ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Dipartimento di Informatica - Scienza e Ingegneria

Master Degree in Computer Engineering

# Efficient Trace for RISC-V: Design, Evaluation, and Integration on a CVA6 multicore design

**Supervisor:**
**Prof.**
**Andrea Bartolini**

**Co-supervisors:**
**Simone Manoni**
**Dr.**
**Emanuele Parisi**

**Presented by:**
**Umberto Laghi**

---

December 2024 session
Academic Year 2023/2024

# Abstract

In modern CPUs, understanding program behavior is essential for effective code debugging and performance analysis. However, commonly used techniques are often invasive and may alter program execution by introducing stalls and exceptions. To address this, CPU vendors have developed less intrusive methods for code monitoring, known as *tracing*.

Tracing encompasses a set of techniques for capturing various types of runtime information from the system. Among these, *Branch Tracing* specifically analyzes the individual instructions executed by the CPU, focusing only on reporting code discontinuities. Each branch tracing standard is tailored to its target Instruction Set Architecture (ISA).

Within the open-source RISC-V standard, a non-ISA-specific branch tracing specification has been developed and ratified, called the *Efficient Trace Specification*. This specification outlines the design rules and packet formats required for implementing Branch Tracing on a RISC-V core.

The contributions of this thesis are i) The design and testing of a Trace Encoder compliant with the Efficient Trace Specification. ii) The integration of this Trace Encoder into a state-of-the-art RISC-V System-on-Chip emulated on FPGA featuring a dual-core host domain. iii) An evaluation of the area overhead and performance introduced by the Trace Encoder.

The implemented Trace Encoder introduces an area overhead of 4.15% relative to the Ariane core and achieves an average compression rate of 95.07% on relevant platform benchmarks.

# Contents

**1 Introduction** **1**
1.1 Branch Tracing . . . 3
1.1.1 What is a block . . . 3
1.1.2 Software Decoder . . . 4
1.2 Tracing techniques . . . 4
1.2.1 Instruction Tracing . . . 5
1.2.2 Data Tracing . . . 5
1.2.3 Instrumentation Tracing . . . 5
1.2.4 System Tracing . . . 5

**2 Trace Encoder** **7**
2.1 Inputs . . . 7
2.2 Packets . . . 8
2.2.1 Format 3 packets . . . 8
2.2.2 Format 2 packet . . . 11
2.2.3 Format 1 packet . . . 11
2.2.4 Format 0 packet . . . 14
2.3 Design . . . 18
2.3.1 te_reg . . . 19
2.3.2 te_branch_map . . . 20
2.3.3 te_filter . . . 21
2.3.4 te_priority . . . 23
2.3.5 te_packet_emitter . . . 27
2.3.6 te_resync_counter . . . 28
2.4 Multiple retirement support . . . 29
2.4.1 Multiple retirement branches only . . . 30
2.4.2 Multiple retirement not only branches . . . 31

**3 CVA6 Trace Encoder Connector** **33**
3.1 CVA6 core . . . 35

3.2 Design . . . . . . . . . . 36
3.2.1 itype_detector . . . . . . . . . . 36
3.2.2 Serialization . . . . . . . . . . 37
3.2.3 Finite State Machine . . . . . . . . . . 38
3.2.4 Deserialization . . . . . . . . . . 39

**4 Evaluation** **41**
4.1 Testing platform . . . . . . . . . . 41
4.2 Timing . . . . . . . . . . 42
4.3 Area utilization . . . . . . . . . . 43
4.3.1 Ariane core . . . . . . . . . . 43
4.3.2 Alsaqr Host Domain . . . . . . . . . . 44
4.4 Benchmarking . . . . . . . . . . 45

# List of Tables


2.1 Format 3 subformat 0 payload . . . . . . . . . . . . . . . . . . . . . . . . 8
2.2 Format 3 subformat 1 payload . . . . . . . . . . . . . . . . . . . . . . . . 9
2.3 Format 3 subformat 2 payload . . . . . . . . . . . . . . . . . . . . . . . . 9
2.4 Format 3 subformat 3 payload . . . . . . . . . . . . . . . . . . . . . . . . 10
2.5 Format 2 payload . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.6 Format 1 payload - address, branch map . . . . . . . . . . . . . . . . . . 13
2.7 Format 1 payload - no address, branch map . . . . . . . . . . . . . . . . 14
2.8 Format 0 subformat 0 payload - no address, branch count . . . . . . . . 15
2.9 Format 0 subformat 0 payload - address, branch count . . . . . . . . . 16
2.10 Format 0 subformat 1 payload - jump target index, branch map . . . 17
2.11 Format 0 subformat 1 payload - jump target index, no branch map . 18

