Analyzing the Techniques to Exploit Instruction Level Parallelism Using a Simulated Five Stage Pipelined RISC Processor

Amit Pandey, K. P. Yadav

Abstract— Any program whether it is using structural programming approach or object oriented programming approach must be executed instruction by instruction on the processor and all processors since 1985 are using pipelining to improve the performance by overlapped execution of instructions. The technique of exploiting the instructions by any possible overlapping is known as instruction level parallelism (ILP). There exist two different approaches to implement ILP. First approach is based on hardware and second is software-based approach. In the hardware-based approach, the ILP is exploited in run time, as the instructions are discovered and overlapped dynamically. On the other hand, to exploit ILP using software, static approach is taken to achieve parallelism at compile time. In this paper, we have discussed various approaches to exploit the ILP.

Index Terms— *Instruction level parallelism, ILP, Processor pipelining, Pipeline hazards, RISC hazards, True data dependencies, Loop unrolling*.

1 Introduction

USUALLY there are five different stages in a classical pipelined RISC processor namely Instruction fetch (IF), Instruction Decode (ID), Execution (EX), Memory (MEM) and Write back (WB). Each of these stages has its own functionalities [1].

IF is the first stage in the processor's pipeline organization. The Program Counter (PC) in the IF stage holds the address of the next instruction to be executed. This address value is continuously incremented at each clock pulse, so that it always points to the next instruction to be executed.

Next in the sequence is ID stage. Together with fetching and storing values to General Purpose Register set (GPR), branch detection and stall detection are the major functions performed by this stage. The branch detector circuit compares the values at the register addresses received with the branch instruction and decides whether the branch condition is true or not, alternatively, whether the branch will be taken or not. The stall detection circuit considers all the stall causing conditions and then decides whether the empty cycle or stall will be induced between the instructions or not.

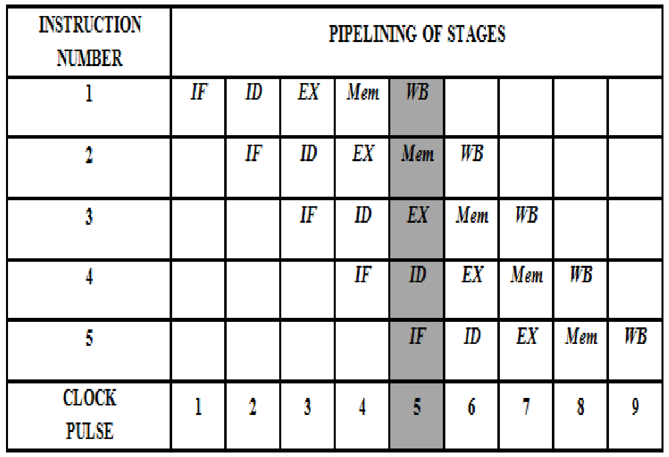
The EX stage is responsible for performing the arithmetic and logical operations by means of Arithmetic and Logic Unit (ALU) in this stage. The load and store memory operations are bypassed to next stage and are not handled by ALU.

The next is MEM stage. This stage has the memory chip and handles the load and store operations. To address any particular location in the memory, there are many possible addressing modes. Such as, direct addressing mode, immediate addressing mode, displacement addressing mode and indexed addressing mode. Load operation is responsible for loading any value from any particular location in the memory to any specific register and Store operation stores the value of any register to any specific location in the memory.

WB is the last stage in the sequence. This stage is responsible for writing the final result forwarded from ALU and MEM stage to the GPR set.

2 Understanding Instruction Level Parallelism

At every clock cycle, each stage forwards its instruction to the next consecutive stage in the sequence. Actually, each stage works as a sub unit and processes the instruction; alternatively, each stage is like a part of a channel or a pipe through which sequence of instructions is flowing.



INSTRUCTION NUMBER	PIPELINING OF STAGES								
1	IF	ID	EX	Mem	WB				
2		IF	ID	EX	Mem	WB			
3			IF	ID	EX	Mem	WB		
4				IF	ID	EX	Mem	WB	
5					IF	ID	EX	Mem	WB
CLOCK PULSE	1	2	3	4	5	6	7	8	9

Fig.1. Pipeline organization Simulation.

This is the reason why such processors are called pipelined processors and this technique of exploiting parallelism by overlapped execution of multiple instructions is called Pipelining *(See Fig. 1)*.

