Unlocking the Power of Inline Floating-Point Operations on Programmable Switches

Yifan Yuan[1], Omar Alama[2], Jiawei Fei[2, 3], Jacob Nelson[4], Dan R. K. Ports[4], Amedeo Sapio[5], Marco Canini[2], Nam Sung Kim[1]

[1]UIUC, [2]KAUST, [3]NUDT, [4]Microsoft Research, [5]Intel

04/05/2022

Background & Motivation

- We are living in the era of programmable network.
- Networking switches with programmable pipeline, a.k.a. programmable switches, have been prevailing.

Programmable switches provide basic compute capability, great programmability and flexibility, while keeping line-rate forwarding.

Background & Motivation

- Programmable switches have been applied to accelerate/offload a wide range of networking and distributed applications.

NetChain (NSDI’18), DistCache (FAST’19)

NOPaxos (OSDI’16), Eris (SOSP’17)

NetCache (SOSP’17), HULA (SOSR’16)

Jaqen (Security’21), Poseidon (NDSS’20)

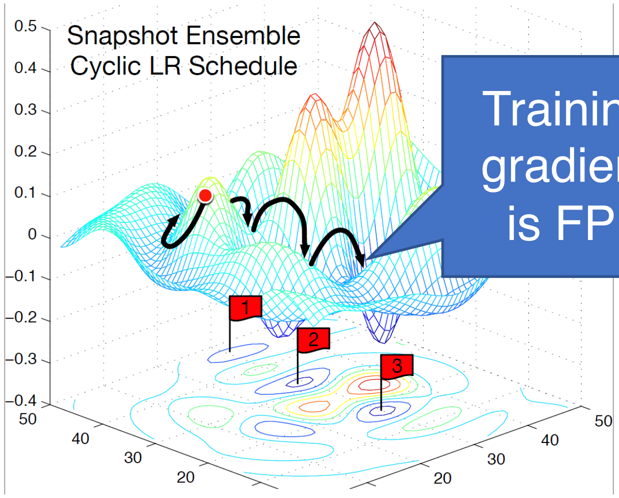
Cheetah (SIGMOD’20), NETACCEL (CIDR’19)

SwitchML (NSDI’21), ATP (NSDI’21)

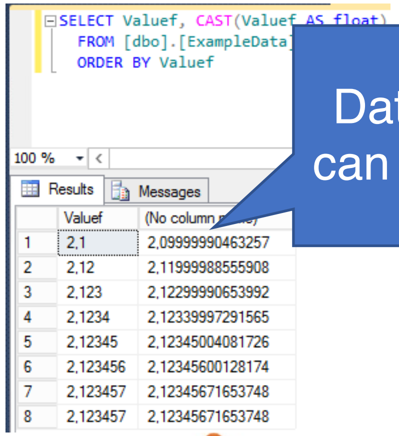
Are we still missing anything?

Background & Motivation

- Protocol-independent switch architecture (PISA), the de-facto programmable switch paradigm, has no support for floating point (FP) data formats, which are common in many use cases.



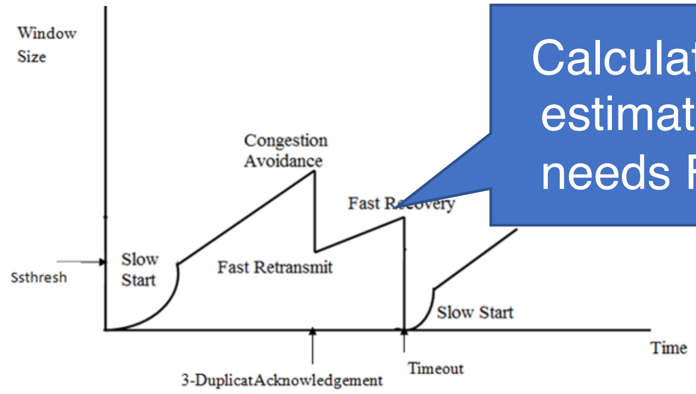
Training gradient is FP!



```
SELECT Valuef, CAST(Valuef AS float)
  FROM [dbo].[ExampleData]
  ORDER BY Valuef
```

	Valuef	(No column
1	2,1	2,09999990463257
2	2,12	2,11999988555908
3	2,123	2,12299990653992
4	2,1234	2,12339997291565
5	2,12345	2,12345004081726
6	2,123456	2,12345600128174
7	2,123457	2,12345671653748
8	2,123457	2,12345671653748

Datatype can be FP!

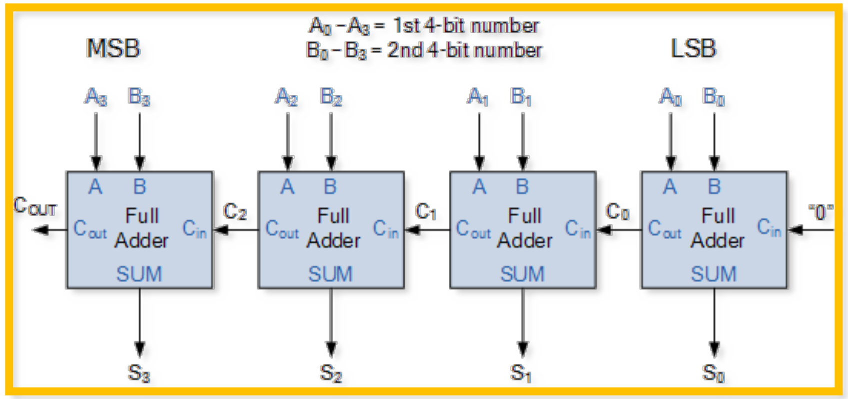


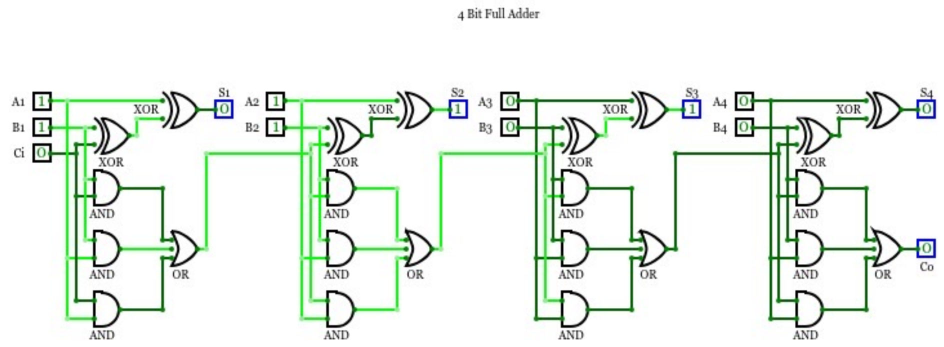
Calculating estimation needs FP!

It will be great if we can enable FP operations on PISA switch!

Challenges

- Why does the current PISA switch not support FP operation?
  - Let's see how arithmetic operation works under the hood at first!
- The goal of this work is to let PISA switch support FP operations efficiently.
- Integer (fixed point)?
  - C = A +/- B, done. Easy and simple.

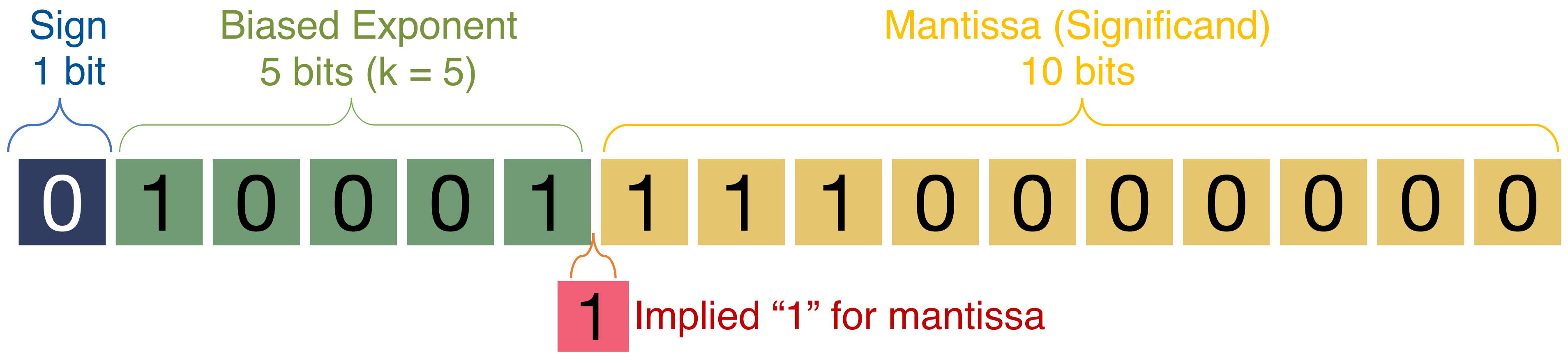




Challenges

- Why does the current PISA switch not support FP operation?
  - Let’s see how arithmetic operation works under the hood at first!

- FP?



Sign = 0

Exponent = $2^4 + 2^0 = 17$
Bias = $2^{k-1} - 1 = 2^{5-1} - 1 = 15$
Biased_exp = $17 - 15 = 2$

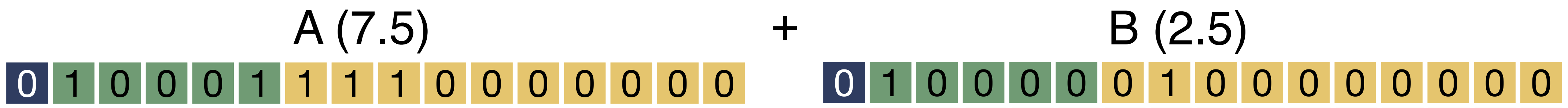
Mantissa = $2^{-1} + 2^{-2} + 2^{-3} = 0.875$

$$-1^{0} \times 2^{2} \times (2^{0} + 0.875) = 7.5$$

Challenges

- Why does the current PISA switch not support FP operation?
  - Let’s see how arithmetic operation works under the hood at first!

