EECE7352 - COMPUTER ARCHITECTURE: HOMEWORK 2

ANNA DEVRIES[1]

2 October 2019

CONTENTS

LIST OF FIGURES

[1] *Department of Electrical and Computer Engineering, Northeastern University, Boston, United States*

LIST OF TABLES

1 PART A

1.1 Write a RISC-V assembly program for multiplication

Description: Write a RISC-V assembly program to compute the product of two integer values. The two values should be initialized in main(). The main() function should call the product(int x, int y) function, passing the two integer values as arguments. The product function should return the product of the two numbers.

1.1.1 *Develop both a non-recursive and recursive implementation of your assembly program. Submit your assembly code on Blackboard through Turnitin.*

1.1.2 *What is the largest product that can be computed in your program?*

1.1.3 *Discuss how you implemented integer multiplication, since it is not directly supported on the simulator. Discuss an alternative implementation for multiplication. Which implementation would you expect to perform better, and why?*

1.2 Write a recursive RISC-V assembly program to compute the factorial

Description: Write a recursive RISC-V assembly program to compute the factorial of a value that is initialized in main (). We provide a recursive factorial program in the c program example on Blackboard.

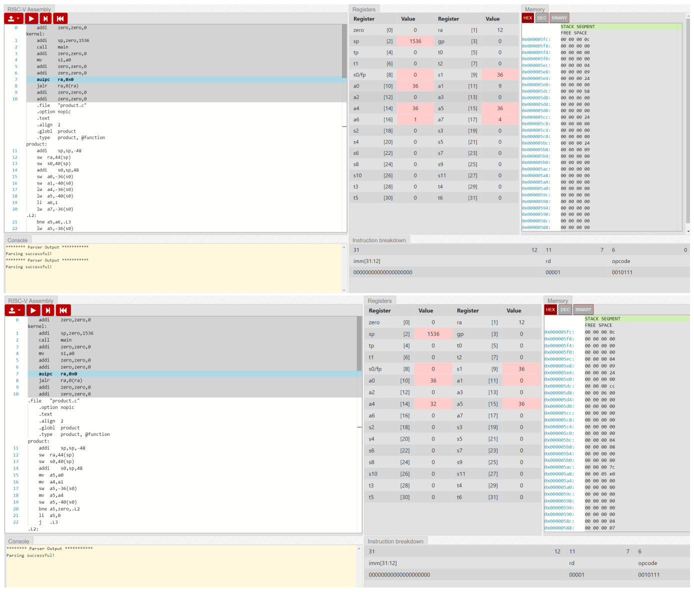
1.2.1 *Submit your assembly code on Blackboard through Turnitin.*

1.3 What is the largest integer value for factorial

Description: What is the largest integer value that you can compute the factorial in your program on the RV32I ISA? Explain why.

1.4 Solution

First, I implemented non-recursive and recursive multiplication programs, end states are shown in Figure 1. The largest value computed by both of these programs was 2,147,483,647 (the max value of a 32-bit signed integer).



```
    addi    zero,zero,0
kernel:
    addi    sp,zero,1536
    call    main
    addi    zero,zero,0
    mv      s1,a0
    addi    zero,zero,0
    addi    zero,zero,0
    auipc   ra,0x0
    jalr    ra,0(ra)
    addi    zero,zero,0
    addi    zero,zero,0
    .file   "product.c"
    .option nopic
    .text
    .align  2
    .globl  product
    .type   product, @function
product:
    addi    sp,sp,-48
    sw  ra,44(sp)
    sw  s0,40(sp)
    addi    s0,sp,48
    sw  a0,-36(s0)
    sw  a1,-40(s0)
    lw  a4,-36(s0)
    lw  a5,-40(s0)
    li  a6,1
    lw  a7,-36(s0)
.L2:
    bne a5,a6,.L3
    lw  a5,-36(s0)
```

Register		Value	Register		Value
zero	[0]	0	ra	[1]	12
sp	[2]	1536	gp	[3]	0
tp	[4]	0	t0	[5]	0
t1	[6]	0	t2	[7]	0
s0/fp	[8]	0	s1	[9]	36
a0	[10]	36	a1	[11]	9
a2	[12]	0	a3	[13]	0
a4	[14]	36	a5	[15]	36
a6	[16]	1	a7	[17]	4
s2	[18]	0	s3	[19]	0
s4	[20]	0	s5	[21]	0
s6	[22]	0	s7	[23]	0
s8	[24]	0	s9	[25]	0
s10	[26]	0	s11	[27]	0
t3	[28]	0	t4	[29]	0
t5	[30]	0	t6	[31]	0

```
             STACK SEGMENT
             FREE SPACE
0x000005fc:  00 00 00 0c
0x000005f8:  00 00 00 00
0x000005f4:  00 00 00 00
0x000005f0:  00 00 00 00
0x000005ec:  00 00 00 04
0x000005e8:  00 00 00 09
0x000005e4:  00 00 00 24
0x000005e0:  00 00 00 00
0x000005dc:  00 00 00 b8
0x000005d8:  00 00 06 00
0x000005d4:  00 00 00 00
0x000005d0:  00 00 00 00
0x000005cc:  00 00 00 24
0x000005c8:  00 00 00 00
0x000005c4:  00 00 00 00
0x000005c0:  00 00 00 00
0x000005bc:  00 00 00 24
0x000005b8:  00 00 00 09
0x000005b4:  00 00 00 00
0x000005b0:  00 00 00 00
0x000005ac:  00 00 00 00
0x000005a8:  00 00 00 00
0x000005a4:  00 00 00 00
0x000005a0:  00 00 00 00
0x0000059c:  00 00 00 00
0x00000598:  00 00 00 00
0x00000594:  00 00 00 00
0x00000590:  00 00 00 00
0x0000058c:  00 00 00 00
0x00000588:  00 00 00 00
```

```
******** Parser Output ***********
Parsing successful!
******** Parser Output ***********
Parsing successful!
```

31	12	11	7	6	0
imm[31:12]		rd		opcode	
00000000000000000000		00001		0010111	

```
    addi    zero,zero,0
kernel:
    addi    sp,zero,1536
    call    main
    addi    zero,zero,0
    mv      s1,a0
    addi    zero,zero,0
    addi    zero,zero,0
    auipc   ra,0x0
    jalr    ra,0(ra)
    addi    zero,zero,0
    addi    zero,zero,0
.file   "product.c"
    .option nopic
    .text
    .align  2
    .globl  product
    .type   product, @function
product:
    addi    sp,sp,-48
    sw  ra,44(sp)
    sw  s0,40(sp)
    addi    s0,sp,48
    mv  a5,a0
    mv  a4,a1
    sw  a5,-36(s0)
    mv  a5,a4
    sw  a5,-40(s0)
    bne a5,zero,.L2
    li  a5,0
    j   .L3
.L2:
```

Register		Value	Register		Value
zero	[0]	0	ra	[1]	12
sp	[2]	1536	gp	[3]	0
tp	[4]	0	t0	[5]	0
t1	[6]	0	t2	[7]	0
s0/fp	[8]	0	s1	[9]	36
a0	[10]	36	a1	[11]	0
a2	[12]	0	a3	[13]	0
a4	[14]	32	a5	[15]	36
a6	[16]	0	a7	[17]	0
s2	[18]	0	s3	[19]	0
s4	[20]	0	s5	[21]	0
s6	[22]	0	s7	[23]	0
s8	[24]	0	s9	[25]	0
s10	[26]	0	s11	[27]	0
t3	[28]	0	t4	[29]	0
t5	[30]	0	t6	[31]	0

```
             STACK SEGMENT
             FREE SPACE
0x000005fc:  00 00 00 0c
0x000005f8:  00 00 00 00
0x000005f4:  00 00 00 00
0x000005f0:  00 00 00 00
0x000005ec:  00 00 00 04
0x000005e8:  00 00 00 09
0x000005e4:  00 00 00 24
0x000005e0:  00 00 00 00
0x000005dc:  00 00 00 cc
0x000005d8:  00 00 06 00
0x000005d4:  00 00 00 00
0x000005d0:  00 00 00 00
0x000005cc:  00 00 00 00
0x000005c8:  00 00 00 00
0x000005c4:  00 00 00 00
0x000005c0:  00 00 00 00
0x000005bc:  00 00 00 04
0x000005b8:  00 00 00 08
0x000005b4:  00 00 00 00
0x000005b0:  00 00 00 00
0x000005ac:  00 00 00 7c
0x000005a8:  00 00 05 e0
0x000005a4:  00 00 00 00
0x000005a0:  00 00 00 00
0x0000059c:  00 00 00 00
0x00000598:  00 00 00 00
0x00000594:  00 00 00 00
0x00000590:  00 00 00 00
0x0000058c:  00 00 00 04
0x00000588:  00 00 00 07
```

```
******** Parser Output ***********
Parsing successful!
```

31	12	11	7	6
imm[31:12]		rd		opcode
00000000000000000000		00001		0010111

Figure 1: Non-recursive end state (A), recursive end state (B).

