Computer Architecture II Assignment

Stephen Rowe ID 14319662

Question 1

For Q1 and Q2, I've written two versions, "*Q1_Q2_useful instructions after callr*" and "*Q1_Q2_nops after callr*", I emailed the lecturer asking a question regarding the use of NOPs after CALLR calls, but I haven't gotten an answer back before submission, so I'll submit both versions, **with "*Q1_Q2_useful instructions after callr*" being the main submission**.

Question 1 passes parameters *a*, *b*, *c* to max() through R10, R11, R12, and uses the NOP after the call to max() to pass parameter *c*. R2 is used for global variable inp_int.

Question 2

To execute the pseudo-code with less instructions, I calculated b/2 and b%2 first, as they will be used later in the subroutine. The first instruction checks if b == 0, but I set up *b* to be a parameter for both div() and mod() in the place of the NOP instruction, as putting something into R10 here will not matter if we are returning (in the case of b==0), and will save an extra instruction if we aren't returning.

After calling div() and mod(), the results are stored in local registers (in this window). I call fun() with the parameters *a+a* and *b/2*, again using NOPs where there are no dependencies. The value that will be returned depends on if b%2 == 0, so the NOP for checking this assumes that it is false (i.e. adds *a* to result), but if the function doesn't branch (i.e. its true), then this *a* gets subtracted again.

Question 3

This function uses the following global variables to keep track of the values that we want to output

```
int num_procedure_calls;
int current_window_depth;
int max_window_depth;
int num_overflows;
int num_underflows;
```

I made a struct for PSW that is passed as a pointer to keep track of the CWP, SWP and NUSED values.

Whenever `compute_pascal_RISC()` is called, the method `call_function_risc(ourPSW)` is also called, to update the CWP, SWP and NUSED values, and checks for overflows. Whenever `compute_pascal_RISC()` returns, the function `return_function_risc(ourPSW)` is called to also update the CWP, SWP and NUSED values, and check for underflow.

To allow for the case that 1 register window must be empty to cause an overflow, I changed the pseudocode in the call method:

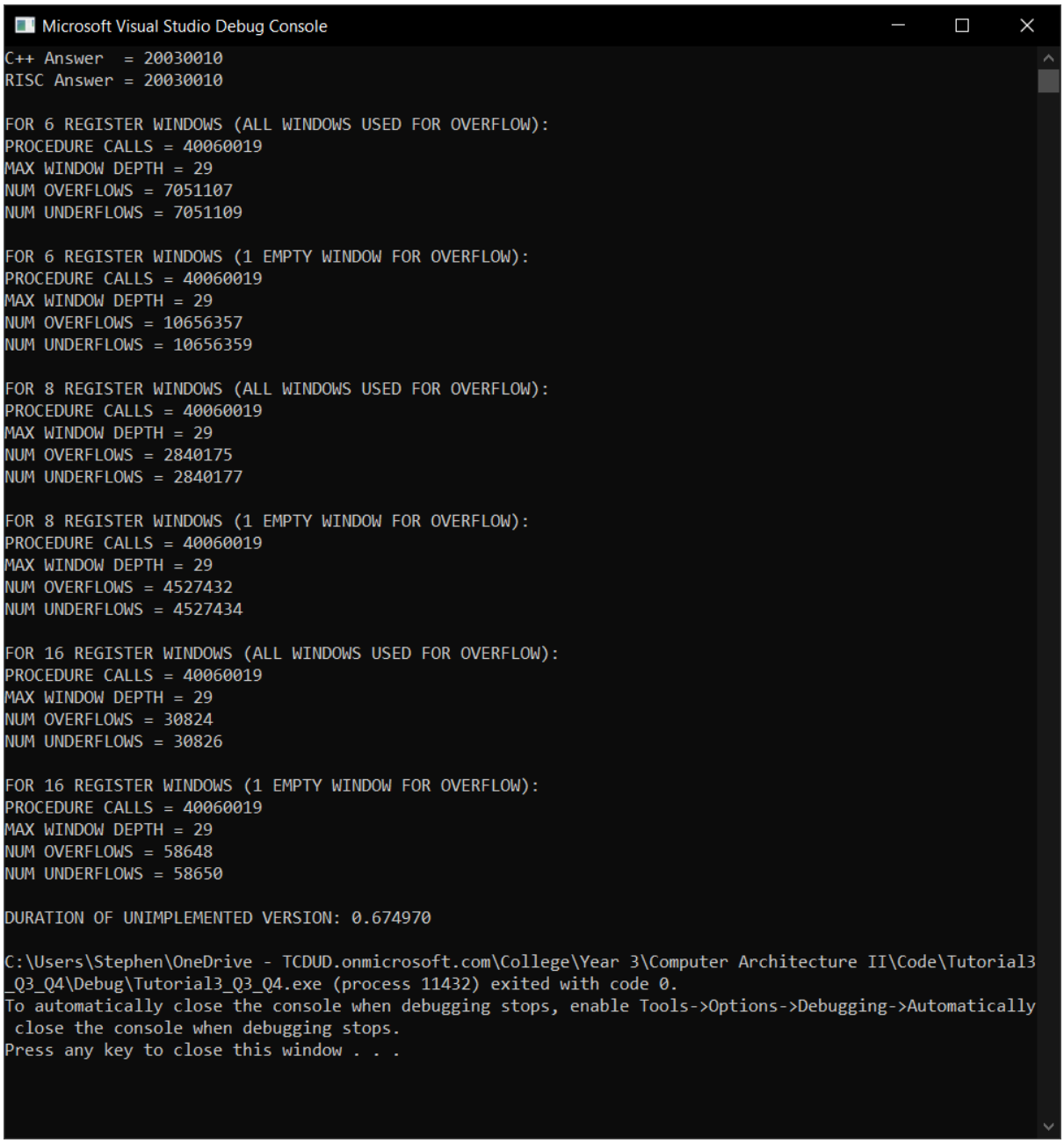
```
if(WUSED = NWINDOWS)
```

to

```
if(WUSED = NWINDOWS - 1)
```

with the number being subtracted from NWINDOWS being passed as a parameter.

(This image is with SWP = 0 and NUSED = 0 when instantiated)



For all versions of the RISC implementation, the number of Procedure Calls will be the same, and the Max Window Depth will be the same. The number of Procedure Calls is dictated by the algorithm, not the hardware, so the number of register windows doesn't affect this. The Max Window Depth is consistent too, as this is also determined by the algorithm, windows will have to be created for the recursion of max depth according to the algorithm for the input values, the depth required isn't affected by the number of register windows available at a time or how overflow is implemented. A new register window must be created every time the function is called (through *call_function_risc())* and it can be removed every time it returns (through *return_function_risc())*, so I keep track of the depth at the start of each of these functions.

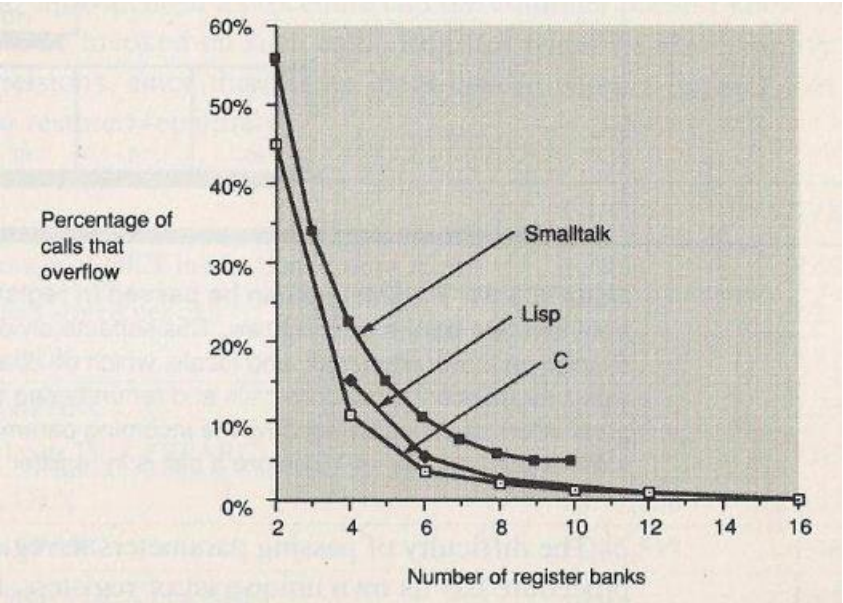
For all the above examples, the number of underflows = the number of overflows + 2. This (+2) is due to the algorithm for checking for underflows

if(WUSED ==2)

