Deploying a Simulated RISC Style Pipelined Processor for Evaluating Stall Estimation Metric against Weyuker's Properties

Amit Pandey, K. P. Yadav

Abstract— Stall estimation metric considers program dependencies together with statement count for determining the software complexity. In this study, we have evaluated the stall estimation metric against the different Weyuker's properties. Further, we have used these results to compare the stall estimation metric with other existing software metrics. We have generated a simulated pipelined RISC processor to evaluate various cases. Various test cases were generated by taking stall causing data dependencies together in account with stall causing control dependencies. Then we have practically implemented these cases on a simulated pipelined RISC processor to obtain the results in form of printed pulse patterns. Using the results obtained from the study, we have shown that stall estimation metric is an efficient metric that follows all the nine Weyuker's properties.

Index Terms— Stall estimation metric, Software Complexity, Architectural metric, Weyuker's Properties, Code complexity, Complexity metric, Program complexity.

1 Introduction

NUMEROUS software metrics have been proposed by researchers till date [1],[2]. Some of them analyze the structure of code lines or the arrangement of code blocks in the software [3],[4],[5]. While, others analyze how they are processed on the hardware platform by studying any dependencies between them [3],[6]. Stall estimation metric belongs to the later class that studies the dependencies on a hardware plateform to determine the complexity of any program [7].

In last three decades lots of work has been done in the field of software complexity estimation and numerous metrics have been proposed by researchers. But most of the work done focuses the code metrics and scope for further exploration in the field of architectural metrics is still open.

Kafura has studied the information flow of the system and proposed some architectural metrics [8]. He stated that any module in the system with higher information flow is considered to be more complex than any other module which has lower information flow. The key problem with these information flow metrics is that they were defined using some informal structure charts and unlike the proposed metrics they are incapable of incorporating all the system architecture attributes properly.

Zhao has defined architectural metrics based on the count of dependence arcs present in an architectural dependence graph of software architecture. He considered the total number of existing program and architectural dependencies as measure for estimating the software complexity [3]. Further in the same category of architectural metrics, Stall estimation metric was proposed in 2016, this metric considers the number of stalls induced during the resolution of dependencies together with the count of statements actually executed as measure for evaluating software complexity [7],[9]. The Stall estimation metric considers only the stall inducing dependencies. As other dependencies can be resolved by simply data forwarding between the stages of the processor and will not actually affect the execution time of the program.

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

Considering the information provided in Table 1, the complexity estimation of any program using Stall estimation metric (SEC) can be expressed as,

SEC = Sc+2.DLd.Br.a+DLd.Br.b+DLd.ALU+DALU.Br+BrPr *(1)*
Where,
SEC = Software complexity using Stall estimation metric.
DLd.Br.a = Occurrences of data dependency when Control instruction (Branch instruction) is just after Load instruction as mentioned in Case I (1).
DLd.Br.b = Occurrences of data dependency when Control instruction (Branch instruction) is third instruction after Load instruction as mentioned in Case I (2). Here there is no stall inducing dependency between Load and the next following instruction.
DLd.ALU = Occurrences of data dependency when the ALU instruction is just after the Load instruction as mentioned in Case II.
DALU.Br = Occurrences of data dependency when Control instruction (Branch instruction) is just after ALU instruction as

- *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*

mentioned in Case III.
BrPr = Considering the worst case, induction of one stall cycle during the resolution of Control hazards.
Sc = It is the count of statements in the program that are actually executed.

Our current study is focused on evaluation of Stall estimation metric against Weyuker's Properties. In this study, we have designed various test cases to chek Stall estimation metric, against Weyuker's Properties, by executing them on a simulated RISC style, pipelined processor.

Table 2 shows the instruction set for the simulated RISC processor [10]. The opcodes mentioned in the table are required to comprehend the results achieved from the simulation.

TABLE 2
INSTRUCTION SET FOR SIMULATED PROCESSOR

ALU Instructions	
OP.Code	Instruction
0000	No Operation
0001	Subtraction
0010	Addition
0011	Add Immediate
0100	Shift Right
0101	Shift Left
0110	Logical And
0111	Logical OR
1000	Logical NOR
1001	Logical NAND
1010	Logical XOR
Load - Store Instructions	
OP.Code	Instruction
1011	Load
1100	Store
Branch - Jump Instructions	
OP.Code	Instruction
1101	Branch Equal
1110	Branch Not Equal
1111	Jump

In the results obtained from the simulated processor. The four-bit value from WIR15 to WIR12 represents the opcode of the executed instruction. Here, WIR15 is the most significant bit. In addition, eight-bit value from RD7 to RD0 represents the output. Here, RD7 is the most significant bit.

2 Evaluation of Stall Estimation Metric Against Weyuker's Properties

Weyuker has proposed nine properties which are used as criteria to test the effectiveness of any software metrics [11],[12],[13],[14],[15],[16]. More the number of properties any software metric satisfies, better it is. In this part of study, we will evaluate the proposed stall estimation metric against Weyuker's properties and show that the proposed metric follows all nine properties.

2.1 Property 1: There exist two programs P and Q such that |P| ≠ |Q|.

P and Q are two programs with different complexity values.

