Fengwei Zhan

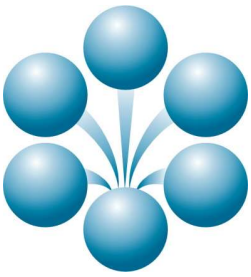
High-speed asynchronous digital interfaces:

Exploiting the spatiotemporal correlations of event-based sensor data

Department of Microelectronics
Faculty of Electrical Engineering, Mathematics and Computer Science

Supervisor: Dr. Charlotte Frenkel
Thesis advisor: Prof. Dr. Kofi A.A. Makinwa

2023



High-speed asynchronous digital interfaces: Exploiting the spatiotemporal correlations of event-based sensor data

Thesis

submitted in partial fulfillment of the
requirements for the degree of

Master of Science

in

Electrical Engineering

by

Fengwei Zhan

Thesis Committee Members

Dr. Charlotte Frenkel
Prof. Dr. Kofi A.A. Makinwa
Dr. Chang Gao
Mr. Nicolas Chauvaux

Electronic Instrumentation Group
Department of Microelectronics
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Delft University of Technology

Abstract

With the introduction of event-based cameras, such as the dynamic vision sensor (DVS), new opportunities have arisen for low-latency real-time visual data processing. Unlike traditional frame-based cameras that capture entire frames at fixed intervals, each pixel in an event-based camera operates asynchronously, generating an event whenever its brightness change exceeds a certain threshold. Although DVS sensors inherently surpass traditional frame-based cameras in capturing transient, high-speed phenomena, their performance bottleneck is usually located in their address event representation (AER) readout interfaces. The commonly used row-scanning synchronous AER, which encodes events in a full row at once, offers high throughput. However, this approach also introduces inherent delays that limit its use in applications requiring high temporal resolution. Conversely, while AER schemes based on asynchronous digital circuits surpass synchronous schemes in temporal resolution, their event-by-event transmission approach limits their overall throughput.

This work proposes a novel high-speed asynchronous AER interface, leveraging spatiotemporal correlations in DVS event-based data, to optimize the tradeoff between temporal resolution and throughput. Supported by the recently proposed open-source asynchronous design toolkit (ACT) flow for asynchronous digital circuits, we propose an address fuser to be integrated into the hierarchical token ring (HTR) AER scheme. This address fuser creates a spatiotemporal window to exploit the inherent spatiotemporal correlations in DVS data. After verification at both switch- and transistor-level simulations, we benchmarked our design against the conventional HTR AER scheme using a representative set of input scenarios. Our design achieved 196% of the throughput for multi-event transmissions when all pixels were activated simultaneously, at the expense of an acceptable 18% latency increase for single-event transmissions with a 10-ns temporal window.

Acknowledgments

Firstly, I would like to thank my thesis supervisor Dr. Charlotte Frenkel, and Ph.D. student Nicolas Chauvaux for their invaluable assistance and support throughout this project. Their extensive knowledge and insights not only enlightened me in the professional realm but also enhanced my research methodologies and thinking. Their meticulous comments and feedback immensely helped me refine and shape my thesis.

Then, I would like to thank all the professors and fellow students I've encountered during my two years at TU Delft. The time spent here has truly been unforgettable. I am also very grateful to Prof. Dr. Kofi Makinwa and Dr. Chang Gao for being on my thesis committee and giving valuable suggestions for my thesis work.

Lastly, I would like to express my gratitude towards my parents for their unconditional support, patience, and encouragement throughout my life.

Contents

Abstract **iii**

Acknowledgments **v**

1 Introduction **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 3
1.3 Thesis Outline . . . 4

2 Background **5**
2.1 Asynchronous digital circuits . . . 5
2.2 The asynchronous circuit toolkit (ACT) design flow . . . 6
2.2.1 Style and data encoding methods . . . 6
2.2.2 Asynchronous circuits design with ACT . . . 9
2.3 Review of key state-of-the-art address event representation (AER) schemes . 17
2.3.1 Synchronous row scanning scheme . . . 17
2.3.2 Asynchronous Binary arbiter tree scheme . . . 18
2.3.3 Asynchronous token-ring scheme . . . 20
2.3.4 Summary and discussion . . . 21

3 Address-fused Hierarchical Token Ring (HTR) design **23**
3.1 Design considerations . . . 23
3.2 Architecture overview . . . 23
3.3 Address fuser . . . 27
3.4 Low-level server . . . 28
3.5 High-level server . . . 30
3.6 Request merger and data merger . . . 32
3.7 Counter . . . 34
3.8 Output buffer . . . 36
3.9 Parameterization of the design . . . 37

4 Results and discussion **39**
4.1 Overview . . . 39
4.2 Verification of the design . . . 39
4.2.1 Simulation setup . . . 39
4.2.2 Switch-level simulation . . . 39
4.2.3 Transistor-level simulation . . . 40
4.2.4 Layout and routing of address fuser . . . 43
4.3 Comparison between address-fused HTR and conventional HTR . . . 44
4.4 Software verification of spatiotemporal correlation . . . 49
4.5 Discussion . . . 50

5 Conclusion **53**
5.1 Claims 53
5.2 Future work 53

A Appendix name **55**
A.1 HSE and PRS of address fuser 55
A.2 Working principle of the used arbiter 56
A.3 HSE of Lserver 56
A.4 HSE of Hserver 57
A.5 Sizing in the ACT flow 57

List of Figures

1.1 Difference between frame-based standard camera output and event-based DVS output in capturing a black dot rotating around the center within a gray circle. The results from the frame-based camera are frames at millisecond intervals. On the other hand, the DVS is capable of rapidly capturing changes in the scene and producing a stream of events at microsecond intervals. Modified from [1]. . . . 2

1.2 The essence of the AER bus. Adapted from [2] . . . 2

1.3 Schematic diagram of falling raindrops in the field of view of a DVS, and each cluster is formed by the accumulation of events with a duration of 0.4ms. OFF and ON events are generated by decreases and increases in brightness, respectively. Events typically occur close in time and space, highlighting their inherent spatiotemporal correlations. Modified from [3]. . . . 3

2.1 Timing diagrams of 4-phase handshake and 2-phase handshake. For the 4-phase handshake, 'Req' and 'Ack' return to their initial states after completing the handshake; for the 2-phase handshake, there are two possible initial states for 'Req' and 'Ack'. This figure is modified from [4]. . . . 6

2.2 Schematic diagram of bundled-data encoding, the example timing diagram for the transmission of '11' data is on the bottom. . . . 7

2.3 Schematic diagram of dual-rail encoding, the example timing diagram for the transmission of '11' data is on the bottom, and uses 5 wires. . . . 8

2.4 Schematic diagram of 1-of-N encoding, the example timing diagram for the transmission of '11' data is on the bottom. . . . 9

2.5 Steps of the ACT flow, examples are provided on the right. . . . 11

2.6 The state of each variable in the HSE code of the buffer example during each execution step. The 'X' in the table represents a don't care state, which cannot be used to infer the final expression and is typically linked to the external inputs of the modules. . . . 13

2.7 A typical state-holding gate. Its PRS expressions are $\tilde{}a\& \tilde{} b \rightarrow c+$ and $a\&b \rightarrow c-$, and due to the asymmetry of pull-up network and pull-down network, a weak keeper is required at the output to hold the previous value. The output _c is the inverted form of c. . . . 15

2.8 Schematic diagram of the row (column) scanning scheme, modified from [5]. It uses control logic to operate the column (row) driver, which will sequentially scan each column. . . . 18

2.9 Schematic diagram of the arbiter tree structure. Each arbiter cell has two inputs and one output channel, and each channel consists of a Req and an Ack wire [6]. . . . 19

2.10 (a). Schematic diagram of the token-ring structure, modified from [7]. It uses a counter to record the location of the token to determine the sending address. (b). Schematic diagram of the hierarchical token rings (HTR) structure, adapted from [8]. It adds an additional layer of Hserver (red blocks) to bypass the token movement of the first-layer server (Lserver), increasing the efficiency of token movement in long-distance moving. 20

3.1 Schematic diagram of the proposed address-fused HTR. 24
3.2 Steps of sending grouped events at once in the proposed design. 26
3.3 (a). Schematic of a 2-pixel address fuser. (b). The internal variables used in this 2-pixel address fuser. 27
3.4 An example timing diagram of the address fuser accompanied by its corresponding CHP and annotated with step numbers. 28
3.5 (a). Schematic diagram of the Lserver. (b). The interconnection between Lservers and address fusers. For brevity, the inputs of address fusers are omitted. 28
3.6 A simple example about requesting tokens between Lservers. The process is shown on the left, and the interconnection of L1, L2, and L3 is shown on the right. 30
3.7 (a). Schematic diagram of the Hserver. (b). The interconnection between Hservers and Lservers. 31
3.8 A simple example about requesting tokens between Lservers via Hservers. The process is shown on the left, and the interconnection of L1, L2, L3, L4, H1, and H2 is shown on the right. 32
3.9 Schematic of the (a). request merger module and (b). data merger. The major difference between a data merger and a request merger is that for a data merger, a delay needs to be added to the 'out.r' OR gate tree to ensure that it propagates slower than the data. 33
3.10 The workflow of request merger and data merger. 34
3.11 Example timing diagram of the N-bit buffer. 35
3.12 Schematic diagram of the counter. 35
3.13 Schematic diagram illustrating the interconnections between the counter, Output buffer, and other modules. 36
3.14 Schematic diagram of the Output buffer. 36
3.15 Schematic diagram of the parameterized address-fused HTR, with the parameters t, N, Y, and Z highlighted in red. 37

4.1 Schematic diagram of a simple 8-input address-fused HTR structure. The signals of interest are marked in red. 40
4.2 Workflow and timing diagram of an 8-input address-fused HTR when p[2], p[3], and p[4] request simultaneously. The sequence numbers in the timing diagram correspond to the steps in the workflow and the signals of interest are highlighted in red. These results were obtained from a switch-level simulation, hence time is shown with arbitrary units (a.u.). 41
4.3 Transistor-level simulation results. For clarity, the numbers at the bottom correspond to the steps presented in the switch-level simulation. 42

4.4 Layout of a two-input address fuser with the Skywater130 PDK, viewed in Magic VLSI, and its corresponding schematic diagram. The abc_acx* names are auto-generated cell names. 44
4.5 Comparison of Spice pre-layout and post-layout simulation results for the of the two-input address fuser. The pre-layout results are in black, while the post-layout results are in red. 45
4.6 Schematic diagram of the three tested input scenarios, a single event, a half frame, and a full frame of (a). Proposed address-fused HTR, and (b). Conventional HTR. 46
4.7 Single-event scenario: overall latency of the conventional HTR and our proposed design. 46
4.8 Half-frame scenario: overall latency of the conventional HTR and our proposed design. 47
4.9 Full-frame scenario: overall latency of the conventional HTR and our proposed design. 48
4.10 (a). Schematic representation of the 256-input configuration, the 4×4 sensing field in the address fuser, and the 2×5 event cluster positioned at the intersection of four adjacent address fusers, with random events highlighted in green. (b). Time needed for event transmission in the random-event scenarios and those with moderate spatiotemporal correlation for both the conventional HTR and our proposed approach. 48
4.11 Trend of the event reduction rate as the size and shape of the spatio-temporal window changes. 49
4.12 Performance comparison the between binary arbiter tree scheme, the token-ring scheme, and the proposed address-fused HTR compared with the conventional HTR scheme. Summarised from [8]. 50

A.1 Schematic of a common arbiter, where 'a' and 'b' are the inputs and they could possibly be both true. '_r' and '_u' represent whether 'a' or 'b' is selected by the arbiter respectively, and there is at most one of them that is low. 'r' and 'u' are the inverted forms of '_r' and '_u' after stabilization, with at most one of them being high. 56

List of Tables

2.1 The 4 possible cases of dual-rail encoding. 8
2.2 The 4 possible cases of 1-of-N encoding. 8
2.3 Advantages and disadvantages of the three introduced schemes. 22

3.1 Functions of each port and internal variable of the Lserver. 29

1 Introduction

Real-time high-temporal-precision image processing plays a critical role in the domains of autonomous driving [9], robotics [10], and the Internet of Things (IoT). While conventional frame-based computation consumes a significant amount of computational resources, event-driven computation with an event-based camera ensures higher temporal resolution and requires much fewer computational resources. Indeed, the event-based camera, also known as the dynamic vision sensor (DVS), works in a different way than a conventional frame-based camera: it detects the brightness changes asynchronously for each pixel. When the brightness change in a pixel exceeds a certain threshold that is usually set by the user, it will generate an event, i.e. a spike. The asynchronously fired spikes are then transmitted through a digital bus for further processing [11]. The difference in operation between the DVS and a traditional frame-based camera is illustrated in Fig 1.1 [1]. For the constantly rotating grey background and dot on the left, the traditional frame-based camera continually outputs the whole image at a fixed frame rate. In contrast, the event-based DVS only outputs the changing parts, which are events from the pixels where the rotating dot is located. This characteristic brings the DVS many advantages over conventional frame-based vision sensors: it can offer higher temporal resolution (in the μs range), lower power consumption (equal to static power in static scenes), and higher dynamic range ($>$ 120dB, against typically 60dB for the conventional frame-based camera) [12].

As the DVS fires events asynchronously and with a μs-range temporal resolution, it requires high-speed interfaces to transmit events. The address event representation (AER) interface [13] has been adopted by most DVS designers because of its high speed, low latency, and compatibility with event-based neuromorphic processing. The essence of the AER bus is shown in Fig. 1.2 [2]. In the AER interface, the addresses of the requesting pixels are transmitted in real-time by a sender-side encoder and received by a receiver-side decoder. The AER encoder shown on the left side of Fig. 1.2 serves two primary functions: encoding and arbitration. The encoder waits for events and encodes them based on their addresses, then immediately sends them to the output bus. The arbitration function is needed because when multiple events simultaneously request to be sent, it can prevent conflicts and determine their sending sequence. The function of the AER receiver's decoder is relatively straightforward. It only needs to decode the encoded event from the AER sender. The AER interface is essentially a kind of time-division multiplexing method. It eliminates the need for excessive wiring among pixels while retaining the temporal information of events, where all pixels of the sender, such as numbers 0, 1, 2, and 3 in Fig. 1.2, appear virtually connected to the corresponding pixels 0, 1, 2, and 3 of the receiver.

1.1 Motivation

The fast growth of DVS sensors outlines new opportunities for the real-time processing of visual data. Indeed, by design, they offer significant advantages over traditional frame-based

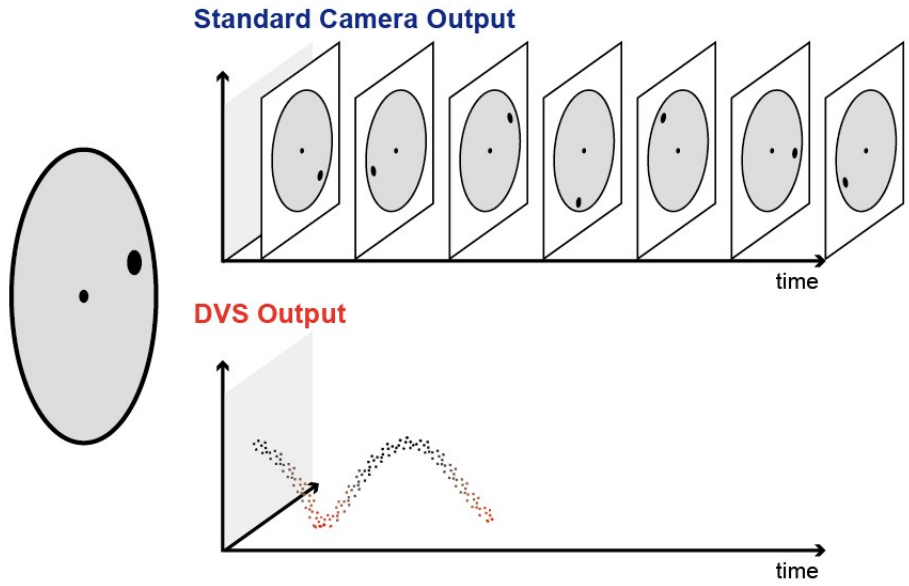


Figure 1.1: Difference between frame-based standard camera output and event-based DVS output in capturing a black dot rotating around the center within a gray circle. The results from the frame-based camera are frames at millisecond intervals. On the other hand, the DVS is capable of rapidly capturing changes in the scene and producing a stream of events at microsecond intervals. Modified from [1].

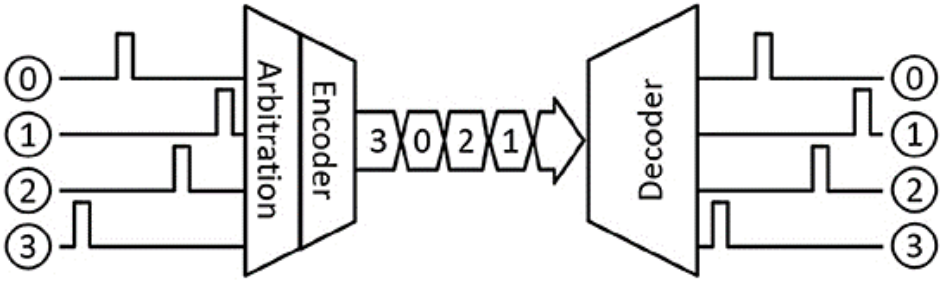


Figure 1.2: The essence of the AER bus. Adapted from [2]

cameras, especially in capturing transient, high-speed phenomena. However, the performance of the DVS heavily relies on the precision and efficiency of its underlying AER interface.

Presently, most DVS implementations rely on a row-scanning synchronous AER [5, 14, 15] primarily because it can transmit a large volume of data in a single burst over extended periods, offering high throughput. However, with synchronous circuits scanning each row in a fixed manner, there's an inherent delay before responding to rapid, sparse changes in individual pixels. The inability of synchronous row-scanning AER to capture intricate temporal dynamics becomes noticeable in applications that require detailed tracking of subtle temporal data changes, such as low-cost velocity monitoring [16] and ultra slow motion video [17].

Asynchronous AER, with its event-driven nature, presents a solution to the temporal resolution challenge by precisely transmitting individual events, offering enhanced time precision. However, this event-by-event transmission approach limits the overall throughput. Therefore, current asynchronous AER implementations find it challenging to handle applications with a dense stream of events. This trade-off between temporal resolution and throughput

significantly hampers the broader adoption of asynchronous AER interfaces in DVS designs.

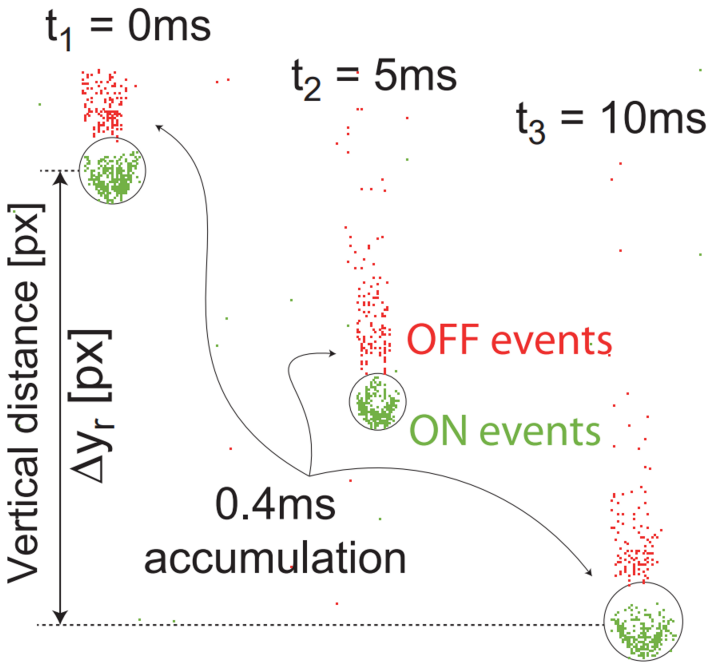


Figure 1.3: Schematic diagram of falling raindrops in the field of view of a DVS, and each cluster is formed by the accumulation of events with a duration of 0.4ms. OFF and ON events are generated by decreases and increases in brightness, respectively. Events typically occur close in time and space, highlighting their inherent spatiotemporal correlations. Modified from [3].

Recognizing the dual need for high temporal precision and robust throughput, there is a compelling demand for a method that bridges this gap. A core principle of our innovative approach to enhancing the throughput of asynchronous AER interfaces lies in exploiting the spatiotemporal correlations between events, which is typical of DVS data as illustrated in Fig. 1.3. By capitalizing on these correlations, we aim to boost throughput without compromising the inherent high temporal precision of asynchronous AER interfaces.

1.2 Contributions

The contributions of this thesis include:

- The design and implementation of a high-speed asynchronous AER interface exploiting the spatiotemporal correlation of events.

- Benchmarks show our design, with a slight compromise in time precision, achieves nearly double the throughput of current asynchronous AERs.

1.3 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 gives a brief introduction to the asynchronous circuit, the used asynchronous circuit toolkit (ACT) flow illustrated with a simple example, and a brief review of key state-of-the-art AER schemes. Chapter 3 provides the design choices as well as implementation details of the proposed design. Then Chapter 4 shows the simulation results of the design. Lastly, in Chapter 5, the conclusion is drawn, and the future work is discussed.

2 Background

This chapter provides a comprehensive background of asynchronous digital circuits, of the ACT flow, and a brief review of key state-of-the-art AER schemes. First, some essential concepts and principles of asynchronous digital circuits are introduced. Then, a simple example is presented to aid the reader's understanding of the selected ACT asynchronous circuit design flow. Lastly, the three state-of-the-art AER schemes are reviewed and their pros and cons are compared to propose our improvement.

2.1 Asynchronous digital circuits

Asynchronous circuits do not use any global clock signal to synchronize their state elements. Instead, the synchronization depends on the arrival of data or of a control signal, which is realized with a handshake protocol. As shown in Fig. 2.1., there are two types of handshake protocol: 4-phase handshake and 2-phase handshake. Taking the most popular 4-phase handshake as an example, when the previous operation is completed, the request (Req) signal is set high, indicating that the next stage is allowed to start its operations for a new transaction. Once the next stage completes its operation, it sets the acknowledge (Ack) signal high. Once detecting a high Ack signal, the previous stage sets the Req signal low. Then if the next stage detects that the Req signal is low, it sets the Ack signal low, finishing the 4-phase handshake. In the end, the Req and Ack signals are set to their initial positions and ready for the next round of handshake. For a 2-phase handshake, there is no resetting of Req and Ack and the handshake is done by alternating between the first half or the second half of the 4-phase handshake. Although it may seem that a 2-phase handshake is simpler and faster, its circuit implementation is usually more complex because there are two possible initial states.

