Assignment 9

Assignment 9 - Objectives:

- Pipeline limitation: Hazards
- Loop unrolling optimization
- Microbenchmarking

For each of our team-based assignments, you can either:

- Create a group with the same partner,
- Change your group (select a different partner), or
- Decide to work on you own.

Group of two (2):

- You can do Assignment 9 in a group of two (2) or you can work on your own.
- If you choose to work in a group of two (2):
  - Crowdmark will allow you to select your teammate (1). Both of you will only need to submit one assignment and when the TAs mark this one assignment, the marks will be distributed to both of you automatically.
  - You can always work with a student from the other section, but both of you will need to submit your assignment separately, i.e., Crowdmark will not consider the both of you as a group.

Requirements for this Assignment 9:

- Always show your work (as illustrated in lectures), if appropriate, and
- Make sure the pdf/jpeg/png documents you upload are of good quality, i.e., easy to read, therefore easy to mark! :) If the TA cannot read your work, the TA cannot mark your work and you will get 0. :(

Marking:

- All questions of this assignment will be marked for correctness.
- The amount of marks for each question is indicated as part of the question.
- A solution will be posted on Monday after the due date.

Deadline:

- Friday April 7 at 23:59:59 on Crowdmark.
- Late assignments will receive a grade of 0, but they will be marked (if they are submitted before the solutions are posted on Monday) in order to provide feedback to the student.

Enjoy!

Q1 (11 points)

Pipeline limitation: Hazards

Consider the following C code fragment:

```
a = b * e;
c = b + f;
```

Below is the corresponding MIPS assembly code. We are assuming that all C variables are in the stack segment of the memory and that they are all accessible as offsets (displacements) from $sp:

```
lw $t1, 0($sp) # Loading b from the stack
lw $t2, 4($sp) # Loading e from the stack
mul $t3, $t1,$t2
sw $t3, 12($sp) # Storing a onto the stack
lw $t4, 8($sp) # Loading f from the stack
add $t5, $t1,$t4
sw $t5, 16($sp) # Storing c onto the stack
```

1. The MIPS assembly code above, as it is, does not produce the same result as the C code fragment does. In order to understand why this is and to better visualize what is happening when the above MIPS assembly instructions are executed in each of the 5 stages: fetch (F), decode (D), execute (E), memory (M) and write back (W) of a simple in-order pipelined execution microprocessor, construct the pipelined execution diagram illustrating the pipelined execution of the machine code instructions corresponding to each of these MIPS assembly instructions above. To do so, use the MIPS assembly instructions above as they are, in the order in which they are presented above.

Important: In creating your pipelined execution diagram, follow the diagrams found on Slides 6, 7, 8, 13, 14, 15 of our Lecture 31, i.e., diagrams in which we show the assembly instructions divided into micro-operations and list the latter in each of the five (5) stages in which they occur: fetch (F), decode (D), execute (E), memory (M) and write back (W) along with the pipeline registers used (valA, valB, valI, valE, valM).

Solution: Solution for 1. Is incorporated in the solution for 2 below.

2. Once you have constructed your pipelined execution diagram, locate the hazards in the illustrated MIPS machine instructions, more specifically, in their micro-operations, and identify which of these hazards are data hazards and/or control hazards?

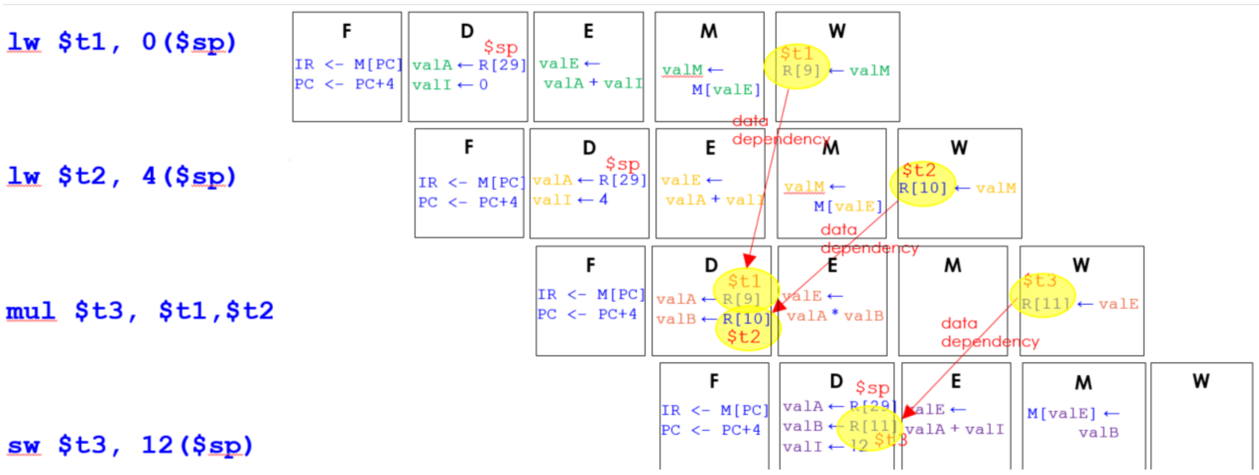
   Note: We shall not concern ourselves with structural hazards since MIPS uses two different memories (same as x86-64 ISA):