Meaning there must always be two “valid” windows on the register windows at all time. When the function is fully finished computing, CWP will return to 0, WUSED will remain at 2 (due to the above pseudocode) and SWP will be CWP-2. **However, if we instantiate SWP to 1, and NUSED to 2 before running the RISC implementation at all**, then when the computations are done, then SWP and NUSED will return to these values of 1 and 2 respectively, meaning we haven’t popped anything unnecessarily from stack. Also note that the num overflows and num underflows are the same this time:

```
BEFORE CALL -  CWP = 0, SWP = 1, NUSED = 2
AFTER CALL - CWP = 0, SWP = 1, NUSED = 2

FOR 6 REGISTER WINDOWS (ALL WINDOWS USED FOR OVERFLOW):
PROCEDURE CALLS = 40060019
MAX WINDOW DEPTH = 29
NUM OVERFLOWS = 7051109
NUM UNDERFLOWS = 7051109

BEFORE CALL -  CWP = 0, SWP = 1, NUSED = 2
AFTER CALL - CWP = 0, SWP = 1, NUSED = 2

FOR 6 REGISTER WINDOWS (1 EMPTY WINDOW FOR OVERFLOW):
PROCEDURE CALLS = 40060019
MAX WINDOW DEPTH = 29
NUM OVERFLOWS = 10656359
NUM UNDERFLOWS = 10656359

BEFORE CALL -  CWP = 0, SWP = 1, NUSED = 2
AFTER CALL - CWP = 0, SWP = 1, NUSED = 2

FOR 8 REGISTER WINDOWS (ALL WINDOWS USED FOR OVERFLOW):
PROCEDURE CALLS = 40060019
MAX WINDOW DEPTH = 29
NUM OVERFLOWS = 2840177
NUM UNDERFLOWS = 2840177

BEFORE CALL -  CWP = 0, SWP = 1, NUSED = 2
AFTER CALL - CWP = 0, SWP = 1, NUSED = 2

FOR 8 REGISTER WINDOWS (1 EMPTY WINDOW FOR OVERFLOW):
PROCEDURE CALLS = 40060019
MAX WINDOW DEPTH = 29
NUM OVERFLOWS = 4527434
NUM UNDERFLOWS = 4527434

BEFORE CALL -  CWP = 0, SWP = 1, NUSED = 2
AFTER CALL - CWP = 0, SWP = 1, NUSED = 2

FOR 16 REGISTER WINDOWS (ALL WINDOWS USED FOR OVERFLOW):
PROCEDURE CALLS = 40060019
MAX WINDOW DEPTH = 29
NUM OVERFLOWS = 30826
NUM UNDERFLOWS = 30826

BEFORE CALL -  CWP = 0, SWP = 1, NUSED = 2
AFTER CALL - CWP = 0, SWP = 1, NUSED = 2

FOR 16 REGISTER WINDOWS (1 EMPTY WINDOW FOR OVERFLOW):
PROCEDURE CALLS = 40060019
MAX WINDOW DEPTH = 29
NUM OVERFLOWS = 58650
NUM UNDERFLOWS = 58650

DURATION OF UNIMPLEMENTED VERSION: 0.810978
```

By increasing the number of register windows, this can massively decrease the number of overflows that can occur. The number of overflows when using **8** register windows is approximately 40% of the number of overflows when using **6** register windows. The number of overflows when using **16** register windows is just over 1% of the number of overflows when using **8** register windows. This is consistent with this diagram from our lecture slides:



If we set the number of register windows that must be free to cause an overflow to 1, then this also massively increased the number of overflows in each case. With 6 register windows, it was ~1.5 times the number of overflows. With 8 register windows, it was ~1.6 times the number of overflows. With 16 register windows, it was almost 2 times the number of overflows. But we can also see that the CWP, SWP and NUSED values will return to the correct values in both calls, so this implementation is also correct. I compared the number of overflows with 8 register windows with 1 free to cause overflow, and with 7 register windows with 0 free to cause overflow, and they both produced the same number of overflows, even though, in the case of 1 free to cause overflow, only the overflow handler is adjusted. Since NWINDOWS is only checked in the overflow handler and not in the underflow handler, then this means that requiring one free window for overflow with N windows will take the same amount of time as having N-1 windows, despite the fact that N windows are still fully used.

Question 4

I used the C++ Chrono library for calculating the duration of the of execution. This is accurate to 0.000001 of a second, or a microsecond. I ran the function 11 times and calculated the average and the median duration for execution.

```
DURATION OF UNIMPLEMENTED VERSION:
        AVERAGE = 1.052818
        MEDIAN = 0.765297
```