Architecture Project 2 parts 1-3

How to run the programs:

1. Define the size of the matrix at the very top where the comments read Number of rows; Number of columns, the example below is for a 3x3 matrix

```
# Number of rows
li      a1, 3
# Number of columns
li      a2, 3
```

2. Then define your test matrix at the very bottom of the code in the .data field.

```
.data
matrix:
   .word 10, 10, 5
   .word 6, 2, 2
   .word 2, 3, 2

avg:
   .word 0
```

3. Compile and run the program
4. Results will be located in memory: the matrix starts at 0x000002fc and ends at 0x0000031c and the avg array starts at 0x00000320.

0x000002f0	02c12403	03010113	00008067	0000000a
0x00000300	0000000a	00000005	00000006	00000002
0x00000310	00000002	00000002	00000003	00000002
0x00000320	00000006	00000005	00000003	00000000

Part 1: Code for unoptimized:

```
lw      ra, 12(sp)
lw      s0, 8(sp)
addi    sp, sp, 16

# Load the address of the matrix into register a0
la      a0, matrix

# Number of rows
li      a1, 8
# Number of columns
li      a2, 8

jal     avg_columns

j      end
```

```
avg_columns:
        addi    sp, sp, -48
        sw      s0, 44(sp)
        addi    s0, sp, 48
        sw      a0, -36(s0)
        sw      a1, -40(s0)
        sw      a2, -44(s0)
        sw      zero, -20(s0)
        j       outer_loop

reinitialize:
        sw      zero, -24(s0)
        sw      zero, -28(s0)
        j       inner_loop

sum_column:
        lw      a4, -28(s0)
        mv      a5, a4
        slli    a5, a5, 1
        add     a5, a5, a4
        slli    a5, a5, 2
        mv      a4, a5
        lw      a5, -36(s0)
        add     a4, a5, a4
        lw      a5, -20(s0)
        slli    a5, a5, 2
        add     a5, a4, a5
        lw      a5, 0(a5)
        lw      a4, -24(s0)
        add     a5, a4, a5
        sw      a5, -24(s0)
        lw      a5, -28(s0)
        addi    a5, a5, 1
        sw      a5, -28(s0)

inner_loop:
        lw      a4, -28(s0)
        lw      a5, -40(s0)
        blt     a4, a5, sum_column
        lw      a5, -40(s0)
        lw      a4, -24(s0)
        divu    a4, a4, a5
        # Load address for avg array
```

```
        la      a5, avg
        lw      a3, -20(s0)
        slli    a3, a3, 2
        add     a3, a5, a3
        sw      a4, 0(a3)
        lw      a5, -20(s0)
        addi    a5, a5, 1
        sw      a5, -20(s0)

outer_loop:
        lw      a4, -20(s0)
        lw      a5, -44(s0)
        blt     a4, a5, reinitialize
        nop
        nop
        lw      s0, 44(sp)
        addi    sp, sp, 48
        jr      ra
end:

.data
#define matrix here:
matrix:
    .word 10, 10, 5
    .word 6, 2, 2
    .word 2, 3, 2

avg:
    .word 0
```

Division we tried to implement but couldn't get to work properly:

```
# Divide 10 by 5 and store the result in a1 without using div instruction
li  a0, 10    # Numerator (10)
li  a2, 5     # Denominator (5)

# Initialize quotient to 0
li  a1, 0

# Loop to perform division
division_loop:
   bge a0, a2, division_subtract
   j   division_done
```

```
division_subtract:
    sub a0, a0, a2    # Subtract denominator from numerator
    addi a1, a1, 1    # Increment quotient
    j division_loop

division_done:
  addi ra, a1, 0
```

