Cache Coherent Interconnect Network

Albert Chiou and Mat Laibowitz

Group 6

Introduction

In a shared memory multiprocessor system where each processor has its own data memory cache, care must be given to make sure that each processor receives good data from its cache, regardless of how other processors may be affecting that memory address.

When the following two conditions are met, the cache correctly handles the memory accesses across the multiple processors, and is called cache coherent:

- A value written by a processor is eventually visible by other processors
- When multiple writes happen to the same location by multiple processors, all the processors see the writes in the same order

There are two general categories of cache coherency protocols: directory based protocols, where a single location keeps track of the status of memory blocks; and snooping protocols, where each cache maintains the status of memory blocks and monitors activity from the other processors' caches, updating the status of the memory blocks if necessary. The protocols that we will implement and investigate for this project will be snooping protocols.

A protocol is further specified as either write-invalidate, where a write to a memory location by one processor invalidates the corresponding cache line, if present, in the other processors' caches; and write-update, where the other processors update the data in their caches rather than invalidating them.

The protocol we will implement is a three state write-invalidate snooping protocol that uses write-back caches. The three states that a cache line can be in at any one time are Modified, Shared, and Invalid. When a processor wants to write to a memory block, it must have exclusive access to that block. It obtains this exclusive access by sending an exclusive write request to the other caches, which invalidates their copy of the memory upon receipt of this request. An exclusive cache line is considered to be in the modified state because it has just been written to. When another processor reads a modified memory location from the bus, it is demoted to shared, indicating that it exists in multiple caches. When a modified memory value becomes shared or invalid, the new data must be "written back" to main memory, hence the name "write-back cache."

Implementation Topology

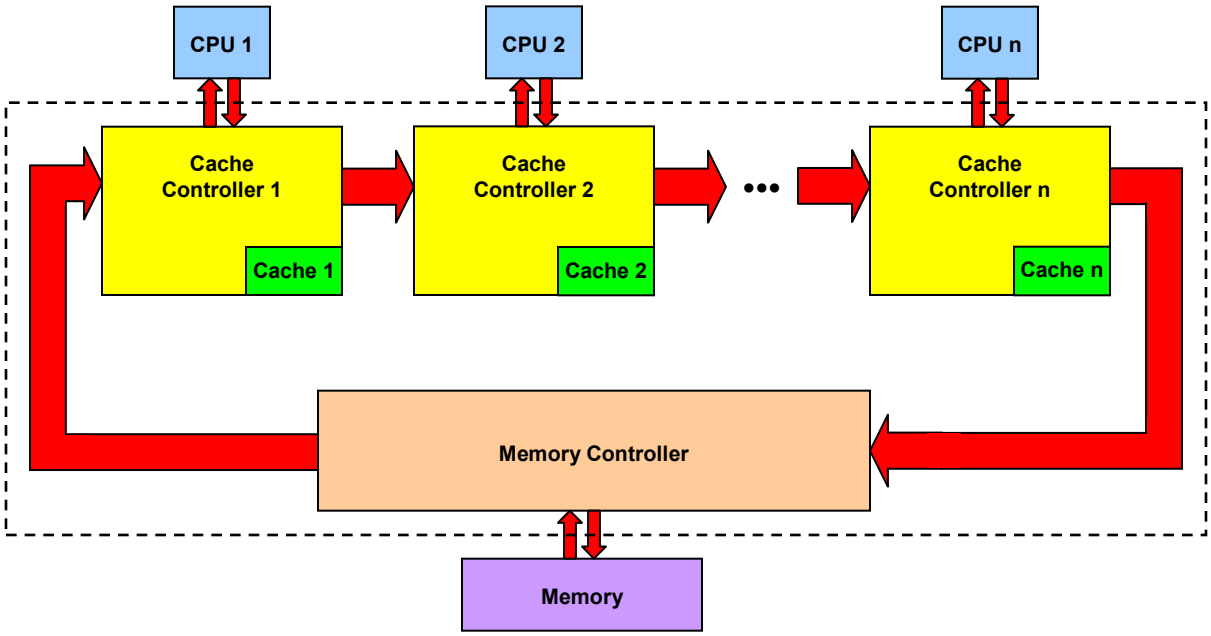

Fig. 1 Block diagram depiction of the hardware

Snooping protocols, like the MSI protocol just described, require certain messages to be sent from a cache controller to all the other cache controllers and the data memory. Classically, this would be implemented as a shared bus, but to in hopes of an increase in speed, we chose to use point-to-point communication over a ring topology. A point-to-point communication protocol will not only leverage the transactional and highly paralleled design capabilities of the Bluespec language, but will allow for much higher data transfer bandwidth between processors. Like a shared bus architecture, a ring still needs some arbitration to prevent multiple messages from being placed on the ring simultaneously and creating situations where the caches can lose their coherency. We solve this problem by passing a token along the ring from processor to processor. When a cache controller receives the token, it checks to see if there is a memory request for it to execute. If there is one, it checks the cache, updates the state if necessary, and sends out the required ring message to the other cache controllers. When the transaction completes and the message returns, if no message was needed or if there was no pending memory request, the cache controller sends the token on to the next cache controller. If a cache controller does not have the token, it is still required to handle incoming ring messages.

