# DEPARTMENT OF ELECTRONIC AND ELECTRICAL ENGINEERING

## Autumn Semester 2013-14 (2 hours)

## Answers to EEE6031 Advanced Computer Architectures 6

**1. a.** *The pipeline shown in **Figure 1** is intended to process data as follows:*

*In →P1→ P2→ P3→ P2→ P4→ P3→ P2→ P3→ P4→ Out*

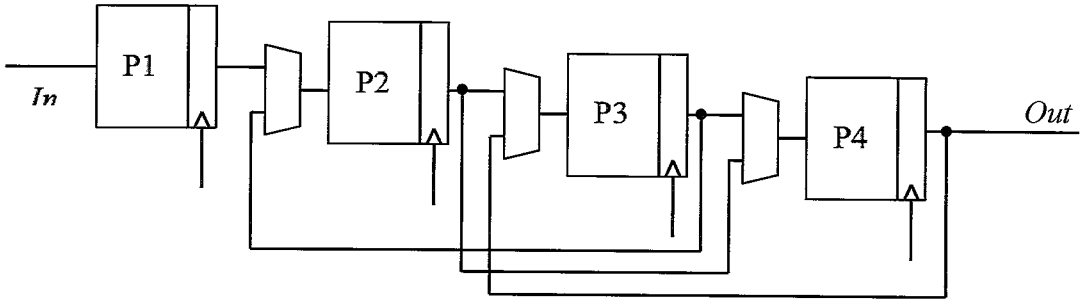



***Figure 1: Processing Pipeline***

*For this processing activity:*

**i.** Write down the reservation table;

| Clk | P1 | P2 | P3 | P4 |
|---|---|---|---|---|
| 0 | D1 | | | |
| 1 | D2 | D1 | | |
| 2 | | D2 | D1 | |
| 3 | | D1 | D2 | |
| 4 | | D2 | | D1 |
| 5 | | | D1 | D2 |
| 6 | | D1 | D2 | |
| 7 | D3 | D2 | D1 | |
| 8 | D4 | D3 | D2 | D1 |
| 9 | | D4 | D3 | D2 |
| | | D3 | D4 | |
| | | D4 | | D3 |

Etc.

**(8)**

**ii.** *Identify the data throughput and the processor utilisation;*

The data throughput is 2/7 datum/clock cycle and the processor utilisations are:

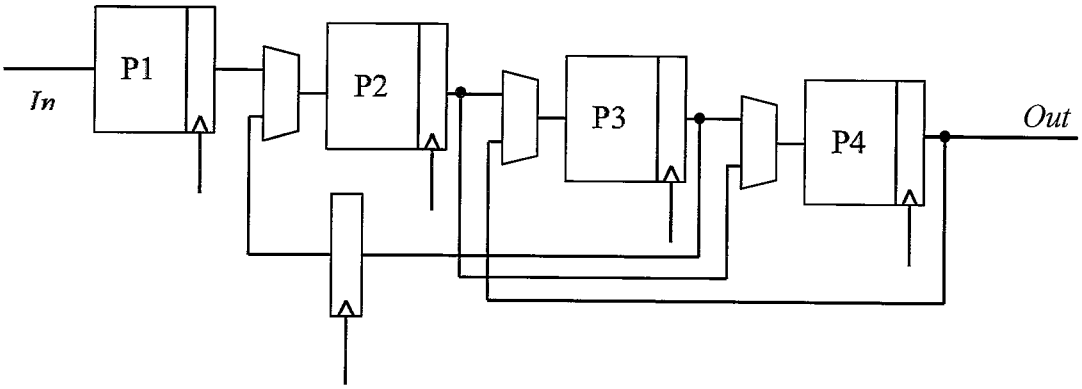
P1 = 2/7, P2 = 6/7, P3 = 6/7, P4 = 4/7

**(2)**

**b.** *You notice that adding a single register, acting as a delay, will improve the data throughput of the pipeline in **Figure 1**.*

**i.** *Where should you put the delay?*

A good place to put the delay would be in the branch feeding back from the output of P3 to the input of P2




(4)