Program P:
Load R1, #3
Load R2, #1
ADD R3, R2, R1

When we analyze the simulation results of program P for property 1 *(See Fig. 1)*. We will see that there is one stall induced between the second load and third instruction. Therefore, the complexity of the code will be,
SEC_1P = Sc + DLd.ALU = 3 + 1 = 4

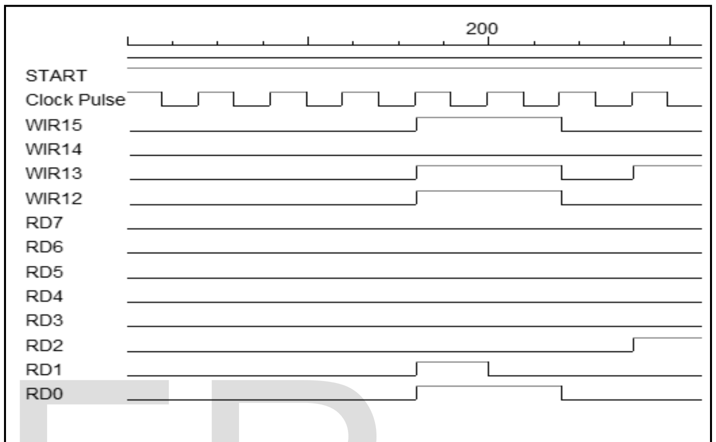

Fig. 1. Simulation result of program P for property 1.

Program Q:
Load R1, #3
Load R2, #1
ADD R3, R2, R1
ADD R4, R3, R1
SUB R5, R4, R1

Now after analyzing the results of program Q for property 1 *(See Fig. 2)*. We find that there is one stall between the second load and third instruction. Therefore, the complexity of the code will be,
SEC_1Q = Sc + DLd.ALU = 5 + 1 = 6

Hence, in this case we have shown that |P| ≠ |Q|.

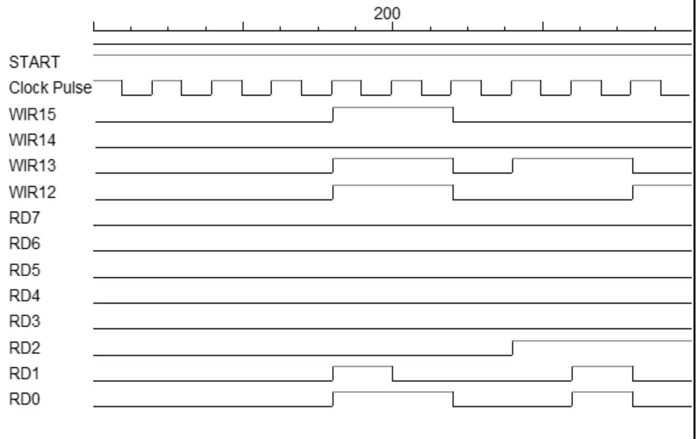

Fig. 2. Simulation result of program Q for property 1.

2.2 Property 2: For any non-negative number 'C', there are only finite many programs with complexity 'C'.

As for the proposed Stall estimation metric, the estimation of software complexity is based on the sum of statement count and the number of stalls in the program. Therefore, it can easily be perceived that there will be only finite possible combinations of defined stall cases and program statements resulting in fixed software complexity of 'C'.

2.3 Property 3: There are distinct programs P and Q such that |P| = |Q|.

Below are two programs P and Q with same complexity value.

Program P:
```
Load  R1, #3
Load  R2, #1
ADD  R3, R2, R1
BREQ  R1, R3, #2
```

Analyzing the simulation results of program P for property 3 *(See Fig. 3)*. We will see that there are two stalls induced between the Second Load and third instruction and third Add instruction and last instruction. So, the complexity of the code can be evaluated as,

SEC_3P = Sc + DLd.ALU + DALU.Br
= 4 + 1 + 1 = 6

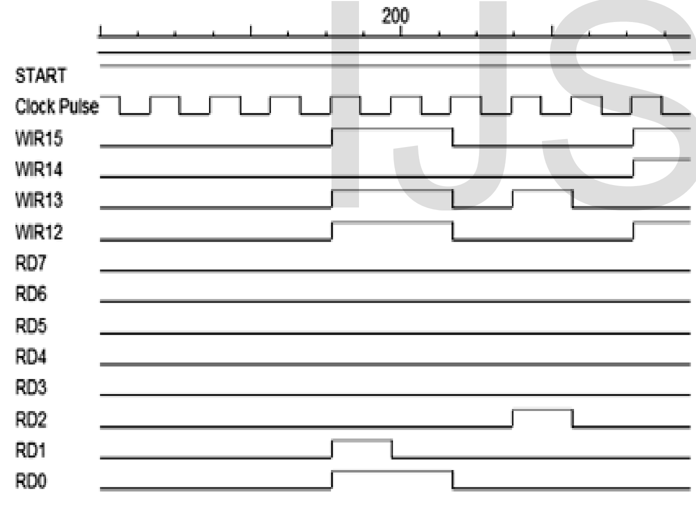

Fig. 3. Simulation result of program P for property 3.

Program Q:
```
Load  R1, #3
Load  R2, #1
ADD  R3, R2, R1
ADD  R4, R1, R3
SUB  R5, R4, R1
```

While analyzing the results of program Q for property 3 *(See Fig. 4)*. We find that there is one stall between the second load instruction and third instruction. Therefore, the complexity of the code will be,

SEC_3Q = Sc + DLd.ALU
= 5 + 1 = 6

Hence, in this case we have shown that |P| = |Q|.

