Lab3 CompArch

Agamjot Singh
EE24BTECH11002

September 21, 2025

Approach and Planning

Looking at the Taylor series approximation of the required functions, it contains x^n and $n!$ terms which calls for the need of `pow` and `fact` helper procedures.

The `pow` procedure is pretty straightforward, it accepts base (FP32) and exponent (INT64) and returns in FP32 format. Exponent is restricted to integer values as taylor series requires for that only.

```
# pow(fa0, a0)
# INPUTS
# fa0 := base (FP32)
# a0 := exponent (INT64)
# OUTPUTS
# fa0 := fa0 ^ a0 (FP32)
pow:
    li t0, 1
    fcvt.s.l ft0, t0 # init to 1 (in float)

    pow_loop:
        bge x0, a0, pow_ret
        fmul.s ft0, ft0, fa0 # repeatedly multiply with fa0
        addi a0, a0, -1
        jal x0, pow_loop

    pow_ret:
        fmv.w.x ft1, x0
        fadd.s fa0, ft1, ft0 # move ft0 to fa0 by adding zero
        ret
```

The `fact` procedure accepts input (INT64) and returns it factorial (INT64). INT64 is used to increase support for higher order factorials like 20! which wouldn't fit on a 32 bit integer.

```
# fact(a0)
# INPUTS
# a0 := input (INT64)
# OUTPUTS
# a0 := a0! (INT64)
fact:
    blt a0, x0, nan_error
    li t0, 1

    fact_loop:
        bge x0, a0, fact_ret
        mul t0, t0, a0 # repeatedly multiply decremented a0
        addi a0, a0, -1 # decrement a0
        jal x0, fact_loop

    fact_ret:
        mv a0, t0
        ret
```

Now we have to use these helper procedures to actually implement the taylor series. We will take the example of taylor series for exp function.

$$\exp x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

Taylor Series Example: Exponential Function

Data: fa0 := input value x (FP32); a0 := number of terms (INT32)
Result: fa0 (FP32) := exp(fa0) computed to a0 terms of the taylor series

```
// Error Handling
if a0 ≤ 0 then
    return NaN // Invalid number of terms
end

Save registers s0, fs0, fs1 to stack
s0 ← 1 // Counter for taylor series terms = n
fs0 ← 1.0 // Result which is initialized to 1 (first term)
a0 ← a0 - 1 // First term considered already

while a0 > 0 do
    Save fa0, a0, ra to stack for calling other procedures

    // Calculate n!
    fs1 ← fact(s0)

    // Calculate x^n
    fa0 ← pow(fa0, s0)

    fa0 ← fa0/fs1 // pow(fa0, s0)/fact(s0) = x^n/n!

    fs0 ← fs0 + fa0
    Load fa0, a0, ra back from stack
    a0 ← a0 - 1
    s0 ← s0 + 1 // Increment n

end

fa0 ← fs0 // move result to return register

Restore saved registers s0, fs0, fs1 from stack
return fa0
```

Modularity is taken into account here, and as per ABI regulations, the saved registers have been saved and the procedure can be independently called.

Major tradeoff: A pretty big tradeoff taken here is splitting the work into `pow` and `fact` helper functions. This is against the much more performant way of just multiplying in the taylor series procedure loop itself and storing it in a variable, thereby reducing the number of instructions by A LOT.

However, this tradeoff is taken to make the code **modular** and much more easier to understand.

After all these functions have been created for everything, we compile it in a main functions which has a **switch case** implementation.

```
lui x3, 0x10000
lw s0, 0(x3) # Number of inputs
addi s1, x3, 4 # Mem Location for reading values
addi s2, x3, 512 # Mem Location for storing values

main:
    bge x0, s0, exit # if s0 <= 0, exit

    lw s3, 0(s1) # function code
    flw fa0, 4(s1) # function input (x)
    lw a0, 8(s1) # number of terms

    # Switch case, s3 := code
    case1:
        bne s3, x0, case2
        jal x1, exp # code 0 for exp

    case2:
        addi s3, s3, -1
        bne s3, x0, case3
        jal x1, sin # code 1 for sin

    case3:
        addi s3, s3, -1
        bne s3, x0, case4
        jal x1, cos # code 2 for cos

    case4:
        addi s3, s3, -1
        bne s3, x0, case5
        jal x1, ln # code 3 for ln

    case5:
        addi s3, s3, -1
        bne s3, x0, default
        jal x1, reciprocal # code 4 for 1/x

    default:
        # 'if any of the previous cases
        # are satsfied then store
        bge x0, s3, store
        jal x1, nan_error # error, code not valid

    store:
        fsw fa0, 0(s2)

    addi s2, s2, 4
    addi s1, s1, 12
    addi s0, s0, -1
    beq x0, x0, main

exit:
    jal x0, exit
```

This switch case implementation is pretty simple, you just decrement the register (s3 in this case) and check when its zero. For example, if s3 = 2, it will go to zero in the `case3` Loop and cos will be called. Please note that the labels are one indexed but in our functions are zero indexed.

Implementation

Exp

The general outline was discussed above, the code for the following is given below. The code has been split into blocks wherever necessary.

Please note that, by default, the stack pointer in RISC V has to be **16 bit aligned**. But when I pull down stack by 16 bits in the simulator, it refused to work and only works in some cases. Pulling the stack down by 32 bits however works everywhere. This issue could arise because of the misalignment in the initialization of the stack pointer. This is further discussed in the **Issued faced** section.

Sin

Sin is given by,

$$\sin x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{(2n+1)}}{(2n+1)!}$$
$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} \dots$$

The first term = x, and then the power of x and factorial is incremented by 2 in each iteration of the loop with alternating sign.