The independence from a clock makes it possible for asynchronous digital circuits to operate faster than synchronous circuits as they do not need to wait for the clock signal to start processing. Therefore, ideally, the speed of asynchronous circuits is only limited by the propagation delays of the circuit elements. Another important characteristic of asynchronous circuits is that the speed depends on the given input case, while for synchronous circuits, the speed of the whole system is dependent on the "worst" case, which defines the maximum clock speed for timing closure. Let us consider a 2-bit carry ripple adder as an example. The clock cycle of the synchronous circuit is limited by the scenario where two carries are rippled (e.g., 2'b11 + 1'b1). On the other hand, for the asynchronous circuit, the processing time scales with the number of carries to propagate. This allows asynchronous circuits to outperform synchronous circuits in situations where the worst-case scenario rarely occurs.

However, the overhead of handshake circuits for inter-module communication in asynchronous circuits may result in poorer performance in terms of power consumption and area, as such handshaking circuits are absent in synchronous circuits. More importantly, asynchronous circuits are more challenging to design, test, and scale compared to synchronous circuits. The primary reason for this is that synchronous digital circuits remain the main-

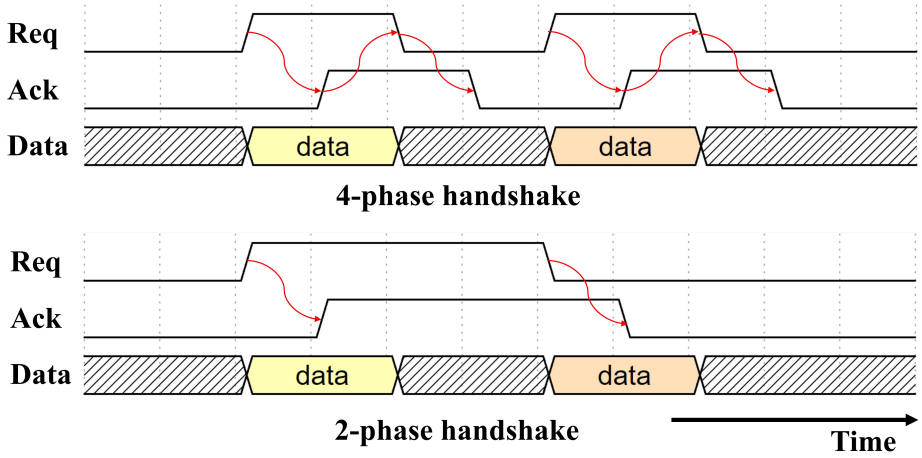


Figure 2.1: Timing diagrams of 4-phase handshake and 2-phase handshake. For the 4-phase handshake, 'Req' and 'Ack' return to their initial states after completing the handshake; for the 2-phase handshake, there are two possible initial states for 'Req' and 'Ack'. This figure is modified from [4].

stream design style; hence, their electronic design automation (EDA) tools are continuously updated and improved. In contrast, the EDA tools for asynchronous circuits have not received the same level of development due to a smaller user base. The ACT asynchronous flow used in this project is a first step toward alleviating this drawback.

2.2 The asynchronous circuit toolkit (ACT) design flow

Just as synchronous circuits have many design styles, asynchronous circuits also have a lot of families. The ACT flow used in this project supports many design styles by using the Communicating Hardware Processes (CHP) language [18] to make the communication between modules abstract, which can be translated into different logic families. This section mainly introduces the circuit styles and flow we adopted, which is well supported by the ACT asynchronous design flow, along with a simple example of a buffer circuit to aid the reader's understanding of the ACT flow.

2.2.1 Style and data encoding methods

The quasi-delay-insensitive (QDI) [19] design style is adopted in this project. In essence, QDI circuits disregard the delay of wires and gates, and they are designed to form a self-locking structure. This self-locking structure ensures that the completion of each stage unlocks the next one, thereby enabling QDI circuits to operate strictly in a specific execution order. To implement QDI circuits, the input, output, and internal intermediate variables are encoded in the following three ways: bundled-data, dual-rail, and 1-of-N. To help reader understand, we show an example of transmitting two-bit data '11' and the corresponding waveform diagrams for each encoding.

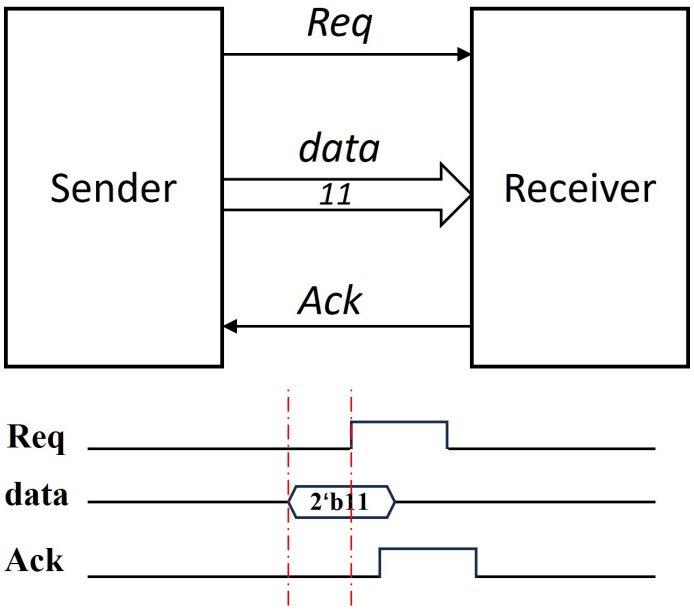


Figure 2.2: Schematic diagram of bundled-data encoding, the example timing diagram for the transmission of '11' data is on the bottom.

2.2.1.1 Bundled-data

Bundled-data is the most efficient way to transmit multi-bit data. As shown in Fig. 2.2, it consists of a Req line, an Ack line, and data lines. There are 4 lines in total, two for the 2-bit data, one for the request, and the last one for the acknowledgment. If N-bit data needs to be transmitted, a total of N+2 lines are required. As shown in the bottom waveform diagram, the data must be ready before the Req is set high. And typical 4-phase handshake is performed between the Req and Ack line. After Ack is set high, it means that the data is received and thus does not need to be valid anymore.

In the bundled-data encoding, to ensure that the data is always available when the sender puts the Req signal high (otherwise the receiver may receive the wrong data), a delay is added to the Req line. This small delay overhead is usually favourable compared to the overhead of the other two encoding schemes.

2.2.1.2 Dual-rail

Dual-rail is an encoding scheme that does not require any delay line. It consists of dual-rail data lines and an Ack signal. In this encoding, each bit of the data is encoded with 2 wires, as shown in Table 2.1.

In dual-rail encoding, 00 indicates no data, 01 represents a data value of 0, 10 represents a data value of 1, and 11 indicates an error. Since 01 and 10 indicate that the data is valid, the data itself can serve as the Req signal. Therefore, as shown in Fig. 2.3, there is no Req line and the 4-phase handshake is performed between the data and the Ack. Here, (d3,d2) and (d1,d0) are used to represent the two transmitted bits, and the Ack is only set high when both bits of the data are ready. In dual-rail encoding, if N-bit data needs to be transmitted, it requires 2N+1 lines, which makes dual-rail encoding less scalable for multi-bit data transmission.

Dual-rail encoding	data
00	No data
01	0
10	1
11	Error

Table 2.1: The 4 possible cases of dual-rail encoding.

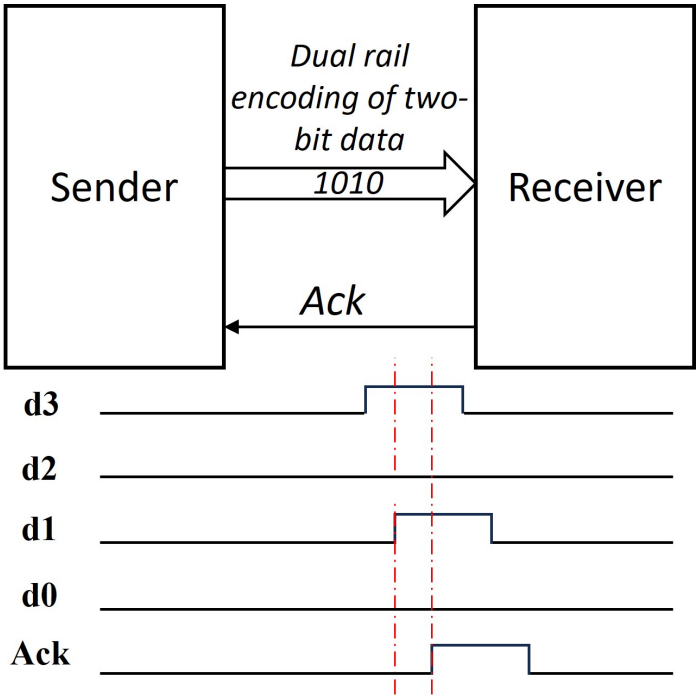


Figure 2.3: Schematic diagram of dual-rail encoding, the example timing diagram for the transmission of '11' data is on the bottom, and uses 5 wires.

2.2.1.3 1-of-N

Like dual-rail, 1-of-N encoding does not require delay lines and represents data with one-hot encoding. The four possible values that the 2-bit transmitted data can have are shown in Table 2.2.

1-of-N encoding	2-bit encoded data
0000	No data
0001	00
0010	01
0100	10
1000	11
Other	Error

Table 2.2: The 4 possible cases of 1-of-N encoding.

The (d3,d2,d1,d0) data wires can also be used as Req by setting any bit high as shown

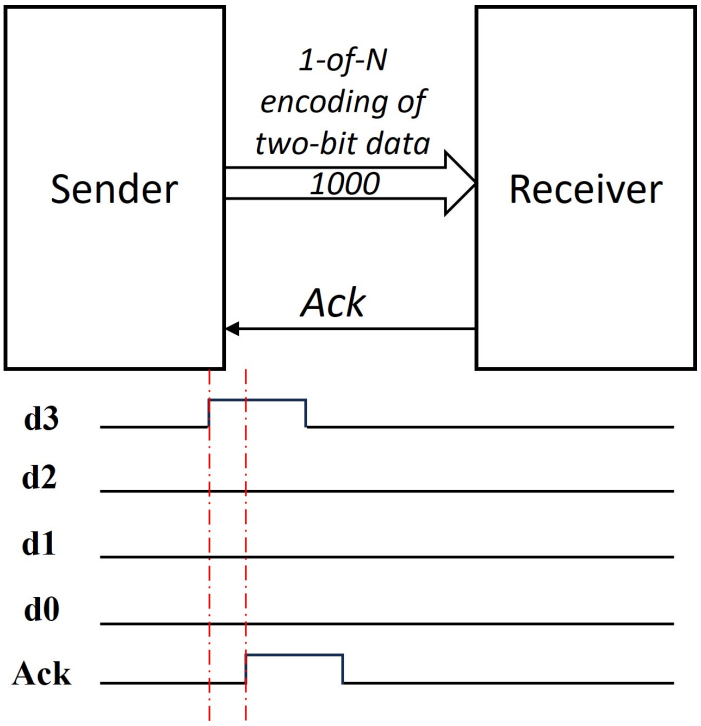


Figure 2.4: Schematic diagram of 1-of-N encoding, the example timing diagram for the transmission of '11' data is on the bottom.

in Fig. 2.4. The Ack is only set high when any of the bits of the data is high to perform the 4-phase handshake. It is costly to transmit data in 1-of-N encoding since it will need $2^N + 1$ wires to encode N-bit data. However, in certain scenarios, 1-of-N encoding is also employed due to its absence of delay lines and, in situations where there are conditional executions based on data values, only one data line needs to be checked for each condition.

2.2.2 Asynchronous circuits design with ACT

The ACT flow supports the design of asynchronous digital circuits from the abstract level to physical design. The design flow chart is shown in Fig. 2.5. This section will take a simple buffer as an example to introduce the ACT design flow. The following sub-sections follow each step of the flow, as outlined in Fig. 2.5.

2.2.2.1 High-level communication

The ACT flow works in a modular design style and starts by describing the circuit functionality using the communicating hardware processes (CHP) language [18]. CHP mainly describes high-level asynchronous communication between concurrently operating hardware modules and their functions. Blocks that operate on their own are connected to each other via so-called "channels". They send data via these channels to exchange information and synchronize with other blocks via handshake. At this stage, neither the encoding scheme of the internal variable nor the handshaking scheme of the channels is determined, but only the highly abstract process of communication between blocks is designed. Similar to Verilog, CHP

Idea

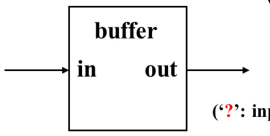
1 High-level communication

2 Handshaking expansion (HSE)

Example:

I want to build a buffer. It immediately outputs when there is an input.

①. Describe the communication with CHP



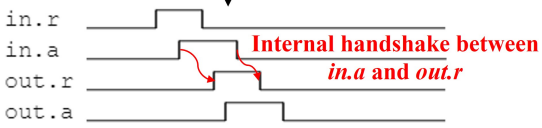
('?': input, '!':output, ';': sequential execution)

High-level asynchronous communication between blocks

②. Choose the bundled-data encoding for *in* and *out*

In is encoded with (*in.r, in.a*) for the request and acknowledge wires
Out is encoded with (*out.r, out.a*) for the request and acknowledge wires

③. Arrange the execution sequence



④. Determine the order of execution

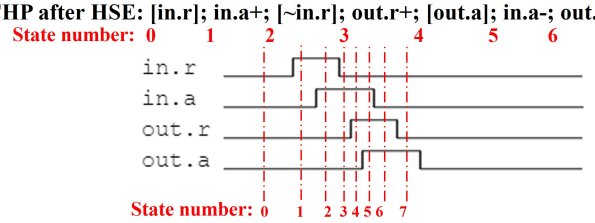
CHP	CHP after HSE:
in?; out!	[in.r]; in.a+; [~in.r]; out.r+; [out.a]; in.a-; out.r-; [~out.a]

('[]': wait until true, '~': low-level, '+': pulled up, '-': pulled down)

Middle form between abstract code and implementable circuits

⑤. Infer the circuit states: (*in.r, in.a, out.r, out.a*), 'X' for unknown

CHP after HSE: [in.r]; in.a+; [~in.r]; out.r+; [out.a]; in.a-; out.r-; [~out.a]



States vector: (*in.r, in.a, out.r, out.a*)

	0	1	2	3	4	5	6	7
HSE	[~out.a]	[in.r]	in.a+	[~in.r]	out.r+	[out.a]	in.a-	out.r-
state	X000	1000	X100	0100	011X	0111	X011	X00X

Here *in.r* and *out.a* only have a definite value after "[]" because they are external variables controlled by other blocks. For robustness, we only assume the values of external variables when necessary.

⑥. Infer the expression of internal variables from circuit state:

in.r & ~out.r & ~out.a→ in.a+
~in.r & in.a → out.r+

out.r & out.a → in.a-
~in.a & out.a → out.r-

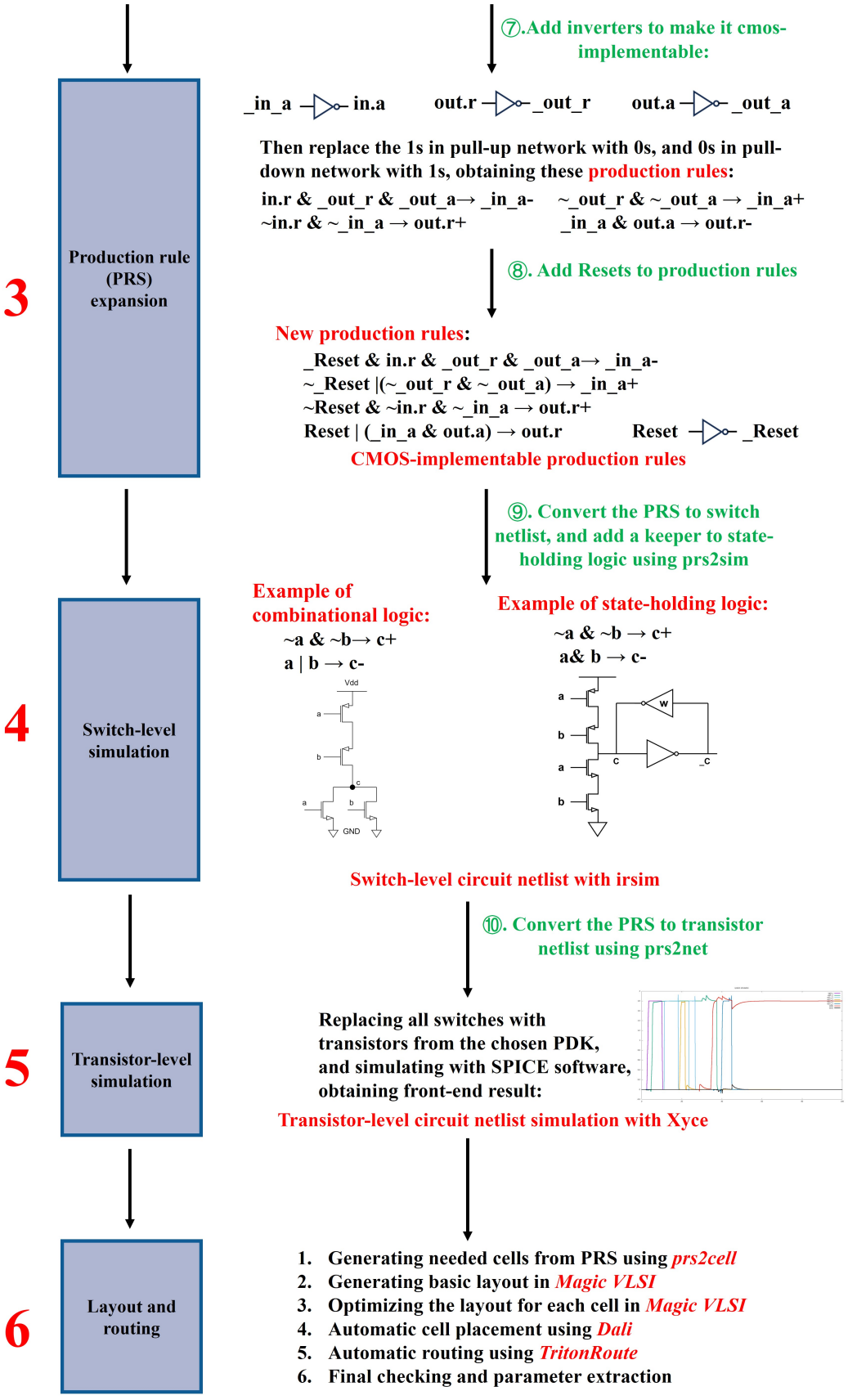


Figure 2.5: Steps of the ACT flow, examples are provided on the right.

supports operations such as serial operations, parallel operations, mathematical calculation, and conditional execution to implement more complex functions. The designed block can be simulated and verified with software called ***actsim*** in ACT. Unlike Verilog, any CHP code that can be simulated by actsim implies that it can be synthesized. During the simulation, actsim will synthesize the CHP code in a structured manner to verify the feasibility of the CHP. However, the code synthesized automatically by actsim usually requires several tens of times more transistors compared to the synthesis method we will use in the *handshaking expansion* (section 2.2.2.2) and *production rules expansion stage* (section 2.2.2.3), hence it should not be used directly.

Example: On the right side of the *high-level communication step* in Figure 2.5 is a simple buffer example. The CHP code **in?; out!** describes the function of the buffer: it inputs data from the port **in** and then immediately outputs from the port **out**. It can be noticed that there is no further specification on the input and output channels. Users can verify the function of the code with actsim.

2.2.2.2 Handshaking expansion

Handshaking expansion (HSE) is an intermediate step in the manual synthesis method proposed by Alain J. Martin in [20]. Unlike the structural synthesis method by actsim mentioned in the previous step, this method ensures the synthesis of a more concise circuit and often leaves room for optimization. HSE transforms the behavioral description in CHP into an intermediate state between behavior and structure, enabling further synthesis into a circuit. It is performed with the original CHP code in high-level communication. While getting HSE from CHP is largely based on experience, it generally involves the following steps:

1. Determine the type of handshake of the channel and the data encoding scheme.

2. Replace the communication on the channel with the selected handshaking scheme, and replace the data with the selected encoding scheme.

3. Rearrange all channel variables and data variables such that all signals have handshakes with other signals, to ensure they can be executed in sequence correctly.

4. Check the obtained HSE by using the real or assumed HSE of other blocks as input and output conditions.

The obtained code after HSE is still based on the CHP language, but with all variables being Boolean-valued and all abstract communication actions being defined. This is necessary for the next step where we will infer their conditions to execute.

Example: As shown in the example on the right side of Fig 2.5, the original abstract communication process and data operations are replaced by the actual operations on variables of the selected encoding. Here, the input channel is implemented with zero-bit bundled-data so there are two variables: the request (**in.r**) and the acknowledge (**in.a**), and similarly for the output channel. All **in.r**, **in.a**, **out.r**, and **out.a** are Boolean-valued. Also, their execution sequence is arranged such that all signals have handshakes with other signals. For example, the variable "**in.a**" has handshakes with "**in.r**" and "**out.r**", which also has a handshake with "**out.a**". This sequential handshake between variables ensures the sequential execution of the QDI circuit.

2.2.2.3 Production rule expansion

After obtaining the correct CHP through handshaking expansion, the next step is to generate a production rule set (PRS) that directly corresponds to complementary metal-oxide-semiconductor (CMOS) circuits. As mentioned above, QDI circuits can ensure a strict execution order, which is realized through conditional execution of variables; that is, the completion of one level's variable acts as a prerequisite condition for the execution of the next level's variable. At this stage, the circuit states (i.e. the values of all variables) are used to determine the execution conditions of the variables. Due to the characteristic inverted logic in CMOS technology, production rules like x $\rightarrow$ y+ (y is pulled up when x is high) are not acceptable, as a PMOS would be turned on with a high level. If the generated production rule is not CMOS-implementable, extra effort, such as changing the variables, must be done. Sometimes CHP cannot be converted into implementable PRS when there are too many identical states, which is usually caused by improperly designed HSE but is hard to observe in the *handshaking expansion* stage [21]. In such a case we need to correct the HSE and go over the second, third, and fourth steps in Fig. 2.5 again. It is also important to consider the initialization of the circuit, for which we need to carefully design the Reset phase of each module.

Example: Following the order indicated in Fig 2.5, we label the states of the circuit based on the vectors (**in.r, in.a, out.r, out.a**) for the CHP after HSE, as shown in Fig. 2.6 where any undetermined external signal is represented by X.