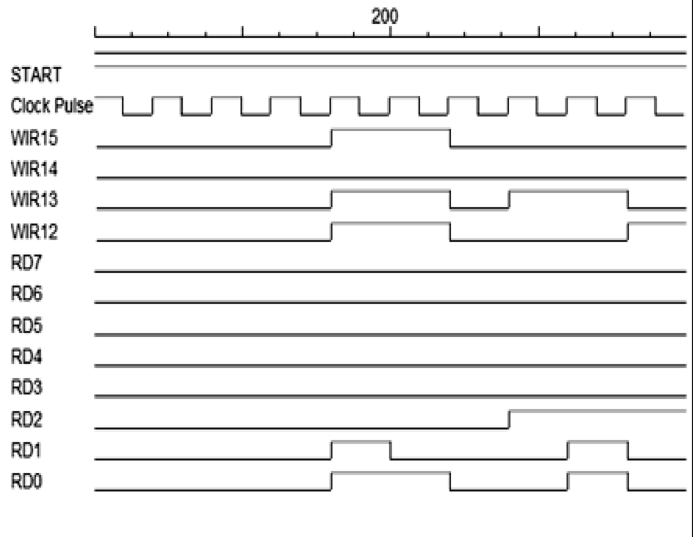

Fig. 4. Simulation result of program Q for property 3.

2.4 Property 4: There exist two equivalent programs P and Q such that |P| ≠ |Q|.

This property states that for any program there can be another more complex program with same functionality. To prove this we have considered the two programs P and Q, which will left shift any binary number by three places.

Program P:
```
Load  R2, #3
Load  R1, #1
LShift  R3, R1, R2
```

Analyzing the simulation result of program P for property 4 *(See Fig. 5)*. We will see that there is single stall induced between the Second Load and third instruction. So the complexity of the program can be evaluated as,

SEC_4P = Sc + DLd.ALU
= 3 + 1 = 4

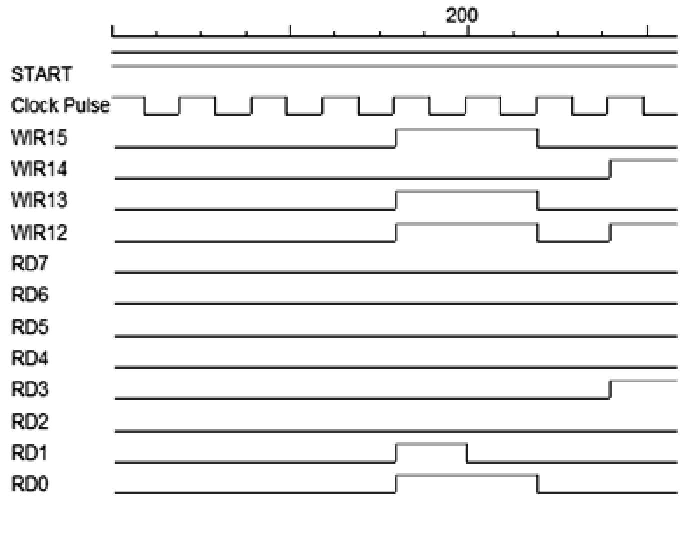

Fig. 5. Simulation result of program P for property 4.

Program Q:

Load R1, #1
Add R2, R1, R1
Add R3, R2, R2
Add R4, R3, R3

While analyzing the simulation result for program Q for property 4 *(See Fig. 6)*. We find that there is one stall between the first load instruction and second instruction. So the complexity of the code will be,

SEC_4Q = Sc + DLd.ALU
= 4 + 1 = 5

Hence, in this case we have shown that |P| ≠ |Q|.

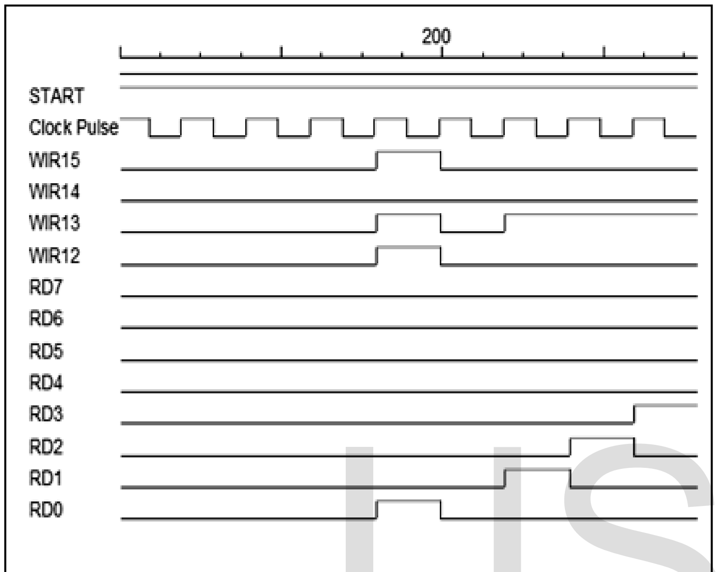

Fig. 6. Simulation result of program Q for property 4.