Tradeoff: A minor tradeoff chosen here is to keep an extra register for the alternating plus minus sign. Another way to do it would be to manipulate $(2n+1)$ and then use the `pow` helper function to do it and then multiply it, which would not be that efficient as `pow` takes a lot of iterations in itself.
Another way to do it would be to keep a float register which we will multiply by -1 in each iteration, but then again it requires the use of one extra `fmul` instruction per loop.

Taylor Series: Sin

```
Data: fa0 := input value x (FP32); a0 := number of terms (INT32)
Result: fa0 (FP32) := exp(fa0) computed to a0 terms of the taylor series

// Error Handling
if a0 ≤ 0 then
|   return NaN // Invalid number of terms
end

Save registers s0, s1, fs0, fs1 to stack
s0 ← 3 // Counter for taylor series terms = 2*n + 1
// NOTE: the counter starts from 3 as we already are
   considering the first term below
fs0 ← fa0 // Result which is initialized to x (first term)
a0 ← a0 - 1 // First term considered already
s1 ← 1 // Register to control alternate plus minus

while a0 > 0 do
|   Save fa0, a0, ra to stack for calling other procedures
|
|   // Calculate (2n + 1)!
|   fs1 ← fact(s0)
|
|   // Calculate x^(2n+1)
|   fa0 ← pow(fa0, s0)
|
|   s1 ← s1 ⊕(−2)
|
|   // NOTE: switching the sign of s1 involves xor with -2 (-2
|      = 0b11111.....1110).  xor with -2 makes it switch from 1
|      to -1 and -1 to 1
|
|   fa0 ← (fa0*s1)/fs1 // pow(fa0, s0)/fact(s0) = x^(2n+1)/(2n + 1)!
|
|   fs0 ← fs0 + fa0
|   Load fa0, a0, ra back from stack
|   a0 ← a0 - 1
|   s0 ← s0 + 2 // Increment s0 = (2n + 1) by 2 as it counts 1,
|      3, 5, 7 ...
|
end
fa0 ← fs0 // move result to return register

Restore saved registers s0, s1, fs0, fs1 from stack
return fa0
```

Cos

$$\cos x = \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n}}{(2n)!}$$
$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} \dots$$

The first term $= x^0 = 1$, and then the power of x and factorial is incremented by 2 in each iteration of the loop with alternating sign.

Tradeoff: A minor tradeoff chosen here is to keep an extra register for the alternating plus minus sign. Another way to do it would be to manipulate $(2n+1)$ and then use the `pow` helper function to do it and then multiply it, which would not be that efficient as `pow` takes a lot of iterations in itself.
Another way to do it would be to keep a float register which we will multiply by -1 in each iteration, but then again it requires the use of one extra `fmul` instruction per loop. This is the same tradeoff taken as in the case of `sin`.

Taylor Series: Cos

```
Data: fa0 := input value x (FP32); a0 := number of terms (INT32)
Result: fa0 (FP32) := exp(fa0) computed to a0 terms of the taylor series

// Error Handling
if a0 ≤ 0 then
    return NaN // Invalid number of terms
end

Save registers s0, s1, fs0, fs1 to stack
s0 ← 2 // Counter for taylor series terms = 2n
// NOTE: the counter starts from 2 as we already are
   considering the first term below
fs0 ← 1 // Result which is initialized to x (first term)
a0 ← a0 - 1 // First term considered already
s1 ← 1 // Register to control alternate plus minus

while a0 > 0 do
    Save fa0, a0, ra to stack for calling other procedures

    // Calculate (2n)!
    fs1 ← fact(s0)

    // Calculate x^(2n)
    fa0 ← pow(fa0, s0)

    s1 ← s1 ⊕(−2)

    // NOTE: switching the sign of s1 involves xor with -2 (-2
       = 0b11111.....1110).  xor with -2 makes it switch from 1
       to -1 and -1 to 1

    fa0 ← (fa0*s1)/fs1 // pow(fa0, s0)/fact(s0) = x^(2n)/(2n)!

    fs0 ← fs0 + fa0
    Load fa0, a0, ra back from stack
    a0 ← a0 - 1
    s0 ← s0 + 2 // Increment s0 = (2n) by 2 as it counts 0, 2, 4,
       6 ...

end
fa0 ← fs0 // move result to return register

Restore saved registers s0, s1, fs0, fs1 from stack
return fa0
```

Ln

$$\ln{(1+x)} = \sum_{n=1}^{\infty} (-1)^{(n+1)} \frac{x^n}{n}$$

$$\ln{(1+x)} = x - \frac{x^2}{2} + \frac{x^3}{3} \dots$$

Transforming $x \to x - 1$,

$$\ln{(x)} = \sum_{n=1}^{\infty} (-1)^{(n+1)} \frac{(x-1)^n}{n}$$

Note that for `ln`, domain has to be considered in the error checking. $x > 0$ for `ln` to be defined. So if $x \leq 0$, `NaN` will be returned.

Convergence check: For $\ln(1 + x)$, the taylor series converges only for $x \in [-1, 1]$. After the transformation $x \to x - 1$, $x \in [0, 2]$ is the region of convergence of ln.

Taylor Series: Ln

Data: fa0 := input value x (FP32); a0 := number of terms (INT32)
Result: fa0 (FP32) := ln(fa0) computed to a0 terms of the taylor series

```
// Error Handling
if a0 ≤ 0 then
    return NaN // Invalid number of terms
end

if fa0 ≤ 0 then
    return NaN // Domain error, ln requires positive input
end

Save registers s0, s1, fs0 to stack

// Transform input for ln(1+x) series
fa0 ← fa0 - 1 // Since we use ln(1+x) = x − x^2/2 + x^3/3 − x^4/4+ ...

s0 ← 2 // Counter for taylor series terms = n
// NOTE: the counter starts from 2 as we already are
   considering the first term below
fs0 ← fa0 // Result which is initialized to x (first term)
a0 ← a0 - 1 // First term considered already
s1 ← 1 // Register to control alternate plus minus

while a0 > 0 do
    Save fa0, a0, ra to stack for calling other procedures

    // Calculate x^n
    fa0 ← pow(fa0, s0)

    s1 ← s1 ⊕(−2)

    // NOTE: switching the sign of s1 involves xor with -2 (-2
       = 0b11111.....1110).  xor with -2 makes it switch from 1
       to -1 and -1 to 1

    fa0 ← (fa0*s1)/s0 // pow(fa0, s0)/s0 = x^n/n

    fs0 ← fs0 + fa0
    Load fa0, a0, ra back from stack
    a0 ← a0 - 1
    s0 ← s0 + 1 // Increment s0 = n by 1 as it counts 2, 3, 4, 5
        ...
end
fa0 ← fs0 // move result to return register

Restore saved registers s0, s1, fs0 from stack
return fa0
```

