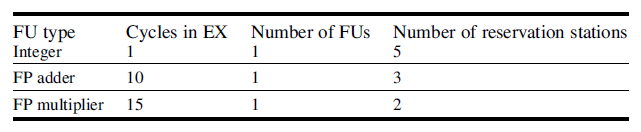
**3.15 [20/20] <3.4, 3.5, 3.7, 3.8> In this exercise, we will look at how variations on Tomasulo’s algorithm perform when running the loop from Exercise 3.14. The functional units (FUs) are described in the following table.**

****

**Assume the following:**

**■ Functional units are not pipelined.**

**■ There is no forwarding between functional units; results are communicated by the common data bus (CDB).**

**■ The execution stage (EX) does both the effective address calculation and the memory access for loads and stores. Thus, the pipeline is IF/ID/IS/EX/WB.**

**■ Loads require one clock cycle.**

**■ The issue (IS) and write-back (WB) result stages each require one clock cycle.**

**■ There are five load buffer slots and five store buffer slots.**

**■ Assume that the Branch on Not Equal to Zero (BNEZ) instruction requires one clock cycle.**

**a. [20] <3.4–3.5> For this problem use the single-issue Tomasulo MIPS pipeline of Figure 3.10 with the pipeline latencies from the preceding table. Show the number of stall cycles for each instruction and what clock cycle each instruction begins execution (i.e., enters its first EX cycle) for three iterations of the loop.**

**How many cycles does each loop iteration take? Report your answer in the form of a table with the following column headers:**

**■ Iteration (loop iteration number)**

**■ Instruction**

**■ Issues (cycle when instruction issues)**

**■ Executes (cycle when instruction executes)**

**■ Memory access (cycle when memory is accessed)**

**■ Write CDB (cycle when result is written to the CDB)**

**■ Comment (description of any event on which the instruction is waiting)**

**Show three iterations of the loop in your table. You may ignore the first instruction.**

***Ans:***

**Assumption:** Only one issue can be done per clock cycle.

* At cycle 1, the base address of array X1 is loaded into F2. This instruction writes into CDV at cycle 3.
* The next instruction to multiply the registers F0 and F2 and store the result in F4, is issued at cycle 2. But, since F2 is still in use, this instruction can’t be executed till clock cycle 4. Meanwhile, during clock cycles 3 and 4, F0 and F4 are waiting in the Multiplication Reservation station. Once, F2 is available, multiplication starts execution and completes at cycle 18. Then it is written to the CDB at cycle 19.
* The next instruction to load the base address of array X2 to F6 is issued at cycle 3. Since, it has no dependency and the load buffer is available, execution starts at cycle 4 and is written to CDB at cycle 5.
* The next instruction to add F4 and F6 and store the result in F6, is issued at cycle 4. Since F4 is in use by instruction 2 till cycle 19, execution starts at cycle 20 and finishes at 29. It is then written to the CDB at cycle 30.
* The fifth instruction to store the base address of X2 to F6 is issued at cycle 5. But, it has to wait till F6 completes execution in the previous instruction. The instruction is then executed at instruction 31. Since, this is storing, it doesn’t write to CDB.
* The next 2 instructions are for addition of unsigned integers that have no dependencies. So, they execute in immediate cycles.
* The last instruction in 1st iteration is issued at cycle 9. But, it has to wait till cycle 10 for R3 to finish execution of previous instruction. It then executes at cycle 11.