2.5 Property 5: For all programs P and Q, |P| ≤ |P:Q| and |Q| ≤ |P:Q|

Let us consider that |P| is SEC_|P| and |Q| is SEC_|Q|. Then software complexity for the concatenated program P:Q will be,

$$|P{:}Q| = SEC\_|P| + SEC\_|Q| + C \qquad (2)$$

Here,

$$C = \begin{cases} 0\,(Whenthereisnodependencebetween PandQ) \\ NumberofStalls\,(Whenstallinducingdependencesexistebetween PandQ) \end{cases}$$

So, from the above expression (2) it can be inferred that,
|P| < |P:Q| and |Q| < |P:Q|

2.6 Property 6: Exists programs P, Q and R. Such that,
a) |P| = |Q| and |P:R| ≠ |Q:R| holds.
b) |P| = |Q| and |R:P| ≠ |R:Q| holds.
Also for both the cases 6 (a) and 6 (b), |P:R| ≠ |R:P|.

Let there exist programs P, Q and R, as shown below. Here, register R1and R4 are type of data segment register and stores static, extern and global values.

Program P:
ADDImm R1, R1, #2
ADDImm R2, R0, #2
ADD R3, R2, R2

Here, the Complexity value for P will be *(See Fig. 7)*,
SEC_6P = Sc = 3

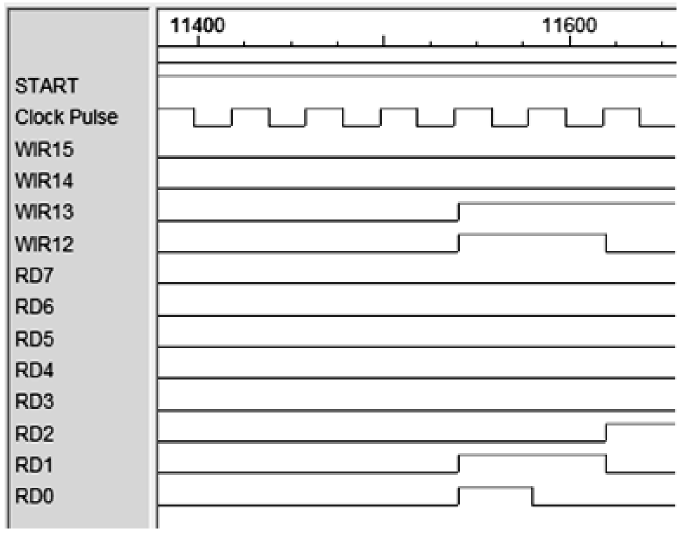

Fig. 7. Simulation result of program P for property 6.

Program Q:
ADDImm R2, R0, #1
ADD R3, R2, R2
Load R4, #2

Here, the Complexity value for Q will be *(See Fig. 8)*,
SEC_6Q = Sc = 3

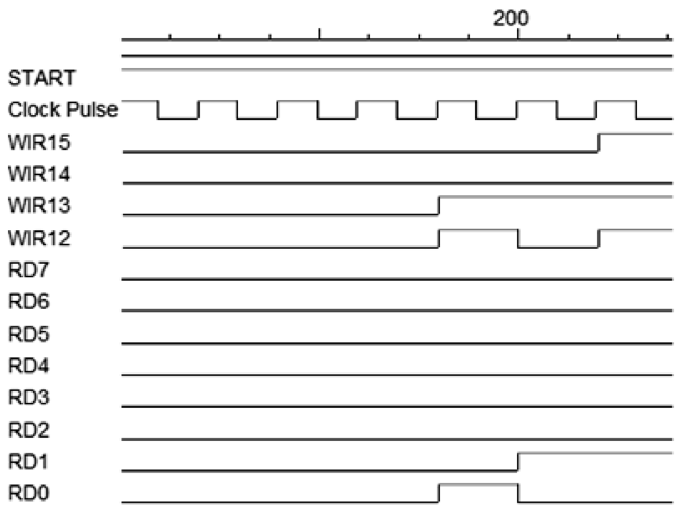

Fig. 8. Simulation result of program Q for property 6.

Program R:
ADDImm R5, R4, #3
Load R1, #3

Proof for 6(a). At this point, it can be concluded from Fig.7

and Fig. 8 that |P| = |Q|.

Now, simulation result for |P:R| is *(See Fig. 9)*,

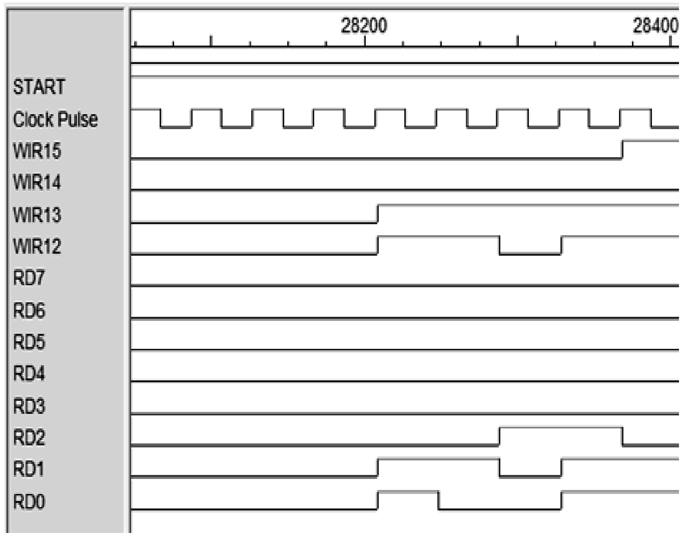

Fig. 9. Simulation result of program P:R for property 6(a).