Reciprocal a.k.a Inverse $(1/x)$

$$
\begin{aligned}
\frac{1}{x} &= \frac{1}{1-(1-x)} \\
&= \sum_{n=0}^{\infty}(1-x)^n \\
&= 1 + (1-x) + (1-x)^2 + (1-x)^3 + \cdots
\end{aligned}
$$

Here the transformation made is $x \to 1 - x$.

Note that for the reciprocal function, the domain excludes $x = 0$ where the function is undefined. So if $x = 0$, `NaN` will be returned.

Convergence check: For the geometric series $\frac{1}{1+u}$, convergence occurs when $|u| < 1$. After substituting $u = x - 1$, this means $|x - 1| < 1$, which gives us the convergence region $x \in (0, 2)$.

Tradeoff: A tradeoff chosen is that transformation of $x \to 1 - x$ on the taylor series of $\dfrac{1}{1-x}$ instead of transformation of $x \to x - 1$ on the taylor series of $\dfrac{1}{1+x}$. This is taken because $\dfrac{1}{1+x}$ taylor series contains alternate plus minus signs, which just adds redundant extra code.

$$
\begin{aligned}
\frac{1}{x} &= \frac{1}{1+(x-1)} \\
&= \sum_{n=0}^{\infty}(-1)^n(x-1)^n \\
&= 1 - (x-1) + (x-1)^2 - (x-1)^3 + \cdots
\end{aligned}
$$

```
Algorithm 5: Taylor Series: Reciprocal
Data: fa0 := input value x (FP32); a0 := number of terms (INT32)
Result: fa0 (FP32) := 1/fa0 computed to a0 terms of the taylor series

// Error Handling
if a0 ≤ 0 then
  | return NaN // Invalid number of terms
end
if fa0 = 0 then
  | return NaN // Division by zero error
end

Save registers s0, fs0 to stack

// Transform input for 1/(1-x) series
fa0 ← 1 - fa0 // Since we use 1/(1-x) = 1 + x + x² + x³ + ...

s0 ← 1 // Counter for taylor series terms = n
// NOTE: the counter starts from 1 as we already are
   considering the first term below
fs0 ← 1 // Result which is initialized to 1 (first term)
a0 ← a0 - 1 // First term considered already

while a0 > 0 do
  | Save fa0, a0, ra to stack for calling other procedures
  |
  | // Calculate x^n
  | fa0 ← pow(fa0, s0)
  |
  | fs0 ← fs0 + fa0 // Add x^n term to result
  | Load fa0, a0, ra back from stack
  | a0 ← a0 - 1
  | s0 ← s0 + 1 // Increment s0 = n by 1 as it counts 1, 2, 3, 4
  |     ...
end
fa0 ← fs0 // move result to return register

Restore saved registers s0, fs0 from stack
return fa0
```

Verification Approach

We will use python to implement the taylor series and compare the results (taylor series and actual).

The following things will be verified for each function,

1. Some general values for checking
2. Increasing number of terms for convergence check

3. Edge cases inlcuding larger values/terms to show failure of taylor series in FP32 because of size limitations, as well as out of convergence values

Python code can be found at
`https://github.com/agamjotsingh1/CS2323/blob/main/labs/lab3/test_script.py`

Exp

```
--- EXP TESTCASES ---
exp(0.0), 5 terms := Taylor FP32 Value = 0x3f800000 = 1.0 | Function value = 1.0
exp(1.0), 8 terms := Taylor FP32 Value = 0x402df7e0 = 2.7182540893554688 | Function value = 2.7182819843292236
exp(0.5), 6 terms := Taylor FP32 Value = 0x3fd30889 = 1.6486979722976685 | Function value = 1.6487212181091309
exp(-0.5), 7 terms := Taylor FP32 Value = 0x3f1b45b1 = 0.6065321564674377 | Function value = 0.6065306663513184
exp(2.0), 15 terms := Taylor FP32 Value = 0x40ec7327 = 7.38905668258667 | Function value = 7.3890557289123535
```

Figure 1: Generic Testcases

VM Memory Dump

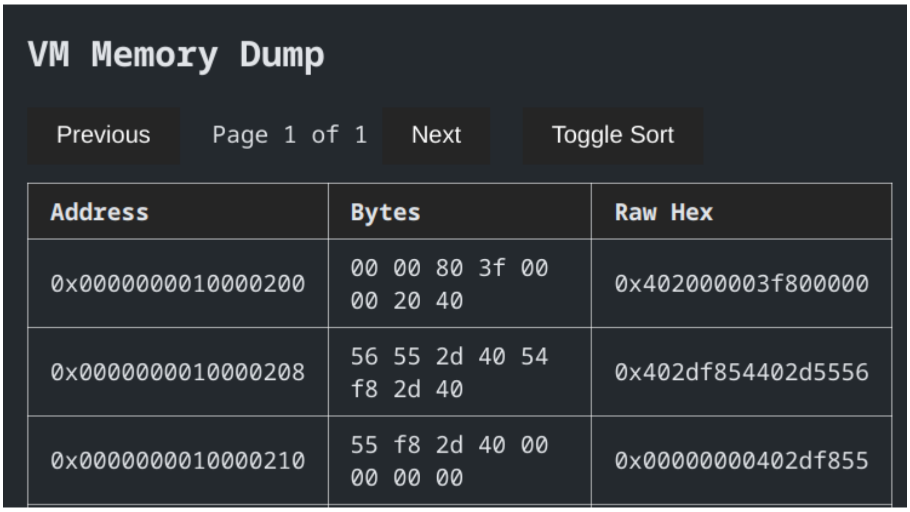
Previous   Page 1 of 1   Next   Toggle Sort

