Folding based zkLLM

IVC

IVC (Incrementally Verifiable Computation) is a new primitive in Folding ZK. Based on Folding, IVC can be constructed more efficiently, where the prover proves the correctness of incremental computation $y = F^{(n)}(x)$.

IVC from SNARK

Previously, IVC was constructed based on SNARK, and the recursive circuit included computation logic and SNARK verification logic.

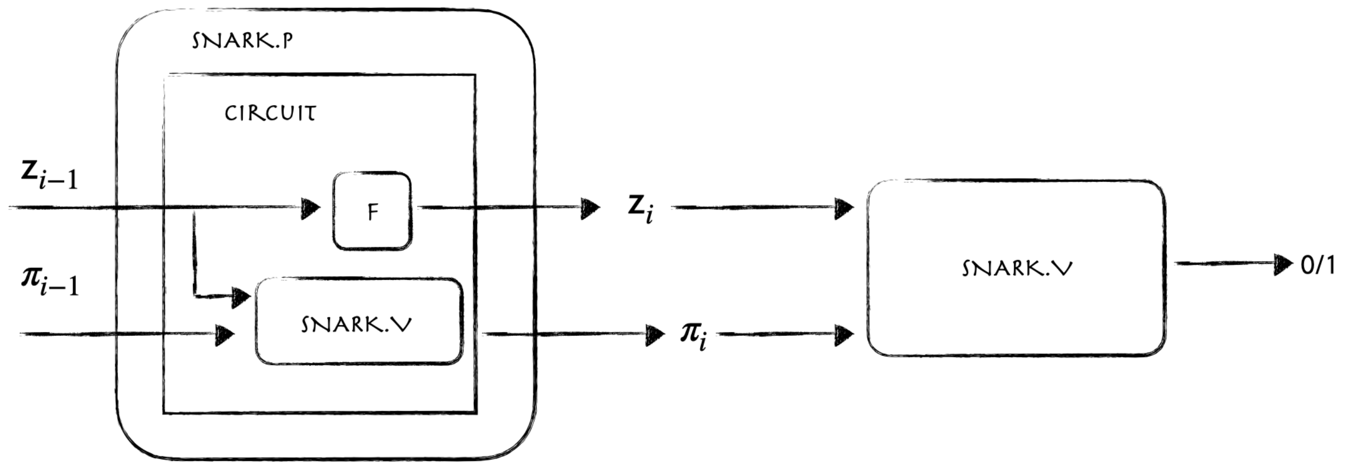


Fig 1.1

Since the SNARK verification logic needs to be expressed in the circuit, and SNARK verification logic often involves some circuit-unfriendly operations, such as pairing and non-native operations, the SNARK-based IVC construction has a large recursive overhead.

IVC from Folding

The IVC construction based on Folding replaces the SNARK verifier circuit with the Folding verifier circuit, greatly reducing the recursive overhead. In Nova, the prover only needs to perform 2 MSM operations of $\mathcal{O}(C)$ scale and a small amount of hashing for each step, without requiring FFT. Practice shows that Nova's recursive overhead is around 10,000 constraints.

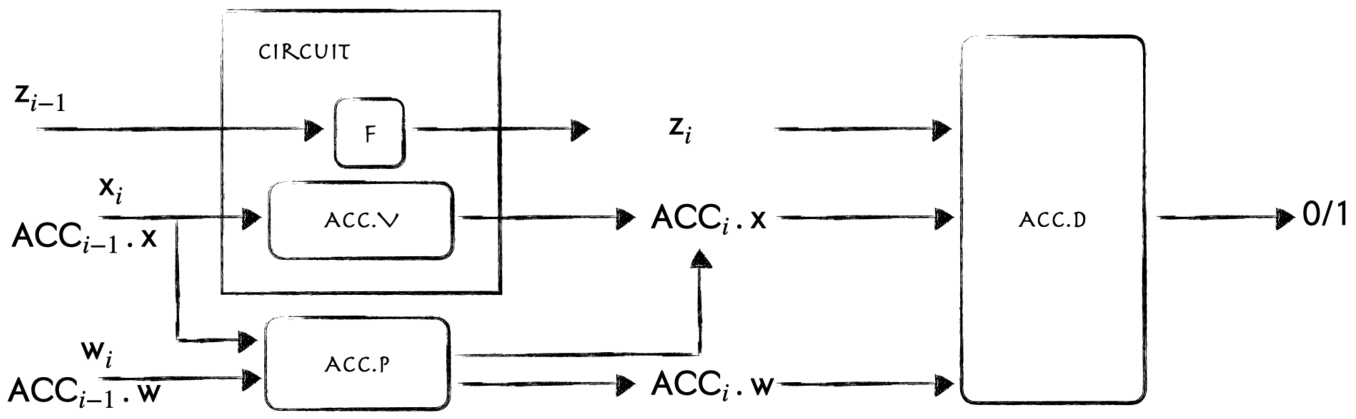


Fig 1.2

NIVC

IVC requires the computation logic of each iteration to be the same, which has a gap with practical applications. For example, a CPU executes arbitrary instructions from the instruction set in each iteration. This requires the use of NIVC. NIVC (Non-uniform IVC) allows the computation function executed in each iteration to vary within the function set.

