A HARDWARE ACCELERATOR STORE FOR LOW POWER PROCESSORS

May 15, 2008

GROUP 1

Mike Lyons
Kevin Brownell
Durlov Khan

Motivation

Custom hardware design is widely known for its ability to improve performance and increase power efficiency. ASICs often rely on custom hardware in the form of IP blocks or hardware accelerators. These accelerators typically follow a completely black-box model and contain all the memory modules required for operation. Including memory in the accelerator gives the designer full control of the memory, and therefore reliable performance.

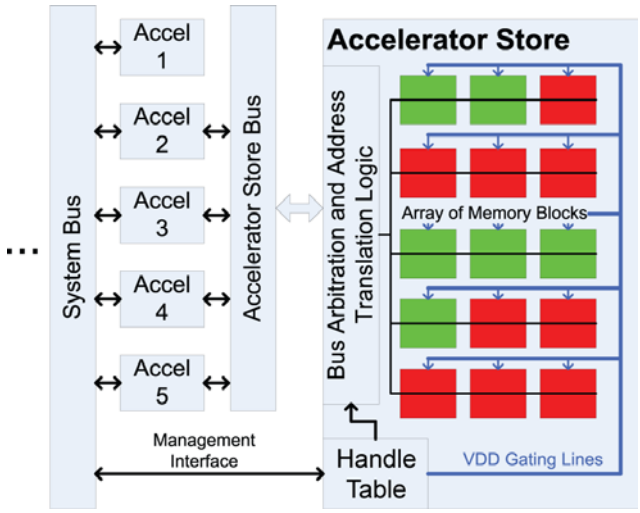
The internalized memory approach can result in inefficient use of area, degraded performance during accelerator interaction, redundant logic and subpar memory power consumption. Area may be wasted because each accelerator may not utilize all of its memory simultaneously. Further, communication performance between IP blocks may be degraded if the ASIC must copy large amounts of memory between private memory blocks in each accelerator. By removing inter-accelerator communication bottlenecks, we can reduce accelerator granularity in some cases by decomposing large accelerators into several smaller and more widely applicable accelerators. If multiple accelerators require support for additional memory features such as power reduction (VDD-gating) or data structures (queues or stacks), this logic would be duplicated across several accelerators. Alternatively, ASIC designers may not be able to add support for memory power reduction to accelerators that lack native support.

The accelerator store is a shared memory framework. It provides a common interface for each accelerator to read and write memory in a variety of ways. This allows accelerator designers to separate logic from state and any non-transient data would be stored in the accelerator store. This simplifies VDD-gating design of the accelerator, since gating a logic-only accelerator would not lose any state. The accelerator store consists of several memory modules. If the memory module contains valid data, the module remains on. If the memory module does not contain valid data, it can be VDD-gated to reduce leakage power.

This design also provides accelerators a richer means of communication. Rather than copying data from one accelerator's private memory to another's, accelerator-to-accelerator communication can follow a producer/consumer model. Data producing accelerators can use FIFOs/Queues to insert data into the accelerator store. Data consuming accelerators can then read this data in order. With this operation style, accelerators do not need to be specifically designed with each other in mind. Simultaneously, no arbitrating logic is required beyond initializing the accelerator store and pointing the producers and consumers to the correct FIFOs/Queues.

One large concern with removing memory from within accelerators and creating a shared memory system is a reduction in performance. Whereas accelerators with embedded memory could specifically design memory accesses to satisfy required bandwidth for internal operations, the memory accesses with a shared memory system could be limited by the shared memory's bandwidth. To address this point, we have previously developed a cycle accurate simulator to demonstrate that the accelerator store does not induce a bottleneck for these internal operations. We can also use this simulator to show performance *improvements* due to improved inter-accelerator communication.

High Level System Description



Our system reflects the figure shown on the right. A processor (not pictured) communicates over a system bus (address/data) to several accelerators. Each accelerator is memory mapped, so the processor can read from and write to accelerators using standard memory instructions. The system is designed for maximum energy savings and targets low frequency applications, such as those found in sensor networks.

A management interface to the accelerator store is available over the system bus via memory mapped operation. This management interface allows the processor to create or remove "handles" that represent a portion of memory in the accelerator store. A handle table in the accelerator store maintains a list of all active handles and any meta-data required for their operation. This meta-data includes head and tail pointers required for queue/FIFO operation, and a few mode flags. This interface is described in a later section.

The processor is responsible for assigning these handles to accelerators. The accelerators then use these handles when accessing the accelerator store.

The accelerator store is able to accept several operation requests per cycle, each sent over an accelerator store port. The accelerator store bus consists of the full set of these accelerator store ports. However, it may not be able to fulfill requests from all ports during the same cycle. Our design multiplies the internal accelerator store clock frequency by 4x so that we can process four requests per accelerator cycle.

We implemented a cycle accurate simulator in prior work to examine the operation, performance, and memory energy usage characteristics of the accelerator store design for multiple sensor network applications. We previously found that the accelerator store was able to reduce powered-on memory, simplify accelerator and application design, reduce memory-related area with negligible or even beneficial effects on performance. We modified the simulator to emit traces in order to validate our hardware implementation and quantify the energy, power, and timing overheads due to the accelerator store.

System Software Architecture

The accelerator store provides two independent software interfaces: the configuration/management interface for use by the general purpose processor over the system bus and the accelerator interface for use by the accelerators in the system.

System software is responsible for configuring the accelerator store and managing the coordination among accelerators, and between accelerators and the accelerator store. The configuration/management interface allows system software to perform configuration tasks such as setting up the handles in the accelerator store's handle table. It also provides support for management functions for coordinating among accelerators, such as when initializing a producer-consumer model. Additionally, system software also coordinates between accelerators and the accelerator store by updating the priority table inside the accelerator store, based on the current system load and objectives. Other management functions include handling interrupts from the accelerator store. Details of the interrupt-based software model are in the following section.

Software running on the microcontroller is responsible for the operation of the accelerators. This software invokes memory access routines much like the way it would do if it had local unshared memory to pass handle ids and start accelerator operations via the system bus.