The main reason behind exploiting ILP is to minimize the cycles per instruction (CPI). The actual value of CPI for a pipe-

- *Amit Pandey is Ph.D scholar of Computer Science at Sunrise University Alwar, India. E-mail: amit.pandey@live.com*
- *Dr. K. P. Yadav (Superviser) is associated with IIMT College of Enginering, Gr Noida, India. E-mail: drkpyadav732@gmail.com*

lined processor is summation of ideal pipeline CPI and stalls induced by other hazards. By minimizing the stalls induced by the hazards, we can minimize the value of the actual pipeline CPI or, alternatively, increase the performance [1]. As shown in equation (1) and (2),

$$\text{Speedup Pipelined} = \frac{\text{Average instruction time unpipelined}}{\text{Average instruction time pipelined}}$$

$$= \frac{\text{CPI unpipelined X Clock Cycle time unpipelined}}{\text{CPI pipelined X Clock Cycle time pipelined}} \quad (1)$$

Aso,

$$\text{CPI pipelined} = \begin{pmatrix} \text{Ideal Pipeline CPI} + \text{Data Hazard Stalls} \\ +\text{Control Hazard Stalls} + \text{Structural Stalls} \end{pmatrix} \quad (2)$$

Actually, the ideal pipelined CPI can be assumed to be one. As it is always almost one for a pipelined architecture [1]. Hence, it can be expressed as shown in equation (3).

$$\text{CPI pipelined} = 1 + \text{pipeline Stall Cycles induced per instruction} \quad (3)$$

If all the stages of a pipelined processor are perfectly balanced. Then there will be ignorable cycle time overhead due to pipelining, alternatively, the clock cycle time for both the processors will be almost equal in such case [1].

Hence, the overall speedup for a pipelined processor can be expressed as shown in equation (4).

$$\text{Speedup Pipelined} = \frac{\text{CPI unpipelined}}{1 + \text{pipeline Stall Cycles induced per instruction}} \quad (4)$$

Also, on careful observation it can be seen that the CPI for a unpipelined processor is same as the CPI for executing a program with single instruction on a pipelined processor, alternatively, CPI for a unpipelined processor is equal to the number of pipelined stages also known as depth of the pipeline.

Hence, we can also express the overall speedup for a pipelined processor as shown in equation (5).

$$\text{Speedup Pipelined} = \frac{\text{Depth of the pipeline}}{1 + \text{pipeline Stall Cycles induced per instruction}} \quad (5)$$

One of the easiest ways to understand ILP is through loop iterations. Here the parallelism is exploited to increase the ILP within iterations of loop. This is usually known as loop level parallelism. The concept of loop level parallelism can be realized by understanding the processing of code mentioned below. Here, the iterations of loop will induce two arrays of 500 elements, running in parallel.

```
for ( j = 0; j < 500; j = j+1 )
    x[ j ] = x[ j ] + y[ j ]
```

Within each iteration, there is very little or almost no opportunity of overlapping. Hence, a proper technique is required to convert the loop level parallelism to ILP. Exploiting data level parallelism is a good solution for the mentioned situation. In data level parallelism, parallelism is imposed on vector operations by parallel use of data items. Four vector operations may be generated for the code mentioned above. As two will be required to load the data items j and i. One will be required to add them and one last operation will be required to write back the calculated result. However, when these vector operations are processed in parallel in a pipelined RISC processor, there may be some type of data dependences between them that can lead to unexpected results. This malfunctioning is known as Hazard [1]. There are three types of hazards possible in a pipelined architecture. Namely structural hazard, data hazard and control hazard.

The structural hazards are caused due to some design flaw in the datapath. The control hazards are caused because of the control dependence. The data hazards can further be classified in to three types namely Read after Write hazards (RAW), Write after Read hazards (WAR) and Write after Write hazards (WAW). The RAW hazards are caused because of the true data dependency. In a five stage pipelined RISC processor there are eight different cases of true data dependency [1],[2],[3].

2.1 TYPE I: Branch after Load

```
Load  RS1, # Offset
BRNEQ  RS3, RS1, Label
```

Here, hazard is caused because of RS1 register and can be resolved by inducing two stall cycles between the instructions.

2.2 TYPE II: ALU instruction after Load

```
Load  RS1, # Offset
SUB  RS4, RS1, RS3
```

Here, hazard is caused because of RS1 register and can be resolved by inducing single stall cycle between the instructions.

2.3 TYPE III: Branch after ALU instruction

```
SUB  RS1, RS2, RS3
BRNEQ  RS4, RS1, Label
```

Here, hazard is caused because of RS1 register and can be resolved by inducing single stall cycle between the instructions.

