Desmond Oketch- SCT212-0083/2021

COMPUTER TECHNOLOGY

Lab 3

Computer Architecture – tutorial 3

E1: Identifying Hazards in MIPS Code Fragments

For each code fragment, identify the type of hazard and the registers involved.

a.

```
LD R1, 0(R2)
DADD R3, R1, R2
```

- Hazard Type: Read After Write (RAW)
- Registers Involved: R1
- Explanation: The DADD instruction reads R1 before the LD instruction writes to it, leading to a RAW hazard.

b.

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

- Hazard Type: Write After Write (WAW)
- Registers Involved: R1
- Explanation: Both instructions write to R1. If DADD completes before MULT, it can overwrite the result of MULT, causing a WAW hazard.

c.

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

- Hazard Type: None
- Explanation: The instructions operate on different registers with no dependencies, so no hazard exists.

d.

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

- Hazard Type: Read After Write (RAW)
- Registers Involved: R1
- Explanation: The SD instruction reads R1 to store its value, but R1 is being written by the preceding DADD instruction, leading to a RAW hazard.

e.

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

- Hazard Type: Read After Write (RAW)
- Registers Involved: R1
- Explanation: Here, R1 is used to calculate the memory address in the SD instruction. Since R1 is being written by DADD, there's a RAW hazard concerning the address computation.

E2: 2-Bit Saturating Counter Branch Predictor

a. Behavior of a 2-Bit Saturating Counter

A 2-bit saturating counter is a finite state machine used in dynamic branch prediction. It has four states:

1. Strongly Not Taken (00)
2. Weakly Not Taken (01)
3. Weakly Taken (10)
4. Strongly Taken (11)

State Transitions:

- If the branch is taken:
  - 00 → 01 → 10 → 11
- If the branch is not taken:
  - 11 → 10 → 01 → 00

The predictor changes its prediction only after two consecutive mispredictions, providing stability against occasional anomalies.

Analyzing the Branch Predictor with Given Code

Code Overview:

```
for (i=0; i<N; i++)
   if (x[i] == 0)
      y[i] = 0.0;
   else
      y[i] = y[i]/x[i];
```

Assembly Snippet:

```
loop: L.D F1, 0(R2)
      L.D F2, 0(R3)
      BNEZ F1, else
      ADD.D F2, F0, F0
      BEZ R0, fall
else: DIV.D F2, F2, F1
fall: DADDI R2, R2, 8
      DADDI R3, R3, 8
      DSUBI R1, R1, 1
      S.D -8(R3), F2
      BNEZ R1, loop
```

Assumptions:

- Every other element of x is zero, starting with the first one.
- Initial state of the predictor: Strongly Not Taken (00).

Branch Instruction of Interest:

BNEZ F1, else

Prediction Analysis:

We'll analyze the first few iterations to observe the predictor's behavior.

Iteration	x[i] == 0	Actual Outcome	Predictor State Before	Prediction	Misprediction?	Predictor State After
1	Yes	Not Taken	00	Not Taken	No	00
2	No	Taken	00	Not Taken	Yes	01
3	Yes	Not Taken	01	Not Taken	No	00
4	No	Taken	00	Not Taken	Yes	01
5	Yes	Not Taken	01	Not Taken	No	00
6	No	Taken	00	Not Taken	Yes	01

Observations:

- The predictor mispredicts every time the branch is **taken** because it requires two consecutive taken outcomes to switch its prediction to taken.
- Since the pattern alternates between not taken and taken, the predictor remains in the lower states (00 or 01), leading to frequent mispredictions on taken branches.