**Question 0 [ 5 minutes, 20 points ]**

*Circle the Statements that Are True.*

The ISA is a binary compatibility layer that insulates programs from the underlying machine implementation.

Add instructions in a stack ISA typically have two operands.

It’s reasonable to expect that a pipelined design has a larger CPI than a single-cycle design.

A multicycle design is likely to have a larger CPI than an equivalent pipelined design.

A two-input mux has three inputs.

A two-bit predictor tends to mispredict once on entry and once on exit from a loop.

Branch taken penalties (as measured in cycles) are typically larger than branch direction penalties.

X86 processors tend to have long front ends.

If a program is run using the same input as it was profiled with, profiling-based static prediction has perfect knowledge of the future and cannot be beaten by dynamic prediction methods.

Tags occupy a significant proportion of the space in the implementation of a branch history table.

**Question 1 [ 15 minutes, 20 points]**
Consider a pipeline with the following stages and latencies (in ps):

<table>
<thead>
<tr>
<th>I</th>
<th>I2</th>
<th>EX1</th>
<th>EX2</th>
<th>MEM1</th>
<th>MEM2</th>
<th>MEM3</th>
<th>WB1</th>
<th>Pipeline Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>120</td>
<td>150</td>
<td>190</td>
<td>200</td>
<td>250</td>
<td>100</td>
<td>70</td>
<td>10</td>
</tr>
</tbody>
</table>

Consider the following code is run on the processor with the pipeline described above:

```assembly
addi $r4, $zero, 100
loop:
lw $r0, 0($r1)
addi $r1, $r0, 1
sw $r1, 0($r3)
addi $r4, $r4, -1
bne $r4, $zero, loop
```

a. What is the latency of a single instruction in a fully pipelined implementation? In a single-cycle implementation?

**Fully pipelined:**

\[(250 \times 8) + (7 \times 10) = 2070 \text{ ps}\]

**Single cycle:**

\[100 + 120 + 150 + 190 + 200 + 250 + 100 + 70 = 1180 \text{ ps}\]

b. Assuming no forwarding, add NOP instructions to the code to remove any data hazards. The register file is read in the EX1 stage, and instructions must exit from WB for other instructions to use its value.

```assembly
addi $r4, $zero, 100
loop:
lw $r0, 0($r1)
NOP x 5
addi $r1, $r0, 1
NOP x 5
sw $r1, 0($r3)
addi $r4, $r4, -1
NOP x 5
bne $r4, $zero, loop
```

c. What is the speedup of the fully pipelined implementation over the single cycle implementation? Assume steady-state execution.
single cycle time: (5 instructions)*1180ps = 5900ps
pipelined time: (20 instructions)*260ps = 5200ps
speedup: 5900/5200 = 1.13

d. What are the possible forwarding paths? Assume data is ready to be forwarded at the end of its final stage (EX2, MEM3).

We can forward EX2->EX1, MEM3->EX1


e. Assuming all possible forwarding paths are implemented, what is the new speedup of the fully pipelined implementation over the single cycle implementation? Assume again steady state execution.

New pipeline: (10 instructions)*260ps = 2600ps
speedup: 5900/2600 = 2.27

Question 2 [12 minutes, 20 points]

The UCSD Greendroid project is a prototype mobile processor that is designed to dramatically reduce energy consumption in smartphones. Many existing smartphones today make use of
accelerators that speed up key kernels, such as graphics, video, or sound, by 10x. Typically these accelerators also reduce the energy consumption of the targeted code by 10x. In these systems where we often must choose between energy and performance, a common metric is energy-delay-product, which is the product of the energy consumption and the execution time of the program. Since energy and delay are both "bad" quantities, lower EDP is better.

a. What is the EDP improvement for a kernel that has been targeted by a typical accelerator? Assume that the entire kernel can be handled by the accelerator.

EDP = Energy * Delay: 10*10 = 100x

b. One problem with accelerators is that only certain pieces of code can be turned into accelerators ("acceleratable code"), typically no more than 10% of the execution time of the workload. Taking into account Amdahl's law, what is the effective EDP of the overall system?

(.90+.10/10) * (.90 + .10/10) = .901*.901 = .812

GreenDroid's key innovation is centered around Conservation cores (C-cores), a new kind of automatically generated energy-saving accelerator which can save 19x energy across most kinds of code, not just acceleratable code. However, they only improve performance by 1.25x. Typically, C-cores can be generated for 95% of the code in the system.

c. What is the EDP improvement of a system with only C-cores, and no accelerators?

[.05 + .95 (4/5)]*[.05 + .95 (1/19)]
= (.81) (.05 + .05)
= .081

d. Which system has better EDP? And more importantly, why? Be concise but complete.

GreenDroid - The EDP savings per instruction is less, but it affects much more of the code.
**Question 3 [10 minutes, 10 points]**

Given the following snippets of code, what percentage of branches are mispredicted given the following branch prediction schemes: 1) *always taken* 2) *backwards taken, forward not taken*? Assume that if statements evaluated as “true” are taken.

a.
```c
for(i=0; i<100; i+=2) {
    a = a + 1;
    b = b - a;
}
```

**Assuming the MIPS code is something like this:**

```mips
loop:
    bne i, 100, end
    addi a,a,1
    sub b,b,a
    addi i,i,-1
    j loop
end:
```

1. 50 mispredicts, 1 predict: 50/51 or 98%
2. 50 predicts, 1 mispredict: 1/51 or 2%

If assumed that branches are backwards:

1. 50 predicts, 1 mispredict: 1/51 or 2%
2. 50 predicts, 1 mispredict: 1/51 or 2%

b.
```c
i=100;
do {
```
```c
A++;
if (i % 5 != 0) {
    A = A/2;
}
    i--;
} while (i > 0);

addi i, 100

loop:
    bne i%5, 0, if
    addi i,-1
    bne i,0, loop
    if:
    div A, 2
    return

1. 100 + 80 predicted, 21 mispredicted, 21/201 or 10.4%
1. 100 + 20 predicted, 81 mispredicted, 81/201 or 40.3%

Extra credit: (3 points)

For Q2, at what level of coverage are the EDP benefits of C-cores and accelerators approximately the same?

[x/1.25 + (1-x)]*[x/19 + (1-x)] = [x/10 + (1-x)]*[x/10 + (1-x)]