**ii.** *How much will the performance improve?*

We can redraw the reservation table adding Del to represent the occupancy of this delay:

| Clk | P1 | P2 | P3 | Del | P4 |
|---|---|---|---|---|---|
| 0 | D1 | | | | |
| 1 | D2 | D1 | | | |
| 2 | D3 | D2 | D1 | | |
| 3 | | D3 | D2 | D1 | |
| 4 | | D1 | D3 | D2 | |
| 5 | | D2 | | D3 | D1 |
| 6 | | D3 | D1 | | D2 |
| 7 | | | D2 | D1 | D3 |
| 8 | | D1 | D3 | D2 | |
| 9 | | D2 | D1 | D3 | |
| 10 | D4 | D3 | D2 | | D1 |
| 11 | D5 | D4 | D3 | | D2 |
| 12 | D6 | D5 | D4 | | D3 |
| 13 | | D6 | D5 | D4 | |
| 14 | | D4 | D6 | D5 | |

Etc.

The performance is now 3/10. That is it has improved from 0.285 to 0.3 datum/clock cycle – a 5% improvement. (6)

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

**i.** *how these problems arise;*

The pipelining means that contiguous instructions are overlapped i.e. a number of instructions will be at different stages of execution at a particular point in time. Unfortunately, instructions tend to fetch operands towards the beginning of their execution and save results towards the end. Consequently, two consecutive instructions: the first of which writes to a register and the second of which reads from the same register will encounter a problem because the second instruction will reach the point where the register is read from before the first instruction reaches the point where it writes the data that should have been read. There are four possible dependency problems: read after write (see above); write after read (no problem); read after read (no problem); write after write (could be a problem). (4)

**ii.** *two methods (other than reorder buffers) used to overcome these problems (you only have to give brief descriptions of the method – not too much detail).*

Scoreboarding – a simple reservation scheme whereby an instruction entering the pipeline set a flag to reserve a resource that it will be using – unless it is already set, denoting that an instruction in the pipeline is already using the resource in which case the instruction stalls until the resource is clear.

Register Forwarding – a local control-to-data-flow transformation whereby instructions can be dispatched to and queued at execution units before the data involved in the instruction is ready (in which case the register from which the data will come is recorded. When data is written to a register it is forwarded directly to any execution unit where an instruction using data from the register is waiting. This means that non-dependent instructions could be executed. **(4)**

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

**(4)**

**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,R1    ; R1 ← R5 + R4
        JMP     L2
L1      ADD     R4,R6,R1    ; R1 ← R6 + R4
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 (any references to R1 along this stream are now referenced to this entry for R1). 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)**

**3. a.** *A network of m processors is as shown in* ***Figure 3****. Three busses link the processors together allowing three communications in parallel but the processors do not support simultaneous internal processing and I/O. However, all internal activity in the processors runs in parallel.*

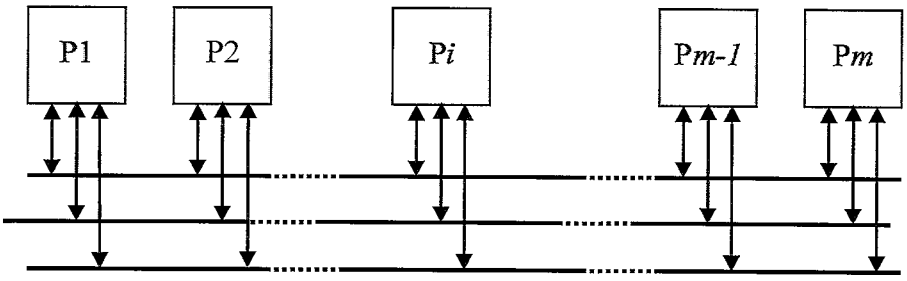



***Figure 3: Network of Processors***

*n tasks are distributed across the m processors. All tasks are the same: the internal execution time of each task is* $t_{exe}$ *and the total communication time between any two tasks is* $t_{int}$ *if the two tasks are on the same processor and* $t_{ext}$ *if the two tasks are on different processors. All tasks communicate equally with all other tasks.*

**i.** *Show that the total time spent on internal computation is:*

$$\frac{n}{mt_{exe}}$$

The tasks are partitioned *n/m* tasks per processor and each processor runs in parallel. Therefore, a simple estimate of the overall processing time is $n/m * t_{exe}$. **(3)**

**ii.** *Show that the total number of task-task communications that need to be carried by the external busses is:*

$$\frac{1}{2}n^2\left(1-\frac{1}{m}\right)$$

The total number of bi-directional communications between *n* tasks is $\frac{1}{2}\cdot n\cdot(n-1)$. However, because the tasks are partitioned, some of these communications are internal.