Address	Bytes	Raw Hex
0x0000000010000200	00 00 80 3f e0 f7 2d 40	0x402df7e03f800000
0x0000000010000208	89 08 d3 3f b1 45 1b 3f	0x3f1b45b13fd30889
0x0000000010000210	27 73 ec 40 00 00 00 00	0x0000000040ec7327

Figure 2: Generic Testcases Verification: VM memory dump

```
--- EXP CONVERGENCE TESTCASES ---
exp(1.0), 1 terms := Taylor FP32 Value = 0x3f800000 = 1.0 | Function value = 2.7182819843292236
exp(1.0), 3 terms := Taylor FP32 Value = 0x40200000 = 2.5 | Function value = 2.7182819843292236
exp(1.0), 5 terms := Taylor FP32 Value = 0x402d5556 = 2.7083334922790527 | Function value = 2.7182819843292236
exp(1.0), 10 terms := Taylor FP32 Value = 0x402df854 = 2.7182817459106445 | Function value = 2.7182819843292236
exp(1.0), 15 terms := Taylor FP32 Value = 0x402df855 = 2.7182819843292236 | Function value = 2.7182819843292236
```

Figure 3: Convergence Testcases

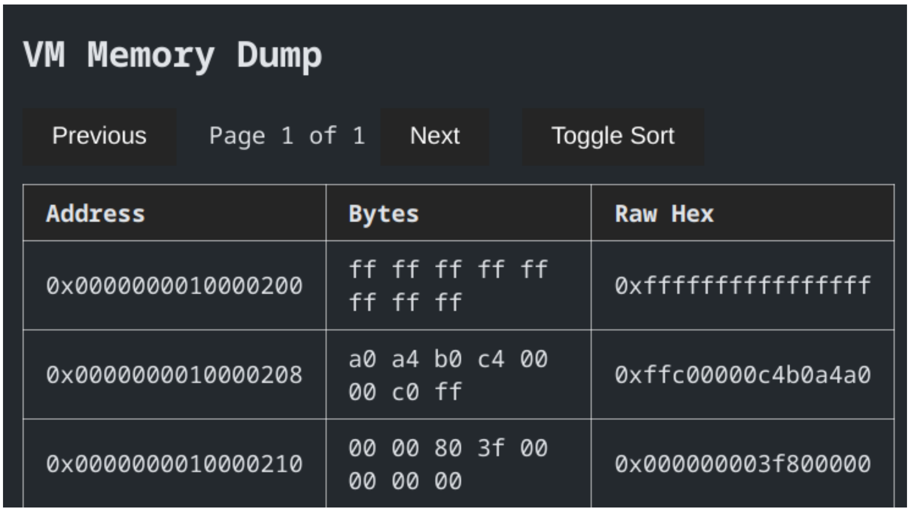


Address	Bytes	Raw Hex
0x0000000010000200	00 00 80 3f 00 00 20 40	0x402000003f800000
0x0000000010000208	56 55 2d 40 54 f8 2d 40	0x402df854402d5556
0x0000000010000210	55 f8 2d 40 00 00 00 00	0x00000000402df855

Figure 4: Convergence Testcases Verification: VM memory dump

```
--- EXP EDGE TESTCASES ---
exp(0.0), 0 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 1.0
exp(1.0), -1 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 2.7182818284590452
exp(-10.0), 10 terms := Taylor FP32 Value = 0xc4b0a4a0 = -1413.14453125 | Function value = 4.5399929762484854e-05
exp(100.0), 30 terms := Taylor FP32 Value = 0xffc00000 = nan | Function value = inf
exp(0.0), 40 terms := Taylor FP32 Value = 0x3f800000 = 1.0 | Function value = 1.0
```

Figure 5: Edge Testcases



Address	Bytes	Raw Hex
0x0000000010000200	ff ff ff ff ff ff ff ff	0xffffffffffffffff
0x0000000010000208	a0 a4 b0 c4 00 00 c0 ff	0xffc00000c4b0a4a0
0x0000000010000210	00 00 80 3f 00 00 00 00	0x000000003f800000

Figure 6: Edge Testcases Verification: VM memory dump

Sin

```
--- SIN TESTCASES ---
sin(0.0), 5 terms := Taylor FP32 Value = 0x00000000 = 0.0 | Function value = 0.0
sin(0.5235987901687622), 8 terms := Taylor FP32 Value = 0x3f000000 = 0.5 | Function value = 0.5
sin(0.7853981852531433), 6 terms := Taylor FP32 Value = 0x3f3504f3 = 0.7071067690849304 | Function value = 0.7071067690849304
sin(1.0471975803375244), 7 terms := Taylor FP32 Value = 0x3f5db3d8 = 0.866025447845459 | Function value = 0.866025447845459
sin(1.5707963705062866), 15 terms := Taylor FP32 Value = 0x3f7fffff = 0.9999999403953552 | Function value = 1.0
```

Figure 7: Generic Testcases

VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

Address	Bytes	Raw Hex
0x0000000010000200	00 00 00 00 00 00 00 3f	0x3f00000000000000
0x0000000010000208	f3 04 35 3f d8 b3 5d 3f	0x3f5db3d83f3504f3
0x0000000010000210	ff ff 7f 3f 00 00 00 00	0x000000003f7fffff

