Digital Systems
EEE4084F

FINAL EXAM

11 June 2015

MEMORANDUM

This memo describes anticipated correct responses that the students should provide and the answers for multiple choice questions.

Section 1: Short Answers [46 marks]

Question 1.1 [10 marks]

1.1 (a) The student should highlight the design in terms of 8$\times$ SPE and 1$\times$ Power Processor Unit. They could go on to elaborate that a usual SMP, which most multicore PCs use, has a collection of the same processors. Drawbacks of the Cell processor includes the difficulty of dealing with more that one processor architecture, having to learn programming pragmas and the like in order to develop programs for the processor. Advantages over standard SMP include advantages of higher performance via the configuration facilities the platform provides in terms of the EIC (element interconnect bus), together with the separate memory controller and IO controller that offload arbitration and memory transfer needs. **[6]**

1.1 (b) An ABI, or application binary interface, is a specification regarding the instruction-level specifics of an interface between software programs. Examples include function calling convention, low-level data types, in which register(s) the return values are stored, etc. It ensures compatibility of code generators and portability of code.

Examples of ABIs for the cell processor are: IMB SPE ABI and Linux Cell ABI. **[2]**

1.1 (c) An API, or application programmer's interface, is a selection of function prototypes that an application programmer can use to make use of library and operating system functionality. The low-level specifics are hidden in abstraction.

An ABI refers specifically to the low-level specifics. **[2]**

Question 1.2 [12 marks]

1.2 (a)
- (i) Contiguous
- (ii) Interleaved / Cyclic / Block cyclic (the diagram does not make it clear whether it's pixel-cyclic or block-cyclic)
- (iii) Partitioned / Block partitioned
- (iv) Interlaced

[4]

1.2 (b) This is somewhat open-ended and depends on the argument of the student.

Using a block cyclic approach, with some overlap, would be effective, or block partitioned (cutting the data into two separate images with a 1-pixel overlap at the borders). This would allow the cores to work largely independently, except at the overlapping boundary regions. A mutex isn't required, as it is using separate input and output memory. Interlaced could also work, where one output line is written but 3 lines are read.

A single-pixel interleaved approach is irrelevant, also interlaced would not work unless there was it was three lines at a time with much overlap.

The block cyclic and block partitioned approaches are scalable regardless of the image size. Interleaved scalability would depend on the size of the image: interleaved with each core going across one row of the image may end up with cores being underutilized if there are more processors than rows of pixels.

Typically the data will be partitioned such that each partition fits into a `Cx`$\times$`Cy` block. The larger the image, the more blocks (and partitions) will be used.

[5]

1.2 (c) It is a highly coarse-grained, since there is little dependence between distant data elements. A new pixel value is only impacted by the values of its immediate neighbouring pixels.

This is in contrast to a fine-grained operation where each pixel may depend on pixels further away or (in an extremely fine-grained case) each new pixel being dependent on every other pixel.

[3]

Question 1.3 [12 marks]

1.3 (a) Each MathBOO contains 2 Mib of memory. This is enough to store 65 536 single-precision floating point numbers. Therefore, if the vectors are smaller than 32 000 elements each (to leave space for the answer and other temporaries), and there are lots of such scalar products that must be calculated, it is worth while to load each MathBOO with one complete scalar product. The MultiMath can then calculate $\mathtt{N}^2$ scalar products at the same time, without the need to access the shared memory.

If, on the other hand, the vectors are much larger, each MathBOO can only calculate a portion of the scalar product. In this case, the mutually exclusive access to the shared memory becomes a bottleneck when the sum of the products must be taken. I would program as much of the vectors into each MathBoo as possible, so that communication between cores is minimised. **[5]**

1.3 (b) For matrices of size $\mathtt{M} \times \mathtt{M}$, multiplication involve the calculation of $\mathtt{M}$ scalar products. In order to minimise communication with the shared memory, the CPU might load a complete row of the first and column of the second matrix into each MathBOO (I'm assuming that $\mathtt{N} < 32\,000$, which is a reasonable assumption given modern technology). This involves significant data duplication, but I believe that the extra overhead is worth the effort so that communication with the shared memory can be avoided.

