Answers to Advanced Computer Architectures 4

1. a. *A pipelined processor will suffer from problems arising from control-flow dependencies. Describe:*

i. *how these problems arise;*

Under normal circumstances, instruction fetching is linearly contiguous. However, at jumps the flow is discontinuous. This is not a problem because JUMP instructions can be detected in the early part of the pipeline and the target address used to control subsequent instruction fetches (the pipeline may stall temporarily if the next instruction is not in the program cache). However, a problem occurs if a conditional jump is encountered. Consider the following section of code:

```
ADD    R3,R1
CMP    R1,R2
JEQ    @target
```

At the point where a decision must be made about whether to continue fetching instructions contiguously or to start fetching from @target, neither the ADD nor the CMP instruction have been executed. Therefore, the state of the Z flag which controls whether the JEQ jumps is not yet valid. In fact the decision is not known until the CMP instruction clears the Execute stage and only at this point in time is the address of the next instruction known.

(3)

ii. *the methods used to overcome these problems.*

Do nothing – The pipeline stalls and NOPs are inserted. This is very inefficient because, effectively in the case shown above, two non-useful instructions are executed every time a conditional jump is encountered.

Fixed choice – accept the fact that if it is the wrong choice then the contents of the first part of the pipeline will need to be discarded. No irrevocable changes must be made before the final decision is made otherwise the proper operation of the program will be destroyed. It is possible to write a compiler so that it generates object code for looping in particular such that a conditional jump is always taken except for when the loop is complete, or such that the conditional jump is only taken when the loop is complete. Obviously, depending on the choice which the compiler makes, this scheme may behave well or badly – the compiler requires knowledge of the architecture.

A Branch History Cache (BHC) – this is a cache which records the choice made whenever a conditional jump is taken at a particular address. Whenever a conditional jump is encountered, the instruction address is applied to the BHC and if there is a hit the choice made last time is used to direct which stream to fetch (A or B). When the true decision is known, the contents of the cache are updated and if the choice was wrong, the early part of the cache is flushed and the correct stream is fetched.

Some BHCs employ a scheme whereby the wrong decision must be made twice in succession before the contents of the BHC are updated.

Delayed Jumps - an intuitive approach which depends upon the compiler. As the name suggests, a delayed jump is one whose effect is delayed relative to its position in the program. That is, it causes a jump some time after its position in the program - not at the point of its occurrence. Consider the initial example with some prior instructions shown (in this case the effect of the jump is delayed by 3 instructions):

```
inst1
inst2
inst3
ADD     R3,R1
CMP     R1,R2
JEQ@target
```

To operate effectively, the compiler must be able to find a number of prior instructions which are definitely executed but on which the result of the comparison does not depend. The compiler re-orders the instruction stream and changes the JEQ to DJEQ (delayed jump equal) as follows:

```
ADD     R3,R1
CMP     R1,R2
DJEQ    @target
inst1
inst2
inst3
```

Obviously, there will be cases where sufficient, un-related prior instructions cannot be found in which case the compiler pads the instructions with NOPs. As pipelines become longer, this would be increasingly the case.

(4)

b. *In some cases, a reorder buffer is used to solve problems caused by data/control dependencies in such processors.*

i. *Describe how such buffers are used and, in particular, identify the difference between instructions completing and committing.*

A reorder buffer is a queue which stores information about instructions (registers, etc.) and has a head and tail pointer associated with it. Information about instructions is held on the queue in the order in which the instructions occur in the instruction stream. That is, the order in which they would have appeared in an unpipelined, simple processor. Forwarding can still used but works differently.

As instructions complete, results (to be written to registers) are stored in the appropriate entry, associated with the instruction, of the reorder buffer and are *not* written to the register

As instructions are despatched, source register operands can be read from the register *unless* there is a result yet to be written to the particular register on the reorder buffer. If the source operand is to be retrieved from the buffer then it must be from the oldest entry on the queue that is ahead of the instruction requiring the source operand (there could be multiple references to a register on the queue). Instructions can complete in any order (because results are written to the reorder buffer *only*). If the instructions complete out of order then the result may not have been written to the queue - in this case the address on the queue will be sent to the reservation station and forwarding is used

An instruction is committed when it reaches the head of the reorder buffer *and* it has completed. When an instruction is committed its result is written to the corresponding register.

