Arlan Prado, Amuldeep Dhillon
Professor Chenhansa
CS 118-02

Lab 6

Purpose:

This purpose of this assignment is to create a square root program using C++ as inputs and GAS/ASM as the calculations. The algorithm to implement into GAS is the babylonian algorithm. Coordination and teamwork is needed to complete the assignment.

Process:

Initially create the CPP file to print strings, floats, and receiving doubles. These programs are the ones that actually display to the console asking for an input. The input is a single positive integer that will be square rooted.

```cpp
#include <iostream>
#include <iomanip>


extern "C" void _asmMain();

extern "C" void _printString(char* s) {
    std::cout << s;
}

extern "C" double _getDouble(){
    double d;
    std::cin >> d;
    return d;
}

extern "C" void _printFloat(float d){
    std::cout << d << std::endl;
}

int main(){
    _asmMain();
    return 0;
}

```

To be able to use the functions in the CPP file we need to include the extern in the GAS file.

```
.extern _getDouble          #receives the double after prompting the user
.extern _printString        #prints a string that was pushed onto the system stack
.extern _printFloat        #prints a double that was pushed onto the system stack

.data
```

Here are the variables used. All the number variables are floats because all of them are going to be used in the FPU stack. div2 and error are the only numerical constants. div2 is just the number 2 which will be used to half any number, and error will be used for comparisons to make sure our answer is within 1 percent. The other variables hold important values such as the input, our guess value, both the most recent root and the previous one.

```
.data

inpPrompt: .ascii "Please input Number: \0"
rootPrompt: .ascii "Your Square Root is: \0"

div2: .float 2.0              # 2, to be stored in the fpu stack
inp: .float 0.0               # the input from the user
guess: .float 0.0             # the guess the computer makes
rootOld: .float 0.0           # the previous rootNew
rootNew: .float 0.0           # the most recent calculated root
error: .float 0.000000000001          # the % error we want to be in
```

We have the necessary function prologue and proper initialization of the fpu stack.

```
22 .text
23 .globl _asmMain, _root, _guess
24 _asmMain:
25 push %ebp
26 movl %esp, %ebp
27
28 finit
29
30 push $inpPrompt      #pushes string to system stack to be printed by _printString
31 call _printString
32 addl $4, %esp           #clears stack
```

Before implementing the babylonian algorithm, understand the babylonian algorithm. The user's input is the radicand, but the computer sees it as "inp". The "guess" by the computer is inp / 2. Next, divide the inp by guess to get the predicted root, or rootNew. Then after adding rootNew and the guess, divide that number by 2 to get a new guess. Divide the inp by the guess again to get a new rootNew, make the previous rootNew the rootOld. We can use the old root and the new one to see if the change is small enough to be considered precise.

The babylonian algorithm will have to be implemented in the FPU stack. Before the algorithm, the variables have to be set. Variable “inp” is set as the radicand and “guess” is set as inp / 2.
As an afterthought we added a zero condition since our code at the time failed with zero. So if zero is input, it jumps straight to the end where it outputs zero.

```
fst inp                    #store the input into variable "inp"

movl inp, %eax
cmp $0, %eax
je exit
movl $0, %eax

#calculate the "guess", guess = inp(radicand) / 2

fstp guess
flds div2
flds guess
fdivp
fsts guess
```

Here is the babylonian algorithm, (_root and _guess will be explained with pictures):

```
babylonian:

calll _root            #root = inp/guess
flds error
flds rootNew           #load rootNew
flds rootOld           #load rootOld, (the previous rootNew)
fsubp
fabs
fcomi                  #compare the difference of rootNew and rootOld
jb exit                #if the difference is less than 0.01 error, then jump out of algorithm
movl rootNew, %eax     #if the error margin is greater than 0.01, keep going
movl %eax, rootOld     #rootNew is now rootOld
calll _guess           #guess = (guess + rootOld) / 2
jmp babylonian
```

After getting the rootNew from _root, compare rootNew and the previous root (rootOld). If the difference between the two is within the margin of error, then the root is found and jumps out of the loop. If the case is not met, then rootNew is now rootOld and a rootNew is going to be calculated after the functions _guess and _root.