Hardware division code:

```
li a0, 10
li a1,5
divu a2, a0, a1
```

Part 2 Code for Optimized:

```
        lw      ra, 12(sp)
        lw      s0, 8(sp)
        addi    sp, sp, 16

        # Load the address of the matrix into register a0
        la      a0, matrix

        # Number of rows
        li      a1, 8
        # Number of columns
        li      a2, 8

        jal     avg_columns

        j      end

    avg_columns:
        addi    sp, sp, -44
        sw      s0, 40(sp)
        addi    s0, sp, 44
        sw      a0, -36(s0)
        sw      a1, -40(s0)
        sw      a2, -44(s0)
        j       outer_loop

    reinitialize:
        sw      zero, -20(s0)
```

```
        sw      zero, -24(s0)
        j       inner_loop

sum_column:
        lw      a4, -24(s0)
        lw      a5, -36(s0)
        slli    a4, a4, 2
        add     a4, a5, a4
        lw      a4, 0(a4)
        lw      a5, -20(s0)
        add     a4, a4, a5
        sw      a4, -20(s0)
        lw      a4, -24(s0)
        addi    a4, a4, 1
        sw      a4, -24(s0)

inner_loop:
        lw      a4, -24(s0)
        lw      a5, -40(s0)
        blt     a4, a5, sum_column
        lw      a5, -40(s0)
        lw      a4, -20(s0)
        divu    a4, a4, a5
        la      a5, avg  # Load address for avg array
        lw      a3, -16(s0)
        slli    a3, a3, 2
        add     a3, a5, a3
        sw      a4, 0(a3)
        lw      a5, -16(s0)
        addi    a5, a5, 1
        sw      a5, -16(s0)

outer_loop:
        lw      a4, -16(s0)
        lw      a5, -44(s0)
        blt     a4, a5, reinitialize
        nop
        nop
        lw      s0, 40(sp)
        addi    sp, sp, 44
        jr      ra
end:

.data
```

```
#define matrix here:
matrix:
   .word 10, 10, 5
   .word 4, 4, 2
   .word 4, 1, 2

avg:
   .word 0
```

Part 3 Analysis:

a) Optimized 8x8 1578 , unoptimized 8x8 2219, Optimized 256x256 1,318,954, unoptimized 256x256 1,974,315, optimized 32x32 21546, unoptimized 32x32 31787.

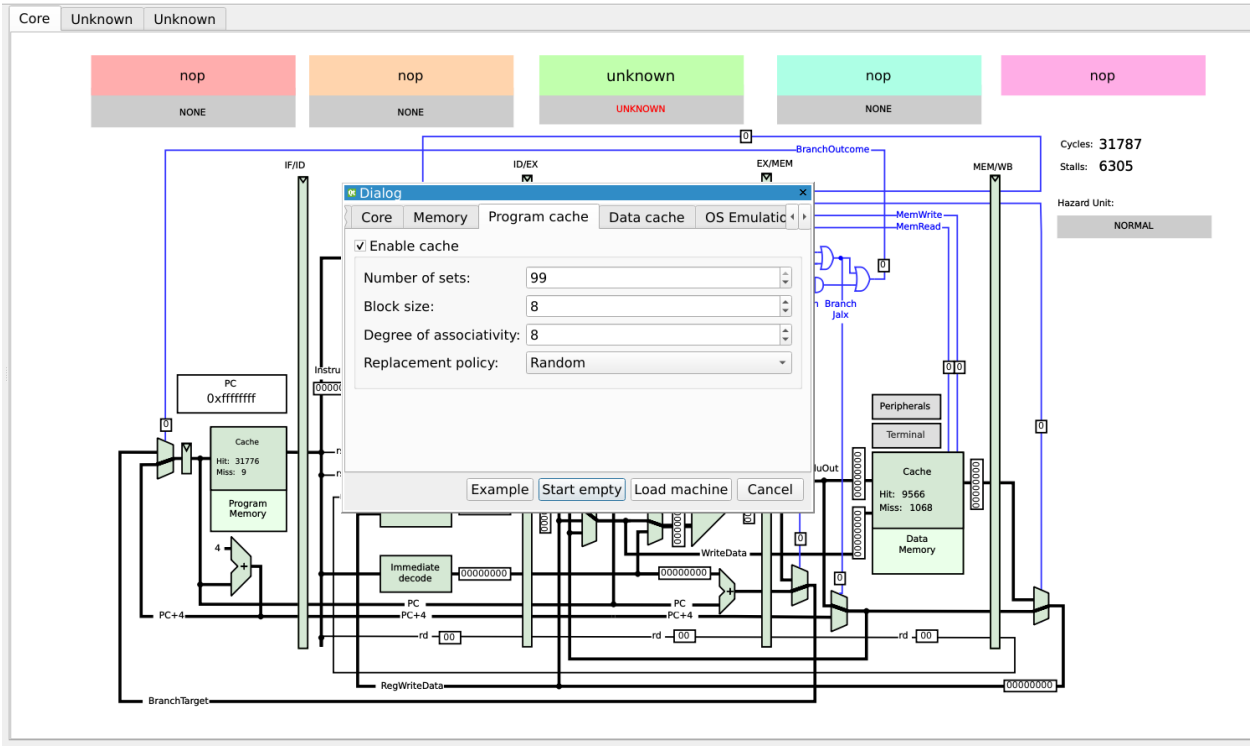
At an 8x8 array the unoptimized program uses 40.62% (base over improved 2219/1578) more cycles to complete the calculation. At an 32x32 array the unoptimized program uses 47.53% (31787/21546) more cycles to complete the calculation. At an 256x256 array the unoptimized program uses 49.69% (1,974,315/1,318,954) more cycles to complete the calculation.