I implemented integer multiplication utilizing addition for both programs. In the non-recursive program, I created a for loop that added the first variable to itself and decremented the second variable each loop. In the recursive program, I utilized a branch-if-not-equal instruction to compared the second variable to the value one. If branch was taken, the first variable added its initial value to itself and called the product() function again. Below I attached the C code logic for implementing the RISC-V programs. The RISC-V programs are submitted through Turnitin.
Alternatively, multiplication may be implemented with shift operators. This method would require variables to be represented in binary since left shift operators produce powers of two. Below is example C++ code showing logic of this method, code provided by geeksforgeeks.org [3].

```c
#include <stdio.h>

int product(int A, int B){
    int C = A;
    for(B;B>0;B--){
        A = A + C;
        }
    return A;
}
int main(){
    int A = 3;
    int B = 5;
    int result;
    result = product(A,B);
    return result;
}
```

Algorithm 1: Non-recursive multiplication C example

```c
#include <stdio.h>

int product(int A, int B){
    if(B==0){
    return 0;
    }
    return A + product(A,--B);
}

int main(){
    int A = 3;
    int B = 18;

    printf("%d\n",product(A,B));
    return 0;
}
```

Algorithm 2: Recursive multiplication C example

```
#include<bits/stdc++.h>
using namespace std;

int multiply(int n, int m){
    int ans = 0, count = 0;
    while (m){
        // check for set bit and left
        // shift n, count times
        if (m % 2 == 1)
        ans += n << count;

        // increment of place value (count)
        count++;
        m /= 2;
    }
    return ans;
}

int main(){
    int n = 20 , m = 13;
    cout << multiply(n, m);
    return 0;
}
```

Algorithm 3: Multiplication with shift operators

Between the two types of implementation (addition versus shift operators), shift operators perform quicker. The while loop structure of the shift operator method produces a time complexity of T(n) = O(log(n)); however, the iterative for loop in the addition method produces a time complexity of T(n) = O(n) because it runs linearly the total number of n. A time complexity of O(log(n)) outperforms O(n) especially as the size of the sample becomes large, see Figure 2 below [5].

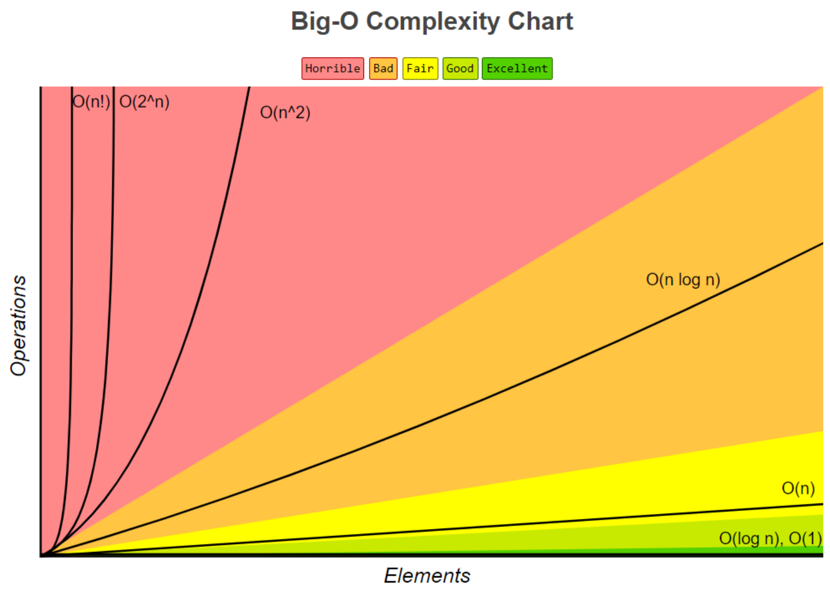


Figure 2: Big-O complexity chart.

Next, I implemented a recursive RISC-V factorial program (RISC-V code was submitted on Turnitin). While the registers can hold an integer value of 2,147,483,647, the BRISC-V simulator limits instruction count. Theoretically, the factorial program should be able to computer 12 factorial; however, the largest integer value it can compute on BRISC-V is 3,628,800 with factorial 10. Code submitted on Turnitin is also attached below.

```
  .file "product.c"
  .option nopic
  .text
  .align  2
  .globl  product
  .type product, @function
product:
  addi  sp,sp,-48
  sw  ra,44(sp)
  sw  s0,40(sp)
  addi  s0,sp,48
  sw  a0,-36(s0)
  sw  a1,-40(s0)
  lw  a4,-36(s0)
  lw  a5,-40(s0)
  li  a6,1
  lw  a7,-36(s0)
.L2:
  bne a5,a6,.L3
  lw  a5,-36(s0)
  sw  a5,-20(s0)
  j .L5
.L3:
  sub a5,a5,a6
  add a4,a7,a4
  sw  a4,-36(s0)
  j .L2
.L5:
  lw  a5,-20(s0)
  mv  a0,a5
  lw  ra,44(sp)
  lw  s0,40(sp)
  addi  sp,sp,48
  jr  ra
  .size product, .-product
  .align  2
  .globl  main
  .type main, @function
main:
  addi  sp,sp,-32
  sw  ra,28(sp)
  sw  s0,24(sp)
  addi  s0,sp,32
  li  a5,4
  sw  a5,-20(s0)
  li  a5,9
  sw  a5,-24(s0)
  lw  a1,-24(s0)
  lw  a0,-20(s0)
  call  product
  sw  a0,-28(s0)
  lw  a5,-28(s0)
  mv  a0,a5
  lw  ra,28(sp)
  lw  s0,24(sp)
  addi  sp,sp,32
  jr  ra
  .size main, .-main
  .ident  "GCC: (GNU) 7.2.0"
  addi  zero,zero,0
  addi  zero,zero,0
  addi  zero,zero,0
  addi  zero,zero,0
  auipc ra,0x0
  jalr  ra,0(ra)
  addi  zero,zero,0
  addi  zero,zero,0
  addi  zero,zero,0
  addi  zero,zero,0
```