One thing to notice is that we created functions that that have no values being pushed to the stack. We did this because the values we want to manipulate are already available to the functions with the floating point stack. Functions are more beneficial to have since they immediately go to the line after the “call”. If the functions were simply labels instead, then I would have to make a label on the lines after each “call” then “jump” to that specific label. No values in the system stack have to be cleared because nothing was pushed in the first place.

The function _root ends with an empty stack so it is not a concern for the current trace. We load error, rootNew, and rootOld into the FPU stack.

st(0)	rootOld
st(1)	rootNew
st(2)	error

We then subtract rootNew from rootOld to calculate the difference, poping rootOld out and replacing rootNew with the difference. The difference is then compared to the margin of error. If the difference is greater we simply go through the loop and check again, once the difference is below our margin of error we keep rootNew and display it.

st(0)	rootOld-rootNew
st(1)	error

The stack is later re-initialized so the leftovers are simply kept

```
 89  _guess:
 90  pushl %ebp
 91  movl %esp, %ebp
 92
 93  finit               #clear fpu stack
 94  flds div2           #load 2
 95  flds guess
 96  flds rootOld
 97  faddp               #add guess & rootOld and stores in st(1) then pop st(0) so st(1) is now st(0)
 98  fdivp               #divide st(0) by st(1) (= 2)
 99  fsts guess          #store the new value in guess
100
101  pop %ebp
102  retl
```

The _guess function calculates a new guess from the most recent guess and root. After clearing the FPU stack, load 2, then the guess, then the root. The FPU stack will look like this: (with labels on the left and values on the right)

st(0)	rootOld
st(1)	guess
st(2)	2

“fadd” will add st(1) and st(0) and store it in st(1). “faddp” will do the same thing except it will pop out st(0) so the sum will move into st(0). FPU stack now:

st(0)	rootOld + guess
st(1)	2

“fdivp” divides st(0) by st(1), stores it in st(1), pops st(0), so now st(0) is the new guess.

st(0)	guess = (rootOld + guess) / 2

The stack is not cleared so the new guess can be used in the _root function.

```
_root:
pushl %ebp
movl %esp,%ebp

flds inp            #load inp
fdivp               #divide inp by guess
fstp rootNew        #the new value is the rootNew

pop %ebp
retl
```

The _root function calculates the new root by loading “inp”.

st(0)	inp
st(1)	Guess (left from the _Guess func)

We then divide the inp by our guess leaving us with our new (more accurate root)

st(0)	rootNew=inp/guess

The result is the new rootNew and the FPU stack is now empty because of “fdivp” and “fstp rootNew”.

The babylonian algorithm will give us the square root within a 0.00000000001 margin of error with minimal loops. The small number is to account for incredibly small floats such as a nano. So where .001 would work for numbers greater than 1, to account for numbers smaller than 1 the margin of error has to be smaller than the number itself.

To print the square root and the string for the rootPrompt, the CPP functions must be called:

```
62  exit:
63  push $rootPrompt
64  call _printString
65  addl $4, %esp           #prints the root prompt and empties the system stack
66
67  pushl rootNew+4
68  pushl rootNew
69  call _printFloat
70  addl $8, %esp           #prints the root and empties the system stack
71
72
73  pop %ebp
74  ret                     #clear stack
```

Test Cases:
Whole Root:

```
debian@debian:~/cs118/lab7$ ./aprilbabyl
Please input Number: 25
Your Square Root is: 5
```

Input is 25 and output is 5 (Correct)
Non-Whole Root:

