Shitty Fucking Useless Draft/Design

Mahmoud Adas, Evram Youssef, Mohamed Shawky, Remonda Talaat

March 20, 2020

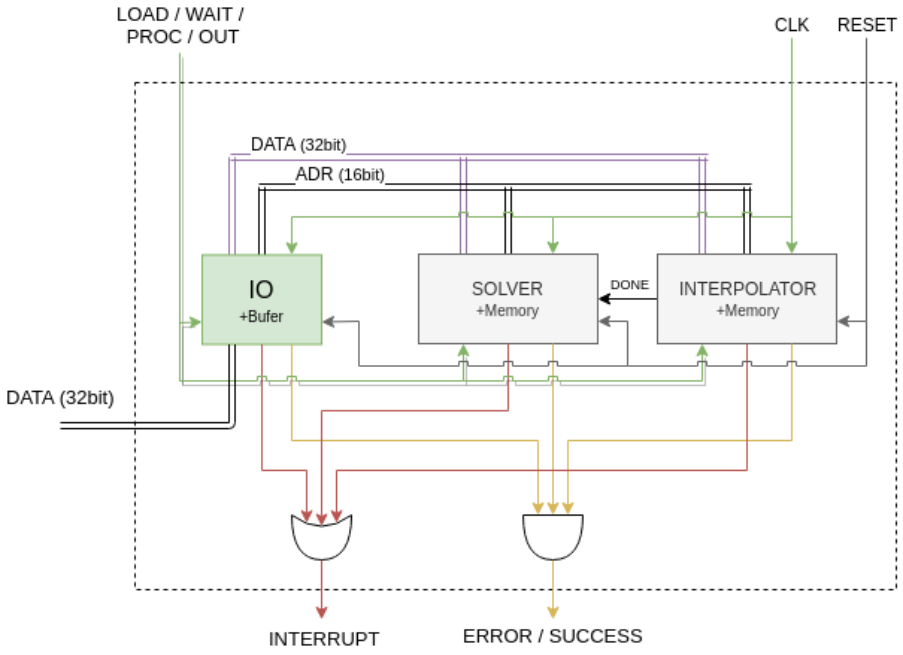

Figure 1: Overall Design

Interfaces and HW Summary

The hardware has the following interfaces that triggers some actions summarized below and detailed in the rest of the document.

- CLK: IN
- RESET: ASYNC IN

  - clears all internal states of all modules:
    * IO internal buffer
    * ERROR/SUCCESS of all modules resets to SUCCESS
    * INTERRUPT resets to zero
  - Memory at solver and interpolator are NOT cleared
  - at next clock, CPU is expected to turn the `LOAD / WAIT / PROC / OUT` into `LOAD` state and we will start loding input again.
- LOAD / WAIT / PROC / OUT (2bit): IN
  - set the current major state of the machine
  - LOAD(0):
    * IO receives *compressed* data from the CPU
    * IO decompresses data into buffer
    * buffer is flushed into data bus with appropriate adderss
    * ends when cpu finishes its data loading and switches to `WAIT` state
  - WAIT(1):
    * Same state as `LOAD`, but IO doesn't receive anymore data from CPU
    * ends when IO flushes all its buffer and raises `INTERRUP` with either `ERROR` or `SUCCESS`
  - PROC(2):
    * SOLVER sends time step to calculate U at
    * SOLVER and INTERPOLATOR work concurrently to calculate their outputs
    * INTERPOLATOR sends `DONE` signal to SOLVER when it finishes the interpolated U
    * SOLVER can request to copy the interpolated U
    * INTERPOLATOR waits for SOLVER to send next time step
    * ends when either SOLVER or INTERP raises INTERRUPT with either `SUCCESS` or `ERROR`
  - OUT(3):
    * IO just copies final outputs to cpu from SOLVER memory
    * ends when IO raises INTERRUPT with either `SUCCESS` or `ERROR`
- DATA (32bit): INOUT
  - Data bus between cpu and io
- INTERRUPT: OUT
  - raised from 0 to 1 when some internal module (IO / SOLVER / INTERPOLATOR) finishes its task
  - if task finished with success the `ERROR / SUCCESS` is set to `SUCCESS`, otherwise it's `ERROR`
- ERROR(0) / SUCCESS(1): OUT
  - CPU should operate on this value ONLY when `INTERRUPT` is 1
  - errros that could happen include: divide by zero, H $>$ 1, incomplete input

Simulation Workflow

Input Preparing

This stage is the responsibility of a script that runs before the simulation:

- INPUT: json file that follows the format stated in main document
- create bit stream of the read data that follows the `Input Data Structure` specifications
- encode the bits following the `Compression` specifications
- collect encoding output in ASCII string, each byte in string is either '0' or '1' in ASCII format
- when the string reaches the length of 32 bytes, push it to output file
- if the last created string didn't reach the length of 32 bytes, complete the rest with '0' and push it to the output file
- OUTPUT:
  - ASCII file that contains multiple lines of compressed data
  - each line has exactly 32 '0' or '1' ASCII characters
  - ONLY the ASCII characters 0 or 1 are permitted in the file and NOTHING ELSE
  - there is NO EMPTY LINE/s in the file or spaces

Instantiating HW

This stage and all the next ones are the responsibility of the CPU simulation code.

CPU is a non-synthesisable HDL test-bench that:

- instantiates the HW main module
- attaches the appropriate signals to the HW main module
- generates CLK with fixed frequency
- loads data into HW
- puts HW into PROCESS state
- load output out from the HW and into a file

Loading Input

- load the output of the former script into array of vectors each is 32bit wide that will hold one line in the file
- put HW at LOAD state
- RESET for one cycle
- for each 32bit vector in the former array:
  - at the positive edge of CLK:
    - load vector into `DATA` bus

- load DATA with 0s
- wait for the positive edge of `INTERRUPT` signal
- check for `ERROR / SUCCESS` and only proceed if it is SUCCESS

Processing

- put HW at PROCESS state
- wait for the `INTERRUPT` positive edge
- check for `ERROR / SUCCESS` and only proceed if it is SUCCESS

Extracting Output

- put high impedence on `DATA` bus
- put HW at OUT state
- keep receiving data into array of vectors and outputting them into file in the same format of the input file
- wait for the positive edge of `INTERRUPT` signal

Simulation is done!

You can turn the output into human-readable json using output-formatting script

Sepecifications

Memory Mapping

The Following addresses are only meant for internal communicating between modules, and they don't need to resemble actual addresses stored at some memory.

The address loaded at the bus resembles what kind of data is on data bus or what kind of data this module should output.

If address bus is loaded with an address `Adr` that some module `M` is not assigned to, module `M` must ignore the data and address bus so the rest can communicate.

This way communicating is simplified.

`A` column for module `M` is the action of the address taken at module `M` when it sees that address. It's either: * `W`: *Write to* module `M`. Module `M` is expected to *read* the data bus and store data internally, so the other module *wrote* to module `M`. * `R`: *Read from* module `M`. Module `M` is expected to *write* some data to the data bus as response to this address, so the other module *reads* from it.

Solver Memory Mapping

Solver module listens at the following addresses: | Address | A | Type | #Words | Name | Description | |———|—|————|———|———|————————————| | 0x0000 | W | `struct Header` | 1 | Header | Includes Dimensions and modes | | 0xXXXX | W | `f64` | 4 | H | Timestep (variable step mode) | | 0xXXXX | W | `f64` | 4 | Error | Error Tolerance (variable step mode) | | 0xXXXX | W | `f64[50][50]` | 10000 | A | Matrix A | | 0xXXXX | W | `f64[50][50]` | 10000 | B | Matrix B | | 0xXXXX | W | `f64[50]` | 200 | X | Initial value of X | | 0xXXXX | R | `f64[50][64]` | 12800 | Xout | Final Output X |

Interpolator Memory Mapping

Interpolator module listens at the following addresses: | Address | A | Type | #Words | Name | Description | |———|—|————|———|———|————————————| | 0x0000 | W | `struct Header` | 1 | Header | Includes Dimensions and modes | | 0xXXXX | W | `f64[64]` | 256 | T | Time points where solutions are required | | 0xXXXX | W | `f64[50]` | 200 | U0 | Initial U vector | | 0xXXXX | W | `f64[50][64]` | 12800 | Us | U vector at required time steps | | 0xXXXX | R | `f64[50]` | 200 | Uint | Interpolated U Vector | | 0xXXXX | W | `f32` | 2 | K | Time to calculate U at |

TODO: size of K and its name TODO: figure out the addresses

Modules

RAM

TODO: figure showing its ports

- Role:
  - Store input data for solver to access
  - Store output data from solver that IO will later will transfer back to CPU
- Ports:
  - INOUT: 32bit data bus
  - IN: 16 bit address bus
  - IN: R/W control signal
- Word: 16 bit
- Size: 33265 words
- Address Range: [0x0000, 0x81F0] all readable and writeable