Reorder buffers support speculative execution - that is executing an instruction before it is known whether it should be executed - following a *Jcond* instruction. In this case, both streams could be executed with the completed instruction results being written to the reorder buffer. However, instructions along a particular branch cannot committed until the *Jcond* instruction is committed. Instructions along a stream can be 'coloured' (that is, all the instructions are tagged to be identifiable as a group). If the branch is wrong then the reorder buffer can be flushed (should still be used with a branch-history cache to minimise flushing). **(5)**

ii. *For the following code snippet, show how a reorder buffer might be used:*

```
     LOAD    @addr,R1      ; R1 ← addr
     MUL     R1,R2,R3      ; R3 ← R1 x R2
     CMP     R3,10         ; is R3 equal to 10
     JEQ     L1
     ADD     R4,R5,R5      ; R5 ← R5 + R4
     JMP     L2
L1   ADD     R4,R6,R6      ; R6 ← R6 + R5
L2   …
```

The LOAD is from memory and let us assume that it takes some time to complete (that is a number of clock cycles) – it is put onto the reorder buffer queue but nothing behind it can be committed until it completes. The following MUL instruction, which depends on the value loaded into R1 (which is not available) is put on to the reorder buffer queue and the address for R1 (associated with the entry for LOAD on the queue is sent to the reservation station for this operation along with, presumably, the value in R2. Similarly, CMP is put on the reorder buffer and the instruction along with the address for R3 on the queue is sent to the corresponding reservation station. The JEQ results in a lookup to the BHC, which, for example says do not jump. Consequently, the ADD and JMP are speculatively executed with the results (which do not depend on the load being stored onto the reorder buffer. Thus, at some point, the ADD has completed but cannot be committed because there are instructions ahead of it on the queue. The LOAD completes and is committed because it is at the head of the queue. The value loaded is also forwarded to the reservation station where the MUL is waiting and this instruction completes with the result being written to the reorder buffer and forwarded to where the CMP is waiting to execute. The MUL is committed. The CMP yields a result and is committed. If the right choice was made then the ADD, which has already completed is also committed (along with any subsequent instructions that may already have been execute). If the wrong choice was made then the speculatively executed instructions are discarded from the buffer – they have, effectively, not been executed. If any speculation along the other branch was done then these instructions continue. If no speculation was done then the instructions can now be executed. **(6)**

c. *What is meant by precise interrupts and a precise architectural state and what is their relevance to reorder buffers?*

At any point an external interrupt could occur or an instruction could cause an exception. In any event, there is a point in the instruction stream, between two instructions – that could be in flight, where the precise interrupt is deemed to occur and all instructions *up to this point* must be completed and committed and all instructions beyond this point must, essentially, be discarded. The precise architectural state is the exact state that the processor must be in when the interrupt occurs and this is the state that must be achieved before the interrupt can be serviced. Essentially, all instructions (and the effects i.e. entries on the reorder buffer and execution units) after this point must be discarded and can be restarted after the interrupt servicing is complete. **(2)**

2. *A common way to perform a division, y/R, in a floating point system, is to calculate the reciprocal of the divisor, R (=R^{-1}), and calculate the product $y.R^{-1}$. The reciprocal of R can be calculated using a Newton Ralphson approach by using an iterative calculation:*

$$x_{n+1} = 2x_n - Rx_n^2 \qquad (1)$$

where x_0 is seeded with an approximation to R^{-1} and the iterative equation is calculated for increasing values of $n \rightarrow N$ until x_N converges on the value of R^{-1}.

You know that the input number, R, is a normalised floating point number stored in the form $m \times 2^e$ where m, the mantissa, is in the range 0.5…1 and e, the exponent is an integer in the range -128…127.