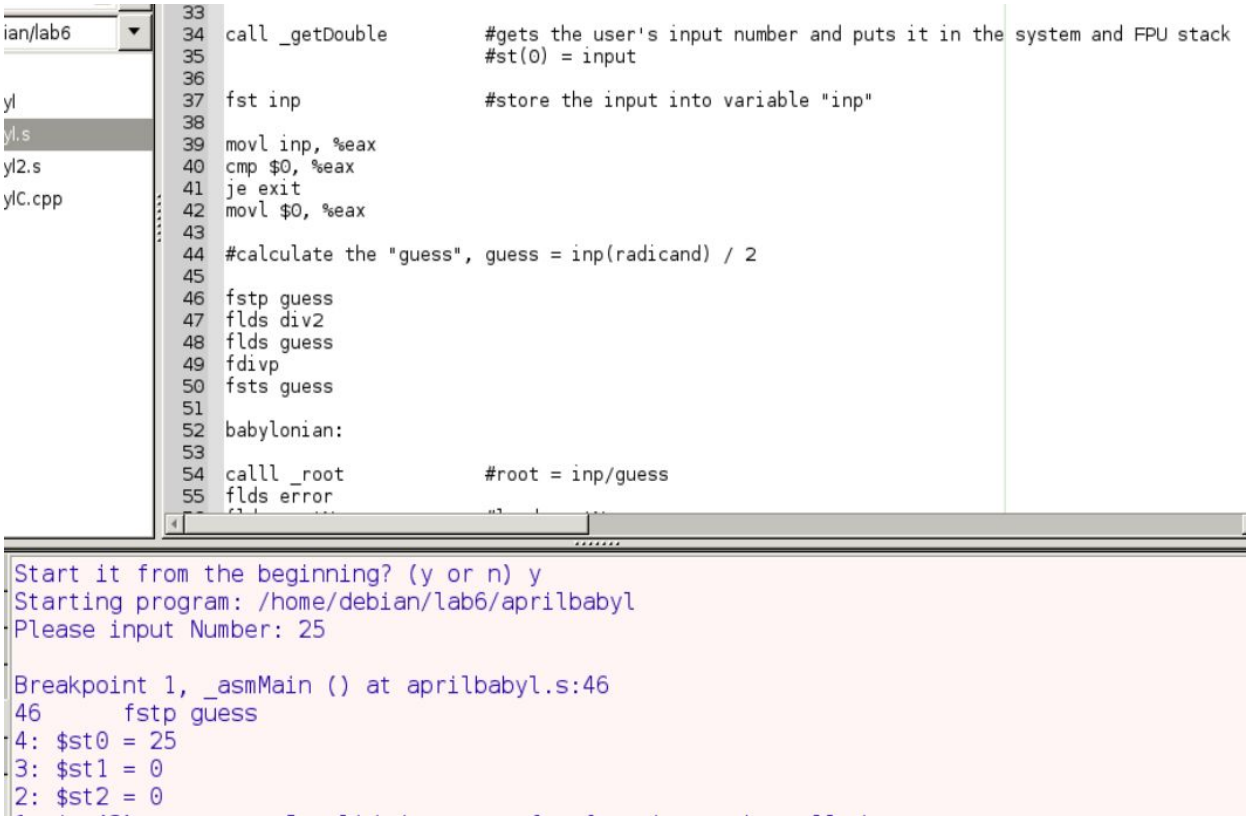
```
debian@debian:~/cs118/lab7$ ./aprilbabyl
Please input Number: 23
Your Square Root is: 4.79583
```

Input is 23 and output is 4.79583 (Calculator answer is 4.79583152331) within 1%

Floating Point Trace:

For this example, I used “25” as the input

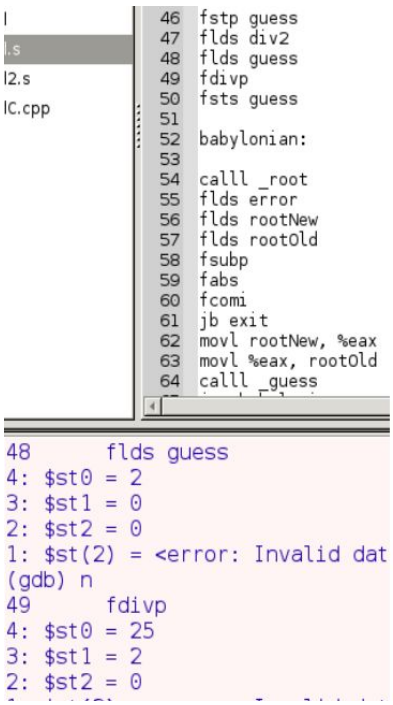
1.



```
ian/lab6
yl
yl.s
yl2.s
ylC.cpp

33
34  call _getDouble         #gets the user's input number and puts it in the system and FPU stack
35                          #st(0) = input
36
37  fst inp                 #store the input into variable "inp"
38
39  movl inp, %eax
40  cmp $0, %eax
41  je exit
42  movl $0, %eax
43
44  #calculate the "guess", guess = inp(radicand) / 2
45
46  fstp guess
47  flds div2
48  flds guess
49  fdivp
50  fsts guess
51
52  babylonian:
53
54  calll _root             #root = inp/guess
55  flds error

Start it from the beginning? (y or n) y
Starting program: /home/debian/lab6/aprilbabyl
Please input Number: 25

Breakpoint 1, _asmMain () at aprilbabyl.s:46
46      fstp guess
4: $st0 = 25
3: $st1 = 0
2: $st2 = 0
```

Before getting the guess, the inputted number is stored in inp and is in the FPU stack.