2.4 TYPE IV: Store after Load

```
LOAD RS1, # Offset
STORE RS1, # Offset
```

Here, hazard is caused because of RS1 register and can be resolved directly by data forwarding between the pipeline stages.

2.5 TYPE V: ALU after ALU instruction

```
SUB RS1, RS2, RS3
ADD RS4, RS1, RS5
```

Here, hazard is caused because of RS1 register and can be resolved directly by data forwarding between the pipeline stages.

2.6 TYPE VI: ALU as third instruction after another ALU Instruction

```
SUB RS1, RS2, RS3
ADD RS4, RS5, RS6
ADD RS7, RS8, RS1
```

Here, hazard is caused because of RS1 register and can be resolved directly by data forwarding between the pipeline stages.

2.7 TYPE VII: Branch as fourth instruction after ALU instruction

```
SUB RS1, RS2, RS3
ADD RS4, RS5, RS6
SUB RS7, RS8, RS9
BRNEQ  RS4, RS1, Label
```

Here, hazard is caused because of RS1 register and can be resolved directly by data forwarding between the pipeline stages.

2.8 TYPE VIII: Store after ALU instruction

```
SUB RS1, RS2, RS3
STORE RS4, RS1, #Offset
```

Here, hazard is caused because of RS1 register and can be resolved directly by data forwarding between the pipeline stages.

The WAR hazards are caused because of the antidependence type name dependency. Here, the WAR hazard is caused between two instructions due to RS3 register.

```
ADD RS1, RS2, RS3
SUB RS3, RS4, RS5
```

The WAW hazards are caused because of the output dependence type name dependency. Here, the WAW hazard is caused between two instructions due to RS1 register.

```
ADD RS1, RS2, RS3
SUB RS1, RS4, RS5
```

As, all the results are written to the GPR only in the WB stage. Therefore, the presence of WB stage eliminates all the possibilities of WAR and WAW hazards.

The instruction level parallelism (ILP) can be exploited over the instructions by marking out the non-dependent instructions that can be overlapped in the pipeline. There must be some pipeline latency in terms of clock cycles and possibly some pipeline stalls between the dependent and source instructions. Usually, such stalls are induced between the instructions because of control dependencies [4],[5],[6],[7],[8],[9],[10] and can be resolved using various static and dynamic branch prediction schemes [11],[12],[13],[14],[15],[16],[17],[18]. Further, compiler can exploit these stall cycles to reschedule the sequence of instruction without affecting the final output to increase the quantity of ILP. The compiler can increase the quantity of instruction level parallelism by rescheduling the instructions in an unrolled loop [1]. For a five stage pipelined RISC processor, the stall latencies induced between the instructions in different cases are mentioned below. *(See Table 1).*

TABLE 1
STALL LATENCIES IN A FIVE STAGE PIPELINED RISC PROCESSOR

Source Instruction	Dependent Instruction	Latency in clock cycles
Load	Branch	2
Load	ALU Operation	1
ALU Operation	Branch	1

Consider a block of code given below, which adds a vector value to a scalar value.

```
for ( i=0; i<3; i++)
A[i] = A[i] + S;
```

The block of code mentioned above can be translated in to the set of assembly level instructions given below. Here, R1 initially is the base address of the array and S1 contains the scalar value S. In addition, R2 refers to the pre-calculated address of the last element of the array.

```
LOOP:   Load S0, 0(R1)
        ADD S2, S0, S1
        Save S2, 0(R1)
        ADDImm R1, R1, #4
        BNE R1, R2, LOOP
```

On scheduling the above assembly level code for the pipeline, the below code segment with stalls will be obtained.

```
LOOP:   Load S0, 0(R1)
        STALL
        ADD S2, S0, S1
        Save S2, 0(R1)
        ADDImm R1, R1, #4
        STALL
        BNE R1, R2, LOOP
```

The above block of code requires 7 clock cycles for its execution, which can be rescheduled, by utilizing the empty stall cycles, to finish in just 5 clock cycles.

```
LOOP:   Load S0, 0(R1)
        ADDImm R1, R1, #4
        ADD S2, S0, S1
        Save S2, -4(R1)
        BNE R1, R2, LOOP
```