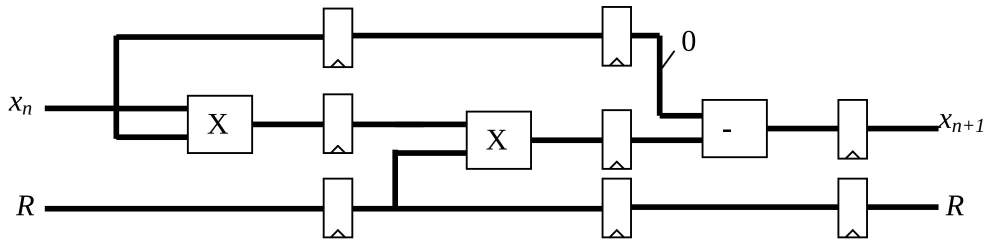
To set the initial seed, a value of x_0 has to be chosen that is less than R^{-1} but is close enough to ensure that x_n converges to R^{-1} quickly. You can show that the number of iterations, N, required to produce an accurate value of R^{-1} can be approximated by:

$N=4 - \log_2(x_0R)$

a. *Based on the format of the numbers, show that the number of iterations required can be limited to 6.*

Based on the knowledge that a number, *R*, is represented as $m \times 2^e$, and that $0.5 \le m < 1$. This means that $R^{-1}=1/m \times 2^{-e}$ and $1 < 1/m \le 2$. Consequently, rather than generating x_0, based on *R*, x_0 can be set to a value less than 1. The question is a bit ambiguous and so an answer anywhere between 0.5 and 1 would be accepted. In this limiting case where x_0=0.5, $N=4-\log_2(0.5x0.5)=4+2=6$. **(4)**

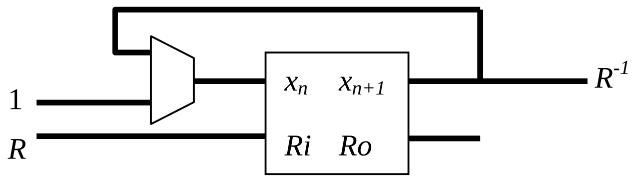
b. *Based on the result shown in part **a.**, design a pipelined datapath block that will allow you to calculate one iteration of the equation (1).*



The organisation that passes *R* through in synchronism with *x* seems to be a little redundant given that its value is fixed for a calculation but it supports the inclusion of the block in a bigger datapath. I would mark it as being correct without this but would assign 6 out of the 8 marks. **(8)**

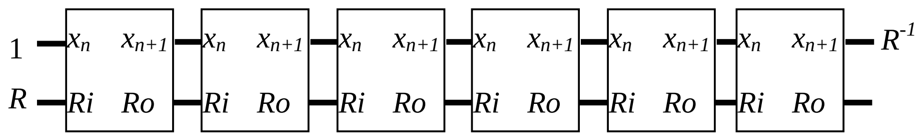
c. *Show how the block in part* **b.** *could be used within a larger schematic to:*

i. *Calculate the reciprocal, R^{-1}, whilst utilising the minimum hardware.*



This minimum hardware solution uses one instance of the data block and recirculates the data 5 times to calculate R^{-1}. Although not asked for, it is worth noting that, to avoid the block having to meet the worst-case word width requirements, in practice the block would be designed to work only with those bits of the output that would be altered during any particular iteration. This is beyond the scope of the question. **(4)**

ii. *Calculate the reciprocal, R^{-1}, whilst achieving the greatest throughput possible*



The same issues that pertain to the solution in part **i.** also apply to this one. In this case, the 6 iterations are unrolled into a sequence and a new value of R can be input at every clock cycle (for which cases, the delay of R in part **b.** is necessary). **(4)**

3. **a.** *State Flynn's Taxonomy – giving an example for each classification.*

Single Instruction Single Data (SISD). This describes a simple uni-processor where a single program is executed on one set of data.

Single Instruction Multiple Data (SIMD). This describes, for example, an set of processors which are constrained to execute the same program one separate sets of data. The implication being that the overall task will be executed more quickly because each part is executed in parallel. This paradigm requires that the task is regular and can be decomposed.

Multiple Instruction Multiple Data (MIMD). This describes a general purpose multi-processor where multiple processors can each work independently on their own data.

Multiple Instruction Single Data (MISD). This classification appears to be anomalous because it seems to imply that multiple, different operations are applied to the same data *which remains the same data*. However, this category could cover fault-tolerant processing (e.g. triple modular redundancy) where multiple processors process the same data yielding the same results. **(4)**

b. *Describe the organisation of an SIMD Array Processor. Ensure that you identify what makes it SIMD.*

Answer based on the diagram here. Identifying:

Common control path that ensures that all processors work in lockstep i.e. single instruction operation simultaneously on multiple data.