4.1 Info for each test . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

# List of Figures


1.1 Complete system architecture . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Example block and info associated . . . . . . . . . . . . . . . . . . . . 4

2.1 High level trace encoder architecture . . . . . . . . . . . . . . . . . . 19
2.2 te_reg internal architecture . . . . . . . . . . . . . . . . . . . . . . . 20
2.3 te_branch_map internal architecture . . . . . . . . . . . . . . . . . . . 21
2.4 Comparator internal architecture . . . . . . . . . . . . . . . . . . . . 22
2.5 te_filter internal architecture . . . . . . . . . . . . . . . . . . . . . . 23
2.6 Flow chart that determines packet type . . . . . . . . . . . . . . . . . 24
2.7 Logic to delay inputs . . . . . . . . . . . . . . . . . . . . . . . . . . 25
2.8 Address compression logic internal architecture . . . . . . . . . . . . 26
2.9 te_priority internal architecture . . . . . . . . . . . . . . . . . . . . . 27
2.10 te_packet_emitter internal architecture . . . . . . . . . . . . . . . . . 28
2.11 te_resync_counter internal architecture . . . . . . . . . . . . . . . . . 29
2.12 Trace encoder internal architecture for up to N branches . . . . . . . 30
2.13 Trace encoder internal architecture for up to N discontinuities . . . . 31

3.1 Whole system architecture including cva6_te_connector . . . . . . . . 34
3.2 CVA6 internal architecture . . . . . . . . . . . . . . . . . . . . . . . . 35
3.3 cva6_te_connector internal architecture . . . . . . . . . . . . . . . . . 36
3.4 itype_detector associated logic . . . . . . . . . . . . . . . . . . . . . . 37
3.5 Serialization logic . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
3.6 FSM states chart . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
3.7 Deserialization logic . . . . . . . . . . . . . . . . . . . . . . . . . . . 40

4.1 Alsaqr SoC architecture . . . . . . . . . . . . . . . . . . . . . . . . . 42
4.2 Area utilization with respect to an Ariane core . . . . . . . . . . . . . 44
4.3 Area utilization with respect to the Alsaqr Host Domain . . . . . . . 45

# Chapter 1

# Introduction

In modern CPUs, comprehending program behavior presents significant challenges. It is not surprising that software in such environments may occasionally deviate from expected performance and behavior. This unpredictability may depend on various factors, including interactions with other cores, software components, peripherals, real-time events, suboptimal implementations, or a combination of these factors. Performing code analysis knowing the program's flow makes the job easier and this is the reason why code profiling techniques are used.

One of the most common techniques used to monitor the behavior of a live system is using a debugger, but this approach is not always feasible due to its intrusive nature, and common debug methods - such as breakpoints and code instrumentation - often disrupt the program's natural execution flow. When a breakpoint is reached, the core halts and initiates the execution of debug instructions. However, such schemes can introduce significant execution overheads, and if an immediate crash occurs, communication between the core and the debug module may become inoperable.

Moreover, the timing of interrupts and exceptions is inherently asynchronous, making it challenging to predict an exact code location to insert a breakpoint. As a result, less invasive methods are required and one is called *Tracing*. The trace method provides a continuous, non-intrusive observation of program flow without extra code or performance overhead, enabling a finer granularity of instruction-level code analysis and profiling compared to function-level profiling in typical code instrumentation. Processor tracing shortens hardware and software development cycles, reduces bugs, and promotes more reliable processor designs.

Tracing is a general word and refers to a family of techniques and the one analyzed is called *Processor Branch Trace*. This technique involves tracking execution from a predetermined starting address and capturing address changes made by the program.

These address changes, or *deltas*, are generated by jump, call, return, and branch instructions, with interrupts and exceptions also contributing to the deltas.

The system generally comprises the following components:

- A core equipped with an instruction trace interface that outputs essential data for generating a branch trace, operating at high bandwidth and providing detailed data (e.g., instruction address, instruction type, context) per execution clock cycle;
- A hardware encoder that compresses this data into lower bandwidth trace packets;
- A transmission channel or memory unit for sending or storing trace packets;
- A decoder, typically a software module on an external PC, that receives the trace packets and reconstructs the program flow by referencing the binary of the running program. This decoding can occur offline or in real-time.