Protocol Implementation

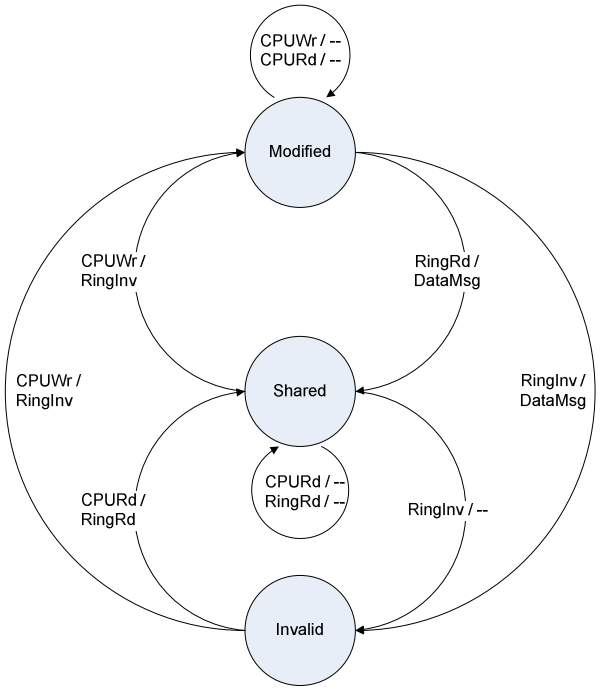

Fig. 2 State diagram for the MSI protocol on a bus architecture

Figure 2 above shows the state machine for the MSI protocol introduced in the first section. This state machine, intended for a bus architecture, will have to be adapted for our ring architecture. Each cache line of each cache of each cache controller has its own copy of such a state machine. However, our first implementation will only have one cache line, so we will only need one state machine per cache.

Write-Miss Case

When a processor makes a write requests and the cache line is in the Invalid state, the cache controller sends a write message to all the other caches. If they have this memory address in their cache they go to the Invalid state and send the data value back to the original cache. When the cache receives it, it sends a WriteBack message around the ring to update main memory in the case that one of the other caches had that data in the Modified state. The cache then goes into the Modified state, as it has the only valid copy of that memory address (with a write-back protocol, the cache contains the most recent data, which it writes back to main memory when required).

If a write is requested and the cache does not have the memory address at all (i.e. a cache miss has occurred), the cache must replace some other cache line that is valid. The method for choosing this "victim" cache line is called a replacement policy. The replacement policy is straightforward in the case of a one line cache, as there is only one line to choose from. However, in multi-line cache, the "victim" we choose can be any of the cache lines. To simplify things, we will be using a direct-mapped cache, in which each memory address can only be stored in one specific place in the cache. In the case of a cache miss, then, we simply choose the

current entry in the cache line that the memory address is associated with. If the "victim" line is in the Invalid or Shared state, then the cache controller can treat the write-miss as a normal write because main memory still has a "good" copy of the data at that memory address. However, if "victim" line is in the Modified state, the cache controller must first send a WriteBack message to update the contents of main memory.

Read-Miss Case

When a processor requests a read and the cache line is Invalid or a cache miss occurs, the cache controller sends a read message around the ring. If another cache has a line with that memory address in a valid state, either Shared or Modified, it puts its data on the message and transitions to the Shared state. When the message returns to the cache controller who sent the request, it puts a WriteBack message on the ring in case some other cache had that memory address in the Modified state. The WriteBack eventually passes through main memory, which updates its contents, and when it arrives back at the original cache, finishing the read operation and allowing the processor to pass the token.

Read-Hit Case

Any time a processor requests a read and its cache has that memory location stored in either the shared or modified state, the cache controller can simply return the value to the processor without changing the state of that cache line or sending any ring transactions.

Write-Hit Case

If the processor requests a write to a memory address that is present in the cache in the modified state, the value is updated, but no state change or ring communication is required as it has exclusive access to that memory address. If the cache line for that memory address is in the shared state, the cache controller has to send a write message around the ring to tell the other cache controllers to invalid their memory, if they have it. The cache executing the request transitions to the Modified state, as it now has exclusive access. It is useful to point out that the fact that the original cache had the memory address in the Shared state means that no other cache could have that memory address in the Modified state, so we do not need to worry about write-backs.

