Lab 6: Hardware-based Sorting

4541.763 Laboratory 6
Assigned: November 27, 2009
Due: December 10, 2009

1 Introduction

In the last two labs we explored taking a design and refining it iteratively for performance. This lab has the same high-level goal. However in this lab we will not tell you exactly what refinements to perform. Instead you will have to determine what should be done to get a good design.

Your design is a hardware-based sorter. We will give you a stream of unsorted data and it's length. You job is to take the unsorted data, sort it, and to store it into the external DRAM in ascending order(smallest value associated with the smallest address) on the XUP board starting from address 0. This problem is based on the 2008 MEMOCode design contest(`http://memocode.irisa.fr/2008/designcontest08/`). The MIT submission(`http://people.csail.mit.edu/mdk/papers/memocode_2008_cryptosorter.pdf`) is also worth reading, though you will not want to do be as aggressive about performance.

2 The Initial Sorter

We have provided you with an initial version of the sorter design in the `mit-6.375 lab6` module directory. The initial sorter is a very simple StmtFSM-based sort. We repeated walk down the array and swap incorrectly ordered consecutive pairs.

At a high-level, this is the algorithm:

```
swap = False;
while(!swap){
  swap = False;
  for(int i = 0; i < N; i++){
    a = mem[i];
    b = mem[i+1];
    if (a > b){
     mem[i]   = b;
     mem[i+1] = a;
     swap = True;
     }
  }
}
```

3 Making Your Design

Copy the `sort_array_exe.apm` and `sort_array_xup.apm` file in the `mit-6.375/lab6` directory to your user directory as `sort_array_modified_exe.apm` and `sort_array_modified_xup.apm` respectively. Similarly copy the `sort_pipeline_default.awb` module and the associated files to your workspace. Rename the awb file sort_pipeline_modified.awb. Do not forget to change the sort_pipeline value in your designs to point to the new pipeline.

4 The Memory Interface

This is the interface to the memory system, which consists of three methods. To start a transfer, send a command with the burst length and base address, tagged appropriately to indicate whether you are reading or writing. Then use either `readData` or `writeData` to transfer data into or out of the system.

```
typedef struct{
  t_ADDR addr;
  Bit#(TAdd#(1,TLog#(n_MAX_BURST))) size;
} BURST_COMMAND#(type t_ADDR, numeric type n_MAX_BURST) deriving(Bits,Eq);
typedef union tagged{
  BURST_COMMAND#(t_ADDR,n_MAX_BURST) ReadReq;
  BURST_COMMAND#(t_ADDR,n_MAX_BURST) WriteReq;
} BURST_REQUEST#(type t_ADDR, numeric type n_MAX_BURST) deriving(Bits,Eq);
interface BURST_MEMORY_IFC#(type t_ADDR, type t_DATA, numeric type n_MAX_BURST);
    method ActionValue#(t_DATA) readRsp();
    method Action writeData(t_DATA data);
    method Action burstReq(BURST_REQUEST#(t_ADDR, n_MAX_BURST) burstReq);
endinterface
```

4.1 Exploiting Burst Memory Transformations

The initial unsorted array is stored in DRAM Memory. To interact with the board we need to use bus transactions. As we discussed in class, bus transactions have an overhead. This is due to the fact that we must reserve the bus for each operation. While this overhead is only a few cycles for each for small operations which only need 1 cycle of the bus traffic, handshaking can waste 90% of the total bandwidth. To make this better, you should use bursts of a larger length. The interface we have will take care of much of the details in the use of memory.

