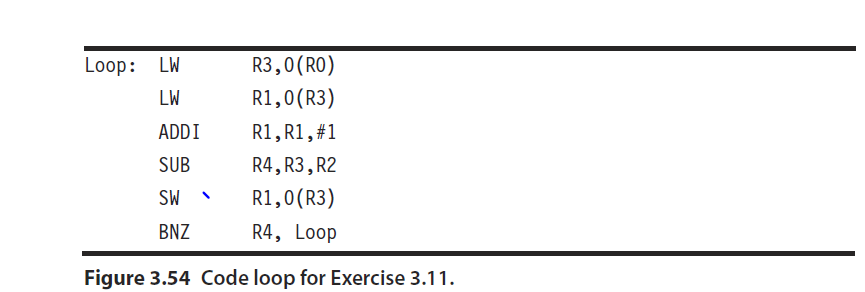
**SUID: 22375-2155 Name: Ravichandra Malapati**

3.11 [10/10/10] <3.3> Assume a five-stage single-pipeline microarchitecture (fetch, decode, execute, memory, write-back) and the code in Figure 3.54. All ops are one cycle except LW and SW, which are 1 + 2 cycles, and branches, which are 1 + 1 cycles. There is no forwarding. Show the phases of each instruction per clock cycle for one iteration of the loop.

a. [10] <3.3> How many clock cycles per loop iteration are lost to branch overhead?

b. [10] <3.3> Assume a static branch predictor, capable of recognizing a backwards branch in the Decode stage. Now how many clock cycles are wasted on branch overhead?

c. [10] <3.3> Assume a dynamic branch predictor. How many cycles are lost on a correct prediction?

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **3.11 a** |  |  |  |  |  |  |
|  |  | **IF** | **ID** | **EX** | **MEM** | **WB** |
| **LW** | **R3,0(R0)** | 1 | 2 | 3 | 6 | 7 |
| **LW** | **R1, 0(R3)** | 2 | 3 | 7 | 10 | 11 |
| **ADDI** | **R1, R1, #1** | 3 | 7 | 11 | 12 | 13 |
| **SUB** | **R4, R3, R2** | 7 | 11 | 12 | 13 | 14 |
| **SW** | **R1, 0(R3)** | 11 | 12 | 13 | 16 | 17 |
| **BNZ** | **R4, Loop** | 12 | 13 | 14 | 16 | 18 |
| **Loop LW** | **R3, 0(R0)** | 17 |  |  |  |  |

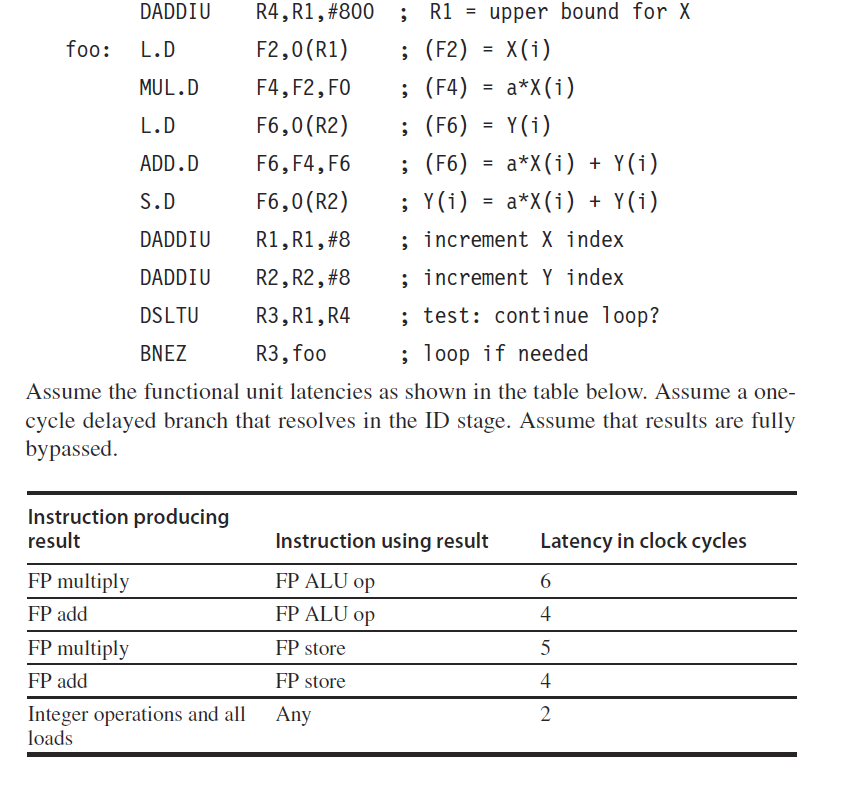
1. From above diagram it’s clear that the loop over head is 17-13 = 4 cycles
2. From the above diagram it’s clear that 2(15-13) cycles are lost after implementing a static branch predictor. The branch is predicted right after the Execution is done instead of waiting for Memory write