The next iterations follow the similar process.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Loop Iteration Number** | **Instruction** | **Issues At(Cycle)** | **Executes At(Cycle)** | **Memory Access (At Cycle)** | **Write CDB (At Cycle)** | **Comment** |
| 1 | LD F2, 0(X1) | 1 | 2 | 2 | 3 | Load the base address of X1 to F2 in cycle 2 |
| 1 | MULD F4, F2, F0 | 2 | 4 | 4 | 19 | Wait for F2 to complete; Multiplication RS uses (3-4} cycles; Multiplication done in cycles (5 - 18) |
| 1 | LD F6, 0(X2) | 3 | 4 | 4 | 5 | Load the base address of X2 to F6 in cycle 4 |
| 1 | ADDD F6, F4, F6 | 4 | 20 | 20 | 30 | Wait for F6 to complete load; Addition RS uses cycles (5-20); Addition uses cycles (21-29) |
| 1 | SD F6, 0(X2) | 5 | 31 | 31 | - | Wait for F6 to finish; Store buffer waits for cycles (6-31) |
| 1 | DADDIU R1, R1, #8 | 6 | 7 | 7 | 8 | No wait |
| 1 | DADDIU R2, R2, #8 | 7 | 8 | 8 | 9 | No wait |
| 1 | DSLTU R3, R1, R4 | 8 | 9 | 9 | 10 | No wait |
| 1 | BNEZ R3, foo | 9 | 11 | 11 | - | Wait for R3 to finish |
| 2 | LD F2, 0(X1) | 10 | 12 | 12 | 13 | Wait for BNEZ to finish; Load buffer waits for the cycles (11-12) |
| 2 | MULD F4, F2, F0 | 11 | 19 | 19 | 34 | Wait for F2 to complete; Multiplication is busy; Multiplication RS waits for cycles (12-19); Multiplication done in cycles (20-33) |
| 2 | LD F6, 0(X2) | 12 | 13 | 13 | 14 | Load buffer executes at cycle 13 |
| 2 | ADDD F6, F4, F6 | 13 | 35 | 35 | 45 | Wait for F4 to complete; Addition RS uses (14-35); Addition done in cycles (36-44) |
| 2 | SD F6, 0(X2) | 14 | 46 | 46 | - | Wait for F6 to finish; Store buffer waits for cycles (15-46) |
| 2 | DADDIU R1, R1, #8 | 15 | 16 | 16 | 17 | No wait |
| 2 | DADDIU R2, R2, #8 | 16 | 17 | 17 | 18 | No wait |
| 2 | DSLTU R3, R1, R4 | 17 | 18 | 18 | 20 | No wait |
| 2 | BNEZ R3, foo | 18 | 20 | 20 | - | Wait for R3 to finish |
| 3 | LD F2, 0(X1) | 19 | 21 | 21 | 22 | Wait for BNEZ to finish; Load buffer waits for the cycles (20-21) |
| 3 | MULD F4, F2, F0 | 20 | 34 | 34 | 49 | Wait for F2 to complete; Multiplication is busy; Multiplication RS waits for cycles (21-34); Multiplication done in cycles (30-48) |
| 3 | LD F6, 0(X2) | 21 | 22 | 22 | 23 | Load buffer executes at cycle 22 |
| 3 | ADDD F6, F4, F6 | 22 | 50 | 50 | 60 | Wait for F4 to complete; Addition RS uses (23-49); Addition done in cycles (51-59) |
| 3 | SD F6, 0(X2) | 23 | 55 | 55 | - | Wait for F6 to finish; Store buffer waits for cycles (24-55) |
| 3 | DADDIU R1, R1, #8 | 24 | 25 | 25 | 26 | No wait |
| 3 | DADDIU R2, R2, #8 | 25 | 26 | 26 | 27 | No wait |
| 3 | DSLTU R3, R1, R4 | 26 | 27 | 27 | 28 | No wait |
| 3 | BNEZ R3, foo | 27 | 29 | 29 | - | Wait for R3 to finish |

**b. [20] <3.7, 3.8> Repeat part (a) but this time assume a two-issue Tomasulo algorithm and a fully pipelined floating-point unit (FPU).**

***Ans:***

**Assumption:** 2 issues can be done at one cycle.

Execution is similar to the process explained in the problem (a).