Accelerators may wish to access memory during operation and can issue memory read and write operations to memory in the accelerator store just as it would for a single-cycle private memory. The only caveat is that an accelerator's memory request may not be serviced on every cycle. Three situations can cause the accelerator store to reject an accelerator store request. First, the accelerator store has many more ports (potential requests) than slots for accepted memory requests. As a result, some requests may be denied if more requests are made by accelerators than can be satisfied in the accelerator store in a given cycle. Second, the accelerator store can store FIFO and queue data structures. A request may be denied for these handles if the data structure is empty and an accelerator attempts to remove an element or the structure is full and the accelerator attempts to add another element. Finally, the accelerator store will VDD-gate (turn off) any memory module that is unused. If an accelerator attempts to access a VDD-gated memory module, it must wait for a certain amount of time before the memory module is fully powered up. In this case, accelerator store requests for handles mapping to powering-up memory modules may be rejected for a brief time.

Provided that a memory request is accepted by the accelerator store, the memory operation will complete after one clock cycle, so minimal timing adjustments are needed to port existing accelerators to an accelerator store compatible design. The only modifications to these accelerators is to handle the case where a request is not accepted.

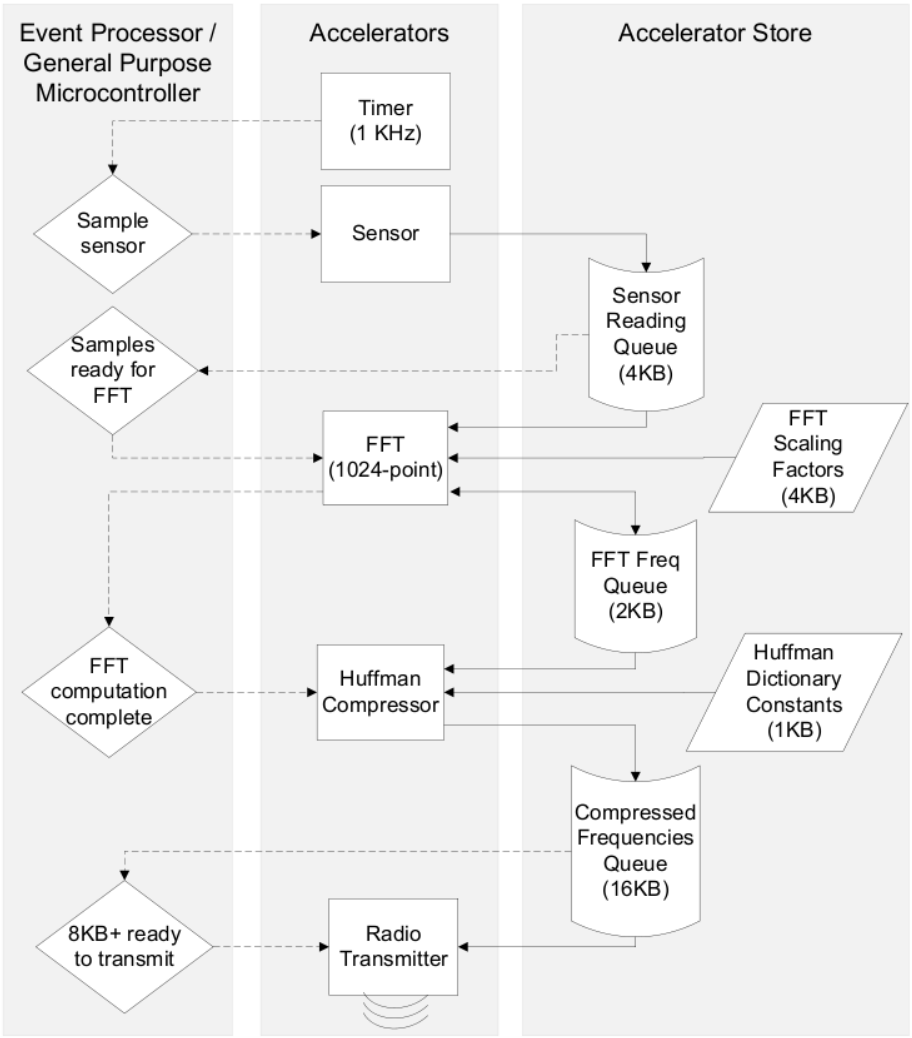
Interrupt Model

System software uses the configuration/management interface to set up threshold triggers in handles that are configured to model complex data structures such as queues or stacks. Threshold triggers can be configured in the accelerator store to dispatch interrupts to the general purpose processor whenever the data structure's element count reaches the threshold value. System software is responsible for implementing interrupt service routines (ISRs) to handle these interrupts in the event-driven programming paradigm. These trigger values can be used by software to indicate that a FIFO is almost full or that enough samples have been taken by a sensor to perform an FFT operation. Upon receiving this interrupt, the ISR would then configure accelerators to process this data.

Programming Considerations

Hardware accelerators do not know about the memory layout of the accelerator store and do not explicitly request memory. Memory assignment decisions are specified by the programmer and reside in software executed by the microcontroller. Handle assignments can be made at compile time to avoid the possibility of memory segmentation or in a software dynamic memory allocator. The accelerator store is designed to manage faults and provide protection against accelerators arbitrarily accessing unallocated memory. If a request is made outside the memory allocated to a handle, the read or write will fail and an error signal will be fired. A memory segment may be accessed by multiple accelerators during application execution.

The arbitrating general purpose microcontroller can preserve concurrency by carefully assigning handle ids to accelerators. Concurrency is generally preserved by allowing any number of agents (whether accelerators, threads, or computers) to read from a data source *or* allowing one agent to modify it. With this approach in mind, the general purpose microcontroller could assign and rescind handle ids to ensure that only one accelerator possesses a handle id when it is making changes (modifying, adding, or removing elements).