The number of communications internal to a processor is $\frac{1}{2}\cdot\frac{n}{m}\cdot(\frac{n}{m}-1)$.

Consequently, the total number of external communications is equal to:

$$\frac{1}{2}\cdot n\cdot(n-1)-\frac{1}{2}\cdot n\cdot(\frac{n}{m}-1)=\frac{1}{2}\cdot n^2\cdot\left(1-\frac{1}{m}\right). \quad \textbf{(5)}$$

**iii.** *Hence, how that the speedup due to the parallelisation is:*

$$speedup=\frac{6\cdot m\cdot t_{exe}+m\cdot 3\cdot(n-1)\cdot t_{int}}{6\cdot t_{exe}+n\cdot(m-1)\cdot t_{ext}+3\frac{(n-m)}{m}\cdot t_{\text{int}}}$$

Therefore, when calculating the total communication time we must remember that the internal communications on each processor run in parallel whilst the external communications can proceed only three at a time. Consequently, a simple estimate of the total communication time would be:

$\frac{1}{3}\cdot\left(\frac{1}{2}\cdot n\cdot(n-1)-\frac{1}{2}\cdot n\cdot(\frac{n}{m}-1)\right)\cdot t_{ext}+\frac{1}{2}\cdot\frac{n}{m}\cdot(\frac{n}{m}-1)\cdot t_{\text{int}}$. The extra 1/3 in the first term takes account of the 'by 3' external communications whilst the last term is the internal communications in one processor (all the others are deemed to run in parallel).

The parallel execution time is:

$$t_{par}=\frac{n}{m}\cdot t_{exe}+\frac{1}{3}\cdot\left(\frac{1}{2}\cdot n\cdot(n-1)-\frac{1}{2}\cdot n\cdot(\frac{n}{m}-1)\right)\cdot t_{ext}+\frac{1}{2}\cdot\frac{n}{m}\cdot(\frac{n}{m}-1)\cdot t_{\text{int}}.$$

If $m = 1$ then this reduced to:

$$t_{\text{sin}gle}=n\cdot t_{exe}+\frac{1}{3}\cdot\left(\frac{1}{2}\cdot n\cdot(n-1)-\frac{1}{2}\cdot n\cdot(n-1)\right)\cdot t_{ext}+\frac{1}{2}\cdot n\cdot(n-1)\cdot t_{\text{int}}=n\cdot t_{exe}+\frac{1}{2}\cdot n\cdot(n-1)\cdot t_{\text{int}}.$$

Dividing $t_{single}$ by $t_{par}$ yields *speedup* and this can be manipulated by multiplying top and bottom by $2m/n$ to yield the following.

$$speedup=\frac{6\cdot m\cdot t_{exe}+3\cdot m\cdot(n-1)\cdot t_{int}}{6\cdot t_{exe}+n\cdot(m-1)\cdot t_{ext}+3\cdot\frac{(n-m)}{m}\cdot t_{\text{int}}}$$

**(8)**

**iv.** *From this, show that parallelisation is only worthwhile when:*

$$\frac{t_{exe}}{\frac{1}{3}t_{ext}-t_{\text{int}}}>\frac{n}{2}\ \textit{(approximately)}$$

*for the case when m » 1, and n » m.*

$$\frac{6\cdot m\cdot t_{exe}+3\cdot m\cdot(n-1)\cdot t_{int}}{6\cdot t_{exe}+n\cdot(m-1)\cdot t_{ext}+3\cdot\frac{(n-m)}{m}\cdot t_{\text{int}}}>1$$

$$6\cdot m\cdot t_{exe}+3\cdot m\cdot(n-1)\cdot t_{int}>6\cdot t_{exe}+n\cdot(m-1)\cdot t_{ext}+3\cdot\frac{(n-m)}{m}\cdot t_{\text{int}}$$

$$6\cdot(m-1)\cdot t_{exe}+3\cdot m\cdot(n-1)\cdot t_{int}>n\cdot(m-1)\cdot t_{ext}+3\cdot\frac{(n-m)}{m}\cdot t_{\text{int}}$$