- FP?
  - C = A +/- B

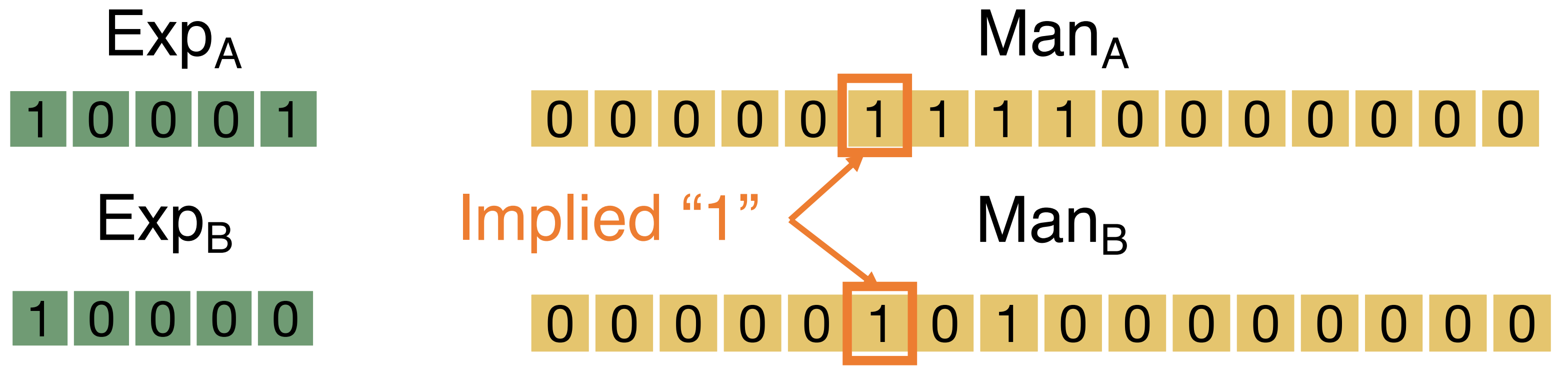


Challenges

- Why does the current PISA switch not support FP operation?
  - Let's see how arithmetic operation works under the hood at first!

- FP?
  - C = A +/- B

1. Extract

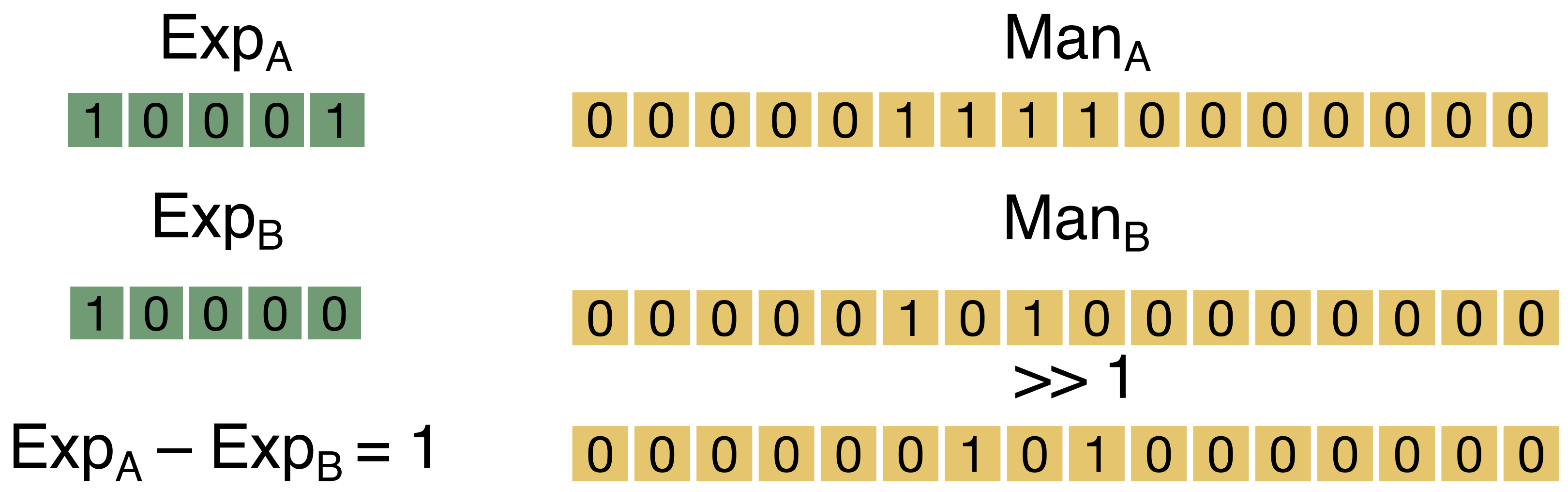


Challenges

- Why does the current PISA switch not support FP operation?
  - Let's see how arithmetic operation works under the hood at first!

- FP?
  - C = A +/- B

1. Extract
2. Align

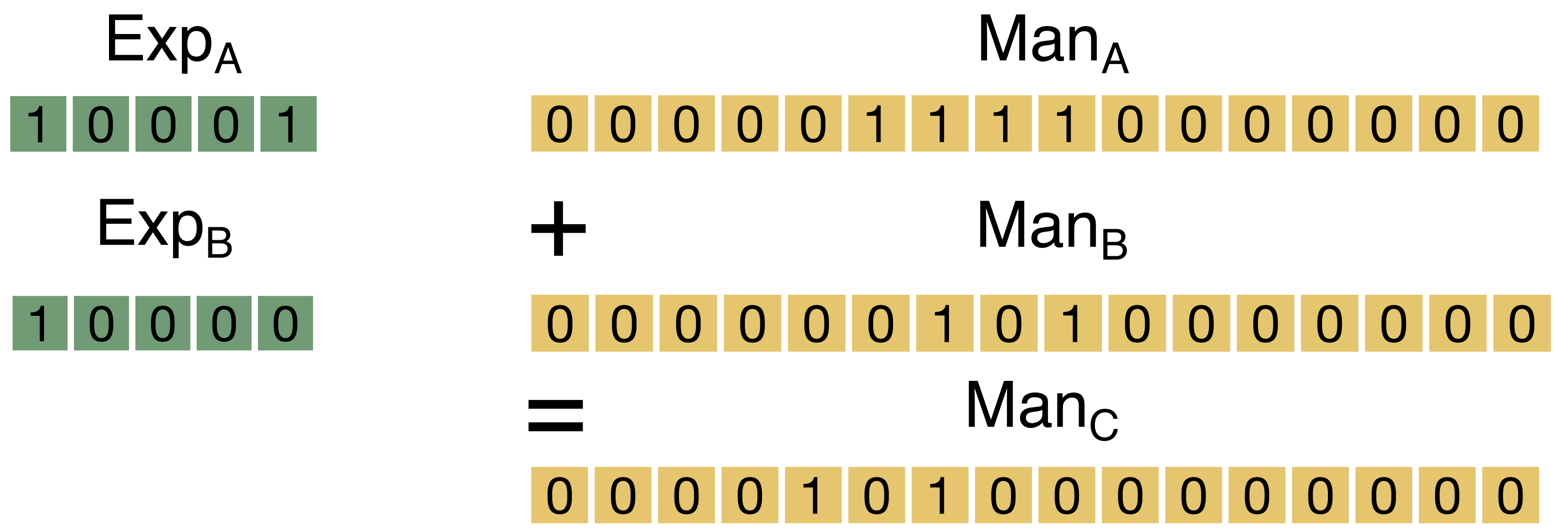


Challenges

- Why does the current PISA switch not support FP operation?
  - Let’s see how arithmetic operation works under the hood at first!