Unit Transaction Model

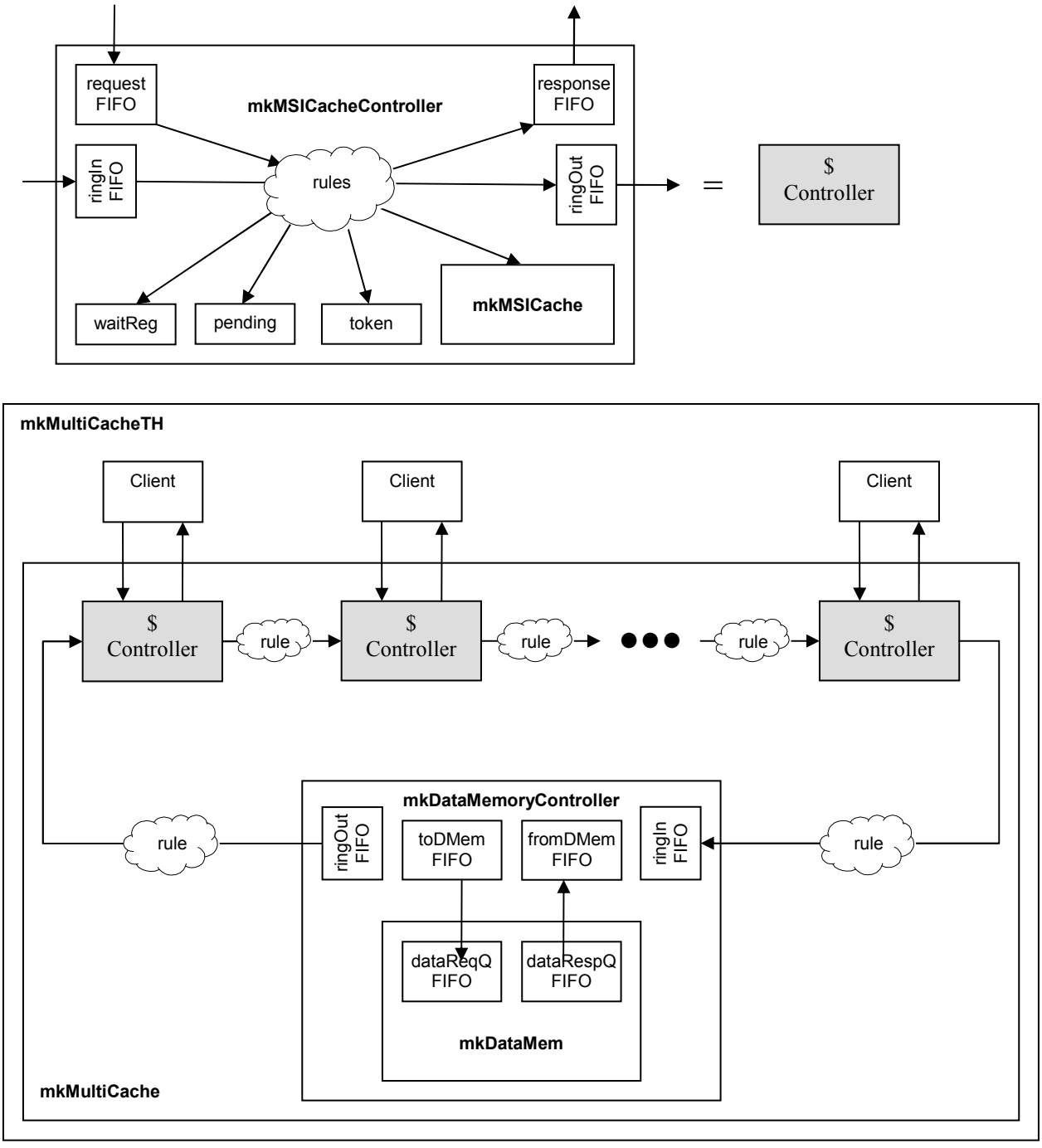

Fig. 3 Diagram of the module hierarchy

Figure 3 above shows in detail the different modules and their relationships. Each cache controller has an input and output FIFO to handling requests to and from the processor, as well as an input and output FIFO to handle ring messages. Rules are needed to move messages from

the output FIFO of one cache controller on the ring to the input FIFO of the next cache controller. Similar rules move messages into and out of the FIFOs of the memory controller. As indicated by the ellipses in the figure, any number of caches (and thus "processors") can be added to the ring in any position. The transaction model diagramed and the state transition table shown (Table 1) are still valid with any number of cache controllers and any size cache.

Cache State	Pending State	Incoming Ring Transaction	Incoming Processor Transaction	Actions
I & Miss	0	-	Read	Pending->1; SEND Read
I & Miss	0	-	Write	Pending->1; SEND Write
I & Miss	0	Read	-	PASS
I & Miss	0	Write	-	PASS
I & Miss	0	WriteBack	-	PASS
I & Miss	1	Read	-	DATA/S->Cache; SEND WriteBack(DATA)
I & Miss	1	Write (I/S)	-	DATA/M->Cache, Modify Cache; SEND WriteBack(DATA)
I & Miss		Write (M)		DATA/M->Cache, Modify Cache; SEND WriteBack(DATA), SEND WriteBack(data), Pending->2
S	0	-	Read(Hit)	-
S	0	-	Write	Pending->1; SEND Write
S	0	Read(Hit)	-	Add DATA; PASS
S	0	Read(Miss)	-	PASS
S	0	Write(Hit)	-	Add DATA; Cache->I & PASS
S	0	Write(Miss)	-	PASS
S	0	WriteBack	-	PASS
S	1	Write	-	Modify Cache; Cache->M & Pass Token
S	1	WriteBack	-	Pending->0, Pass Token
M	0	-	Read(Hit)	-
M	0	-	Write(Hit)	-
M	0	Read(Hit)	-	Add DATA; Cache->S & PASS
M	0	Read(Miss)	-	PASS
M	0	Write(Hit)	-	Add DATA; Cache->I & PASS
M	0	Write(Miss)	-	PASS
M	0	WriteBack	-	PASS
M	1	WriteBack	-	Pending->0 & Pass Token
M	2	WriteBack	-	Pending->1

Table 1 State transition table for a single cache line

In each cache controller there are several state components in the form of registers. The token register indicates if the controller is currently holding the token. When a controller has the token, it has exclusive access to the ring and may generate ring transactions. The pending register indicates various if the controller is still processing a memory request and is waiting for a message to return from the ring. Note that there may be in some cases more than one pending

state, as often multiple ring transactions are required to complete a memory request. The waitReg register is a temporary holding place for the memory request currently being executed. The final component of the cache controller is the actual cache, which consists of one or more lines of cache, which in turn contains registers for state (M, S, or I), address, and data. In multiline cache, an extra entry for age must be included, for use with the LRU replacement policy. Some different types of caches can be seen below in Figure 4.

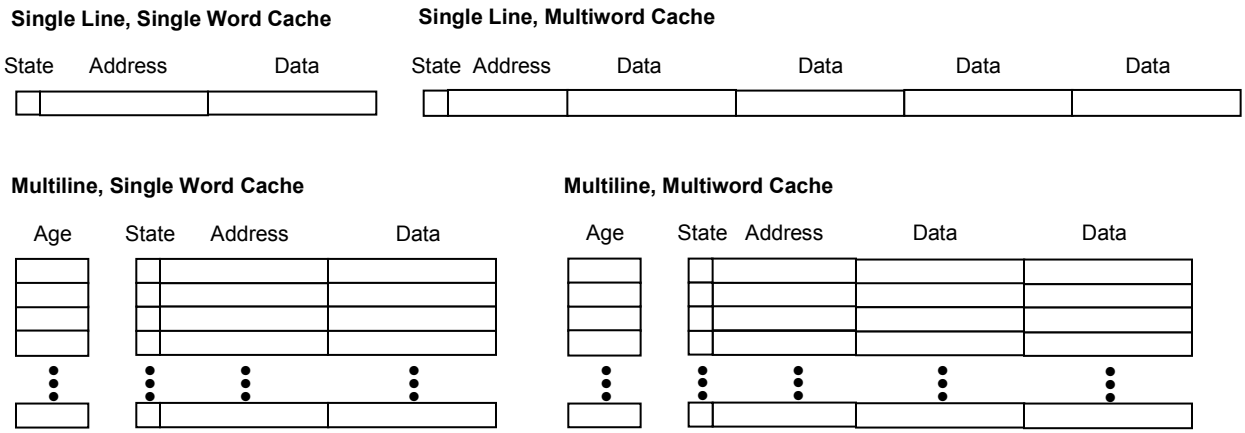

Figure 4 Different types of cache and the data fields they contain

Each component is capable of communicating in a number of ways. A processor can issue either a memory read, which contains an ID tag and a memory address, or a memory write, which contains an ID tag, a memory address, and data. In response to processor reads, the cache controller can send a memory response back to the processor, which contains an ID tag and data.

