1 a i) What is the main goal of pipelining a CPU?

To increase the rate *at which instructions are* completed.

It is **not** *simply to increase the clock rate, nor to reduce the execution time of each instructions.*

Memorisation. Should take at most 2 minute if they know it.

Marks: 2

ii) Give one example of a MIPS ISA feature which shows that MIPS CPU implementations were expected to be pipelined.

The branch delay slot, to hide branch latency.

The lack of instructions which do both memory operations and ALU operations.

Memorisation. Should take at most 3 minutes, assuming they know the ISA.

Marks: 2

iii) Construct a plausible *quantitative* example (i.e. containing numbers) showing how pipelining can make a CPU worse. Invent numbers for parameters such as clock-rate, IPC, and so on as needed.

Assume the original CPU has clock-rate 100MHz, and achieves an IPC of 0.95, *for a throughput of 95MIPS. If we pipeline it into two stages to get a 150MHz processor, but hazards cause IPC to drop to* 0.6, *then the throughput will drop to 90MIPS, resulting in a drop in achieved rate.*

Synthesis. Probably takes around 3 minutes to understand question and decide on strategy, then 3 minutes to come up with an answer.

Marks: 4

b Consider the following MIPS assembly sequence:

```
f:
  addi $sp, $sp, -4
  div $a0, $a1
  mfhi $v0
  beq $a1, $0, g
  addi $sp, $sp, +4
  jr $ra
  nop
```

i) What is the high-level purpose of the `beq` instruction in this assembly sequence – why would a human or a compiler have inserted it?

It is used to detect divide by zero, and jump to error handling or special-case code.

Note that division by zero exceptions can't occurr in MIPS.

Analysis: around 5 minutes to read code, then 2 minutes to think through interaction.

Marks: 2

ii) Give an example of an exception that could occur while executing the sequence, and the necessary conditions for the exception to occur.

Example 1 : arithmetic overflow. This could occur on first addi when $sp == -2^{31}$

Example 2 : Access violation/bus error. If part of the code has been put in a non-executable location or page, then an exception would occur when the instruction is fetched.

Memorisation: need to think through exceptions they know, then check if it applies. around 4 minutes.

Marks: 2

iii) What is the range of addresses that the linker could use for symbol `g`? Show your working.

*The jump will be relative to (f+12). There is a signed 16-bit immediate, padded out by two zero LSBs. So the reachable range is -32768*4..32767*4, stepping by 4. Combined, the reachable range is -32765*4+f .. +32770*4+f (inclusive), including only multiples of four.*

Analysis: Around 3 minutes to think through problem, and work out it must be relative to f. Around 3 minutes to actually do the maths. Total 6 minutes.

Marks: 3

iv) Optimise this sequence while maintaining functionality, and give the number of cycles needed to execute it in a single-cycle MIPS CPU. State any assumptions you make.

First optimisation:

```
f:
```

```
  div $a0, $a1
  mfhi $v0
  beq $a1, $0, g
  nop  # Need to add a no-op for safety
  jr $ra
  nop
```

Second optimisation:

```
f:
  div $a0, $a1
  beq $a1, $0, g
  mfhi $v0
  jr $ra
  nop
```

Third optimisation:

```
f:
  beq $a1, $0, g
  div $a0, $a1
  jr $ra
  mfhi $v0
```

This relies on the fact that mfhi will be undefined in the case that a1 is zero, so it doesn't matter if it isn't set in that case.

So four cycles, if we assume that the divide instruction produces it's result within two cycles, and that the divide-by-zero branch is not taken.

Analysis: optimisations are fairly straightforward, though need to think through. Around 3 minutes to think of optimisation, 3 minutes to do it, 1 minute to check. Total 6 minutes.

Optimistically, around 36 minutes for this question.

Marks: 5

The two parts carry, respectively, 40%, and 60% of the marks.

2 Consider the following C function:

```
int f(unsigned N, unsigned M, int *x)
{
  int y=0;
  // Begin: code of interest
  for(unsigned i=0; i<N; i++){     // outer loop
    for(unsigned j=0; j<M; j++){  // inner loop
      y+=x[j*N+i];
    }
  }
  // End: code of interest
  return y;
}
```

a While manually converting the function to MIPS-1 assembly, an embedded systems designer first optimised the inner loop of the C code to:

```
for(int j=M-1; j>=0; j--){
```

Why did they view this as an optimisation?

MIPS1 doesn't support branching on a<b for arbitrary a and b, but does allow for branching on a>=0. So the assembly can use one less instruction.

Analysis: 5 minutes for reading the code. Then need to remember ISA restrictions or pipeline effects on branch conditions, then connect to this piece of code (3 minutes). Total 8 minutes.