Each MathBOO can then calculate an independent element of the result matrix.

If the memory bandwidth between the MultiMath and the CPU is of a similar magnitude as the internal memory bandwidth of the MultiMath shared memory, it is worth it to let the CPU perform the data duplication directly to each MathBOO. If, on the other hand, the internal MultiMath memory bandwidth is significantly larger, the CPU can copy the matrices to the shared memory and each MathBOO can perform the data duplication as required.

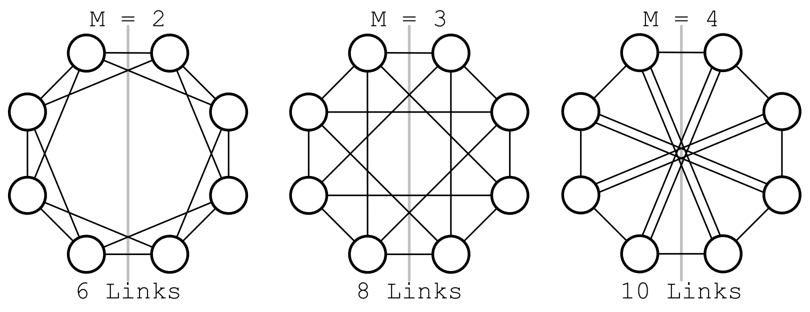
The result can remain in the MathBOO, where the CPU can fetch it directly. **[5]**

1.3 (c) The addition of IPC would likely be of little benefit in this case because it may cause a bottleneck, whereby one core is waiting for results from another core, or waiting for another core to complete communication with a different core before it can send its data on. For example, if core $\mathtt{X}$ was gathering from the other cores, it may likely become a bottleneck, especially since each core is likely to take the same amount of time in getting through a row $\bullet$ column operation. **[2]**

Question 1.4 [12 marks]

1.4 (a) `6`, $\forall$ `N` $>$ `3` **[1]**

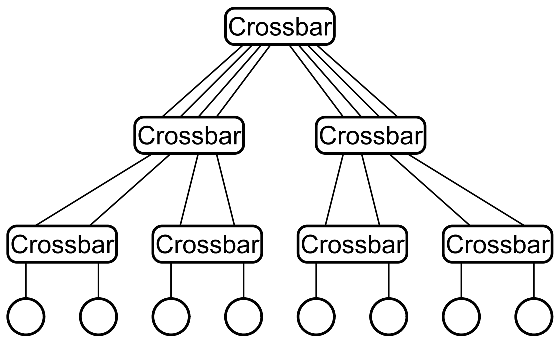
1.4 (b) Consider the cases for `N = 8` below:



A pattern emerges. For the case where `M = N/2`, the bisection width is `N+2`. For every reduction in `M`, two fewer links cross the bisection line. The general expression for the bisection width is therefore `2M+2`.

The bisection bandwidth is therefore `(2M+2)·32bit / 2ns`. This may be simplified to `32(M+1)` Gb/s. **[4]**

1.4 (c)



[3]

1.4 (d) Per definition of the binary fat tree, the bisection width is `N/2`. **[1]**

1.4 (e) Each node must execute:

1. Send element to distant node (2 ns)
2. Receive element from distant node (2 ns)
3. Swap locations in local memory (2 ns)

The bisection bandwidth is sufficient to perform this in parallel without bottleneck, so the total time is 6 ns. The first two steps are independent due to the fact that links are half-duplex. Half the nodes must swap the order of the first two steps. The need for the final step is debatable. **[3]**

Section 2: Multiple Choice [34 marks]

Choose one option for each of the questions 2.1 to 2.10 in this section.

