Floating point

CSC 236

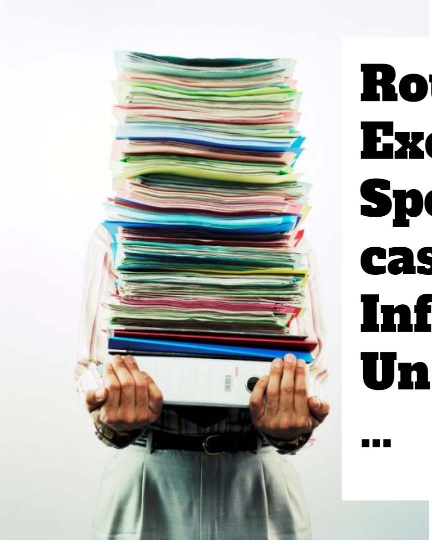
Floating point

- Floating point
  - Beyond integers
  - Real* numbers
  - $\pm n \times 10^{exp}$
- Theory of FP is vast
  - One half of semester in numerical methods
- Two areas
  - FP number format
  - Basic coding of FP co-processor

(*) actually rationals, ie, fractions

Floating point standard

- In 1985 the IEEE issued a standard for Floating-Point IEEE 754-1985
- Updated in 2008 IEEE 754-2008



Rounding,
Exceptions,
Special
cases,
Infinity,
Un-normals,
...

IEEE 754-2008

- Single precision 32 bits
- Double precision 64 bits
- Quadruple precision 128 bits

Fractions

- Decimal fraction
  - 3.14159
  - 3 + 1/10 + 4/100 + 1/1000 + 5/10000 + 9/100000
- Binary fraction
  - 11.00100100001
  - 3 + 1/8 + 1/64 + 1/2048 = 3.14110

IEEE 754-2008 32 bit format

1	8	23
±	**exponent**	**significand**

Real number: $\pm n \times 10^{exp}$

IEEE standard: $\pm \text{significand} \times 2^{\text{exponent}}$

IEEE 754-2008 32 bit format

1	8	23
±	**exponent**	**significand**

- Sign
  - 0 = pos
  - 1 = neg

IEEE 754-2008 32 bit format

1	8	23
±	**exponent**	**significand**

- Exponent
  - Biased = excess 127
  - `value = e + 127`
  - `e = value - 127`
- Store into the field the value
  - 2^0 ⇒ `exponent = 0` ⇒ `value = 127`
  - 2^{44} ⇒ `exponent = 44` ⇒ `value = 171`
  - 2^{-20} ⇒ `exponent = -20` ⇒ `value = 107`

Largest value is 254

- **255 is reserved**
- **exponent = 127**
- 2^{127}

Smallest value is 1

- **0 is reserved**
- **exponent = -126**
- 2^{-126}

IEEE 754-2008 32 bit format

1	8	23
±	**exponent**	**significand**

- Many ways to encode 20_{10}
  - $10100. \times 2^0$ (20×1)
  - 1010.0×2^1 (10×2)
  - 101.00×2^2 (5×4)
  - 10.100×2^3 (2.5×8)
  - 1.0100×2^4 (1.25×16)
  - $.10100 \times 2^5$ $(.625 \times 32)$

IEEE 754-2008 32 bit format

1	8	23
±	**exponent**	**significand**

- Many ways to encode 20_{10}
  - $10100. \times 2^0$
  - 1010.0×2^1
  - 101.00×2^2
  - 10.100×2^3
  - 1.0100×2^4 normalized $\Rightarrow 1 \leq N < 2$
  - $.10100 \times 2^5$

IEEE 754-2008 32 bit format

1	8	23
±	**exponent**	**significand**

- 1.0100×2^4 normalized $\Rightarrow 1 \leq N < 2$

- In normalized format the leading digit
  - Is always 1
  - So don't store it
- The significand of 20_{10} is