[in.r]; in.a+; [~in.r]; out.r+; [out.a]; in.a-; out.r-; [~out.a]

0 1 2 3 4 5 6 0

State	in.r	in.a	out.r	out.a
0	X	0	0	0
1	1	0	0	0
2	X	1	0	0
3	0	1	0	0
4	0	1	1	X
5	0	1	1	1
6	X	0	1	1
0	X	0	0	X

Figure 2.6: The state of each variable in the HSE code of the buffer example during each execution step. The 'X' in the table represents a don't care state, which cannot be used to infer the final expression and is typically linked to the external inputs of the modules.

The reader might wonder why '**in.r**' and '**out.a**' in the state diagram of Figure 2.6 are

replaced by 'X' even though they clearly have certain values. This is because these variables are external variables controlled by other blocks. If we can infer the execution conditions of other variables without using them (i.e. without assuming an external environment), then the robustness of this block will be better. That way, an error in an external block does not propagate. However, the values of external variables appearing after the waiting command '[]' should be determined and used. By examining these state vectors, we can infer the execution conditions for each step. For example, from the second to the third state when $out.r$ transitions from 0 to 1, we observe that $in.r$ is 0 and $in.a$ is 1. Because among these state vectors only the third step satisfies this condition, we can determine the condition for $out.r+$ is $(\tilde{}\ in.r)\&in.a$. Here, the tilde symbol "~" denotes a low voltage level of the variable, while the absence of the tilde indicates a high voltage level. Similarly, we can write out the conditions for other transitions. However, when more than one state satisfies the inferred condition, we need to introduce new variables or modify the HSE. All the PRS for this buffer example can be expressed as:

```
in.r & ~out.r & ~out.a→ in.a+
out.r & out.a → in.a-
~in.r & in.a → out.r+
~in.a & out.a → out.r-,
```

Finally, to ensure that the obtained PRS is CMOS-implementable, we need to set all variables in the pull-up networks to a low voltage level to drive PMOS transistors, and set all variables in the pull-down networks to a high voltage level, to drive NMOS transistors. To do this, we need to invert some variables. For example, in $(\tilde{}in.r)\&in.a \rightarrow out.r+$, we have $in.a$ in its high voltage level form in the pull-up network of $out.r$, and it is therefore not CMOS-implementable. So we use an inverter and create the inverted form of $in.a$, which is $\_in\_a$, and in this way the original PRS becomes $(\tilde{}in.r)\&(\tilde{}\_in\_a) \rightarrow out.r+$. By using the new variable $\_in\_a$, we can make the PRS CMOS-implementable. For detailed constraints associated with using inverted variables, which are outside the scope of this thesis, we refer the reader to [22]. The properly inverted PRS of the buffer example is:

```
in.r & _out_r & _out_a→ _in_a-
~_out_r & ~_out_a → _in_a+
~in.r & ~_in_a → out.r+
_in_a & out.a → out.r-
```

where _in_a is the inverted form of in.a, _out_r is the inverted form of out.r, _out_a is the inverted form of out.a.

The last step is to add a reset mechanism and perform optimization to the PRS. The **Reset** signal is a global signal shared by all modules of the asynchronous circuit. Normally, we can simply add **Reset** to the left side of the PRS:

```
_Reset & in.r & _out_r & _out_a→ _in_a-
~_Reset |(~_out_r & ~_out_a) → _in_a+
~Reset & ~in.r & ~_in_a → out.r+
Reset | (_in_a & out.a) → out.r-,
```

Where **_Reset** is the inverted form of **Reset**. In this way, when the **Reset** is pulled up in the reset phase, every variable in the circuit is reset.

2.2.2.4 Switch-level simulation

After obtaining a PRS that is CMOS-implementable, the ***prs2sim*** software from the ACT flow can be used to directly convert the PRS into a transistor-based implementation. The process design lit (PDK) is not yet determined at this stage, so the switch model is used instead. The switch-level model simply considers PMOS transistors as turned on by a low voltage and NMOS transistors as turned on by a high voltage, independently of the technology. They can be approximated as switches, which brings a considerable improvement in simulation speed.

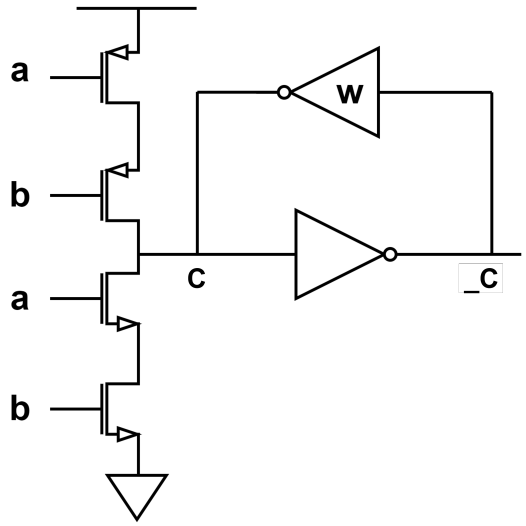


Figure 2.7: A typical state-holding gate. Its PRS expressions are $\tilde{}a\& \tilde{} b \rightarrow c+$ and $a\&b \rightarrow c-$, and due to the asymmetry of pull-up network and pull-down network, a weak keeper is required at the output to hold the previous value. The output _c is the inverted form of c.

During this step, the software will also determine whether the generated circuit belongs to combinational logic or state-holding logic based on the input PRS. State-holding logic is characterized by non-complementary pull-up and pull-down networks. For example, a pull-up network $\tilde{}a\& \tilde{}b$, and its pull-down network $a\&b$ make a gate state-holding instead of combinational. In this example, the generated circuit consists of two PMOS and two NMOS transistors in series, as shown in Fig. 2.7. When the inputs a and b are such that one is 0 and the other is 1, neither the pull-up network nor the pull-down network is activated, resulting in a floating output. Usually, we want to maintain the output at its last activated state, and therefore an additional state keeper is needed and connected to the output, which results in extra circuit overhead. Therefore, when designing circuits, it is essential to avoid using state-holding logic where it is not strictly necessary.

Example: The ***prs2sim*** in the ACT flow automatically generates the corresponding switch-level netlist for the PRS with combinational logic or state-holding logic. In the example shown in Fig 2.5, all PRS expressions translate to state-holding logic and the tool will automatically generate keepers for them. Afterwards, we can simulate the asynchronous digital circuit using the obtained switch-level netlist, which is similar to simulating synchronous digital circuits.

2.2.2.5 Transistor-level simulation

After verifying the circuit in switch-level simulation, we then replace the obtained switch models with Spice models from the PDK and perform transistor-level simulation using Spice simulation software, as for analog circuit simulations. This step can be accomplished using the ***prs2net*** tool from the ACT flow. By comparing the transistor-level waveforms with the switch-level waveforms under different inputs, we can identify issues related to sizing, arbiters, and other hidden problems that have not been discovered in earlier steps, and better characterize the performance of the design. For the open-source Spice simulation software Xyce [23], simulating thousands of transistors requires a large amount of memory and is prone to convergence issues. Therefore, the simulation and verification of large-scale asynchronous circuits is still a major scalability bottleneck for asynchronous circuit design.

2.2.2.6 Layout and routing

The placement and routing in the ACT flow primarily consist of the following steps as mentioned in Fig 2.5 (the open-source Skywater 130nm process is assumed [24])

1. Generating required cells from PRS using prs2cell: The ***prs2cell*** software in the ACT flow can divide the PRS of a module into different cell implementations, where a cell can be assimilated to a logic gate. A cell contains a different number of inputs and one output and may include the required weak keeper. These cells are typically reused and encompass all components for implementing an asynchronous circuit. Hence, for the Skywater130 PDK, numerous standard cells have been provided. However, due to the variability and asymmetry of PRS, most cells typically still require custom design.

2. Generating basic layout for each cell in ***Magic VLSI***: The ACT flow includes the conversion of the generated cells into scripts for the widely used open-source layout tool, Magic VLSI [25]. By running this script in Magic VLSI, basic layers including N, P diffusion, polysilicon, and labels representing connection relationships are generated automatically. This step generates a first draft for each cell.

3. Optimizing the layout for each cell in ***Magic VLSI***: The automatically generated layout is often not optimally placed and requires intra-cell connections. Manual connections need to be done with usually three layers of polysilicon (poly), metal1 (m1), and metal2 (m2) and follow a basic rule of preferred vertical or horizontal wiring directions.

4. Automatic cell placement using ***Dali***: Unlike the cells with standardized heights in synchronous circuits, cells in the ACT flow often have non-identical heights. To address the placement challenges of cells with non-identical heights, Dali was developed [26]. After completing all cell layouts, a script in the ACT flow first converts the layout of cells in Magic VLSI into a .rect file that can be read by Dali. Then, we can use Dali in the ACT flow for placement. Dali abstracts all cells into rectangles of a certain size and attempts to place them according to the density provided by the designer, and then generates .lef and .def files.

5. Automatic routing using ***TritonRoute***: TritonRoute [27] is a routing tool designed with compatibility with Dali. It can interpret the .lef and .def files provided by Dali and uses metal3 (m3) and metal4 (m4) for automatic routing, thereby generating new .lef and .def files. m3 and m4 also follow the basic rule of preferred vertical or horizontal routing directions.

6. Final checking: Importing the routed .lef and .def files in Magic VLSI results in the post-placement and routing layout. However, at this stage and due to a lack of maturity, there are typically some design rule check (DRC) errors, usually caused by metal layer distance. After

manually correcting these DRC errors, to verify if the layout design matches the functionality of the previous PRS, we can use the extract command in Magic VLSI and the ***ext2sim*** software in the ACT flow to convert the layout back into a switch-level and a transistor-level netlist, so as to perform switch-level and transistor-level simulation using irsim and Xyce. This step can be viewed as an equivalent of the layout versus schematic (LVS) in the synchronous digital circuit design flow.

2.3 Review of key state-of-the-art address event representation (AER) schemes

In this section, we will introduce the three prevalent AER schemes: synchronous row scanning, the binary arbiter tree, and the token ring. The pros and cons of each scheme will also be discussed at the end of this section.

2.3.1 Synchronous row scanning scheme

To accommodate the increase in resolution, many commercial DVS [5, 15] employ a synchronous row scanning scheme. The synchronous row scanning scheme (or column scan scheme, depending on the definition) operates by sequentially scanning all rows and sending all events in the scanned row at once. As illustrated in Fig. 2.8, the scan is controlled by the control logic at the top. It operates the row (column) driver to sequentially inspect each scanned row (column) for requests. If there is no request, it will skip this row (column). If any request is detected, the entire row's (column's) event is copied as a one-dimensional vector and sent out. This method of encoding multiple events in a certain order and sending them in one packet is also called group-encoding. Group-encoding and sending these events imply an overhead of a few clock cycles. After sending the vector, the row (column) driver will continue to scan the next row (column). The synchronous row scanning AER interface in [5] directly copies the entire row and transmits events in all 960 pixels in the row, while the one in [15] copies the entire row of 1280 pixels. With synchronous row scanning AER, they achieve the throughput with the maximum event rates of 1300 Meps (Mega events per second) and 1066 Meps, respectively. Such throughputs are notably higher than those seen in earlier AER designs. A primary advantage of these synchronous designs is the maturity and scalability of synchronous digital circuit design, making it well-suited for Very Large Scale Integration (VLSI). However, there are inherent drawbacks to the synchronous row scanning scheme:

1). Sequential scanning, instead of event-based operation, introduces timing inaccuracy caused by the event-handling order. For example, at low event rates, this scheme scans in a fixed sequence, regardless of the presence or absence of events requiring scanning. In particular, such low event rates (¡ 10 Meps) are the typical operating condition for DVS, occurring over 70% of the time according to [28]. However, rows scanned recently have to wait a full cycle until they are scanned again. The time required to scan one row without skipping in [5] is approximately 1.47 μs. For the scan length of hundreds of rows, the μs-range temporal resolution of DVS will be lost. Moreover, this always-on scanning by the AER does not match the event-driven power consumption of the DVS.

2). Low efficiency in row/column-based group-encoding. The maximal event rate scenario

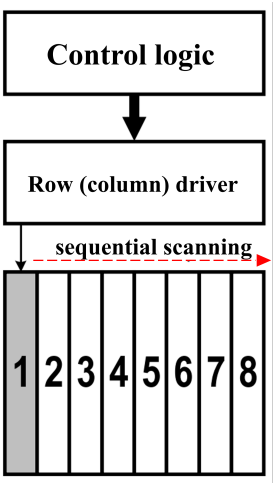


Figure 2.8: Schematic diagram of the row (column) scanning scheme, modified from [5]. It uses control logic to operate the column (row) driver, which will sequentially scan each column.

of the synchronous row-scanning scheme is unrealistic, as it would imply that every pixel in a row (or column) gets activated every time a row is scanned. In reality, events are more likely to appear in spatiotemporal clusters rather than aligning in rows or columns. Even worse, if every pixel of a column requests, using a row-based scan to transmit these events does not allow for any row skipping.

2.3.2 Asynchronous Binary arbiter tree scheme

In 2000, Boahen for the first time proposed the concept of AER for multi-point interconnection in large-scale neuromorphic chips [6]. His design follows an asynchronous 2-D binary arbiter tree scheme, which performs a hierarchical row-first column-second arbitration to select one of the simultaneous events to output. The 2-D arbiter tree circuit has been widely popular ever since. It works as follows:

1. Select one of the requesting rows using a row-wise binary arbiter tree.

2. Then select one of the requesting pixels in the selected row using a column-wise binary arbiter tree.

The binary arbiter trees of rows and columns are formed by 2-to-1 arbiter cells, as shown in Fig. 2.9. The working principle of a 2-to-1 arbiter cell is that when it has a request from only one input, it will transmit that request. When it has requests from both inputs, it will randomly select and transmit one input's request, leaving the other input's request to wait. The circuit details of the arbiter cells are out of the scope of this thesis and can be found in [6]. In the binary arbiter tree, the handshake logic of each row and column is connected to an input of an arbiter cell, and the outputs of the arbiter cells become the input for the next layer of arbiter cells until there is only one single output, thus N rows/columns are connected to a $log_2 N$-level binary arbiter tree. Each connection includes two wires (Req and Ack) to realize a handshake protocol. When two rows/columns request simultaneously, their requests will traverse the layer of arbiter cells until there is an arbiter cell for which both inputs have requests. This arbiter cell will then randomly select one of the inputs and propagate it while having the other input waiting. Subsequently, the acknowledgment is propagated

back along the same path as the request to the selected requesting row/column, completing the handshake protocol. This type of AER circuit is called "greedy", which means that the arbitration is completely random because of the use of fair arbiters. In the greedy binary arbiter tree, if the rows in a column that just has been selected continue to trigger requests, this column may be immediately selected again, and the other columns have to wait until they are randomly selected. This behavior might not be desirable because it will cause timing distortion in waiting columns.

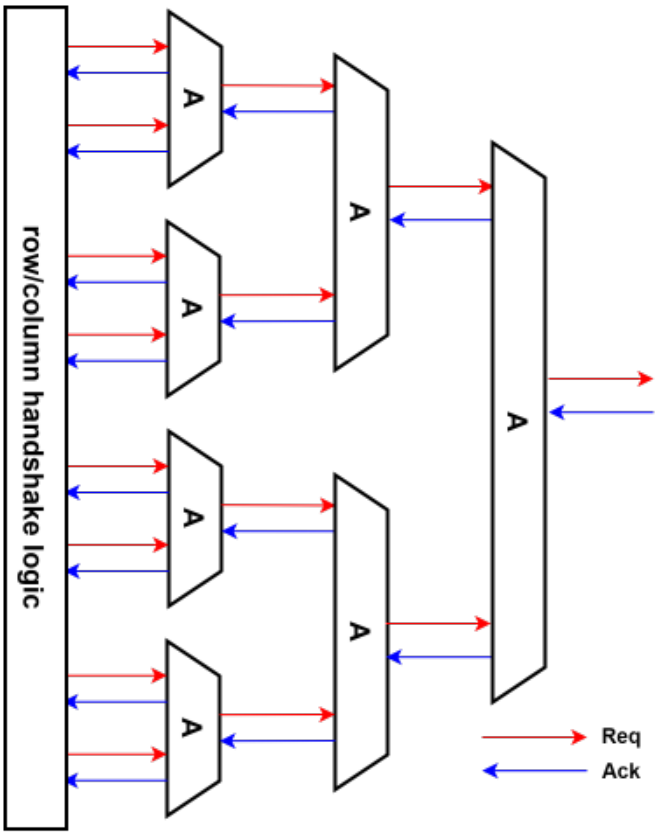


Figure 2.9: Schematic diagram of the arbiter tree structure. Each arbiter cell has two inputs and one output channel, and each channel consists of a Req and an Ack wire [6].

In 2008, Lichsteiner et al. proposed modifications to this scheme in their popular DVS [11], where the "greedy" behavior is changed to a "non-greedy" one. This modification ensures that a serviced row will not be serviced again until all other registered requests have been served. The "non-greedy" scheme significantly decreases the potential timing distortion caused by random selection as there will be no rows/columns that need to wait infinitely if they are not lucky enough to be randomly selected.

The binary arbiter tree offers a key advantage in providing optimal performance when sending single or sparse events. For example, according to [8], it takes the binary arbiter tree scheme 2.8 ns to process a single event among 256 inputs, which is much lower than the typical cycle time of the row scanning scheme. However, the binary arbiter tree also has evident drawbacks:

1). Due to the randomness of the arbiter's selection, events generated in close regions may be sent separately due to random selection, so the spatiotemporal correlation between events is not exploited. For two events generated in adjacent rows/columns, the worst case for non-greedy binary arbiter trees is one being selected first and the other being processed after all other requests have been processed; while greedy trees offer no bound.

2.3.3 Asynchronous token-ring scheme

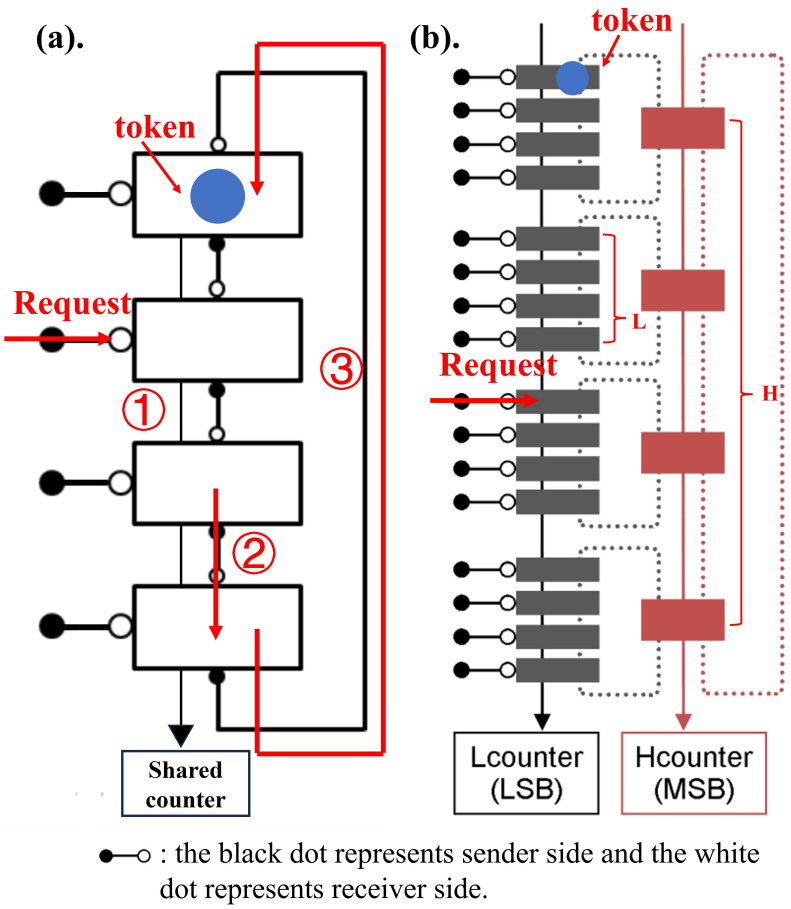


Figure 2.10: (a). Schematic diagram of the token-ring structure, modified from [7]. It uses a counter to record the location of the token to determine the sending address. (b). Schematic diagram of the hierarchical token rings (HTR) structure, adapted from [8]. It adds an additional layer of Hserver (red blocks) to bypass the token movement of the first-layer server (Lserver), increasing the efficiency of token movement in long-distance moving.

In 2011, a new AER scheme was proposed in [7]. It replaces the binary arbiter tree with a token-ring mutual exclusion architecture. This architecture mainly consists of three elements, namely the server, the shared counter, and the token. Servers are used to communicate with each other and to pass the token, the shared counter is used to track the current position of the token, and the token is used to grant a server permission to access the output bus. When a server receives a request, instead of sending the request through an arbiter tree, it requests a token from its neighboring server. If its neighboring server has the token, it will pass the

token to the neighboring requesting server and increase the shared counter by one, otherwise, it will request further neighboring servers. The request is sent in one direction, and therefore the data in the shared counter (i.e. the number of times the token has been moved) can be used to represent the position of the token. If it receives a token, the request is processed. For example, as shown in Fig. 2.10(a), when the second input requests, its server will first request the token from the server below it. Secondly, the requested server will again request the token from the server below it. And thirdly, the last server will request the token from the first server, which processes the token. There are three requests in total, the token will thus be moved three times and the counter will increase by three, thereby recording the current position of the token.

In 2021, a new asynchronous AER arbitration scheme called hierarchical token rings (HTR) [8] was proposed as an optimization over the original token-ring scheme. This structure builds upon the initial token-ring scheme of [7] with an additional bypass layer to accelerate the movement of the token within the circuit, as shown in Fig. 2.10(b). In the HTR scheme, the token can be transferred through both the first layer server (Lserver) and the high-level server (Hserver). To track the token's position, two counters are employed: an Lcounter for the first layer server and an Hcounter for the high-level server. The HTR addresses the inherent latency challenge in the original token-ring during sparse event handling through the incorporation of a bypass layer. For N rows (columns), assuming the initial token position is at the first row (column), and an event occurs in the middle of these rows (columns), the HTR would require only $(H + L)/2$ moves [8], where N = H $\times$ L, with H representing the total number of Hserver and L representing the number of Lservers in each Hserver. This improvement comes at the cost of an additional circuit overhead from the Hserver and the fact that the token can no longer freely be moved between adjacent servers because it might have to be first moved through an Hserver.