[PR] p. 5

**Figure 1.1:** Complete system architecture

The RISC-V consortium developed the open-source RISC-V ISA and also connected non-ISA specification, among these is present the *RISC-V Efficient Trace (E-Trace)*. This specification is a standardized processor trace method that defines an efficient compressed branch trace algorithm, input port specifications, and trace output packet formats.

Within RISC-V architecture, instructions execute unconditionally or can be inferred from the program binary, allowing assumptions about the instructions situated between deltas.

## 1.1 Branch Tracing

The branch trace technique fundamentally monitors the *deltas in Program Counter (PC) value* throughout the program execution. The deltas are generated by jumps and they are classified in the following categories:

- *inferable discontinuity*, it is possible to determine the address to which the PC jumps to by only knowing the opcode and the PC value. An example is the JAL instruction;
- *uninferable discontinuity*, it is not possible to determine the address to which the PC jumps to by only knowing the opcode and the PC value. An example is the JALR instruction;
- *branch*, this category of instructions is regarded as distinct from other types of discontinuities, as they are conditional in nature, allowing for the possibility that a discontinuity may or may not occur.

[PR] pp. 11–16

Branch tracing is done by splitting the code in *blocks* depending on the discontinuities and branches that happens during execution and than they are processed by the hardware encoder.

### 1.1.1 What is a block

A block is a sequence of instruction in which:

- The first instruction is a normal instruction (i.e. not a jump) following a special instruction.
- The last instruction is a *special instruction* (i.e. a jump).

An ideal blocks is composed by a sequence of normal instructions followed by a special instruction, but a common corner case occurs when a special instruction follows another special instruction: in this case the block is composed by a single instruction, in which the last instruction overlaps the first one.

Each block is associated with the following mandatory data:

- *itype*, the termination type, so what the last instruction is;
- *cause*, the exception or interrupt cause, considered only in case of an interrupt or exception;
- *tval*, the trap value associated with exception;

- *priv*, privilege level for all instructions retired in the block;
- *iaddr*, the address of the first instruction in the block;
- *iretire*, the length of the instructions retired in the block expressed in halfwords;
- *ilastsize*, the size of the last retired instruction expressed in $2^{ilastsize}$ halfwords

The iaddr, iretire and ilastsize are used to compute the PC value in which the discontinuity happens.[PR] pp. 17–24

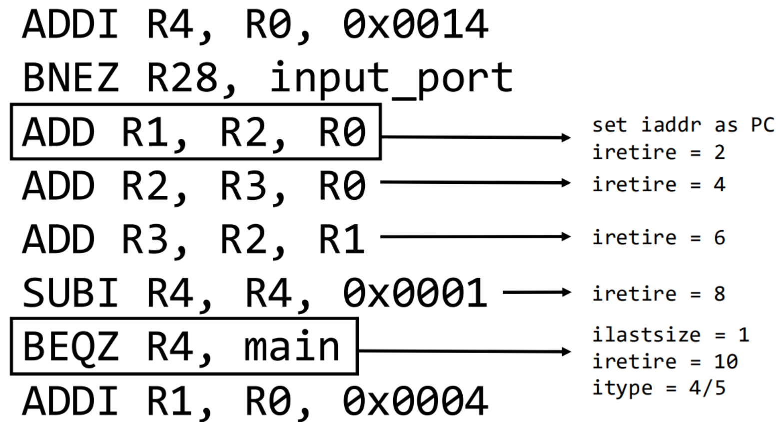


**Figure 1.2:** Example block and info associated

### 1.1.2 Software Decoder

The hardware encoder processes the blocks and produces packets accordingly to the specification. These packets are then sent via transmission channel to a host/debugger (depending if an FPGA or a silicon chip is used) and through decoding it is possible to reconstruct the core execution.

## 1.2 Tracing techniques

The tracing technique explained in the E-Trace specification is the *Branch Trace*, the only one ufficialy supported by the RISC-V architecture so far. Other architecture - like x86 or ARM - may supported other tracing techniques, each one designed for the specific ISA.

To give a greater view on the tracing world, in the following lines are reported some other tracing techniques specific for the ARM ISA.