The Complexity value for |P:R| will be,
SEC_6aPR = Sc = 5

Also, simulation result for |Q:R| is *(See Fig. 10)*,

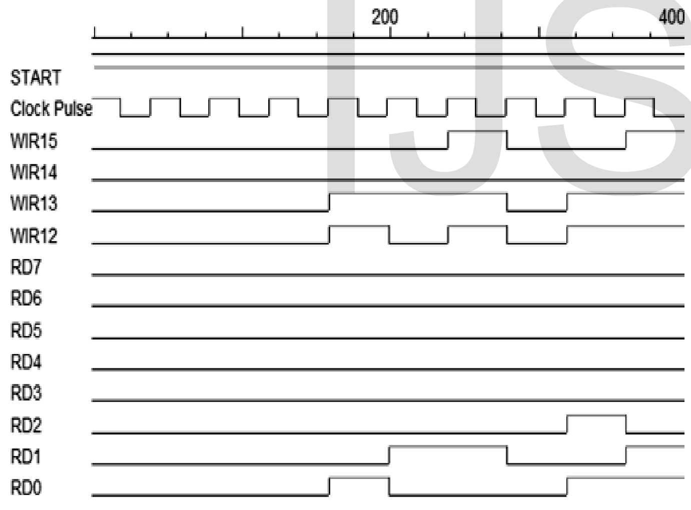

Fig. 10. Simulation result of program Q:R for property 6(a).

There is a stall inducing dependency between last instruction of Q and first instruction of R. Due to which the Complexity value for |Q:R| comes out to be,
SEC_6aQR = Sc + DLd.ALU
= 5 + 1 = 6

Now using the results of both the simulations *(See Fig.9, Fig.10)*, it can be clearly stated that,
|P:R| ≠ |Q:R|

Proof for 6(b): As from Fig. 7 and Fig. 8, it is clear that |P| = |Q|.

Now, simulation result for |R:P| is *(See Fig. 11)*,

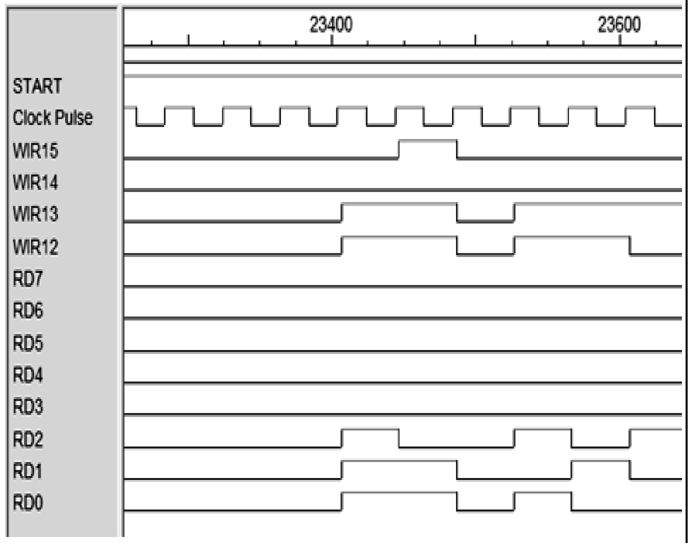

Fig. 11. Simulation result of program R:P for property 6(b).

There is a stall inducing dependency between last instruction of R and first instruction of P. Due to which the Complexity value for |R:P| comes out to be,
SEC_6bRP = Sc + DLd.ALU
= 5 + 1 = 6

Also, simulation result for |R:Q| is *(See Fig. 12)*,

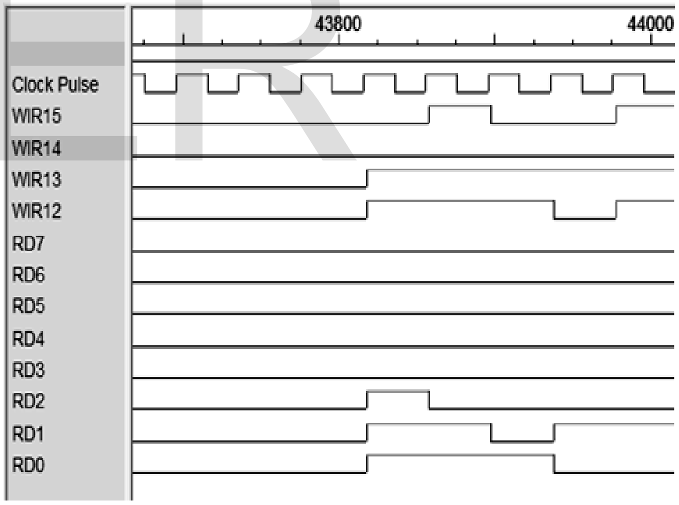

Fig. 12. Simulation result of program R:Q for property 6(b).