The general advantage of the token-ring architecture is that requests near the token can be processed quickly as the token moves in the token ring. For processing N requests from N rows (column), the binary tree scheme needs 2N $\times$ ($\log_2 N$ - 1) operations, while the token-ring scheme requires only N. The advantage of the token-ring scheme becomes increasingly pronounced as N grows. However, its disadvantages include:

1). For individual or sparse events, the distance by which the token needs to move may be long. For example, if the token is in the first server and there is a request in the N-th server, it will need to move N times for the traditional token ring, and H+L times for HTR to reach the requesting server, resulting in a higher delay than the binary tree.

2). Access to a shared counter is an inherent issue for the token-ring scheme. Because each time a token is moved, the server will request the shared counter to record the token movement. Since all servers need to be connected to the shared counter, when the number of servers increases, accessing the counter necessitates a more complex structure. Consequently, this can slow down the system and potentially become a bottleneck [8].

2.3.4 Summary and discussion

The advantages and disadvantages of the three schemes are summarized in the Table 2.3: Currently, most high-resolution DVS systems employ the synchronous row scanning scheme. This preference stems from its potential to offer substantial theoretical throughput. However, a notable limitation of this scheme is its fixed, always-on scanning behavior. In contrast, asynchronous schemes are inherently event-driven, their execution sequences adapt in real

Table 2.3: Advantages and disadvantages of the three introduced schemes.

Schemes	Advantage	Disadvantage
Synchronous row scanning	1. Mature and easily scalable 2. Theoretically highest throughput	1. Timing inaccuracy caused by event-handling sequence 2. Inefficient row/column-based group encoding 3. Power wastage due to always-on scanning at low event rates
Token-ring	Fastest for dense event streams	1. Slow for sparse event streams due to long-distance token movement 2. Slower access to the shared counter as the number of servers increases
Binary arbiter tree	Fastest for sparse event streams	1. Lower throughput as event streams become denser 2. Spatiotemporal correlation not exploited due to random selection

time based on event requests, thereby resolving the issues associated with fixed sequences. Furthermore, studies suggest that properly designed asynchronous circuits can outperform their synchronous counterparts in terms of speed [29, 30]. Yet, for binary arbiter tree and token-ring schemes, each arbitration or selection only transmits one event, which significantly constrains their throughput.

It is apparent from Table 2.3 that combining (i) an asynchronous HTR scheme with (ii) a group encoding scheme based on spatiotemporal correlations of input events would allow combining the strength of the various schemes without their drawbacks. To our knowledge, this scheme has not been introduced yet, which we will further elaborate upon in Chapter 3.

Address-fused Hierarchical Token Ring (HTR) design

3

In this chapter, we first provided the design considerations and choices made based on the previous discussion. Then, an overview of the proposed address fused HTR structure is presented. Next, we briefly introduced each module in the address-fused HTR: address fuser, Lserver, Hserver, request merger and data merger, Counter, and lastly, the output buffer. Finally, we discussed the method we adapted in parameterizing the design.

3.1 Design considerations

We have considered the following requirements:

- The design should have an improved throughput to accommodate the high event rate (>100 Meps as suggested by [31]) generated by high-resolution DVS.
- The design should not introduce significant timing inaccuracy to maintain the μs-range temporal resolution in conventional DVS.
- The architecture of the design should exploit the spatiotemporal correlation between events.
- The design must be scalable.

Considering various factors, we believe that among the introduced AER schemes, HTR is the most fitting choice for integrating with the group-encoding that leverages spatiotemporal correlations. While the binary arbiter tree is simple in structure and easy to scale, its random arbitration makes it unsuitable for leveraging spatial correlation. The traditional token-ring architecture, on the other hand, is efficient for transmitting multiple events but is considerably slower for sparse events due to the extended token movement. This slowdown does not align with our objective of minimizing temporal distortion. HTR strikes a balance by combining the strengths of both, introducing an additional bypass layer to the token-ring for quicker token movement across longer distances, ensuring optimal performance for both sparse and dense event streams. Consequently, our approach involves refining the HTR scheme, leveraging spatiotemporal correlations, and adapting HTR components accordingly.

3.2 Architecture overview

To exploit spatiotemporal correlations, we propose an address fuser that collects all events within a certain spatiotemporal window, encodes them as a group, and then sends them into the modified HTR scheme. The address fuser is connected to a region encompassing multiple pixels, known as the sensing field, and any pixel within this field can trigger the

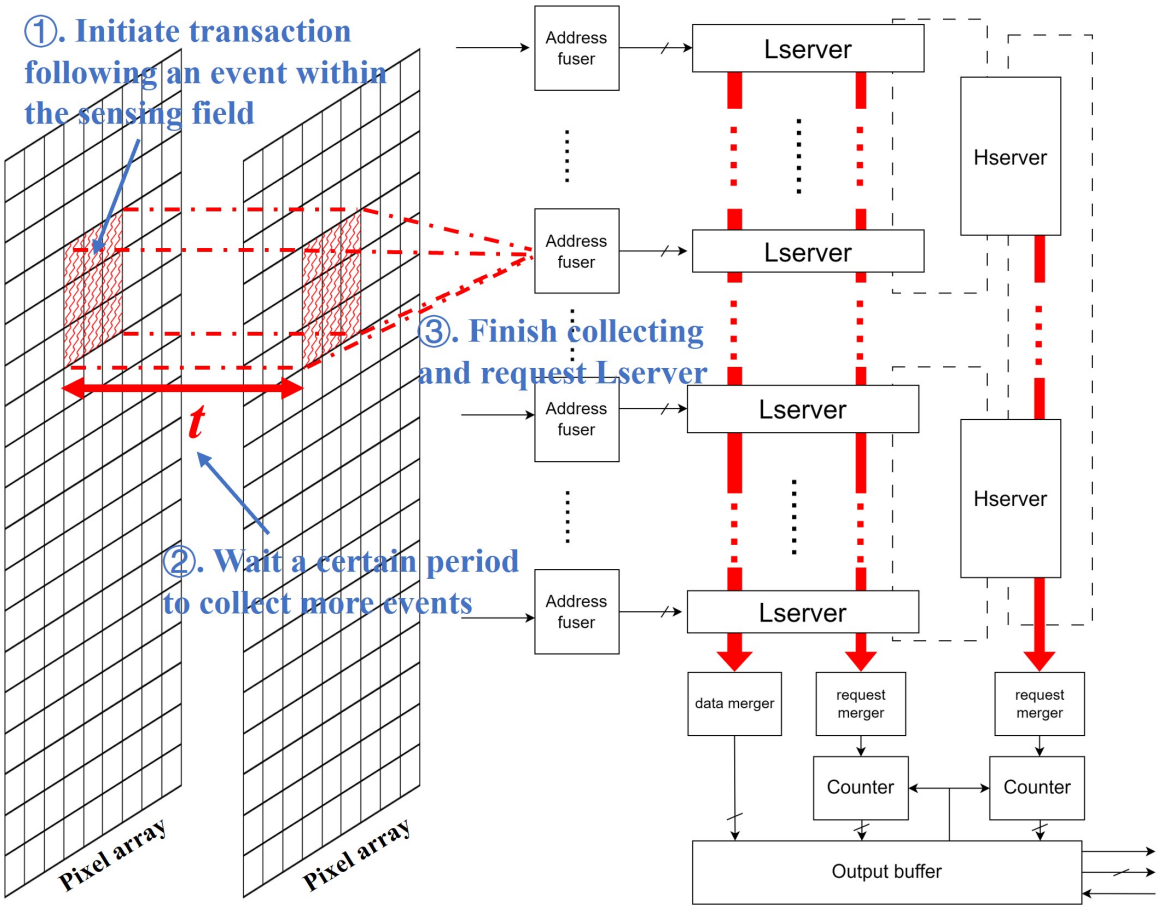


Figure 3.1: Schematic diagram of the proposed address-fused HTR.

address fuser. Fig 3.1 illustrates the overall structure of the proposed asynchronous address-fused HTR. The proposed design consists of several modules, namely the **Address fuser**, the **Low-level server (Lserver)**, the **High-level server (Hserver)**, the **Counter**, the **data merger**, the **request merger**, and the **Output buffer**. When an event is emitted in its sensing field, as shown in step 1 in Fig. 3.1, the address fuser will store the event and wait for a certain time **t**. During this time, it is able to collect more events from the sensing field, as shown in step 2 in Fig. 3.1. This principle is the core idea of the proposed "group-encoding" principle. After the delay, it will send a request to the modified HTR along with the sampled data, as shown in step 3. All ports in this design use bundled-data encoding because of its better scalability efficiency in sending multiple bits (see section 2.2.1).

Fig. 3.2 provides an example that briefly describes the steps of sending events at once. In this figure, each step illustrates the operation related to token movement with blue lines and arrows, while operations related to pixel data are indicated by green lines and arrows. The token in this design has the same function as the token in the token-ring introduced in section 2.3.2. There is only one token and only the Lserver holding the token can access the Output buffer. During the Reset phase, one Lserver is reset to possess the token, while the other Lservers or Hservers do not. For a clear description, only the address fusers and Lservers involved in this event transmission, and the first and last Hserver are shown in Fig. 3.2. The

four shown address fusers and Lservers are labeled with numbers from [0] to [3], and the two shown Hservers are labeled with numbers [0] and [1]. In the Reset phase, the token is initialized to Lserver[0]. The other steps are as follows:

1. When any pixel sends a request, it triggers the circuitry within the address fuser that corresponds to the region to which the pixel belongs. From this first request, the address fuser will accept other requests during a certain period of time. Once this period is finished, the address fuser encodes the collected requests using bundled-data. The address fuser then requests the Lserver with the data attached (green triangle in Fig. 3.2). In this example, the Address fuser[2] requests the Lserver[2].

2. Then, Lserver[2] receives the pixel data from Address fuser[2] and sends a request to Lserver[3] to request the token.

3. Since Lserver[3] does not possess the token, it will again request the token from its neighbouring server, which is Hserver[1].

4. Because Hserver[1] or any of its Lservers still does not possess the token, it will request the token from its neighbouring Hserver, this process continue until Hserver[0], which has the token in one of its Lservers, is requested.

5. The Hserver with the token in one of its Lserver is requested, and then it requests the token from the Lserver that possesses the token.

6. The token is moved in the opposite direction from which it was requested. To track the token's location, each movement of the token in the Lserver or Hserver causes an increment in the corresponding counter. The increment is done by the request sent from the Lserver or Hserver before moving the token. For example, Lserver[0] will request the shared counter and then move the token to Hserver[0].

7. The token is moved to Lserver[2] that requested the token in the first place. Now Lserver[2] is granted access to the output bus. The pixel data in Lserver[2] received from address fuser[2] will then be sent to the output buffer via the data merger.

8. After the output buffer receives pixel data from Lserver[2] via the data merger, it will request the shared counters for their current values to obtain the address of the Lserver possessing the token. Now the final data including the pixel data and address of the address fuser, which is represented by the number of times the token has moved, is in the output buffer and ready to be sent with a 4-phase handshake protocol.

With the working principle above and supported by the ACT flow, the proposed design can overcome the limited throughput and scalability issues for which asynchronous circuits have not yet been shown to outperform synchronous designs. In terms of throughput, first of all, the high-speed operation of asynchronous circuits ensures the event-driven and circuit-delayed-based execution of each operation with low temporal distortion. Secondly, the use of group-encoding increases the maximum number of events that can be sent each time. Thirdly, the address fuser exploiting the spatiotemporal correlation of events increases the encoding efficiency compared to the row/column-based group-encoding used by synchronous designs. Finally, group encoding reduces the number of required Lservers per Hserver, thereby easing fan-in requirements. The modularity and scalability of the proposed design are also guaranteed as it is composed of multiple reusable modules, which can be easily reconfigured to form the required size and structure thanks to the support of the ACT flow.

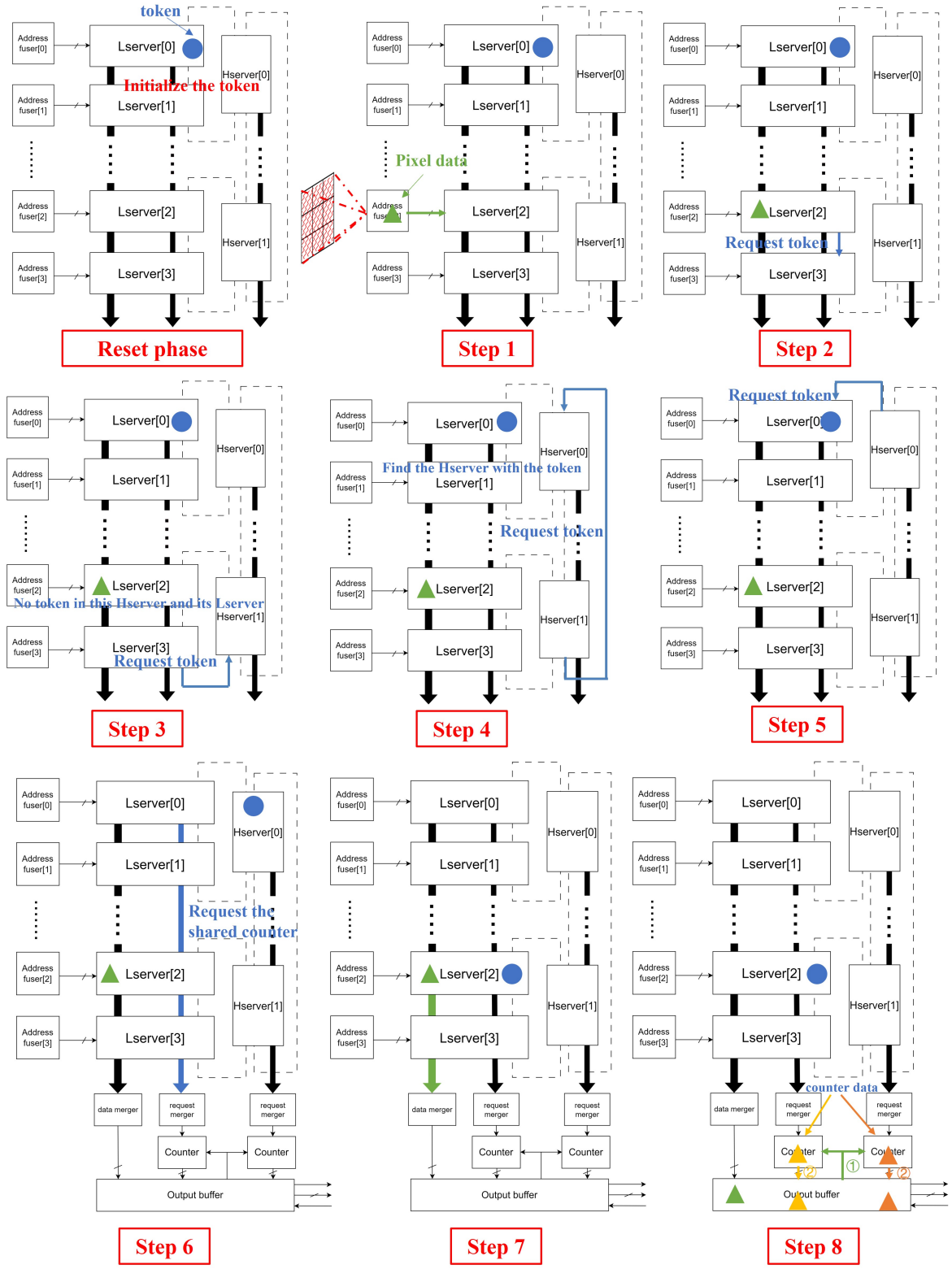
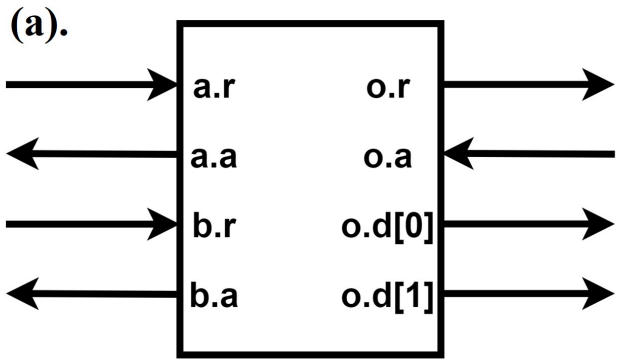


Figure 3.2: Steps of sending grouped events at once in the proposed design.

3.3 Address fuser



(b).

Name	Function
z	Indicates the end of the request collection
va	Indicates that the **a** port did not make a request at the end of the sampling window
vb	Indicates that the **b** port did not make a request at the end of the sampling window

Figure 3.3: (a). Schematic of a 2-pixel address fuser. (b). The internal variables used in this 2-pixel address fuser.

The address fuser is the component directly connected to pixels. It can be connected to multiple pixels, but for clarity, let us consider the example of two pixels: **a** and **b** as shown in Fig. 3.3(a). **a** and **b** both use two wires to communicate with the address fuser, namely the request wire (.r) and the acknowledge wire (.a). For the 2-pixel address fuser, the output **o** is encoded with 2-bit bundled-data with the data lines o.d[0] and o.d[1]. When a pixel requests, it sets request .r high and performs handshaking with the address fuser. After the request passes through the address fuser, the encoded request in the sensing field will be outputted through **o**. The encoding scheme is fairly simple, here o.d[0] and o.d[1] indicate whether **a** or **b** have a request, respectively. To perform the encoding operation, three internal variables are used, which are defined in Fig. 3.3(b). The CHP code of the address fuser is as follows:

```
CHP: [ (#a|#b)-> z+;       // z is set high if there is request in a or b.
     [[#a->o.d[0]+;a? [] ~#a -> va+],  // Detect if there is request in a.
     [#b->o.d[1]+;b?  [] ~#b -> vb+]]; // Detect if there is request in b.
      o!;                  // Send out the detection result via o.
      o.d[0]-,o.d[1]-; z-;va-,vb-]  //Resetting everything for the next round.
```

Here [X→... []Y→...] is the format of so-called deterministic selection, which means one and only one of the conditions X or Y is true and executed. The comma between operations means that they are executed in parallel. The "#" symbol represents a probe, which is used to determine if there is a pending communication action on a port. For example, "#**a**" indicates querying whether "**a**" is attempting to send. In QDI circuits, a probe is usually translated into a wait instruction at the HSE stage. When the address fuser is triggered by the request of **a** or **b**, z is pulled up. There is a delay line between the input (#**a** | #**b**) and the z variable, which is not shown in the CHP as CHP only describes high-level communication. The details of how the delay line functions in z can be found in Appendix A.1. In order to facilitate the reader's understanding, an example timing diagram of the address fuser, accompanied by its corresponding CHP and annotated with step numbers, is presented in Fig. 3.4.

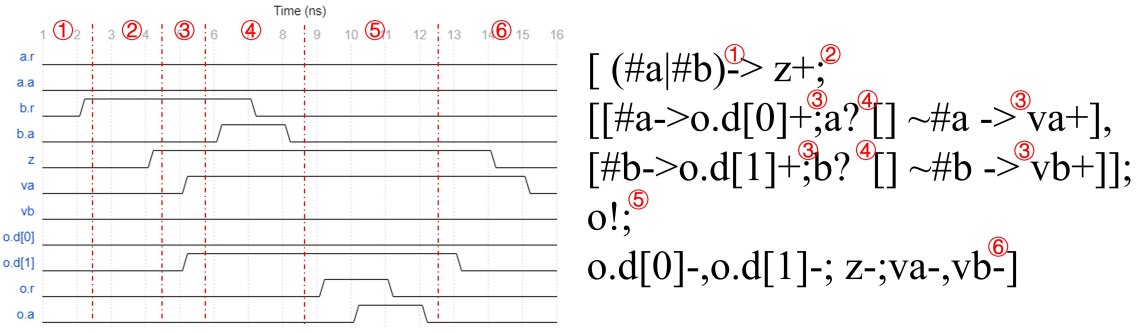


Figure 3.4: An example timing diagram of the address fuser accompanied by its corresponding CHP and annotated with step numbers.

3.4 Low-level server

The Lserver is connected to the address fuser, neighbouring Lservers, request merger, and data merger. It is responsible for receiving pixel data from the address fuser, requesting the token, requesting the shared counter to track the position of the token, and sending out the pixel data. Compared to the CHP for the Lserver provided in [8], a new control flow is applied to support the use of the address fuser.

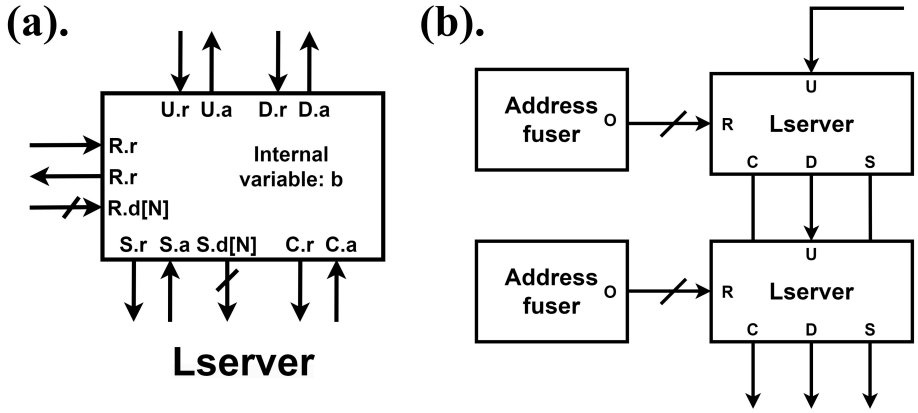


Figure 3.5: (a). Schematic diagram of the Lserver. (b). The interconnection between Lservers and address fusers. For brevity, the inputs of address fusers are omitted.

The structure of the Lserver is shown in Fig. 3.5(a), assuming that each address fuser is connected to N pixels. It has five ports, namely **R**, **U**, **C**, **D**, and **S**, and an internal variable **b** to represent if the Lserver possesses the token. The **R** and **S** ports are encoded in N-bit bundled-data to transfer the N-bit data from address fuser, while the other three ports **U**, **D** and **C** use a request (.r) and an acknowledge wire (.a) as they only involve handshaking and not data transfer. The functions of each port and the internal variable are shown in Table 3.1.

Fig 3.5(b) shows the interconnections between Lservers and address fusers. For brevity, the input of the address fuser is omitted. The **R** port is connected to the **o** port of the address fuser, and the **D** port of the top Lserver is connected to the **U** port of the bottom Lserver.

Port name	Function
R	Receive request and data from the **o** port of the address fuser.
U	Receive request for the token from another Lserver or Hserver.
C	Request the counter to increase by 1
D	Request the token from Lserver or Hserver
S	Request the output buffer and send the received data
b	Indicate whether the current Lserver has the token.

Table 3.1: Functions of each port and internal variable of the Lserver.