   - one for the instructions and
   - one for the data.

   So, in MIPS, structural hazards do not create a problem.

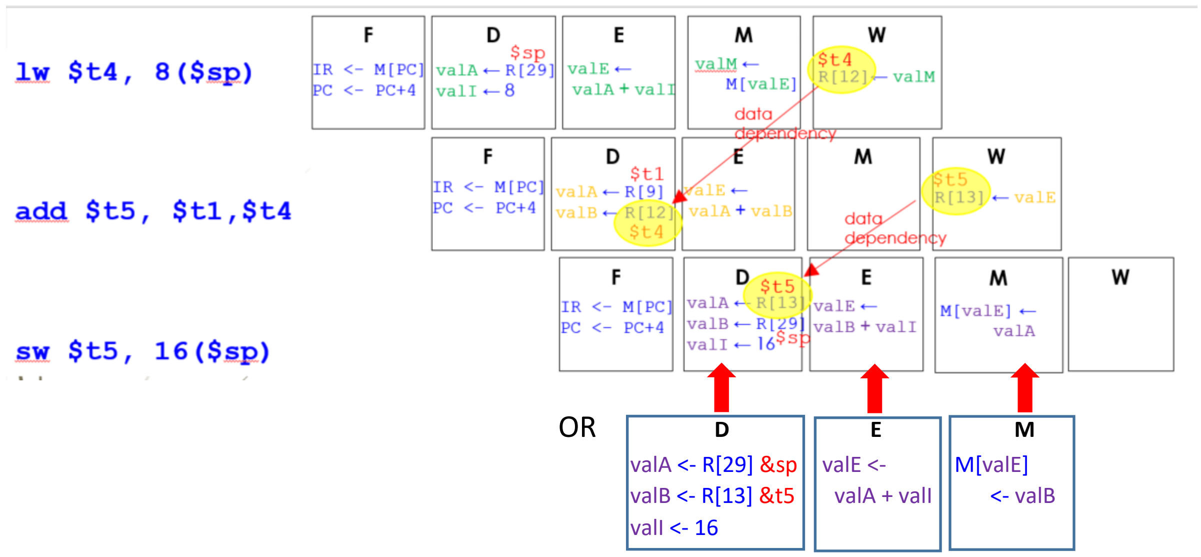
Solution: Below is the **Pipelined Execution Diagram** for 1. and 2. solutions

Data Hazard – There are 5 data dependencies in the given code:



```
1. lw $t1, 0($sp)
2. lw $t2, 4($sp)
3. mul $t3, $t1, $t2
4. sw $t3, 12($sp)
```

1. Data hazard (creating data dependency) between W stage of instruction 1 (`lw $t1, 0($sp)`) and D stage of instruction 3 (`mul $t3, $t1, $t2`)
2. Data hazard (creating data dependency) between W stage of instruction 2 (`lw $t2, 4($sp)`) and D stage of instruction 3 (`mul $t3, $t1, $t2`)
3. Data hazard (creating data dependency) between W stage of instruction 3 (`mul $t3, $t1, $t2`) and D stage of instruction 4 (`sw $t3, 12($sp)`)



```
5. lw $t4, 8($sp)
6. add $t5, $t1, $t4
7. sw $t5, 16($sp)
```

4. Data hazard (creating data dependency) between W stage of instruction 5 (`lw $t4, 8($sp)`) and D stage of instruction 6 (`add $t5, $t1, $t4`)
5. Data hazard (creating data dependency) between W stage of instruction 6 (`add $t5, $t1, $t4`) and D stage of instruction 7 (`sw $t5, 16($sp)`)