Array of simple, 1, 2, 4 bit processors working nibble serially.

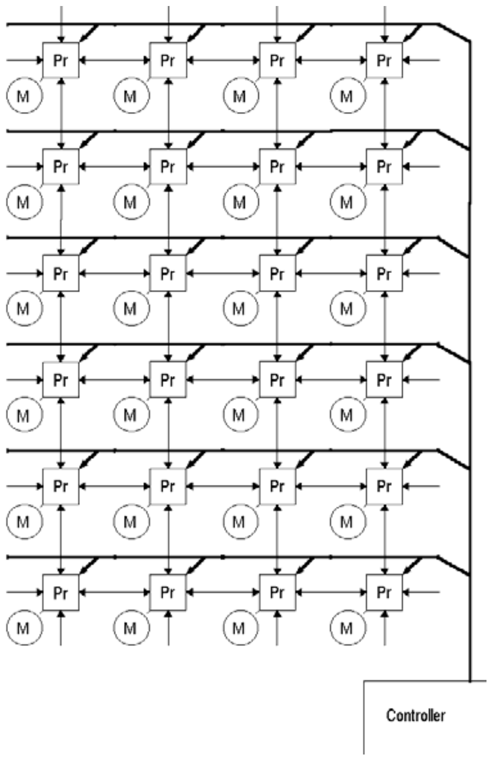
Local memory

Communication links.

PE-PE communication – primitive part of the instruction set

I/O mechanism at boundary

Compact implementation to create IC with large numbers of PEs



(6)

c. *A square-connected NxN SIMD array holds an array of data, f(x,y), distributed across it so that f(x,y) is held in memory location 0 of processor PEx,y (in the x*th *column and y*th *row). The array of data is to be sorted, within rows, so that the biggest datum from row y of data is held in PE0,y, the next biggest in PE1,y, the next in PE2,y, all the way to the smallest datum being held in PEN-1,y.*

The PEs support the following instructions:

```
BCAST  <a>         ; output memory location <a> to N,S,E,
                     and W
INd    <a>         ; input from d (=N,S,E, or W) to memory
                     location <a>
INd    <a>,F       ; input from d (=N,S,E, or W) to memory
                     location <a> but only if F=1
MOVE   <a>,<b>     ; copy memory location <a> to <b>
MOVCA  <a>         ; a PE-dependent instruction that loads
                     the column address of the PE into
                     address <a>
MOVF   <a>         ; copy the lsb of the value in address
                     <a> to F
CSWP   <a>,<b>,F   ; exchange memory locations <a> and <b>
                     if the contents of <a> is greater than
                     the contents of <b> but only operate if
                     F=1
AND    <a>,#K      ; logical and of value in address <a>
                     with constant value K
INV    F           ; Invert the F flag
```

Write a program that will sort the rows, as described above.

Hint: if each column is always performing a calculation then a program will not work but note that the CSWP instruction is conditional and the column address can be used as an operand …

This requires a bit of thought – hence the hint at the bottom of the section of the question. Essentially, the program should compare the values in two horizontally adjacent processors and the westward one should keep the bigger and copy the smaller back eastward. However, it is important that only one copy of each individual value is kept and if all processors are engaged on this then it is quite easy for the same value to go in two different directions at the same time – this will not work. In this case, we can avoid the problem by doing it in processors whose column address is odd and then in processors whose column address is even and then repeating sufficient times for values to bubble from one end of a row to the other: hence, the MOVCA, MOVF and conditional CSWAP instructions. An additional constraint is that the W connection on PE(x,0) must supply a big positive number whilst the E connection on PE(x,N-1) must supply a big negative number.