The **C** port and **S** port are connected to the request merger and data merger, respectively, to access shared resources. The CHP code of the Lserver is as follows:

```
[|#R -> [b -> skip[]~b -> D!]; //If it receives a request from the
                               //address fuser and has the token,
                               //then skip, otherwise request.
b+; R?v; S!v;                  //Once the token is received, then
                               //receive the data from the address
                               //fuser and output it via the S port.
[]#U -> [b -> skip[]~b -> D!]; //If requested for the token, check if there
                               //is a token, If so, skip, otherwise request.
C!; b-; U?|],                  //Increase the counter and lose the token.
```

Here [|X→... []Y→... |] is the format of so-called non-deterministic selection, meaning that more than one condition may be true at the same time and an arbiter is needed. This non-deterministic selection is used for the input ports **R** and **U** because the data input from the address fuser and the requests from the neighbouring Lserver could occur simultaneously. In such cases, in order to ensure the circuit's state is strictly correct, an arbiter is used. The arbiter is able to select one of its inputs to execute and keep the other one waiting (the working principle of an arbiter is introduced in Appendix A.2).

When the **R** port receives a request from the address fuser, the Lserver first checks whether it has the token. If it does, it skips the request on **D**; if it does not, it passes the request to its neighbouring Lserver or Hserver through the **D** port and once the handshake of **D** finishes, **b** is set to 1. A value of 0 means no token, while a value of 1 means it has the token. Then, the Lserver completes the handshake on the **R** port, receiving the pixel data from the address fuser. Finally, it sends the data to the Output buffer via the **S** port. Here, **v** is a dummy variable used to conform to the grammar of the CHP language, and it is used to indicate that the data sent by the **S** port is identical to the data received by the **R** port.

When the **U** port receives a request, the Lserver also checks if it has the token. Similarly, if it does, it skips the request on **D**; if it does not, it passes the request to its neighbouring Lserver or Hserver through the **D** port. Once the handshake of **D** finishes, it requests the shared counter to increment by 1 through the **C** port, and then it sets **b** to 0, indicating it loses the token.

In order to understand how the Lserver requests tokens from other Lservers, the reader needs to remember that the **D** port of the Lserver is connected to the **U** port of the neighbouring Lserver. A simple example and flowchart are shown in Fig. 3.6 with red arrows

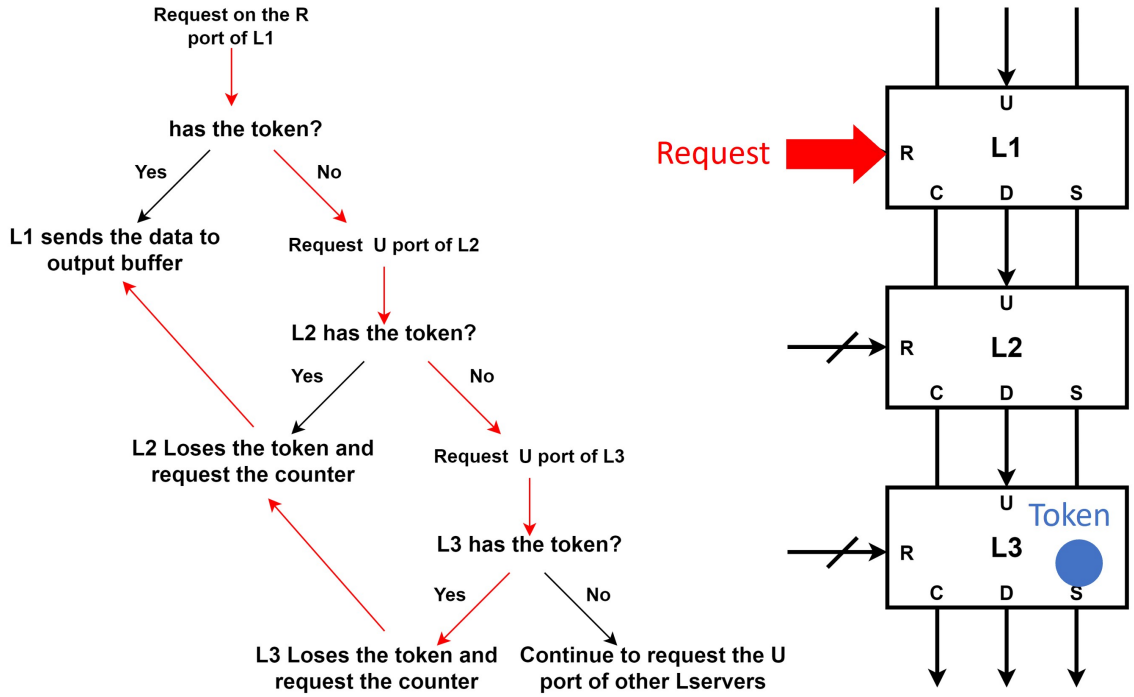


Figure 3.6: A simple example about requesting tokens between Lservers. The process is shown on the left, and the interconnection of L1, L2, and L3 is shown on the right.

indicating the execution sequence. There are three interconnected Lservers L1, L2, and L3, among which L3 has the token and the **R** port of L1 requests.

3.5 High-level server

The Hserver is connected to both Lservers and Hservers, acting as a bypass for token movement. The ports of the Hserver are illustrated on the left side of Fig. 3.7. It has five ports, which are **U**, **UH**, **D**, **DH**, and **CH**, and an internal variable **b** to represent the position of the token. Its **U** and **D** ports are similar to those in the Lservers. They are connected to the Lservers, which enables the token movements between Lservers and their Hserver. The **UH** and **DH** ports of the Hserver serve similar functions except that they are connecting Hservers with each other. The **CH** port serves as the port to access the shared counter for Hservers. Since there is no multi-bit data transmission involved, all ports of the Hserver are implemented with only two wires, namely the request wire and the acknowledge wire. The CHP code of the Hserver has been provided in [8] and it is reused unchanged:

```
*[[| #U ->                  // Request from its Lserver.
   [b=0 -> skip             // If it has the token, skip.
   []b=1 -> D!              // If b is 1, request the token from an Lserver.
   []b=2 -> DH!];b:=1; U?   // If b is 2, request the token from other
                            // Hservers, and then set b=1 and finish
```

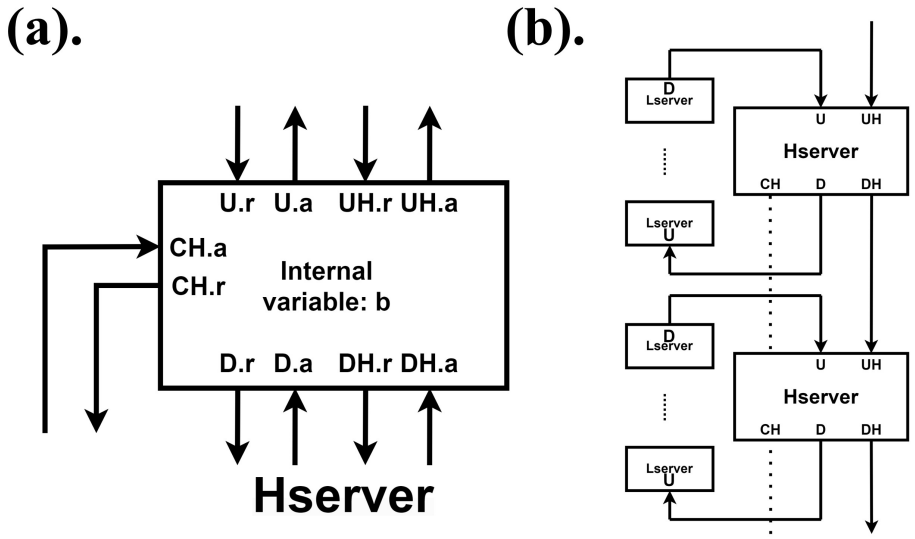


Figure 3.7: (a). Schematic diagram of the Hserver. (b). The interconnection between Hservers and Lservers.

```
                              // the handshake on U.

[]  #UH ->                    // Request from another Hserver.
      [b=0 -> skip            // If it has the token, skip.
    [] b=1 -> D!              // If b is 1, request the token from an Lserver.
    [] b=2  -> DH!];          // If b is 2, request the token from another
    CH!; b:=2; UH?            // Hserver, increase the counter, set b=2
|]]                           // and finish handshake on UH.
```

Similar to the Lserver, the **U** and **UH** ports in the Hserver may be requested simultaneously, so non-deterministic selection is used. The variable ***b*** represents the token status in the Hserver. The three possible values of ***b*** are as follows:

1. When b=0, it indicates that the Hserver holds the token.
2. When b=1, it means that one of its Lservers possesses the token.
3. When b=2, it means that neither the Hserver nor its Lserver has the token.

When the **U** port is requested by an Lserver, the Hserver will operate differently depending on the value of **b**. If **b=0**, it skips requesting the token and sets **b** to 1, indicating that the token is now in one of its Lservers; if **b=1**, it will request the token from the first of its Lservers; if **b=2**, then it will request the token from other Hservers via the DH port and the handshake of **DH** will not be completed until it finds the token. Regardless of the situation, the result at the end must be **b=1**, that is, the token is moved to one of its Lservers. Lastly, the handshake on the **U** port will be completed.

When the **UH** port is requested by another Hserver, similarly, the Hserver will operate differently depending on the value of **b**. If **b=0**, it skips and sets **b** to 2, indicating that the token has been passed to another Hserver; if **b=1**, it will request the token from its Lserver; if **b=2**, it will continue to requests token from other Hservers. The result at the end must be **b=2**, that is, the token is moved to another Hserver, and the shared counter of Hservers

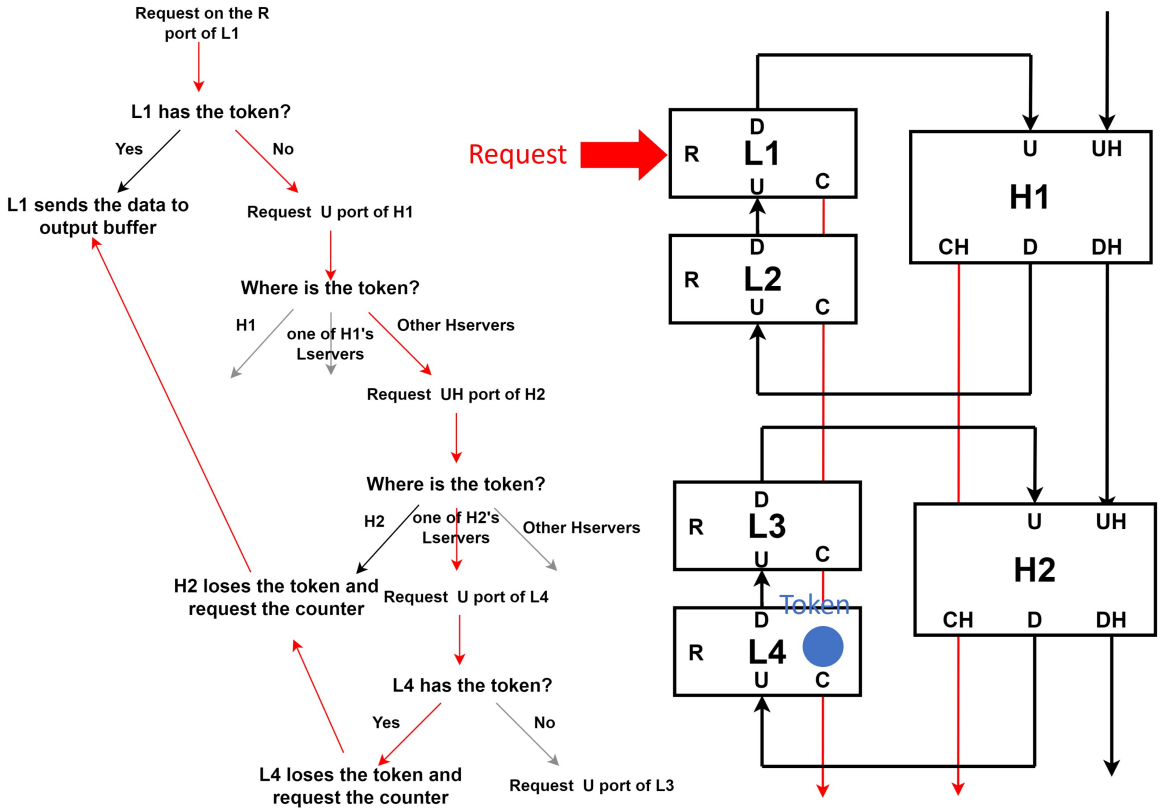


Figure 3.8: A simple example about requesting tokens between Lservers via Hservers. The process is shown on the left, and the interconnection of L1, L2, L3, L4, H1, and H2 is shown on the right.

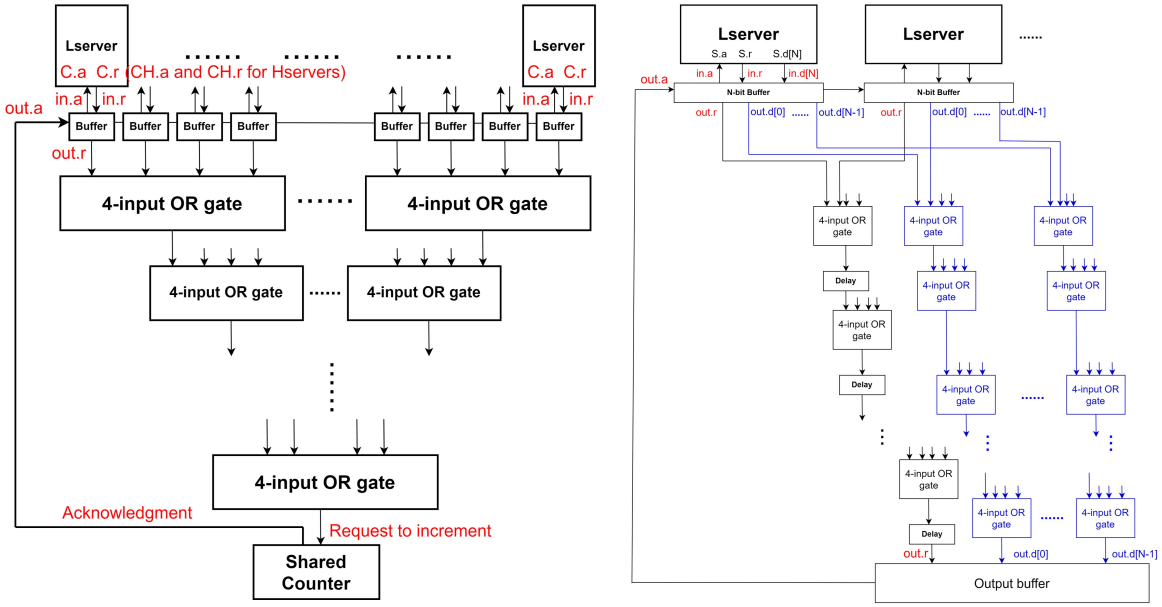
will be requested to record the position of the token. Lastly, the handshake on the UH port will be completed

As we did for Lserver, we also provide a simple example flowchart shown in Fig. 3.8 with red arrows indicating the execution sequence to help the reader understand the working principles of the Hserver. There are two interconnected Hservers, H1 and H2, and their Lservers L1, L2, L3, and L4, among which L4 has the token and L1 requests it.

3.6 Request merger and data merger

A module that can effectively handle the situation where multiple modules need to access the same shared resource is needed in the proposed address-fused HTR. For example, all Lservers, as well as Hservers, will access the corresponding shared counter every time they move the token in order to track its position. Also, when an Lserver acquires the token and tries to send out the pixel data, it needs to access the shared Output buffer. While also needed in the original HTR scheme [8], no specific implementation was proposed, hence we propose our solution here.

Fig. 3.9(a) shows the structure of the request merger module. The request to access the



(a.) Request merger **(b). Data merger**

Figure 3.9: Schematic of the (a). request merger module and (b). data merger. The major difference between a data merger and a request merger is that for a data merger, a delay needs to be added to the 'out.r' OR gate tree to ensure that it propagates slower than the data.

counter from an Lserver or an Hserver will not be directly connected to the OR-gate tree but will be connected to a buffer first. This is because if the handshake with the counter is performed directly through the OR-gate tree, since the acknowledge signal of the counter is shared by all, the C ports of Lservers would be directly interconnected with each other, while the CH ports of Hservers would also be interconnected, which is dangerous. For **N** inputs from servers, the request merger consists of **N** buffers and an OR-gate tree with **N** inputs. The **C** ports of Lservers (or the **CH** ports of Hservers) are connected to the **in** ports of the buffers, and the output request wires of the buffers are connected to the inputs of the OR-gate tree. The output acknowledge wires of all buffers are connected to a single acknowledge wire from the shared counter. The tree formed by the 4-input OR gate shown in Fig. 3.9(a) is just an example. The design and sizing of the OR-gate tree is flexible and can be optimized based on specific cases.

The data merger is similar to the request merger, so they are introduced together. The function of the data merger is to ensure quick and conflict-free accesses from the **S** ports of all Lservers to the output buffer. As shown in Fig. 3.9(b), the key difference with the request merger is two-fold: (i) it requires an OR gate tree for all data, and (ii) the handshaking OR gate tree contains a delay to ensure that the data arrives first.

An example flowchart shown in Fig. 3.10 is provided to illustrate the workflow of request merger and data merger. When **out.r** and **out.d**[**N**] are propagated through the OR-gate tree, **out.r** may propagate faster than some of the bits of **out.d**[**N**], causing erroneous data received by the Output buffer. The delay lines in the OR-gate tree of **out.r** are used to ensure

that **out.r** always arrives at the Output buffer later than **out.d[N]**. For this reason, we delay the output of each OR-gate in the OR-gate tree for **out.r** with a delay line of 1 ns.

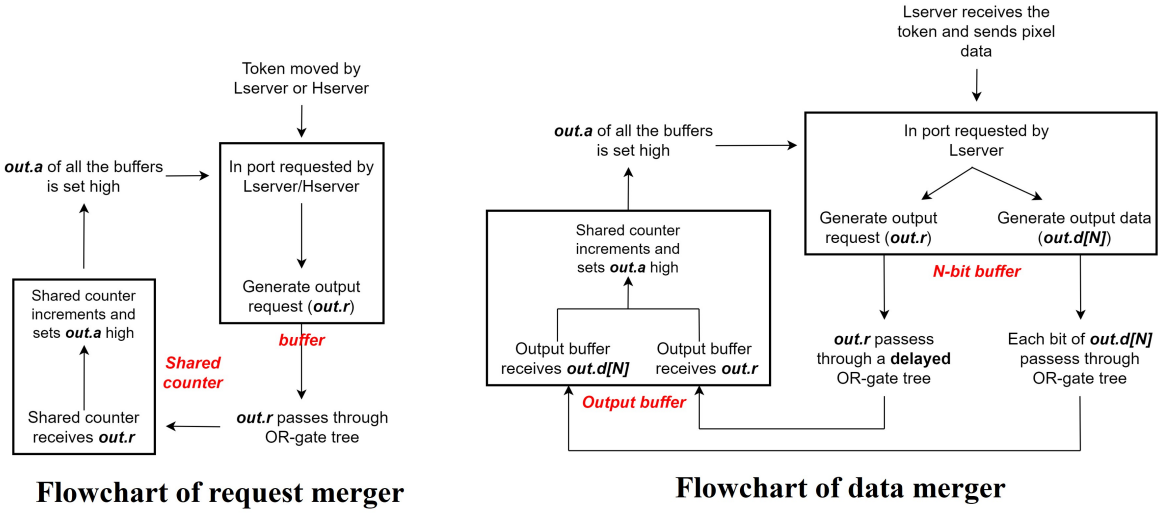


Figure 3.10: The workflow of request merger and data merger.

The buffer used in the request merger is the same as the one in the example of Section 2.2, and the implementation details of the buffer can be referred to in Fig. 2.5. The CHP code for the buffer is as follows,

```
*[in?;out!]   // If 'in' is requested, immediately
              // send this request via 'out'.
```

The CHP code for an **N**-bit buffer used in the data merger is as follows:

```
*[in?n;out!n]   // If 'in' is requested, receives data
                // and immediately send this data via 'out'.
```

Here, '**n**' is a dummy variable used to conform to the grammar of CHP language, indicating that the data sent via the **out** port is the data received by **in** port. The **S** ports of Lservers are connected to the **in** port of **N**-bit buffers. The function of this buffer is to accept all the input data immediately from the **in** port and send it via the **out** port. To transfer correct data with the buffer, it should be guaranteed that the output data **out.d[N]** is correct before **out.r** is pulled up. To ensure this, a small delay is added to **out.r** to delay the execution of **out.r+** (the complete HSE code can be referred to in Fig. 2.5). The example timing diagram of the buffer that transmits n-bit data is shown in Fig. 3.11:

3.7 Counter

To track the address of the L/Hserver that possesses the token, a simple ripple carry-based counter is used, as shown in Fig. 3.12. The increment of the counter is designed to match

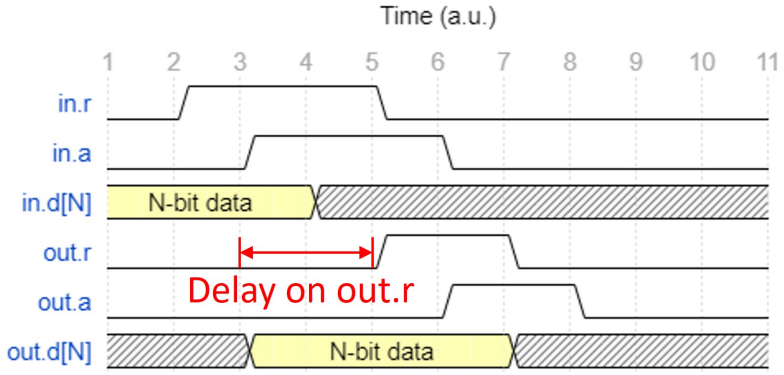


Figure 3.11: Example timing diagram of the N-bit buffer.

each request made by an L/Hserver when moving the token. In this way, the value of the counter corresponds to the position of the token. In this project, a simple ripple carry counter based on [32] is used.

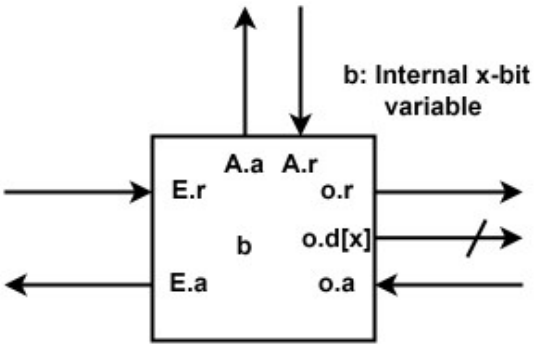


Figure 3.12: Schematic diagram of the counter.