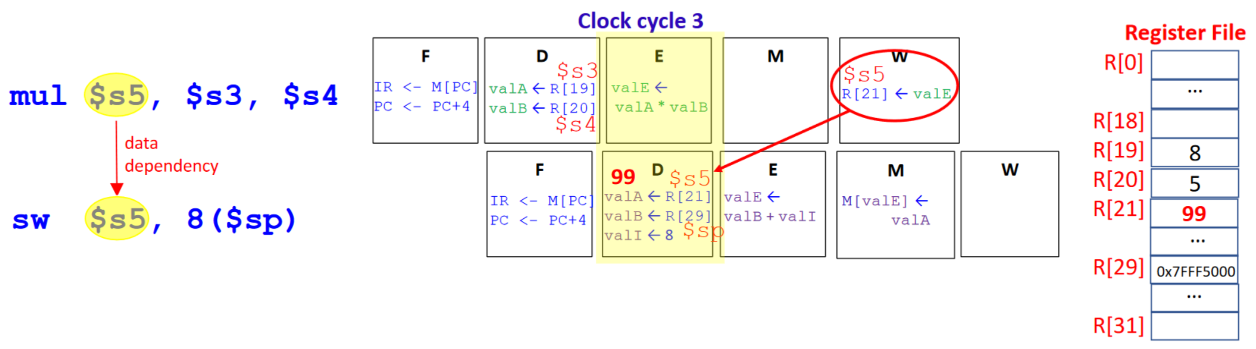
Control Hazard

Since there are no *branching* instructions (conditional jumps) in our MIPS assembly code, there are no control hazard (potential problem created by the microprocessor's mis-predictions).

3. Microprocessors fix hazards by using several techniques such as
   - stalling instruction execution by injecting bubbles (i.e., nop instructions) between the instructions that are dependent upon each other,
   - forwarding data,
   - a mix of both (called **load interlock**), and lastly
   - reordering instructions. We did not present this technique in our lectures. It is a very simple technique in which one avoid hazards by reordering instructions. Click here for an example.

Here is what you see when you «click here»:

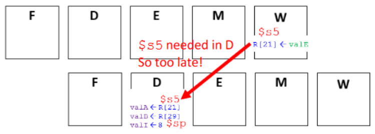
Data Hazard leading to data dependency



Problem: At Stage D of *sw* (Stage E of *mul*), *sw* needs value of $s5. But *mul* stores value it has computed into $s5 only at its Stage W (Stage M of *sw*), i.e., 2 “ticks” later! **=> Too late!!!**

Solving data dependency using interleaving

```
mul  $s5, $s3, $s4
sw   $s5, 8($sp)
```

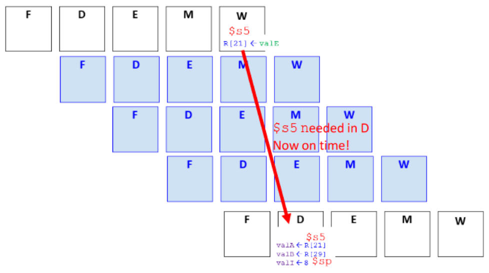


If the original program is:

```
lw   $s0, 0($sp)
lw   $s1, 4($sp)
...
add  $s2, $s6, $s7
...
mul  $s5, $s3, $s4
sw   $s5, 8($sp)
...
```

I can reorder the instructions, i.e., I can interleave independent instructions between **mul** and **sw**, hence solving their data dependency:

```
mul  $s5, $s3, $s4
lw   $s0, 0($sp)
lw   $s1, 4($sp)
add  $s2, $s6, $s7
sw   $s5, 8($sp)
```



Fix the hazard(s) you have discovered by using the reordering instruction (interleaving) technique along with the data forwarding technique. In answering this question, you are not to use the stalling technique nor the load interlock.