- FP?
  - C = A +/- B

1. Extract
2. Align
3. Add/sub

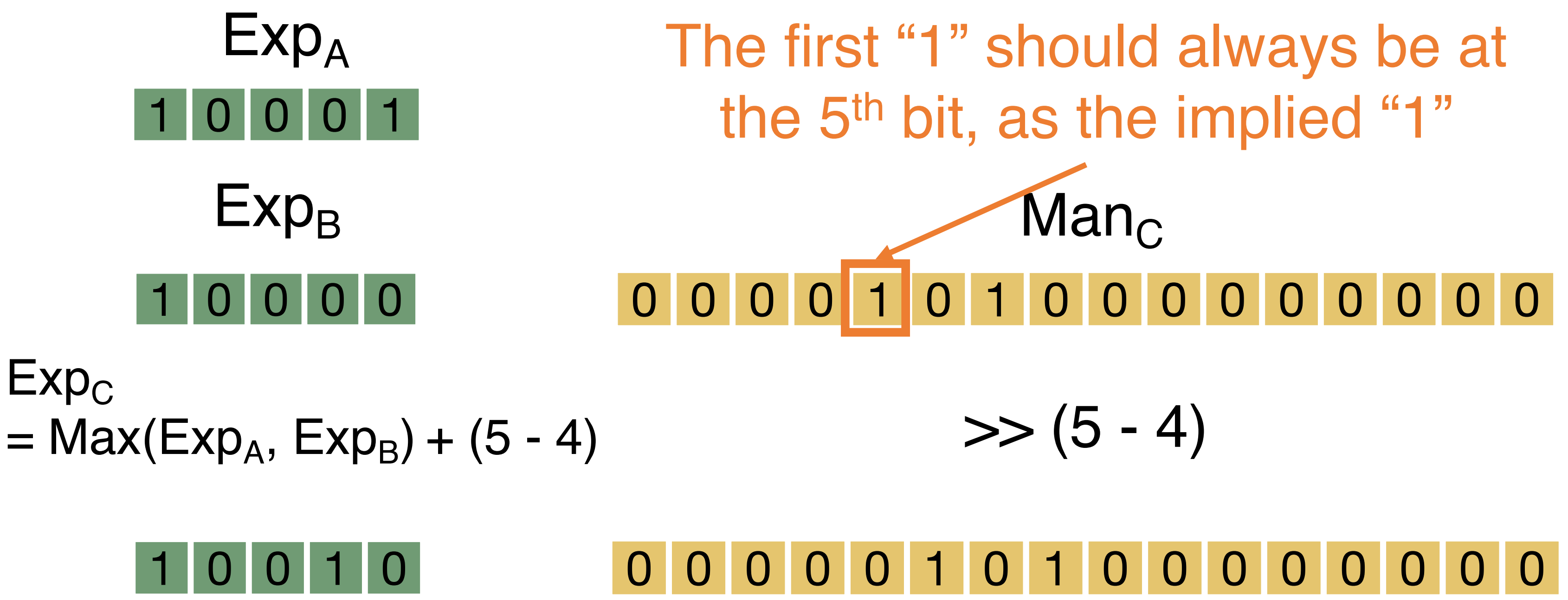


Challenges

- Why does the current PISA switch not support FP operation?
  - Let’s see how arithmetic operation works under the hood at first!

- FP?
  - C = A +/- B

1. Extract
2. Align
3. Add/sub
4. Renormalize

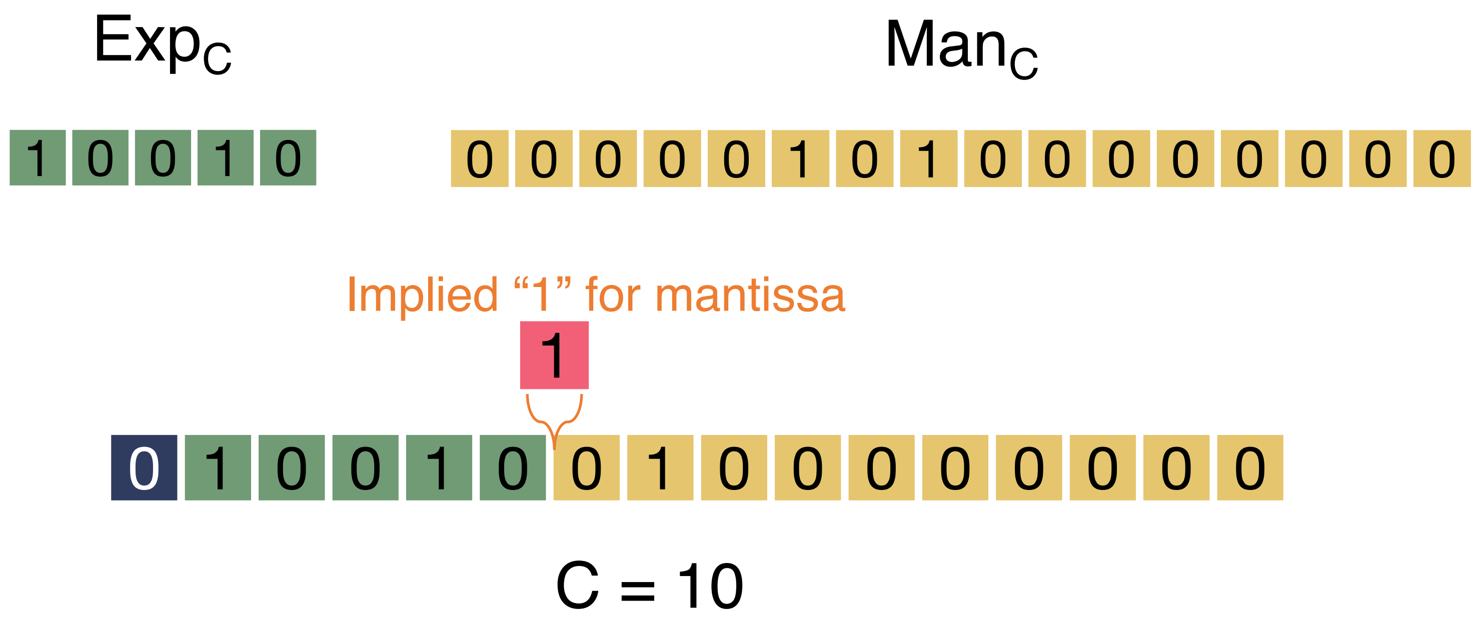


Challenges

- Why does the current PISA switch not support FP operation?
  - Let's see how arithmetic operation works under the hood at first!

- FP?
  - C = A +/- B

1. Extract
2. Align
3. Add/sub
4. Renormalize
5. Assemble

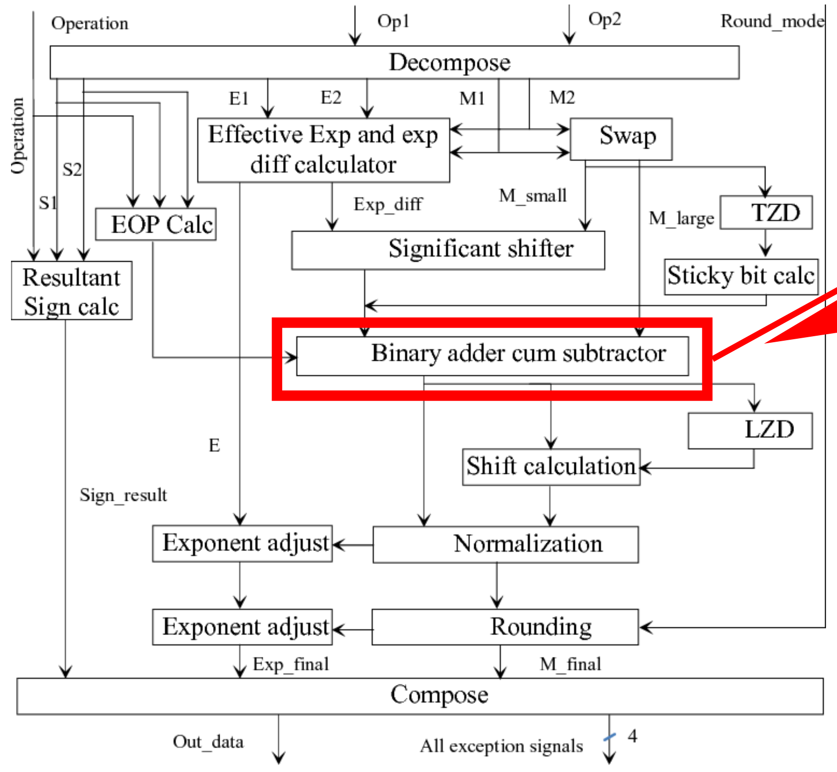


Challenges

- Why does the current PISA switch not support FP operation?
  - Let's see how arithmetic operation works under the hood at first!

- FP?
  - C = A +/- B

1. Extract
2. Align
3. Add/sub
4. Renormalize
5. Assemble

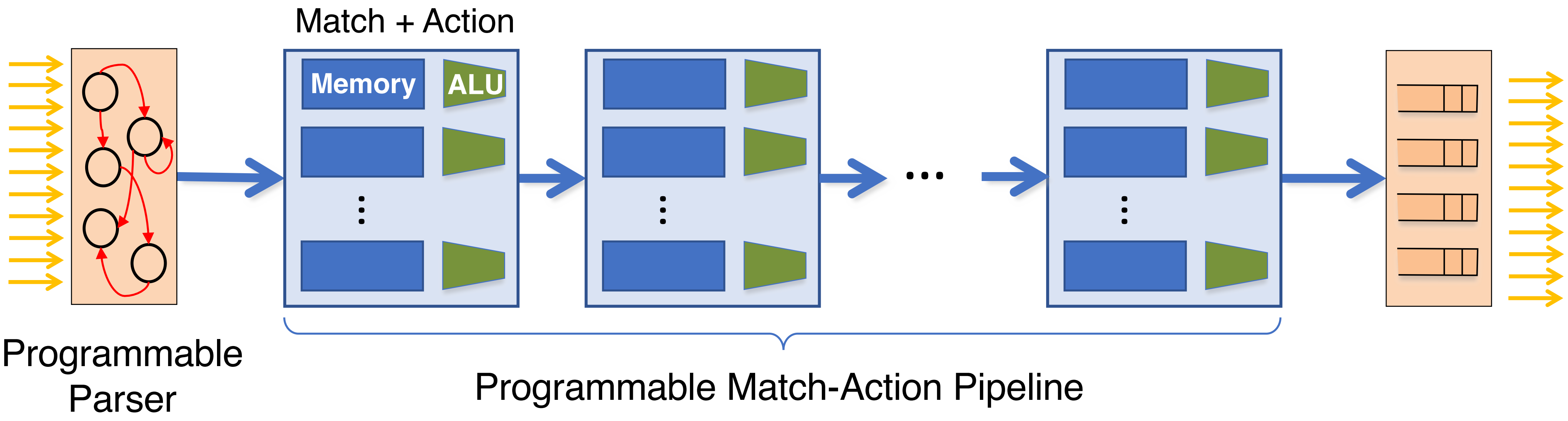


Every single block is complicated!

FP operations are not single-clock-cycle.

Challenges

- Going back to PISA architecture…
  - Fully-pipelined streaming design (cannot go backward, cannot stall)
  - ONE single action per stage
  - ONE access per memory location per packet



FP cannot be done in single pipeline stage anyway!