The Complexity value for |R:Q| will be,
SEC_6bRQ = Sc = 5

Now using the results of both the simulations *(See Fig.11, Fig.12)*, it can be clearly stated that,
|R:P| ≠ |R:Q|

In addition, if we will see we will find that requirement for considering 6(a) and 6(b) as two different cases is also satisfied. That is,
|P:R| ≠ |R:P|

As, complexity values for |P:R| and |R:P| are 5 and 6 respectively.

2.7 Property 7: There are programs P and Q. Such that

program Q can be obtained by permuting statements of program P, and |P| ≠ |Q|.

Here we have two programs P and Q. The program Q is obtained after permuting the statements of P.

Program P:
Load R1, #3
ADDImm R2, R1, #1
ADDImm R3, R0, #3
ADD R4, R3, R2

There is a data dependency between the first load instruction and second instruction. This will induce stall between them. This can also be seen in simulation result *(See Fig. 13)*.
So the software complexity can be calculated as,
SEC_7P = Sc + DLd.ALU = 4 + 1 = 5

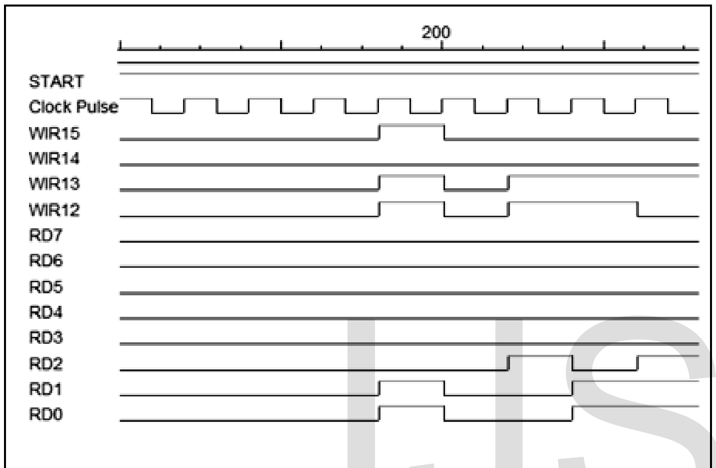

Fig. 13. Simulation result of program P for property 7.

Program Q:
Load R1, #3
ADDImm R3, R0, #3
ADDImm R2, R1, #1
ADD R4, R3, R2

Here, the software complexity can be calculated as shown below. This can also be verified from the simulation result shown in Fig. 14.

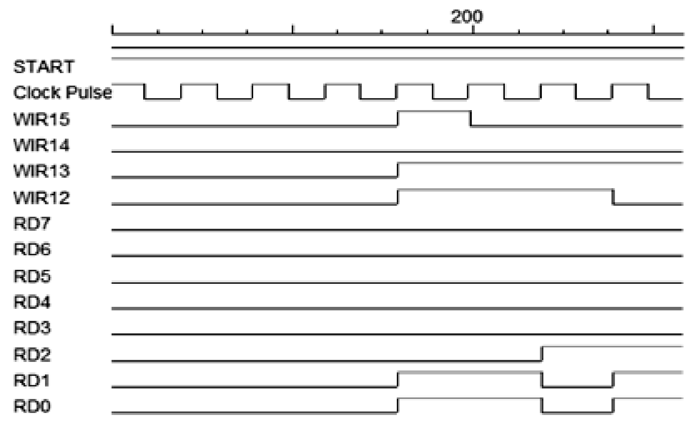

Fig. 14. Simulation result of program Q for property 7.

SEC_7Q = Sc = 4

Hence, we have shown that in this case |P| ≠ |Q|.

2.8 Property 8: If P is renamed to Q. Then |P| = |Q|.

This property states that for two given programs P and Q, where Q is obtained by renaming P, such that the meaning of the program is conserved. Then |P| = |Q|. Let us consider two programs P and Q.

Program P:
Load R1, #3
Load R2, #1
Add R3, R1, R2

Here there is stall inducing data dependency between second load and last instruction *(See Fig. 15)*. So the software complexity can be calculated as,
SEC_8P = Sc + DLd.ALU
= 3 + 1 = 4

Program Q:
Load R4, #3
Load R5, #1
Add R6, R4, R5

Here there is stall inducing data dependency between second load and last instruction *(See Fig. 15)*. So the software complexity can be calculated as,
SEC_8Q = Sc + DLd.ALU
= 3 + 1 = 4

Hence, we have shown that in this case |P| = |Q|.

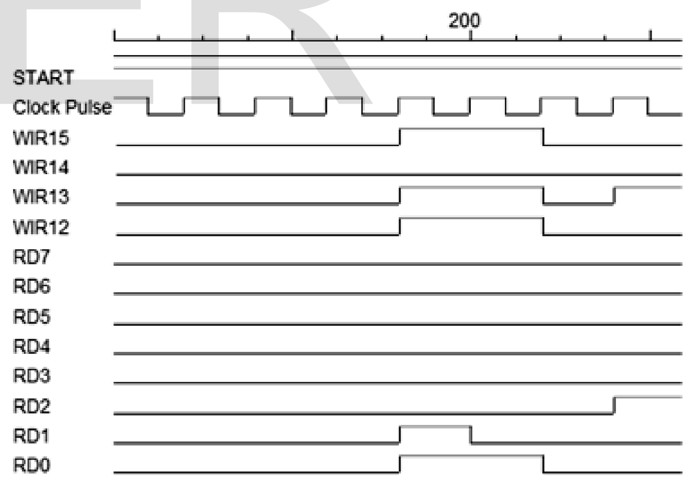

Fig. 15. Simulation result of programs P and Q for property 8.