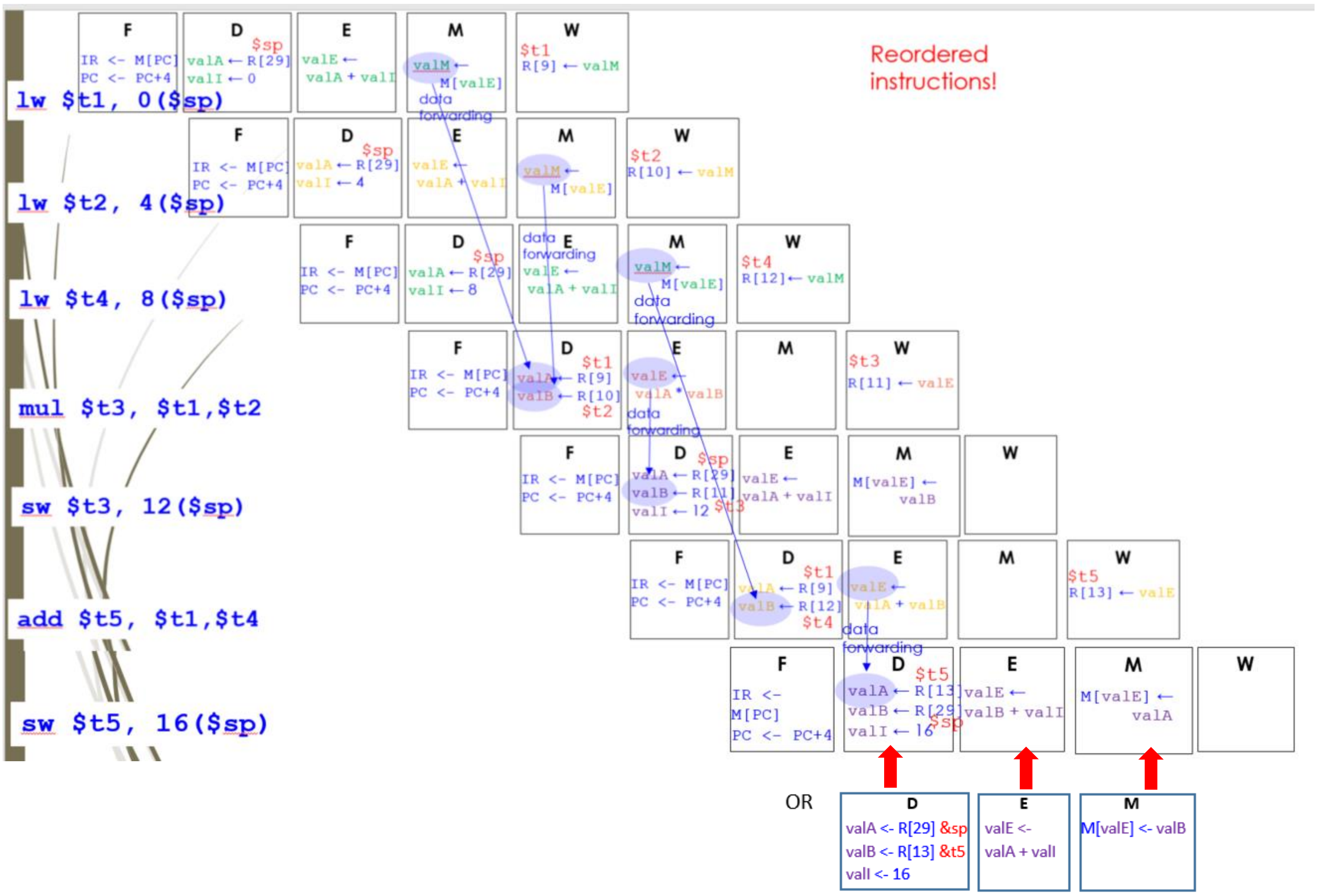
As part of your answer, include

- your reordered MIPS assembly instructions (MIPS assembly code) which now should produce the same result as the C code fragment above,
- the corresponding pipelined execution diagram, which must illustrate the new order of your instructions as well as the data being forwarded, and
- clearly indicate on your new pipelined execution diagram, which data is being forwarded and from which stage to which other stage.

Possible solution:

New order:

```
1. lw $t1, 0($sp)
2. lw $t2, 4($sp)
3. lw $t4, 8($sp)
4. mul $t3, $t1,$t2
5. sw $t3, 12($sp)
6. add $t5, $t1,$t4
7. sw $t5, 16($sp)
```



Q2 (6 points)

Loop unrolling optimization

a. Below, write a version of the inner product C function described in Homework Problem 5.13 (at page 570 in our textbook) that uses 6 x 1 loop unrolling. Make sure you comment your function. You do not have to answer the questions A., B., C. and D. of this Homework Problem 5.13 in the textbook.

