**TANIA MAINA.**

**SCT212-0179/2021.**

**BCT2408: COMPUTER ARCHTECTURE.**

E1.

(a)

LD R1, 0(R2)  
DADD R3, R1, R2  
Hazard type: RAW (Read After Write)

Registers Involved: R1. The LD instruction loads a value from memory into R1. The next instruction (DADD) tries to use R1 as a source before the load has completed. If there's no forwarding, this results in a stall.

(b)

MULT R1, R2, R3  
DADD R1, R2, R3

Hazard type: WAW (Write After Write)

Registers Involved: R1. Both instructions write to R1. If MULT takes multiple cycles and DADD completes earlier, it could overwrite the value MULT was supposed to write.

(c)

MULT R1, R2, R3  
MULT R4, R5, R6

Hazard type: Structural hazard

Both instructions are using the MULT functional unit. If the processor has only one multiply unit and it takes several cycles, the second MULT has to wait. This is a structural hazard,

(d)

DADD R1, R2, R3  
SD 2000(R0), R1

Hazard type: RAW (Read After Write)

Registers involved: R1. DADD writes to R1, and then SD uses R1 as a source to store its value to memory. If SD executes before DADD writes its result, the wrong value could be stored.

(e)

DADD R1, R2, R3  
SD 2000(R1), R4

Hazard type: RAW (Read After Write)

Registers involved: R1. DADD computes a value into R1, which is then used as an address in the SD instruction. If SD uses the old value of R1 before DADD writes the new value, it will store to the wrong memory location.

E2

a)

A 2-bit saturating counter branch predictor uses 2 bits to keep track of how likely a branch is to be taken. This creates 4 states, typically defined as:

|  |  |  |
| --- | --- | --- |
| **State** | **Meaning** | **Predicts** |
| 00 | Strongly Not Taken | Not Taken |
| 01 | Weakly Not Taken | Not Taken |
| 10 | Weakly Taken | Taken |
| 11 | Strongly Taken | Taken |

**State transitions:**

|  |  |  |
| --- | --- | --- |
| **Current State** | **Branch Taken?** | **New State** |
| 00 (SN) | Yes | 01 (WN) |
| 00 (SN) | No | 00 (SN) |
| 01 (WN) | Yes | 10 (WT) |
| 01 (WN) | No | 00 (SN) |
| 10 (WT) | Yes | 11 (ST) |
| 10 (WT) | No | 01 (WN) |
| 11 (ST) | Yes | 11 (ST) |
| 11 (ST) | No | 10 (WT) |

**Predictor behavior:**

If current state is 00 or 01, it predicts Not Taken.

If current state is 10 or 11, it predicts Taken.

b)

The branch in question is:

BNEZ F1, else

This checks if F1! = 0, i.e., if x[i]!= 0.

Every other element of x has value 0, starting with the first one.

So:

x [0] = 0 ⟹ branch is NOT taken

x [1] ≠ 0 ⟹ branch is taken

x [2] = 0 ⟹ branch is NOT taken

x [3] ≠ 0 ⟹ branch is taken

... and so on.

Let’s assume N = 8. The iterations and outcomes will be as follows, with the predictor starting at 00 (Strongly Not Taken).

Step-by-Step Simulation

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **iteration** | **x[i] Value** | **Actual Outcome (Taken?)** | **Prediction** | **Prediction Correct?** | **Counter Before** | **Counter After** |
| 0 | 0 | No | Not Taken | ✅ Yes | 00 | 00 |
| 1 | ≠0 | Yes | Not Taken | ❌ No | 00 | 01 |
| 2 | 0 | No | Not Taken | ✅ Yes | 01 | 00 |
| 3 | ≠0 | Yes | Not Taken | ❌ No | 00 | 01 |
| 4 | 0 | No | Not Taken | ✅ Yes | 01 | 00 |
| 5 | ≠0 | Yes | Not Taken | ❌ No | 00 | 01 |
| 6 | 0 | No | Not Taken | ✅ Yes | 01 | 00 |
| 7 | ≠0 | Yes | Not Taken | ❌ No | 00 | 01 |

Conclusively: The predictor never gets enough consistent taken outcomes to move into the "Taken" states. It always predicts Not Taken, because it resets to 00 every time a Not Taken branch is encountered (which is every other time).

Out of 8 predictions, only 4 were correct (those for x[i] = 0). The other 4 predictions (for x[i] ≠ 0) were incorrect. This illustrates a limitation of the 2-bit counter in alternating branch behavior patterns it keeps bouncing between 00 and 01, never gaining confidence to predict Taken.