To construct NIVC, a naive method is to "use a universal circuit to express multiple step functions", which is the commonly used Selector scheme nowadays: laying out all instruction circuits and then activating one of the step instruction circuits through a Selector. This approach has a major drawback: the scale of the universal circuit will expand to the sum of all instruction circuits. In a VM or LLM, an instruction circuit represents a supported instruction. In this case, if the supported instruction set contains a large number of instructions, the final universal circuit will have a large expansion. This path is not good, so we have to find another way: is there a Folding scheme that supports multiple step functions?

SuperNova

The answer is SuperNova[1], an extension of Nova[2]. Its outstanding feature is that the cost of each step of the prover is only proportional to the size of the instruction circuit invoked by the program, which is extremely advantageous when the instruction set is complex.

Folding More

SuperNova only supports R1CS. In practical applications, circuits with richer expressiveness may be needed, such as custom gates and lookups. There are many existing schemes exploring how to fold circuits with richer expressiveness, such as Sangria[3], Origami[4], HyperNova[5], and Protostar[6].

zkLLM

Design

We construct a RAM machine that supports an instruction set $\mathsf{IS} = F_{i \in [\ell]}$ of size ℓ. This RAM machine has 3 control registers ts, pc, flag for flow control; 1 general-purpose register $[\mathsf{gpr}]$ for recording the commitment of a general-purpose register of length k ; 1 stack pointer register and 1 stack register sp , $[\mathsf{sm}]$ for recording the commitment of a stack pointer and stack memory of length m , respectively. in and out are used to record input and output, and $[\mathsf{p}]$ is used to record the input program.

ts	pc	flag	[gpr]	sp	[sm]	in	out	[p]

Table 3.1

The design distinguishes between general-purpose registers and stack memory, referring to the design of registers and memory in typical computer architectures. When operating only on registers, the overhead is smaller because the length of registers is limited and the cost of opening their commitments is small.

For each opcode, an corresponding augmented circuit is defined:

$F'_{j\in[\ell]}(\mathsf{vk}, \mathsf{U}_i, \mathsf{u}_i, (\mathsf{ts}_i, \mathsf{pc}_i, \mathsf{ip}_i, \mathsf{flag}_i, \mathsf{sp}_i, [\mathsf{sm}]_i, \mathsf{in}_0, \mathsf{out}_i, [\mathsf{gpr}]_i, [\mathsf{p}]_0), \bar{T}) \rightarrow x$:

1. Update $\mathsf{ts}_{i+1} \leftarrow \mathsf{ts}_i + 1$
2. Calculate the instruction $[\mathsf{ip}]_{i+1} \in \mathbb{Z}^*_{\ell+1} \leftarrow \varphi(\mathsf{pc}_i, [\mathsf{p}]_0)$ to be executed in the current step based on the pc_i input from the previous step
3. If $\mathsf{ts}_i == 0$ (for INIT):
    a. Initialize the running instance list $\mathsf{U}_{i+1} \leftarrow \mathsf{u}^{\ell}_{\perp}$
    b. Check that in_0 is correctly uploaded to gpr , which is a index-lookup check, as shown in Figure 3.1