The figure above shows the program flow of a base application implemented in the system. Diamonds represent interrupts serviced by the general purpose processor. Arrows directed into interrupts indicate raising an interrupt. Arrows directed away from interrupts indicate actions by interrupt service routines. Rectangles represent accelerators. Parallelograms indicate memory in the accelerator store, logically separated. Rectangular crescents indicate queues in the accelerator store.

The flow diagram above illustrates how a base application is programmed using a small amount of code on the general purpose processor and a few general accelerators. The base application first initializes the system. Memory is allocated in the handle table and pointers to queues, stacks, and unconstrained memory are written to accelerators. Interrupts and accelerator ports to the accelerator store are prioritized. Accelerator-specific details are also written to the appropriate accelerators.

The application must sample the sensor every 1ms. The application configures the timer accelerator to raise an interrupt every 100 cycles. This interrupt executes a few instructions to turn on the sensor accelerator and take a sample. The sensor accelerator adds the 16-bit reading into the sensor reading queue. The sensor accelerator correctly specifies the output queue by providing the corresponding handle ID to the accelerator store. The accelerator store only turns on the memory blocks needed to store the readings and VDD-gates unused memory blocks.

The base application also needs to compute the frequency response when 1024 samples arrive. The base application configures the accelerator store to raise an interrupt when the sensor reading queue contains 1024 samples. When this interrupt is called, another small set of instructions turns the FFT on and starts FFT computation. The FFT accelerator reads in the sensor sample queue and scaling factor constants from the accelerator store. When computation is complete, the frequency response is inserted into the FFT output queue, also stored in the accelerator store. Note that the FFT computation may require many cycles but does not prevent other accelerators from processing data and raising interrupts. The sensor takes several readings during FFT computation and stores them in the accelerator store unaware the FFT was running.

Next, the base application compresses the frequency response after the FFT computation completes. The application configures the FFT accelerator to raise an interrupt when the FFT finishes. This interrupt turns the Huffman compressor on and begins reading in data from the FFT output queue. The Huffman compressor also reads in dictionary entries by using an unconstrained handle in the accelerator store. As input and dictionary entries are read from the accelerator store, the compressor inserts compressed data into the radio transmission queue. The compressor's input queue should be empty when compression is complete; the accelerator store will then VDD-gate the block of memory if no other handles are storing data in the same block of memory.

Radio transfers are typically more efficient when sending large amounts of data with a bulk protocol to minimize overhead. Rather than sending every compressed set of frequency responses immediately, the application waits until 8KB of data is available and sends the data together. The base application configures the accelerator store to raise an interrupt when the radio transmission queue reaches 8KB of data. At this point, the application begins trying to send data. The queue can hold up to 16KB of data before dropping data. If radio contention does not end before the queue fills the 16KB maximum, the accelerator store can automatically drop the oldest or newest data depending on the application designer's preference. When contention stops, the radio transmitter quickly drains the transmission queue.

Handle Table

Handle Id	*Valid*	*Type*	*Starting Address*	*Size (bytes)*	*Head Offset*	*Element Count*	*Trigger Level*
0	Y	Unconstrained	0	1024	0	512	0
1	Y	FIFO	1024	1024	2	1	5
2	Y	Stack	2048	2048	0	1	5
3	Y	CFIFO	4096	4096	0	0	5
...	N	0	0	0	0	0	0

Seen above, the entries of a handle table consist of meta data about the various handles active in the system at any given time. The handle ID is a unique identifier for each active handle. The type field describes the type of handle at that entry. Handle types will be described in the next section.

The starting address corresponds with the physical address where the allocated memory begins. The size indicates the amount of space accessible to a handle. The head offset field is only relevant for constrained handle types. It indicates the current offset of the head pointer with respect to the starting address. The element count filed, also only applicable to constrained handle types, keeps a count of the number of elements in the data structure. The trigger level filed is used by the handle table to fire interrupts based on the current occupancy of the data structure. For example, for handle 1 above, whenever the FIFO reaches a size of five elements, an interrupt is sent to the processor.

Handle Types

The accelerator store supports a number of different handle types. These include:

Unconstrained

This handle type allows for accessing memory without any constraints. An accelerator may directly access any location within the memory space allocated for this handle, in any order, by specifying a handle ID and offset pair. Adding or removing elements to the handle is not allowed since reads and writes to all elements in the handle must always be allowed.

FIFO

This handle type supports the management of a FIFO data structure. Elements can be added or removed in a enqueue/dequeue fashion. The handle table maintains the head pointer, so that it always points to the oldest element in the structure. The handle table also keeps track of the number of elements currently in the structure. If a FIFO has consumed all of its allocated space, further enqueues will fail. In addition to enqueues and dequeues, an accelerator can also directly read and write anywhere within the first and last element of the FIFO. The tail (the item returned by a *get*) is considered handle offset *0*. The head is located at handle offset *(elemCount -1)*.

Circular FIFO

This handle type is identical to a FIFO, except that it allows overwriting of its elements if an add is preformed when the FIFO is full.

Stack

This handle supports the management of a stack data structure. Elements can be added or removed in a push/pop fashion. Similar to how it manages a FIFO, the handle table maintains the head pointer of the stack. As with a FIFO, an accelerator can also directly read and write anywhere within the first and last element of the stack.

Accelerator Store Interfaces

General Purpose Processor Interface

The general purpose processor is responsible for managing some aspects of the Accelerator store. These tasks include adding and removing handles in addition to setting the priority mapping. Several operations are allowed:

modifyHandle(ModifyHandle hReq)

it'sThe modifyHandle operation allows the general purpose processor to either create or remove a handle from the handle table, depending on the ModifyHandle object, which is defined as:

```
typedef union tagged
{
    HandleRequest   CreateHandle;
    Hid             RemoveHandle;
} ModifyHandle
```

When creating a handle, the general purpose processor should build a HandleRequest object, which contains a handle id number, the starting physical address of the handle, the size of the handle, and the type of the handle.

To remove a handle, the processor must simply make a request with the handle id of the handle to be removed.