`010 0000 0000 0000 0000 0000`

IEEE 754-2008 32 bit format

1	8	23
±	**exponent**	**significand**

- 1.0100×2^4 normalized $\Rightarrow 1 \leq N < 2$

- In normalized format the leading digit
  - Is always 1
  - So don't store it
- The significand of 20_{10} is

`1.010 0`

`010 0000 0000 0000 0000 0000`

The hidden or implied 1 provides one more bit of precision

IEEE 754-2008 32 bit format

1	8	23
±	**exponent**	**significand**

$20_{10} = +1.01 \times 2^4$

```
[ 0 ] [ 127 + 4   ] [ .01                            ]
[ 0 ] [ 131       ] [ .01                            ]
[ 0 ] [ 1000 0011 ] [ 010 0000 0000 0000 0000 0000 ]
  0     1000 0011     010 0000 0000 0000 0000 0000
    0 1000 0011 010 0000 0000 0000 0000 0000
     0100 0001 1010 0000 0000 0000 0000 0000
       4    1    A    0    0    0    0    0
            41 A0 00 00
```

Floating point has two serious issues (1)

- Can only represent a limited number of decimal digits
  - Finite decimals are limited in precision as well
  - But different magnitudes
    - 8 decimal digits = 100,000,000
    - 8 binary digits = 256 = $100{,}000{,}000_2$ = $1\ 0000\ 0000_2$
    - 32 decimal digits = 100,000,000,000,000,000,000,000,000,000,000
    - 32 binary digits = 4,294,967,296
  - Single precision will handle approximately 7 - 8 decimal digits

Floating point has two serious issues (1)

- Every (finite) decimal number
  - Is rational
  - A fraction : `3.14159 = 314159 / 100000` $\neq \pi$
  - Cannot represent irrational number exactly
- Most binary numbers
  - Are approximations of rational decimals
  - Translation happens in both directions
  - $.333_{10} = 0.01010101_2$
  - $0.010101_2 = 0 + 1/4 + 1/16 + 1/64 + 1/256$
  - $= .250 + .06250 + .015625 + .003906$
  - $= .326831$

Floating point has two serious issues (1)

```
x dd 0.987654321   ; 9 digits

3F7CD6EA     ; hex constant created by C
```

Convert back to decimal

```
0.987654328     ; only 8 digits are correct
```

Floating point has two serious issues (1)

```
x dd 12345.987654321; 14 digits

4640E7F3        ; hex constant created by C
```

Convert back to decimal

```
12345.987304687   ; only 8 digits are correct
```

Floating point has two serious issues (2)

- Not all decimal fractions can be represented
  - Can be critical
  - `x dd 0.1 ; 1/10 cannot be represented in binary`
  - `3DCCCCCD ; 0.100000001`$_{10}$
  - It is very close but not exact
- Combination of
  - Limited precision and
  - Inaccurate representation
  - Is deadly combination

Data declaration	Hex constant created	Actual decimal value
a dd 1000.1	447A0666	1000.09997559
b dd 1000.0	447A0000	1000.0
c dd 0.1	3DCCCCCD	0.100000001

```
if ((a - b) == c) …

1000.09997559 - 1000.0 = 0.09997559

0.09997559 == 0.100000001
```

Data declaration	Hex constant created	Actual decimal value
a dd 1000.1	447A0666	1000.09997559
b dd 1000.0	447A0000	1000.0
c dd 0.1	3DCCCCCD	0.100000001

- Generally, don't try to test equality in floating point math
    - It's usually a bad idea to do this:

```
if ((a - b) == c) …
```

- It's generally a better idea to do this:

```
if (abs((a-b) - c) < epsilon) …
```

Special numbers

- Two numbers in exponent are reserved
    - 0
    - 255
- Because of hidden, implied leading 1
    - Cannot represent zero
    - Smallest number is 1.0×2^{-126}
    - Small but not zero
- Zero is defined as
    - Exponent & significand both equal to zero
    - Sign can be either ⇒ two representations of zero