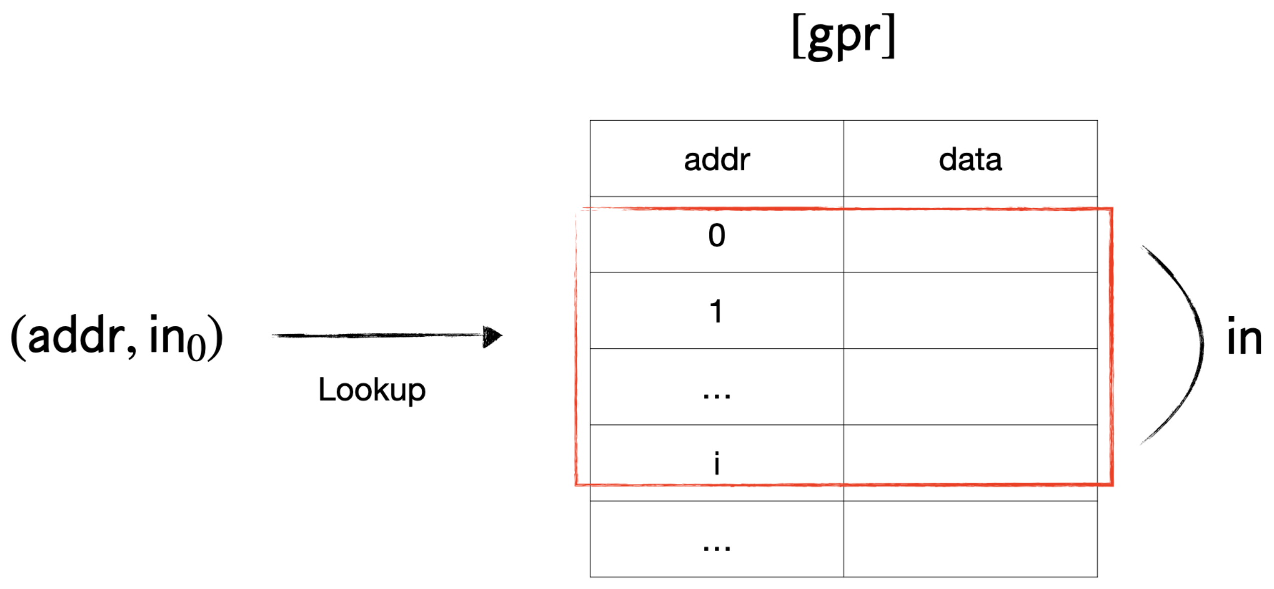


Fig 3.1

c. Verify $[\mathsf{sm}]_i = [0^m]$, $\mathsf{pc} = 0$, $\mathsf{flag} = 0$, $\mathsf{sp} = 0$ to ensure correct initialization

Otherwise:

4. Verify $\mathsf{u}_i.x = \mathsf{hash}(\mathsf{vk}, \mathsf{U}_i, \mathsf{ts}_i, \mathsf{pc}_i, \mathsf{ip}_i, \mathsf{flag}_i, \mathsf{sp}_i, [\mathsf{sm}]_i, \mathsf{in}_0, \mathsf{out}_i, [\mathsf{gpr}]_i, [\mathsf{p}]_0)$ to ensure the output of the previous step is the input of the current step
5. Verify $j = \mathsf{ip}_{i+1}$ to ensure the correct instruction circuit is constrained
6. Verify $(\mathsf{u}_i.\bar{E}, \mathsf{u}_i.u) = (0, 1)$ to ensure the augmented circuit strictly holds
7. Update the running instance list $\mathsf{U}_{i+1}[\mathsf{ip}_i] \leftarrow \mathsf{NIFS.V}(\mathsf{vk}[\mathsf{ip}_i], \mathsf{U}_i[\mathsf{ip}_i], \mathsf{u}_i, \bar{T})$, folding the augmented circuit
8. Update the register state according to the opcode
$(\mathsf{pc}_{i+1}, \mathsf{flag}_{i+1}, [\mathsf{gpr}]_{i+1}, \mathsf{sp}_{i+1}, [\mathsf{sm}]_{i+1}) \leftarrow F_j(\mathsf{pc}_i, \mathsf{flag}_i, [\mathsf{gpr}]_i, \mathsf{sp}_i, [\mathsf{sm}]_i)$
9. Output
$x \leftarrow \mathsf{hash}(\mathsf{vk}, \mathsf{U}_{i+1}, \mathsf{ts}_{i+1}, \mathsf{pc}_{i+1}, \mathsf{ip}_{i+1}, \mathsf{flag}_{i+1}, \mathsf{sp}_{i+1}, [\mathsf{sm}]_{i+1}, \mathsf{in}_0, \mathsf{out}_{i+1}, [\mathsf{gpr}]_{i+1}, [\mathsf{p}]_0)$

Notes:

1. $[\mathsf{p}]$, $[\mathsf{gpr}]$, $[\mathsf{sm}]$ represent the commitment of the input vector (general-purpose register), input code, and stack memory
2. p represents the input code, which is a table consisting of $(\mathsf{pc}, \mathsf{ip}, \mathsf{oprand})$, as shown in Figure 3.2

[p]

pc	ip	oprand

Fig 3.2

3. The $\varphi(\cdot)$ in step 2 is a decoder that takes the program p and program counter pc as input and calculates the instruction and operands to be executed in the current step, which is essentially an index-lookup