Illogical or inconsistent handle create operations will result in undefined behavior. The processor is responsible for ensuring that at most one handle maps to a physical memory address. If this rule is not followed, modifying data in one handle may change data in other handles and memory modules may be incorrectly VDD-gated. All other handle modification errors will not corrupt the accelerator store state. For example, removal of a nonexistent handle will not result in any handle table modifications.

priorityMapChange(PortPriorityMapping pMap)

The priorityMapChange operation allows the processor to order the priority of the memory requests from the accelerator ports. A PortPriorityMapping object is defined as:

```
typedef struct {PortID portId; PriorityLevel portPriority;} PortPriorityMapping
```

When invoking a priorityMapChange operation, the processor provides a numerical port id and a numerical port priority, which is saved into the priority table, within the accelerator store. A low number will result in a higher priority. For example, a port with a priority of 0 will always be accepted by the accelerator store whereas larger priority numbers will be rejected due to contention more often.

changeTrigger(Hid handleId, HElemCount elemCount)

The changeTrigger operation allows the set a threshold trigger on a handleId or turn a threshold trigger off. For example, the microcontroller could set the accelerator store to raise an interrupt when at least 5 elements are stored in handle 1 by issusing the command *changeTrigger(1, 5)*. Triggers can also be disabled by setting the elemCount parameter to zero.

Accelerator Interface

Each accelerator communicates with the accelerator store through one or more ASPorts. The ASPorts allow the accelerator to access the accelerator store memory. Two operations allowed:

portRequest(ASPortReq portReq);

Each ASPort allows an accelerator to issue a portRequest operation. A request is made by first building an ASPortReq, which is defined as:

```
typedef union tagged {
  struct {Hid handleId; HOffset handleOffset;}                  Read;
  struct {Hid handleId; HOffset handleOffset; Data dataWord;}   Write;
  struct {Hid handleId; Data dataWord;}                         Add;
  struct {Hid handleId;}                                        Remove;
}  ASPortReq
```

The semantics of a ASPortReq vary depending on the type of handle indicated by the handle ID in the request.

If the handle is of type “Unconstrained,” only Read and Write requests are allowed, with Add and Remove causing “FailStruct” port responses. Read and Write both take in a handle ID and a handle offset, with Write also containing a data word. The accelerator store accesses memory by adding the handle offset to the base address of the handle (stored in the handle table) to preform a Read or Write operation.

If the handle is of a constrained type (FIFO, Circular FIFO, stack), all four ASPortReq types are allowed. Read and Write requests access memory by adding their handle offset with the current base pointer of the structure (which is stored in the handle table). Unlike with an unconstrained handle, a constrained handle only allows memory accesses within the bounds of the data structure it defines. Otherwise, a FailStruct response is returned to the ASPort. Constrained handle types also support Add and Remove requests. Depending on the particular handle type, these requests act differently. In the case of a stack, an add acts as logical 'PUSH', which appends the data word to the head of the structure. A remove acts as a logical 'POP', which remove the data word at the head of the structure. In the case of a FIFO, an add acts as a logical “enqueue,” and a remove acts as a logical “dequeue.” Unlike with Read and Write requests, Add and Remove requests alter the head and size of the data structure.

ASPortReply portReply();

The portReply operation returns the results of the port request made in the previous cycle. An ASPortReply object is defined as:

```
typedef union tagged {
  void        WriteSuccess;
  Data         ReadSuccess;
  void        FailBusy;
  void        FailStruct;
  void        FailPower;
} ASPortReply
```

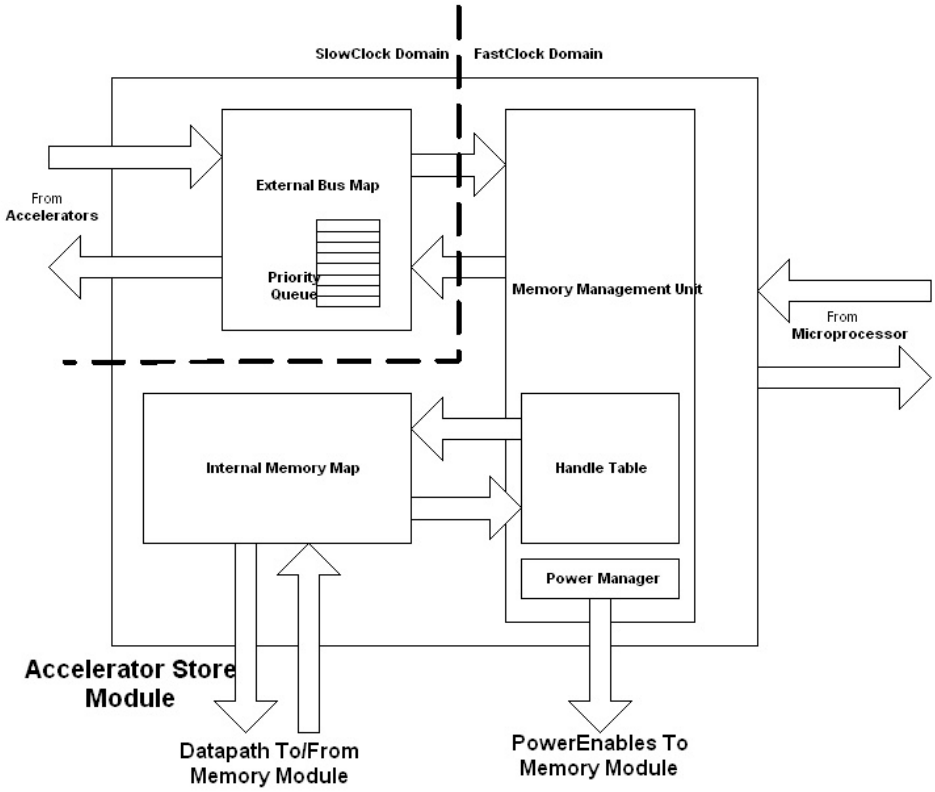
A Write or Add request which completes without a failure returns an ASPortReply of type WriteSuccess. Likewise, a Read or Remove request which completes without a failure returns a ASPortReply of ReadSuccess, along with the requested data.