There are three ports: **A**, **E**, and **o**, along with an internal variable **b**. As shown in Fig. 3.13, the A port is connected to the output of the request merger and receives increment requests from an L/Hserver. The **E** port is connected to the **E** port of the Output buffer, which controls when the counter will output its value. Lastly, the **o** port is connected to the **C1/C2** port of the Output buffer, and is responsible for outputting the current count value, ***b***, with ***o.d[x]***. The variable ***b*** is an x-bit internal variable that keeps track of the current value of the counter. The value of x is determined by the number of Lservers and Hservers. For instance, with 8 Hservers, a 3-bit ***b*** is required to represent the token's position among Hservers. The CHP of the counter is as follows:

```
*[[#A -> b:=b+1 ;A?  //When there is a request from an L/Hserver,
                     //make b=b+1, and finish the handshake.
 []#E -> o!b;E?]]}   //When there is a request from the output buffer,
                     //send out the data and finish the handshake.
```

The function of this counter is very straightforward. It has only two functions: 1) When **A** receives the request, it will increment itself and record the data into ***b***. 2) When **E** receives

the request, it will send the data ***b*** out from **o** port.

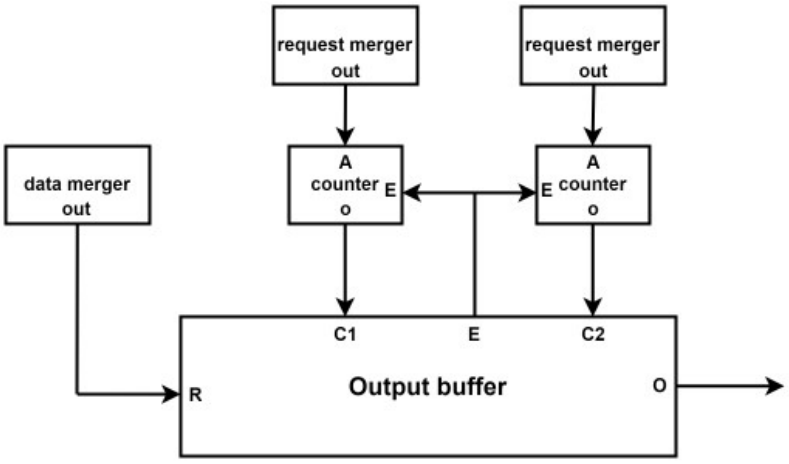


Figure 3.13: Schematic diagram illustrating the interconnections between the counter, Output buffer, and other modules.

3.8 Output buffer

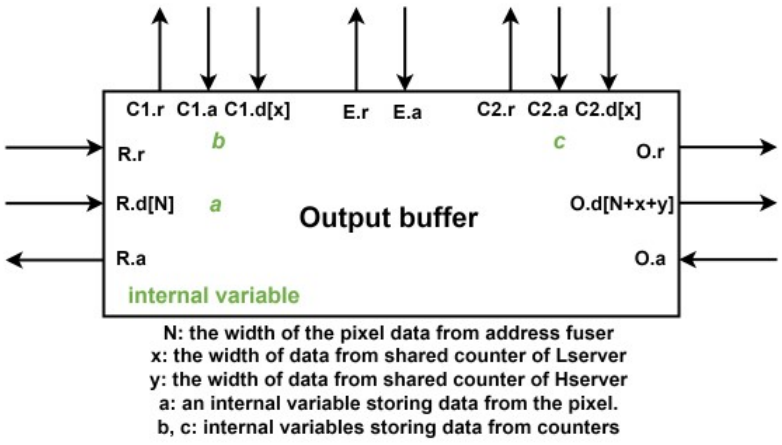


Figure 3.14: Schematic diagram of the Output buffer.

The purpose of the Output buffer is to receive data from the data merger and values in the counter for the final output. The final output data consists of the grouped pixel data and the address of the token. As illustrated in Fig. 3.13, the **R** port is connected to the **out** port of the data Merger. The **E** port is connected to the **E** port of the shared counters of both Lservers and Hservers. **C1** and **C2** are respectively connected to the ***o*** port of the shared counters of Lserver and Hserver. Finally, the ***O*** port, representing the final output of the design, is connected externally. The ports and internal variables of the Output buffer are shown in Fig. 3.14 and the CHP code of the Output buffer is as follows:

```
*[[#R-> R?a;    // Receive pixel data and store in variable a.
   E!;          // Requesting two counters for token address.
   C1?b,C2?c;   // Receive the token address from C1 and C2 port.
   O!(a,b,c)]   // Output the pixel data and corresponding token address.
```

3.9 Parameterization of the design

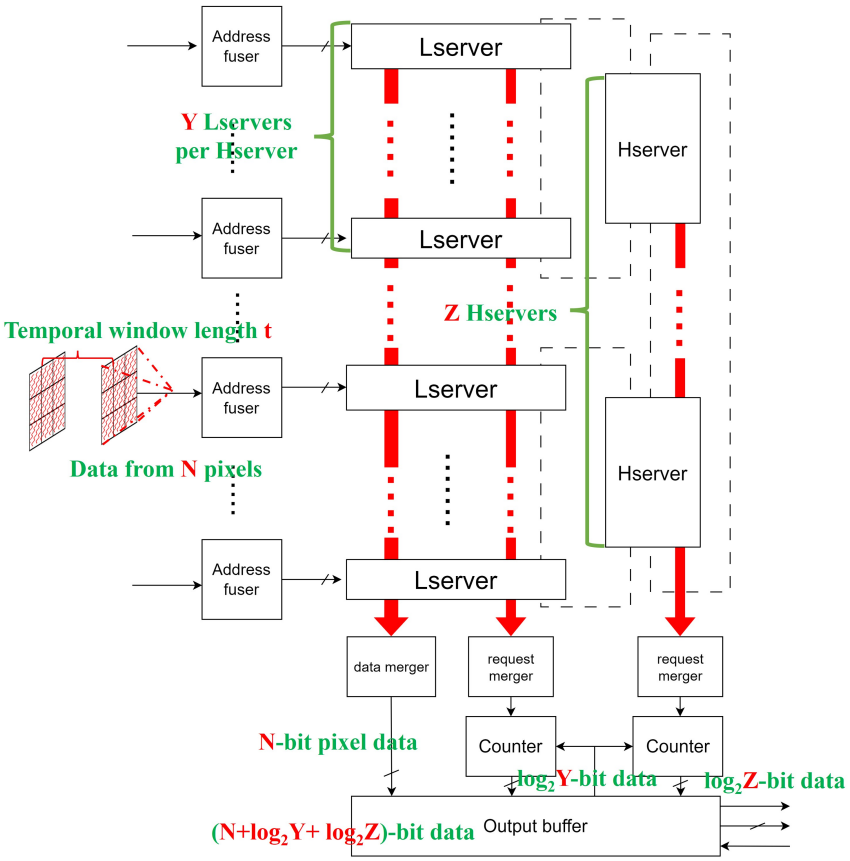


Figure 3.15: Schematic diagram of the parameterized address-fused HTR, with the parameters t, N, Y, and Z highlighted in red.

In Section 3.3, we only introduce a 2-pixel address fuser for clarity, but the spatiotemporal window of the address fuser is configurable. Also, the number of Lservers in each Hserver is configurable as well. Different application scenarios require different spatiotemporal windows and indeed, configuration of HTR topology. To quickly adjust the address-fused HTR, we parameterized the entire design. For a total of $\mathbf{P}$ pixels and an address fuser delay of $\mathbf{t}$, we assume that the number of pixels in the sensing field of each address fuser is $\mathbf{N}$, that $\mathbf{Y}$ Lservers belong to each Hserver, and that there are $\mathbf{Z}$ Hservers in total. They follow the relationship of $\mathbf{P} = \mathbf{N} \times \mathbf{Y} \times \mathbf{Z}$, and the corresponding schematic diagram is shown in Fig 3.15.

In the ACT process, similar to most programming languages, there are built-in parameterization features that allow for module generation using parameters. We used this feature

and parameterized the PRS of every module including the length of the delay line in the address fuser. However, the connections between these modules cannot be automated by the tools in the ACT flow. To solve this issue, a simple program has been written in Python for this project. It can generate the corresponding interconnections between the parameterized modules. In this way, the PRS of an address-fused HTR for **N** pixels with a temporal window of **t** can be easily generated by specifying **(t,N,Y,Z)**. This can provide good scalability for our design and ease our simulations in the next chapter.

4 Results and discussion

4.1 Overview

This chapter presents the simulation results and the discussion. Simulation results include switch-level and transistor-level simulation, layout and verification of the address fuser, and a comparison between conventional HTR and the proposed address-fused HTR with 16, 64, and 256 inputs, under different input behavior modes. Lastly, the discussion of the results is presented.

4.2 Verification of the design

4.2.1 Simulation setup

The PRS of a (2,2,2,2) example generated with the parameterization method mentioned in Section 3.9 is used to verify the correctness of our design. The generated PRS is converted into a switch-level netlist and a transistor-level one. Then, the netlists are simulated with Irsim and Xyce, for switch- and transistor-level simulation, respectively. The schematic diagram of the example is shown in Fig. 4.1. It consists of two Hservers, two Lservers for each Hserver, and each address fuser is connected to two pixels with a temporal window of 2 ns. We make three pixels, p[2], p[3], and p[4], request simultaneously after the Reset phase and observe the behavior of our design. The operations that happened in the Reset phase can be referred to in Section 3.2 and Fig. 3.2.

4.2.2 Switch-level simulation

The process of token movement between Lservers and Hservers, as described in their HSE in Appendix A.3 and A.4, involves a large number of variables. In order to allow readers who are not familiar with the detailed HSE and PRS of the whole architecture to grasp the workflow of our design, visualization and a subset of representative signals are shown in Fig. 4.2. The signals of interest include the external input request and acknowledge signals for p[2], p[3], and p[4] (p[2].r, p[2].a, p[3].r, p[3].a, p[4].r, p[4].a), the stored pixel data in the **Output buffer** (out[0], out[1]), the request signals to the shared **Counters** via the **request merger** (c1.A.r and c2.A.r), as well as the signals from the **Output buffer** used to request the internal values of the two shared **Counters** (c1.E.r, c2.E.r). The waveform at the bottom of Fig. 4.2 shows the switch-level simulation results obtained using Irsim and the corresponding steps in the workflow are marked in red font.

In the switch-level simulation, the design is shown to operate correctly. While switch-level simulation offers a fast and accurate behavioral simulation of the system, it is based

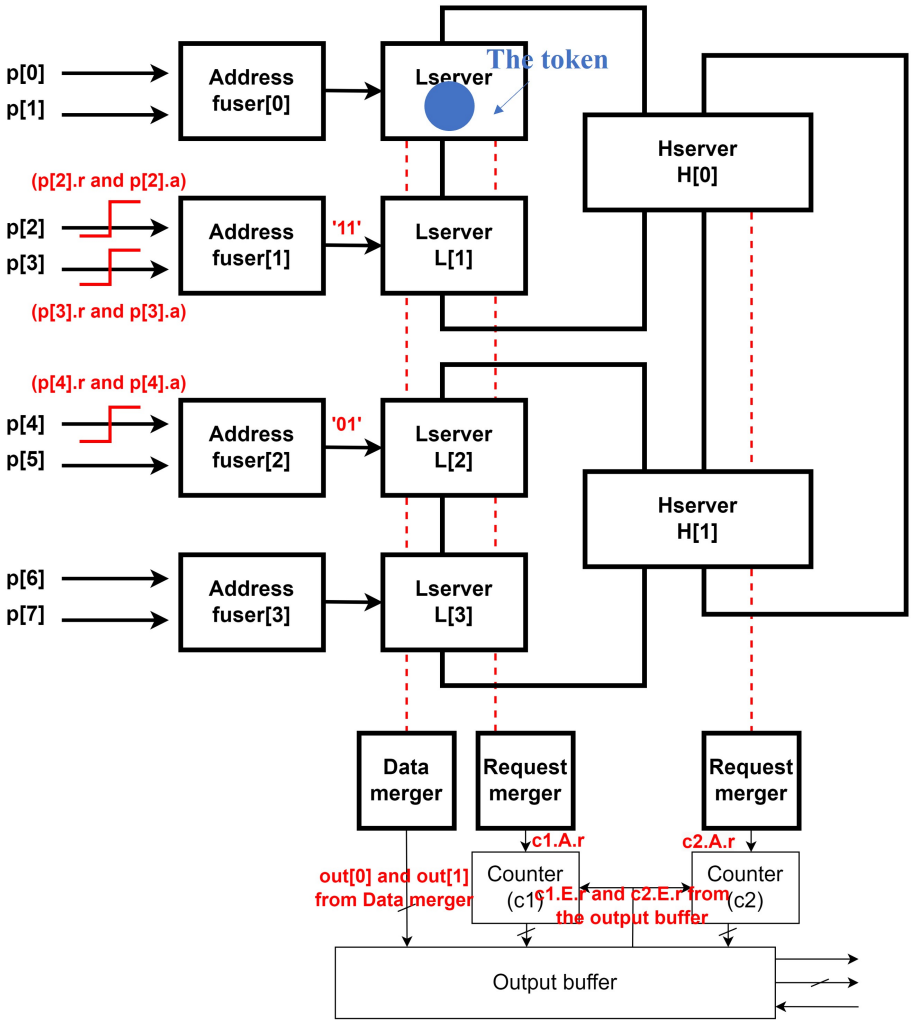


Figure 4.1: Schematic diagram of a simple 8-input address-fused HTR structure. The signals of interest are marked in red.

solely on simplified transistor models and does not take into account the complex transistor-level phenomena that may be introduced by arbiters and weak keepers (see Section 2.2.2.4). Therefore, transistor-level simulation is also necessary to validate the system.

4.2.3 Transistor-level simulation

In the transistor-level simulation, we replaced the original switch model with CMOS transistors from the Skywater 130nm PDK [24]. The Skywater 130nm PDK is an open-source PDK and well-supported by the ACT flow. With the same input scenario and the same signals of interest, transistor-level simulation is performed, and results are shown in Fig. 4.3.

As per the test setup, the requests for p[2], p[3], and p[4] are simultaneously pulled up at t=2.5 ns. After the requests are acknowledged by the corresponding address fusers, they are then pulled down at t=10 ns. To facilitate a straightforward comparison between switch-level and transistor-level simulation results, we have annotated the numbers corresponding to the

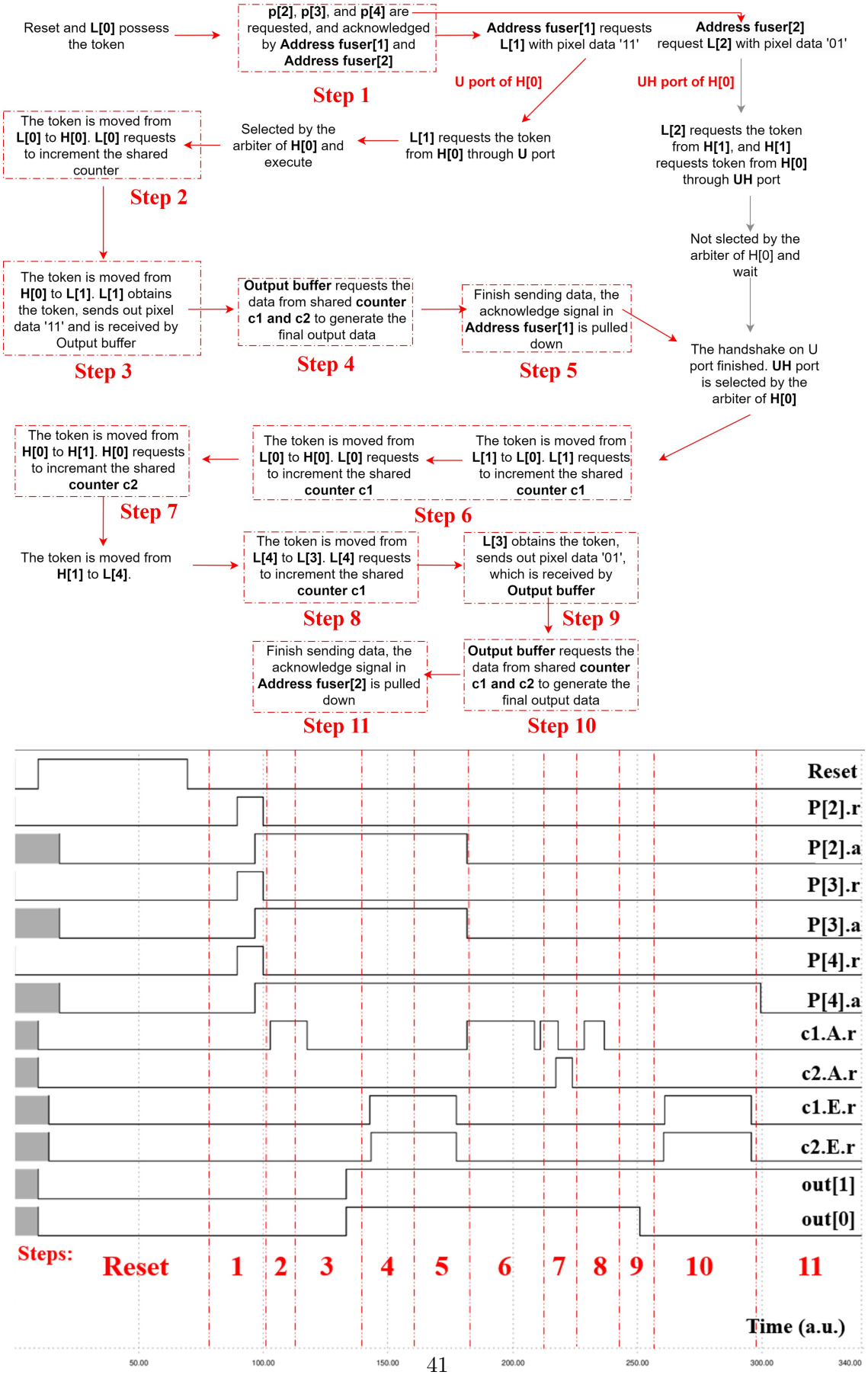


Figure 4.2: Workflow and timing diagram of an 8-input address-fused HTR when p[2], p[3], and p[4] request simultaneously. The sequence numbers in the timing diagram correspond to the steps in the workflow and the signals of interest are highlighted in red. These results were obtained from a switch-level simulation, hence time is shown with arbitrary units (a.u.).

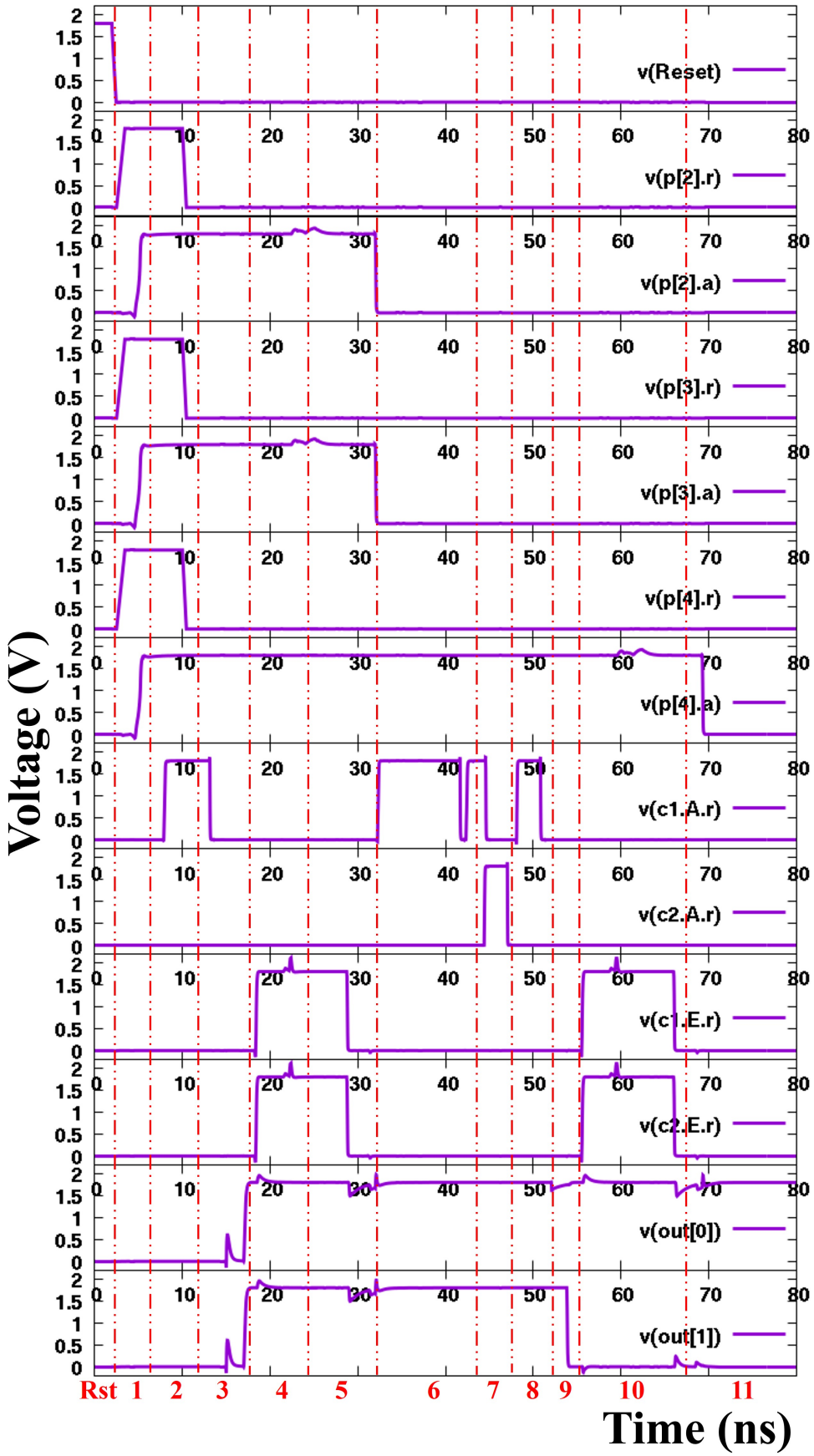


Figure 4.3: Transistor-level simulation results. For clarity, the numbers at the bottom correspond to the steps presented in the switch-level simulation.

steps presented in the switch-level simulation at the bottom of Fig. 4.3. It can be observed that the order of appearance of the signals is exactly the same as simulated in the switch-level simulation. This result confirms the circuit's integrity when the transistor-level effects, such as those in arbiters and weak keepers, are considered in the design.