Algorithm 4: Non-recursive multiplication

```
.file   "product.c"
    .option nopic
    .text
    .align  2
    .globl  product
    .type   product, @function
product:
    addi    sp,sp,-48
    sw  ra,44(sp)
    sw  s0,40(sp)
    addi    s0,sp,48
    mv  a5,a0
    mv  a4,a1
    sw  a5,-36(s0)
    mv  a5,a4
    sw  a5,-40(s0)
    bne a5,zero,.L2
    li  a5,0
    j   .L3
.L2:
    lw  a5,-40(s0)
    addi    a5,a5,-1
    sw  a5,-40(s0)
    lw  a4,-40(s0)
    lw  a5,-36(s0)
    mv  a1,a4
    mv  a0,a5
    call product
    mv  a5,a0
    mv  a4,a5
    lw  a5,-36(s0)
    add a5,a5,a4
.L3:
    mv  a0,a5
    lw  ra,44(sp)
    lw  s0,40(sp)
    addi    sp,sp,48
    jr  ra
    .size   product, .-product
    .align  2
    .globl  main
    .type   main, @function
main:
    addi    sp,sp,-32
    sw  ra,28(sp)
    sw  s0,24(sp)
    addi    s0,sp,32
    li  a5,4
    sw  a5,-20(s0)
    li  a5,9
    sw  a5,-24(s0)
    lw  a1,-24(s0)
    lw  a0,-20(s0)
    call    product
    sw  a0,-28(s0)
    lw  a5,-28(s0)
    mv  a0,a5
    lw  ra,28(sp)
    lw  s0,24(sp)
    addi    sp,sp,32
    jr  ra
    .size   main, .-main
    .ident  "GCC: (GNU) 7.2.0"
    addi    zero,zero,0
    addi    zero,zero,0
    addi    zero,zero,0
    addi    zero,zero,0
    auipc   ra,0x0
    jalr    ra,0(ra)
    addi    zero,zero,0
    addi    zero,zero,0
    addi    zero,zero,0
    addi    zero,zero,0
```

Algorithm 5: Recursive multiplication

```
  .file "factorial.c"
  .option nopic
  .text
  .align  2
  .globl  multiply_by_add
  .type multiply_by_add, @function
multiply_by_add:
  addi  sp,sp,-48
  sw  ra,44(sp)
  sw  s0,40(sp)
  addi  s0,sp,48
  sw  a0,-36(s0)
  sw  a1,-40(s0)
  lw  a4,-36(s0)
  lw  a5,-40(s0)
  li  a6,1
  lw  a7,-36(s0)
.L2:
  bne a5,a6,.L3
  lw  a5,-36(s0)
  sw  a5,-20(s0)
  j .L5
.L3:
  sub a5,a5,a6
  add a4,a7,a4
  sw  a4,-36(s0)
  j .L2
.L5:
  lw  a5,-20(s0)
  mv  a0,a5
  lw  ra,44(sp)
  lw  s0,40(sp)
  addi  sp,sp,48
  jr  ra
  .size multiply_by_add, .-multiply_by_add
  .align  2
  .globl  factorial
  .type factorial, @function
factorial:
  addi  sp,sp,-48
  sw  ra,24(sp)
  sw  s0,20(sp)
  addi  s0,sp,48
  sw   a0,-20(s0)
  lw   a4,-20(s0)
  li   a5,1
  bgt  a4,a5,.L6
  lw   a5,-20(s0)
  j  .L7
.L6:
  lw   a5,-20(s0)
  addi   a5,a5,-1
  mv   a0,a5
  call   factorial
  mv   a1,a0
  lw   a0,-20(s0)
  call   multiply_by_add
  mv   a5,a0
.L7:
  mv   a0,a5
  lw   ra,24(sp)
  lw   s0,20(sp)
  addi   sp,sp,48
  jr   ra
  .size  factorial, .-factorial
  .align   2
  .globl   main
  .type  main, @function
main:
  addi   sp,sp,-32
  sw   ra,28(sp)
  sw   s0,24(sp)
  addi   s0,sp,32
  li   a5,6
  sw   a5,-20(s0)
  lw   a0,-20(s0)
  call   factorial
  sw   a0,-24(s0)
  lw   a5,-24(s0)
  mv   a0,a5
  lw   ra,28(sp)
  lw   s0,24(sp)
  addi   sp,sp,32
  jr   ra
  .size  main, .-main
  .ident   "GCC: (GNU) 7.2.0"
```

Algorithm 6: Recursive factorial

2 PART B

2.1 Profile Quick-sort

Description: For this problem you will use the qsort.c (quicksort) program provided, and you need to produce a dynamic instruction mix table (similar to Figure A.29 in your textbook) to characterize the execution of the quicksort program. You can perform this study on any architecture of your choice. There are a number of approaches you can take to produce this data. Please make sure to explain how you produced the data in your table and provide details of the tools that you used.

2.1.1 *You could instrument the code to capture the execution frequency of each basic block, and then, using an assembly listing of the program, provide instruction counts (this is slightly imprecise, but very acceptable for this assignment).*

2.1.2 *You could find a tracing program that can capture an instruction trace. You would then have to write a program to count individual instructions (challenging, but not impossible).*

2.1.3 *You could find a tool out on the Internet that provides this capability already for you. While this sounds easy, it may be a bit of work to learn the particular tool you have chosen to use.*

2.2 Solution

For Part B, I utilized DynamoRIO and created a python parsing tool on the ARMv8 machine. DynamoRIO is an opensource binary instrumentation tool. First, I compiled DynamoRIO from source and ran DynamoRIO's instruction trace, creating a log of assembly instructions (command below).

```
./dynamorio/bin64/drrun -c ./dynamorio/api/bin/libinstrace_simple.so -- ./qsort
```

Algorithm 7: DynamoRIO's instruction trace command

Next, I created a python parsing tool which categorized instructions as store, load, branch, jump, and ALU. The parsing script created Table 1.

Table 1: Qsort dynamic instruction mix table.

Program	Loads	Stores	Branches	Jumps	ALU Operations
Qsort	53.1%	6.8%	13.3%	0.1%	26.7%

```
store =  ['stp', 'str', 'strb', 'strh', 'stur', 'msr', 'stxr', 'stlxr']

load =   ['ldr', 'movk', 'movz', 'ubfm', 'sbfm', 'ldur', 'ldp','ldrb', 'adr', 'ldrh', 'bfm', '
    movn', 'ld1', 'ldurb', 'mrs', 'ldaxr', 'ldxr', 'ldrsw']

alu =    ['add', 'sub', 'subs', 'adrp', 'orr', 'adds', 'and', 'svc', 'bic', 'bics', 'rev', 'clz',
     'eor', 'madd', 'umaddl', 'lslv', 'umulh', 'cmeq', 'addp', 'lsrv', 'rbit', 'nop', 'orn', '
    asrv', 'sys', 'dmb', 'udiv', 'msub', 'sdiv', 'ands', 'xx']

jump =   ['ret']

branch = ['bl', 'bcond', 'b', 'cbz', 'cbnz', 'csel', 'br', 'ccmp', 'blr', 'csinc', 'tbz', 'tbnz',
     'csinv']

def find_unique_ops():
        cmds = []
        with open('qsort.log', 'r') as f:
                for line in f:
                        line = line.strip().split(',')[1]
                        if line not in cmds:
                                cmds.append(line)
                                print("Adding: {}".format(line))


def main():
        stores = 0
        loads = 0
        alus = 0
        jumps = 0
        branches = 0
        instructions = 0
        with open('qsort.log', 'r') as f:
                for line in f:
                        line = line.strip().split(',')
      try:
                         line = line[1]
      except Exception as e:
        continue
                        if line in store:
                                stores += 1
                        elif line in load:
                                loads += 1
                        elif line in alu:
                                alus += 1
                        elif line in jump:
                                jumps += 1
                        elif line in branch:
                                branches += 1
                        else:
                                print("Unrecognized Argument! {}".format(line))
                        instructions += 1

        print("Total instructions: {}".format(instructions))
        print("Stores: {} ({}%)".format(stores, (float(stores)/float(instructions)) * 100.0))
        print("Loads: {} ({}%)".format(stores, (float(loads)/float(instructions)) * 100.0))
        print("ALUs: {} ({}%)".format(stores, (float(alus)/float(instructions)) * 100.0))
        print("Branches: {} ({}%)".format(stores, (float(branches)/float(instructions)) * 100.0))
        print("Jumps: {} ({}%)".format(stores, (float(jumps)/float(instructions)) * 100.0))

if __name__ == '__main__':
        # find_unique_ops()
        main()
```