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **3.11. b** |  |  |  |  |  |  |
|  |  | **IF** | **ID** | **EX** | **MEM** | **WB** |
| **LW** | **R3,0(R0)** | 1 | 2 | 3 | 6 | 7 |
| **LW** | **R1, 0(R3)** | 2 | 3 | 7 | 10 | 11 |
| **ADDI** | **R1, R1, #1** | 3 | 7 | 11 | 12 | 13 |
| **SUB** | **R4, R3, R2** | 7 | 11 | 12 | 13 | 14 |
| **SW** | **R1, 0(R3)** | 11 | 12 | 13 | 16 | 17 |
| **BNZ** | **R4, Loop** | 12 | 13 | 14 | 16 | 1418 |
| **Loop LW** | **R3, 0(R0)** | 15 |  |  |  |  |

C. When dynamic branch predicter is implemented the LW operation in the next loop will execute at 13th cycle itself hence no cycles are lost due to brach overload

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **3.11. c** |  |  |  |  |  |  |
|  |  | **IF** | **ID** | **EX** | **MEM** | **WB** |
| **LW** | **R3,0(R0)** | 1 | 2 | 3 | 6 | 7 |
| **LW** | **R1, 0(R3)** | 2 | 3 | 7 | 10 | 11 |
| **ADDI** | **R1, R1, #1** | 3 | 7 | 11 | 12 | 13 |
| **SUB** | **R4, R3, R2** | 7 | 11 | 12 | 13 | 14 |
| **SW** | **R1, 0(R3)** | 11 | 12 | 13 | 16 | 17 |
| **BNZ** | **R4, Loop** | 12 | 13 | 14 | 16 | 18 |
| **Loop LW** | **R3, 0(R0)** | 13 | Implementing dynamic predictor | | |  |

3.14 [25/25/25] <3.2, 3.7> In this exercise, we look at how software techniques can extract instruction-level parallelism (ILP) in a common vector loop. The following loop is the so-called DAXPY loop (double-precision *aX* plus *Y*) and is the central operation in Gaussian elimination. The following code implements the DAXPY operation, *Y* = *aX* + *Y*, for a vector length 100. Initially, R1 is set to the base address of array *X* and R2 is set to the base address of *Y*:



a. [25] <3.2> Assume a single-issue pipeline. Show how the loop would look both unscheduled by the compiler and after compiler scheduling for both floating-point operation and branch delays, including any stalls or idle clock cycles. What is the execution time (in cycles) per element of the result vector,*Y*, unscheduled and scheduled? How much faster must the clock be for processor hardware alone to match the performance improvement achieved by the scheduling compiler? (Neglect any possible effects of increased clock speed on memory system performance.)

Answer for 3.14a : assuming the single issue pipe line and unscheduled by compiler.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Unscheduled** |  |  |  |  |
| **Instruction** | **Operands** | **Comments** | **Clock cycle** |  |
| DAADIU | R4, R1, #800 |  | 1 |  |
| L.D | F2, 0(R1) | F2= X(i) | 2 |  |
| stall |  |  | 3 |  |
| MUL.D | F4, F2, F0 | F4=a.X(i) | 4 | Multiplication will take 6 cycles from requirements and hence stalls till 10th cycle |
| L.D | F6, 0(R2) | F6=Y(i) | 5 |  |
| stall |  |  | 6 |  |
| Stall |  |  | 7 |  |
| Stall |  |  | 8 |  |
| stall |  |  | 9 |  |
| ADD.D | F6, F4, F6 | F6=a.X(i)+Y(i) | 10 | The addition is dependent on F4 from MUL and hence stalls till MUL is executed |
| stall |  |  | 11 | Addition takes 4 cycles and hence starts at 10 and ends at 13 |
| stall |  |  | 12 |  |
| stall |  |  | 13 |  |
| S.D | F6, 0(R2) | Y(i)=a.X(i)+Y(i) | 14 | Storage will take single cycle |
| DAADIU | R1, R1, #8 | Increment X index | 15 | The integer operation will take two cycles and end at 17th cycle |
| DAADIU | R2, R2, #8 | Increment Y index | 16 | This is intiger instruction and will take two cycles starts at 16 and ends at 17 |
| DSLTU | R3, R1, R4 | test: Continue loop? | 17 | This is integer instruction and will execute for two cycles atrts at 17 and ends at 18 |
| stall |  |  | 18 |  |
| BNEZ | R3, foo | loop if needed | 19 | This instruction will wait for R3 till 19th and executes the instruction |
| stall |  |  | 20 |  |

