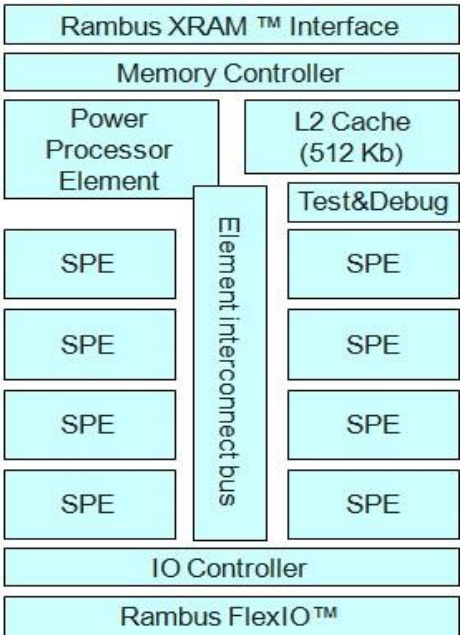
EEE4084F QUIZ 3 2014 - ANSWERS

Q1 (i) d
(ii) Y
(iii) N
(iv) Y

Q2 In the cell processor, there is a separate I/O controller which any one of the processor cores can communicate with via the EIB. For example, for an SPE to perform I/O, it needs to send requests, via the EIB, to the I/O Controller. The individual SPEs thus do not need to rely on the PPU to perform the I/O processing.



Q3 i. ABI = Application Binary Inteface
ii. The ABI defines data types, register usage, calling conventions, and object formats to ensure compatibility of code generators and portability of code. i.e. It doesn't define the operation and parameter lists of specific methods as is the case for an API. An ABI provides an added level of modularity, in terms of making an API compatible with a number of execution platforms, at the lower level, without having to manually implement parameter translation routines. Also it gives more flexibility, but allowing the implementation of particular data types for instance to be separated, and abstracted out, of the API definition.
iii. Examples for the Cell Processor are: IBM SPE ABI, and the Linux Cell ABI

Q4 i. f represents the portion of the program that will run in parallel; making 1 - f the portion that runs in series (isn't parallelized). The parameter n is the number of processing nodes, if n=1 then it is just running on a single processor.

ii. This could be done experimentally by using timers - if you may make changes to the code. For example, start one timer at the start of the program and end it at the end of the program, to find the time the whole program takes to run. In a simple case, where there is a clear sequential part that branches at a point into parallel execution and then some sort of join, it can be simply: read the timer value just before the parallel execution starts in order to find the start-up time, and then read it again after the join to see the time spent in the parallel portion. From this you can get, probably a fairly accurate estimate, for f, where

f = (time at join - time at end of initialization) / total time

Alternatively, one would need to log entry and exit times and counts to threads, getting reports such as

t=1 entry 1 of thread 1
t=2 enter 2 of thread 1
t=3 exit 1 of thread 1

... etc. and from this do calculations to see what sort over overlap is established between the executing threads, as suggested in the diagram to the right.

Q5(i) f = fraction of computation that can be parallelized
1 - f = fraction that has to be sequential
n = number of processors / processing nodes

Q5(ii) The value f would could be determined experimentally using times within the code. Essentially you could do a rough calculation to find f for the parallelized portion by:

A = time at join - timing at start of threads, and B would simply be
B = time at end of program - time at start of program

So, f = A/B

Of course this would give quite an overestimate most likely because there would be some delays in the thread scheduling, semaphore handling, etc.

Q5(iii) It is more likely a worst case predition for a coarse grained problem. Considering that a course grained problem has less inter-dependence between datum and this likely suited to being easy to parallel. My logic for choosing course grained is that you need to remember worstcase means likely overestimate, i.e. it is likely to work better in most cases. The calculation is more likely a 'best-case' for a a fine grained problem considering that a complex fine grained problem may degenerate into sequential performance because each datum might be dependent on the rest of the data.

Q6 Service Oriented Architecture