Algorithm 8: Python dynamic instruction mix table parsing script

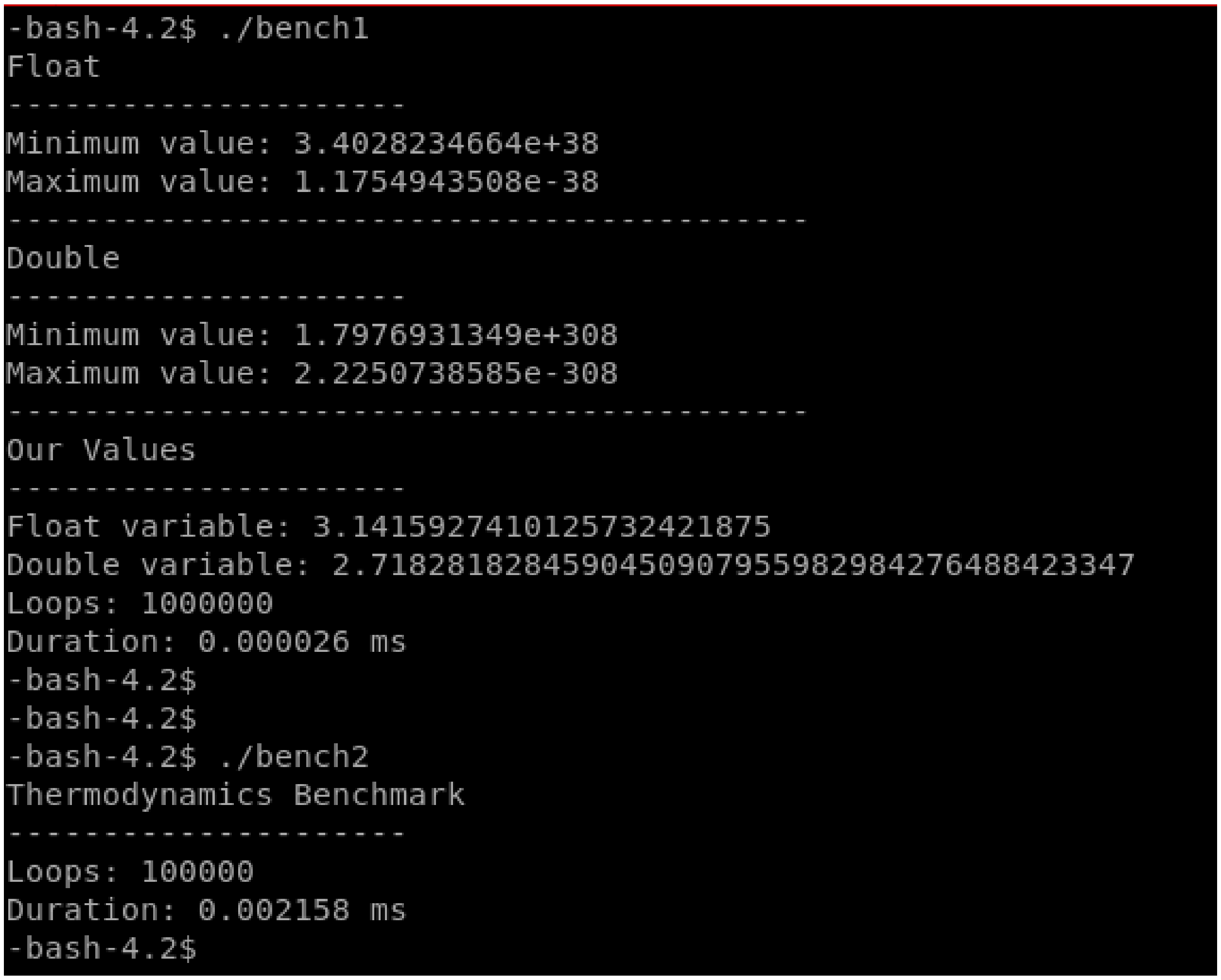
3 PART C

3.1 Floating point benchmarks

Description: For this part of the assignment, write two different benchmark programs on your own that contain significant floating-point content. Compile the programs on X86 and generate an assembly listing of the benchmarks. Then identify 4 different floating-point instructions used in each program (a total of 8) and explain both the operands used by each instruction and the operation performed on the operands by the instruction.

3.2 Solution

For part C, I created two benchmark tests. The first test looped through two functions, circle() and operations(), and tested both float and double floating-point numbers. Operations() contained different floating point operations, including multiplication, division, addition, subtraction, and sign change. Circle() contained sin, cos, tan, and tanh operations. This benchmark measured performance by average time in milliseconds to complete each loop. The x86-64-bit architecture averaged 0.000026ms per loop across 10 iterations of the benchmark.
The second benchmark tested float floating-point numbers and looped through four functions: summation(), heattransfer(), work(), and exponents(). This benchmark focused on testing common operations and equations found in Thermodynamics. Summation() added 0.2 iteratively 360 times; heattransfer() calculated mass of gas given pressure, volume, temperature and delta H; work() calculated energy given pressure, initial volume and final volume; and exponents() calculated different exponential and modulus values. This benchmark measured performance in milliseconds per loop. The x86-64-bit architecture averaged 0.002157ms per loop across 10 iterations of the benchmark. Example results are in Figure 3. C++ code of each benchmark is attached at the end of this section.



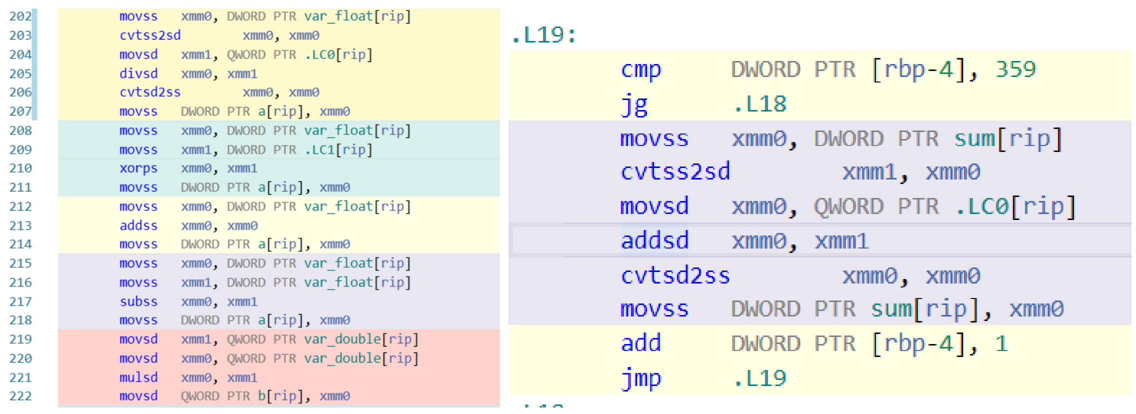
```
-bash-4.2$ ./bench1
Float
---------------------
Minimum value: 3.4028234664e+38
Maximum value: 1.1754943508e-38
-----------------------------------------
Double
---------------------
Minimum value: 1.7976931349e+308
Maximum value: 2.2250738585e-308
-----------------------------------------
Our Values
---------------------
Float variable: 3.1415927410125732421875
Double variable: 2.7182818284590450907955982984276488423347
Loops: 1000000
Duration: 0.000026 ms
-bash-4.2$
-bash-4.2$
-bash-4.2$ ./bench2
Thermodynamics Benchmark
---------------------
Loops: 100000
Duration: 0.002158 ms
-bash-4.2$
```

Figure 3: Results from a single iteration of both benchmarks.

Four unique floating-point instructions for benchmark 1 were xorps, divsd, subsd, and mulsd, and another four for benchmark 2 were movss, movsd, addsd, and cvtsd2ss. Figure 4 illustrates the instructions within the respective benchmark assembly snippets. Operands and operations performed by each instruction are annotated in Table 2.

Table 2: Unique floating-point instructions.