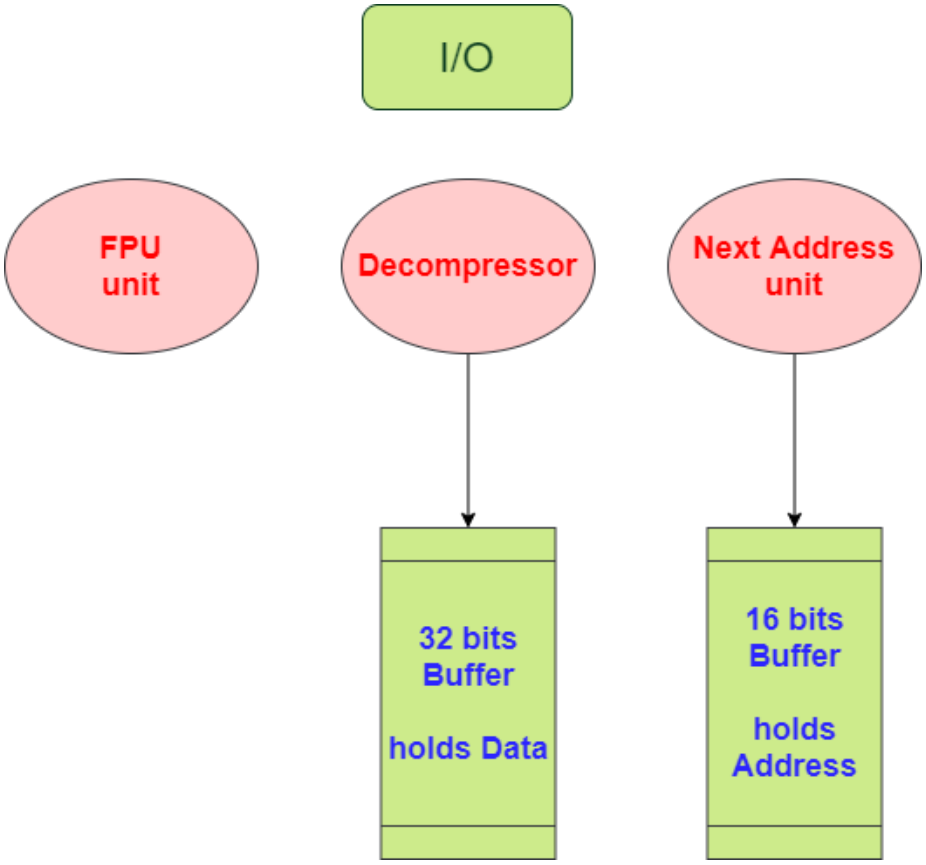

Figure 2: I/O Design

IO

- Role:
  - Receive backets of 32 bits from the CPUm through `DATA` bus.
  - Decompress the data
  - Send the data to other modules (Solver/Interpolation/RAM).
- Ports:
  - INOUT: 32bit data bus with other modules
  - INOUT: 32bit data bus with CPU
  - IN: 16bit address bus
  - IN: CLOCK
  - IN: Reset
  - IN: 2bit Load/Process/Out
  - OUT: Interrupt to CPU
  - OUT: R/W to RAM
  - OUT: Error to CPU

IO Job and sub_modules:

- On a large scale, it receives 32bit streams and pass them to both `Solver` and `Interpolator`, and when `CPU` requests output result, and they are available, sends them out.
- `Decompressor` component is discussed here [URI]

Solver

- Role:
  - Computes the upcoming X knowing h, the previous X and U.
  - Counts the error difference and the new h.
  - Checks for arithmetic errors that may occurs (e.g. div. by zero)
  - Outputs the final X's at the desired times to the RAM.
- Ports:
  - IN: Done signal from interpolator
  - INOUT: 32bit data bus with other modules
  - IN: 16bit address bus
  - IN: CLOCK
  - IN: Reset
  - IN: 2bit Load/Process/Out
  - OUT: Interrupt to CPU
  - OUT: R/W to RAM
  - OUT: Error to CPU