The possible ring transactions are Read, Write, WriteBack, and Token. Read, Write, and WriteBack transactions are generated by cache controllers in response to processor requests. When Read and Write transactions pass through the memory controller, it will add data to them if their "datavalid" bit is set to zero. If the "datavalid" bit is set to one, the memory controller will simply pass the transaction on. WriteBack transactions should always contain valid data, so the data contents of all WriteBack transactions are added to main memory when it passes through the memory controller.

The Token transaction can be generated by the cache controllers or the memory controller, and is used to pass control of the ring from one cache controller to the next. The memory controller needs to be able to generate a Token transaction in the order to pass the Token from the last member of the ring to the first member. It is also responsible for generating the initial Token transaction, since all cache controllers have their token registers initialized to a zero value.

Design Exploration

The first design space exploration we will perform is to go from a one-line cache to a multi-line cache. We will examine the effects that this has on the state machine and overall complexity, and then investigate its effect on the ring traffic. Aforementioned, our multi-line cache will be direct-mapped. Our current implementation of cache uses a register file rather than an SRAM, as there were problems with the memory generator.

The next design space exploration we will perform is varying the number of controllers and tokens on the ring. Adding cache controllers increases the size of the ring, meaning longer average access times since ring transactions have a longer distance to go. As a test of scalability, we would like to see how the average access time is hurt by adding more cache controllers.

After designing out protocol, we realized that much of the time, the ring is not being fully utilized. So, we decided to explore the effects of putting multiple tokens on the ring instead of one. Each token would allow the processor to perform a ring transaction on a particular set of memory addresses. This way, more transactions can be going around the ring at the same time, and we can take more advantage of the fact that processors communication on different ring segments is disjoint.

Testing

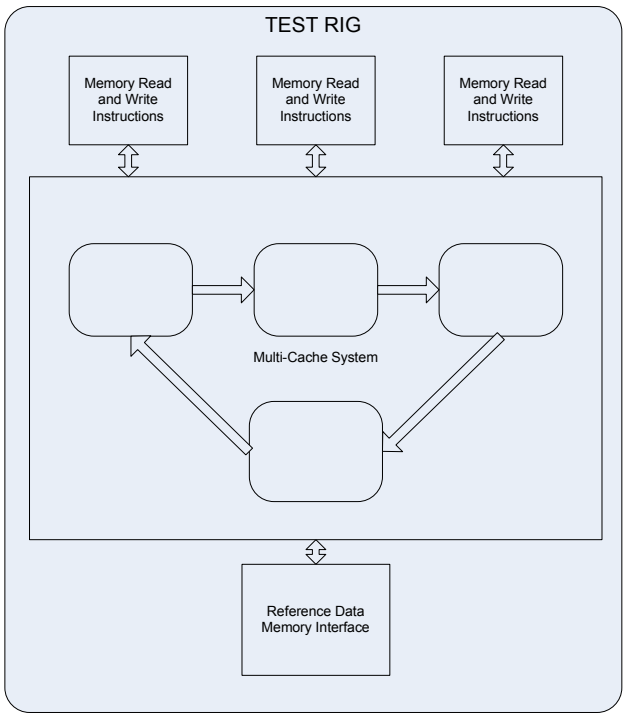

Figure 5 Verification test rig

For our project we actually used two test rigs. The first test rig, used for verification of cache coherence and sequential consistency, was a test harness that would multiplex the signals

across the processor, waiting for each operation of each virtual CPU to complete before moving on and issuing the next command. By waiting for the CPU to finish its operation, the test rig can check against the expected results to test cache coherency and sequential consistency. The second test rig, depicted in figure 5, was designed to test performance and "real-time" operation. It was simply a set of virtual CPUs using the Host-Client interface to communicate. For both rigs, the memory system portion of the test harness was the same (it also used the Host-Client interface).