Code block mentioned above is for single iteration. However, compiler will run the loop until last iteration. Hence, the evaluation of intermediate ADDImm and branch conditions repeatedly with each iteration is an overhead. To achieve better performance compiler will unroll the complete loop and replace all the intermediate ADDImm and branch instructions with the modified ADDImm and branch instruction at the end of the code.

```
LOOP:   Load S0, 0(R1)
        STALL
        ADD S2, S0, S1
        Save S2, 0(R1)
        Load S3, 4(R1)
        STALL
        ADD S4, S3, S1
        Save S4, 4(R1)
        Load S5, 8(R1)
        STALL
        ADD S6, S5, S1
        Save S6, 8(R1)
        ADDImm R1, R1, #12
        STALL
        BNE R1, R2, LOOP
```

This process of expanding the loop body by replicating the iteration of the loop multiple times and then adjusting the code is known as loop unrolling. This technique improves the performance by modifying and eliminating the unnecessary branch overhead. Also, to avoid any possible data dependence, it is required that different registers must be used in each replicated loop. This use of unique registers will also increase the count of registers used during loop unrolling. The unrolled loop mentioned above still has stall cycles. So the final version of the above unrolled loop, scheduled for the pipeline and without any stall cycle is given below.

```
LOOP:   Load S0, 0(R1)
        Load S3, 4(R1)
        Load S5, 8(R1)
        ADD S2, S0, S1
        ADD S4, S3, S1
        ADD S6, S5, S1
        Save S2, 0(R1)
        Save S4, 4(R1)
        ADDImm R1, R1, #12
        Save S6, -4(R1)
        BNE R1, R2, LOOP
```

Loop unrolling is an efficient technique for utilizing the stall cycles induced because of control hazards. However, with the limited number of general-purpose registers in RISC style architecture, sometimes it becomes difficult to apply this technique over the loop with heavy register usage. As this technique is completely based on register renaming. Consider any such loop with upper bound of 'n' iterations. Now to handle such loop we have to partition the 'n' iterations into groups of 'K' iterations. The first group iterates for 'n mod K' times and following it are the remaining groups, which will iterate for 'n/K' times. In this way, we do not have to unroll the complete 'n' iterations at the same time and hence reduce the load of register renaming.

TABLE 2
THE TEST CODE I

Code	Instruction Opcode	Expected Outcome
Load R2, #2	1011	R2<<2
Load R3, #1	1011	R3<<1
Load R4, #1	1011	R4<<1
BREQ R3, R4, #2	1101	Branch Taken
ADD R3, R2, R2	0010	Eluded
ADD R4, R4, R2	0010	R4<<3
BRNOTEQ R3, R3, #3	1110	Not Taken
ADDIMM R3, R3, #6	0011	R3<<7

3 RESULTS FROM SIMULATED RUN

3.1 Running the code with data hazards and control hazards

To understand the Instruction Level Parallelism (ILP) in the practical aspects, we have implemented the below code on a simulated five stage pipelined RISC processor *(See Table 2)*.

The figure shows the result obtained after executing the code on a simulated pipelined processor *(See Fig. 2)*. Here, WIR15 to WIR12, shows the operation (Opcode), where WIR15 is the most significant bit and RD7 to RD0, shows the results obtained for each operation, where RD7 is the most significant bit [19].

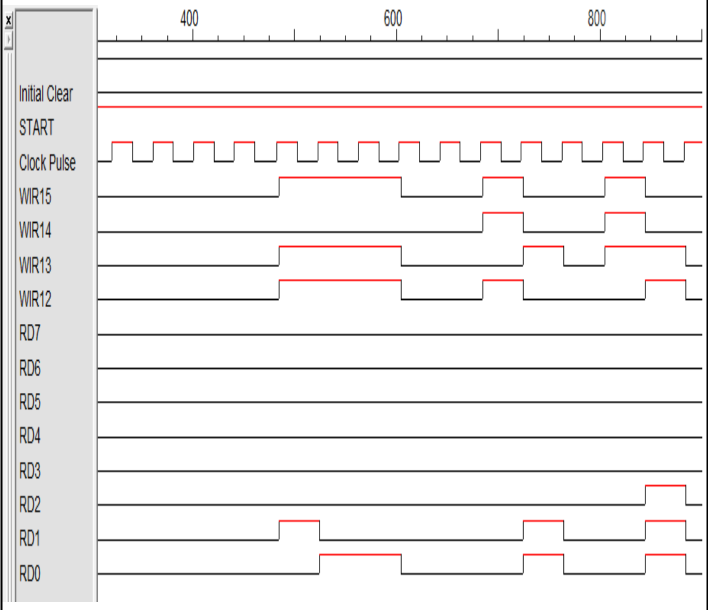


Fig.2. Result for test code I.

