SOLUTION NOTES FOR EXAMINERS AND SUPERVISORS ONLY

Computer Design 2000 Paper 5 Question 2 (SWM)

(a) Data and control hazards (or dependencies) limit throughput in any pipeline.

Example data dependency (on r0):

```
load  r0, (r4)
add   r1,r2,r0
```

In this example, the add instruction has to wait for the load instruction to complete so that the correct value of r0 is available for use. To achieve this, the pipeline will have to stall:

step	IF	RF	EX	MA	WB	comment
1	load	?	?	?	?	
2	add	load	?	?	?	
3	?	add	load	?	?	load address calculated
4	?	add	bubble	load	?	add stalled
5	?	?	add	bubble	load	load result via feedforward

The add instruction is stalled at step 4 to wait for the load result to complete. A bubble (no-operation) is introduced in the pipeline as the add instruction waits and represents wasted CPU time.

Example control hazard:

```
prev_instruction
branch label
next_instruction
```

Even if a branch can be taken early, e.g. at the RF stage, the next_instruction will already have been fetched. Most processors would not execute next_instruction so a bubble is introduced into the pipeline. This means that the pipeline isn't doing useful work.

The the branch was conditional then the decision on whether it was to be taken would often have to wait until prev_instruction had completed the execute stage. This would introduce a further bubble.

(b) Static data-flow processors execute instructions when the data is available rather than in a sequential order. A dyadic static data-flow instruction might look like:

```
opcode | src1 | src2 | dest1 | dest2 | ack1 | ack2
```

```
where:
  opcode = instruction code
  src1   = storage space for operand 1 (including empty/full flag)
  src2   = storage space for operand 2 (including empty/full flag)
  dest1  = destination address 1 (including destination busy/clear flag)
  dest2  = destination address 2 (optional)
  ack1   = acknowledgement address for operand 1
  ack2   = acknowledgement address for operand 2

Execution proceeds as follows:
  1. - operands 1 and 2 are written to in either order setting
       the presence flags
     - wait for destination flag(s) to be "clear"
  2. the instruction is scheduled and the result is computed
  3. - the result is written to destinations 1 and optionally to 2
     - destination flags are set to "busy"
     - acknowledgements are sent to clear the previous instruction's
       destination flags

Data is sent as "tokens":
  <address, data, which_operand>

Acknowledgements used for flow control are sent as meta data tokens:
  <address, which_destination_flag>

Tokens cause data and flags to be updated as part of the instruction.
The act of waiting for two operand tokens to be written is called
"matching".  Matching is the process of resolving data dependencies in
a static data-flow machine.
```