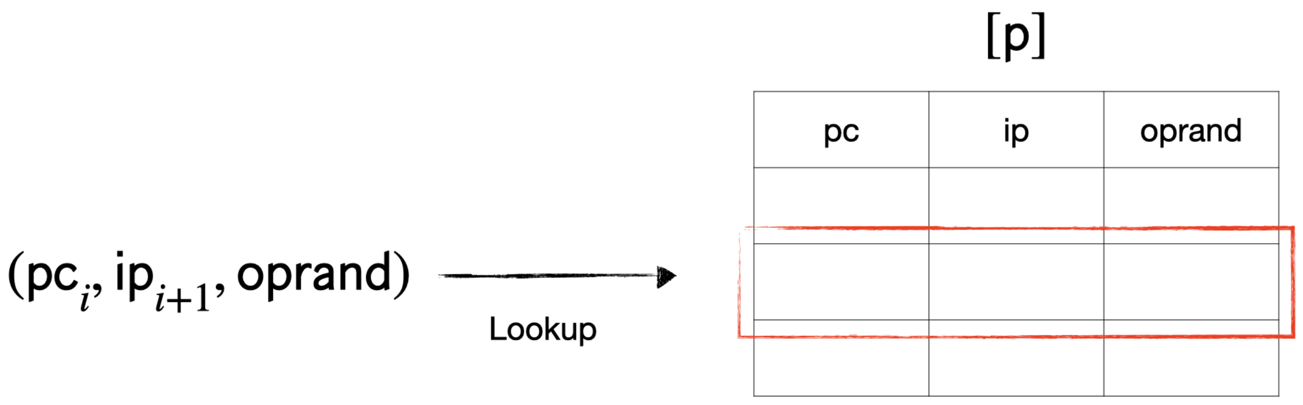


Fig 3.3

4. The opcode updates some registers and outputs, such as MUL and ADD modifying the pc ip $[\mathsf{gpr}]$ registers, JMP and JMPC modifying the pc ip registers, and $\mathsf{S\_MUL}$ modifying the sp $[\mathsf{sm}]$ registers

The prover updates the proof Π using the trace at each step:

$\mathcal{P}(\mathsf{pk}, \Pi_i) \rightarrow \Pi_{i+1}$:

1. Parse the proof Π_i of step i as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), (\mathsf{ts}_i, \mathsf{pc}_i, \mathsf{ip}_i, \mathsf{flag}_i, \mathsf{sp}_i, [\mathsf{sm}]_i, \mathsf{in}_0, \mathsf{out}_i, [\mathsf{gpr}]_i, [\mathsf{p}]_0))$
2. If $\mathsf{ts}_i == 0$:
   a. Initialize $((\mathsf{U}_{i+1}, \mathsf{W}_{i+1}), \bar{T}) \leftarrow (\mathsf{u}_\perp^\ell, \mathsf{w}_\perp^\ell, \mathsf{u}_\perp.\bar{E})$
   b. Initialize $[\mathsf{sm}]_i = [0^m]$, $\mathsf{pc} = 0$, $\mathsf{flag} = 0$, $\mathsf{sp} = 0$, $[\mathsf{gpr}]$

Otherwise:

3. Update the corresponding running instance according to the instruction pointer $(\mathsf{U}_{i+1}[\mathsf{ip}_i], \mathsf{W}_{i+1}[\mathsf{ip}_i], \bar{T}) \leftarrow \mathsf{NIFS.P}(\mathsf{pk}[\mathsf{ip}_i], (\mathsf{U}_i[\mathsf{ip}_i], \mathsf{W}_i[\mathsf{ip}_i]), (\mathsf{u}_i, \mathsf{w}_i))$
4. Calculate the instruction $[\mathsf{ip}]_{i+1} \in \mathbb{Z}_{\ell+1}^* \leftarrow \varphi(\mathsf{pc}_i, [\mathsf{p}]_0)$ to be executed in the current step
5. Calculate the trace of the current step's augmented circuit

$$(\mathsf{u}_{i+1}, \mathsf{w}_{i+1}) \leftarrow$$
$$\mathsf{trace}(F'_{\mathsf{ip}_{i+1}}, \mathsf{vk}, \mathsf{U}_i, \mathsf{u}_i, \mathsf{ts}_i, \mathsf{pc}_i, \mathsf{ip}_i, \mathsf{flag}_i, \mathsf{sp}_i, [\mathsf{sm}]_i, \mathsf{in}_0, \mathsf{out}_i, [\mathsf{gpr}]_i, [\mathsf{p}]_0, \bar{T})$$