$m$ and $n$ are both large and so approximate $m$-1 to $m$, etc:

$$6\cdot m\cdot t_{exe}+3\cdot m\cdot n\cdot t_{int}>n\cdot m\cdot t_{ext}+3\cdot\frac{n}{m}\cdot t_{\text{int}}$$

Get rid of the last term – it is much smaller:

$$6\cdot m\cdot t_{exe}+3\cdot m\cdot n\cdot t_{int}>n\cdot m\cdot t_{ext}$$

The rest is just manipulation.

$$6\cdot t_{exe}+3\cdot n\cdot t_{int}>n\cdot t_{ext}$$

$$6\cdot t_{exe}>n\cdot(t_{ext}-3\cdot t_{int})$$

$$2\cdot t_{exe}>n\cdot(\tfrac{1}{3}t_{ext}-t_{int})$$

$$\frac{t_{exe}}{\frac{1}{3}t_{ext}-t_{int}}>\frac{n}{2}$$

**(4)**

**4. a.** *There are a number of schemes for classifying processor systems. What are the objectives of processor classification?*

Processor classification is concerned with describing the salient features of a processor (often to provide a comparative basis for how well a processor will execute a particular type of application). Measures attempt to quantify the organisation, number of processors, amount of memory, communication, etc. in a way which allows might allow processors to be ranked, compared, categorised, or for the classification to give a very terse, but meaningful description of the processor. **(3)**

**b.** *Describe Flynn's Classification.*

Flynn's Classification is one of the simplest and most useless classifications; although, this is the one which is quoted most readily. The classification describes the number of instruction streams and data streams which can be processed simultaneously as being either single or multiple. This, obviously gives rise to 4 classifications (of which one is slightly anomalous).

Single Instruction Single Data (SISD): 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.

Multiple Instruction Multiple Data (MIMD): This describes a general purpose multi-processor where multiple processors can each work independently on their own data. The uselessness of the classification is obvious because it does not describe how well the processors can co-operate in completing a set of inter-related tasks.

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). **(3)**

**c.** *You estimate that an algorithm can be split into three sequential parts making up, respectively, 25%, 60%, and 15% of the algorithm: the first part can be parallelised four ways; the second part three ways; the third part two ways.*

*What speed-up could you expect to achieve?*

Amdahl's Rule is $speedup = \dfrac{1}{1-\alpha+\dfrac{\alpha}{N}}$ A generalisation of this would be where:

$(1-\alpha)+\dfrac{\alpha}{N}$ would be replaced by $\sum_{i\in \text{ all steps}} \dfrac{\alpha_i}{N_i}$ (note: $1=\sum_{i\in \text{ all steps}} \alpha_i$)

where $\alpha_i$ represents an elemental proportion of the task and $N_i$ is the degree of parallelism associated with that part of the task. So, for this example, there are 3 steps and $speedup = \dfrac{1}{\dfrac{0.25}{4}+\dfrac{0.6}{3}+\dfrac{0.15}{2}} = \dfrac{24}{8.1} \approx 3$ **(6)**

**(2)**

*Which stage of the algorithm is the major limit to this speed up?*

The second stage – it represents the biggest term in the denominator above and, therefore, acts as a limit on the speed-up.

**d.** *You decide to implement the parallelised algorithm by mapping each separate, parallel part of the algorithm onto separate, identical processing units (rather than implementing the whole algorithm on a single such processing unit). What is the cost benefit ratio associated with this parallelisation?*

Clearly, if we assume that one unit executes the whole algorithm then cost = 1 and benefit (speed-up) = 1. So, cost-benefit ratio is 1. However, if we execute each parallelisable part with a separate unit then the cost is approximately 9 (there will be 4 for the first phase, 3 for the second, 2 for the third). However, speed-up is only 3 and so cost-benefit is 3 i.e. considerably worse than one.

**e.** *You recognise that, because the phases of the algorithm are sequential, the system could be used as a pipeline. What would the improvement in speed-up be in this case?*

In this case by pipelining consecutive tasks, each phase will be working overlapped with the others. In this case, the determining factor is the time spent on each phase. The first is 0.25/4 = 0.075, the second is 0.6/3 = 0.2, and the third is 0.15/2 = 0.075. Therefore, the throughput is determined by the time spent in the second phase. In this case, therefore, speedup would be 5 and cost-benefit would be 1.8. **(3)**

**NLS**