2.1 (d) **[3]**

2.2 (c) **[3]**

2.3 (d) **[3]**

2.4 (a) **[3]**

2.5 (b) **[3]**

2.6 (b) **[3]**

2.7 (d) **[3]**

2.8 (b) **[3]**

2.9 (c) **[3]**

2.10 (d) **[3]**

2.11 True / False questions: **[4]**

(a) False
(b) True
(c) True
(d) False

Section 3: Long Answers [40 marks]

Question 3.1 [8 marks]

3.1 (a) All the transfer links have 10% overhead (the DDR has a refresh state and the PCIe has headers, etc.). The transfer rate can therefore be compared based on the link bandwidth exclusively.

Link	**Transfer rate**
DDR3 1600	800 MHz · 2 · 128 b = 204.8 Gb/s
PCIe 1.0	= 4.0 Gb/s
LPDDR2 320 MHz	320 MHz · 2 · 32 b = 20.48 Gb/s

The PCIe link is the bottleneck. **[3]**

3.1 (b) The CPU will take $128^3 \cdot 5$ = 10 485 760 instructions to complete. There are 4 cores, each processing instructions at a rate of 25·3GHz = 75 GIPS. The CPU will therefore take 34.95 µs to encrypt each block, resulting is an encryption rate of **28 610 blocks/s**. **[5]**

Question 3.2 [10 marks]

3.2 (a) To read and write will take the same amount of time. To read: 10 / 320MHz + 4096b / 320MHz / 2 / 32b = 231.25 ns. To copy is twice this: **462.5 ns**. **[3]**

3.2 (b) Ignoring the refresh state will imply that the maximum rate is 1 / 462.5ns = 2 162 162 blocks/s. The refresh state removes 10%, resulting in **1 945 946 blocks/s**. **[3]**

3.2 (c) Each processing pipeline processes blocks at a rate of 1 block every 4 096 / 10MHz = 409.6 µs, which implies 2 441 blocks/s. By using the memory performance calculated in the previous question, one can deduce that the maximum number of processing pipelines the DRAM supports is 1 945 946 blocks/s / 2 441 blocks/s = **797**. **[4]**

Question 3.3 [22 marks]

3.3 (a) We need to consider each step individually:

Step	Time
1	10 ns
2	4 096N / (4 Gb/s·0.9)
3	1 ns
4	8 192 / (4 Gb/s·0.9)
6	10 ns
7	4 096N / (4 Gb/s·0.9)
8	1 ns

By taking the sum, we obtain **(N+1)·2.276µs + 22ns**. **[6]**

3.3 (b) When $N \leq 100$, all the pipelines only have one block in them. The processing therefore takes 20 480/10 MHz = **2.048 ms**, independent of N. To transfer the data to and from the FGPA DRAM takes N·462.5 ns. For $N \leq 100$, this is 2.26% of the processing time and can therefore not be ignored. The 1 ns overhead for the interrupt can be ignored. The time it takes to process N blocks is therefore N·**462.5ns + 2.048ms**.

When $N \gg 100$, all the pipelines are full. The overhead involved because the pipelines are not full for the first 400 blocks, or the last 400 blocks, can be ignored as a small effect (for large enough N). The DRAM transfer can also be ignored. The time it takes to process N blocks is therefore 4 096N / 10MHz / 100 = N·**4.096 µs**. **[8]**

3.3 (c) Let us assume that this occurs when $N \leq 100$. The total time it takes when processed on the FPGA is the sum of all the steps, which can be approximated as (N+1)·2.276µs + N·462.5ns + 2.048ms. The CPU takes N·34.95µs. By equating these two we obtain N = 63.65. N cannot be fractional, so we round up. It becomes worth while to use the FPGA for $N \geq$ **64**. **[4]**

3.3 (d) For 100 MiB, N = 204 800, which may be considered large. The FPGA will take (N+1)·2.276µs + N·4.096µs, which can be simplified (for large N) to N·6.372 µs. The CPU takes N·34.95 µs. The expected speed-up is therefore 34.95/6.372 = **5.48**. **[4]**

END OF EXAMINATION