We can also see some non-idealities in the circuit. For example, slow rise and fall times due to load, signal overshoot, and undershoot due to frequency response, etc. They did not make the operation of the design erroneous but are still worthy of attention. These non-idealities are largely due to the sizing of transistors. Bad sizing will produce non-ideal effects, affect performance, and even make the circuit unable to operate if the sizing is not large enough to drive the next-level signal in the QDI circuits. However, in the ACT flow, we do not directly design the size of a single transistor, but specify the drive strength of the PRS for sizing (the sizing method in the ACT flow can be referred to Appendix A.5). In contrast to the well-established optimization techniques for synchronous circuits, modern design tools for asynchronous circuits, including the ACT flow, do not have an effective solution to automatically optimize the driving strength for the whole system. In our design, we employ the minimal unit drive strength for all signals, excluding the output requests within each module. For those output requests, we allocate double the unit drive strength. This allows the circuit to function properly but might not be an optimal setup. How to optimize the drive strength as a whole for large-scale asynchronous circuits is still an ongoing research topic.

4.2.4 Layout and routing of address fuser

Following the process of the ACT flow, we used Magic VLSI [25], Dali [26], and TritonRoute [27] to complete the layout and routing with the Skywater130 PDK. Due to the large number of cells that need to be manually drawn in asynchronous circuits and the limited time, we only demonstrate the layout and routing of the key module, an example address fuser with two inputs. The layout of a 2-input address fuser in the Magic VLSI and its schematic diagram is shown in Fig. 4.4.

As mentioned in Chapter 2, the needed cells are first generated by prs2sim in the ACT flow, which are the blocks named abc_acx* on the left side of Fig. 4.4. These automatically generated cells are incomplete, as they do not contain connections between ports. Hence, designers are required to handcraft the layout for each cell using both metal1 and metal2 layers. An example cell 'abc_acx3' is shown on the bottom right of Fig.4.4. The layout of a 2-input address fuser includes a total of 147 transistors, with automated placement by Dali with a density of 0.58. The routing of the third metal layer (metal3) and fourth metal layer (metal4) was completed by TritonRoute and four metal layers in total were used in the layout process. After manually rectifying a few DRC errors that arose during the routing phase, we obtained the final layout of the 2-input address fuser, of which the area is 35.5 μm by 30 μm. For the post-layout simulation and final check, we used Magic VLSI and ACT to generate the netlist of the address fuser and compared it with the result in the pre-layout simulation by manually setting r[1].r high at t=4ns. The r[1].r here is a wire of the address fuser, connected to the request emitted from an external pixel. Given that the internal module signal execution sequence delves into specific PRS translation, we have included the PRS code of the address fuser in Appendix A.1 and conducted a side-by-side analysis in Fig. 4.5, which presents the transistor-level simulation results of active signals from both pre-layout and post-

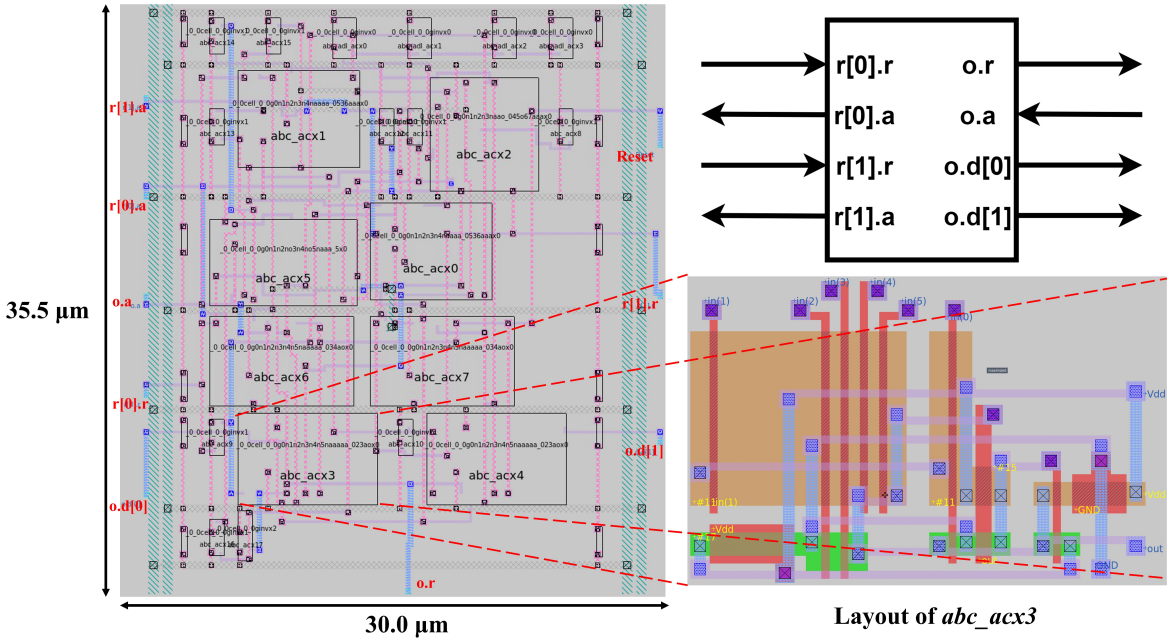


Figure 4.4: Layout of a two-input address fuser with the Skywater130 PDK, viewed in Magic VLSI, and its corresponding schematic diagram. The abc_acx* names are auto-generated cell names.

layout netlists, placed side-by-side with pre-layout results in black and post-layout results in red.

Most signals align closely with the pre-layout simulation results. However, the fall time of **o.r** deviates by approximately 1 ns from its pre-layout counterpart. This can be attributed to the parasitic capacitance and parasitic resistance in the layout. However, because of the quasi-delay insensitivity characteristic of QDI circuits, such delays impact only the performance and not the correctness of the circuit operations.

4.3 Comparison between address-fused HTR and conventional HTR

In comparison to the HTR proposed in [8], the proposed address-fused HTR introduces the address fuser, extra control logic in Lserver, as well as the request merger, data merger, and Output buffer. Conversely, the address-fused HTR can send multiple events within the defined spatiotemporal window at once, reducing the number of transactions required. In order to fairly evaluate the proposed scheme, we also implemented the HTR structure proposed in [8] with the same Skywater 130nm PDK and compared it with the proposed address-fused HTR under different input scenarios. As shown in Fig. 4.6, these input scenarios include a single event located in the middle of the input pixel frame, simultaneous requests from half of the pixels in the middle of the input pixel frame (half frame), and simultaneous requests from all pixels (full frame). Furthermore, though not shown in Fig. 4.6, the random events scenario

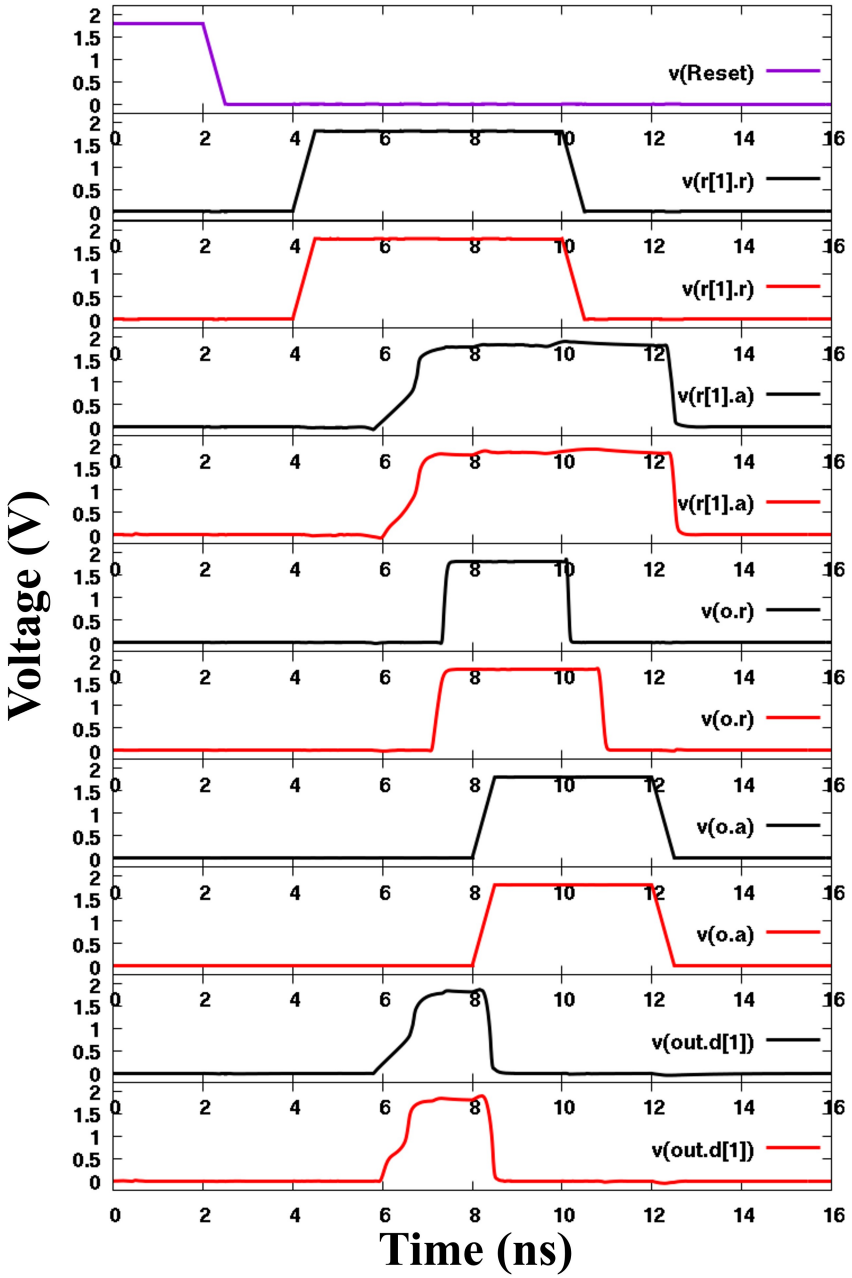


Figure 4.5: Comparison of Spice pre-layout and post-layout simulation results for the of the two-input address fuser. The pre-layout results are in black, while the post-layout results are in red.

and clustered events scenario are also included.

The parameterization method in Chapter 3 (section 3.9) is used. For $\mathbf{N}\times\mathbf{Y}\times\mathbf{Z}$ inputs with a temporal window of **t** nanoseconds in address fuser, the address-fused HTR is represented by **(t, N, Y, Z)**, where **N** is the data width in each address fuser (i.e., the number of connected pixels in each address fuser), **Y** is the number of Lservers in each Hserver, and **Z** is the number of Hservers. On the other hand, the conventional HTR is parameterized by

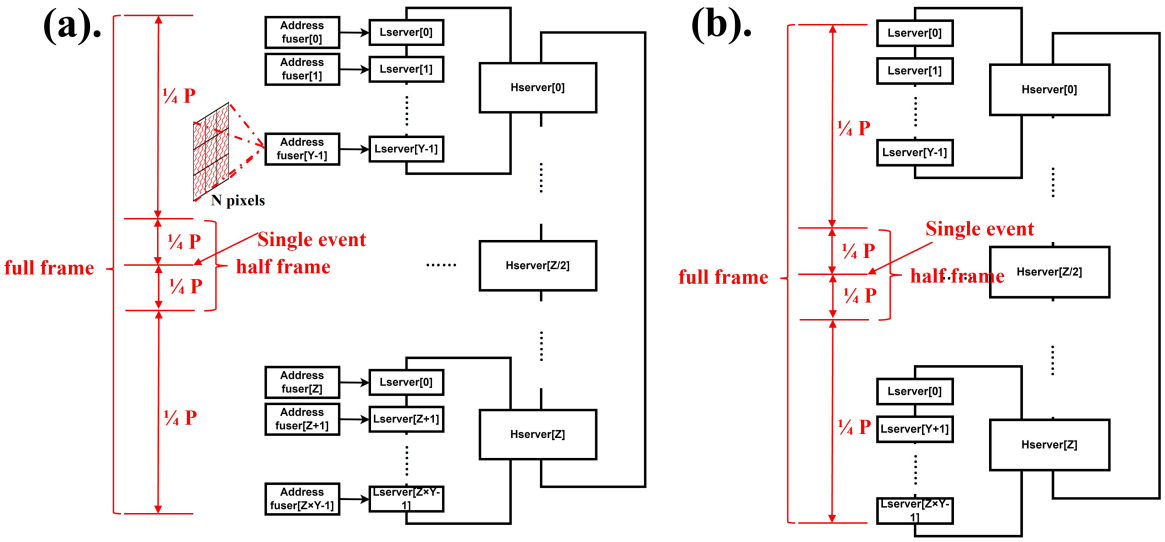


Figure 4.6: Schematic diagram of the three tested input scenarios, a single event, a half frame, and a full frame of (a). Proposed address-fused HTR, and (b). Conventional HTR.

(Y, Z), where **Y** is the number of Lservers in each Hserver, and **Z** is the number of Hservers. Based on the findings in [8], for the conventional HTR, we select $\mathbf{Y} = \mathbf{Z}$ to achieve a balanced performance across various input scenarios.

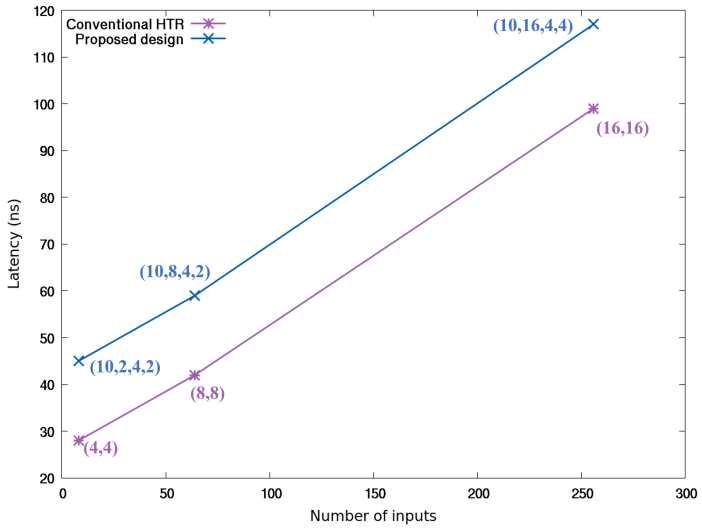


Figure 4.7: Single-event scenario: overall latency of the conventional HTR and our proposed design.

We begin with the input scenario of a single event, with results illustrated in Fig. 4.7. It is evident that the latency of our proposed design is consistently higher by approximately 18 ns compared to the conventional HTR. This delay primarily stems from two sources: firstly, the latency within the address fuser. In this example, with a 10 ns temporal window set in the address fuser, requests from pixels already experience a 10 ns delay before reaching

the Lserver. The second component of the delay is the additional control logic, such as the need in our design to route pixel data from the address fuser through the data merger to the output buffer after the Lserver has acquired the token. It is noteworthy that this delay remains almost constant, irrespective of the increase in input activity.

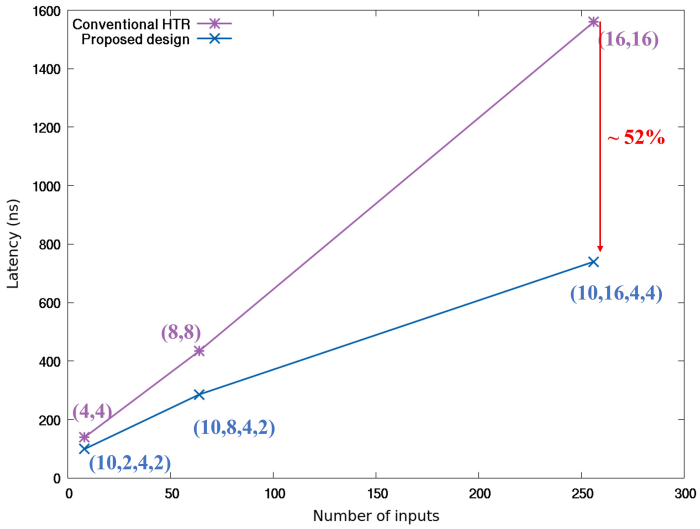


Figure 4.8: Half-frame scenario: overall latency of the conventional HTR and our proposed design.

In the context of the half-frame input scenario, the results are depicted in Fig. 4.8. Our proposed design exhibits a significant reduction in the required time compared to the conventional HTR, which is accentuated with the growth in input count: for 16 inputs and X=2, our design outperforms the original by approximately 28%; for 256 inputs and X=16, it exceeds the original performance by 52%. This pronounced advantage is attributed to the high spatiotemporal correlation in the half-frame scenarios. The address-fused HTR encodes and transmits multiple events simultaneously, substantially reducing the number of transmissions, which consequently mitigates the overhead of the control flow in the address fuser. Meanwhile, the delay introduced by the temporal window of the address fuser becomes less pronounced. Therefore, the speed benefits of the address fuser increase with the input activity and the number of pixels per address fuser (**N**).

The results for the full-frame input scenario corroborate our aforementioned discussion, as illustrated in Fig. 4.9. Still, with the presence of a high spatiotemporal correlation in the inputs, both the conventional HTR and address-fused HTR transmit double the events of the half frame roughly twice the time.

Since the aforementioned benchmarks were conducted under high spatiotemporal correlation conditions, we further tested our proposed design and the conventional HTR under a random-event scenario and a scenario with a moderate spatiotemporal correlation with the 256-input configurations. Both of these scenarios comprised ten pixel requests. In the random-event configuration, ten events were generated by a simple random number generator with values ranging from 0 to 255, yielding the following ten numbers: 134, 70, 17, 188, 197, 143, 179, 249, 105, and 132. Conversely, the scenario with a moderate spatiotemporal

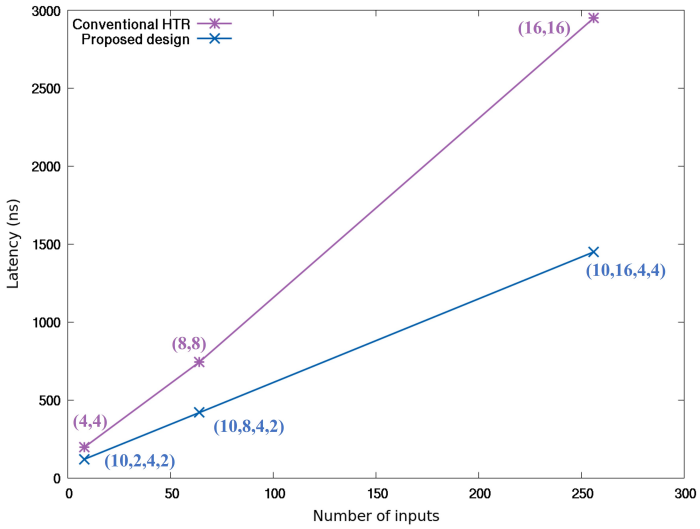


Figure 4.9: Full-frame scenario: overall latency of the conventional HTR and our proposed design.

correlation featured a 2x5 event cluster. We postulated that the temporal window employed was long enough to encompass this 2x5 event cluster. To simulate a typical scenario, this 2x5 event cluster was placed at the juncture of four neighbouring address fusers, as shown in Fig. 4.10(a). Under these two input scenarios, the time required for the proposed design and the conventional HTR are shown in Fig. 4.10(b).

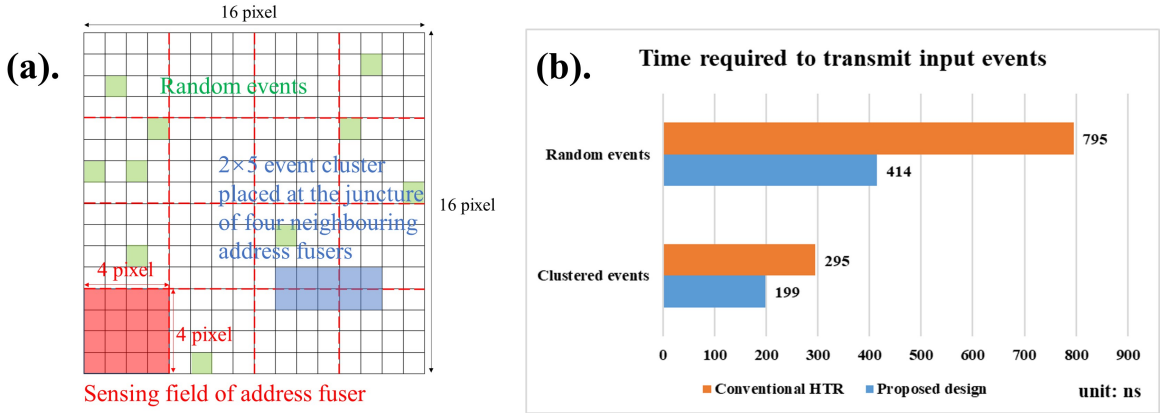


Figure 4.10: (a). Schematic representation of the 256-input configuration, the 4×4 sensing field in the address fuser, and the 2×5 event cluster positioned at the intersection of four adjacent address fusers, with random events highlighted in green. (b). Time needed for event transmission in the random-event scenarios and those with moderate spatiotemporal correlation for both the conventional HTR and our proposed approach.

In both the random-event scenario and the moderate-correlation scenario, our proposed design consistently outperforms the conventional HTR in terms of the required time. In the random-event scenario, there are two main reasons for which the address-fused HTR

has a more favourable latency. Firstly, it effectively groups events 134, 143, 132, and 179, 197 together, thereby reducing the transmission count from ten to seven. Secondly, with fewer Lservers and Hservers needed, the token movement among these events is expedited. In contrast, the conventional HTR requires ten separate transmissions and a longer token travel time, leading to nearly double the processing time. Interestingly, in the scenario with a moderate spatiotemporal correlation, the advantage of our design over the conventional HTR becomes less pronounced. This is due to our design transmitting the event cluster in four times since it spans four address fusers. Even though the conventional HTR still needs ten transmissions, the closeness of these requests results in shorter token movements, improving efficiency.

Overall, at the exception of the single-event scenario, the proposed address-fused HTR design is able to exploit spatiotemporal correlation in a broad range of scenarios and considerably outperform the conventional HTR scheme.

4.4 Software verification of spatiotemporal correlation

To understand the influence of spatiotemporal window dimensions on the event reduction rate, we conducted experiments with the Neuromorphic-MNIST (N-MNIST) dataset [33]. The NMNIST dataset is an artificially generated event dataset based on the conventional MNIST dataset of handwritten digits at the same visual scale (28x28 pixels). Given that the NMNIST dataset is an early small-scale dataset that may not reflect high-speed scenarios, we accelerated it to achieve an event rate of 166 Meps, which surpasses the event rate observed in noisy environments as reported in [28]. We test a random sample with the digit '9' and obtain the results shown in Fig. 4.11.