The actual tests we ran were designed to test all of the transitions of the cache controllers' state diagrams. In case we missed some, we also used a random memory operation generator supplied by the professor. Trace functions within the modules displayed information to the terminal so we could monitor the process of our tests. A sample trace is shown below:

```
=> Cache 2: toknMsg op->Tk8
=> Cache 5: toknMsg op->Tk2
=> Cache 3: ringMsg op->WrBk addr->0000022c data->aaaaaaaa valid->1 cache->1
=> Cache 3: getState I
=> Cache 1: newCpuReq St { addr=00000230, data=ba4f0452 }
=> Cache 1: getState I
=> Cycle =  56
=> Cache 2: toknMsg op->Tk7
=> Cache 6: ringMsg op->Rd addr->00000250 data->aaaaaaaa valid->1 cache->6
=> DataMem: ringMsg op->WrBk addr->00000374 data->aaaaaaaa valid->1 cache->5
=> Cache 6: getState I
=> Cache 8: ringReturn op->Wr addr->000003a8 data->aaaaaaaa valid->1 cache->7
=> Cache 8: getState I
=> Cache 8: writeLine state->M addr->000003a8 data->4ac6efe7
=> Cache 3: ringMsg op->WrBk addr->00000360 data->aaaaaaaa valid->1 cache->4
=> Cache 3: getState I
=> Cycle =  57
=> Cache 6: toknMsg op->Tk2
=> Cache 3: toknMsg op->Tk8
=> Cache 4: ringMsg op->WrBk addr->0000022c data->aaaaaaaa valid->1 cache->1
=> Cache 4: getState I
=> Cycle =  58
=> dMemReq:  St { addr=00000374, data=aaaaaaaa }
=> Cache 3: toknMsg op->Tk7
=> Cache 7: ringReturn op->Rd addr->00000250 data->aaaaaaaa valid->1 cache->6
=> Cache 7: writeLine state->S addr->00000250 data->aaaaaaaa
=> Cache 7: getState I
=> Cache 1: ringMsg op->WrBk addr->00000374 data->aaaaaaaa valid->1 cache->5
=> Cache 1: getState I
=> Cache 4: ringMsg op->WrBk addr->00000360 data->aaaaaaaa valid->1 cache->4
=> Cache 4: getState I
=> Cache 9: ringMsg op->WrBk addr->000003a8 data->aaaaaaaa valid->1 cache->7
=> Cache 9: getState I
=> Cycle =  59
=> Cache 5: ringMsg op->WrBk addr->0000022c data->aaaaaaaa valid->1 cache->1
=> Cache 5: getState I
=> Cache 7: toknMsg op->Tk2
=> Cache 3: execCpuReq Ld { addr=000002b8, tag=00 }
=> Cache 3: getState I
=> Cache 4: toknMsg op->Tk8
=> Cycle =  60
=> DataMem: ringMsg op->WrBk addr->000003a8 data->aaaaaaaa valid->1 cache->7
=> Cache 2: ringMsg op->WrBk addr->00000374 data->aaaaaaaa valid->1 cache->5
=> Cache 2: getState I
=> Cache 8: ringMsg op->WrBk addr->00000250 data->aaaaaaaa valid->1 cache->6
=> Cache 8: getState I
=> Cache 5: ringReturn op->WrBk addr->00000360 data->aaaaaaaa valid->1 cache->4
=> Cache 5: getState S
=> Cycle =  61
=> Cache 5: toknMsg op->Tk8
```

Figure 6 In this trace you can see what happens at each clock cycle. For instance, during cycle 60, some of the caches can be seen processing writebacks moving around the ring.

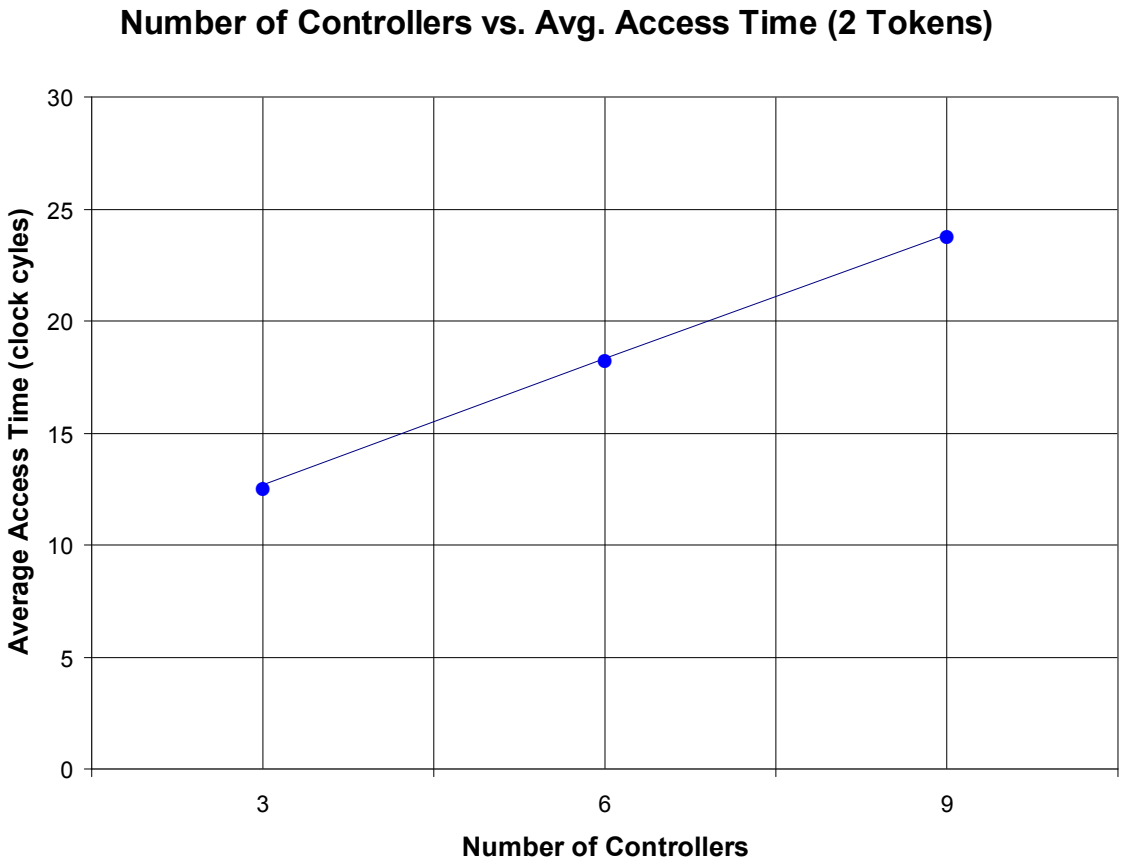

Figure 7 Shows the relationship between the number of cache controllers and the average access time

In figure 7 above, the results of one of our tests can be seen. In this test, we added cache controllers onto a ring with a fixed number of tokens, creating a relationship between access time and the number of controllers on a ring. As expected, the results fit a linear curve—the average access time increase in direct proportion to the number of cache controllers you place in the ring. In figure 8 on the following page, the relationship between the number of tokens and the average access time is plotted. As expected, the access time decreases when you add additional tokens. However, each additional token you add decreases the access time by less and less. This makes sense because each token you add doesn't change the address space by a constant amount. If you have 2 tokens, you need to add 2 tokens to half the address space. But then you need 4 tokens to further half the address space covered by each token. It continues in this exponential matter, which is supported by the graph in figure 8.