Instruction	Operands	Operation performed on the operands by the instruction
Xorps	xmm0 and xmm1 (register values)	Returns bitwise logical XOR with single-precision floating-point values in xmm1 and xmm2/mem
Divsd	xmm0 and xmm1 (register values)	Divides low double-precision floating-point values xmm0 by xmm1
Subsd	xmm0 and xmm1 (register values)	Subtracts low double-precision floating-point values xmm0 by xmm1
Mulsd	xmm0 and xmm1 (register values)	Multiples low double-precision floating-point values xmm0 and xmm1
Movss	xmm registers or memory location	Moves a scalar single-precision floating-point value from the second operand to the first operand
Movsd	xmm registers or memory location	Moves a scalar double-precision floating-point value from the second operand to the first operand
Addsd	xmm0 and xmm1 (register values)	Adds low double-precision floating-point values xmm0 and xmm1
Cvtsd2ss	xmm0 and xmm1 (register values)	Converts double-precision floating-point value in xmm1 to one single-precision floating-point value in xmm0



```
        movss   xmm0, DWORD PTR var_float[rip]
        cvtss2sd        xmm0, xmm0
        movsd   xmm1, QWORD PTR .LC0[rip]
        divsd   xmm0, xmm1
        cvtsd2ss        xmm0, xmm0
        movss   DWORD PTR a[rip], xmm0
        movss   xmm0, DWORD PTR var_float[rip]
        movss   xmm1, DWORD PTR .LC1[rip]
        xorps   xmm0, xmm1
        movss   DWORD PTR a[rip], xmm0
        movss   xmm0, DWORD PTR var_float[rip]
        addss   xmm0, xmm0
        movss   DWORD PTR a[rip], xmm0
        movss   xmm0, DWORD PTR var_float[rip]
        movss   xmm1, DWORD PTR var_float[rip]
        subss   xmm0, xmm1
        movss   DWORD PTR a[rip], xmm0
        movsd   xmm1, QWORD PTR var_double[rip]
        movsd   xmm0, QWORD PTR var_double[rip]
        mulsd   xmm0, xmm1
        movsd   QWORD PTR b[rip], xmm0
```

```
.L19:
        cmp     DWORD PTR [rbp-4], 359
        jg      .L18
        movss   xmm0, DWORD PTR sum[rip]
        cvtss2sd        xmm1, xmm0
        movsd   xmm0, QWORD PTR .LC0[rip]
        addsd   xmm0, xmm1
        cvtsd2ss        xmm0, xmm0
        movss   DWORD PTR sum[rip], xmm0
        add     DWORD PTR [rbp-4], 1
        jmp     .L19
```

Figure 4: Assembly code snippet from benchmark 1 (A), assembly code snippet from benchmark 2 (B).

```
#include <float.h>
#include <bits/stdc++.h>
#include <sys/time.h>
#include <math.h>
#include <cstdio>

// Prevent some GCC optimizations with global variables
int loops = 1000000;
float var_float = 3.14159265358979323846264338327;
double var_double = 2.71828182845904509079559829842764884233 47;
float a;
double b;
clock_t t;


```

```
// Trig Functions
int circle(){
    sin(var_double);
    sin(var_float);
    cos(var_float);
    cos(var_double);
    tan(var_double);
    tan(var_float);
    tanh(var_double);
    tanh(var_float);

    return 0;
}

// Floating Point Operations
int operations(){
    a = var_float * var_float;
    a = a / var_float;
    a = var_float * 0.2;
    a = var_float / 0.2;
    a = -var_float;
    a = var_float + var_float;
    a = var_float - var_float;

    b = var_double * var_double;
    b = b / var_double;
    b = var_double * 0.2;
    b = var_double / 0.2;
    b = -var_double;
    b = var_double + var_double;
    b = var_double - var_double;

    return 0;
}

// Determine precision for floating-point numbers
int main( ){
    struct timeval start, end;

    printf("Float\n");
    printf("------------------------\n");
    printf("Minimum value: %.10e\n", FLT_MAX);
    printf("Maximum value: %.10e\n", FLT_MIN);
    printf("----------------------------------------------\n");
    printf("Double\n");
    printf("------------------------\n");
    printf("Minimum value: %.10e\n", DBL_MAX);
    printf("Maximum value: %.10e\n", DBL_MIN);
    printf("----------------------------------------------\n");

    int i=0;
    gettimeofday(&start, NULL);
    std::ios_base::sync_with_stdio(false);
    for(i;i<loops;i++){
        circle();
        operations();
    }
    gettimeofday(&end, NULL);
    double time_taken;
    time_taken = (end.tv_sec - start.tv_sec) * 1e6;
    time_taken = (time_taken + (end.tv_usec - start.tv_usec)) * 1e-6;
    time_taken = time_taken / loops;

    printf("Our Values\n");
    printf("------------------------\n");
    printf("Float variable: %1.22f\n",var_float);
    printf("Double variable: %1.40f\n",var_double);
    printf("Loops: %d\n",loops);
    std::cout << "Duration: " << std::fixed
              << time_taken * 1000 << std::setprecision(5);
    std::cout << " ms\n";

  return 0;
}
```

Algorithm 9: Floating operations benchmark 1

```
#include <float.h>
#include <bits/stdc++.h>
#include <sys/time.h>
#include <math.h>
#include <cstdio>

// Prevent some GCC optimizations with global variables
int loops = 100000;
float pressure,volume,temperature,q,n,m,delta_H;
float vol2,vol1;
float sum;
float R = 0.08314;;

int summation(){
    for(int i=0;i<360;i++){
        sum = sum + 0.2;
    }
    return 0;
}

int heattransfer(float P, float V, float T, float delta_H){
    n = (P*V) / (R*T);
    q = delta_H * n;
    m = q*100*4.184*(373 - T);
    return 0;
}

int work(float P, float Vol2, float Vol1){
    float w = -(P)*(Vol2 - Vol1);

    return 0;
}

int exponents(){
    n = R * R;
    n = R * R * R;
    n = R * R * R * R;
    n = 4 % 2;
    n = 10 % 3;

    return 0;
}

// Determine precision and size for floating-point numbers
int main( ){

    struct timeval start, end;
    gettimeofday(&start, NULL);
    std::ios_base::sync_with_stdio(false);
    for(int i=0;i<loops;i++){
        summation();
        pressure = 0.98;
        volume = 8.5;
        temperature = 292;
        delta_H = 1437.17;
        heattransfer(pressure,volume,temperature,delta_H);
        pressure = 200.5678;
        vol2 = 12.5;
        vol1 = 11.1;
        work(pressure,vol2,vol1);
        pressure = 0.76893;
        volume = 3.567;
        temperature = 299;
        delta_H = 1437.17;
        heattransfer(pressure,volume,temperature,delta_H);
        pressure = 156.34;
        vol2 = 1.5;
        vol1 = 0.3;
        work(pressure,vol2,vol1);
        exponents();
    }
    gettimeofday(&end, NULL);
    double time_taken;
    time_taken = (end.tv_sec - start.tv_sec) * 1e6;
    time_taken = (time_taken + (end.tv_usec - start.tv_usec)) * 1e-6;
    time_taken = time_taken / loops;

    printf("Thermodynamics Benchmark\n");
    printf("-----------------------\n");
    printf("Loops: %d\n",loops);
    std::cout << "Duration: " << std::fixed
              << time_taken * 1000 << std::setprecision(5);
    std::cout << " ms\n";

  return 0;
}
```

Algorithm 10: Thermodynamics benchmark 2

4 PART D

4.1 Appendix K

Description: For this problem you will need to read through Appendix K in your text, covering a number of instructions sets, and then answer the following questions:

4.1.1 *Name 2 CISC instruction set architectures and 2 RISC instruction set architectures.*

4.1.2 *Describe 3 characteristics of the DEC Alpha instruction set.*

4.1.3 *Discuss the differences/similarities between MIPS and PowerPC in terms of how they handle conditional branches.*

4.1.4 *Given an example of how register windows work on the SPARC ISA.*

4.1.5 *In your opinion, which generation of the Intel x86 architecture was the most significant advancement from the previous generation of the ISA.*