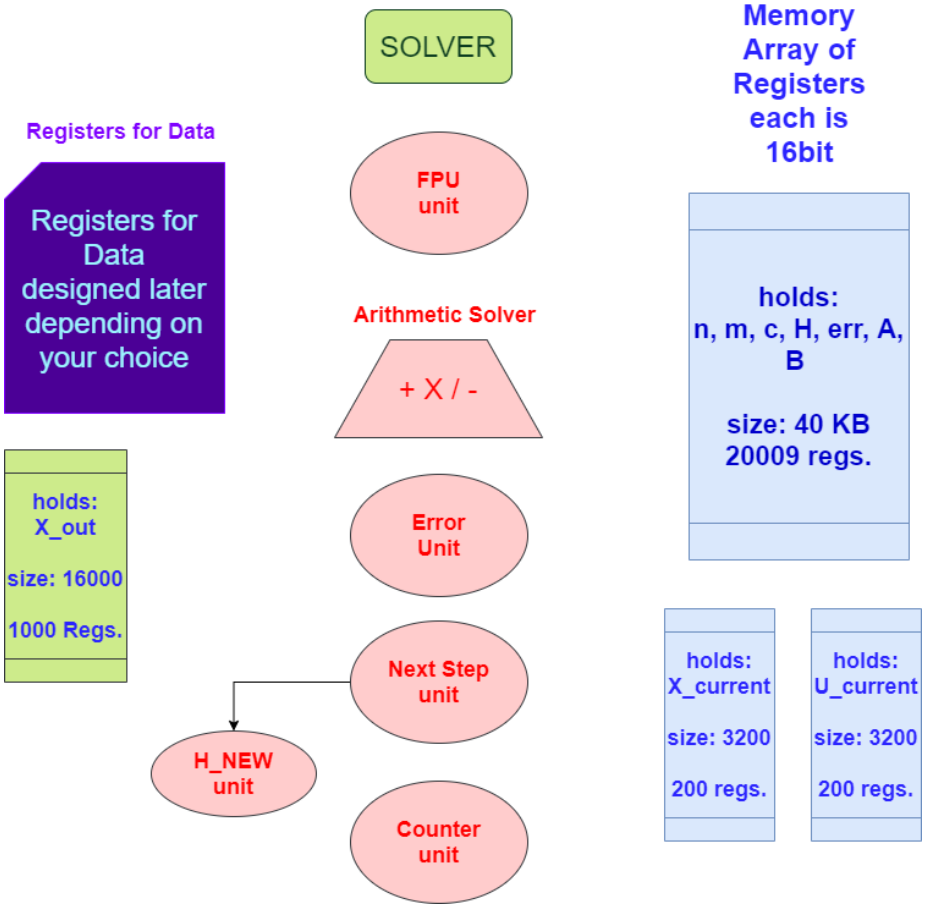

Figure 3: Solver Design

Solver Job and sub_modules:

- On a large scale it receives `U_h` from interpolator, gives it another `h` to compute `U_hnew` at, then computes `X_h` and decides to stop and flush output to I/O or continue.
- At the begining it receives its data from I/O such as N,M,err,h... etc.
- `FPU` helps knowing mode and fp.
- `Arithmetic Solver` where the absolute mathemetical operations reley.
- `Error Unit` to detect any error in sizes, h, numbers... etc.
- `Next Step Unit` helps create the upcoming `h_new` so that when solver is busy calculating `X_h`, interpolator is calculating `U_hnew`, more here [URL], this unit represents teh stepper unit, holds the logic of calculating the adaptive `h`, and detects when to stop, in summation it calculates the next `h`, even if it was fixed step.
- `Counter Unit`, tells you when to calculate more, when to advance to next time (in T_s), and when to stop the whole operations.

Interpolator

- Role:
  - Calculates the upcoming U knowing h, U initial and U final.
- Role:
  - OUT: Done signal to Solver
  - INOUT: 32bit data bus with other modules
  - IN: 16bit address bus
  - IN: CLOCK
  - IN: Reset
  - IN: 2bit Load/Process/Out
  - OUT: Interrupt to CPU
  - OUT: Error to CPU

Interpolator Job and sub_modules:

- On a large scale, it only computes U at a specific time.
- At the begining it receives from `IO` the `U_s` and `T_s` fixed variables/data, and stores them.
- Each U_s is at most of size [64*50] = 3200 bits = 200 registers
- Each T_s is at most of size [64] bits = 4 regs.
- T_s keeps hold of the time where each U_s represents, for example, T_s = [1,2,3], there fore the first 200 regs. in U_s are the value of `U` at time `1`, and so on...
- You will need an iterator to target the appropriate U_s from the array of memory, that's the `Address pointer Unit`.