Figure 8: Generic Testcases Verification: VM memory dump

```
--- SIN CONVERGENCE TESTCASES ---
sin(1.0), 1 terms := Taylor FP32 Value = 0x3f800000 = 1.0 | Function value = 0.8414710164070129
sin(1.0), 3 terms := Taylor FP32 Value = 0x3f577777 = 0.8416666388511658 | Function value = 0.8414710164070129
sin(1.0), 5 terms := Taylor FP32 Value = 0x3f576aa4 = 0.8414709568023682 | Function value = 0.8414710164070129
sin(1.0), 10 terms := Taylor FP32 Value = 0x3f576aa4 = 0.8414709568023682 | Function value = 0.8414710164070129
sin(1.0), 15 terms := Taylor FP32 Value = 0x3f576aa4 = 0.8414709568023682 | Function value = 0.8414710164070129
```

Figure 9: Convergence Testcases

VM Memory Dump

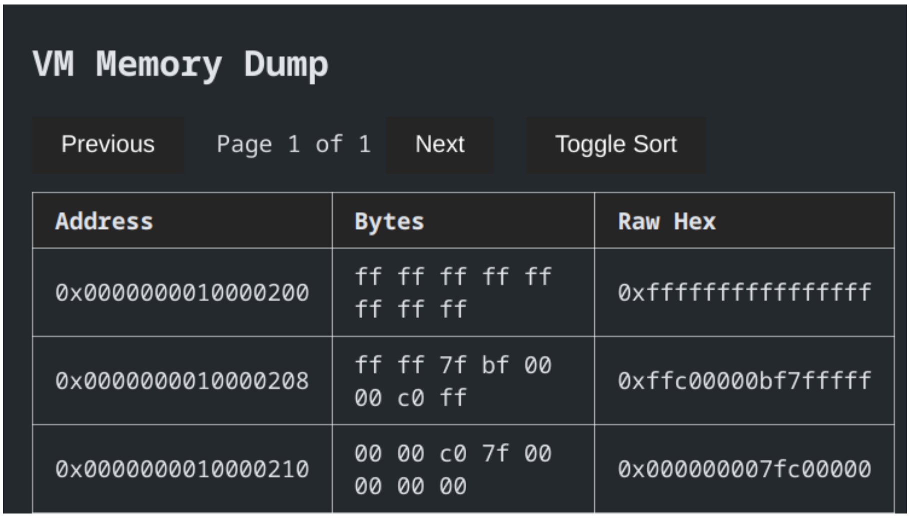
Previous Page 1 of 1 Next Toggle Sort

Address	Bytes	Raw Hex
0x0000000010000200	00 00 80 3f 77 77 57 3f	0x3f5777773f800000
0x0000000010000208	a4 6a 57 3f a4 6a 57 3f	0x3f576aa43f576aa4
0x0000000010000210	a4 6a 57 3f 00 00 00 00	0x000000003f576aa4

Figure 10: Convergence Testcases Verification: VM memory dump

```
--- SIN EDGE TESTCASES ---
sin(0.0), 0 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 0.0
sin(1.0), -1 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 0.8414710164070129
sin(-1.5707963705062866), 10 terms := Taylor FP32 Value = 0xbf7fffff = -0.9999999403953552 | Function value = -1.0
sin(6.2831854820251465), 30 terms := Taylor FP32 Value = 0xffc00000 = nan | Function value = 1.7484555314695172e-07
sin(0.0), 40 terms := Taylor FP32 Value = 0xffc00000 = nan | Function value = 0.0
```

Figure 11: Edge Testcases



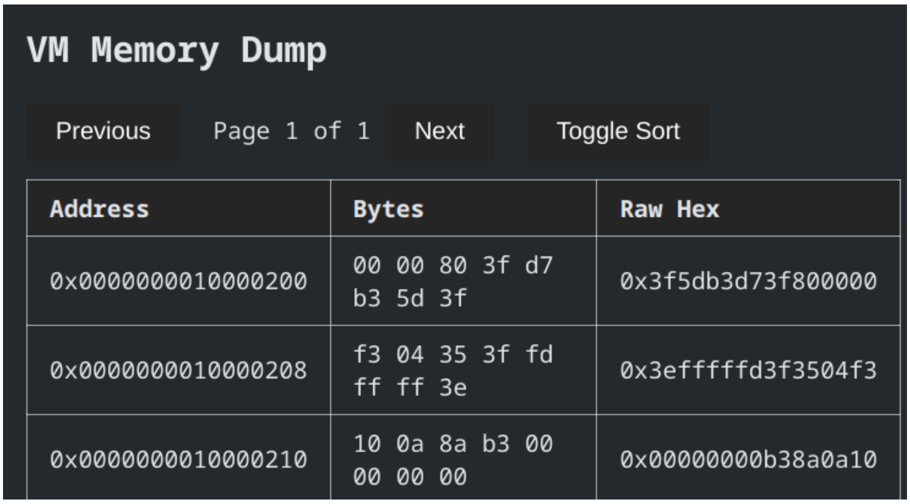
Address	Bytes	Raw Hex
0x0000000010000200	ff ff ff ff ff ff ff ff	0xffffffffffffffff
0x0000000010000208	ff ff 7f bf 00 00 c0 ff	0xffc00000bf7fffff
0x0000000010000210	00 00 c0 7f 00 00 00 00	0x000000007fc00000

Figure 12: Edge Testcases Verification: VM memory dump

Cos

```
--- COS TESTCASES ---
cos(0.0), 5 terms := Taylor FP32 Value = 0x3f800000 = 1.0 | Function value = 1.0
cos(0.5235987901687622), 8 terms := Taylor FP32 Value = 0x3f5db3d7 = 0.8660253882408142 | Function value = 0.8660253882408142
cos(0.7853981852531433), 6 terms := Taylor FP32 Value = 0x3f3504f3 = 0.7071067690849304 | Function value = 0.7071067690849304
cos(1.0471975803375244), 7 terms := Taylor FP32 Value = 0x3efffffd = 0.49999991059303284 | Function value = 0.4999999701976776
cos(1.5707963705062866), 15 terms := Taylor FP32 Value = 0xb38a0a10 = -6.42795612293412e-08 | Function value = -4.371138828673793e-08
```