Continued …

```
MOVCA  1          ; get the column address

MOVF   1          ; put the LSB into the flag register i.e.
                    odd or even

Repeat remainder N/2 times

BCAST  0          ; output the value held in each PE

INE    1          ; read the value from the next eastmost
                    processor

CSWP   1,0,F      ; swap the bigger for the smaller but
                    only in odd columns

BCAST  1          ; re-broadcast the values (which may have
                    changed in the odd columns). In even
                    columns no change will have been made

INV    F          ; invert the flag so that any instruction
                    that depends on F being 1 will now only
                    operate in an even column

INW    0,F        ; read the value back in – in even
                    columns the value will overwrite the
                    original data, in odd columns it will
                    be discarded

                  ; the section above is repeated but now,
                    because F was inverted in the first
                    section, it only swaps in even columns
                    and reads back to odd columns when F is
                    inverted below

BCAST  0          ; output the value held in each PE

INE    1          ; read the value from the next eastmost
                    processor

CSWP   1,0,F      ; swap the bigger for the smaller but
                    only in even columns

BCAST  1          ; re-broadcast the values (which may have
                    changed in the even columns). In odd
                    columns no change will have been made

INV    F          ; invert the flag so that any instruction
                    that depends on F being 1 will now only
                    operate in an odd column

INW    0,F        ; read the value back in – in odd columns
                    the value will overwrite the original
                    data, in even columns it will be
                    discarded
```

(10)

4. *A set of N interconnected processors (where N is odd) is organised in a ring, as shown in **Figure 4**. Each processor is connected to its neighbouring processors by communication links that will transfer a message between the two connected processors in R_E seconds. An application, consisting of M tasks, is distributed equally across the N processors and during the course of the application each task sends T messages to every other task. If a message is sent from a processor to a non-adjacent processor it is done by store-and-forward via all of the intervening processors (using the shortest route around the ring). If a message is sent between two tasks on the same processor, the communication is assumed to take a time equal to R_I.*

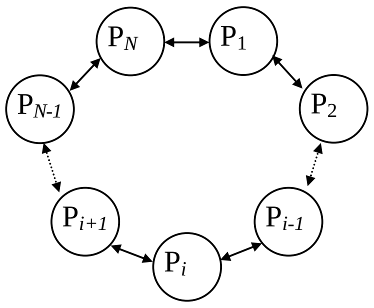


Figure 4: Processor Ring

a. *Show that the total number of messages transmitted between any two processors (that is, the message originates on one processor and finishes its journey at the other processor) is:*

$$2\frac{M^2}{N^2}T$$

If there are M tasks spread across N processors then there are M/N tasks per processor. Considering any two processor, the total number of tasks is $2M/N$ and the total number of uni-directional inter-task links is:

$$\frac{2M}{N}\left(\frac{2M}{N}-1\right)$$

By a similar argument, the number of links between the M/N tasks residing in the same processor is:

$$\frac{M}{N}\left(\frac{M}{N}-1\right)$$

and, consequently, the total number of uni-directional links that go between the pair (any pair) of processors must be:

$$\frac{2M}{N}\left(\frac{2M}{N}-1\right)-2\frac{M}{N}\left(\frac{M}{N}-1\right)=2\frac{M^2}{N^2}$$

If each task transmits T messages to every other task then the total number of messages transmitted between a pair of processors will be:

$$2\frac{M^2}{N^2}T$$

(4)

b. *From the result in **a.** and fact that messages between any two non-adjacent processors appear as individual messages across each separate communication link, show that the total number of messages sent during the execution of the application across each communication link is:*

$$\frac{T}{4}\frac{M^2}{N^2}(N-1)(N+1)$$

The result in **a.** is true for any pair of processors. Now we need to work out the total number of inter-processor links for all the processors – remembering that a link that passes between non-adjacent processors will generate messages on each link. Starting with P_1, this processor connect to P_2 over 1 link, P_3 over 2 links, P_4 over 3 links all the way up to $P_{(N+1)/2}$ (given that N is odd, this will be an integer) over $(N-1)/2$ links. Consequently, the total number of individual links between P_1 and these processors is:

$$\sum_{i=1}^{\frac{N-1}{2}} i = \frac{1}{2}\left(\frac{N-1}{2}\right)\left(\frac{N-1}{2}+1\right) = \frac{1}{8}(N-1)(N+1)$$

By an argument of symmetry, the total number of links between P_1 and all the processors will be:

$$\frac{1}{4}(N-1)(N+1)$$

Aggregating this across all of the N processors yields:

$$\frac{1}{4}N(N-1)(N+1)$$

but this replicates the total number of links between each pair of processors. Consequently, the total number of processor to processor links passing between processors is:

$$\frac{1}{8}N(N-1)(N+1)$$

Multiplying this by number of task to task messages between each pair of processors (from **a.**) yields:

$$\frac{1}{8}N(N-1)(N+1)2\frac{M^2}{N^2}T = \frac{1}{4}(N-1)(N+1)\frac{M^2}{N}T$$