Special numbers

- Exponent 255 is reserved
  - When significand equals zero (and exponent is 255)
  - Number is *infinity*
  - Can have plus and minus infinity
  - Two different numbers
  - … not two representations of the same number

Special numbers

- Some operations, like
  - 0/0
  - $0 \times \infty$
  - $\sqrt{-2}$
- Generate Not a Number (NaN)
  - Sign = +
  - Exponent = 255
  - Significand $\neq$ 0
- Program continues
  - But all operations having a NaN operator generate NaN as result

Floating point co-processor

Part 2

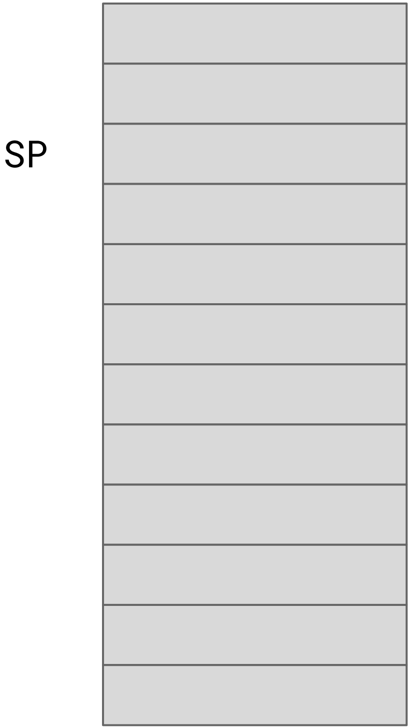
AKA Numeric data processor

Originally separate chip

Executes in parallel

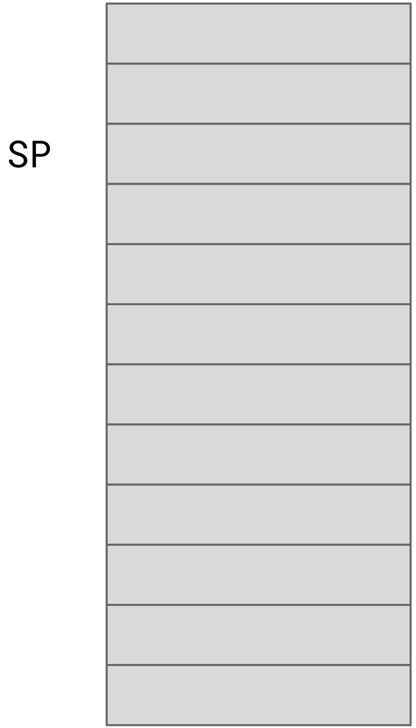
Floating point co-processor

- Stack
  - Passes all data to/from co-proc
  - Works like JVM
  - 32-bit words
  - Single-precision: 1 word
  - Double: 2 words
  - Quadruple: 4 words



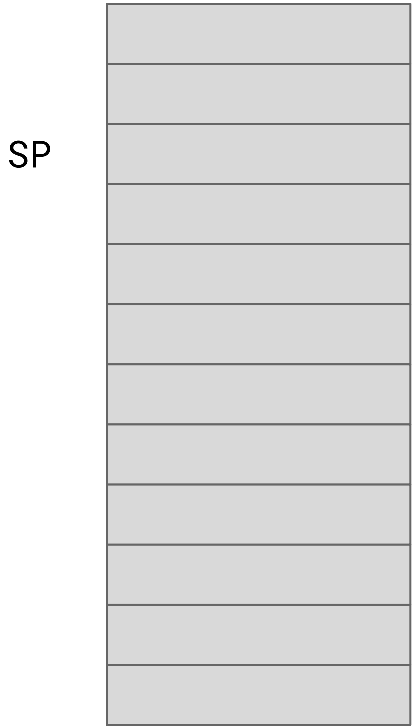
Data movement

- Push
  - dec SP
  - Insert data
- Pop
  - Remove data
  - inc SP
- Like x86 pop/push instructions