2.



```
l
.s
l2.s
lC.cpp

46  fstp guess
47  flds div2
48  flds guess
49  fdivp
50  fsts guess
51
52  babylonian:
53
54  calll _root
55  flds error
56  flds rootNew
57  flds rootOld
58  fsubp
59  fabs
60  fcomi
61  jb exit
62  movl rootNew, %eax
63  movl %eax, rootOld
64  calll _guess

48      flds guess
4: $st0 = 2
3: $st1 = 0
2: $st2 = 0
1: $st(2) = <error: Invalid dat
(gdb) n
49      fdivp
4: $st0 = 25
3: $st1 = 2
2: $st2 = 0
```

The FPU stack at line 49 is ready for “fdivp” to get the guess.

3.

```
Breakpoint 1, babylonian () at aprilbabyl.s:54
54      calll _root             #root = inp/guess
3: $st0 = 12.5
2: $st1 = 0
1: $st2 = 0
```

The guess will be used as the divisor to the input. Will go to the _root function now.

4.

```
87      fdivp                   #divide inp by guess
3: $st0 = 25
2: $st1 = 12.5
1: $st2 = 0
(gdb) n
88      fstp rootNew            #the new value is the rootNew
3: $st0 = 2
2: $st1 = 0
1: $st2 = 0
```

In the first run of the _root function, the predicted root (or rootNew) should be 2 because of the initial calculation for the guess (guess = input / 2). After dividing st0 is popped so rootNew is st0.

5.

```
55      flds error
3: $st0 = 0
2: $st1 = 0
1: $st2 = 0
(gdb) n
56      flds rootNew            #load rootNew
3: $st0 = 9.99999996004197200250018795486539 6e-12
2: $st1 = 0
1: $st2 = 0
(gdb) n
57      flds rootOld            #load rootOld, (the previous rootNew)
3: $st0 = 2
2: $st1 = 9.99999996004197200250018795486539 6e-12
1: $st2 = 0
(gdb) n
58      fsubp
3: $st0 = 0
2: $st1 = 2
1: $st2 = 9.99999996004197200250018795486539 6e-12
```

After the loads, the FPU stack is ready to calculate if the new root and the old root (which is 0) is within the margin of error that we want it to be. It should automatically fail since the distance between the numbers is larger than the margin for error.

6.

```
59      fabs
3: $st0 = -2
2: $st1 = 9.99999996004197200250018795486539 6e-12
1: $st2 = 0
(gdb) n
60      fcomi                   #compare the difference of rootNew and rootOld
3: $st0 = 2
2: $st1 = 9.99999996004197200250018795486539 6e-12
1: $st2 = 0
(gdb) n
```

So the difference between 0 and 2 is -2, then the top of the stack is popped so fabs (absolute value) would work. fcomi compares the absolute difference with the margin of error. If the absolute difference is less than the margin of error it would jump out, but it does not since 2 is greater than 0.00000000001.
Now rootOld is 2 and it will be used in the _guess function.

7.

```
98      flds div2           #load 2
3: $st0 = 0
2: $st1 = 0
1: $st2 = 0
(gdb) n
99      flds guess
3: $st0 = 2
2: $st1 = 0
1: $st2 = 0
(gdb) n
100     flds rootOld
3: $st0 = 12.5
2: $st1 = 2
1: $st2 = 0
(gdb) n
101     faddp               #add guess & rootOld and stores in st(1) then pop st(0) so st(1
) is now st(0)
3: $st0 = 2
2: $st1 = 12.5
1: $st2 = 2
(gdb) n
102     fdivp               #divide st(0) by st(1) (= 2)
3: $st0 = 14.5
2: $st1 = 2
1: $st2 = 0
(gdb) n
103     fsts guess          #store the new value in guess
3: $st0 = 7.25
2: $st1 = 0
1: $st2 = 0
```

The stack is cleared then we load in the appropriate numbers up to line 100. From 101 to 103, the formula (guess + rootOld) / 2 = guess is implemented using floating point instructions. Leaving the new guess in the FPU stack for the _root function.