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Loop Iteration Number** | **Instruction** | **Issues At(Cycle)** | **Executes At(Cycle)** | **Memory Access (At Cycle)** | **Write CDB (At Cycle)** | **Comment** |
| 1 | LD F2, 0(X1) | 1 | 2 | 2 | 3 | Load the base address of X1 to F2 at cycle 2 |
| 1 | MULD F4, F2, F0 | 1 | 4 | 4 | 19 | Wait for F2 to finish; Multiplication RS uses cycles (2-4); Multiplication uses cycle 5 |
| 1 | LD F6, 0(X2) | 2 | 3 | 3 | 4 | Load the base address of X2 to F6 at cycle 2 |
| 1 | ADDD F6, F4, F6 | 2 | 20 | 20 | 30 | Wait for F4 to complete; Addition RS waits for cycles (3-20); Addition uses cycle 21 |
| 1 | SD F6, 0(X2) | 3 | 31 | 31 | - | Wait for F6 to complete; Store buffer waits for cycles (4-31) |
| 1 | DADDIU R1, R1, #8 | 3 | 4 | 4 | 5 | No wait |
| 1 | DADDIU R2, R2, #8 | 4 | 5 | 5 | 6 | No wait |
| 1 | DSLTU R3, R1, R4 | 4 | 6 | 6 | 7 | Integer FU busy; Integer RS waits for cycles (5-6) |
| 1 | BNEZ R3, foo | 5 | 7 | 7 | - | Integer FU busy; Integer RS waits for cycles (6-7) |
| 2 | LD F2, 0(X1) | 6 | 8 | 8 | 9 | Wait for BNEZ to complete |
| 2 | MULD F4, F2, F0 | 6 | 10 | 10 | 25 | Wait for F2 to finish; Multiplication RS uses cycles (7-10); Multiplication uses cycle 11 |
| 2 | LD F6, 0(X2) | 7 | 9 | 9 | 10 | Integer FU is busy; Integer RS waits for cycles (8-9) |
| 2 | ADDD F6, F4, F6 | 7 | 26 | 26 | 36 | Wait for F4 to complete; Addition RS waits for cycles (8-26); Addition uses cycle 27 |
| 2 | SD F6, 0(X2) | 8 | 37 | 37 | - | Wait for F6 to complete |
| 2 | DADDIU R1, R1, #8 | 8 | 10 | 10 | 11 | Integer FU is busy; Integer RS waits for cycles (8-10) |
| 2 | DADDIU R2, R2, #8 | 9 | 11 | 11 | 12 | Integer FU is busy; Integer RS waits for cycles (10-11) |
| 2 | DSLTU R3, R1, R4 | 9 | 12 | 12 | 13 | Integer FU is busy; Integer RS waits for cycles (10-12) |
| 2 | BNEZ R3, foo | 10 | 14 | 14 | - | Wait for R3 to finish |
| 3 | LD F2, 0(X1) | 11 | 15 | 15 | 16 | Wait for BNEZ to complete |
| 3 | MULD F4, F2, F0 | 11 | 17 | 17 | 32 | Wait for F2 to complete; Multiplication RS uses cycles (12-17); Multiplication uses cycle 17 |
| 3 | LD F6, 0(X2) | 12 | 16 | 16 | 17 | Integer FU is busy; Integer RS waits for cycles (13-16) |
| 3 | ADDD F6, F4, F6 | 12 | 33 | 33 | 43 | Wait for F4 to finish; Addition RS waits for cycles (13-33); Addition uses cycle 33 |
| 3 | SD F6, 0(X2) | 14 | 44 | 44 | - | Wait for F6 to complete; Integer RS is full |
| 3 | DADDIU R1, R1, #8 | 15 | 17 | 17 | - | Integer RS is full; Integer RS uses cycle 17 |
| 3 | DADDIU R2, R2, #8 | 16 | 18 | 18 | - | Integer RS is full; Integer RS uses cycle 18 |
| 3 | DSLTU R3, R1, R4 | 20 | 21 | 21 | - | Integer RS is full |
| 3 | BNEZ R3, foo | 21 | 22 | 22 | - | Integer RS is full |