2.9 Property 9: There are programs P and Q. Such that |P| + |Q| < |P:Q|.

Let us consider two programs P and Q as given below.

Program P:
ADDImm R2, R0, #1
ADD R3, R2, R2
ADDImm R3, R3, #2
Load R4, #2

Simulation result of program P is shown in Fig 16. The program complexity in this case can be evaluated as shown below,

SEC_9P = Sc = 4

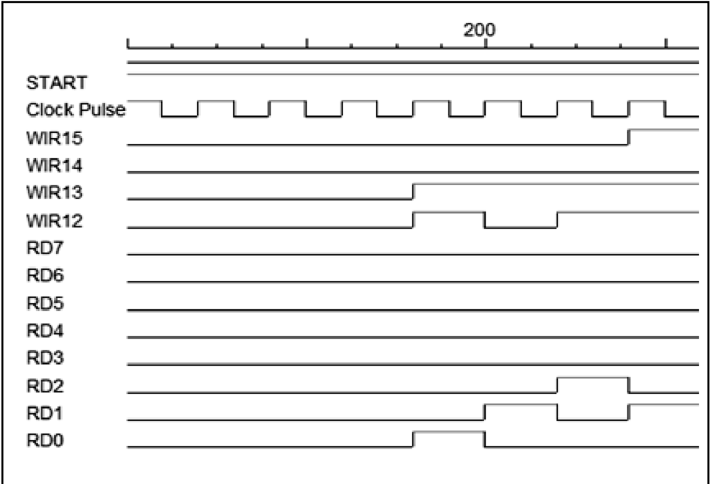

Fig. 16. Simulation result of programs P for property 9.

Program Q:
ADDImm R4, R4, #2
Load R1, #3
ADDImm R5, R1, #4

Here, the second load instruction and last instruction has a stall inducing data dependency. This can also be seen in the simulation result shown in Fig. 17. Now, the software complexity can be evaluated as,

SEC_9Q = Sc + DLd.ALU
= 3 + 1 = 4

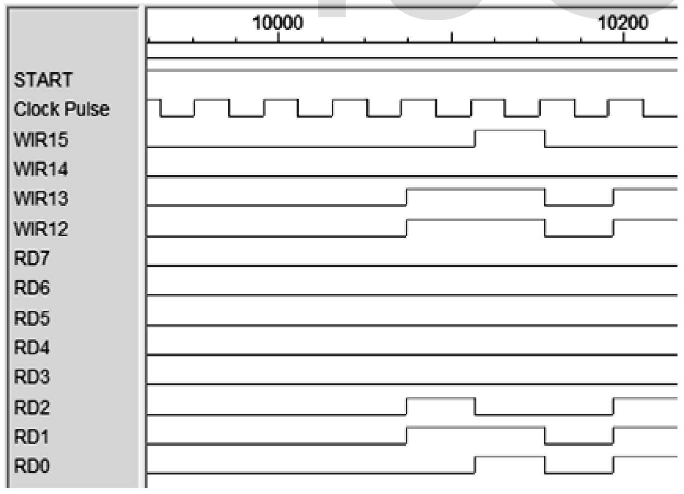

Fig. 17. Simulation result of programs Q for property 9.

Now we execute the concatenation of program P and Q to obtain the simulation result shown in Fig. 18. There is stall inducing data dependency between last instruction of P and first instruction of Q. Also one stall inducing data dependency is present inside Program Q as mentioned above. Keeping these things in mind the software complexity can be evaluated as,

SEC_9PQ = Sc + DLd.ALU
= 7 + 2 = 9

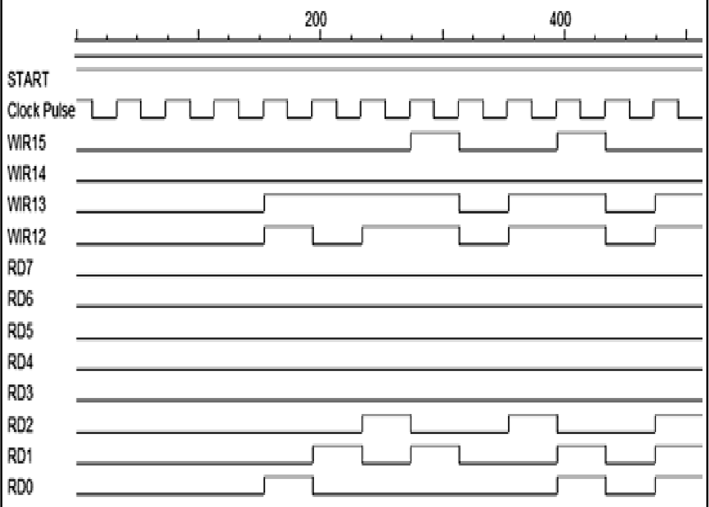

Fig. 18. Simulation result of programs P:Q for property 9.