Possible solution:

```
/* 6x1unrolling
 * where k = 6 -> dealing with six (6) elements of arrays u and v
 * at each iteration of the loop
 * where c = 1 -> only one (1) accumulator at each iteration
 * of the loop, i.e., only one (1) running sum (variable in
 * which to "accumulate" the running sum).
 */

void inner6x1unrolling(vec_ptr u, vec_ptr v, data_t *dest) {
   long k = 6;
   long i;
   long length = vec_length(u);
   data_t *udata = get_vec_start(u);
   data_t *vdata = get_vec_start(v);

   // Initialize 1 accumulator
   data_t sum = (data_t) 0;

   // Compute number of iterations considering i += k (6)
   long limit = length - k + 1;

   // Dealing with k (6) elements at a time per iteration
   for (i = 0; i < limit; i += k){
      sum = sum + udata[i]   * vdata[i]
                + udata[i+1] * vdata[i+1]
                + udata[i+2] * vdata[i+2]
                + udata[i+3] * vdata[i+3]
                + udata[i+4] * vdata[i+4]
                + udata[i+5] * vdata[i+5];
   }

   // For cases when length < k OR length not a multiple of k
   for (; i < length; i++) {
      sum = sum + udata[i] * vdata[i];
   }

   *dest = sum;

   return;
}
```

b. Below, write a second version of this inner product function described in Homework Problem 5.13 that uses 6 x 6 loop unrolling. Make sure you comment your function. Again, you do not have to answer the questions A., B., C. and D. of this Homework Problem 5.13 in the textbook.

Possible solution:

```
/* 6x6unrolling
 * where k = 6 -> dealing with six (6) elements of arrays u and v
 * at each iteration of the loop
 * where c = 6 -> using six (6) accumulators at each iteration
 * of the loop, i.e., six (6) running sums (variables in
 * which to “accumulate” the six running sums.
 */

void inner6x6unrolling(vec_ptr u, vec_ptr v, data_t *dest) {
   long k = 6;
   long i;
   long length = vec_length(u);
   data_t *udata = get_vec_start(u);
   data_t *vdata = get_vec_start(v);

   // Initialize all 6 accumulators
   data_t sum0 = (data_t) 0;
   data_t sum1 = (data_t) 0;
   data_t sum2 = (data_t) 0;
   data_t sum3 = (data_t) 0;
   data_t sum4 = (data_t) 0;
   data_t sum5 = (data_t) 0;

   // Compute number of iterations considering i += k (6)
   long limit = length - k + 1;

   // Dealing with k (6) elements at a time per iteration
   for (i = 0; i < limit; i += k) {
      sum0 = sum0 + udata[i]   * vdata[i];
      sum1 = sum1 + udata[i+1] * vdata[i+1];
      sum2 = sum2 + udata[i+2] * vdata[i+2];
      sum3 = sum3 + udata[i+3] * vdata[i+3];
      sum4 = sum4 + udata[i+4] * vdata[i+4];
      sum5 = sum5 + udata[i+5] * vdata[i+5];
   }
```

```
    // For cases when length < k OR length not a multiple of k
    for (; i < length; i++) {
      sum0 = sum0 + udata[i] * vdata[i];
    }

    // Total up all accumulators
    *dest = (sum0 + sum1) + (sum2 + sum3) + (sum4 + sum5);
    return;
  }
```

Q3 (5 points)

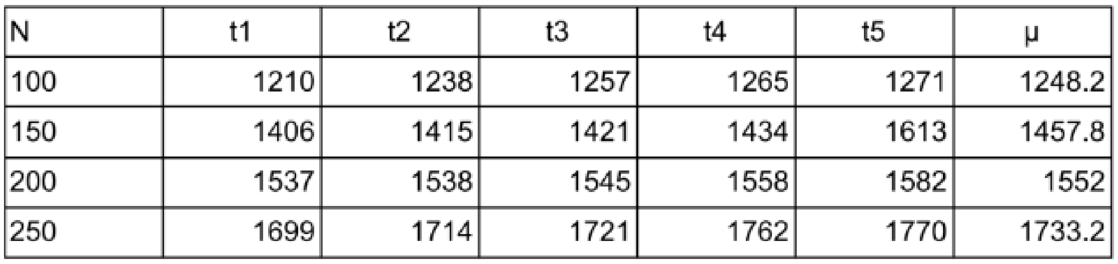
Benchmark data sets and graph from our Lab 7 data

In Lab 7, you are asked to microbenchmark code and to graph your results.

In this question, upload the completed data sheet you built in Lab 7 as well as the graph you produced.