There are a few different possible failure responses. The first, FailBusy, is returned in the case that the ASPort which made the request was too low of a priority given the number of concurrent requests.

The second, FailStruct, is returned in several cases. If the request was a Read or a Write, FailStruct is returned when the attempted memory access was out of the structure's bounds. If the request was an Add or Remove, FailStruct is returned if the structure is full or empty, respectively.

The final failure response, FailPower, is returned if the attempted memory access touches a memory block which is currently unpowered or not yet fully powered up.

Block Level Architecture



The accelerator store is comprised of the accelerator store module and the memory module.

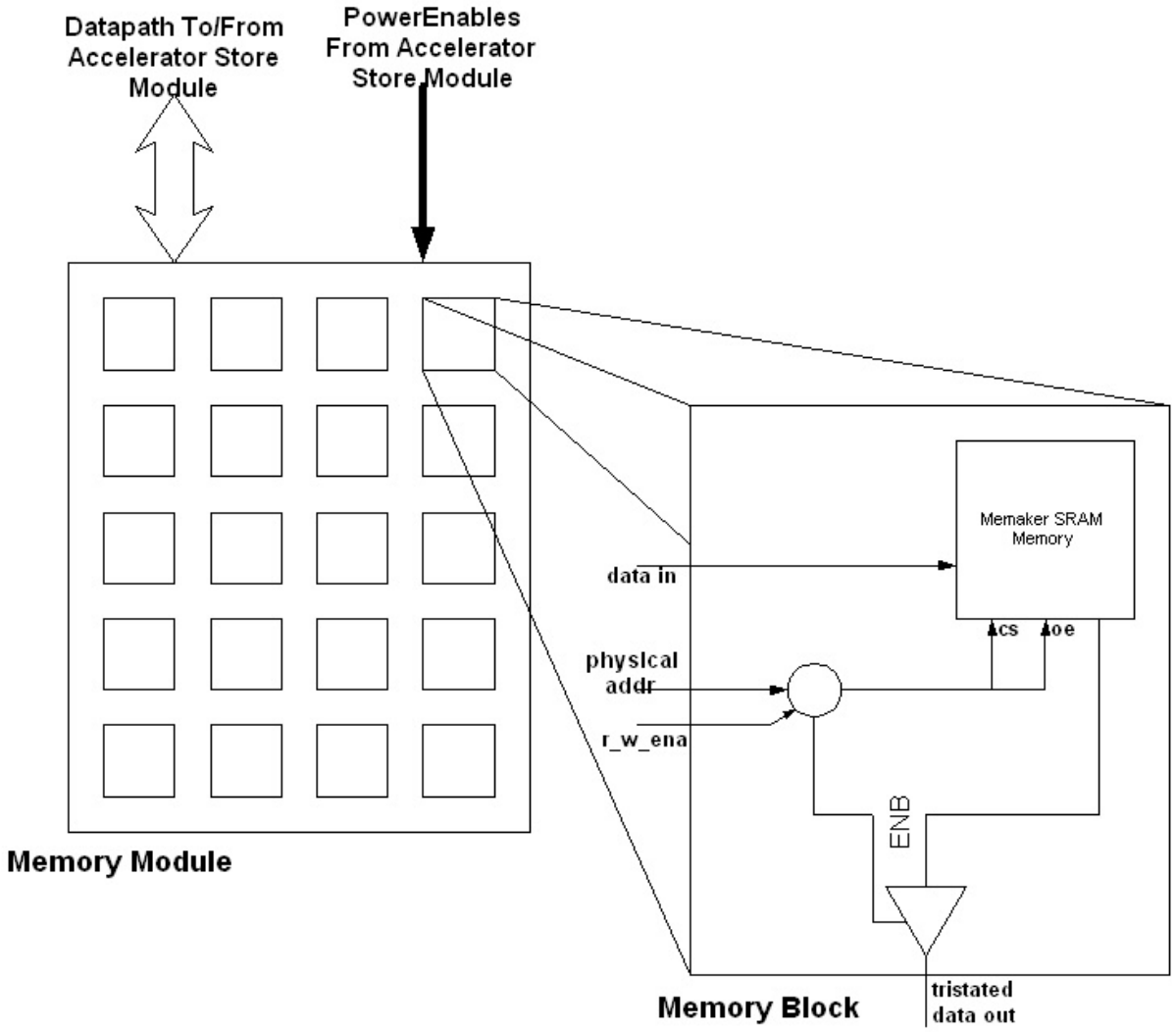
The accelerator store datapath starts at the interface to the Accelerators.

At every cycle, the External Bus Map interacts with every accelerator port. The External Bus Map maintains a priority queue which is programmable by the system processor to decide which requests are accepted and which are rejected with a FailBusy response. The priority determines which requests are passed on to memory during this cycle.

The External Bus Map passes up to *p* prioritized requests to the MMU, where *p* is the speed-up factor of the Fast Clock with respect to the Slow Clock. This allows the memory which is in the Fast Clock domain to service *p* requests within one cycle of the Slow Clock. If more than *p* requests are made per cycle, some accelerators will be denied service, and must retry on a later cycle. Our target application domain uses slow clock frequencies and as a result memory is clock well below its maximum speed.

The external bus map was carefully designed for scalability. In our initial naïve implementation, the Bluespec compiler was unable to implement an external bus map with more than eight accelerator ports. We created a series of submodules for selecting one port request at a time with the selected ports passed on to the next submodule. The first submodule would check if the port with priority 0 was making a request. If so, it would add this port's request to the accepted request stream sent to the next submodule. This process would continue until all ports were considered or the maximum number of accepted requests were chosen. Our application domains require slow clock frequencies, so the added delay due to the serially designed circuit did not cause performance problems. Further, this new design scales well beyond the sixteen accelerator store ports we require.