Execution time per element of Y is 20 cycles

Scheduled by compiler

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Operands** | **Comments** | **Clock cycle** |
| DAADIU | R4, R1, #800 |  | 1 |
| L.D | F2, 0(R1) |  | 2 |
| L.D | F6, 0(R2) |  | 3 |
| MUL.D | F4, F2, F0 |  | 4 |
| DAADIU | R1, R1, #8 |  | 5 |
| DAADIU | R2, R2, #8 |  | 6 |
| Stall |  |  | 7 |
| Stall |  |  | 8 |
| stall |  |  | 9 |
| ADD.D | F6, F4, F6 |  | 10 |
| DSLTU | R3, R1, R4 |  | 11 |
| stall |  | Waiting for R3 | 12 |
| BNEZ | R3, foo |  | 13 |
| S.D | F6, 0(R2) |  | 14 |

Execution time per element of Y is 14 cycles

**The scheduled execution time for unscheduled code is 19 clock cycles**

**The scheduled execution time for scheduled code is 14 cycles**

**The processor should be 6 cycles faster to complete the unscheduled code in 14 cycles.**

**6/14=42.86 % faster the processor should be**

b. [25] <3.2> Assume a single-issue pipeline. Unroll the loop as many times as necessary to schedule it without any stalls, collapsing the loop overhead instructions. How many times must the loop be unrolled? Show the instruction schedule. What is the execution time per element of the result?

|  |  |  |  |
| --- | --- | --- | --- |
| **Instruction** | **Operand** | **Clock cycle** | **Comments** |
| DAADIU | R4, R1, #800 | 1 | Single cycle |
| L.D | F2, 0(R1) | 2 | Single cycle |
| L.D | F6, 0(R2) | 3 | Single Cycle |
| MUL.D | F4, F2, F0 | 4 | Multiplication takes 6 cycles |
| L.D | F2, 8(R1) | 5 |  |
| L.D | F6, 8(R2) | 6 |  |
| MUL.D | F8, F2, F0 | 7 | Using different registers F8 |
| L.D | F2, 8(R1) | 8 |  |
| L.D | F14, 0(R2) | 9 | Using different registers F14 |
| MUL.D | F12, F2, F0 | 10 | Using different registers F12 |
| ADD.D | F6, F4, F6 | 11 | No dependency |
| DAADIU | R1, R1, #24 | 12 | No dependency use offset #24 |
| ADD.D | F10, F8, F10 | 13 | Using different registers F10 |
| DAADIU | R2, R2, #24 | 14 | Use offser #24 |
| DSLTU | R3, R1, R4 | 15 |  |
| ADD.D | F14, F12, F14 | 16 | Use different registers F14, F12 |
| S.D | F6, -24(R2) | 17 | Use different Offset |
| S.D | F6, -16(R2) | 18 | Use different offset |
| BNEZ | R3, foo | 19 |  |
| S.D | F14, -8(R2) | 20 | Different register and different offset so no dependency |

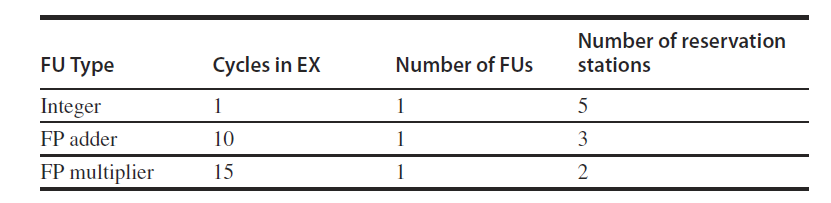
By Eliminating extra loops and test and ignoring loop overheads as shown above the instruction schedule can be unrolled 3 times without any stalls

No WAR and RAW are notice in the above loop unroll.