Block diagram credit: Xin jin, et al., NetCache: Balancing Key-Value Stores with Fast In-network Caching (SOSP'17)

Challenges

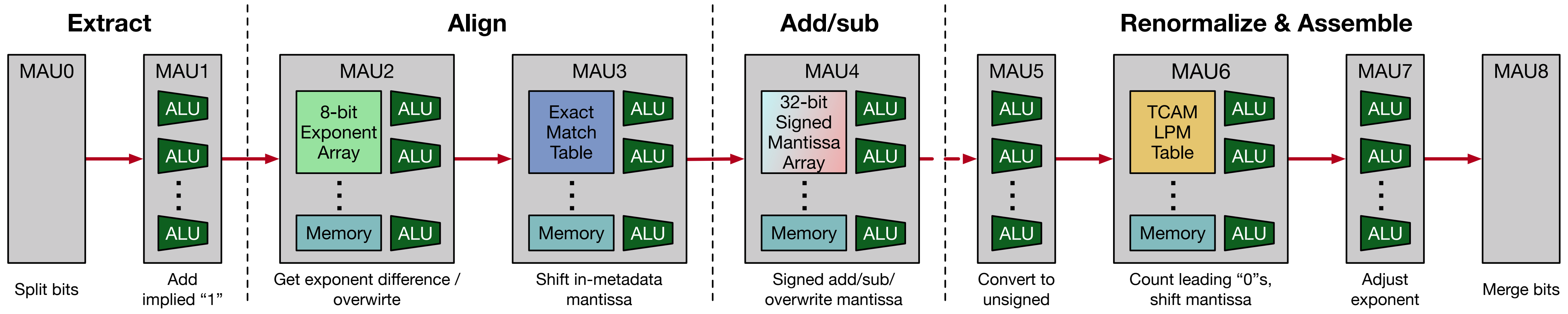
- Other programmable switch paradigms instead of PISA?
  - Switch with specific arithmetic support (e.g., Mellanox SHARP)?
    - High-performance (throughput, latency, and scalability)
    - Fixed functionalities, inflexible for emerging numerical formats (FP16, bfloat, MSFP, etc.)
  - FPGA-based “switch”?
    - Flexible enough
    - Not as high-performance (overall-throughput) as ASIC

PISA has the potential of balancing performance and flexibility.

FPISA: Native FP representation and operations in PISA

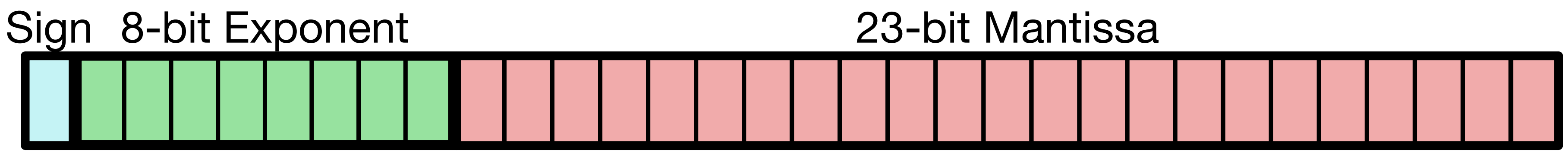
FPISA: High-level idea

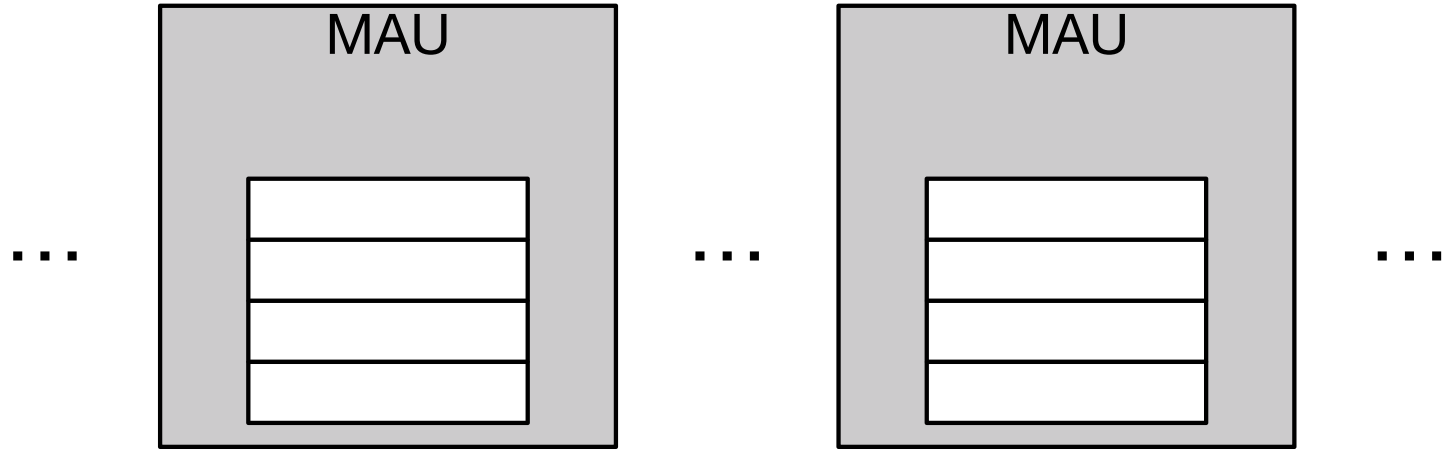
- Decompose an FP's representation (storage) and operation to mutual-independent, PISA-friendly steps.
- Keep the intermediate FP representation in PISA, until we need to get back to the end-host(s).
- Leverage networking-specific hardware units for FP sub-operations.



FPISA: FP representation and storage in PISA

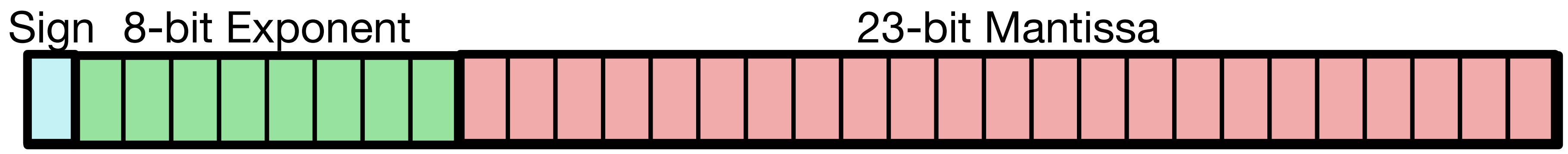
- We decouple the three components of a FP number and store them separately in PISA pipeline.