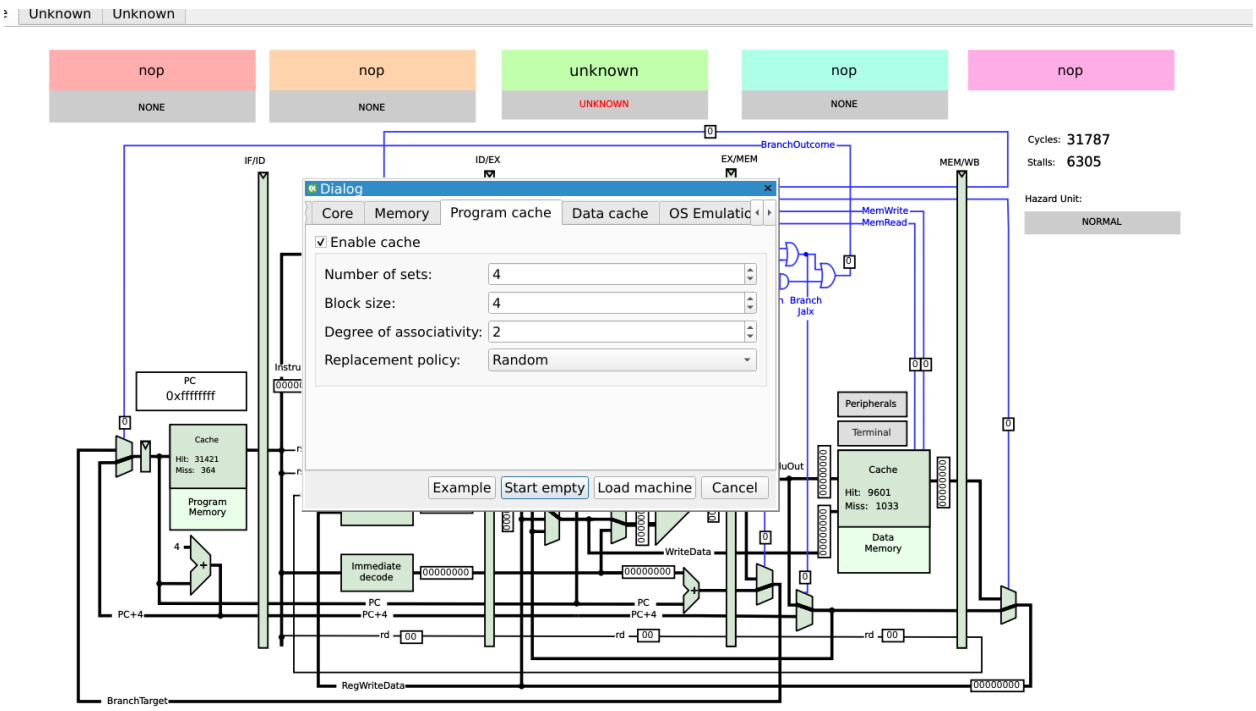
The memory aware implementation is faster than the non-memory aware version because the non-memory aware implementation uses an extra variable to calculate the averages and fill the array. This means extra instructions are executed to move the variable into the array holding the averages, causing the number of cycles to increase as the number of columns in the matrix increases.