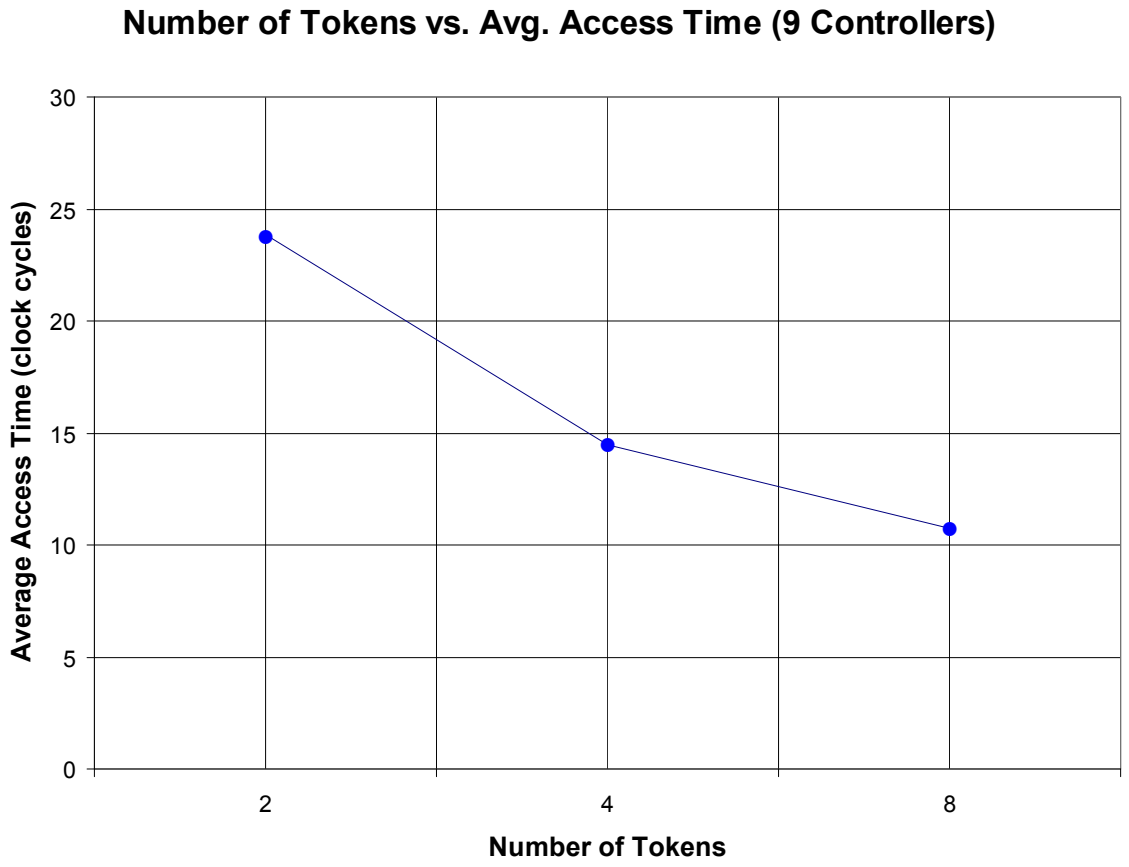

Figure 8 Adding tokens decreases average access time, but the benefit of each additional token decreases when there are more tokens.

Synthesis

Synthesizing the entire project was a challenge, as the parameterized modules caused us many problems. In the end, we were not able to explicitly synthesize the modules separately, though we were able to isolate them by selecting the appropriate cells in Encounter. A simple, yet lengthy, fix to this would have simply been to fix all the values for the purpose of synthesis. Other problems included the long time required to synthesize large and even medium sized register files, which we used for our cache. Also, the Host-Client interface for the memory and the CPUs did give us some problems until we modified the interface slightly. Initial passes of synthesizing out 9-controller, 8-token design gave rough area figures of about 1.3 mm^2.

Placing and Routing

After the synthesizing process was resolved, the placing and routing process went without any major bugs, and was mostly a waiting process. While the synthesis took between 10-15 minutes, the placing and routing took somewhere between a half hour and an hour. After the final run, the following results were returned:

- Total area: 1,232,183.4 μm^2, total gates: 241,974
- Average cache controller area (without cache): ~52,500 μm^2
  Average gates: 10,300
- Memory Controller area (without memory): 47,847.2 μm^2, gates: 9,393
- Average cache size: 60,000 – 75,000 μm^2, gates: 12,000 – 15,000
- Memory size: 96,636.4 μm^2, gates: 18,977
- Critical path: Cache controller 3 readIn FIFO, writing back to CPU and cache

An interesting aberration was the variation in creating the caches. A few were very small or large, and had much fewer or many more gates than the rest. This is certainly a result of optimizations, since we were not able to utilize the (* synthesize *) directive. Figures 9 – 11 (on pages 13 – 15) show the results of the synthesize through Encounter. In figure 9, you can see the separate blocks designating the memory controller and the various cache controllers, although some of them overlap is strange ways due to optimization. Figure 10 shows all of the cache controllers highlighted in red, with a single cache controller boxed in light blue and the memory controller boxed in orange (both are in the bottom left). Finally, figure 11 shows (in red) "the ring"—all of the input and output FIFOs of the cache controllers and memory controllers, in addition to the paths internal and external to each module.

Conclusion

Interconnection networks of different topologies offer many potential gains in performance, especially in multithreaded environments, but it is necessary to maintain sequential consistency in the memory system and cache coherency for each processor. A ring topology offers some key advantages over a classical bus topology, including scalability and increased bandwidth, but requires a more complicated protocol that addresses the issues of communication through the ring. There are still many more optimizations and additions to be explored with a point-to-point ring interconnection network, including full utilization of the concurrency allowed by the topology, and these should be the target of any further exploratory studies in the field.