This is the total traffic across the N links in the network of processors and by an argument of equivalence all of the links are equally loaded. Hence, the traffic across a single link must be:

$$\frac{T}{4}\frac{M^2}{N^2}(N-1)(N+1)$$

(6)

c. *Each task involves an internal computation time of C seconds and you may assume that internal computation is not overlapped with communication via the links. However, you may assume that all of the internal computation runs in parallel on the processors and the communication links can all operate in parallel with each other. Show that the speed-up involved in running the application on the ring of processors (relative to running the application on a single processor) is:*

$$speedup = \frac{N\left(1+(M-1)\frac{TR_I}{C}\right)}{1+\frac{1}{4}\frac{TR_E}{C}\frac{M}{N}(N-1)(N+1)+\frac{TR_I}{C}\frac{(M-N)}{N}}$$

There are M tasks, distributed across N processors giving M/N tasks per processor. Consequently, the time taken to complete the internal computation will time be:

$$\frac{MC}{N}$$

Given that there are:

$$\frac{M}{N}\left(\frac{M}{N}-1\right)$$

uni-directional links between the *M/N* tasks on the same processor, the internal communication time in each processor will be:

$$\frac{TR_I M}{N}\left(\frac{M}{N}-1\right)=\frac{TR_I M}{N^2}(M-N)$$

The computation time, summed with the internal and external communication time is the overall time, because all the links work in parallel and all the processors work in parallel but computation is not overlapped with communication. Consequently, the total time for the application running on the network is:

$$\frac{MC}{N}+\frac{TR_I M}{N^2}(M-N)+\frac{TR_E}{4}\frac{M^2}{N^2}(N-1)(N+1)$$

If only one processor is involved, *N*=1, then this expression becomes:

$$MC+TR_I M(M-1)$$

The speed-up is, therefore:

$$\frac{MC+TR_I M(M-1)}{\frac{MC}{N}+\frac{TR_I M}{N^2}(M-N)+\frac{TR_E}{4}\frac{M^2}{N^2}(N-1)(N+1)}$$

Multiplying this by *N/CM* yields:

$$\frac{N+\frac{TR_I N}{C}(M-1)}{1+\frac{TR_I}{CN}(M-N)+\frac{TR_E}{4C}\frac{M}{N}(N-1)(N+1)}=speedup$$

Therefore:

$$speedup=\frac{N\left(1+(M-1)\frac{TR_I}{C}\right)}{1+\frac{1}{4}\frac{TR_E}{C}\frac{M}{N}(N-1)(N+1)+\frac{TR_I}{C}\frac{(M-N)}{N}}$$

(8)

d. *Show that the parallelisation of the algorithm is only worthwhile if:*

$$\frac{C}{\left(\frac{R_E}{4} - R_I\right)T} > M$$

For the case where N (and, hence, M) is large (and the above is an approximation).

Speedup>1. Therefore,

$$N\left(1 + (M-1)\frac{TR_I}{C}\right) > 1 + \frac{1}{4}\frac{TR_E}{C}\frac{M}{N}(N-1)(N+1) + \frac{TR_I}{C}\frac{(M-N)}{N}$$

Replacing *M-1, N-1, N+1* by *M* or *N* (they are large).

$$N\left(1 + M\frac{TR_I}{C}\right) > 1 + \frac{1}{4}\frac{TR_E}{C}MN + \frac{TR_I}{C}\frac{(M-N)}{N}$$

$$(N-1) + \frac{TR_I}{C}\left(MN - \frac{(M-N)}{N}\right) > \frac{1}{4}\frac{TR_E}{C}MN$$

recognising that *MN* is much greater than (*M-N)/N* yields:

$$N + \frac{TR_I}{C}MN > \frac{1}{4}\frac{TR_E}{C}MN$$

$$1 > \frac{TM}{C}\left(\frac{R_E}{4} - R_I\right)$$

Therefore:

$$\frac{C}{\left(\frac{R_E}{4} - R_I\right)T} > M$$

(2)

NLS / NA