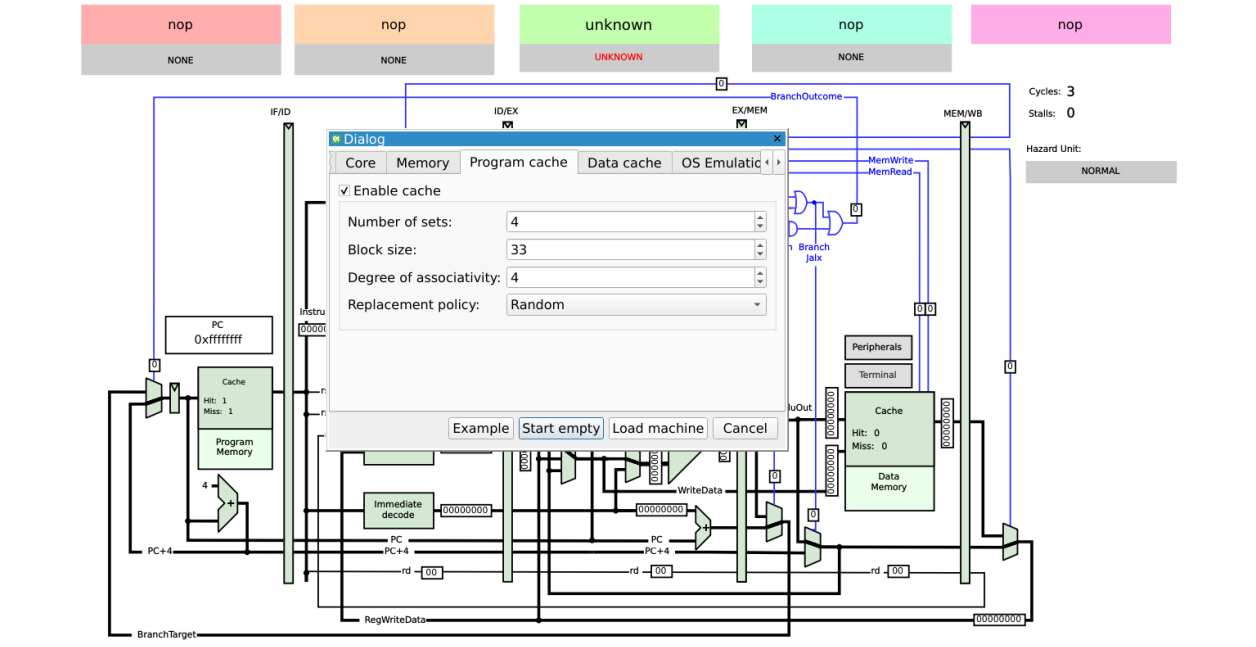
b) Yes, hardware division is much faster than a manufactured division function. We were unable to implement our custom division function into our program, but determined that the function will only execute as many times as there are columns in the matrix. Our function takes 35 cycles to execute while the hardware division takes only 8 cycles. By taking the difference between the two (27) we estimated that if we were able to implement our division function on an 8x8 matrix it would add an additional 216 cycles to both versions of the program.

c) Optimized 8x8 1578 , unoptimized 8x8 2219, Optimized 256x256 1,318,954, unoptimized 256x256 1,974,315, optimized 32x32 21546, unoptimized 32x32 31787.
At an 8x8 array the unoptimized program uses 33.76% more cycles to complete the calculation. At an 32x32 array the unoptimized program uses 38.40% more cycles to complete the calculation. At an 256x256 array the unoptimized program uses 39.80% more cycles to complete the calculation.

Changing the sets, block size, associative, and restarting and recompiling made no difference on cycles used by the program. We could not figure out why that was but we tried many different values at every different possible location with no effect. When the block size went over 32 the cycles immediately went to 3 but other than that there was no change that made an effect. The following two examples are the unoptimized 32x32 cycles with the cache variables shown and the same results. We tested many more but the results were always the same. If I had to choose between implementing division in hardware or quadrupling the size of cache I would choose to implement hardware division. In this program specifically where division is used for each column, more time and cycles would be saved using hardware division. I am not surprised by the conclusion because with the division implementation we tried it got much more complex quickly.




Unknown
Unknown
nop
NONE
nop
NONE
unknown
UNKNOWN
nop
NONE
nop
Cycles: 31787
Stalls: 6305
Hazard Unit:
NORMAL
IF/ID
ID/EX
EX/MEM
MEM/WB
BranchOutcome
Dialog
Core
Memory
Program cache
Data cache
OS Emulatic
Enable cache
Number of sets: 4
Block size: 4
Degree of associativity: 2
Replacement policy: Random
Example
Start empty
Load machine
Cancel
MemWrite
MemRead
Branch
Jalx
PC
0xffffffff
Cache
Hit: 31421
Miss: 364
Program Memory
Peripherals
Terminal
Cache
Hit: 9601
Miss: 1033
Data Memory
Immediate decode
WriteData
PC
PC+4
rd
RegWriteData
BranchTarget


Core
Unknown
Unknown
nop
NONE
nop
NONE
unknown
UNKNOWN
nop
NONE
nop
Cycles: 3
Stalls: 0
Hazard Unit:
NORMAL
IF/ID
ID/EX
EX/MEM
MEM/WB
BranchOutcome
Dialog
Core
Memory
Program cache
Data cache
OS Emulatic
Enable cache
Number of sets: 4
Block size: 33
Degree of associativity: 4
Replacement policy: Random
Example
Start empty
Load machine
Cancel
MemWrite
MemRead
Branch
Jalx
PC
0xffffffff
Cache
Hit: 1
Miss: 1
Program Memory
Peripherals
Terminal
Cache
Hit: 0
Miss: 0
Data Memory
Immediate decode
WriteData
PC
PC+4
rd
RegWriteData
BranchTarget