

## Computer Architecture II (CSU34021) Tutorial 3 Report

### Question 1

We are given the following C/C++ code to translate into RISC-I assembly language.

```
inp_int = 4
int min (int a, int b, int c)
{
    int v = a;
    if (b < v)
        v = b;
    if (c < v)
        v = c;
    return v;
}

int min5(int i, int j, int k, int c) {
    return min (min (inp_int, i, j), k, 1);
}
```

We are also told that we can try to make unoptimized code first with NOPS (An instruction that does nothing) in the delay slots, and then optimize. We must follow the following assumptions:

- **r1: Register for return arguments**
- **r15/r31: Register for return address**
- **Following conditional jump instructions available:**
  - **jeq: Jump if equal**
  - **jne: Jump if not equal**
  - **jge: Jump if greater than or equal**
  - **jg: Jump if greater than**
  - **jle : Jump if less than or equal**
  - **jl : Jump if less than**

Let's first work on the global variable and the function min. I was able to come up with some general comments as I first went through the C/C++ code.

```
inp_int = 4 //Global variables can be anything from R0-R9
            //so we can use any of those here. R0 is fixed to 0
            //and R1 is the register for return arguments. So,
            //R2 can be used as a stack pointer, so use R3!
int min (int a, int b, int c) //R26=a, R27=b, R28=c
{ //We use v below for the return value, so it should be R1!
    int v = a;
    if (b < v) //Use sub with {C} to set flags & jump instruction
        v = b;    //Probably need a NOP here after the jump
    if (c < v) //Use sub with {C} to set flags & jump instruction
        v = c;    //Probably need a NOP here after the jump
    return v; //We are told the return address is R31 (R15 for
            //any functions calling min)
}
```

Then I came up with my **un-optimized** version of the function with NOPS.

```

add r0, #4, r3           ;inp_int is a global variable, in R3

min:
add r26, r0, r1          ; R1 (v) = R26 (a) + R0 (0)
sub r27, r1, r0 {C}      ; if(b < v) condition with flags set.
jge check1              ; jump instruction & NOP after since
xor r0, r0, r0           ; the instruction right after will be
                        ; executed after the jump is fetched
                        ; but before the jump is executed.

add r27, r0, r1          ; R1 (v) = R27 (b) + R0 (0)
check1:
sub r28, r1, r0 {C}      ; if(c < v) condition with flags set.
jge check2              ; jump instruction & NOP after since
xor r0, r0, r0           ; the jump instruction takes place
                        ; after the next normal non NOP
                        ; instruction, by the time we fetch
                        ; the instruction there where we have
                        ; to jump, the next instruction would
                        ; have already been executed.

add r28, r0, r1          ; R1 (v) = R28 (c) + R0 (0)
check2:
ret (r31)0              ;R31 is used for the return address
xor r0, r0, r0           ;NOP after ret or callr instruction

```

Then I tried to optimize my code, but I was aware that it is not possible to remove any of the NOPS and execute other useful instructions in those delay slots to optimize it. Doing so would have messed up the execution of the jump instruction jge by executing the instruction after it too early, and the same with the return instruction ret. **Hence, this was actually already the most optimized solution.**

Then, I started working on the min5 function, making comments on the C/C++ code to begin my thought process.

```

int min5(int i, int j, int k, int l) {
//R26=i, R27=j, R28=k, R29=l
return min (min (inp_int, i, j), k, l);
//For the return address, it's R15/R31,
//Therefore we would do callr r15, min
//and ret r31, 0. Also remember R10-R15
//are for the parameters for the the
//next function.
}

```

I then started making my **un-optimized** versions of my RISC-I translation of the function min5 (Called min5UO).

```

min5UO:
; Min function call 1:
add r3, r0, r10      ; R10 (param 1) = R3 (inp_int) + R0 (0)
add r26, r0, r11     ; R11 (param 2) = R26 (i) + R0 (0)
add r27, r0, r12     ; R12 (param 3) = R27 (j) + R0 (0)
callr r15, min       ; Return address saved in R15 & call min
xor r0, r0, r0       ; NOP after callr

; Min function call 2:
add r1, r0, r10      ; R10 (param 1) = R1 (call 1 result) + R0 (0)
add r28, r0, r11     ; R11 (param 2) = R28 (k) + R0 (0)
add r29, r0, r12     ; R12 (param 3) = R29 (l) + R0 (0)
callr r15, min       ; Return address saved in R15 & call min
xor r0, r0, r0       ; NOP after callr

; Return with R1 containing the result and address in R31.
ret (r31)0           ; R31 is used for the return address.
xor r0, r0, r0       ; NOP after ret