6. Update the proof $\Pi_{i+1} \leftarrow ((\mathsf{u}_i, \mathsf{w}_i), \mathsf{ip}_{i+1})$

The verifier obtains the proof Π_i from the last folding

$\mathcal{V}(\mathsf{vk}, \Pi_i) \rightarrow \{0, 1\}$:

If $\mathsf{ts}_i == 0$:

1. Verify that in_0 is correctly uploaded to gpr
2. Verify $[\mathsf{sm}]_i = [\mathsf{0}^m]$, $\mathsf{pc} = 0$, $\mathsf{flag} = 0$, $\mathsf{sp} = 0$ to ensure correct initial values

Otherwise:

3. Parse the proof Π_i as $((\mathsf{U}_i, \mathsf{W}_i), (\mathsf{u}_i, \mathsf{w}_i), (\mathsf{ts}_i, \mathsf{pc}_i, \mathsf{ip}_i, \mathsf{flag}_i, \mathsf{sp}_i, [\mathsf{sm}]_i, \mathsf{in}_0, \mathsf{out}_i, [\mathsf{gpr}]_i, [\mathsf{p}]_0))$
4. Verify $\mathsf{u}_i.x = \mathsf{hash}(\mathsf{vk}, \mathsf{U}_i, \mathsf{ts}_i, \mathsf{pc}_i, \mathsf{ip}_i, \mathsf{flag}_i, \mathsf{sp}_i, [\mathsf{sm}]_i, \mathsf{in}_0, \mathsf{out}_i, [\mathsf{gpr}]_i, [\mathsf{p}]_0)$
5. $\varphi(\mathsf{pc}_i, [\mathsf{p}]_0) = \mathsf{endvar}$
6. Verify $(\mathsf{u}_i.\bar{E}, \mathsf{u}_i.u) = (0, 1)$
7. For $F'_{\mathsf{ip_i}}$, verify the $(\mathsf{u}_i, \mathsf{w}_i)$ of the last iteration
8. For all F' , verify all $(\mathsf{U}_i[j], \mathsf{W}_j)_{j \in [\ell]}$

Opcode Examples

INIT

INIT is used to upload the initial input in_0 agreed upon by both parties (including the consensus of private input) to the predetermined addresses of gpr . The INIT circuit needs to constrain:

1. $\mathsf{ts}_i = 0$, $[\mathsf{sm}]_i = [\mathsf{0}^m]$, $\mathsf{pc} = 0$, $\mathsf{flag} = 0$, $\mathsf{sp} = 0$
2. The values at the predetermined positions of gpr are equal to the input, $\mathsf{in}_0 = \mathsf{OPEN}(\mathsf{gpr}_i, \mathsf{addr})$, where OPEN can be implemented as the index lookup constraint shown in Figure 3.1
3. Update $\mathsf{pc_{i+1}} = \mathsf{pc_i} + 1$, and the rest of the register states remain unchanged

Note: The maximum length of the initial input is specified as d , and less than that is padded with 0.

For example, if the maximum initial length is $d = 4$, and the initial input is $\mathsf{in}_0 = [1, 2, 0, 0]$, then 2 zeros are padded. The initial register state satisfying the constraints should be:

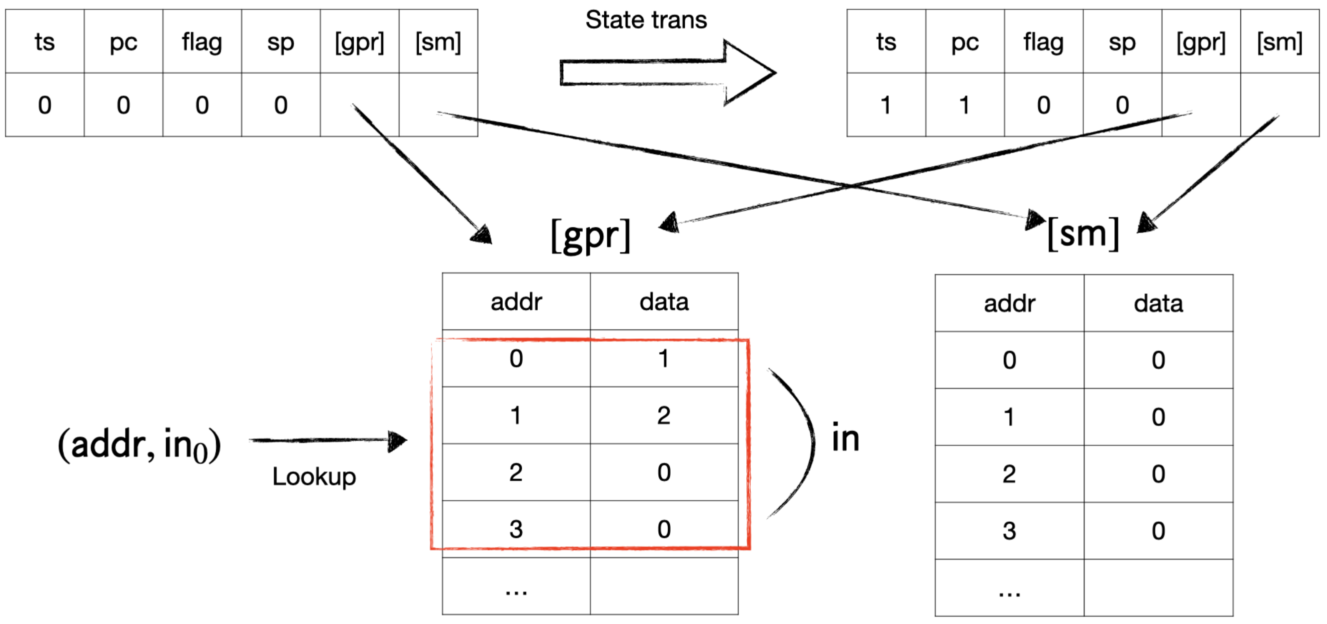


Fig 3.4: INIT legal state

ADD1_4

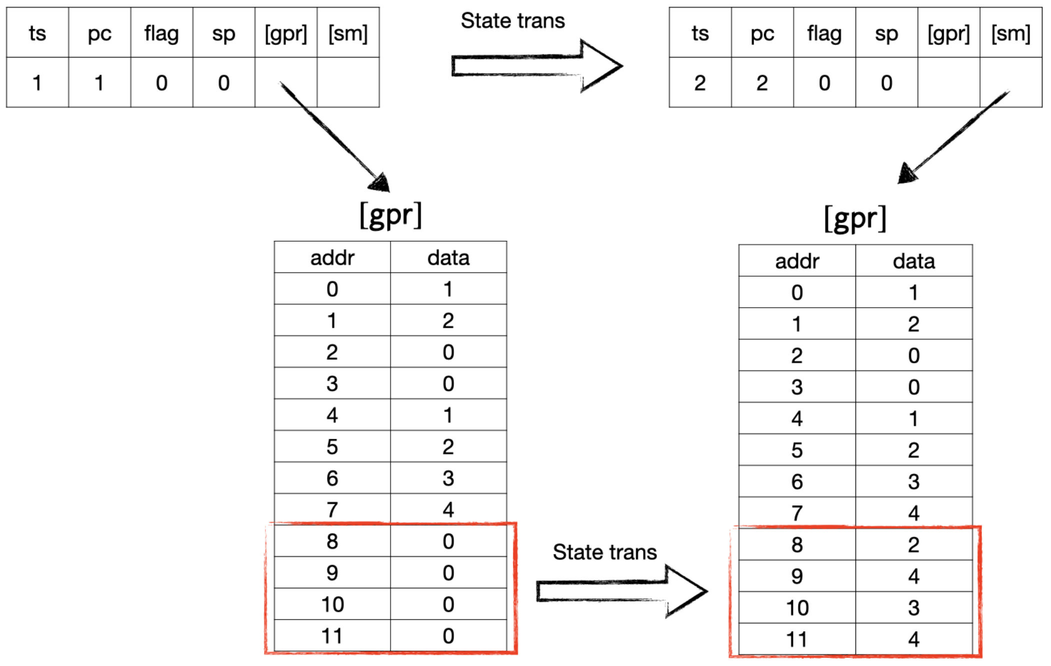
$\text{ADD1\_4 addr}_0 \text{ addr}_1 \text{ addr}_2$ is used for the addition of two 1*4 tensors at specified addresses, and the result is stored at the specified address. The update requires constraints:

1. The computation at the specified addresses of gpr is correct:
   $\text{left} \leftarrow \text{OPEN}([\text{gpr}]_i, \text{addr}_0)$, $\text{right} \leftarrow \text{OPEN}([\text{gpr}]_i, \text{addr}_1)$,
   $\text{output} \leftarrow \text{OPEN}([\text{gpr}]_{i+1}, \text{addr}_2)$, $\text{output} = \text{left} + \text{right}$
2. Update $\text{pc}_{\text{i}+1} = \text{pc}_\text{i} + 1$, $[\text{gpr}]_{i+1} = \text{UPDATE}([\text{gpr}]_i, \text{addr}_2)$

Note: ← represents generating an intermediate variable and constraining it.