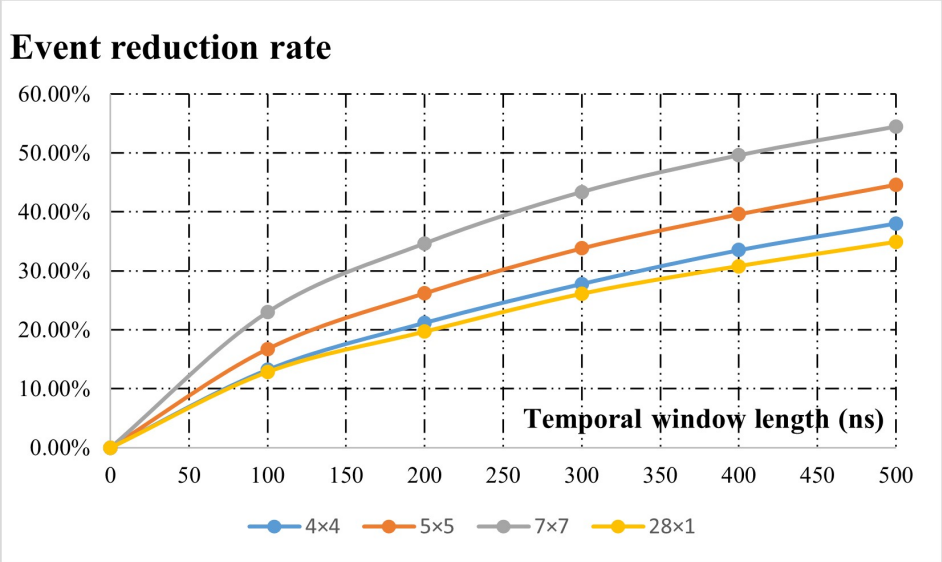


Figure 4.11: Trend of the event reduction rate as the size and shape of the spatio-temporal window changes.

Where the event reduction rate is calculated by :

$$\text{Event reduction rate} = (1 - \frac{\text{Number of transmissions}}{\text{Total number of events}}) \times 100\% \tag{4.1}$$

In Fig. 4.11, the legend 4x4, 5x5, 7x7, and 28×1 refer to the shape of the spatial windows. For instance, when using the 28x28 NMNIST dataset, a 4x4 window implies that the dataset will be divided into 7x7 pitches of 4×4 pixels each. The x axis denotes the size of the temporal window, which is essentially the span of time during which the address fuser can sample and transmit data collectively. It is evident that the event reduction rate increases with the size of the spatiotemporal window, while this increase gradually flattens out. Among these spatiotemporal windows, the conventional row/column-based 28x1 spatial window proves to be the least efficient when transmitting the NMNIST dataset. It encompasses 28 pixels, yet its event reduction rate is slightly below that of the 4x4 window which only encompasses 16 pixels.

4.5 Discussion

	Comparison done in 65nm bulk CMOS		Comparison done in Skywater 130nm
Input scenarios	Binary arbiter tree	token-ring	Proposed address fused HTR
256-input single event	-83.3%	+734.5%	depends on n (+18% for n=10 ns)
256-input half frame	+139.5%	-6.7%	-52.5%
256-input full frame	+128.6%	-10.2%	-50.8%

Latency compared to conventional HTR

Figure 4.12: Performance comparison the between binary arbiter tree scheme, the token-ring scheme, and the proposed address-fused HTR compared with the conventional HTR scheme. Summarised from [8].

This research started with the proposition of a novel asynchronous AER interface that capitalizes on spatiotemporal correlations between events, aiming to combine the benefits of enhanced throughput while maintaining high temporal precision. To evaluate our results, we implemented the conventional HTR as described in [8], using the Skywater 130nm process. Combining our results with the findings in [8], where the conventional HTR is contrasted with the binary arbiter tree scheme and token-ring scheme under a 65nm process, we developed the comparative analysis presented in Fig. 4.12. This comparison is grounded on two assumptions:

1. We overlook circuit-level optimizations, like the sizing of the asynchronous circuits, focusing more on the architectural comparison rather than detailed circuit implementations.

2. We assume that the relative latencies of the compared schemes are primarily determined by their architectures rather than the technology node. Their relative latencies should not exhibit significant variations under different processes.

Under these two assumptions, the proposed address-fused HTR shows clear advantages in overall performance. In single-event scenarios, our design introduces an additional time of 18% when the temporal window of the address fuser is set to 10 ns compared to the conventional HTR scheme. This trade-off results in nearly double the throughput in the half-frame, full-frame, and random-event scenarios, as well as in scenarios with a moderate spatiotemporal correlation. Compared to the token-ring design, ours is faster in the half- and full-frame scenarios and also better for single-event transmissions, establishing itself as a clear winner. When benchmarked against the binary arbiter scheme, our design exhibits a seven-fold latency penalty for single-event transmissions when the temporal window is set to 10ns. However, as the number of events increases, our design achieves approximately five times the speed. Thus, provided that the single-event transmission delay of our design meets specific requirements, it stands out as offering the most favourable trade-off among the three.

5 Conclusion

In this chapter, we revisit the initial design motivation and objectives, then conclude the work and results of this thesis, and propose directions for future research.

5.1 Claims

To reconcile the dual objectives of achieving high temporal precision while also delivering robust throughput, we designed a novel asynchronous AER interface based on the HTR architecture that can capture events within a certain spatiotemporal window, using the open-source ACT asynchronous digital design flow. It makes the following advances over the prevalent binary arbiter-tree, token-ring, and HTR:

1. **Striking a balance between high throughput and low temporal distortion**: Our design, when compared to the conventional HTR, achieves roughly double the throughput for multi-event transmissions at the expense of an 18% efficiency penalty in single-event transmission for a temporal window of 10 ns. Furthermore, in comparison with the token-ring scheme, our design not only outperforms in single-event transmissions but also in both half and full-frame scenarios. This advantage firmly establishes its superiority over the token-ring scheme. In contrast to the binary arbiter scheme, our design faces a seven-fold latency penalty for single-event transmissions with a temporal window of 10 ns. Yet, when the event count rises, the latency of our design is roughly one-fifth that of the binary arbiter scheme.

2. **Leveraging the spatiotemporal correlation of events with a customizable spatiotemporal window**: Unlike the traditional row- or column-based AER interfaces, our design captures events within a customizable spatiotemporal window to reduce the required number of transmissions. Software simulations on the NMNIST dataset indicate that the customizable spatiotemporal window, specifically a 4×4 window, achieves over 1% better event reduction rate than traditional 28×1 row/column-based windows, even while being 42% smaller in size. To the best of our knowledge, we are the first to integrate the address fuser, which exploits spatiotemporal correlations within a given window, with AER circuits.

In summary, while there remains room for optimization in transistor sizing and the selection of spatiotemporal window dimensions, our design effectively balances throughput enhancement while maintaining temporal precision. This aligns with our aim to combine the benefits of improved throughput without sacrificing high temporal precision.

5.2 Future work

Several areas in our work are identified that could benefit from further exploration:

- The binary arbiter tree scheme and the synchronous row-scanning scheme should be implemented using the same Skywater 130nm process to allow for an unbiased comparison with our design.

- The transistor sizes in our design can be further optimized based on criteria such as area, speed, and balanced configurations. For now, only a simple operating design point was demonstrated.
- The effectiveness of the design can be further evaluated using larger, more complex, and longer-duration datasets. For instance, the dataset in [28] captures challenging nighttime urban driving scenarios, while the dataset from [34] presents high-speed bullet scenes. Using these varied datasets can provide a more comprehensive assessment.
- This design can be applied to other domains requiring point-to-point transmission across multiple nodes rather than just DVS. We have already identified its potential for counting photons of varying energies [35]. In this context, hundreds of sensors asynchronously receive photons and emit different numbers of spikes in different channels based on their energies. Our design may have the potential to be used in photon energy readout by making slight modifications to the address fuser.

Appendix name

A

A.1 HSE and PRS of address fuser

The HSE of the a 2-input address fuser is as follows:

```
HSE: [(r0.r|r1.r)->z+;
     [r0.r->o.d[0]+;r0.a+;
     [] ~r0.r->vr0+],
     [r1.r->o.d[1]+;r1.a+;
     [] ~r1.r->vr1+],
     o.r+;[o.a];o.d[0]-,o.d[1]-;
     vr0-,vr1;z-; o.r-;
     [~o.a&~r0.r&~r1.r];
     r0.a-,r1.a-; ]]
```

The PRS of the a 2-input address fuser is as follows:

```
PRS:
 delay_line<2> dl(_z,_2z);  //Place a delay line of 2ns
                            //between _z and _2z.
 ~_r0_r & ~o2_r & ~o.a & ~_2z & ~vr0-> r0.a+
 _r0_r & _o_r & _2z & _o_a -> r0.a-
 ~_r1_r & ~o2_r & ~o.a & ~_2z & ~vr1-> r1.a+
 _r1_r & _o_r & _2z & _o_a -> r1.a-


 ~_Reset |(~_o_a & ~vr0 & ~vr1) -> _z+
 _Reset & (r0.r | r1.r) & _r0_a & _r1_a -> _z-

 ~Reset & ~_r0_r & ~o2_r & ~o.a & ~_2z & ~vr0 -> o.d[0]+
 Reset |(o2_r & o.a)-> o.d[0]-
 ~Reset & ~_r1_r & ~o2_r & ~o.a & ~_2z & ~vr1 -> o.d[1]+
 Reset |(o2_r & o.a)-> o.d[1]-

(~_vr0|~_r0_a) & (~_vr1|~_r1_a) & ~o.a & ~_2z -> o_r+
 _2z -> o_r-

~Reset & ~r0.r & ~r0.a & ~o2_r & ~o.a & ~_2z -> vr0+
Reset |(o2_r & o.a)->vr0-

~Reset & ~r1.r & ~r1.a & ~o2_r & ~o.a & ~_2z -> vr1+
```

```
Reset |(o2_r & o.a)->vr1-

Reset=>_Reset- // It means that _Reset is the inverted version of Reset.
r0.r => _r0_r-
r1.r => _r1_r-
r0.a => _r0_a-
r1.a => _r1_a-
vr0 => _vr0-
vr1 => _vr1-
o.a => _o_a-
o_r => _o_r-
_o_r => o2_r-
```

A.2 Working principle of the used arbiter

This arbiter consists of an input NAND gate pair and a NOR gate pair utilizing transmission gate logic. The input NAND gate pair allows that when both inputs are high, one of '_r' and '_u' is set low, where a low '_r' indicates that 'a' is selected, and a low '_u' indicates that 'b' is selected. As this selection process might take a while, a NOR-gate pair is used as a stabilizer in the output to ensure that the arbiter consistently outputs either a high or low-level voltage.

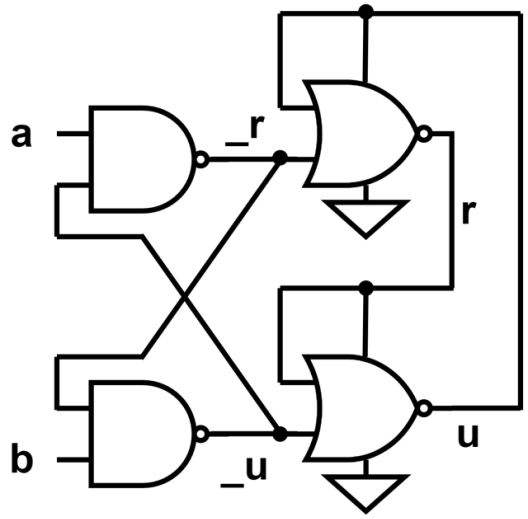


Figure A.1: Schematic of a common arbiter, where 'a' and 'b' are the inputs and they could possibly be both true. '_r' and '_u' represent whether 'a' or 'b' is selected by the arbiter respectively, and there is at most one of them that is low. 'r' and 'u' are the inverted forms of '_r' and '_u' after stabilization, with at most one of them being high.

A.3 HSE of Lserver

The HSE of the Lserver is as follows:

```
HSE: *[[| R.r -> r+; [~R.r]; r-
      []  U.r -> u+; [~U.r]; u-
      |]],

     *[[r ->[w -> skip   // (w,v): dual-rail encoding of b.
            []v -> d.r+;[d.a];w+;d.r-;[~d.a]];
              v-;s.d[n]=r.d[n],s.r+;[s.a];
              R.a+;[~r];s.r-;[~s,a];R.a-
      []u -> [w -> c.r+;[c.a];U.a+;[~u];
            []v -> d.r+;[d.a];U.a+;[~u];d.r-;[~d.a];c.r+;[c.a]
             ];w-,v+;U.a-;c.r-;[~c.a]
      ]]
```

A.4 HSE of Hserver

The HSE of the Hserver is as follows:

```
HSE :
  *[[| u.r -> a+; [~u.r]; a-
   [] uh.r -> c+; [~uh.r]; c-
      |]],
  *[[a ->
  [ ~b1 & ~b2 -> u.a+;[~a];    // (b2,b1): bundled-data encoding of b.
  []b1 & ~b2 -> d.r+; [d.a]; u.a+;[~a];d.r-; [~d.a]
  []~b1 & b2 -> dh.r+;[dh.a];u.a+;[~a];dh.r-;[~dh.a]
  ]; b1+, b2-; u.a-
  []
  c ->
  [ ~b1 & ~b2 -> ch.r+;[ch.a];uh.a+;[~c];
  []b1 & ~b2 -> d.r+; [d.a];uh.a+;[~c];d.r-;[~d.a];ch.r+;[ch.a];
  []~b1 & b2 -> dh.r+;[dh.a];uh.a+;[~c];dh.r-;[~dh.a];ch.r+;[ch.a];
  ]; b1-, b2+;uh.a-;ch.r-;[~ch.a]
  ]]
```

A.5 Sizing in the ACT flow

Sizing in ACT requires manual definition by the designer, rather than automatic optimization by EDA tools as in synchronous circuits. The manual definition method involves specifying P/N ratios and driving strength. For example:

```
PRS of a NOR gate:
                          ~a<20> | ~b<20> -> c+
                           a<20> & b<20> -> c-
```

Here the width of the NMOS and PMOS of variables a and b are manually specified. But specifying the width of all the transistors one by one is simply unrealistic for large-scale design. Therefore, the sizing in ACT is usually done by specifying the driving strength of the output variable. For example, the above PRS will be replaced with:

PRS of a NOR gate:

```
 a & b -> c-
~a | ~b -> c+
sizing{c{-1}}
```

The ***sizing{}*** here is used to specify the sizing (i.e. driving strength) of the output. The negative sign in ***-1*** refers to the pull-down network, and ***1*** refers to a driving strength of 1. In this way, ***sizing{c{-1}}*** commands the tool to adjust the sizes of the PMOS and NMOS transistors inside the PRS of ***c*** to achieve a driving strength of 1 unit.

Bibliography

[1] UZH Robotics and Perception Group. Event-based, 6-dof pose tracking for high-speed maneuvers using a dynamic vision sensor, 2014. Accessed: 2023-08-06.

[2] Prafull Purohit and Rajit Manohar. Field-programmable encoding for address-event representation. *Frontiers in Neuroscience*, 16:1018166, 2022.

[3] Jan Steiner, Kire Micev, Asude Aydin, Jörg Rieckermann, and Tobi Delbruck. Measuring diameters and velocities of artificial raindrops with a neuromorphic dynamic vision sensor disdrometer. *arXiv preprint arXiv:2211.09893*, 2022.

[4] Xu-Guang Guan, Xing-Yuan Tong, and Yintang Yang. Quasi delay-insensitive high speed two-phase protocol asynchronous wrapper for network on chips. *J. Comput. Sci. Technol.*, 25(5):1092–1100, 2010.

[5] Yunjae Suh, Seungnam Choi, Masamichi Ito, Jeongseok Kim, Youngho Lee, Jongseok Seo, Heejae Jung, Dong-Hee Yeo, Seol Namgung, Jongwoo Bong, et al. A 1280× 960 dynamic vision sensor with a 4.95-μm pixel pitch and motion artifact minimization. In *2020 IEEE international symposium on circuits and systems (ISCAS)*, pages 1–5. IEEE, 2020.

[6] Kwabena A Boahen. Point-to-point connectivity between neuromorphic chips using address events. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(5):416–434, 2000.

[7] Nabil Imam and Rajit Manohar. Address-event communication using token-ring mutual exclusion. In *2011 17th IEEE International Symposium on Asynchronous Circuits and Systems*, pages 99–108. IEEE, 2011.

[8] Prafull Purohit and Rajit Manohar. Hierarchical token rings for address-event encoding. In *2021 27th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC)*, pages 9–16. IEEE, 2021.

[9] Guang Chen, Hu Cao, Jorg Conradt, Huajin Tang, Florian Rohrbein, and Alois Knoll. Event-based neuromorphic vision for autonomous driving: A paradigm shift for bio-inspired visual sensing and perception. *IEEE Signal Processing Magazine*, 37(4):34–49, 2020.

[10] Yulia Sandamirskaya, Mohsen Kaboli, Jorg Conradt, and Tansu Celikel. Neuromorphic computing hardware and neural architectures for robotics. *Science Robotics*, 7(67):eabl8419, 2022.

[11] Patrick Lichtsteiner, Christoph Posch, and Tobi Delbruck. A 128 $\times$128120db15μ s latency asynchronous temporal contrast vision sensor. *IEEE journal of solid-state circuits*, 43(2):566–576, 2008.

[12] Guillermo Gallego, Tobi Delbrück, Garrick Orchard, Chiara Bartolozzi, Brian Taba, Andrea Censi, Stefan Leutenegger, Andrew J Davison, Jörg Conradt, Kostas Daniilidis,

et al. Event-based vision: A survey. *IEEE transactions on pattern analysis and machine intelligence*, 44(1):154–180, 2020.

[13] Massimo Antonio Sivilotti. *Wiring considerations in analog VLSI systems, with application to field-programmable networks*. California Institute of Technology, 1991.

[14] Eric Ryu. Industrial DVS Design; Key Features and Applications. Online, 2019. Available: https://rpg.ifi.uzh.ch/docs/CVPR19workshop/CVPRW19_Eric_Ryu_Samsung.pdf.

[15] Thomas Finateu, Atsumi Niwa, Daniel Matolin, Koya Tsuchimoto, Andrea Mascheroni, Etienne Reynaud, Pooria Mostafalu, Frederick Brady, Ludovic Chotard, Florian LeGoff, et al. 5.10 a 1280× 720 back-illuminated stacked temporal contrast event-based vision sensor with 4.86 μm pixels, 1.066 geps readout, programmable event-rate controller and compressive data-formatting pipeline. In *2020 IEEE International Solid-State Circuits Conference-(ISSCC)*, pages 112–114. IEEE, 2020.

[16] Kai Golibrzuch, Sven Schwabe, Tianli Zhong, Kim Papendorf, and Alec M Wodtke. Application of an event-based camera for real-time velocity resolved kinetics. *The Journal of Physical Chemistry A*, 126(13):2142–2148, 2022.

[17] Henri Rebecq, René Ranftl, Vladlen Koltun, and Davide Scaramuzza. High speed and high dynamic range video with an event camera. *IEEE transactions on pattern analysis and machine intelligence*, 43(6):1964–1980, 2019.

[18] Charles Antony Richard Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[19] Alain J Martin. The limitations to delay-insensitivity in asynchronous circuits. In *Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 302–311. Springer, 1990.

[20] Alain J Martin. Compiling communicating processes into delay-insensitive vlsi circuits. *Distributed computing*, 1(4):226–234, 1986.

[21] Rajit Manohar. Production rule synthesis, 2018. Accessed: 21 August 2023.

[22] Broccoli LLC. Bubble reshuffling, 2023. [Online; accessed 12-August-2023].

[23] S Hutchinson, E Keiter, R Hoekstra, H Watts, A Waters, T Russo, R Schells, S Wix, and C Bogdan. The xyce™ parallel electronic simulator–an overview. *Parallel Computing: Advances and Current Issues*, pages 165–172, 2002.

[24] R Timothy Edwards. Google/skywater and the promise of the open pdk. In *Workshop on Open-Source EDA Technology*, 2020.

[25] John K Ousterhout, Gordon T Hamachi, Robert N Mayo, Walter S Scott, and George S Taylor. The magic vlsi layout system. *IEEE Design & Test of Computers*, 2(1):19–30, 1985.

[26] Yihang Yang, Jiayuan He, and Rajit Manohar. Dali: A gridded cell placement flow. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[27] Andrew B Kahng, Lutong Wang, and Bangqi Xu. Tritonroute: The open-source detailed router. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 40(3):547–559, 2020.

[28] Mathias Gehrig, Willem Aarents, Daniel Gehrig, and Davide Scaramuzza. Dsec: A stereo event camera dataset for driving scenarios. *IEEE Robotics and Automation Letters*, 2021.

[29] Davide Bertozzi, Gabriele Miorandi, Alberto Ghiribaldi, Wayne Burleson, Greg Sadowski, Kshitij Bhardwaj, Weiwei Jiang, and Steven M Nowick. Cost-effective and flexible asynchronous interconnect technology for gals systems. *IEEE Micro*, 41(1):69–81, 2020.

[30] Steven M Nowick and Montek Singh. Asynchronous design—part 1: Overview and recent advances. *IEEE Design & Test*, 32(3):5–18, 2015.

[31] Bongki Son, Yunjae Suh, Sungho Kim, Heejae Jung, Jun-Seok Kim, Changwoo Shin, Keunju Park, Kyoobin Lee, Jinman Park, Jooyeon Woo, et al. 4.1 a 640× 480 dynamic vision sensor with a 9μm pixel and 300meps address-event representation. In *2017 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 66–67. IEEE, 2017.

[32] Ned Bingham and Rajit Manohar. Qdi constant-time counters. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 27(1):83–91, 2018.

[33] Garrick Orchard, Ajinkya Jayawant, Gregory K Cohen, and Nitish Thakor. Converting static image datasets to spiking neuromorphic datasets using saccades. *Frontiers in neuroscience*, 9:437, 2015.

[34] Henri Rebecq, René Ranftl, Vladlen Koltun, and Davide Scaramuzza. High speed and high dynamic range video with an event camera. *IEEE transactions on pattern analysis and machine intelligence*, 43(6):1964–1980, 2019.

[35] Stefan J van der Sar, Stefan E Brunner, and Dennis R Schaart. Silicon photomultiplier-based scintillation detectors for photon-counting ct: A feasibility study. *Medical physics*, 48(10):6324–6338, 2021.