```

I quickly realized that I could optimize my solution by removing the NOP instructions and adding useful instructions here unlike the function min. I called this new attempt min5 instead of min5UO which was for the un-optimized version.

```

min5:
; Min function call 1:
add r3, r0, r10      ; R10 (param 1) = R3 (inp_int) + R0 (0)
add r26, r0, r11     ; R11 (param 2) = R26 (i) + R0 (0)
callr r15, min       ; Return address saved in R15 & call min
add r27, r0, r12     ; R12 (param 3) = R27 (j) + R0 (0)

; Min function call 2:
add r1, r0, r10      ; R10 (param 1) = R1 (call 1 result) + R0 (0)
add r28, r0, r11     ; R11 (param 2) = R28 (k) + R0 (0)
callr r15, min       ; Return address saved in R15 & call min
add r29, r0, r12     ; R12 (param 3) = R29 (l) + R0 (0)

; Return with R1 containing the result and address in R31.
ret (r31)0           ; R31 is used for the return address.
xor r0, r0, r0       ; NOP after ret

```

By optimizing my code and removing the NOP instructions, my code will execute in less cycles and will reduce the length of the code itself by simply rearranging the order of certain lines of code.

## Question 2

We are given the following code.

```
int compute_pascal(int row, int position)
{
    if(position == 1)
    {
        return 1;
    }
    else if(position == row)
    {
        return 1;
    }
    else
    {
        return compute_pascal(row-1, position)+compute_pascal(row-1, position-1);
    }
}
```

By instrumenting the code above, we are meant to find the number of procedure calls, maximum register window depth, the number of register window overflows and the number of register window underflows that would occur during the calculation of compute pascal(30,20) given a RISC-I processor with 6, 8 and 16 overlapping register windows. Also we must calculate the number of register windows pushed onto the stack for each case of overlapping register windows. There are a few assumptions we must follow:

- Overflow only occurs when all register windows have been utilized.
- Underflow occurs when less than two register windows are left active. In other words, the minimum number of active register windows is two.

At the end, we must modify our code with the assumption that overflow occurs when there is one empty register window and compare the two approaches.

For instrumenting the code, I was planning on using Java 8 as I was most comfortable with it, and we are told that we can use any compiler/language we are comfortable with (mentioned at 19:41 in the video CSU34021\_Tutorial3\_Intro by Dr. Syed Asad Alam).

I knew that I must initialize all variables I will be using at the start which will help me keep count of information such as the procedure calls, maximum register window depth, and more. Most of these are all 0 initially, except WUSED which is 2, since there are always 2 valid register windows in the register file (mentioned in page 17/18 in Session2\_handout.pdf).

```
public static void initializeVariables(){
    WUSED = 2; //We always have 2 register windows.
    NWINDOWS = 0;
    CWP = 0;
    SWP = 0;
    procedureCalls = 0;
    numberOfOverflows = 0;
    numberOfUnderflows = 0;
    registerWindowDepth = 0;
    maxRegisterWindowDepth = 0;
}
```

I then thought about the places where we have to check underflow and overflow. I ended up designing my code so that my overflow checking function can be called before a function call (which must test if a new maximum register window depth has been reached, and if an overflow has occurred meaning WUSED is equal to NWINDOWS i.e. all register windows are being utilized) hence I combined both of those checks in the one checkForOverflow function, incrementing both CWP and SWP at the appropriate times as discussed in the lectures.

```
public static void checkForOverflow(){
    registerWindowDepth++;
    if(registerWindowDepth > maxRegisterWindowDepth) {
        maxRegisterWindowDepth = registerWindowDepth;
    }
    if(WUSED == NWINDOWS){
        numberOfOverflows++;
        SWP++;
    }
    else {
        WUSED++;
    }
    CWP++;
}
```

I then designed my underflow checking function to be called after a return statement, meaning the register window depth can be decremented, and we can check for underflow meaning WUSED is equal to 2 i.e. when less than two register windows are starting to be left active. I also decremented SWP and CWP at the appropriate times as discussed in the lectures.

```
public static void checkforUnderflow(){
    registerWindowDepth--;
    if(WUSED == 2){
        SWP--;
        numberOfUnderflows++;
    }
    else {
        WUSED--;
    }
    CWP--;
}
```