Marks: 2

b Describe the data locality of the function when using a multi-word set-associative cache in the following scenarios:

i) $N = 1, M >> 1$ ($A >> B$ means "*A much greater than B*")

Reads the data linearly; good spatial locality and block-level temporal locality.

Analysis: working through code/drawing sketch (3 minutes). Answer 1 minute.

Marks: 2

ii) $N >> 1, M = 2$

Reading pairs of values at widely spaced points. No temporal locality, some spatial locality.

Analysis: working through code/drawing sketch (2 minutes). Answer 1 minute.

Marks: 2

iii) $N >> 1, M >> 1$

$N >> 1, M >> 1$: *No spatial or temporal locality*

Analysis: working through code/drawing sketch (1 minutes). Answer 1 minute.

Marks: 2

c i) Assume a static predictor that predicts backwards branches as taken, and forwards branches as not taken. What proportion of branches will be correctly predicted in the function body (denoted "code of interest" in the source)? Include any assumptions and your reasoning.

We'll assume the assembly looks like:

```
outer_top:
  beq ... outer_bottom
  inner_top:
     beq ... inner_bottom
     ...
     b inner_top
  ...
  b outer_top
```

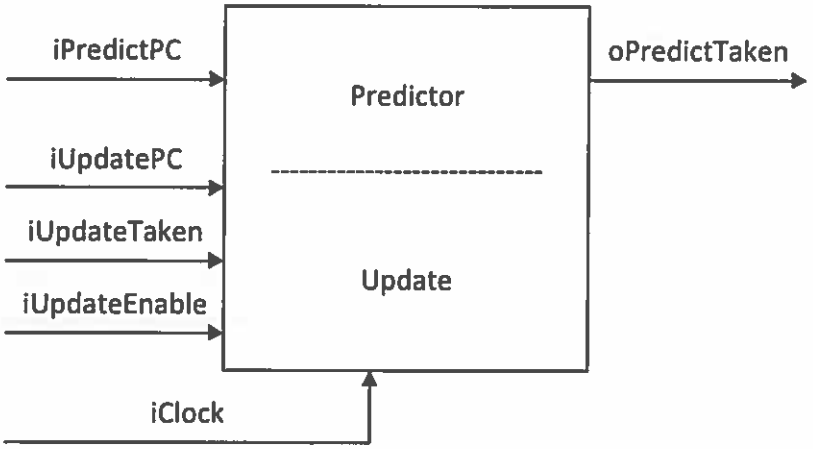
The inner loop will predict correctly on all but the last iteration, so it will predict correctly $2M$ *out of* $2M + 1$ *times. The outer loop will also predict correctly* $2N$ *for the same reason, so the total correct predictions is* $4 * M * N$ *out of* $(2M + 1) * (2N + 1)$ *predictions.*

Analysis: working out which branches are there (3 minutes). Working out how to count the predictions (2 minutes).

Marks: 3

ii) Sketch a top-level interface of a *dynamic* branch prediction circuit, showing the key input and output signals. Briefly describe the signals.

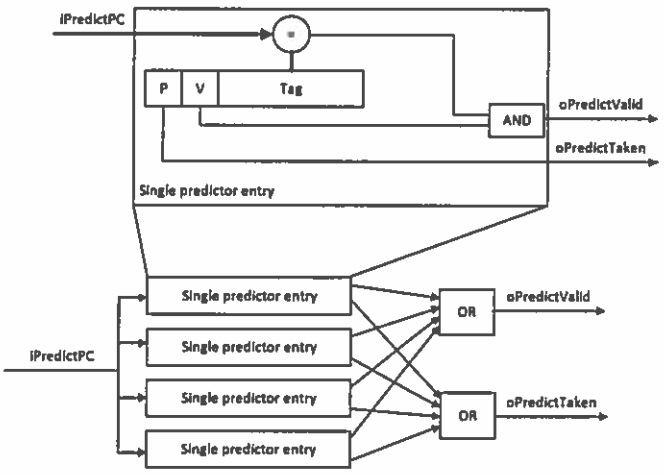
There needs to be a prediction path and an update path. The prediction path takes in a PC, and produces a prediction. The update path takes in a PC and whether that branch was taken, and updates the internal state based on whether the branch was taken. Synthesis: working out what a predictor needs to consume



and produce (4 minutes). Sketching the diagram (4 minutes).

Marks: 4

iii) Design and sketch the implementation of a 4-entry 1-bit dynamic branch predictor in terms of logic and state. You do not need to include logic to update the predictor state, only to make the prediction.



Synthesis: working out the overall state/logic needed (4 minutes). Sketching the diagram (6 minutes).

Total for question: around 40 minutes

Marks: 5

The three parts carry, respectively, 10%, 30%, and 60% of the marks.