The UPDATE constraint is $[\text{gpr}]_{i+1} = [\text{gpr}]_i + \sum_{\text{addr}_2} ([\text{gpr}]_{i+1}^{\text{addr}_2} - [\text{gpr}]_i^{\text{addr}_2}) \cdot [L^{\text{addr}_2}(X)]$, and its complexity is related to the number of modified addresses.

For example, $\text{ADD1\_4}\ 0\ 4\ 8$ adds $[1, 2, 0, 0]$ at address 0 and $[1, 2, 3, 4]$ at address 4, and places the result at address 8.



ts	pc	flag	sp	[gpr]	[sm]
1	1	0	0		

ts	pc	flag	sp	[gpr]	[sm]
2	2	0	0		

[gpr]

addr	data
0	1
1	2
2	0
3	0
4	1
5	2
6	3
7	4
8	0
9	0
10	0
11	0

[gpr]

addr	data
0	1
1	2
2	0
3	0
4	1
5	2
6	3
7	4
8	2
9	4
10	3
11	4

$$[\mathrm{gpr}]_{i+1} = [\mathrm{gpr}]_i + \sum_{\mathrm{addr}_2 \in [8,9,10,11]} ([\mathrm{gpr}]_{i+1}^{\mathrm{addr}_2} - [\mathrm{gpr}]_i^{\mathrm{addr}_2}) \cdot [L^{\mathrm{addr}_2}(X)]$$

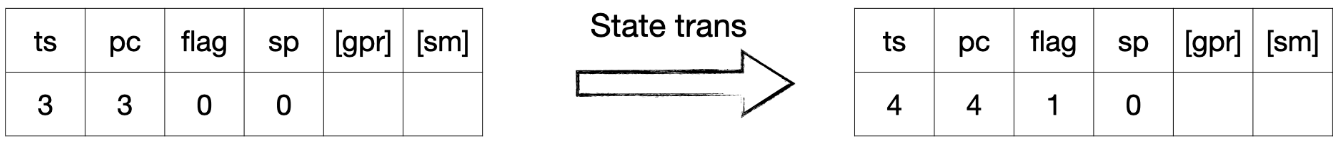
Fig 3.5: ADD1_4 legal state (ignoring the unchanged stack memory and output)

LE1_4

$\mathrm{LE1\_4}$ addr_0 addr_1 is used to compare two 1*4 tensors at specified addresses, and the flag is updated based on the comparison result. The circuit needs to constrain:

1. If a ⩾ b: $\mathrm{flag}_{i+1} = 1$
2. Update $\mathrm{pc}_{i+1} = \mathrm{pc}_i + 1$

For example, LE1_4 0 4 compares [1,2,0,0] at address 0 with [1,2,3,4] at address 4.



ts	pc	flag	sp	[gpr]	[sm]
3	3	0	0		

ts	pc	flag	sp	[gpr]	[sm]
4	4	1	0		

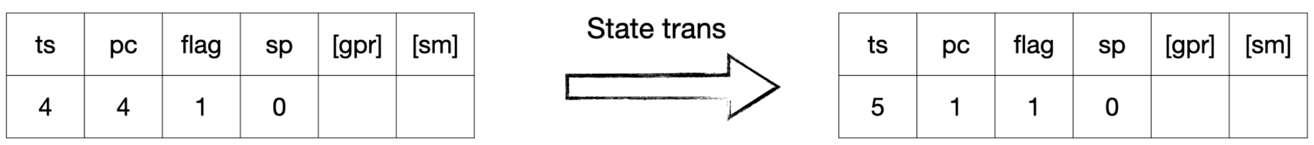
Fig 3.6: LE1_4 legal state (ignoring the unchanged registers, stack memory, and output)

JMPC

JMPC addr_0 addr_1 is used to jump based on flag . The circuit needs to constrain:

1. If flag_{i+1} holds, update $\mathrm{pc}_{i+1} = \mathrm{addr}_0$, otherwise update $\mathrm{pc}_{i+1} = \mathrm{addr}_1$

For example, $\mathsf{JMPC}\ 1\ 4$ sets **pc** to 1



ts	pc	flag	sp	[gpr]	[sm]
4	4	1	0		

ts	pc	flag	sp	[gpr]	[sm]
5	1	1	0		

FIg 3.7: **JMPC** legal state (ignoring the unchanged registers, stack memory, and output)