4.2 Solution

1. Two CISC instruction set architectures are the Motorola 68000 (68K) ISA and DEC VAX ISA. Two RISC instruction set architectures are the RISC-V RV64G ISA and SPARCv9 ISA.
2. One characteristic of the DEC Alpha is it was a RISC architecture and had load/store memory architecture. A second characteristics is it was a super-scalar architecture allowing two wide instruction issue. A third characteristics is it was a 64-bit architecture, intended to replace the CISC 32-bit architectures at the time.
3. First, PowerPC utilizes eight copies of four different condition codes (less than, greater than, equal and summary overflow) which create redundancy and allow multiple condition codes at once without conflict [4]. Second, PowerPC supports more complex conditions on a single branch with CRAND, CROR, CRXOR, CRNAND, CRNOR, and CREQV 4-bit condition code registers, and supports optional floating-point instructions [4]. Third, PowerPC provides count registers that automatically decrement for flow control [4]. MIPS, on the other hand, is more limited in conditional branches. Similar to PowerPC, MIPS also supports compare and branch, but it limits itself to equality and tests against zero [4]. MIPS does not require ALU for conditional branching, simplifying branch determination, but requires the use of simple set-on-less-than instructions [4]. MIPS also may compare float-point by a floating-point conditional branch [4].
4. SPARC implements register windows as a method to reduce register traffic on procedure calls. SPARC utilizes SAVE and RESTORE instead of call/return instructions. The SAVE and RESTORE overlap between current and next windows as the outgoing and incoming parameters respectively. Each window has incoming parameters, global and locals, and outgoing parameters. One example provided from UNM Computer Science Program is below [1]. This code prints a NULL terminated string with SPARC assembly language [1]. This also demonstrates an example of outgoing and incoming parameters in the register windows [1]. Outgoing parameters are placed into %o0-%o5 and incoming parameters are expected to be in %o0 [1].

```
.text
! pr_str - print a null terminated string
!
! Temporaries:  %i0 - pointer to string
!               %o0 - character to be printed
!
pr_str: save    %sp, -96, %sp    ! PROLOGUE - save the current window and
                                 !   allocate the minimum stack frame

pr_lp:  ldub    [%i0], %o0       ! load character
        cmp     %o0, 0           ! check for null
        be      pr_dn
        nop
        call    pr_char          ! print character
        nop
        ba      pr_lp
        inc     %i0              ! increment the pointer (branch delay)

pr_dn:  ret                      ! EPILOGUE return from procedure and
        restore                  !   restore old window; no return value
```

Algorithm 11: SPARC assembly language procedure example

5. In my opinion, the Intel Pentium Pro P6 made the most significant advancement from previous generation Intel x86 architectures. The Pentium Pro P6 introduced super-scalar properties to Intel processors by increasing to a three wide data path, introduced out of order issue, separated instruction and data caches, and created a 14-stage pipeline. Most importantly, however, the Pentium Pro P6 introduced instruction cracking or hardware based binary translation. This allowed for the conversion between x86 and micro operations (which were RISC). This optimized instruction level parallelism as much as possible, peaking the single core processor era. This made Intel more competitive with RISC processors being developed at the time and protected Intel's backward compatibility without losing performance.

5 PART E

5.1 Amdahl

Description: Read the Amdahl, Blaauw and Brooks 1964 paper on the IBM 360 Architecture. Given the timeframe of the paper, what do you find the most impressive feature of the architecture as described by the authors? Justify why you feel this is such a great feature. Also, discuss the representation of the various data types supported on this important ISA, and contrast it with the RISC-V.

5.2 Solution

The most impressive feature of the IBM 360 Architecture was its open-ended design – publishing the ISA Manual of "Principles of Operations" and becoming the standard for backward compatibility. At the time, machine versions were only compatible with themselves. Users must buy the latest processor released for new software or support. This new feature, however, brought forth two major concepts: universal use of assembly programs, compiles and other meta-programs, and any successor machines would be developed from the same family with backward compatibility to previous machines [2]. Additionally, this design feature provided a dependable base for consumers and programmers, promising support for decades through inter-modal compatibility [2]. Changing the fundamental design of processors, these were no longer a single design but rather a family of processors. This encouraged users to invest both time and money into development of programs on this computer series. Backward compatibility made computers more accessible to users and a more understandable investment, allowing users to upgrade processors without loss of data.

IBM 360 Architecture supports the following representation of data types: 64-bit and 32-bit length floating point, 6-bit character size, hexadecimal floating-point radix, two's complement for sign representation, variable-length decimal fields, field specification length, and ASCII and BCD code options. IBM allowed users to choose between the 64- and 32-bit length floating point based upon program requirements for speed/space or precision [2]. IBM chose the 6-bit character size based on existing I/O equipment, familiar usage, and simple specification of field structure; and the floating-point base 16 parameter was chosen to enhance speed through simpler shifting paths [2]. IBM 360 utilized two' complement to represent negative numbers for its fixed-point arithmetic binary system, which does not require any true/complement gates in hardware and works quickly; and implemented variable-length decimal fields to exploit the gain in effective tape rate (which outperformed the disadvantages of internal performance) [2]. While word-marks have storage efficiency in business data, IBM chose length because it simplifies program relocation and instruction modification over different applications [2]. IBM allowed user choice for either BCD or ASCII coding by designing a high degree of code independence with generalized code translation facilities in the processor [2].

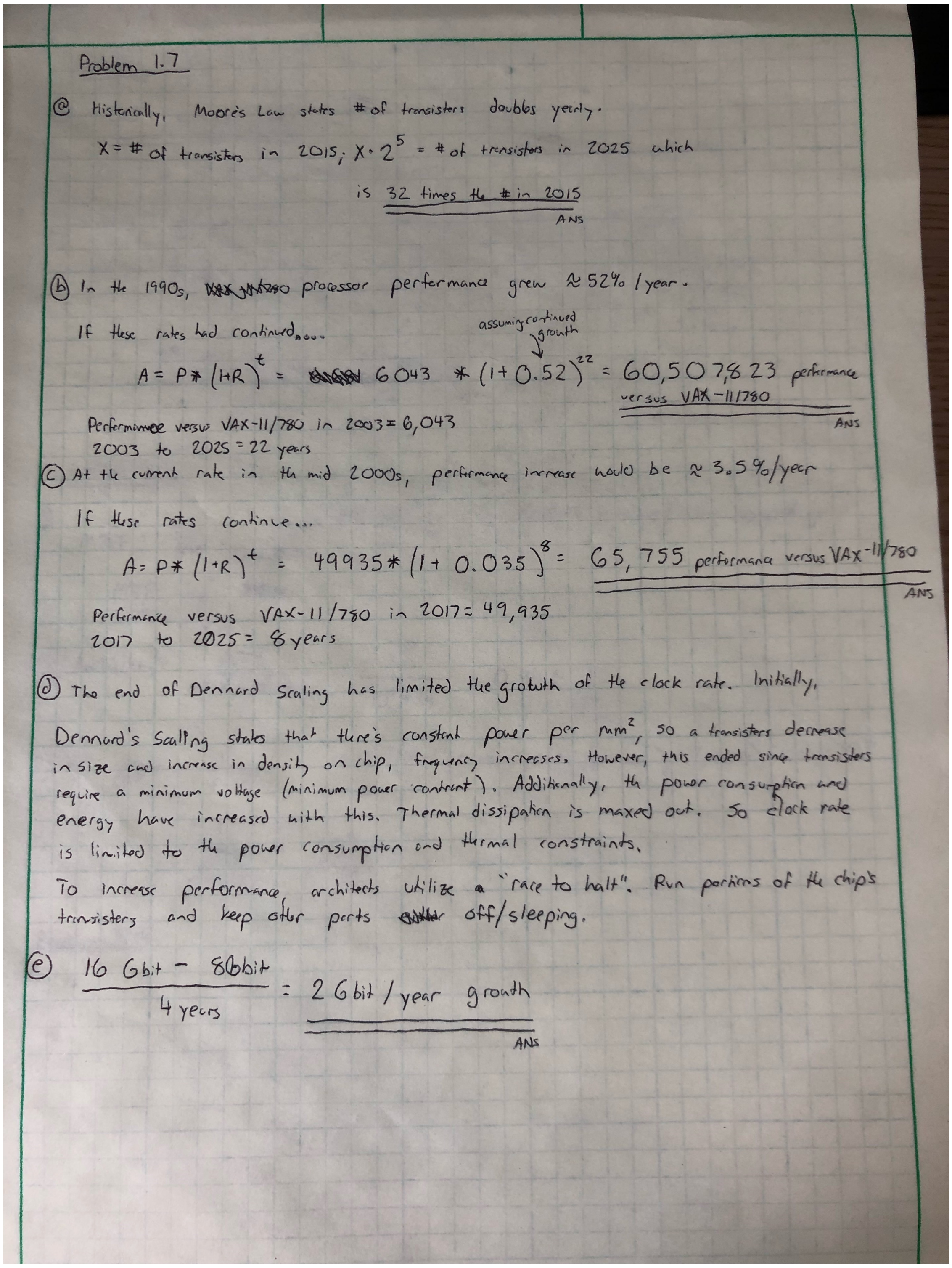
Similar to IBM 360, RISC-V supports various float-point lengths, two's complement and variable-length instruction encoding. However, RISC-V supports a larger variety of floating-point lengths (16-bit, 32-bit, 64-bit and 128-bit) [6]. Furthermore RISC-V supports a larger variety of integer and fixed data lengths (8-bit, 16-bit, 32-bit, 64-bit and 128-bit) rather than just 6-bit lengths, and base 2 floating-point radix [6].