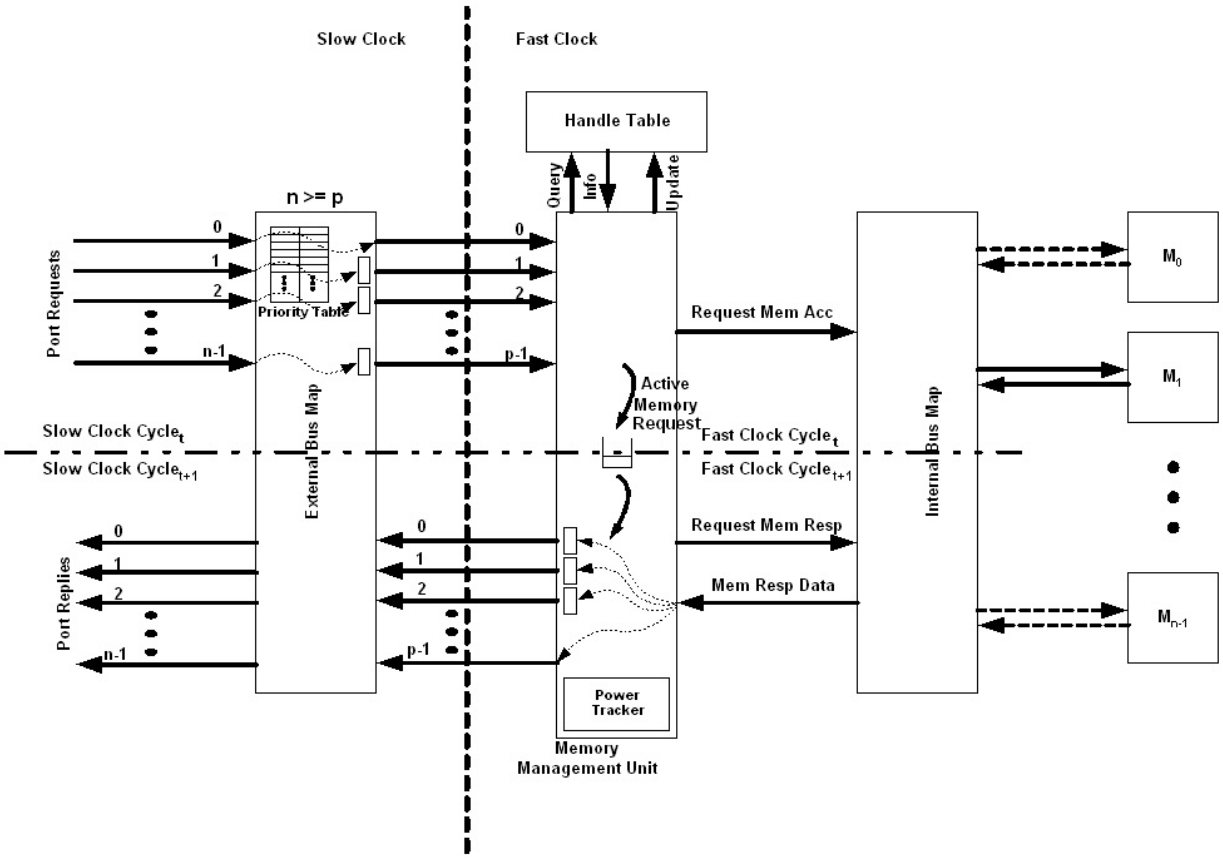
The Memory Management Unit runs at the faster clock speed and services requests one at a time. The Memory Management Unit accesses the Handle Table to perform the necessary address translation and operation translation (read/write, push, pop, etc.) as well as the Power Tracker for turning memory on and off.



The memory requests are passed one by one over to the Internal Bus Map. The Internal Bus Map makes a memory request to the appropriate memory block.

The memory module is made up of a parameterized number of memory blocks, each of a parameterized number of words. Each block can be VDD-gated individually based on the power-enable bit vector from the MMU.

Detailed Datapath



The above figure shows the block level architecture of the system. The Accelerator store consists of the ExternalBusMap, the MemoryManagementUnit, the HandleTable, and the InternalBusMap modules. In every cycle:

First, port requests arrive from each of the ASPorts which are making accesses this cycle. The ExternalBusMap, after consulting the priority table, decides which port requests will be serviced, and latches them (except for the 1st request, for timing reasons elaborated below). All ASPorts not being serviced this cycle are informed of their failure. The ExternalBusMap passes the chosen requests to the MemoryManagementUnit, which lies across a clock domain crossing. This new clock domain is run at four times the rate of the accelerators and the ExternalBusMap.

During each fast clock period, the MMU queries the handle table for the requested handle's meta data info, including the physical address of the handle's allocated memory. Depending on the type of request, the MMU updates the handle's entry in the handle table. For example, if the request is a add, and the handle is a FIFO, the handle entry's size and head pointer fields need to be updated.

After communicating with the handle table, the MMU requests a memory access by passing the physical address of the request to the InternalBusMap. Simultaneously, the MMU saves this memory request in a Bluespec FIFO for use after the request is completed.

The InternalBusMap decides which memory block the address is located in, and preforms the access.

After the next rising edge of the fast clock, the memory has completed its access. At this point, the MMU begins processing the 2nd prioritized port request, while simultaneously requesting a memory response from the InternalBusMap for the 1st request, if it was a read. The MMU then builds a ASPortReply based off of the stored active memory request register, including any response data from the InternalBusMap, in the case of a read. This reply is stored in a register.

This process continues, with one request being fulfilled every cycle of the fast clock. The only deviation from this flow is that the last response from memory is not stored in a register, but is instead passed directly back to the ExternalBusMap (for timing reasons elaborated below) and out to the accelerator ports, where it can be latched if necessary, along with the other responses.

This entire cycle repeats for every period of the slow clock, with four requests serviced for each slow clock.