Figure 13: Generic Testcases

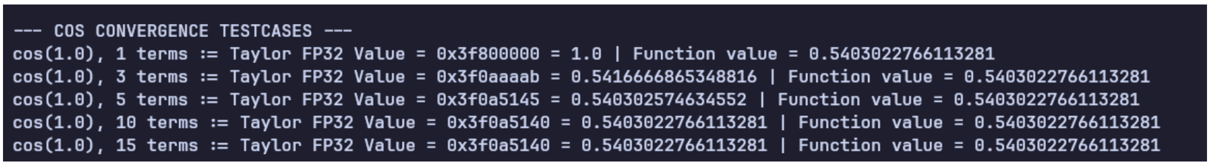


VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

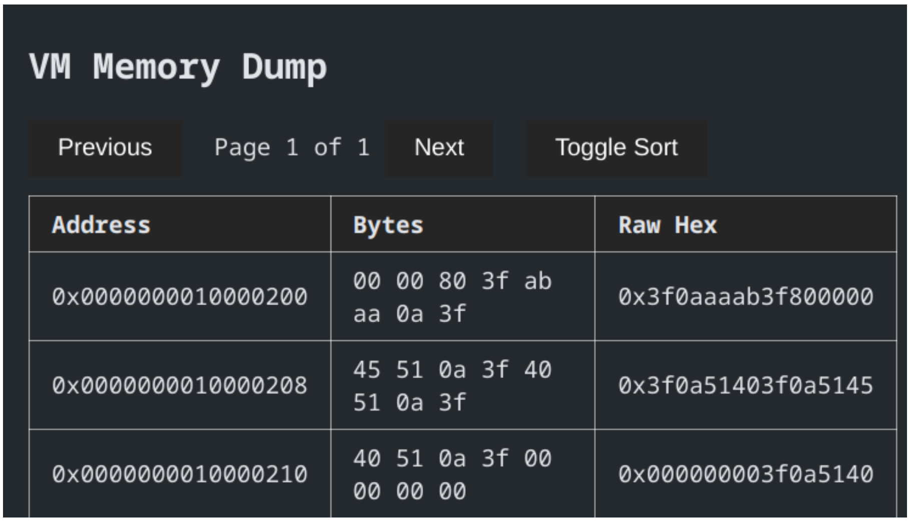
Address	Bytes	Raw Hex
0x0000000010000200	00 00 80 3f d7 b3 5d 3f	0x3f5db3d73f800000
0x0000000010000208	f3 04 35 3f fd ff ff 3e	0x3efffffd3f3504f3
0x0000000010000210	10 0a 8a b3 00 00 00 00	0x00000000b38a0a10

Figure 14: Generic Testcases Verification: VM memory dump



```
--- COS CONVERGENCE TESTCASES ---
cos(1.0), 1 terms := Taylor FP32 Value = 0x3f800000 = 1.0 | Function value = 0.5403022766113281
cos(1.0), 3 terms := Taylor FP32 Value = 0x3f0aaaab = 0.5416666865348816 | Function value = 0.5403022766113281
cos(1.0), 5 terms := Taylor FP32 Value = 0x3f0a5145 = 0.540302574634552 | Function value = 0.5403022766113281
cos(1.0), 10 terms := Taylor FP32 Value = 0x3f0a5140 = 0.5403022766113281 | Function value = 0.5403022766113281
cos(1.0), 15 terms := Taylor FP32 Value = 0x3f0a5140 = 0.5403022766113281 | Function value = 0.5403022766113281
```

Figure 15: Convergence Testcases

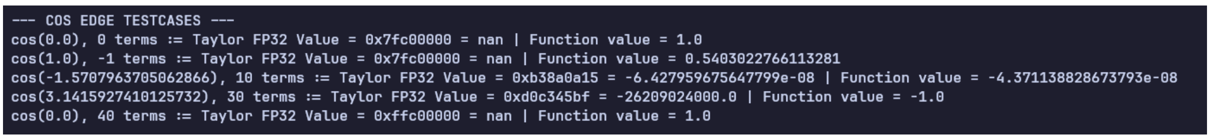


VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

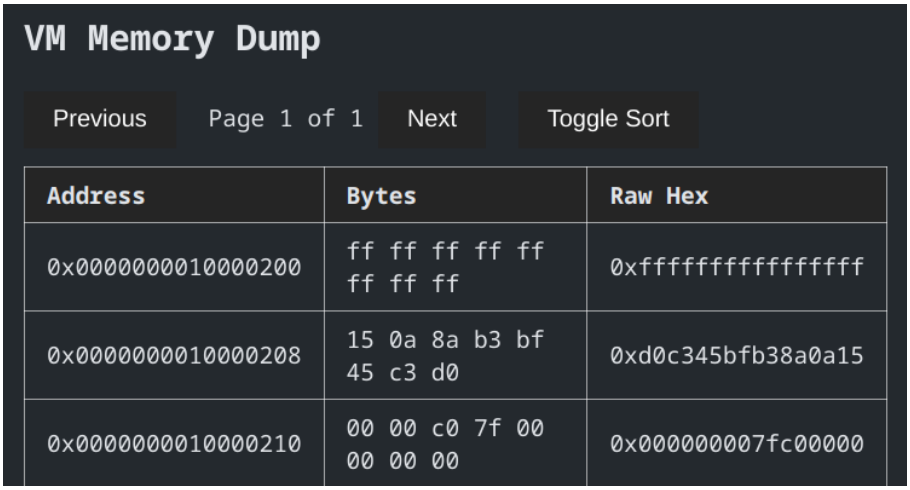
Address	Bytes	Raw Hex
0x0000000010000200	00 00 80 3f ab aa 0a 3f	0x3f0aaaab3f800000
0x0000000010000208	45 51 0a 3f 40 51 0a 3f	0x3f0a51403f0a5145
0x0000000010000210	40 51 0a 3f 00 00 00 00	0x000000003f0a5140