**Total cycles = 20 for 3 iterations**

**20/3= 6.65 per iteration**

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 table below.



**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 3.6 with the pipeline latencies from the table above. 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.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Operands** | **Comments** | **Issues at** | **Execute** | **Memory** | **Write  CDB** | **Iteration** |
| foo L.D | F2, 0(R1) | 1st issue Execution starts at 2nd cycle | 1 | 2 | 2 | 3 | 1 |
| MUL.D | F4, F2, F0 | 2nd issue Waits for F2 from first instruction and executes at 4th cycle multiple will continue executing till 19th cycle as it takes 15 cycles | 2 | 4 | 4 | 19 | 1 |
| L.D | F6, 0(R2) | 3rd issue this is single cycle instruction and executes at 4th cycle | 3 | 4 | 4 | 5 | 1 |
| ADD.D | F6, F4, F6 | 4th issue This instruction waits for F4 which is written in 19th cycle and continues execution at 20th cycle it stalls for 16 cycles | 4 | 20 | 20 | 30 | 1 |
| S.D | F6, 0(R2) | 5th issue stalls waiting for F6 for 26 cycles | 5 | 31 | 31 |  | 1 |
| DAADIU | R1, R1, #8 | 6th issue no stall | 6 | 7 | 7 | 8 | 1 |
| DAADIU | R2, R2, #8 | 7th issue no stall | 7 | 8 | 8 | 9 | 1 |
| DSLTU | R3, R1, R4 | 8th issue no stall | 8 | 9 | 9 | 10 | 1 |
| BNEZ | R3, foo | 9th issue no stall | 9 | 11 | 11 |  | 1 |
| foo L.D | F2, 0(R1) | 10th issue no stall | 10 | 12 | 12 | 13 | 2 |
| MUL.D | F4, F2, F0 | 11th issue stall wait for F4 till 19th cycle | 11 | 19 | 19 | 34 | 2 |
| L.D | F6, 0(R2) | 12th issue no stall | 12 | 13 | 13 | 14 | 2 |
| ADD.  D | F6, F4, F6 | 13th issue stall for 22 cycles wait for F4 | 13 | 35 | 35 | 45 | 2 |
| S.D | F6, 0(R2) | 14th issue Wait for ADD to provide F6 | 14 | 46 | 46 |  | 2 |
| DAADIU | R1, R1, #8 | 15th issue no stall | 15 | 16 | 16 | 17 | 2 |
| DAADIU | R2, R2, #8 | 16th issue no stall | 16 | 17 | 17 | 18 | 2 |
| DSLTU | R3, R1, R4 | 17th issue no stall | 17 | 18 | 18 | 19 | 2 |
| BNEZ | R3, foo | 18th issue wait for F3 | 18 | 20 | 20 |  | 2 |
| foo L.D | F2, 0(R1) | stall for one cycle | 19 | 21 | 21 | 22 | 3 |
| MUL.D | F4, F2, F0 | 20th issue Wait for F4 to free stall for 14 cycles Mul 15 cycles stall till 34-49 | 20 | 34 | 34 | 49 | 3 |
| L.D | F6, 0(R2) | 21st issue No stall | 21 | 22 | 22 | 23 | 3 |
| ADD.D | F6, F4, F6 | 22nd issue Wait for F4 22-49 cycles wait for ADD 50-60 | 22 | 50 | 50 | 60 | 3 |
| S.D | F6, 0(R2) | 23rd issue Wait for F6 | 23 | 60 | 60 |  | 3 |
| DAADIU | R1, R1, #8 | 24th issue | 24 | 25 | 25 | 26 | 3 |
| DAADIU | R2, R2, #8 | 25th issue | 25 | 26 | 26 | 27 | 3 |
| DSLTU | R3, R1, R4 | 26th issue | 26 | 27 | 27 | 28 | 3 |
| BNEZ | R3, foo | 27th issue | 27 | 29 | 29 |  | 3 |