Then, in my `compute_pascal_instrumented` function, I slotted in function calls to the above functions at the correct times (before function calls and before return statements appropriately), making sure to increment the `procedureCalls` variable at the start to ensure it is updated accordingly.

```
public static int compute_pascal_instrumented(int row, int position) {
    procedureCalls++;
    if(position == 1) {
        checkforUnderflow();
        return 1;
    }
    else if(position == row) {
        checkforUnderflow();
        return 1;
    }
    else {
        checkForOverflow();
        int result1 = compute_pascal_instrumented(row-1, position);
        checkForOverflow();
        int result2 = compute_pascal_instrumented(row-1, position-1);
        checkforUnderflow();
        return result1+result2;
    }
}
```

I knew that I couldn't just call the function from the `main()`, as I would need to do some setup before calling the function. Hence, I made another function to do the setup for each function call for me, to avoid repeated code. We were asked to calculate the number of register windows pushed onto the stack for each case of overlapping register windows, but I realized that this would be the same value as the `numberOfOverflows` by definition. This is because an overflow occurs when a register window is pushed onto the stack because of overlapping register windows.

```
public static void call_compute_pascal_instrumented(int a, int b, int numberOfWindows){
    try {
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%");
        initializeVariables();
        NWINDOWS = numberOfWindows;
        checkForOverflow();
        int result = compute_pascal_instrumented(a, b);
        System.out.println("Using a RISC-I processor with " + numberOfWindows + " register windows");
        System.out.println("The result from compute_pascal(" + a + "," + b + ") is " + result);
        System.out.println("The number of procedure calls is " + procedureCalls);
        System.out.println("The maximum register window depth is " + maxRegisterWindowDepth);
        System.out.println("The number of register window Overflows is " + numberOfOverflows);
        System.out.println("The number of register window Underflows is " + numberOfUnderflows);
        System.out.println("The number of register windows pushed onto the stack for each case of overlapping register windows is " + numberOfOverflows);
        System.out.println("%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%\n");
    } catch(Exception e) {
        e.printStackTrace();
    }
}
```

Here is the screenshot of the output:

```

Tutorial3.java Tutorial3Modified.java
3
4 public class Tutorial3 {
5
6     public static int procedureCalls = 0;
7     public static int registerWindowDepth = 0;
8     public static int M5ED = 0;
9     public static int numberOfOverflows = 0;
10    public static int numberOfUnderflows = 0;
11    public static int CWP = 0;
12    public static int SWP = 0;
13    public static int MWINDOW = 0;
14    public static int maxRegisterWindowDepth = 0;
15
16    public static void main(String args[]) {
17        call_compute_pascal_instrumented(30,20, 6);
18        call_compute_pascal_instrumented(30,20, 8);
19        call_compute_pascal_instrumented(30,20, 16);
20    }
21 }

-terminated- Tutorial3 (1) [Java Application] /Library/Java/JavaVirtualMachines/adoptopenjdk-8-jdk/Contents/Home/bin/java (29 Nov 2021, 14:23:46 - 14:23:47)
=====
Using a RISC-I processor with 6 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 7051109
The number of register window Underflows is 7051109
The number of register windows pushed onto the stack for each case of overlapping register windows is 7051109
=====
Using a RISC-I processor with 8 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 2840177
The number of register window Underflows is 2840177
The number of register windows pushed onto the stack for each case of overlapping register windows is 2840177
=====
Using a RISC-I processor with 16 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 30826
The number of register window Underflows is 30826
The number of register windows pushed onto the stack for each case of overlapping register windows is 30826
=====

```

In text form, the results I got were this:

```

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Using a RISC-I processor with 6 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 7051109
The number of register window Underflows is 7051109
The number of register windows pushed onto the stack for each case of overlapping
register windows is 7051109
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Using a RISC-I processor with 8 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 2840177
The number of register window Underflows is 2840177
The number of register windows pushed onto the stack for each case of overlapping
register windows is 2840177
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Using a RISC-I processor with 16 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 30826
The number of register window Underflows is 30826
The number of register windows pushed onto the stack for each case of overlapping
register windows is 30826
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```

I have also attached all my java files that led me to these results with this submission.



We are then asked to modify our code with the assumption that overflow occurs when there is one empty register window and compare the two approaches.

I quickly realized that for this to happen, I only needed to change one line in the overflow checking function. When there is one empty register window, that means WUSED is equal to NWINDOWS-1. An example would be, if NWINDOWS=6, and WUSED=5, that means there is one empty register window exactly, and hence  $5 = 6 - 1$ .