Figure 16: Convergence Testcases Verification: VM memory dump



```
--- COS EDGE TESTCASES ---
cos(0.0), 0 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 1.0
cos(1.0), -1 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 0.5403022766113281
cos(-1.5707963705062866), 10 terms := Taylor FP32 Value = 0xb38a0a15 = -6.427959675647799e-08 | Function value = -4.371138828673793e-08
cos(3.1415927410125732), 30 terms := Taylor FP32 Value = 0xd0c345bf = -26209024000.0 | Function value = -1.0
cos(0.0), 40 terms := Taylor FP32 Value = 0xffc00000 = nan | Function value = 1.0
```

Figure 17: Edge Testcases



VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

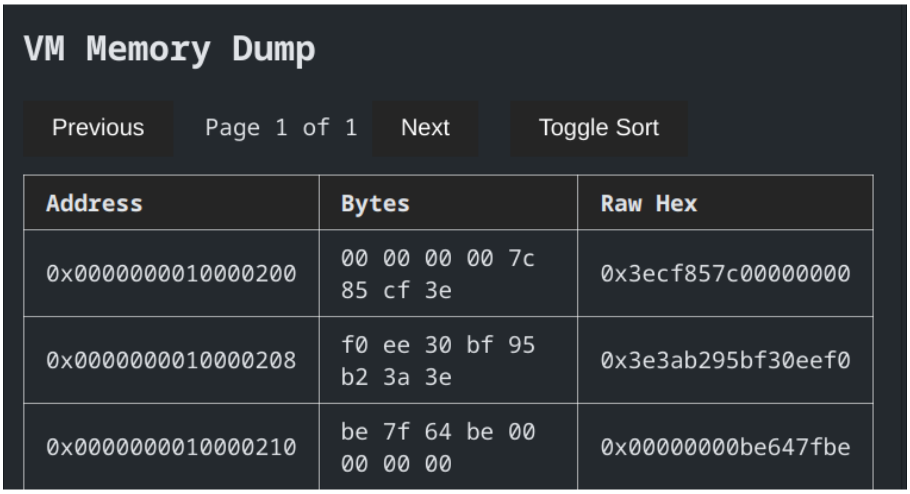
Address	Bytes	Raw Hex
0x0000000010000200	ff ff ff ff ff ff ff ff	0xffffffffffffffff
0x0000000010000208	15 0a 8a b3 bf 45 c3 d0	0xd0c345bfb38a0a15
0x0000000010000210	00 00 c0 7f 00 00 00 00	0x000000007fc00000

Figure 18: Edge Testcases Verification: VM memory dump

Ln

```
--- LN TESTCASES ---
ln(1.0), 5 terms := Taylor FP32 Value = 0x00000000 = 0.0 | Function value = 0.0
ln(1.5), 8 terms := Taylor FP32 Value = 0x3ecf857c = 0.4053152799606323 | Function value = 0.40546509623527527
ln(0.5), 6 terms := Taylor FP32 Value = 0xbf30eef0 = -0.6911458969116211 | Function value = -0.6931471824645996
ln(1.2000000476837158), 7 terms := Taylor FP32 Value = 0x3e3ab295 = 0.18232186138629913 | Function value = 0.18232160806655884
ln(0.800000011920929), 15 terms := Taylor FP32 Value = 0xbe647fbe = -0.2231435477733612 | Function value = -0.2231435328722
```

Figure 19: Generic Testcases



VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

Address	Bytes	Raw Hex
0x0000000010000200	00 00 00 00 7c 85 cf 3e	0x3ecf857c00000000
0x0000000010000208	f0 ee 30 bf 95 b2 3a 3e	0x3e3ab295bf30eef0
0x0000000010000210	be 7f 64 be 00 00 00 00	0x00000000be647fbe

Figure 20: Generic Testcases Verification: VM memory dump

```
--- LN CONVERGENCE TESTCASES ---
ln(1.5), 1 terms := Taylor FP32 Value = 0x3f000000 = 0.5 | Function value = 0.40546509623527527
ln(1.5), 3 terms := Taylor FP32 Value = 0x3ed55555 = 0.4166666567325592 | Function value = 0.40546509623527527
ln(1.5), 5 terms := Taylor FP32 Value = 0x3ed08888 = 0.4072916507720947 | Function value = 0.40546509623527527
ln(1.5), 10 terms := Taylor FP32 Value = 0x3ecf9521 = 0.40543463826179504 | Function value = 0.40546509623527527
ln(1.5), 15 terms := Taylor FP32 Value = 0x3ecf9934 = 0.4054657220840454 | Function value = 0.40546509623527527
```

Figure 21: Convergence Testcases

VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

Address	Bytes	Raw Hex
0x0000000010000200	00 00 00 3f 55 55 d5 3e	0x3ed555553f000000
0x0000000010000208	88 88 d0 3e 21 95 cf 3e	0x3ecf95213ed08888
0x0000000010000210	34 99 cf 3e 00 00 00 00	0x000000003ecf9934

Figure 22: Convergence Testcases Verification: VM memory dump

```
--- LN EDGE TESTCASES ---
ln(1.0), 0 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 0.0
ln(1.5), -1 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 0.40546509623527527
ln(0.0), 10 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = undefined (out of domain)
ln(-1.0), 10 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = undefined (out of domain)
ln(3.0), 30 terms := Taylor FP32 Value = 0xcbb40004 = -23592968.0 | Function value = 1.0986123085021973
```