In Fig. 2 there are two stalls induced between the third load instruction and the fourth branch-equal instruction to resolve the data hazard. In addition, there is one stall induced between the Add instruction and the second branch-not equal instruction for the same reason.

3.2 Running the code after loop unrolling

To understand the Loop Unrolling in the practical aspects we have implemented the below code on a simulated five stage pipelined RISC processor.

```
for ( i=0; i<2; i++)
X = X + C;
```

The block of code mentioned above can be translated in to the set of assembly level instructions given below. Here, I is representing 'i' and J holds the upper limit to which 'i' can be increment.

```
LOOP: Load X, R1
Load C, R2
Load I, R3
Load J, R4
LOOP:   ADD X, X, C
ADDImm I, I, #1
BNE I, J, LOOP
```

On scheduling the above assembly level code for the pipeline, the below code segment with stalls will be obtained.

```
LOOP: Load X, R1
Load C, R2
Load I, R3
Load J, R4
LOOP:   ADD X, X, C
ADDImm I, I, #1
STALL
BNE I, J, LOOP
```

Code block mentioned above is only one iteration. However, compiler will run the loop until last iteration. Hence, the evaluation of intermediate ADDImm and branch conditions repeatedly with each iteration is an overhead. To achieve better performance compiler will unroll the complete loop and replace all the intermediate ADDImm and branch instructions with the modified ADDImm and branch instruction at the end of the code.

```
LOOP: Load X, R1
Load C, R2
Load I, R3
Load J, R4
LOOP:   ADD X, X, C
ADD X, X, C
ADDImm I, I, #2
STALL
BNE I, J, LOOP
```

The unrolled loop mentioned above still has stall cycles. So the final version of the above-unrolled loop, scheduled for the pipeline and without any stall cycle is given below.

```
LOOP: Load X, R1
Load C, R2
Load I, R3
Load J, R4
LOOP:   ADD X, X, C
ADDImm I, I, #2
ADD X, X, C
BNE I, J, LOOP
```

TABLE 3
THE TEST CODE II

Code	Instruction Opcode	Expected Outcome
Load X, #3	1011	X<<3
Load C, #2	1011	C<<2
Load I, #0	1011	I<<0
Load J, #2	1011	J<<2
ADD X, X, C	0010	X<<5
ADD Imm I, I, #2	0011	I<<2
ADD X, X, C	0010	X<<7
BNE I, J, Loop	1110	Not Taken

The Fig. 3 shows the result obtained after executing the code, shown in Table 3, on a simulated pipelined processor. Here, WIR15 to WIR12 shows the operation (Opcode), where WIR15 is the most significant bit and RD7 to RD0 shows the results obtained for each operation, where RD7 is the most significant bit [19].

It can be seen that, because of the loop unrolling there are no induced stalls between instructions, in the figure given below *(See Fig. 3)*.

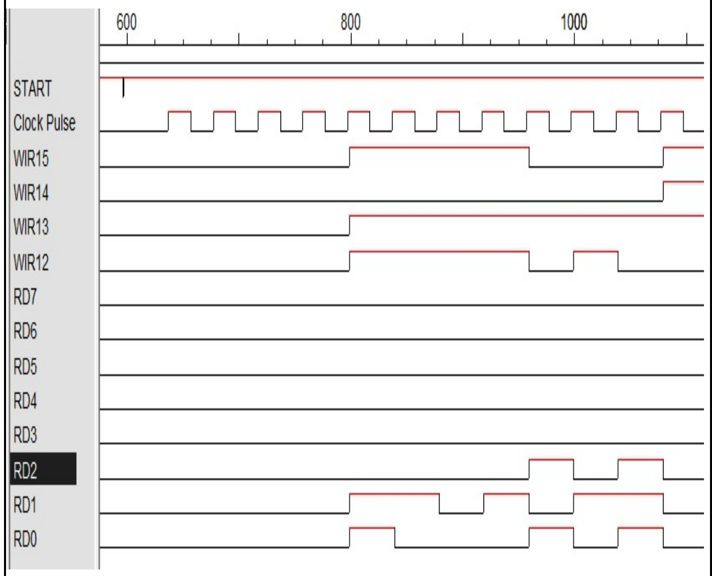


Fig.3. Result for test code II.

It can be observed that because of successful exploitation of ILP we are able to execute the code in Table II, with instructions running in parallel in five different stages of a pipelined RISC processor, only in 15 clock cycles. On the other hand, same code can take 40 clock cycles if executed sequentially instruction by instruction on the same processor.