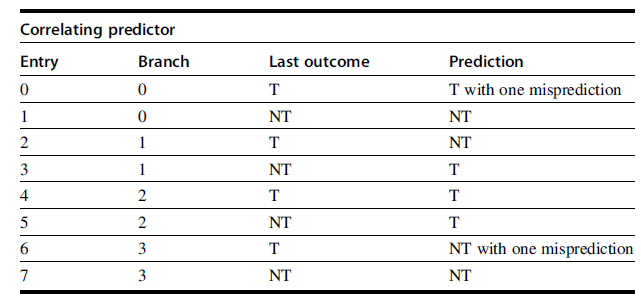
**3.16 [10] <3.4>Tomasulo’s algorithm has a disadvantage: only one result can compute per clock per CDB. Use the hardware configuration and latencies from the previous question and find a code sequence of no more than 10 instructions where Tomasulo’s algorithm must stall due to CDB contention. Indicate where this occurs in your sequence.**

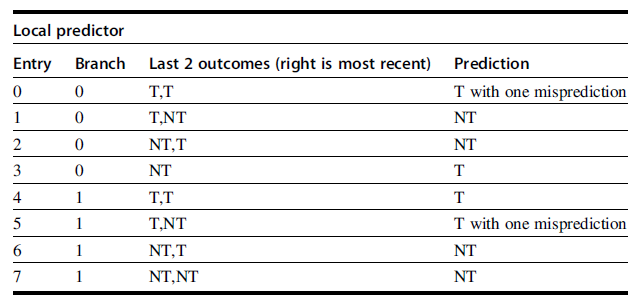
***Ans:*** Consider the below sequence of instructions. CDB conflict occurs at cycle 12 when the Floating-point addition and unsigned integer addition want to access the CDB at the same time.

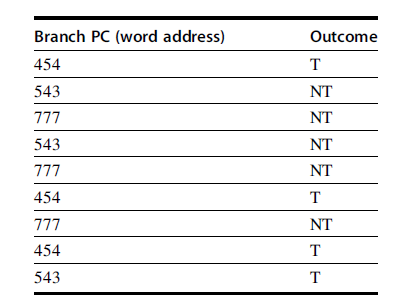
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Instruction | Issues At(Cycle) | Executes At(Cycle) | Memory Access (At Cycle) | Write CDB (At Cycle) |
| ADDD F2, F4, F6 | 1 | 2 | 2 | **12** |
| ADDIU R1, R1, #8 | 2 | 3 | 3 | 4 |
| ADDIU R1, R2, R1 | 3 | 5 | 5 | 6 |
| ADDIU R2, R2, #8 | 4 | 7 | 7 | 8 |
| ADDIU R2, R1, R2 | 5 | 9 | 9 | 10 |
| ADDIU R1, R1, #8 | 6 | 11 | 11 | **12** |

**3.17 [20] <3.3> An (m,n) correlating branch predictor uses the behavior of the most recent m executed branches to choose from 2m predictors, each of which is an n bit predictor. A two-level local predictor works in a similar fashion, but only keeps track of the past behavior of each individual branch to predict future behavior. There is a design trade-off involved with such predictors: correlating predictors require little memory for history, which allows them to maintain 2-bit predictors for a large number of individual branches (reducing the probability of branch instructions reusing the same predictor), while local predictors require substantially more memory to keep history and are thus limited to tracking a relatively small number of branch instructions. For this exercise, consider a (1,2) correlating predictor that can track four branches (requiring 16 bits) versus a (1,2) local predictor that can track two branches using the same amount of memory. For the following branch outcomes, provide each prediction, the table entry used to make the**

**prediction, any updates to the table as a result of the prediction, and the final misprediction rate of each predictor. Assume that all branches up to this point have been taken. Initialize each predictor to the following:**