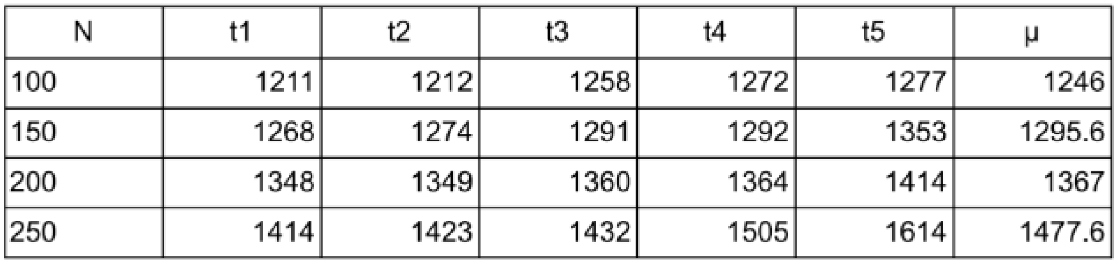
Make sure you graph all data sets onto the same graph and label each resulting line. This will ease comparison and allow you to promptly reach a conclusion.

Version 1 - Branch:



N	t1	t2	t3	t4	t5	µ
100	1210	1238	1257	1265	1271	1248.2
150	1406	1415	1421	1434	1613	1457.8
200	1537	1538	1545	1558	1582	1552
250	1699	1714	1721	1762	1770	1733.2

Version 2 - Conditional Move:

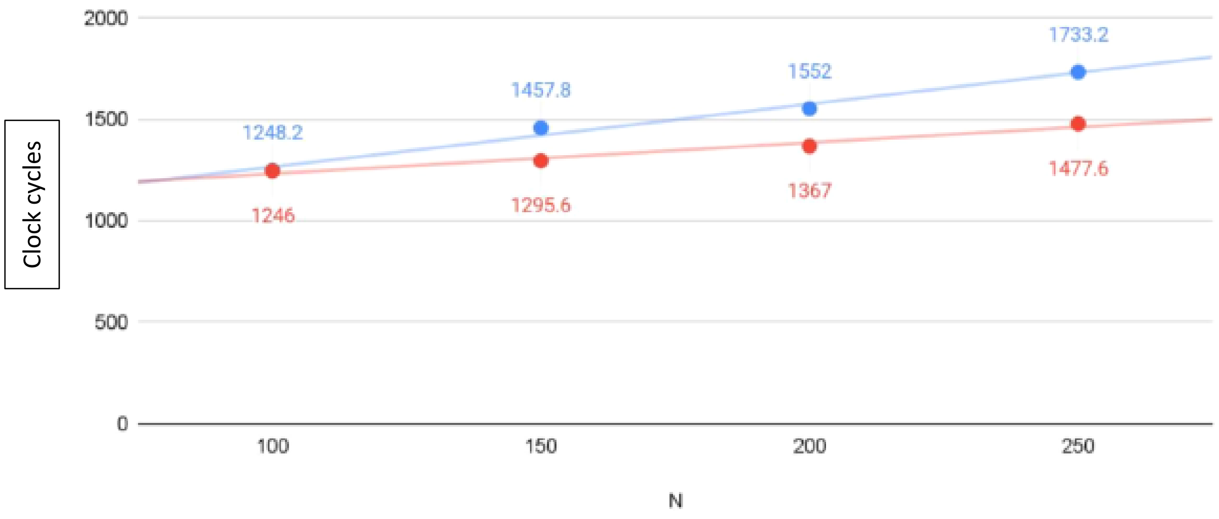


N	t1	t2	t3	t4	t5	µ
100	1211	1212	1258	1272	1277	1246
150	1268	1274	1291	1292	1353	1295.6
200	1348	1349	1360	1364	1414	1367
250	1414	1423	1432	1505	1614	1477.6

Slope equation = $\frac{y_2 - y_1}{x_2 - x_1}$

Slope of Branch: $\frac{1733.2 - 1248.2 \text{ clock cycles}}{250 - 100 \text{ elements}} = \frac{485 \text{ clock cycles}}{150 \text{ elements}} = 3.23$ CPE

Slope of Conditional Move: $\frac{1477.6 - 1246 \text{ clock cycles}}{250 - 100 \text{ elements}} = \frac{231.6 \text{ clock cycles}}{150 \text{ elements}} = 1.544$ CPE



My Observation (looking at the graph and the computed CPE figures above):

The version of sumPlus(…) that uses the "jle" conditional jump instruction, i.e., that branches, requires more clock cycles per element to execute than the version of sumPlus(…) that uses the "cmovl "conditional move instruction: 3.23 CPE versus 1.544 CPE.

My Conclusion:

The version of sumPlus(…) that uses the "cmovl "conditional move instruction executes faster than the version of sumPlus(…) that uses the "jle" conditional jump instruction.