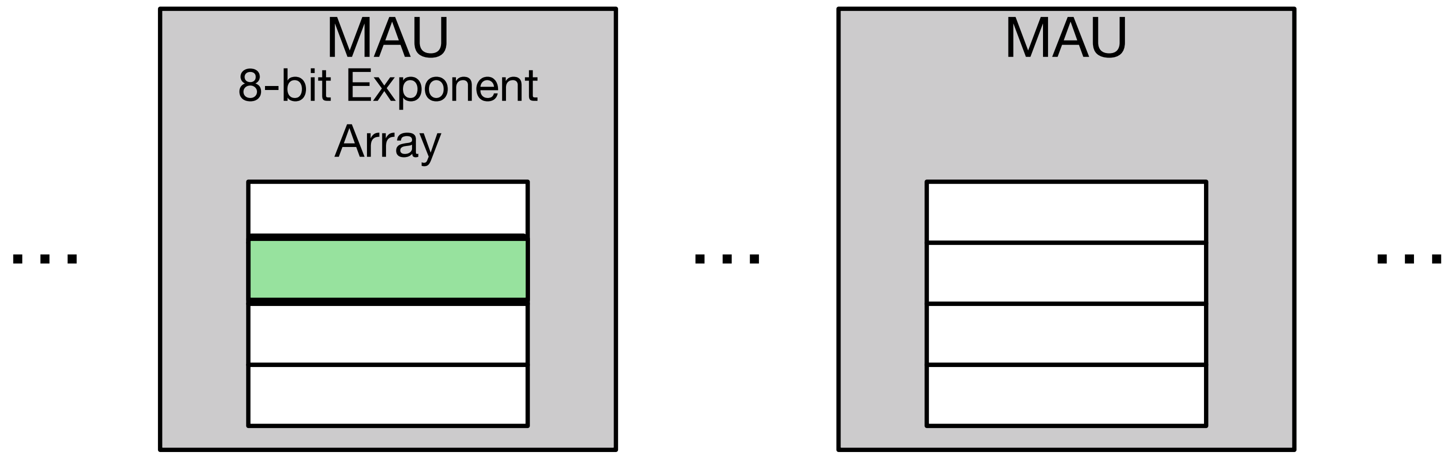




FPISA: FP representation and storage in PISA

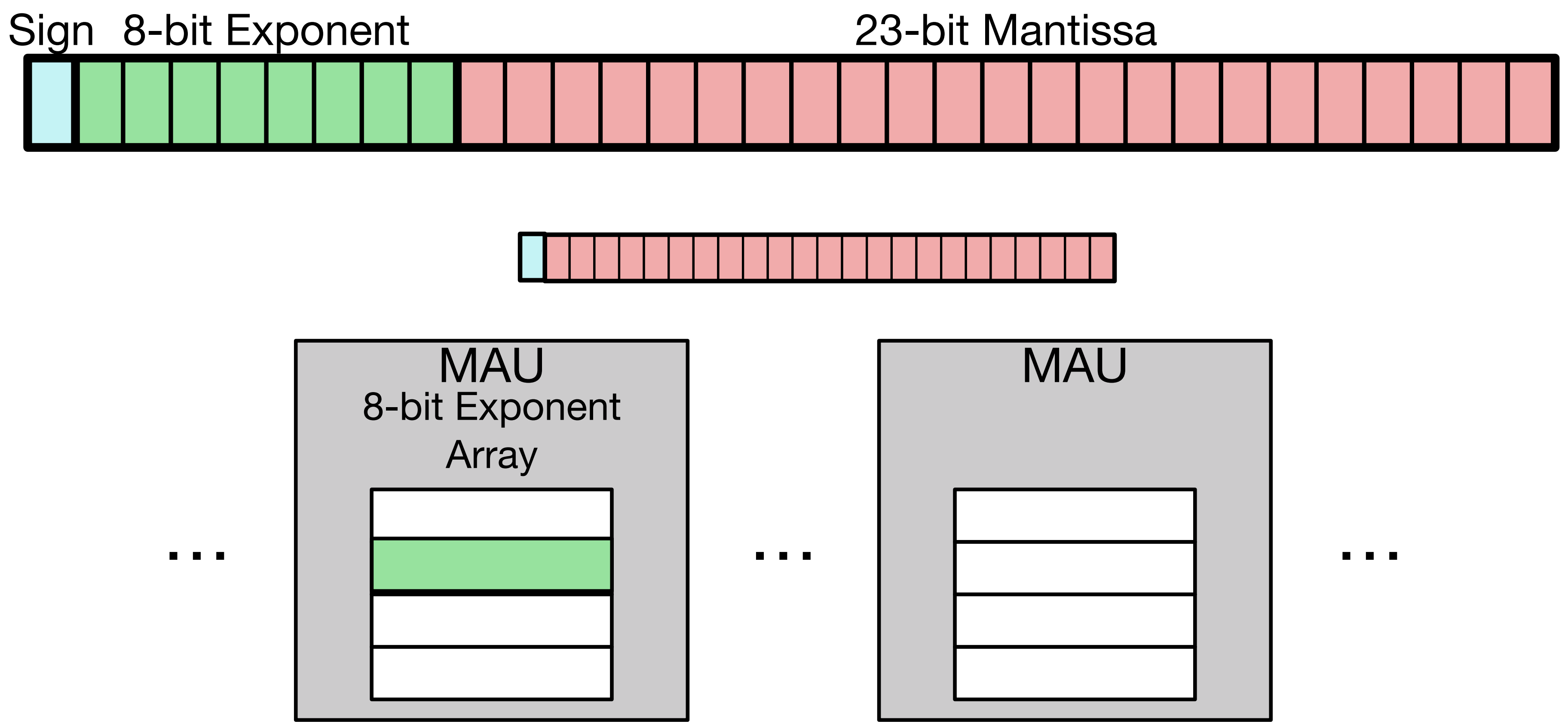
- We decouple the three components of a FP number and store them separately in PISA pipeline.





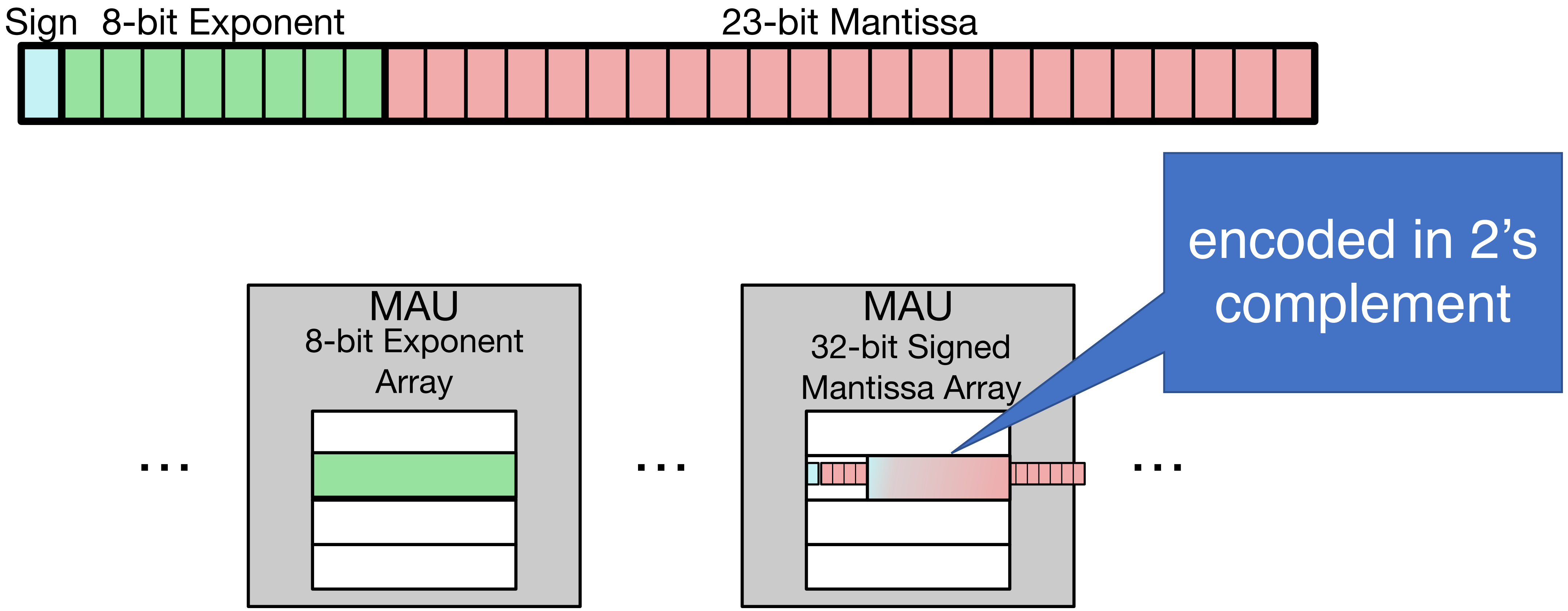
FPISA: FP representation and storage in PISA

- We decouple the three components of a FP number and store them separately in PISA pipeline.



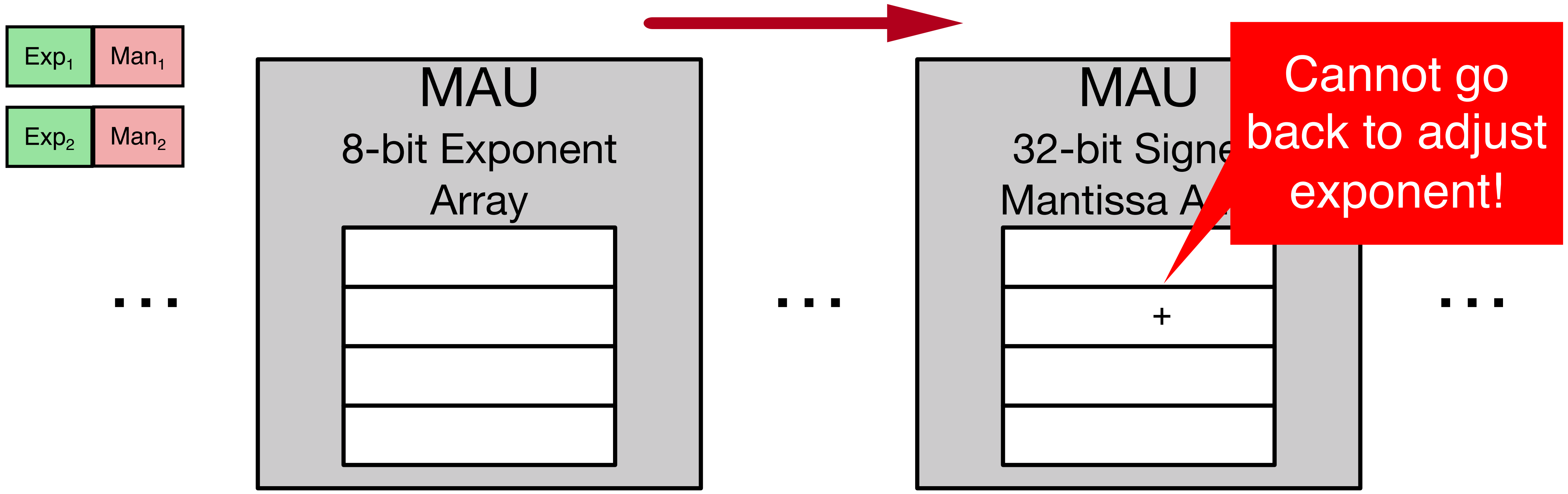
FPISA: FP representation and storage in PISA

- We decouple the three components of a FP number and store them separately in PISA pipeline.