***Ans:*** Below are the outcomes for correlating predictor:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Branch PC** | **Branch PC mod 4** | **Entry** | **Prediction** | **Outcome** | **Misprediction?** | **Update** |
| 454 | 2 | 4 | T | T | No | None |
| 543 | 3 | 6 | NT | NT | No | None |
| 777 | 1 | 2 | NT | NT | No | None |
| 543 | 3 | 7 | NT | NT | No | None |
| 777 | 1 | 3 | T | NT | Yes | Change to 'NT' (First misprediction) |
| 454 | 2 | 4 | T | T | No | None |
| 777 | 1 | 3 | T | NT | Yes | Change to 'NT' (Second Misprediction) |
| 454 | 2 | 4 | T | T | No | None |
| 543 | 3 | 7 | NT | T | Yes | Change to 'NT' (Third misprediction) |

Misprediction rate = Number of mispredictions / Total number of entries

= 3 / 9

= 0.33

Below are the outcomes for local predictor:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Branch PC** | **Branch PC mod 2** | **Entry** | **Prediction** | **Outcome** | **Misprediction?** | **Update** |
| 454 | 0 | 0 | T | T | No | Change to 'T' |
| 543 | 1 | 4 | T | NT | Yes | Change to 'T' (First misprediction) |
| 777 | 1 | 1 | NT | NT | No | None |
| 543 | 1 | 3 | T | NT | Yes | Change to 'T' (Second misprediction) |
| 777 | 1 | 3 | T | NT | Yes | Change to 'NT' (Third misprediction) |
| 454 | 0 | 0 | T | T | No | None |
| 777 | 1 | 3 | NT | NT | No | None |
| 454 | 0 | 0 | T | T | No | None |
| 543 | 1 | 5 | T | T | No | None |

Misprediction rate = Number of mispredictions / Total number of entries

= 3 / 9

= 0.33

**3.18 [10] <3.9> Suppose we have a deeply pipelined processor, for which we implement a branch-target buffer for the conditional branches only. Assume that the misprediction penalty is always four cycles and the buffer miss penalty is always three cycles. Assume a 90% hit rate, 90% accuracy, and 15% branch frequency. How much faster is the processor with the branch-target buffer versus a processor that has a fixed two-cycle branch penalty? Assume a base clock cycle per instruction (CPI) without branch stalls of one.**

***Ans:***

Base CPI without branch stalls = 1

Hit rate = 90%

Accuracy = 90%

Branch Frequency = 15%

**Processor with branch target buffer:**

Penalty for correct prediction = 0 cycles

Misprediction penalty = 4 cycles

Buffer miss penalty = 3 cycles

Probability that the branch is not in buffer = Branch Frequency \* Miss rate

= 15 % \* 10 % (Since hit rate is 90%, miss rate is 10%)

= 1.5 %

Probability that the branch is in buffer and taken = Branch Frequency \* Hit rate \* Accuracy

= 15 % \* 90 % \* 90 %

= 12.1 %

Probability that the branch is in buffer and not taken = Branch Frequency \* Hit rate \* Inaccuracy

= 15 % \* 90 % \* 10 %

(Inaccuracy is 10 % since accuracy is 90%)

= 1.3 %

BTB Stalls = (Probability that branch is not in buffer \* Buffer miss penalty) + (Probability that branch is in buffer and taken \* Penalty for correct prediction) + (Probability that the branch is in buffer and not taken \* Misprediction penalty)

= (1.5 % \* 3) + (12.1 % \* 0) + (1.3 % \* 4)

= 4.5 % + 0 % + 5.2 %

= 9.7 %

= 0.097

CPI with BTB = Base CPI + BTB stalls

= 1 + 0.097

= 1.097

Processor that has a fixed 2 cycle branch penalty:

Stalls with no BTB = Branch Frequency \* Penalty

= 15 % \* 2

= 30 %

= 0.3

CPI with no BTB = Base CPI + Stalls with no BTB

= 1 + 0.3

= 1.3

Speed = CPI with no BTB / CPI with BTB

= 1.3 / 1.097

= 1.185

Hence, a processor with BTB is 1.2 times faster than the processor with a fixed 2-cycle penalty.