For this problem, assume that you have discovered a room full of discarded 5-stage pipelined betas.

These betas fall into four categories.

The first is completely functional 5-stage Betas with full bypass and annulment logic.

The second are betas with a bad register file.

In these betas, all data read directly from the register file is zero.

Note that if the data is read from a bypass path, then the correct value will be read.

The third set are betas without bypass paths.

And finally, the fourth are betas without annulment of branch delay slots.

The problem is that the betas are not labeled, so we do not know which falls into which category.

You come up with the test program shown here.

Your plan is to single step through the program using each Beta chip, carefully noting the address that the final JMP loads into the PC.

Your goal is to determine which of the four classes each chip falls into via this JMP address.

Notice that on a fully functional beta, this code would execute the instructions sequentially skipping the MULC instruction.

Here we see a pipeline diagram showing execution of this program on a fully functional beta from category C1.

It shows that the although the MULC instruction is fetched in cycle 2, it gets annulled when the BEQ is in the RF stage and it determines that the branch to label X, or the SUBC instruction, will be taken.

The MULC is annulled by inserting a NOP in its place.

The ADDC and BEQ instructions read R31 from the register file.

The SUBC, however, gets the value of R2 via the bypass path from the BEQ instruction which is in the MEM stage.

The ADD then reads R0 and R2.

R0 has already made it back to the register file because the ADDC instruction completed by the end of cycle 4.

R2, however, is read via the bypass path from the SUBC instruction which is in the ALU stage.

Finally, the JMP, reads the value of R3 via the bypass path from the ADD instruction which is in the ALU stage in cycle 6.

When run on a fully functional beta with bypass paths and annulment of branch delay slots, the code behaves as follows:

The ADDC sets R0 = 4.

The BEQ stores PC + 4 into R2.

Since the ADDC is at address 0, the BEQ is at address 4, so PC + 4 = 8 is stored into R2, and the program branches to label X.

Next, the SUBC subtracts 4 from the latest value of R2 and stores the result which is 4 back into R2.

The ADD adds R0 and R2, or 4 and 4, and stores the result which is 8 into R3.

The JMP jumps to the address in R3 which is 8.

When run on C2 which has a bad register file that always outputs a zero, the behavior of the program changes a bit.

The ADDC and BEQ instructions which use R31 which is 0 anyways behave in the same way as before.

The SUBC, which gets the value of R2 from the bypass path, reads the correct value for R2 which is 8.

Recall that only reads directly from the register file return a zero, whereas if the data is coming from a bypass path, then you get the correct value.

So the SUBC produces a 4.

The ADD reads R0 from the register file and R2 from the bypass path.

The result is that the value of R0 is read as if it was 0 while that of R2 is correct and is 4.

So R3 gets the value 4 assigned to it, and that is the address that the JMP instruction jumps to because it too reads its register from a bypass path.

When run on C3 which does not have any bypass paths, some of the instructions will read stale values of their source operands.

Let's go through the example in detail.

The ADDC and BEQ instructions read R31 which is 0 from the register file so what they ultimately produce does not change.

However, you must keep in mind that the updated value of the destination register will not get updated until after that instruction completes the WB stage of the pipeline.

When the SUBC reads R2, it gets a stale value of R2 because the BEQ instruction has not yet completed, so it assumes that R2 = 0.

It then subtracts 4 from that and tries to write -4 into R2.

Next the ADD runs.

Recall that the ADD would normally read R0 from the register file because by the time the ADD is in the RF stage, the ADDC which writes to R0 has completed all the pipeline stages.

The ADD also normally read R2 from the bypass path.

However, since C3 does not have bypass paths, it reads a stale value of R2 from the register file.

To determine which stale value it reads, we need to examine the pipeline diagram to see if either the BEQ or the SUBC operations, both of which eventually update R2, have completed by the time the ADD reads its source operands.

Looking at our pipeline diagram, we see that when the ADD is in the RF stage, neither the BEQ nor the SUBC have completed, thus the value read for R2 is the initial value of R2 which is 0.

We can now determine the behavior of the ADD instruction which is that it assumes R0 = 4 and R2 = 0 and will eventually write a 4 into R3.

Finally, the JMP instruction wants to read the result of the ADD, however, since there are no bypass paths, it reads the original value of R3 which was 0 and jumps to address 0.

Of course, had we looked at our code a little more closely, we could have determined this without all the intermediate steps because the ADD is the only instruction that tries to update R3, and you know that the JMP

would normally get R3 from the bypass path which is not available, therefore it must read the original value of R3 which is 0.

In category C4, the betas do not annul instructions that were fetched after a branch instruction but aren't supposed to get executed.

This means that the MULC which is fetched after the BEQ is actually executed in the pipeline and affects the value of R2.

Let's take a close look at what happens in each instruction.

Once again we begin with the ADDC setting R0 to 4 and the BEQ setting R2 to 8.

Since our bypass paths are now working, we can assume that we can immediately get the updated value.

Next, we execute the MULC which takes the latest value of R2 from the bypass path and multiplies it by 2.

So it sets R2 = 16.

The SUBC now uses this value for R2 from which it subtracts 4 to produce R2 = 12.

The ADD then reads R0 = 4 and adds to it R2 = 12 to produce R3 = 16.

Finally, the JMP jumps to address 16.

Since each of the four categories produces a unique jump address, this program can be used to sort out all the betas into the four categories.