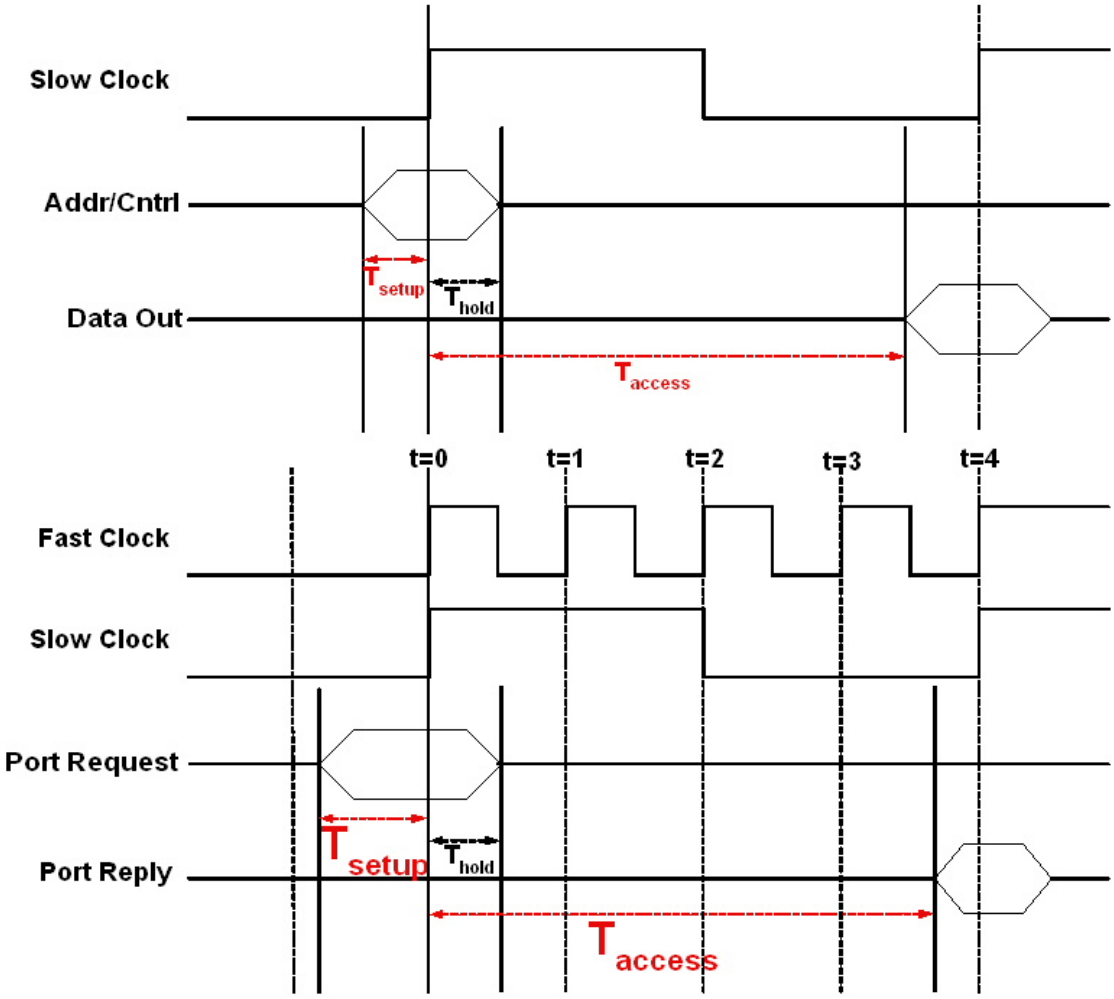
In addition to the structural failure and the busy failure previously discussed, the MMU also houses the PowerTracker submodule, which is used to determine whether there is a power failure. If the accessed memory block was not fully powered at the time of the access, the MMU would build a power failure reply for the associated active memory request.

Timing

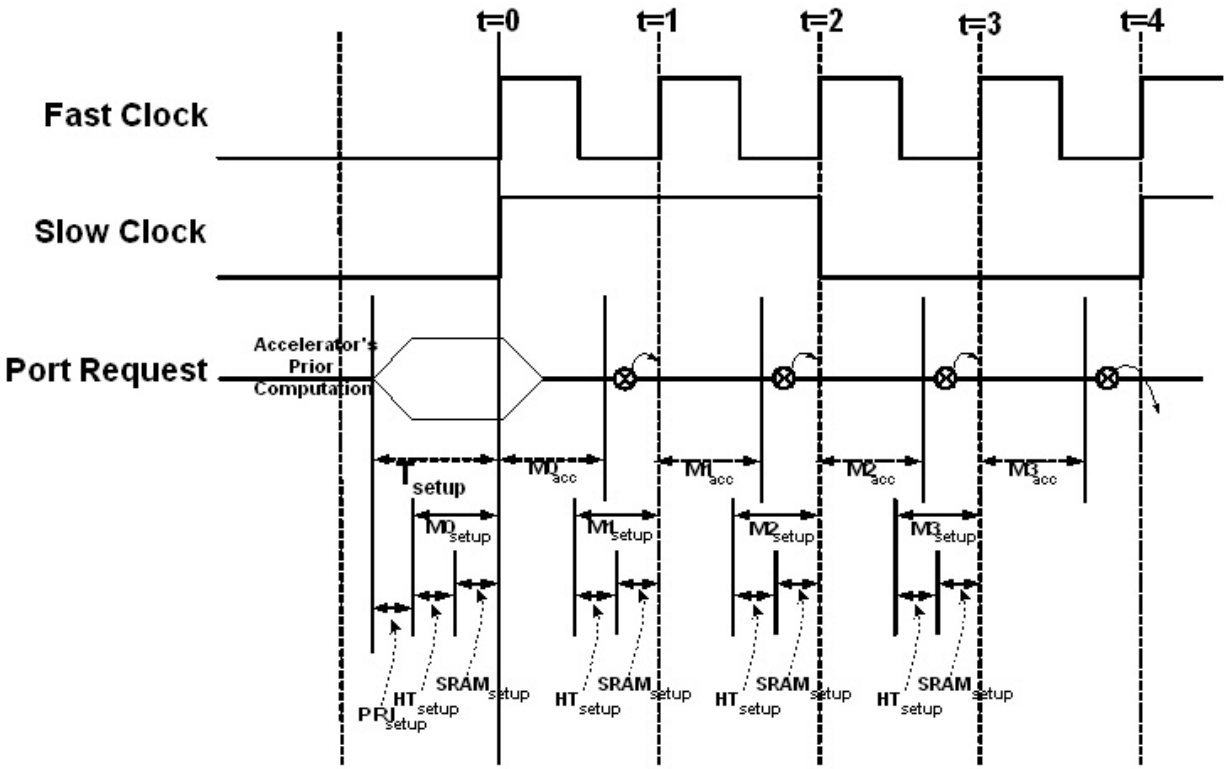
It is the goal of the accelerator store to look as much like a normal SRAM to the accelerators as possible. It is very important, then, to ensure the system adheres to stringent timing specifications. A standard SRAM completes a read access and returns the read data within one cycle.