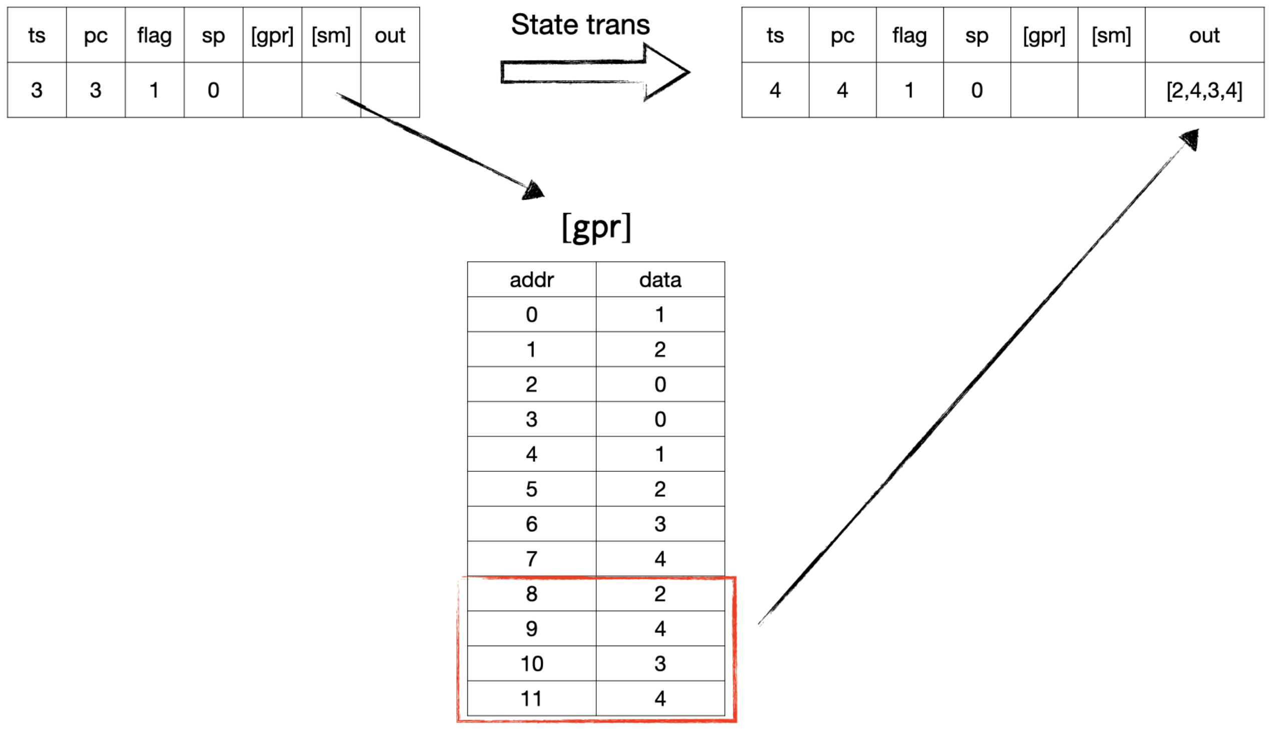
RETURN

RETURN addr is used to download the data at the specified address in **addr** to **out** . The circuit needs to constrain:

1. Update $\mathsf{out}_{i+1} = \mathsf{OPEN}([\mathsf{gpr}]_i, \mathsf{addr})$
2. Update $\mathsf{pc}_{i+1} = \mathsf{pc}_{\text{end}}$

The maximum length d of the output **out** is specified.

For example, $\mathsf{RETURN}\ 8$ outputs $[2,4,3,4]$ at address 8 to **out**



ts	pc	flag	sp	[gpr]	[sm]	out
3	3	1	0			

ts	pc	flag	sp	[gpr]	[sm]	out
4	4	1	0			[2,4,3,4]

addr	data
0	1
1	2
2	0
3	0
4	1
5	2
6	3
7	4
8	2
9	4
10	3
11	4

Fig 3.8: **RETURN** legal state (ignoring the unchanged stack memory)

END

`END` is used to indicate the end of the program execution. The circuit needs to constrain:

1. All states remain unchanged

References

[1] https://eprint.iacr.org/2022/1758.pdf

[2] https://eprint.iacr.org/2022/1758.pdf

[3] https://geometry.xyz/notebook/sangria-a-folding-scheme-for-plonk

[4] https://hackmd.io/@aardvark/rkHqa3NZ2

[5] https://eprint.iacr.org/2023/573.pdf

[6] https://eprint.iacr.org/2023/620.pdf