the mesh bandwidth overhead.

To achieve minimum barrier latency, the depth of the tree should be kept to a minimum. To gain the minimum depth the cores with dependencies should connect with as many cores as possible. In the Epiphany implementation a branch core can connect to as many as four other cores. Figure 3 shows a barrier tree with a depth of 3. To achieve this some of the barrier signals require at least one router hop.

The Insight barrier implementation keeps the signalling between the cores to a minimum. When a core is waiting for its dependants to enter the waiting-state, it polls a local memory address. No mesh traffic is created as part of this polling. When a core enters the waiting-state it will inform its barrier-parent of its state by writing one word over the mesh. In the waiting-state a core will poll a local memory address until the go signal is given.

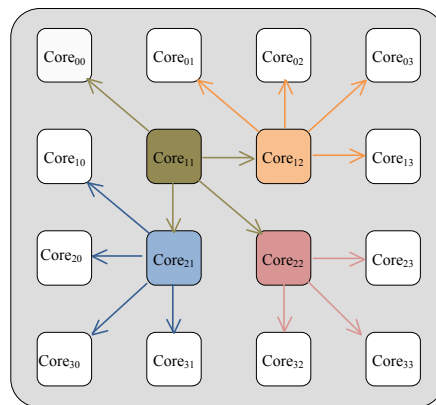


Figure 3 Barrier tree interconnections with core₁₁ acting as the barrier master. The barrier tree has a maximum depth of 3 with signalling requiring mesh router hops.

4.2 Barrier initialisation

The barrier initialisation function takes an attribute parameter that defines the barrier tree. One core must be set as the master as 1; all other cores must set the master as 0. The initial release only allows one barrier to be initialised, however the id parameter will allow multiple barriers to be created in future releases.

```
struct barrierattr_t {
    unsigned int master;
    unsigned int id;
};
```

```
        unsigned int coreid[4];  
    };
```

The last parameter, `coreid`, allows the user to define the barrier tree structure. In the barrier initialisation, the barrier master will synchronise with up to four cores. Once a non-master core is synchronised with its parent core it will then proceed to synchronise with each of its dependent cores. A core does not need to know who its barrier-parent is, the core will wait until its barrier-parent makes contact with it.

When a core has synchronised with each of its dependent cores or when the core has no dependents, it will “hit” the barrier. Once all cores hit the barrier the barrier master will send out the go signal and all cores will proceed with the application code.

5 Summary

The barrier functionality on a multi-core system, such as the Epiphany mesh, is an essential component for synchronisation at critical points in the application code. The barrier implementation needs to be low latency, robust and create the smallest amount of inter-core data traffic. The Insight implementation of a barrier is robust and uses a minimum of inter-core data traffic. When a core is waiting at the barrier it only polls local memory addresses. A one-word waiting-state flag is passed up the barrier tree from the barrier leafs through to the barrier master, and a one-word go-signal is passed down the barrier tree from the barrier master to the barrier leafs.

The barrier latency depends on the user-definable tree structure. The barrier tree depth is the number of cores the go-signal has to pass through between the barrier master and the barrier leafs. The latency also depends on the bandwidth of the connections and distance that the signals have to travel. The Insight barrier will operate with any core in the system memory map, this includes cores over several processors. The bandwidth decreases and distances increase when cores across multiple processors are connected

6 Further reading

1. Epiphany Architecture Reference Manual
2. <http://openmp.org/mp-documents/ntu-vanderpas.pdf>
3. Mellor-Crummey, J. M. and Scott, M., "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors ", ACM Transactions on Computer Systems, Feb. 1991.
4. Umakishore Ramachandran, Gautam Shah, S. Ravikumar, and Jeyakumar Muthukumarasamy. "Scalability Study of the KSR-1", Parallel Supercomputing, Vol 22, 1996, 739-759.
5. Martha Torres, Sergio Takeo Kofuji, “The Barrier Synchronization Impact on the MPI-Programs Performance Using a Cluster of Workstations”, 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97)

Paralant Ltd.

Copyright 2011 Paralant Ltd. The information contained herein is subject to change without notice.
Paralant shall not be liable for technical or editorial errors or omissions contained herein.
All marks are the property of their respective owners.