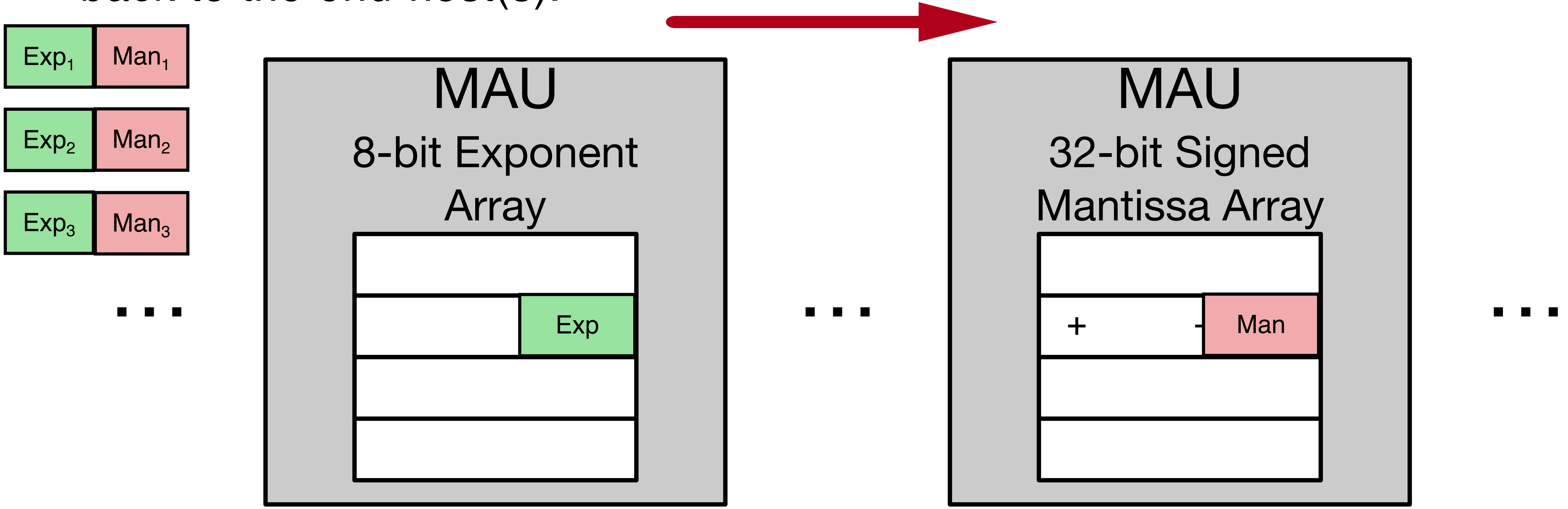
FPISA: Delayed normalization

- Suppose we want to calculate $V_1 + V_2 + V_3 = V_4$



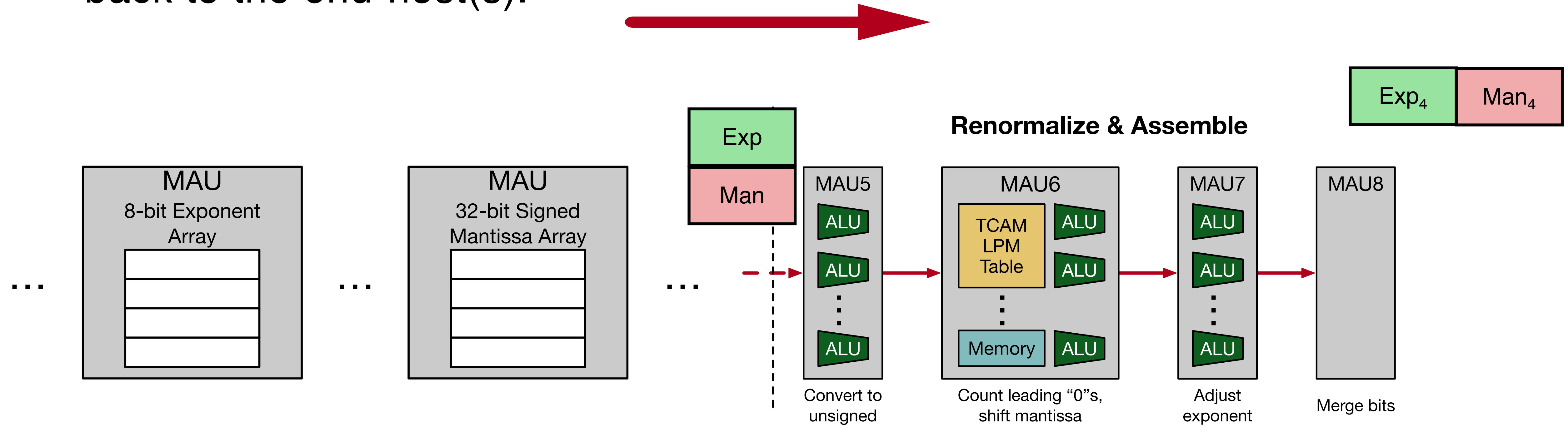
FPISA: Delayed normalization

- Suppose we want to calculate $V_1 + V_2 + V_3 = V_4$
- We delay the step "renormalization" until we need to get the result back to the end-host(s).



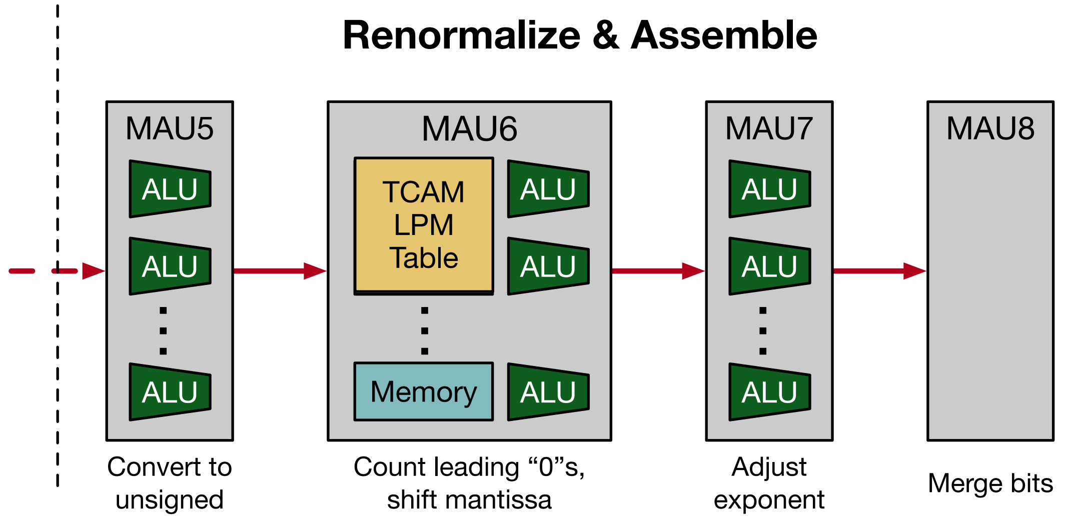
FPISA: Delayed normalization

- Suppose we want to calculate $V_1 + V_2 + V_3 = V_4$
- We delay the step “renormalization” until we need to get the result back to the end-host(s).



FPISA: Leverage networking hardware

- For renormalization, we need to find how many leading “0” we have in the operated mantissa, so that we can shift it and adjust the exponent.
- How can we do this efficiently and quickly?



FPISA: Leverage networking hardware

- For renormalization, we need to find how many leading “0” we have in the operated mantissa, so that we can shift it and adjust the exponent.

- How can we do this efficiently and quickly?

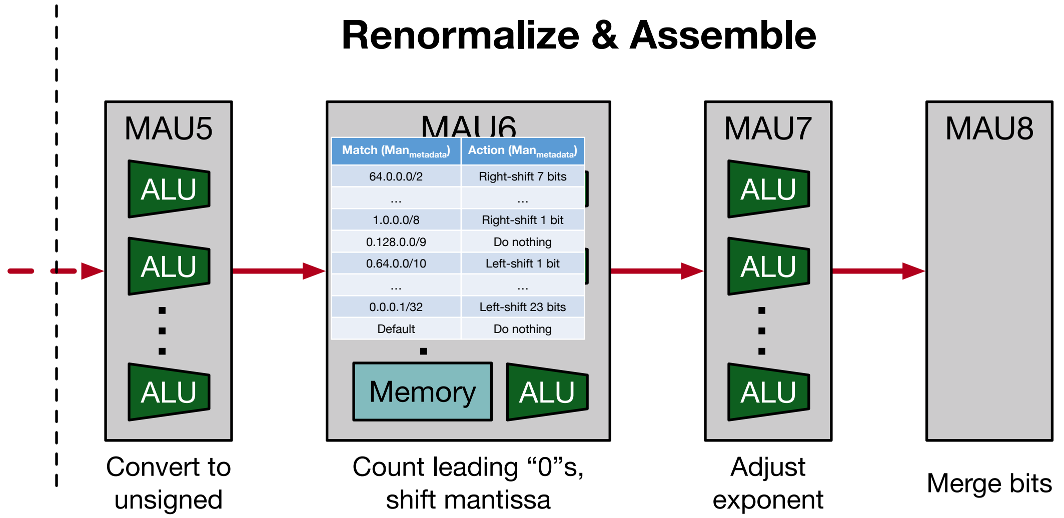
Match	Action
IP address/mask	Action
…	…

TCAM LPM Table

Match ($Man_{metadata}$)	Action ($Man_{metadata}$)
64.0.0.0/2	Right-shift 7 bits
…	…
1.0.0.0/8	Right-shift 1 bit
0.128.0.0/9	Do nothing
0.64.0.0/10	Left-shift 1 bit
…	…
0.0.0.1/32	Left-shift 23 bits
Default	Do nothing

Are we done?