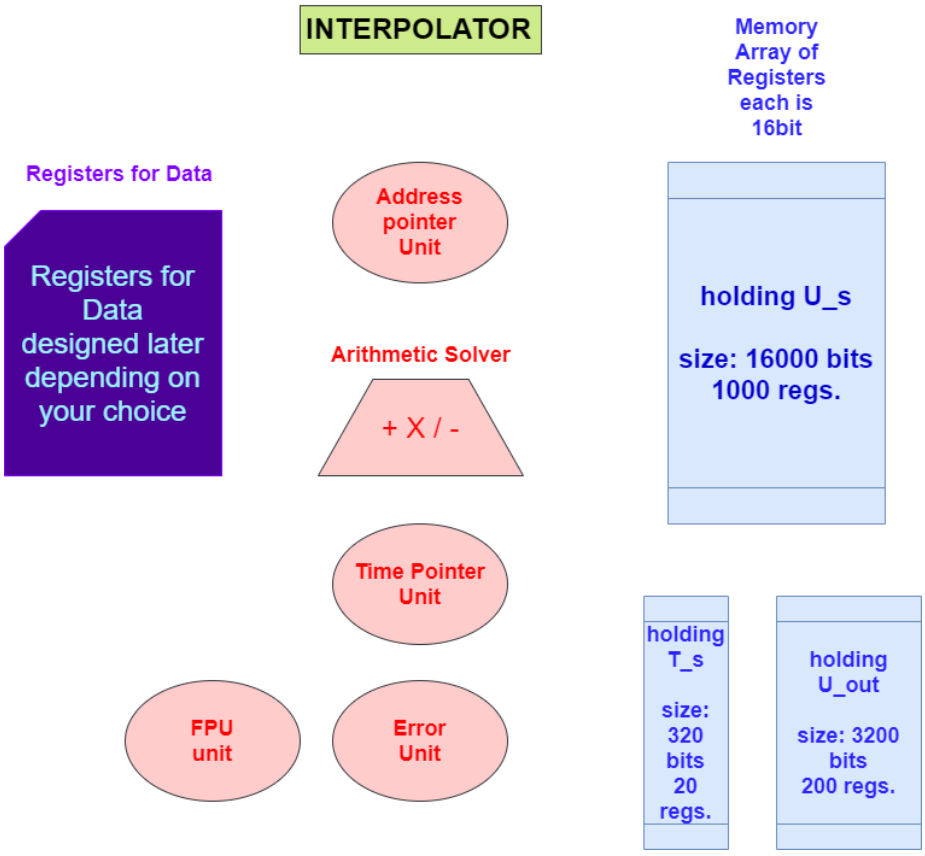

Figure 4: Interpolator Design

- `Arithmetic Solver` is where you're absolute mathematical operations reley.
- `Time Pointer Unit` helps you to identify which T of the T_s array we are currently handling.
- Please notice that, at the begining of the program T_init = 0, T_final = T_s[0], after a successfull output, T_init = T_final, and T_final = T_s[1], and so on...
- `Error Unit` it's responsible for detecting when an arithmetic error may occurr, like dividing by zero, `h` is getting bigger every time, different sizes....etc.
- `FPU` helps you to know what mode are we in (fixed/variable step size), and what is type of fixed point operations.

Fixed/Floating Point Unit (FPU)

TODO: role TODO: figure showing its ports TODO: ports

Header Data Structure

Bit Index	Name	Description	Datatype	Total Size
15:10	N	Dimension of X	`uint`	6 bits
9:4	M	Dimension of U	`uint`	6 bits
3	Solver Mode	Fixed Step or Variable Step	`enum`	1 bit
2:1	FPU Precision	fixed point, f64 or f32	`enum`	2 bits
0	NOT USED	——	——	1 bit

Compression

Follow bit-level Run-length encoding to compress ram content before sending them, by taking each (one to eight) [1:8] repeating bits and compressing them into four bits, using `RLE` (Run length encoding) algorithm.

Input

Bit stream of `X` bits

Output

`Y` compressed 4bit packets, where `X >= Y >= ceil(X/8)` Each packet must follow this format: | Bit Index | Description | Size | |————|————————

———|———| | 3:1 | Number of bits to generate - 1 | 3 bits | | 0 | Bit to generate | 1 bit |

NOTE: the 3 bits must be incremented before decode

TOOO: encoding figure

Example

Original	Compression
1111111!	1111
0000	0110

Pseudo-code

```
c = first bit in bit_stream
count = 0
for b in bit_stream:
    if c == b and count < 7:
        count++
    else:
        emit_packet(count-1, c)
        count = 1
        c = b
```

Reason for choosing this size

Because the occurence of more that 7 ones or zeros simultaneously is very rare.

Problem

This compression algorithm may not compress the data, rather than that it may increase the number of bits.

Decompression

TODO: mention algorithm TODO: note the increment of numbers

Decompression, like a dummy operator, takes four bits. extract the count/existence of the fourth bit from the first three. then place the output in a buffer.