8.

```
Breakpoint 2, _root () at aprilbabyl.s:86
86      flds inp            #load inp
3: $st0 = 7.25
2: $st1 = 0
1: $st2 = 0
(gdb) n
87      fdivp               #divide inp by guess
3: $st0 = 25
2: $st1 = 7.25
1: $st2 = 0
(gdb) n
88      fstp rootNew        #the new value is the rootNew
3: $st0 = 3.448275862068965517308674617602548 6
2: $st1 = 0
1: $st2 = 0
```

So with this, the forumla (input / guess) = rootNew is executed.

9.

```
56      flds rootNew            #load rootNew
3: $st0 = 9.99999996004197200250018795486 5396e-12
2: $st1 = 0
1: $st2 = 0
(gdb) n
57      flds rootOld            #load rootOld, (the previous rootNew)
3: $st0 = 3.448275804519653320 3125
2: $st1 = 9.99999996004197200250018795486 5396e-12
1: $st2 = 0
(gdb) n
58      fsubp
3: $st0 = 2
2: $st1 = 3.448275804519653320 3125
1: $st2 = 9.99999996004197200250018795486 5396e-12
(gdb) n
59      fabs
3: $st0 = -1.448275804519653320 3125
2: $st1 = 9.99999996004197200250018795486 5396e-12
1: $st2 = 0
(gdb) n
60      fcomi                   #compare the difference of rootNew and rootOld
3: $st0 = 1.448275804519653320 3125
2: $st1 = 9.99999996004197200250018795486 5396e-12
1: $st2 = 0
```

So now to compare the absolute difference with the error. The error, rootOld, and rootNew is now loaded. The absolute difference is still bigger than the error but the absolute difference has gone smaller compared to before.

The most noticeable thing is the change in numbers. rootNew and guess are getting closer to 5 with each iteration, and the absolute difference between the old root and the new root is getting closer to the margin of error we want to be in.

	Iteration 1	Iteration 2
guess	12.5	7.25
rootNew	2	3.448
rootOld	0	2
Error (absolute difference)	2	1.448

Through more iterations: guess, rootNew, and rootOld will get closer to 5 and the error between rootNew and rootOld will be decreasing. Once the error less than 0.00000000001, then the root will be displayed to the user giving the correct answer.

```
Your Square Root is: 5
[Inferior 1 (process 29790) exited normally]
```

Pitfalls:

There was one major pitfall that we could not comprehend. The variable “inp” was previously “num”, but whenever we tried to see what the value of num is through gdb (using p/f num), it would give random junk before and after we assigned it a value. num was still receiving and putting the float from the user into the FPU stack correctly, but it was still giving us the same junk. We were confused to what to do since the only lines num was involved in were initializing the variable, storing the user’s value from the FPU stack, and storing the value back into the FPU stack. The work around for this problem was just to rename num. So there was a weird problem with naming the variable num.

A simple pitfall was the constant “typos” we encountered, the most notorious example was an instance where we had written “fstps guess” instead of “fsts guess” which was only uncovered after numerous breakpoints and constant fpu stack checks.

A pitfall we did not see that happened during the live demo was if root was smaller than the margin of error. When that happened, the program would think that root

is within the margin of error making it correct, but it was not. The margin of error was too large so to fix it, the solution was to make the margin of error smaller. The error now is 0.00000000001. This now allows the problem to by bypassed since the root of a whole number can never be smaller than that.

Possible Improvements:

A possible improvement on the code was to make it so there was only one root variable. We did not need the two root variables because we could have just compared the guess and the root. The root and guess were eventually getting closer together so a comparison between the two would have been fine. Though when I tested out the program with replacing rootOld and guess, the root that comes out of the rootOld method is more accurate than the guess method. The rootOld method is more accurate because it lets the program loop one more time to get another compare.