6 PART F

6.1 Chapter 1 Problems (Extra credit)

Description: Complete problems 1.7, 1.8, 1.10, and 1.12 (only a and b) from the text.

6.2 Solution



Problem 1.7

(a) Historically, Moore's Law states # of transisters doubles yearly.

X = # of transistors in 2015; $X \cdot 2^5$ = # of transistors in 2025 which is 32 times the # in 2015

ANS

(b) In the 1990s, ~~VAX-11/780~~ processor performance grew ≈ 52% / year.

If these rates had continued...

assuming continued growth

$$A = P * (1+R)^t = 6043 * (1+0.52)^{22} = 60,507,823$$ performance versus VAX-11/780

ANS

Performance versus VAX-11/780 in 2003 = 6,043

2003 to 2025 = 22 years

(c) At the current rate in the mid 2000s, performance increase would be ≈ 3.5%/year

If these rates continue...

$$A = P * (1+R)^t = 49935 * (1+0.035)^8 = 65,755$$ performance versus VAX-11/780

ANS

Performance versus VAX-11/780 in 2017 = 49,935

2017 to 2025 = 8 years

(d) The end of Dennard Scaling has limited the growth of the clock rate. Initially,

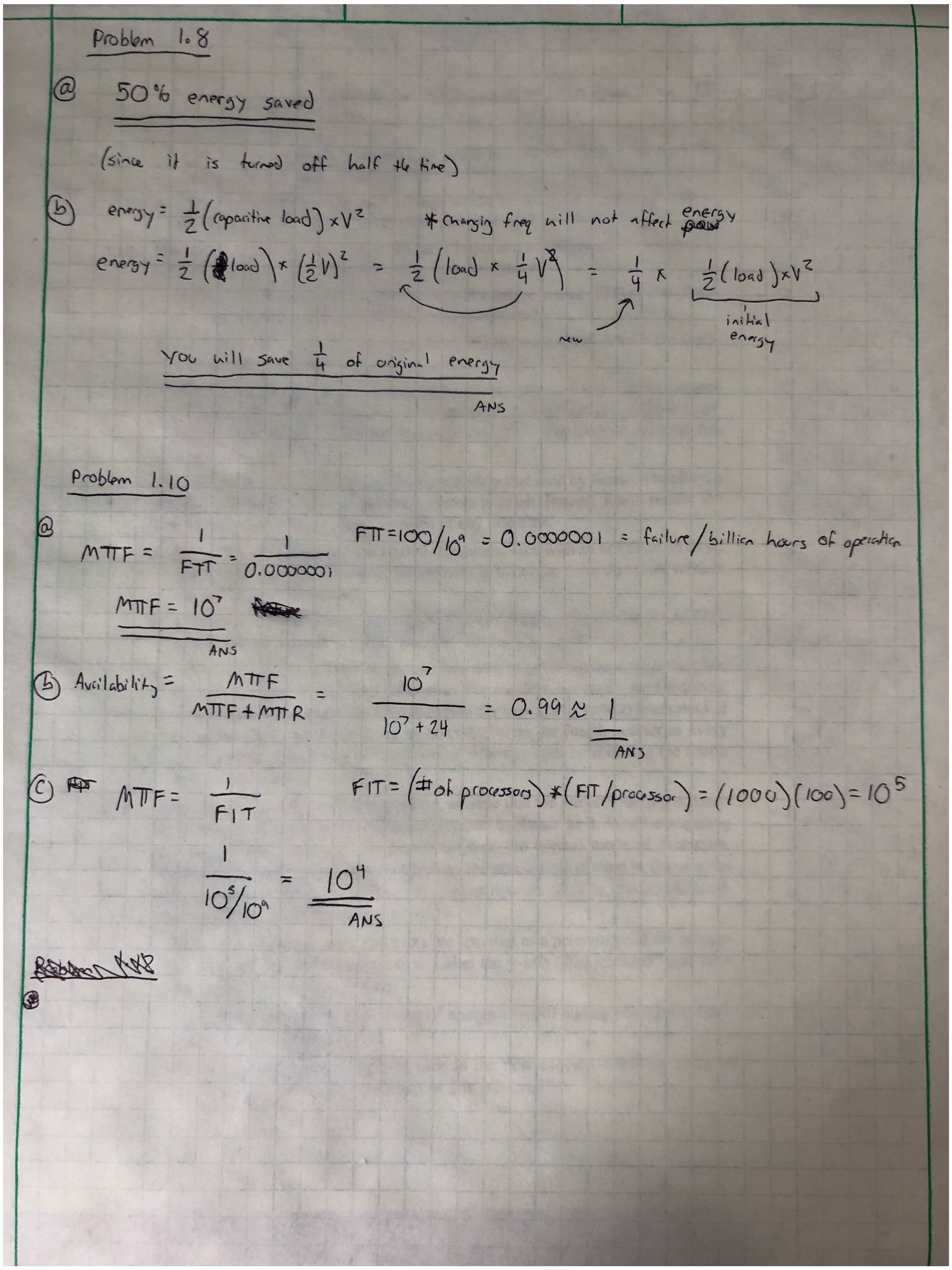
Dennard's Scaling states that there's constant power per mm^2, so a transisters decrease in size and increase in density on chip, frequency increases. However, this ended since transisters require a minimum voltage (minimum power contraint). Additionally, the power consumption and energy have increased with this. Thermal dissipation is maxed out. So clock rate is limited to the power consumption and thermal constraints.

To increase performance, architects utilize "race to halt". Run portions of the chip's transisters and keep other parts off/sleeping.

(e) $$\frac{16\text{ Gbit} - 8\text{ Gbit}}{4\text{ years}} = 2\text{ Gbit / year growth}$$

ANS

Figure 5: Problem 1.7.