- We implement FPISA with P4 in Intel’s Tofino-1 and find it not efficient enough.
- Example-1: saturated VLIW instruction slots –> limited data parallelism

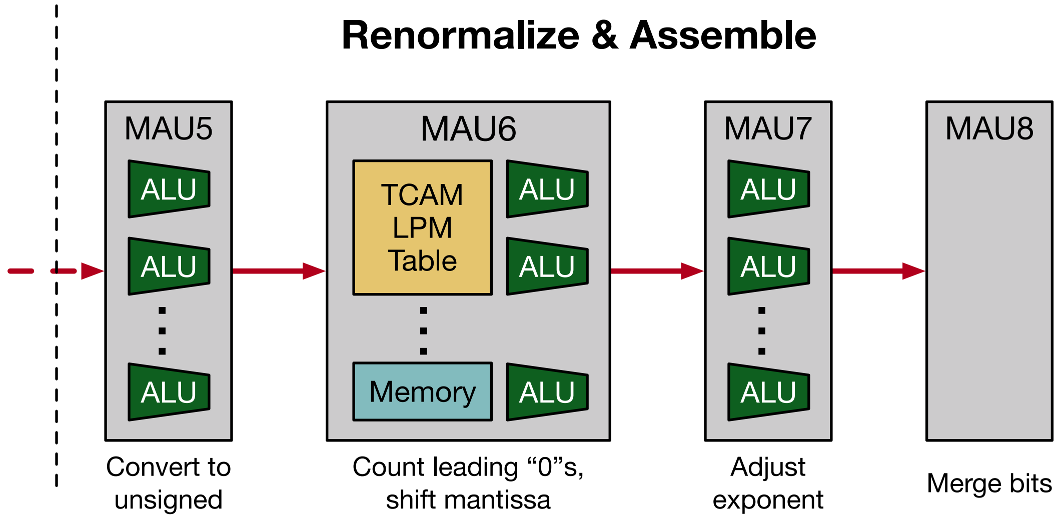


Match ($Man_{metadata}$)	Action ($Man_{metadata}$)
64.0.0.0/2	Right-shift 7 bits
...	...
1.0.0.0/8	Right-shift 1 bit
0.128.0.0/9	Do nothing
0.64.0.0/10	Left-shift 1 bit
...	...
0.0.0.1/32	Left-shift 23 bits
Default	Do nothing

Are we done?

- We implement FPISA with P4 in Intel's Tofino-1 and find it not efficient enough.

- Example-1: saturated VLIW instruction slots –> limited data parallelism
  - Enhancement: 2-operand shift instruction –> *"shift [operand0] [operand1]"*



Match ($Man_{metadata}$)	Action ($Man_{metadata}$)
64.0.0.0/2	Right-shift 7 bits
…	…
1.0.0.0/8	Right-shift 1 bit
0.128.0.0/9	Do nothing
0.64.0.0/10	Left-shift 1 bit
…	…
0.0.0.1/32	Left-shift 23 bits
Default	Do nothing

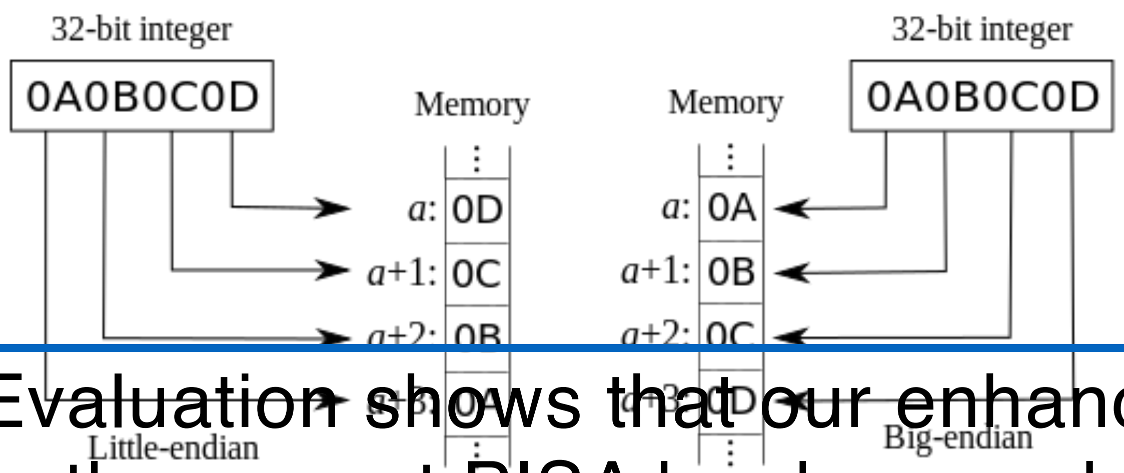
Each action is a single instruction stored in the small buffer!

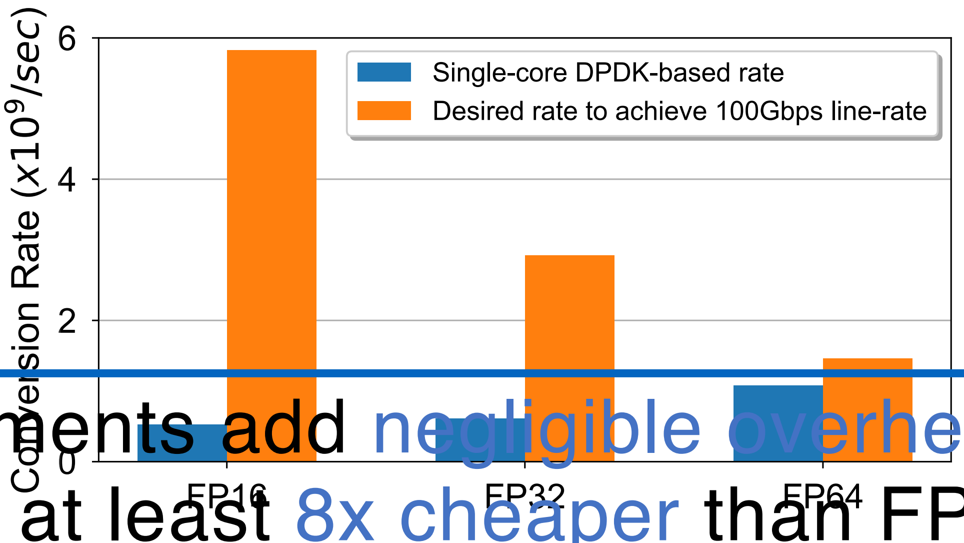
Are we done?

- We implement FPISA with P4 in Intel's Tofino-1 and find it not efficient enough.

- Example-1: saturated VLIW instruction slots –> limited data parallelism
  - Enhancement: 2-operand shift instruction –> *"shift [operand0] [operand1]"*

- Example-2: CPU-network endianness difference -> conversion overhead on end-host
  - Enhancement: byte-wise shuffling in switch pipeline/parser



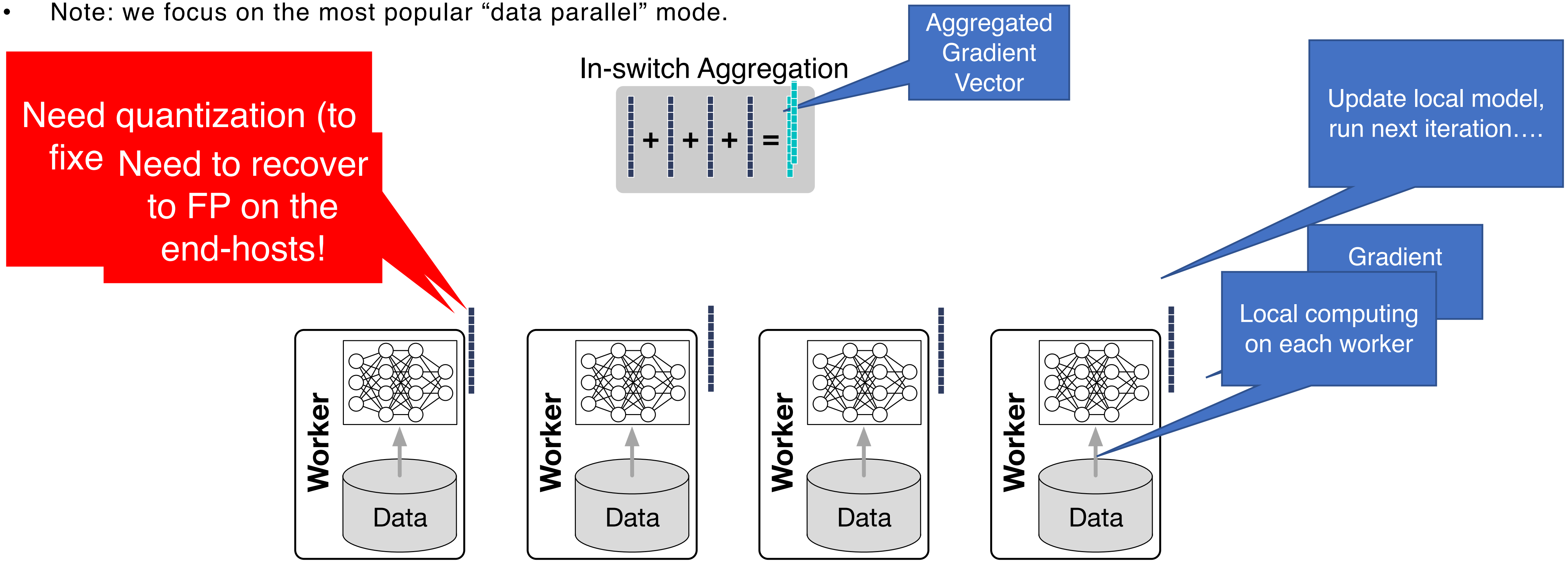


Endianness conversion rate that a core can achieve and that is desired to achieve 100Gbps line-rate.

Evaluation shows that our enhancements add negligible overhead to the current PISA hardware, while at least 8x cheaper than FPU.

Usecase: In-network aggregation for distributed ML training

- What's the procedure of data communication in state-of-the-art frameworks?
  - Note: we focus on the most popular "data parallel" mode.