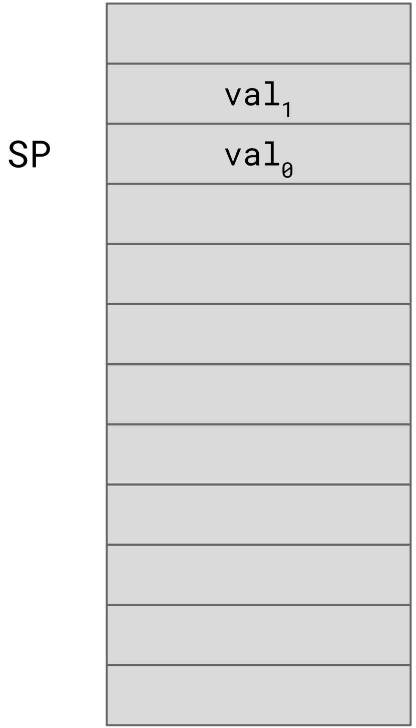
Data movement

Data movement	Explanation
`FLD [var]`	**Push** **var is pushed onto the stack**
`FSTP [var]`	**Pop** **Data popped off stack; put into var**



Arithmetic

Arithmetic	Explanation
FADD (add)	**push = $val_1 + val_0$** **top two words replace with their sum**
FMUL (multiply)	**push = $val_1 * val_0$** **top two words replaced with their product**
FSUB (subtract)	**push = $val_1 - val_0$** **top word subtracted from 2d from top difference pushed onto stack**
FDIV (divide)	**push = val_1 / val_0** **top word divided into 2d from top quotient pushed onto stack**

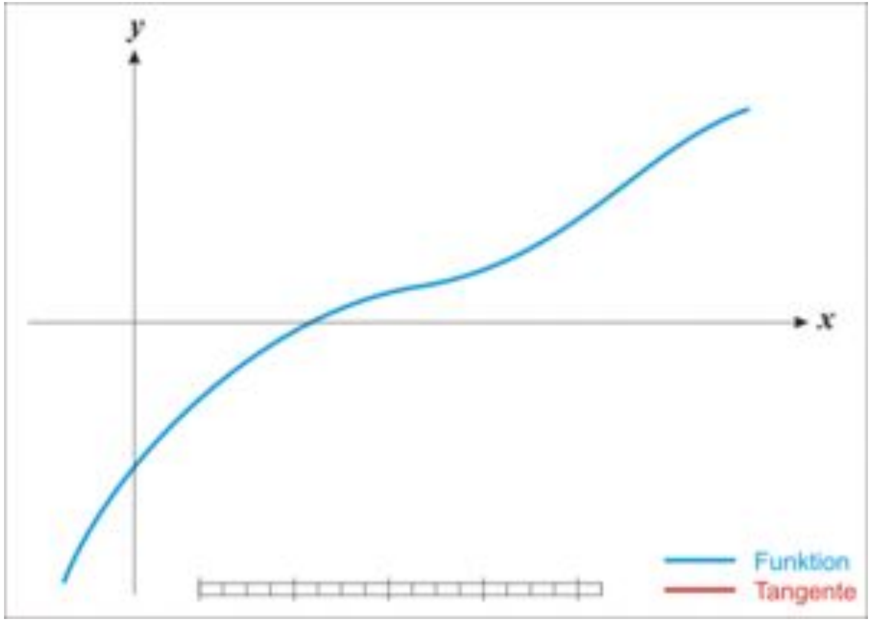


Newton's method for finding root of a function

- Newton's method for root
  - Root is when $f(x) == 0$
  - Guess x
  - Next guess $x^* = x - f(x)/f'(x)$
  - Stop when $x^* == x$
  - Well, when close enough

Newton's method for finding root of a function

- Newton's method for root
  - Root is when $f(x) = 0$
  - Guess x
  - Next guess $x^* = x - f(x)/f'(x)$
  - Stop when $x^* == x$
  - Well, when close enough