The diagram below shows a high level timing diagram of a typical SRAM vs the accelerator store. The top timing diagram illustrates the address and control inputs being driven by an accelerator on clock edge 0. Valid data is returned by the SRAM on clock edge 1. There is some setup time and hold time associated with the driving of the address and control inputs. The read data is valid before the next clock edge but some access time later after the address and control signals were latched by the SRAM.

The bottom timing diagram shows the effective timing introduced by the accelerator store. Note the moderate increase in setup time, and the slight increase in the access time for the accelerator store in comparison to the SRAM.



The accelerator store timing acts, from the point of view of the accelerators, just like an SRAM, except for an increased setup time. This setup time is comprised of the time necessary to access the priority table, accept up to four requests, access the handle table for the first processed request (details below), and the basic SRAM setup time. This extra setup time seen by the accelerator allows the fast clock of the accelerator store to process four requests using four fast clock cycles, within a single slow clock cycle, and returning the result of all requests before the next slow clock edge.



A detailed timing diagram of one slow clock of the accelerator store is shown above.

On a cycle which an accelerator wishes to preform memory accesses, it must have all of its requests stable on its ASPort request lines for a period of time before the clock edge. This period of time must be long enough for the AcceleratorStore to prioritize all requests, and for the first chosen request, to access the handle table, and for the first request to have its physical memory address be stable on the SRAM address lines. In the diagram, this is denoted by T_{setup}, which is broken down into priority processing time (PRI_{setup}), handle table lookup time (HT_{setup}), and the SRAM setup time ($SRAM_{setup}$). We denote the sum of the handle table lookup time and SRAM setup time as $M0_{setup}$. Since the first request is processed and sent to the SRAM before the slow clock cycle edge, it is not latched by the accelerator store, whereas the other requests are.

After the next fast clock edge, the result of the first request is available from the SRAM, and is latched by the MMU. As can be seen in the diagram, each subsequent fast clock edge incurs the same M_{setup} time. However, while $M0_{setup}$is exposed to the accelerator, the subsequent M_{setup} times are absorbed by the accelerator store.

In the diagram, the time necessary for a read to be returned from memory is denoted as M_{acc}. The bullets represent the latching of the port response (including any read data) within the MMU. Note that the final memory response is not latched. Rather, it simply is passed directly to the accelerator reply port, along with all other replies. The accelerator is responsible for latching or otherwise using the replies to their requests by the rising edge of the slow clock.

Current Implementation and Simulation Results

The accelerator store is implemented in Bluespec, as described in this document. Some Verilog was used in order to link certain modules with the rest of the design, such as memory compiler generated SRAM. Currently, the accelerator store supports eight 4KB blocks, for a total of 32KB of memory.

The system has been thoroughly unit tested, and operates as specified. The implementation is highly parametrized. This allows for a significant amount of future design exploration. The following parameters are easy to change:

- Number of ASPorts
- Number of memory accesses per system clock cycles
- Number of handles in the handle table
- Memory power up time
- Size and quantity of SRAM blocks

The design has been successfully synthesized using Design Compiler, and easily meets the slow clock requirement intended for the system. Synthesis reports exhibit favorable power numbers, which support the practicality of a shared memory architecture such as the accelerator store. Design Compiler power estimates our dynamic power at 105uW and leakage power at 400nW at the 180nm TSMC process. We estimate this translates to a total system power of about 55uW at the 130nm UMC process used by our memory compiler. The accelerator store uses less than 33% of the power of one 4KB memory block and is roughly only 4.2% of the power consumed by all 32KB blocks if they were on. If, through the automatic VDD-gating of unused portions of memory, just one of the eight 4KB blocks can be powered down, the accelerator store has already more than made up for its power cost.

Area is roughly 21,700 um^2 and maximum frequency (at a target 100KHz external / 400KHz internal) to be 15.7MHz at 180nm TSMC. These numbers should only improve as we transition to 130nm. Area and timing results were beyond acceptable for our needs but could be rebalanced with modifications to our Design Compiler settings if desired.

We also validated our accelerator store implementation against our simulator for up to 150,000 cycles of execution from the application previously described. Although we were able to generate larger traces from our simulator, we were unable to run larger simulations due to trace file sizes exceeding our account disk quotas. However, we believe our 150kcycle trace is large enough to demonstrate the accuracy of our implementation as well as its low energy, area, and timing costs.

Conclusion

ASICs are a necessary component of embedded systems in today's world. Hardware accelerators used in such systems often take a black box approach and self contain all the memory they require, which can lead to inefficient use of power and area. Our work provides a solution to these inefficiencies, creating a shared memory framework for accelerators.

This framework addresses memory limitations of typical private memory architectures with the accelerator store, a centralized memory structure with built in power management, address translation stacks, and interrupt triggering.

The accelerator store simultaneously simplifies accelerator design by hiding all physical addressing information from the hardware accelerators, fulfilling the memory requirements of individual accelerators, and reducing leakage current by providing hardware managed VDD-gating.

Our implementation in Bluespec, validated by traces of a cycle accurate simulator, has confirmed the benefits of using a shared memory framework. If just one memory block out of eight in the accelerator store can be automatically VDD-gated, then the accelerator store has already more than made up for its power cost, thus justifying its existence.