Problem 1.8

ⓐ 50% energy saved

(since it is turned off half the time)

ⓑ energy $= \frac{1}{2}(\text{capacitive load}) \times V^2$ * changing freq will not affect energy ~~[illegible]~~

energy $= \frac{1}{2}(\text{load}) \times (\frac{1}{2}V)^2 = \frac{1}{2}(\text{load} \times \frac{1}{4}V^2) = \underset{\text{new}}{\frac{1}{4}} \times \underbrace{\frac{1}{2}(\text{load}) \times V^2}_{\text{initial energy}}$

You will save $\frac{1}{4}$ of original energy

ANS

Problem 1.10

ⓐ $\text{MTTF} = \frac{1}{\text{FIT}} = \frac{1}{0.0000001}$ FIT $= 100/10^9 = 0.0000001$ = failure/billion hours of operation

$\text{MTTF} = 10^7$ ~~[illegible]~~

ANS

ⓑ $\text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} = \frac{10^7}{10^7 + 24} = 0.99 \approx 1$

ANS

ⓒ ~~[illegible]~~ $\text{MTTF} = \frac{1}{\text{FIT}}$ FIT $= (\#\text{ of processors}) * (\text{FIT}/\text{processor}) = (1000)(100) = 10^5$

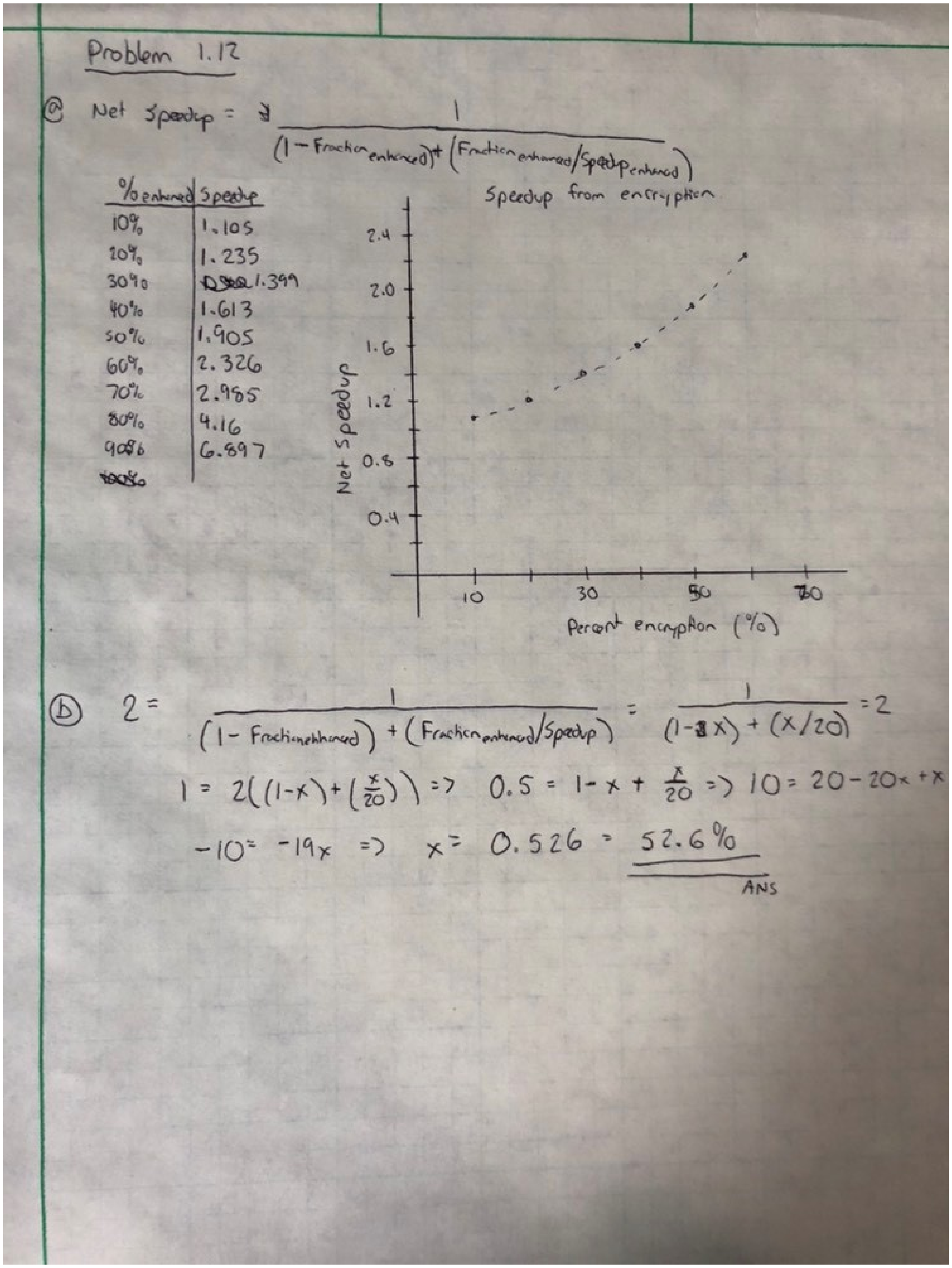
$\frac{1}{10^5/10^9} = 10^4$

ANS

~~Problem 1.18~~

ⓐ

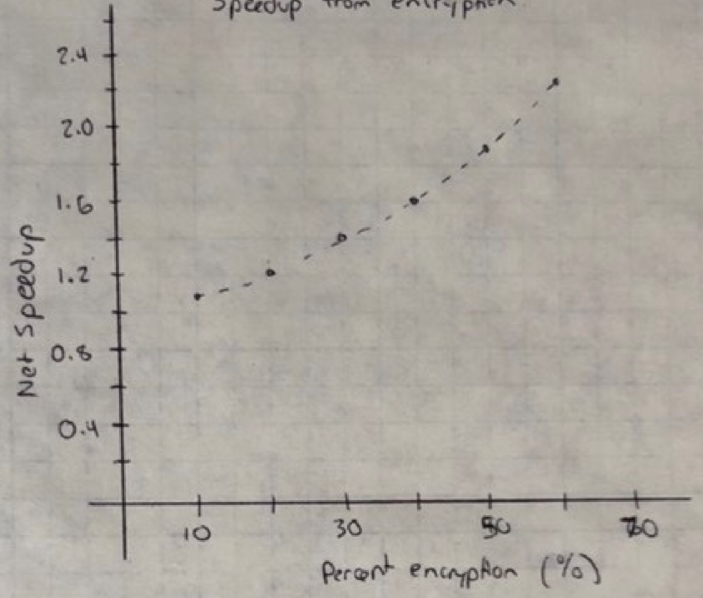
Figure 6: Book problems 1.8 and 1.10.



Problem 1.12

ⓐ Net Speedup = $\dfrac{1}{(1-\text{Fraction}_{\text{enhanced}}) + (\text{Fraction}_{\text{enhanced}}/\text{Speedup}_{\text{enhanced}})}$

% enhanced	Speedup
10%	1.105
20%	1.235
30%	1.399
40%	1.613
50%	1.905
60%	2.326
70%	2.985
80%	4.16
90%	6.897
~~100%~~	



ⓑ $2 = \dfrac{1}{(1-\text{Fraction}_{\text{enhanced}}) + (\text{Fraction}_{\text{enhanced}}/\text{Speedup})} = \dfrac{1}{(1-x) + (x/20)} = 2$

$1 = 2\left((1-x) + \left(\frac{x}{20}\right)\right) \Rightarrow 0.5 = 1 - x + \frac{x}{20} \Rightarrow 10 = 20 - 20x + x$

$-10 = -19x \Rightarrow x = 0.526 = \underline{\underline{52.6\%}}$

ANS

Figure 7: Book problem 1.12 (a and b).

REFERENCES

[1] Register windows. URL https://www.cs.unm.edu/~maccabe/classes/341/labman/node11.html. [Online; accessed 25-September-2019].

[2] G. Amdahl, G. Blaauw, and F. Brooks. *Architecture of the IBM System/360*, volume 08. IBM, 1964.

[3] Geeks for Geeks. Multiplication of two numbers with shift operator. URL https://www.geeksforgeeks.org/multiplication-two-numbers-shift-operator/. [Online; accessed 25-September-2019].

[4] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[5] Barney Maccabe. Big-o complexity chart. URL https://www.bigocheatsheet.com/. [Online; accessed 25-September-2019].

[6] Andrew Waterman, Krste Asanovic, and SiFive Inc., editors. *The RISC-V Instruction Set Manual, Volume I User-Level ISA, Document Version 2.2*. RISC-V Foundation, May 2017.