Newton's method for calculating sqrt

- Newton's method for root
  - Root is when $f(x) = 0$
  - Guess x
  - Next guess $x^* = x - f(x)/f'(x)$
  - Stop when $x^* == x$
  - Well, when close enough
- Square root of x
  - $f(x) = x^{½}$
  - $f'(x) = ½x^{-½}$
- But that requires finding the square root

Newton's method for calculating sqrt

- Newton's method for root
  - Root is when $f(x) = 0$
  - Guess x
  - Next guess $x^* = x - f(x)/f'(x)$
  - Stop when $x^* == x$
  - Well, when close enough
- Square root of x
  - $f(x) = x^{½}$
  - $f'(x) = ½x^{-½}$
- But that requires finding the square root

Not exactly what we want, but we're already in trouble here.

- Solve
  - Method will find x, st, $f(x) = 0$
  - Create function that will find sqrt(a)
  - $f(x) = x^2 - a$
  - When $f(x^*) = 0$
- Choosing the next guess
  - $f'(x) = 2x$
  - $x^* = x - f(x) / f'(x)$
  - $= x - (x^2 - a)/2x$
  - $= (2x^2 - x^2 + a)/2x$
  - $= (x + a/x)/2$

That's better. We can solve this

Coding Newton's

```
def sqrt(x):
    x0 = x/2            # initial guess
    print(x0)

    while True:
        x1 = (x0 + x/x0)/2
        print(x1)
        if abs(x1 - x0) < 0.001:
            break
        x0 = x1
    return x0
```

Coding Newton's

```
def sqrt(x):
    x0 = x/2            # initial guess
    print(x0)

    while True:
        x1 = (x0 + x/x0)/2
        print(x1)
        if abs(x1 - x0) < 0.001:
            break
        x0 = x1
    return x0
```

```
% python3 sqrt.py 16
8.0
5.0
4.1
4.001219512195122
4.0000001858445895
4.000000000000004
```

Coding Newton's

```
def sqrt(x):
    x0 = x/2                # initial guess
    print(x0)

    while True:
        x1 = (x0 + x/x0)/2
        print(x1)
        if abs(x1 - x0) < 0.001:
            break
        x0 = x1
    return x0
```

```
% python3 sqrt.py 144
72.0
37.0
20.445945945945947
13.744453475286258
12.110703489698958
12.000505968238837
12.000000010666378
```

Coding Newton's

```
def sqrt(x):
    x0 = x/2                # initial guess
    print(x0)

    while True:
        x1 = (x0 + x/x0)/2
        print(x1)
        if abs(x1 - x0) < 0.001:
            break
        x0 = x1
    return x0
```

```
% python3 sqrt.py 4096
2048.0
1025.0
514.4980487804878
261.22960314045366
138.4546481089779
84.01917126201654
66.38497483370875
64.04284181000048
64.00014329631
64.0000000000016
```

Coding Newton's

```
def sqrt(x):
    x0 = x/2            # initial guess
    print(x0)

    while True:
        x1 = (x0 + x/x0)/2
        print(x1)
        if abs(x1 - x0) < 0.001:
            break
        x0 = x1
    return x0
```

```
% python3 sqrt.py 34567890123456789
3.4567890123456788e+16
1.7283945061728394e+16
8641972530864198.0
…
207862730.8922772
187082130.31646848
185928002.55568016
185924420.4946944
185924420.46018803
185924420.46018803
```

33 steps

Coding Newton's

```
def sqrt(x):
    x0 = x/2            # initial guess
    print(x0)

    while True:
        x1 = (x0 + x/x0)/2
        print(x1)
        if abs(x1 - x0) < 0.001:
            break
        x0 = x1
    return x0
```