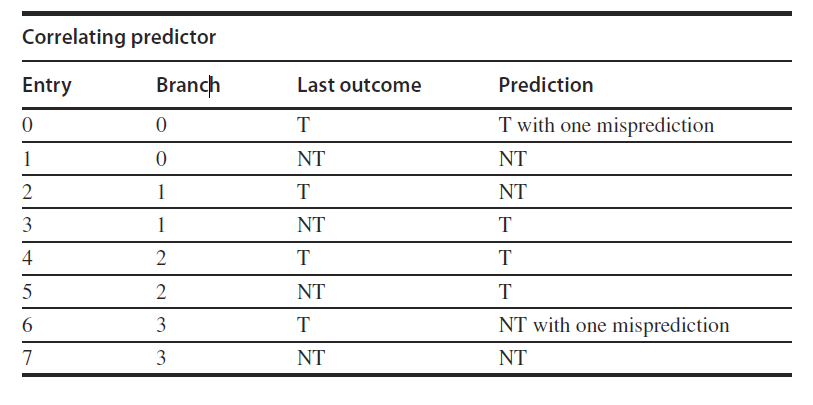
The above is the model of thomasulo’s that is used to provide solution for 3.15b question

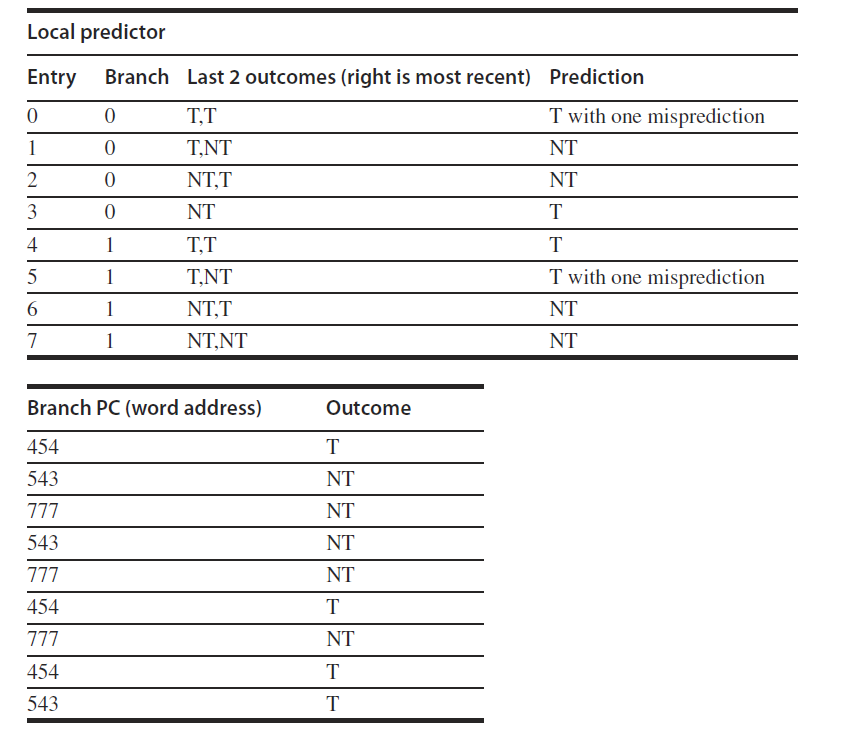
b. [20] <3.7, 3.8> Repeat part (a) but this time assume a two-issue Tomasulo algorithm and a fully pipelined floating-point unit (FPU).