4 Conclusion

In In the current study, we have discussed the theoretical as well as practical aspects of instruction level parallelism (ILP) by means of a simulated five stage pipelined RISC processor. By means of this study, we have shown that instruction level parallelism (ILP) is an efficient technique that we have successfully implemented in our simulated five stage pipelined RISC processor.

References

[1] J.L. Hennessy and D.A. Patterson, *Computer architecture: a quantitative* approach, Elsevier, 2012.

[2] A. Pandey, "Stall estimation metric: An architectural metric for estimating software complexity," *Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO), 2016 5th International Conference on*, IEEE, 2016. doi: 10.1109/ICRITO.2016.7784987.

[3] A. Pandey, "Study of data hazard and control hazard resolution techniques in a simulated five stage pipelined RISC processor," *Inventive Computation Technologies (ICICT), International Conference on*, IEEE, Vol. 2, 2016. doi:10.1109/INVENTIVE.2016.7824864.

[4] J. Lee and A. Smith, Branch prediction strategies and branch target buffer design, *In Instruction-level parallel processors*, IEEE Computer Society Press, pp. 83-99, 1995. doi: 10.1109/MC.1984.1658927.

[5] Z. Su and M. Zhou, "A comparative analysis of branch prediction schemes," University of California at Berkeley, Computer Architecture Project, 1995.

[6] T. Ball and J.R. Larus, "Branch prediction for free," ACM, Vol.28, No. 6, pp. 300-313, 1993. doi: http://dx.doi.org/10.1145/155090.155119.

[7] J.B. Chen, M.D. Smith,C. Young and N. Gloy, "An analysis of dynamic branch prediction schemes on system workloads," *23rd Annual International Symposium on Computer Architecture*. IEEE, pp. 12-12, 1996.

[8] D.A. Jiménez, and C. Lin, "Dynamic branch prediction with perceptrons," *In High-Performance Computer Architecture The Seventh International Symposium on HPCA*, IEEE, pp. 197-206, 2001. doi: 10.1109/HPCA.2001.903263.

[9] T.Y. Yeh, and Y.N. Patt, "Two-level adaptive training branch prediction," *In Proceedings of the 24th annual international symposium on Microarchitecture*, ACM, pp. 51-61, 1991.

[10] C.C. Cheng, "The schemes and performances of dynamic branch predictors," http://bwrcs.eecs.berkeley.edu/Classes/CS252/Projects/Reports/terry_chen.pdf. 2000.

[11] C.A. Moritz, "Computer Architecture: Dynamic Branch Prediction," www.ecs.umass.edu/ece/andras/courses/ECE668/Mylectures/brnchprdct.ppt. 2017.

[12] J.E. Smith, "A study of branch prediction strategies," *In Proceedings of the 8th annual symposium on Computer Architecture*, IEEE Computer Society Press, pp. 135-148, 1981.

[13] J.A. Fisher and S.M. Freudenberger, "Predicting conditional branch directions from previous runs of a program,"ACM, Vol. 27, No. 9, 1992.

[14] N. Gloy, C. Young, J.B. Chen and M.D. Smith, "Analysis of dynamic branch prediction schemes on system workloads," *in Conf. Proc. - Annu. Int. Symp. Comput. Archit. ISCA*, pp. 12–21, 1996. doi:10.1145/232974.232977.

[15] C. Young, N. Gloy and M.D. Smith, "Comparative analysis of schemes for correlated branch prediction," *in ACM SIGARCH (Association Comput. Nachinery Spec. Interes. Gr. Comput. Archit. - Conf. Proc.*, pp. 276–286, 1995. doi:10.1145/225830.224438.

[16] S. Mcfarling, "Combining Branch Predictors," *West. Res. Lab.*, 1–29, http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-TN-36.pdf. 1993.

[17] S.T. Pan, K. So and J.T. Rahmeh, "Improving the accuracy of dynamic branch prediction using branch correlation," *in ACM Sigplan Notices*, Vol. 27, No. 9, pp. 76-84, 1992.

[18] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt, "An analysis of correlation and predictability: What makes two-level branch predictors work," *in ACM SIGARCH Computer Architecture News*, ACM, Vol. 26, No. 3, pp. 52-61, 1998.

[19] A. Pandey, " Simulating a pipelined RISC processor," *Inventive Computation Technologies (ICICT), International Conference on*, IEEE, Vol. 2, 2016. doi: 10.1109/INVENTIVE.2016.7824854.