- Code in FP co-proc
- Simplify function at left

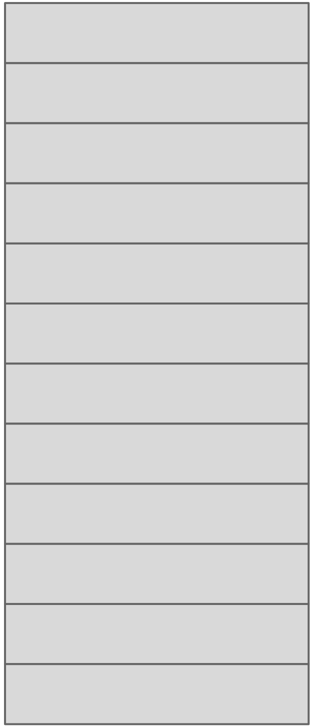
```
while (x1 != x0): x1 = (x0 + x/x0)/2
```

- Rename variables

```
Root: x ⇒ N
Previous guess: x0 ⇒ g
This guess: x1 ⇒ next

while (next != g): next = (g + N/g)/2
```

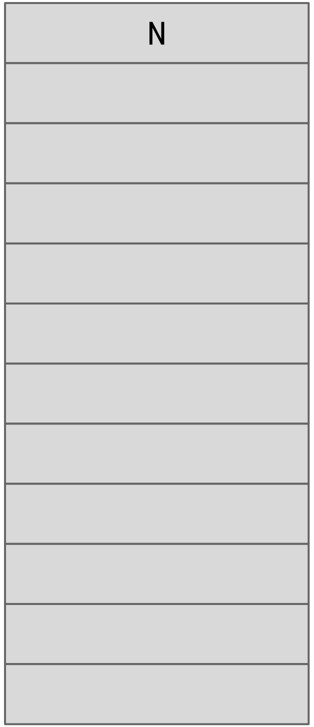
Coding Newton's

```
next = (g + N/g)/2
```



Coding Newton's

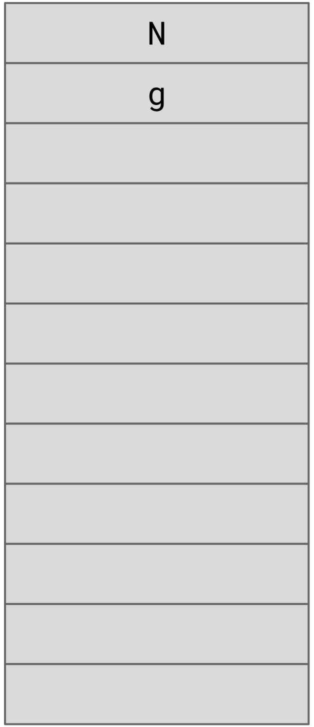
```
next = (g + N/g)/2

fld [N]       ;push [N]
```



Coding Newton's

```
next = (g + N/g)/2
```

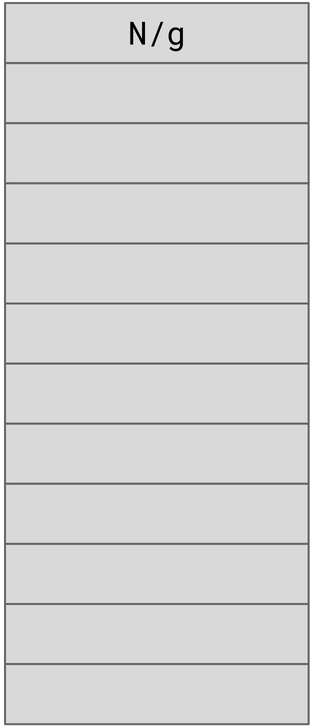
```
fld [N]       ;push [N]
fld [g]       ;push [g]
```



Coding Newton's

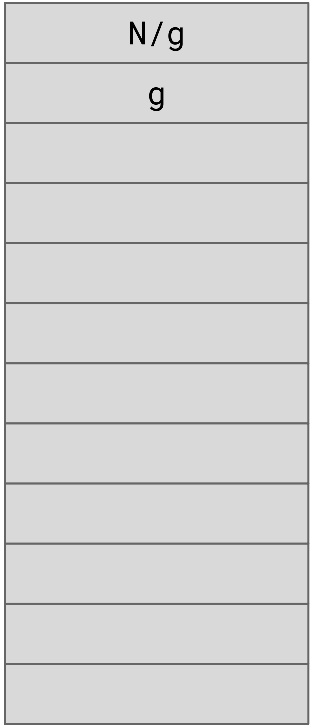
```
next = (g + N/g)/2
```

```
fld [N]        ;push [N]
fld [g]        ;push [g]
fdiv     ;N/g
```