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Instruction** | **Operand** | **Comments** | **Issues at** | **Exec** | **Memory** | **Write CDB** | **Iteration** |
| foo L.D | F2, 0(R1) | 1st issue Execution starts at 2nd cycle | 1 | 2 | 2 | 3 | 1 |
| MUL.D | F4, F2, F0 | Stall for two cycles 2-4 wait for F2 MUL for 15 cycles  till 19 | 1 | 4 | 4 | 19 | 1 |
| L.D | F6, 0(R2) | no stall | 2 | 3 | 3 | 4 | 1 |
| ADD.D | F6, F4, F6 | wait for F4 2-20 ADD from 20-30 | 2 | 20 | 20 | 30 | 1 |
| S.D | F6, 0(R2) | wait for F6 3-31 | 3 | 31 | 31 |  | 1 |
| DAADIU | R1, R1, #8 | no stall | 3 | 4 | 4 | 5 | 1 |
| DAADIU | R2, R2, #8 | noo stall | 4 | 5 | 5 | 6 | 1 |
| DSLTU | R3, R1, R4 | no stall | 4 | 6 | 6 | 7 | 1 |
| BNEZ | R3, foo | We have one INT  function unit and  it is busy in 5 cycle | 5 | 7 | 7 |  | 2 |
| foo L.D | F2, 0(R1) | Wait for R3 till  7th cycle | 6 | 8 | 8 | 9 | 2 |
| MUL.D | F4, F2, F0 | wait for F2 till 10th cycle | 6 | 10 | 10 | 25 | 2 |
| L.D | F6, 0(R2) | INT is busy 8-9 | 7 | 9 | 9 | 10 | 2 |
| ADD.D | F6, F4, F6 | wait for F4 till 26th cycle ADD busy from 26-36 | 7 | 26 | 26 | 36 | 2 |
| S.D | F6, 0(R2) | wait for F6 till 36th  cycle | 8 | 37 | 37 |  | 2 |
| DAADIU | R1, R1, #8 | No stall | 8 | 10 | 10 | 11 | 2 |
| DAADIU | R2, R2, #8 | No stall | 9 | 11 | 11 | 12 | 2 |
| DSLTU | R3, R1, R4 | INT is busy from  10-11 | 9 | 12 | 12 | 13 | 2 |
| BNEZ | R3, foo | lwait for R3 | 10 | 14 | 14 |  | 2 |
| foo L.D | F2, 0(R1) | Wait for BNEZ | 11 | 15 | 15 | 16 | 3 |
| MUL.D | F4, F2, F0 | Wait for F4 Mul reservation station  waiting for F2 Mul use 17-32 | 11 | 17 | 17 | 32 | 3 |
| L.D | F6, 0(R2) | integer reservation  station from 12-16 | 12 | 16 | 16 | 17 | 3 |
| ADD.D | F6, F4, F6 | wait for F4 12-33 in  reservation station | 12 | 33 | 33 | 43 | 3 |
| S.D | F6, 0(R2) | wait for F6 till 44 | 14 | 44 | 44 |  | 3 |
| DAADIU | R1, R1, #8 | FULL | 15 | 17 | 17 |  | 3 |
| DAADIU | R2, R2, #8 | FULL | 16 | 18 | 18 |  | 3 |
| DSLTU | R3, R1, R4 | FULL | 20 | 21 | 21 |  | 3 |
| BNEZ | R3, foo | FULL | 21 | 22 | 22 |  | 3 |

3.17 [20] <3.3> An (*m*,*n*) correlating branch predictor uses the behavior of the most recent *m* executed branches to choose from 2*m* 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:





**Correlating predictor**

**In the question it is given to use 1,2 correlating branch predictor which means to**

* **Use the outcome of the previous 1 branch executed**
* **And use a 2 bit branch predictor**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Branch PC mod 4** | **Entry** | **Prediction** | **Outcome** | **Table Update** | **Mis predicted** |
| 2 | 4 | T | T | - | no |
| 3 | 6 | NT | NT | change to “NT” | no |
| 1 | 2 | NT | NT | - | no |
| 3 | 7 | NT | NT | - | no |
| 1 | 3 | T | NT | change to “T with one misprediction” | yes |
| 2 | 4 | T | T | - | no |
| 1 | 3 | T | NT | change to “NT” | yes |
| 2 | 4 | T | T | - | no |
| 3 | 7 | NT | T | change to “NT with one misprediction” | yes |

3/9= 33.33 percentage of mispredicts

**Local predictor**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Branch PC Mod 2** | **Entry** | **Prediction** | **Outcome** | **Table update**  **Change to** | **Mis predict** |
| 0 | 0 | T | T | T | no |
| 1 | 4 | T | NT | T | yes |
| 1 | 1 | NT | NT | No change | no |
| 1 | 3 | T | NT | T | yes |
| 1 | 3 | T | NT | NT | yes |
| 0 | 0 | T | T | No change | no |
| 1 | 3 | NT | NT | No change | no |
| 0 | 0 | T | T | No change | no |
| 1 | 5 | T | T | T | no |

3/9= 33.33 percentage of mis predict

References:

<http://academic.csuohio.edu/yuc/comp-arch/lec04-loop_unroll.pdf>

<https://www.cs.umd.edu/class/spring2012/cmsc411/lectures/lec08.pdf>

<https://blackboard.syr.edu/webapps/blackboard/content/listContent.jsp?course_id=_332255_1&content_id=_3765553_1&mode=reset>

Computer Architecture A Quantitative Approach (5th edition) book