```
public static void checkForOverflowModified(){
    registerWindowDepth++;
    if(registerWindowDepth > maxRegisterWindowDepth) {
        maxRegisterWindowDepth = registerWindowDepth;
    }
    if(WUSED == NWINDOWS-1){
        numberOfOverflows++;
        SWP++;
    }
    else {
        WUSED++;
    }
    CWP++;
}
```

When I used this modified code, here is the screenshot of the output:

```

Tutorial3.java  Tutorial3Modified.java  24
4  public class Tutorial3Modified {
5
6      public static int procedureCalls = 0;
7      public static int registerWindowDepth = 0;
8      public static int WUSED = 0;
9      public static int numberOfOverflows = 0;
10     public static int numberOfUnderflows = 0;
11     public static int CWP = 0;
12     public static int SWP = 0;
13     public static int NWINDOWS = 0;
14     public static int maxRegisterWindowDepth = 0;
15
16     public static void main(String args[]) {
17         call_compute_pascal_instrumented(30,20, 6);
18         call_compute_pascal_instrumented(30,20, 8);
19         call_compute_pascal_instrumented(30,20, 16);
20     }
21 }

Console 25
<terminated> Tutorial3Modified [Java Application] [Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java (29 Nov 2021, 14:29:07 - 14:29:08)
=====
Using a RISC-I processor with 6 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 10656359
The number of register window Underflows is 10656359
The number of register windows pushed onto the stack for each case of overlapping register windows is 10656359
=====
Using a RISC-I processor with 8 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 4527434
The number of register window Underflows is 4527434
The number of register windows pushed onto the stack for each case of overlapping register windows is 4527434
=====
Using a RISC-I processor with 16 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 58650
The number of register window Underflows is 58650
The number of register windows pushed onto the stack for each case of overlapping register windows is 58650
=====

```



In text form, the modified code results I got were this:

```

Using a RISC-I processor with 6 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 10656359
The number of register window Underflows is 10656359
The number of register windows pushed onto the stack for each case of overlapping
register windows is 10656359

Using a RISC-I processor with 8 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 4527434
The number of register window Underflows is 4527434
The number of register windows pushed onto the stack for each case of overlapping
register windows is 4527434

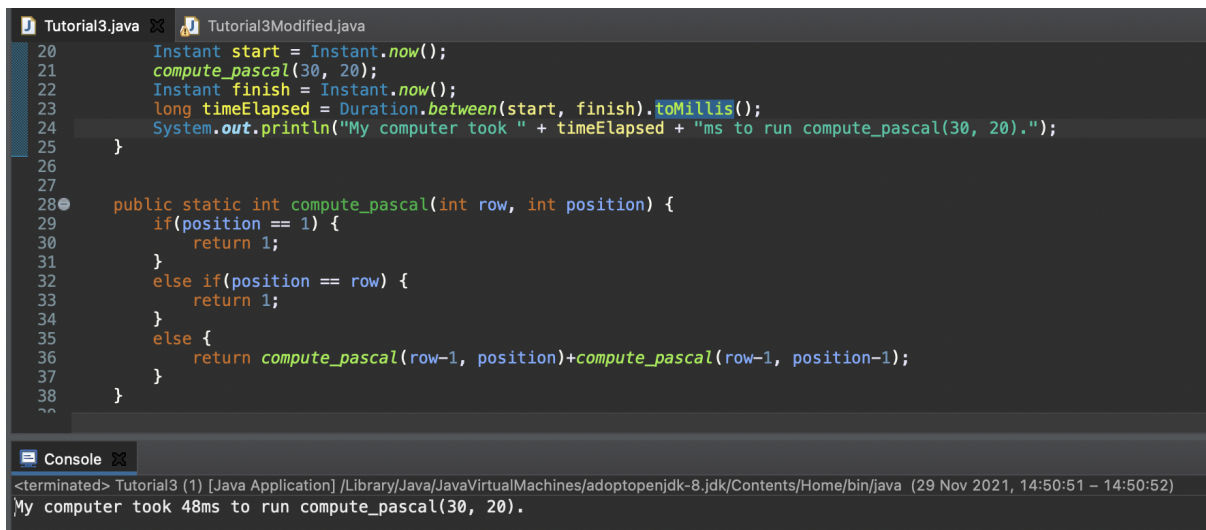
Using a RISC-I processor with 16 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 58650
The number of register window Underflows is 58650
The number of register windows pushed onto the stack for each case of overlapping
register windows is 58650

```

By comparing both approaches, I can see that my original approach had the fact that overflow only occurs when all register windows have been utilized, meaning  $WUSED = NWINDOWS$ . In this modified approach, it followed the fact that overflow occurs when there is one empty register window. As we can see in the results, **this causes an increase in the number of overflows AND underflows**. This seems to be because we were comparing  $WUSED = NWINDOWS$  but now we are comparing  $WUSED = NWINDOWS - 1$  hence there is more opportunity for code to break into the if-condition here, and therefore increasing the number of overflows and underflows. The number of procedure calls and maximum register depth is the same in both versions, and that is expected, since this modification of the assumption of when overflow occurs should not, and will not affect the number of procedure calls and the maximum register depth.

### Question 3

I ran the non-instrumented, release version of the compute\_pascal function. I ran this from the main, and this was the result in a screenshot.



The screenshot shows an IDE with two tabs: 'Tutorial3.java' and 'Tutorial3Modified.java'. The code in 'Tutorial3Modified.java' is as follows:

```
20 Instant start = Instant.now();
21 compute_pascal(30, 20);
22 Instant finish = Instant.now();
23 long timeElapsed = Duration.between(start, finish).toMillis();
24 System.out.println("My computer took " + timeElapsed + "ms to run compute_pascal(30, 20).");
25 }
26
27
28 public static int compute_pascal(int row, int position) {
29     if(position == 1) {
30         return 1;
31     }
32     else if(position == row) {
33         return 1;
34     }
35     else {
36         return compute_pascal(row-1, position)+compute_pascal(row-1, position-1);
37     }
38 }
```

The console output at the bottom shows:

```
<terminated> Tutorial3 (1) [Java Application] /Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java (29 Nov 2021, 14:50:51 - 14:50:52)
My computer took 48ms to run compute_pascal(30, 20).
```

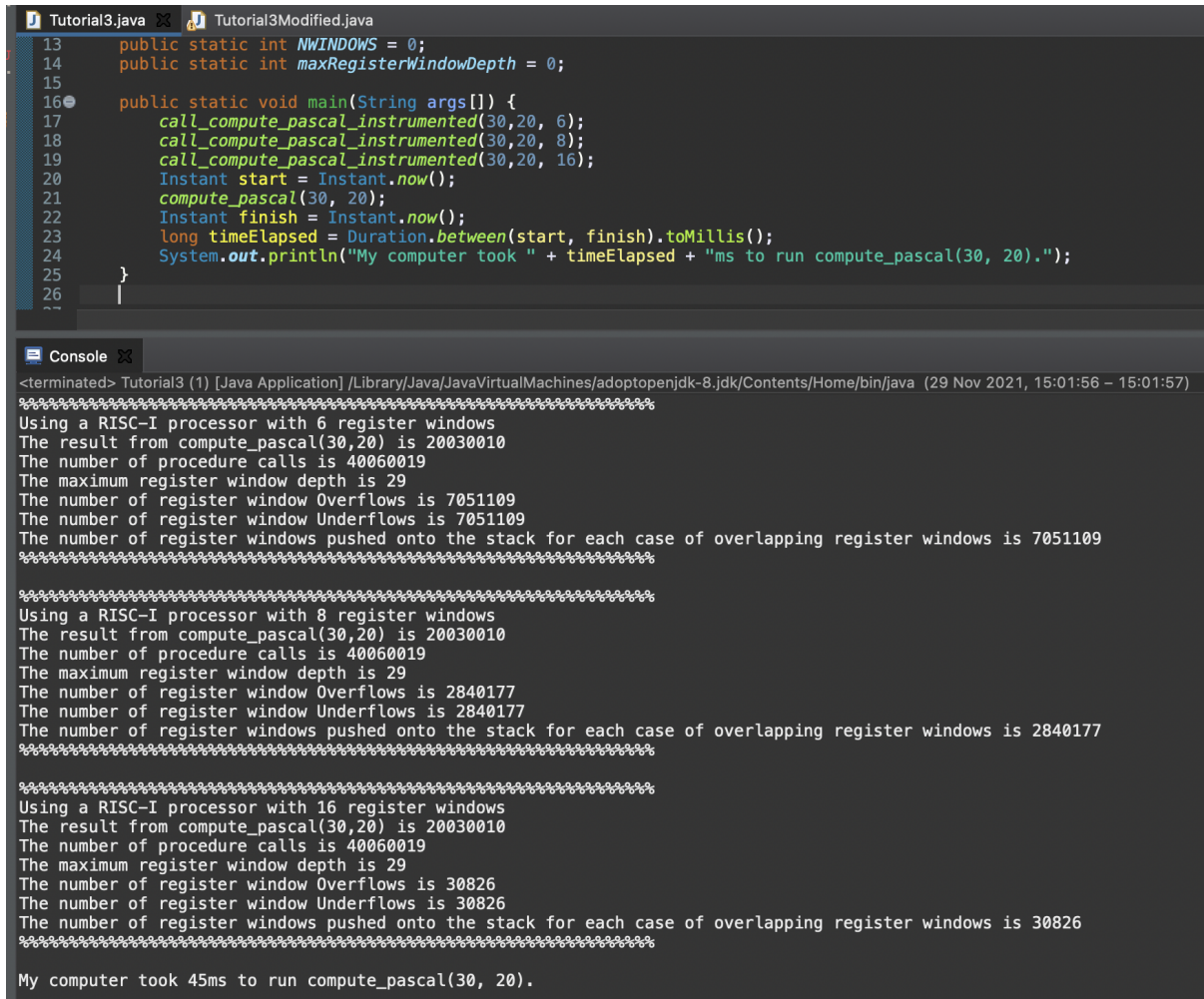
As seen above, my computer took generally 48ms to run the compute\_pascal(30,20). The main approach we need to calculate the time elapsed while our function is running is this:

```
getCurrentTimeInitially(); //get current time
compute_pascal(30,20);
getCurrentTimeFinally(); //get current time
System.out.println("Time taken: " + getCurrentTimeFinally - getCurrentTimeInitially);
```

In Java 8, we can use the Instant class to get the current time. By doing Instant.now(), we can store the current time. After storing the current time, we can get the time elapsed by using the Duration class in Java 8 to get the time difference between the first and second times. We can use the function .toMillis() to convert this to milliseconds for better understanding.

The accuracy of the code is pretty good. I think the time elapsed would have been quicker if this was on C/C++, but with Java it was evident that it was still better to get the elapsed time with the Instant and Duration class, instead of using System.currentTimeMillis() which is known for being inaccurate. I could have also used System.nanoTime(), but that does not have the same accuracy as using the Instant and Duration class which are both **immutable, thread-safe and use their own time-scale which is the Java Time-Scale**. This makes the approach I took to determine how long it takes to compute pascal(30, 20) on my computer a lot more accurate than other methods, **so I think the accuracy for timing the release code in Java is pretty good.**

Here is a screenshot containing my original Q2 code running with the Q3 code:



```
Tutorial3.java Tutorial3Modified.java
13 public static int NWINDOWS = 0;
14 public static int maxRegisterWindowDepth = 0;
15
16 public static void main(String args[]) {
17     call_compute_pascal_instrumented(30,20, 6);
18     call_compute_pascal_instrumented(30,20, 8);
19     call_compute_pascal_instrumented(30,20, 16);
20     Instant start = Instant.now();
21     compute_pascal(30, 20);
22     Instant finish = Instant.now();
23     long timeElapsed = Duration.between(start, finish).toMillis();
24     System.out.println("My computer took " + timeElapsed + "ms to run compute_pascal(30, 20).");
25 }
26
27

Console
<terminated> Tutorial3 (1) [Java Application] /Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java (29 Nov 2021, 15:01:56 – 15:01:57)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Using a RISC-I processor with 6 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 7051109
The number of register window Underflows is 7051109
The number of register windows pushed onto the stack for each case of overlapping register windows is 7051109
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Using a RISC-I processor with 8 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 2840177
The number of register window Underflows is 2840177
The number of register windows pushed onto the stack for each case of overlapping register windows is 2840177
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Using a RISC-I processor with 16 register windows
The result from compute_pascal(30,20) is 20030010
The number of procedure calls is 40060019
The maximum register window depth is 29
The number of register window Overflows is 30826
The number of register window Underflows is 30826
The number of register windows pushed onto the stack for each case of overlapping register windows is 30826
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

My computer took 45ms to run compute_pascal(30, 20).
```