Figure 23: Edge Testcases

VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

Address	Bytes	Raw Hex
0x0000000010000200	ff ff ff ff ff ff ff ff	0xffffffffffffffff
0x0000000010000208	ff ff ff ff ff ff ff ff	0xffffffffffffffff
0x0000000010000210	04 00 b4 cb 00 00 00 00	0x00000000cbb40004

Figure 24: Edge Testcases Verification: VM memory dump

Reciprocal

```
--- RECIPROCAL TESTCASES ---
1/(1.0), 5 terms := Taylor FP32 Value = 0x3f800000 = 1.0 | Function value = 1.0
1/(0.5), 8 terms := Taylor FP32 Value = 0x3fff0000 = 1.9921875 | Function value = 2.0
1/(1.5), 6 terms := Taylor FP32 Value = 0x3f280000 = 0.65625 | Function value = 0.6666666865348816
1/(0.800000011920929), 7 terms := Taylor FP32 Value = 0x3f9fff7a = 1.2499840259552002 | Function value = 1.25
1/(1.2000000476837158), 15 terms := Taylor FP32 Value = 0x3f555555 = 0.8333333134651184 | Function value = 0.8333333134651184
```

Figure 25: Generic Testcases

VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

Address	Bytes	Raw Hex
0x0000000010000200	00 00 80 3f 00 00 ff 3f	0x3fff00003f800000
0x0000000010000208	00 00 28 3f 7a ff 9f 3f	0x3f9fff7a3f280000
0x0000000010000210	55 55 55 3f 00 00 00 00	0x000000003f555555

Figure 26: Generic Testcases Verification: VM memory dump

```
--- RECIPROCAL CONVERGENCE TESTCASES ---
1/(0.5), 1 terms := Taylor FP32 Value = 0x3f800000 = 1.0 | Function value = 2.0
1/(0.5), 3 terms := Taylor FP32 Value = 0x3fe00000 = 1.75 | Function value = 2.0
1/(0.5), 5 terms := Taylor FP32 Value = 0x3ff80000 = 1.9375 | Function value = 2.0
1/(0.5), 10 terms := Taylor FP32 Value = 0x3fffc000 = 1.998046875 | Function value = 2.0
1/(0.5), 15 terms := Taylor FP32 Value = 0x3ffffe00 = 1.99993896484375 | Function value = 2.0
```

Figure 27: Convergence Testcases

VM Memory Dump

Previous Page 1 of 1 Next Toggle Sort

Address	Bytes	Raw Hex
0x0000000010000200	00 00 80 3f 00 00 e0 3f	0x3fe000003f800000
0x0000000010000208	00 00 f8 3f 00 c0 ff 3f	0x3fffc0003ff80000
0x0000000010000210	00 fe ff 3f 00 00 00 00	0x000000003ffffe00

Figure 28: Convergence Testcases Verification: VM memory dump

```
--- RECIPROCAL EDGE TESTCASES ---
1/(1.0), 0 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 1.0
1/(0.5), -1 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = 2.0
1/(0.0), 10 terms := Taylor FP32 Value = 0x7fc00000 = nan | Function value = undefined (division by zero)
1/(-1.0), 10 terms := Taylor FP32 Value = 0x447fc000 = 1023.0 | Function value = -1.0
1/(3.0), 30 terms := Taylor FP32 Value = 0xcdaaaaaa = -357913920.0 | Function value = 0.3333333432674408
```

Figure 29: Edge Testcases

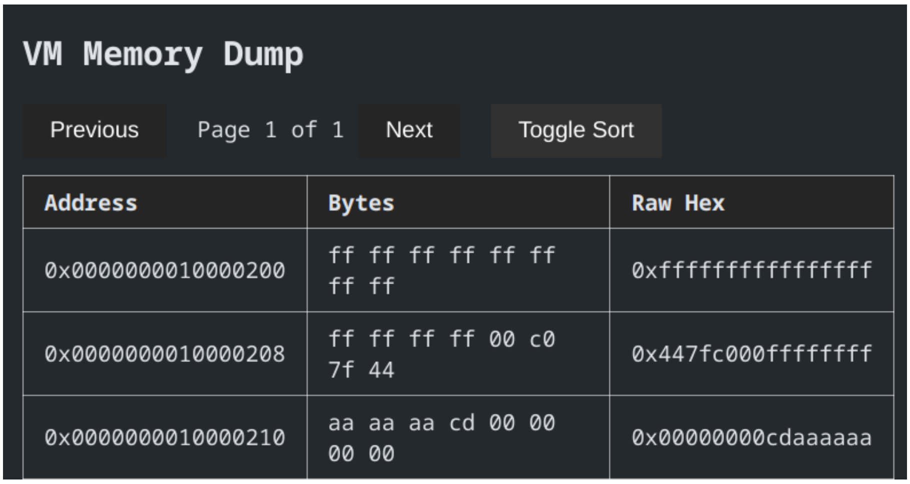


Figure 30: Edge Testcases Verification: VM memory dump

Issues faced

- A direct issue faced while writing the assembly is the **stack pointer**. Stacks in RISC V have to be **16 byte aligned** always according the official unprivilegded manual. Before I was pulling down stack pointer by 12 or 14 and the VM would just crash.
- Now that we learnt about 16 byte alignment, it still didnt work. It only seemed to work when i pulled down the stack by 32 bytes. What I couldnt figure out it is this a problem with the simulator or a RISC V guildline (I couldnt find anything online about 32 byte alignement)
- For large values, the taylor series terms were coming horrendously wrong. This occured because of overflow issues of the integer (mostly) or precision errors in FP32.
- The `fact` procedure was first handling INT32 values, which does not allow terms with high factorials (which will overflow). Changing this to INT64 instantly improves convergence.