The following statement FSMs shows you an example of a valid use of the multicycle usage for reading and writing a `length`-word burst to the memory.

```
Stmt makeRReq = seq
  for(i <= 0;i<length;i<=i+1) seq
      memory.readReq(BURST_MEMORY_REQUEST{addr:addr, size:length})
      action
        let x <- memory.readData();
        data[i] <= x;
      endaction
    endseq endseq;

Stmt makeWReq = seq
  for(i <= 0;i<length;i<=i+1) seq
      memory.writeReq(BURST_MEMORY_REQUEST{addr:addr, size:length})
      memory.writeData(data[i]);
      endaction
    endseq endseq;
```

You should keep in mind that the system handles bursts size that are power of 2 from 1-64. Commands that are not powers of two will get translated to many smaller (and therefore less efficient) bursts. Additionally, burst addresses to the DRAM should also be aligned to the burst length. That is if you are making an 8-word burst to address 40 (a multiple of 8) it will be much more efficient than an 8-word burst to address 41.

5 Interfacing with the Host System

One of the key points of getting your design working is to have it interface with the host system which is responsible for providing the data and checking the result. We isolated all interactions that your sort pipeline will have to do with the host system to the SortControl module. The interface for the SortControl is given below:

```
  interface SortControl;
     method Bit#(22) array_length();
     method Action start_sort();
     method Action end_sort();
     interface Get#(BURST_MEMORY_WORD) data;
endinterface
```

Your interaction with sort control should follow these steps:

1. All the `start_sort` method to signal that you are beginning your sort. The guard of this method will force you to wait until the rest of the system is also ready.
2. Use the `array_length` method to get the length N of the unsorted stream. This value will remain constant for the entire sort. **Note:** To make memory references easier we will limit the length to powers of two.
3. Grab the N unsorted values in the stream one at a time from the `Get` subinterface `data`.
4. Sort the data and storing it in order in the memory starting at address 0 (The smallest value should be at address 0, the largest at address N).
5. call the `end_sort` method to signal that you are done. This will also stop your counter and check the correctness of your design.

6 Grading

We will judge our designs solely on performance. There will be a required speedup that you will need to achieve. however we have not determined what speedup is reasonable to do, but still challenging for the lab. You should expect something within an order of magnitude of 40x speedup. We will send the class a concrete value to the class in a few days.

To encourage you to try not just to meet the requirement, but to make the best possible design you can, a special prize will be awarded the student with the best design.

7 Debugging

To help debug you designs it is useful to be able to observe the memory accesses. We have made two burst memory interfaces (found in `fpgaenv` ⇒ `low_level_platform_interface` ⇒ `burst_mem`). The first "Burst Memory Interface Using DDR Memory" uses the hardware DDR on XUP. The other module "Hybrid Burst Memory Interface" uses a software-based memory. This is of course much slower but will let you see internals of the data.

To look at the store memory take the `SortArray.cpp` (found in the sorting `connect_application` module "Array Sorting Application"). You can dump the first N addresses in the memory by calling:

```
BURST_MEM_SERVER_CLASS::instance.dump(N);
```

For debugging you may want to read multiple times during the sorting. Remember that the sorting hardware will be running concurrently so the results may not be coherent.

A useful case to consider for sorting is the fully reversed stream. This represents the worst-case for many sort algorithms. To switch the stream to be in reverse order, in the `SortArray.cpp` uncomment the line: `clientStub->SetReg(4,1);`.

8 Some Advice

1. In computer science when we consider the complexity of sorting we use comparison as the measure of complexity. In this case, the scarce resource is memory bandwidth, so you should consider measurements by how many memory reads are needed. This is closely related to the comparison count, but this may help you decide between sorting algorithms with the same complexity.

2. It is a good idea to have a predictable access pattern. This is necessary so that you can use memory bursts effectively. Our initial algorithm has this property.

3. While we want expect the final data from address 0 to address $N-1$ to hold the sorted array, we do not care about the state of the rest of the memory. You are free to use this extra memory space as a scratch pad. Bouncing data between two copies of the memory space, partially sorting the data each time, is a reasonable approach.

4. A simple way to get the benefits of memory bursts without changing how you access memory is to introduce a cache. Your hardware can continue to read and write 1 memory address at a time, and when data isn't in the cache it will be loaded and stored into the main memory in large bursts. Remember that writing into the cache is not the same as writing to the memory. You must flush the data back into the main memory to complete.

   There is an example of a very simple cache in Lecture 25 which is available on the website.