Hence from the above results it can be inferred that,

|P| + |Q| < |P:Q|

As, (4 + 4) < 9

3 Comparing Stall Estimation Metric with other popular Metrics

Numerious metrics have been proposed by the researchers till date. Among those metrics, Statement count metric, Halstead complexity metric, Cyclomatic complexity metric, Data flow complexity metric and Cognitive complexity metric are the varients explored most by the researchers.

The statement count metric usually considers the count of lines in the program and can be viewed as the measure of the program size [17].

As each operand is associated with a logic. The Halstead complexity metric measures the logic volume of the program by taking in account the count of operators and operands present in the program [4],[17].

In 1976, Thomas J. McCabe proposed a new Cyclomatic complexity metric that takes in account the topological ordering of the program for estimating the software complexity. It considers the count of linearly independent paths in the program as measure for estimating the software complexity [5].

On the other hand data flow complexity metric takes in account the use-definition graph of the program for estimating the software complexity. It considers only those edges in the graph which contributes in data flow between the blocks. The concept is that the program complexity will increase if the variable definition and usage both are in different blocks [18].

Further in cognitive complexity metric the complication in logical composition of the program is studied. The cognitive complexity metric indirectly reflects the efforts required in perceiving the meaning of the program [15],[19].

Now, we will judge all these metrics against nine Weyuker's Properties *(See Table 3)*.

TABLE 3
COMPARING METRICS ON THE SCALE OF NINE WEYUKER'S PROPERTIES

Properties	Statement count metric	Halstead complexity metric	Cyclomatic complexity metric	Data flow complexity metric	Cognitive complexity metric	Stall estimation Metric
1	Yes	Yes	Yes	Yes	Yes	Yes
2	Yes	Yes	*No*	*No*	Yes	Yes
3	Yes	Yes	Yes	Yes	Yes	Yes
4	Yes	Yes	Yes	Yes	Yes	Yes
5	Yes	*No*	Yes	*No*	Yes	Yes
6	*No*	Yes	*No*	Yes	*No*	Yes
7	*No*	*No*	*No*	Yes	Yes	Yes
8	Yes	Yes	Yes	Yes	Yes	Yes
9	*No*	Yes	*No*	Yes	Yes	Yes

4 CONCLUSION

In the current study, we have tested the Stall estimation metric against the nine Weyuker's Properties and found that all of them are satisfied by the proposed metric. This makes the proposed metric as one of the preeminent available software metric that can be used for estimating the software complexity.

REFERENCES

[1] N. Fenton and J. Bieman, *Software metrics: a rigorous and practical approach*. CRC Press, 2014.

[2] H. Zuse, "Software complexity," *NY, USA: Walter de Cruyter*, 1991.

[3] J. Zhao, "On assessing the complexity of software architectures," *Proceedings of the third international workshop on Software architecture*. ACM, 1998. doi:10.1145/288408.288450.

[4] M. H. Halstead, *Elements of software science*. Vol. 7. New York: Elsevier, 1977.

[5] T. J. McCabe, "A complexity measure," *IEEE Transactions on software Engineering* 4: 308-320, 1976. doi:10.1109/TSE.1976.233837.

[6] R. Kazman and M. Burth, "Assessing Architectural Complexity," *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering (CSMR'98)*. IEEE Computer Society, 1998. doi:10.1109/CSMR.1998.665762.

[7] 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.

[8] S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE transactions on Software Engineering* 5: 510-518, 1981. doi:10.1109/TSE.1981.231113.

[9] 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*. Vol. 2. IEEE, 2016. doi:10.1109/INVENTIVE.2016.7824864.

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

[11] E. J. Weyuker, "Evaluating software complexity measures," *IEEE transactions on Software Engineering* 14.9: 1357-1365, 1988. doi:10.1109/32.6178.

[12] S. Misra and I. Akman, "Applicability of weyuker's properties on oo metrics: Some misunderstandings," *Computer Science and Information Systems* 5.1: 17-23, 2008.

[13] D. Mishra, "New Inheritance Complexity Metrics for Object-Oriented Software Systems: An Evaluation with Weyuker's Properties," *Computing and Informatics* 30.2 : 267-293, 2012.

[14] P. Gandhi and P. K. Bhatia, "Analytical analysis of generic reusability: Weyuker's Properties," *IJCSI International Journal of Computer Science Issues* 9.2, 2012.

[15] S. Misra and A. K. Misra, "Evaluating cognitive complexity measure with Weyuker properties," *Cognitive Informatics, 2004. Proceedings of the Third IEEE International Conference on*. IEEE, 2004. doi:10.1109/COGINF.2004.1327464.

[16] D. Beyer and P. Häring, "A formal evaluation of DepDegree based on weyuker's properties," *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 2014. doi:10.1145/2597008.2597794.

[17] S. Yu and S. Zhou, "A survey on metric of software complexity," *Information Management and Engineering (ICIME), 2010 The 2nd IEEE International Conference on*. IEEE, 2010.

[18] E. I. Oviedo, "Control flow, data flow and program complexity," *Software engineering metrics I*. McGraw-Hill, Inc., 1993.

[19] S. Misra, "Validating modified cognitive complexity measure," *ACM SIGSOFT Software Engineering Notes* 32.3: 1-5, 2007. doi:10.1145/1241572.1241583.