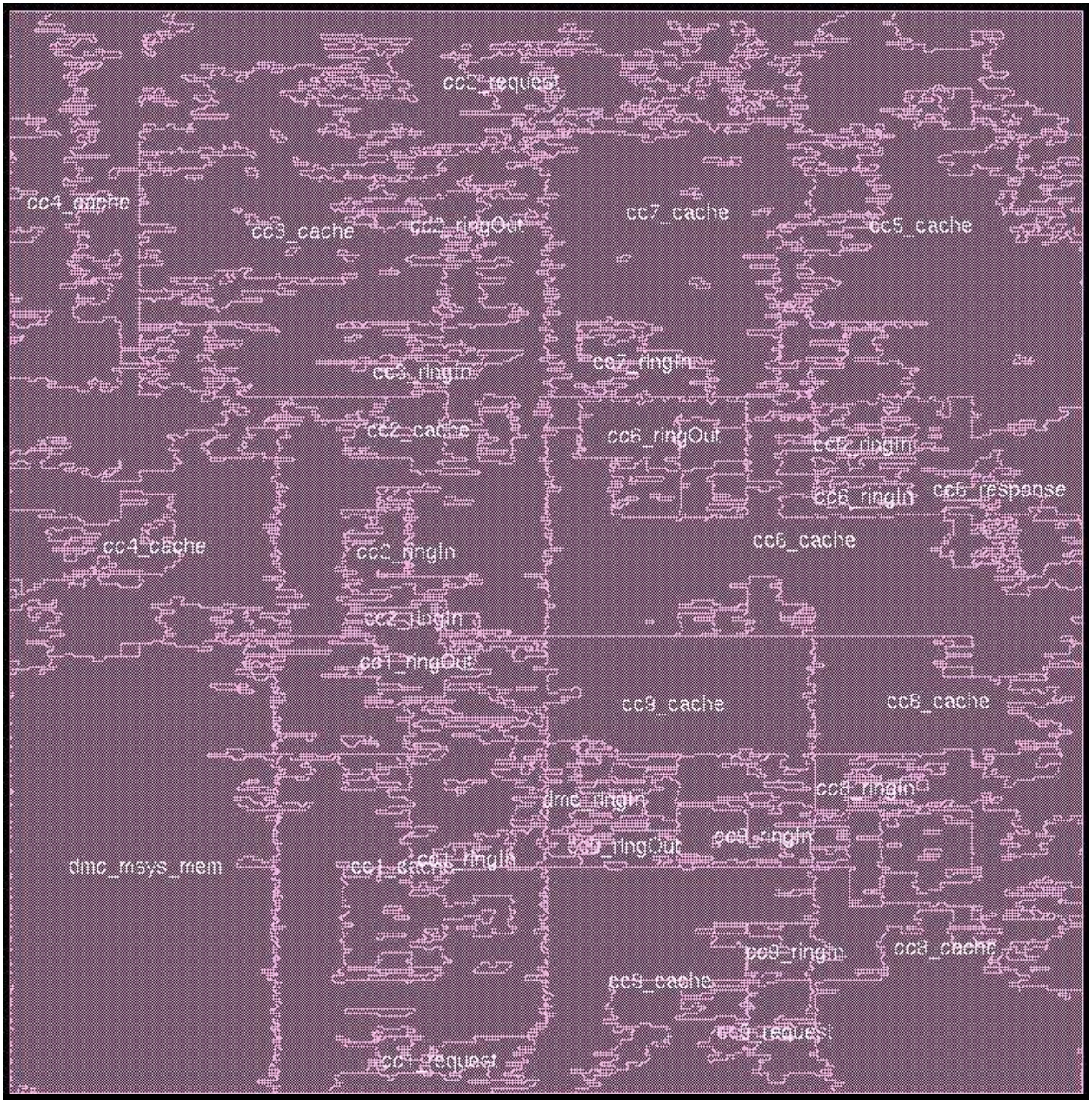
cc4_cache
cc3_cache
cc2_ringOut
cc7_cache
cc5_cache
cc3_ringIn
cc7_ringIn
cc2_cache
cc6_ringOut
cc6_ringIn
cc6_response
cc4_cache
cc2_ringIn
cc6_cache
cc1_ringOut
cc9_cache
cc8_cache
dmc_ringIn
cc9_ringIn
dmc_msys_mem
cc9_ringOut
cc1_ringIn
cc8_ringIn
cc8_cache
cc5_cache
cc1_request