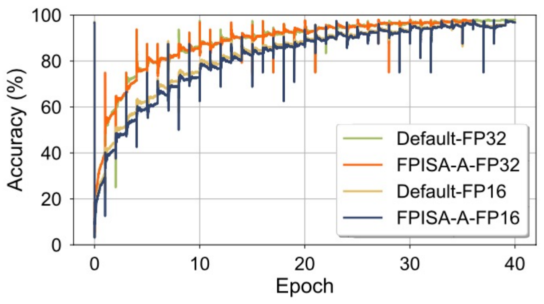
Need quantization (to
fixe
Need to recover to FP on the end-hosts!
In-switch Aggregation
+ + + =
Aggregated Gradient Vector
Update local model, run next iteration….
Gradient
Local computing on each worker
Worker
Data
Worker
Data
Worker
Data
Worker
Data

Evaluation

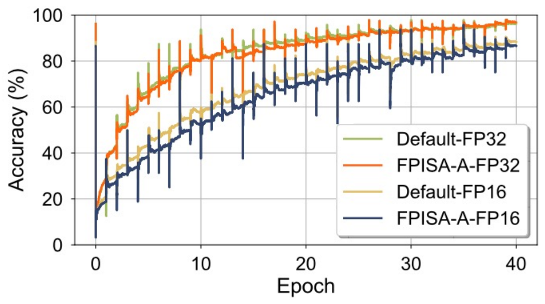
- Given the aforementioned hardware limitation, we develop a C program exactly simulating FPISA addition behavior (both FP32 and FP16) for model convergence evaluation.
- We also leverage the SwitchML (NSDI’21) framework to evaluate the (emulated) end-to-end training time speedup in a real cluster.

Evaluation – Training accuracy and convergence

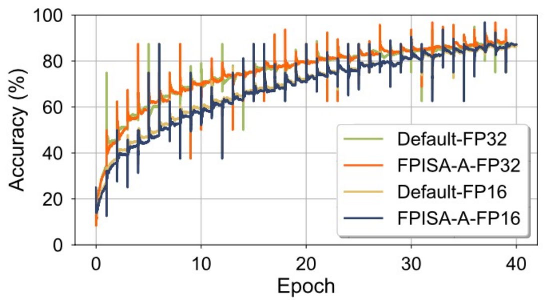
- We apply FPISA's addition (both FP132 and FP16) to models training, and compare the accuracy curves against the ones generated with default standard FP addition.



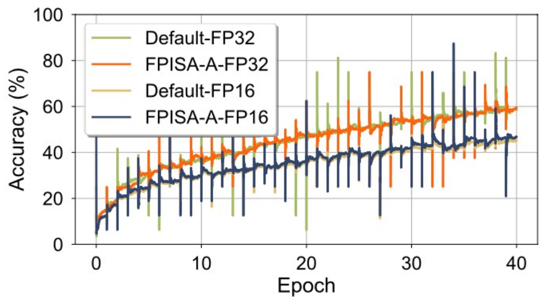
(a) GoogleNet.



(b) ResNet-50.



(c) VGG19.

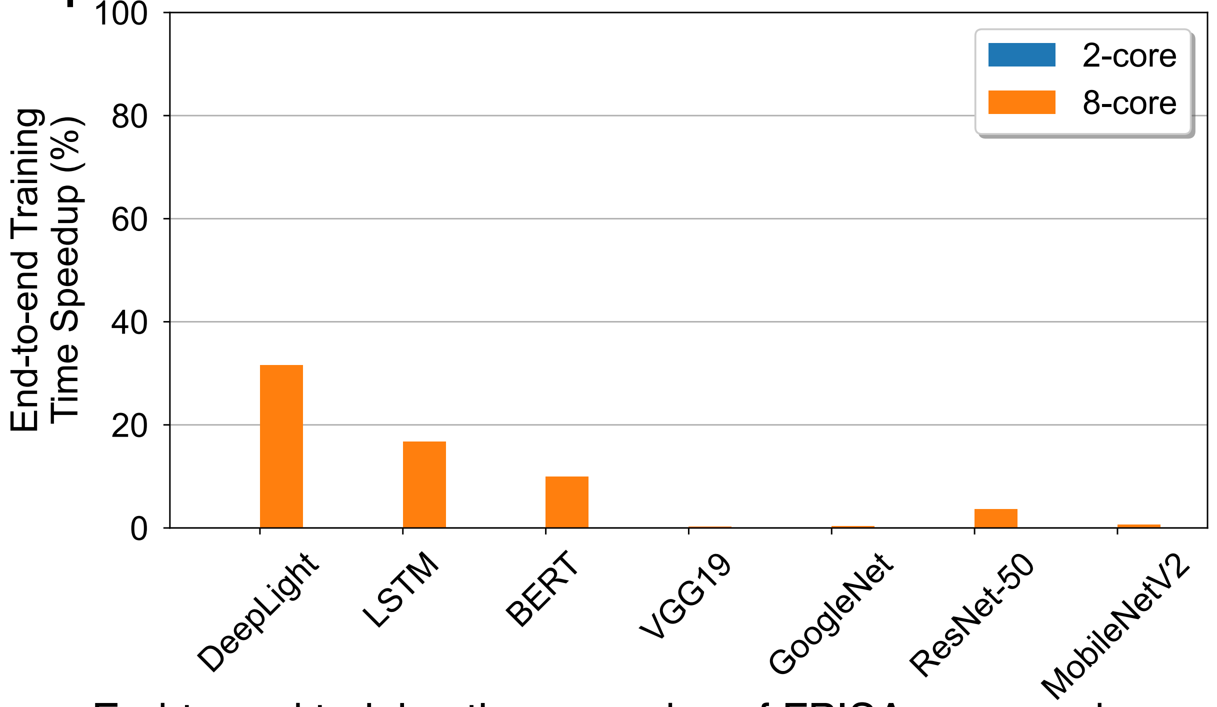


(d) MobileNetV2.

FPISA has negligible impact on trained model's convergence.

Evaluation – Training time speedup

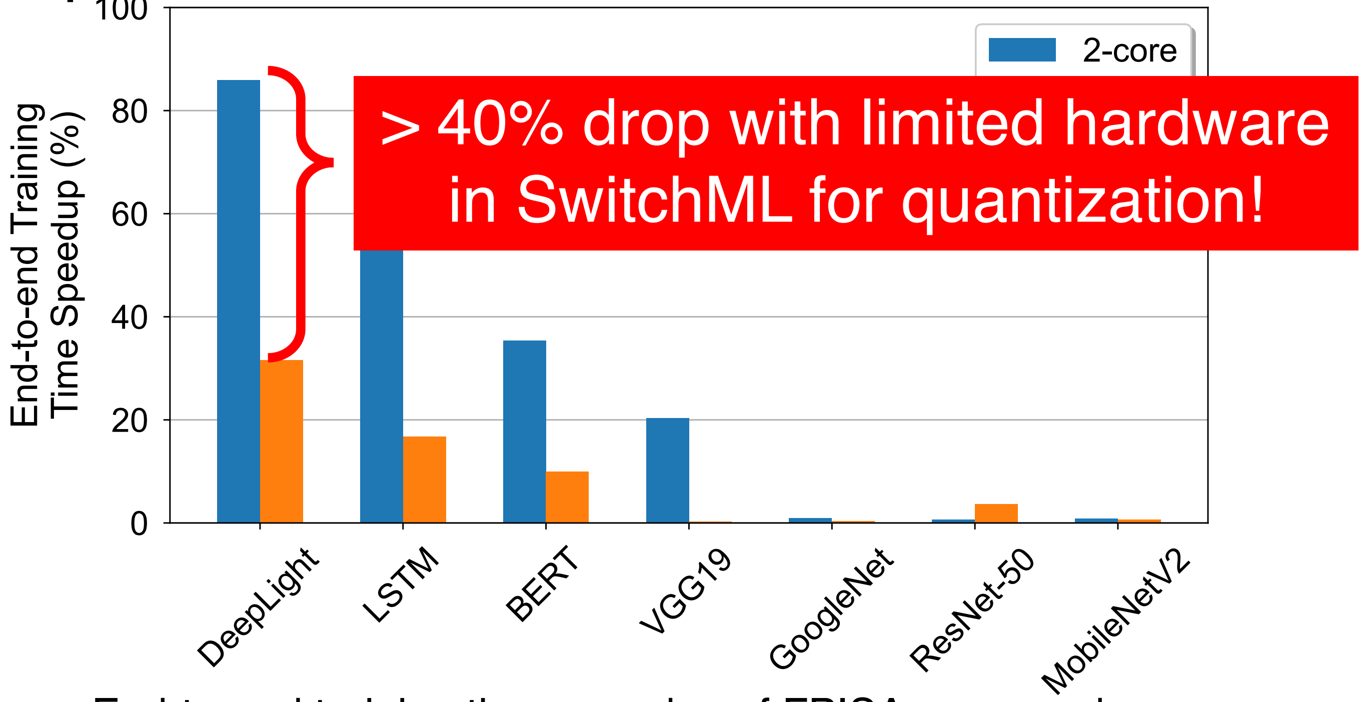
- We compare FPISA’s training time with fixed point based SwitchML, which conducts quantization with 2 or 8 CPU cores.



End-to-end training time speedup of FPISA compared to the default SwitchML with 8 cores.

Evaluation – Training time speedup

- We compare FPISA's training time with fixed point based SwitchML, which conducts quantization with 2 or 8 CPU cores.



End-to-end training time speedup of FPISA compared to the default SwitchML with 2 cores.

FPISA can bring training speedup as well as efficient end-host resource usage compared to the state-of-the-art solutions.

More details in the paper

- FPISA's error and precision analysis.
- Error-tolerance and numerical characteristics of gradient aggregation in distributed training.
- GPU's potential for gradient quantization.
- Additional FP features and advanced FP operations in PISA.
- ……

Conclusion

- Floating point is an important format that is desirable to be supported on modern programmable dataplane with low cost and high flexibility.
- We Propose FPISA approach and a couple of cheap hardware enhancements, which, together, store and operate floating-point numbers in common PISA pipeline.
- Our evaluation on distributed ML training shows that FPISA can significantly facilitate the application execution and reduce end-host resource usage.

Questions? Contact me!
yifany3@Illinois.edu