Coding Newton's

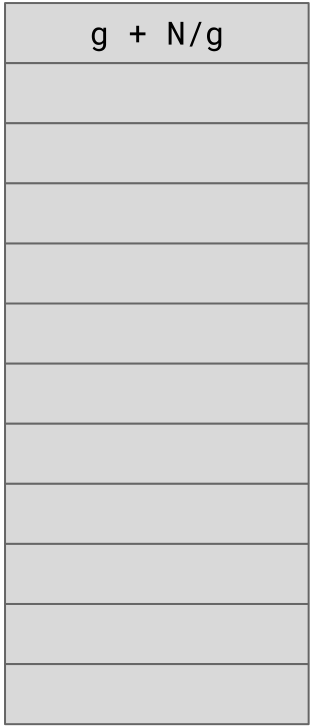
```
next = (g + N/g)/2

fld [N]      ;push [N]
fld [g]      ;push [g]
fdiv     ;N/g
fld [g]      ;push [g]
```



Coding Newton's

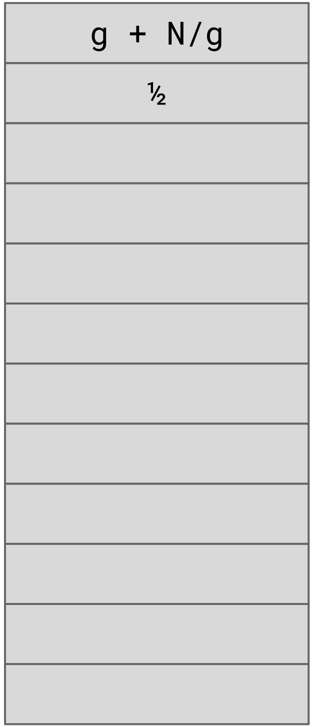
```
next = (g + N/g)/2
```

```
fld [N]       ;push [N]
fld [g]       ;push [g]
fdiv      ;N/g
fld [g]       ;push [g]
fadd
```



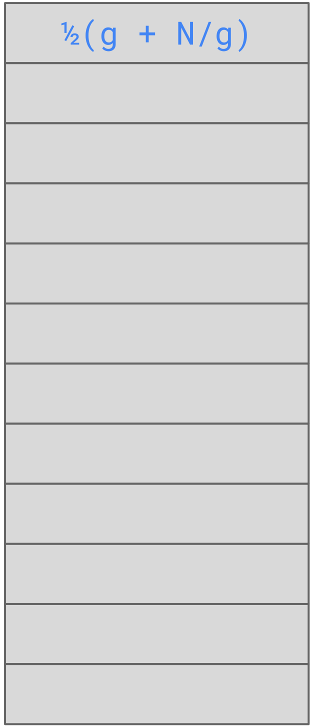
Coding Newton's

```
next = (g + N/g)/2

fld [N]       ;push [N]
fld [g]       ;push [g]
fdiv     ;N/g
fld [g]       ;push [g]
fadd
fld [half]    ;push ½
```



Coding Newton's

```
next = (g + N/g)/2

fld [N]       ;push [N]
fld [g]       ;push [g]
fdiv     ;N/g
fld [g]       ;push [g]
fadd
fld [half]    ;push ½
fmul
```



Coding Newton's

```
next = (g + N/g)/2

fld [N]       ;push [N]
fld [g]       ;push [g]
fdiv     ;N/g
fld [g]       ;push [g]
fadd
fld [half]    ;push ½
fmul
fstp [next]   ;store next guess
```