Fig. 9

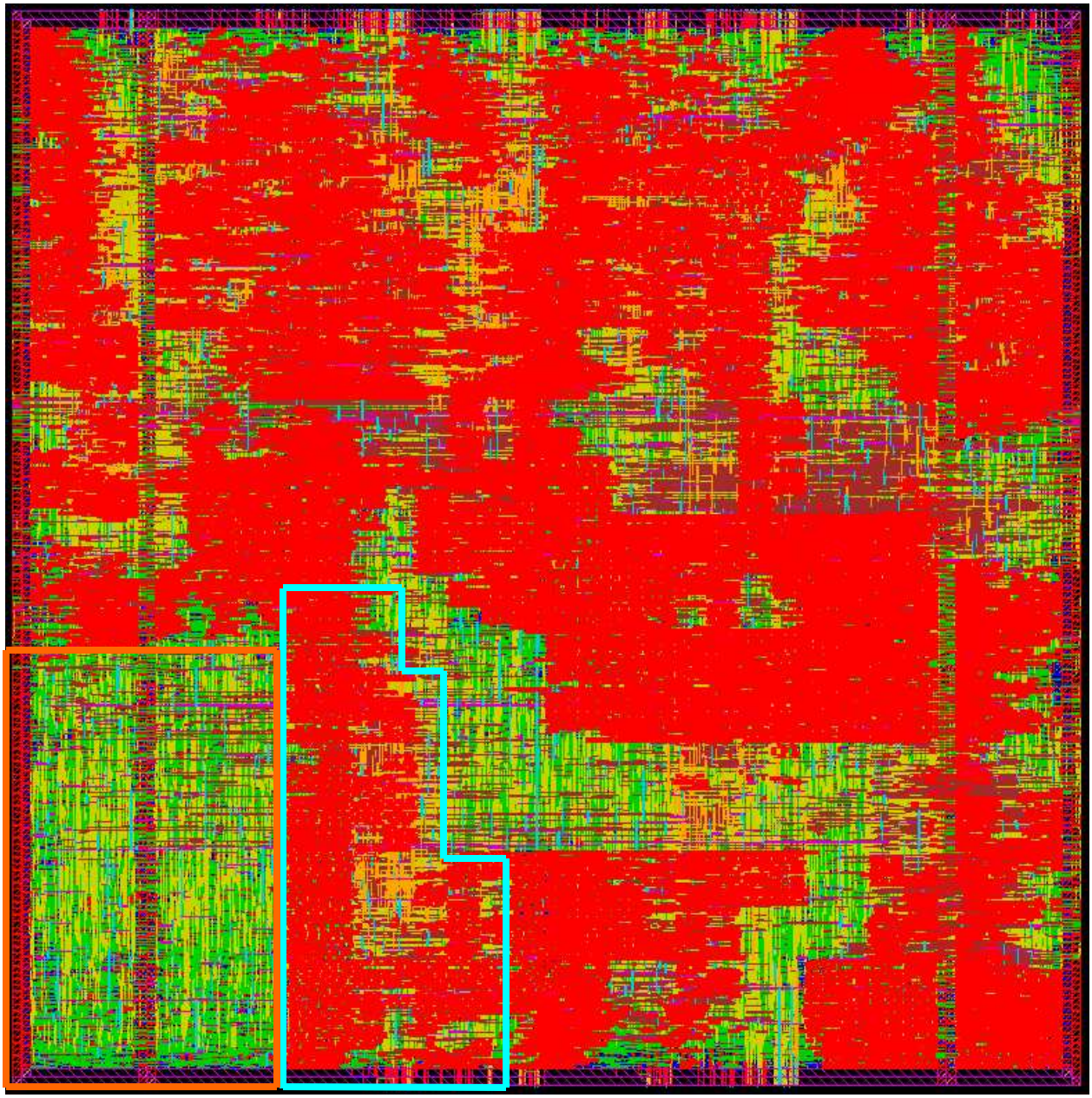

Fig. 10

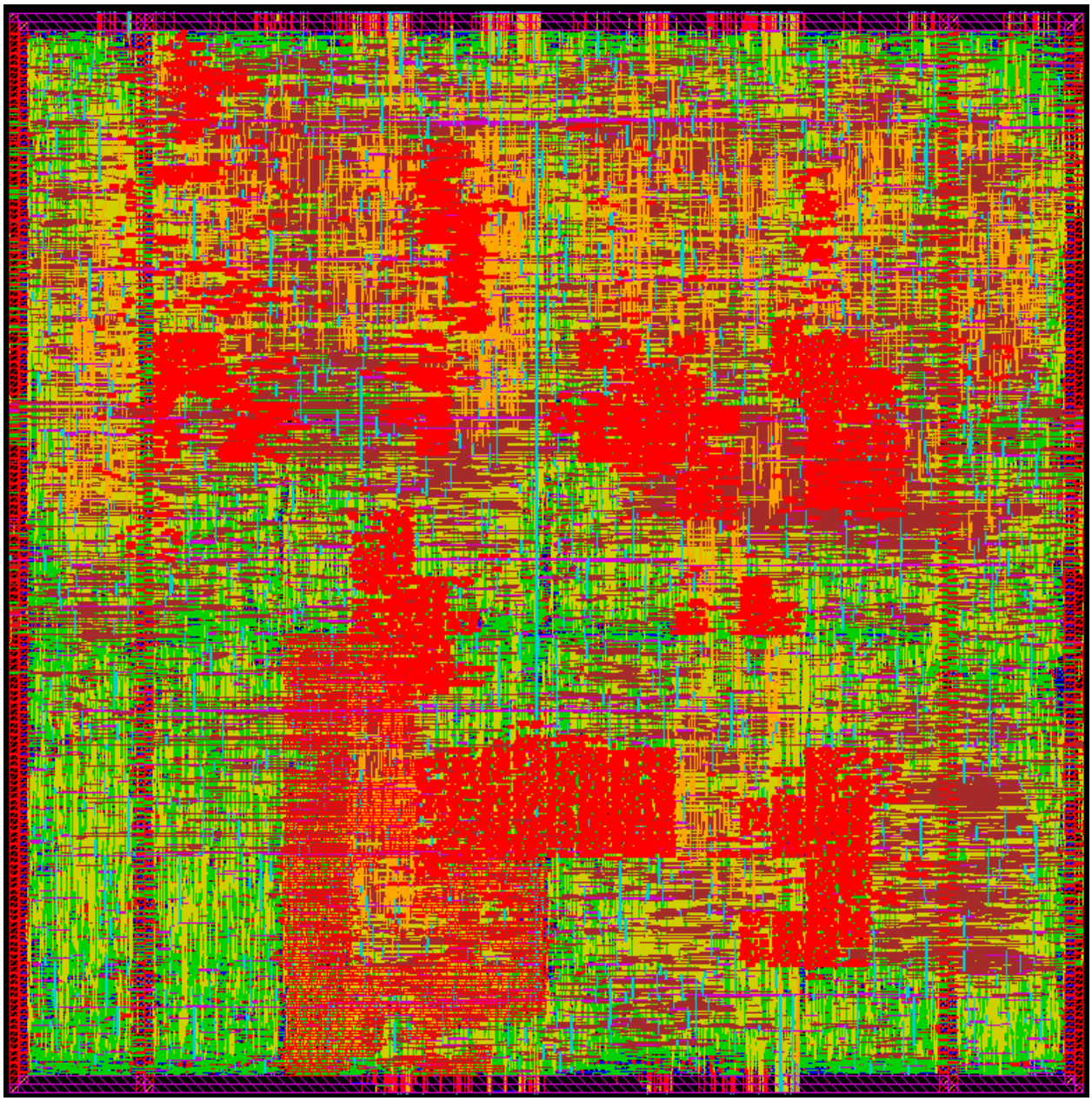

Fig. 11