### 1.2.1 Instruction Tracing

Instruction tracing offers valuable insights into how a core or processor executes instructions. For example, if a core executes a loop ten times with instruction tracing enabled, the resulting decoded trace data will indicate that the loop code was executed on ten separate occasions.

The tracing capabilities of your target may allow for the gathering of additional information, such as cycle counter values or timestamps, in conjunction with the trace data. You can choose the supplementary data that best meets your tracing requirements.

### 1.2.2 Data Tracing

Data trace provides insights into the data access activities of a core or processor. For instance, when a memory load instruction is executed with data trace activated, it records the load instruction along with the corresponding load address and value.

### 1.2.3 Instrumentation Tracing

Instrumentation trace provides outputs related to Operating System (OS) and application events, as well as system information. For instance, when an event takes place during the execution of an application with instrumentation trace activated, the environment transmits valuable runtime data to the instrumentation trace source for subsequent analysis.

The versatility of instrumentation trace is notable; however, its effectiveness is contingent upon the manner in which it is integrated into the intended design.

### 1.2.4 System Tracing

System trace provides information regarding various components within the system. For instance, the system trace facilitates the generation of events for both target hardware and software. The functionality of system trace components encompasses a broader range than that of instrumentation trace components, indicating that there are numerous commonalities between the two. While system trace is adaptable, its effectiveness is contingent upon its implementation within the specific design.[ARM24]

subsectionContributions This thesis introduces the following contributions:

- The development of a RISC-V compliant Efficient Trace Encoder (TE).
- The development of the hardware module required to extract the signals for CPU-TE communication. The open-source CVA6 core has been adopted as

the CPU target for tracing.

- The integration of the tracing solution on a state-of-the-art RISC-V SoC based on a dual-core CVA6 host domain.
- The performance evaluation of the hardware designed on several platform-relevant tests. The whole trace encoder system has an area overhead of 4.15% relative to the core itself and the resulting compression rate achieves an average of 95.07% w.r.t. tracing all the single instructions committed by the core.

# Chapter 2

# Trace Encoder

Like the RISC-V ISA, the E-Trace specification does not provide any guidelines regarding its architecture. The specification contains only the formal procedures. The few things that are defined are:

- Inputs
- Packet payload structure
- Which packet to output based on inputs
- Outputs

The rest is up to the designer to choose.

## 2.1 Inputs

The mandatory inputs the *Trace Encoder* takes in are the following:

- *itype*
- *cause*
- *tval*
- *priv*
- *iaddr*
- *iretire*
- *ilastsize*

In addition to these inputs, other optional ones are described for various operational modes or to provide information.

## 2.2 Packets

To convey information to the decoder and reduce the bandwidth used, the specification defines different type of packets, each one to use in different situations.

### 2.2.1 Format 3 packets

This family of packets is mainly used to convey supporting information for the decoder, such as synchronisation, traps, reporting context and other supporting information.

#### Subformat 0 - Synchronisation

This subformat is used to communicate the decoder two things:

- The first instruction has been traced;
- The resynchronisation timer has expired.

The payload is organized as follows:

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 11 (sync): synchronisation. |
| subformat | 2 | 00 (start): start of tracing or resync. |
| branch | 1 | Set to 0 if the address points to a branch instruction, and the branch was taken. Set to 1 if the instruction is not a branch or if the branch is not taken. |
| privilege | 2 | The privilege level of the reported instruction. |
| time | XLEN | The time value. |
| context | TBD | The instruction context. |
| address | XLEN | Full instruction address. |

**Table 2.1:** Format 3 subformat 0 payload

#### Subformat 1 - Trap

This subformat is used to communicate to the decoder the cause and trap value associated to an interrupt or exception and it is sent following them.

The payload is organized as follows:

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 11 (sync): synchronisation. |
| subformat | 2 | 01 (trap): Exception or interrupt cause and trap handler address. |
| branch | 1 | Set to 0 if the address points to a branch instruction, and the branch was taken. Set to 1 if the instruction is not a branch or if the branch is not taken. |
| privilege | 2 | The privilege level of the reported instruction. |
| time | XLEN | The time value. |
| context | TBD | The instruction context. |
| ecause | XLEN | Exception or interrupt cause. |
| interrupt | 1 | Interrupt. |
| thaddr | 1 | When set to 1, address points to the trap handler address. When set to 0, address points to the EPC for an exception at the target of an updiscon, and is undefined for other exceptions and interrupts. |
| address | XLEN | Full instruction address. |
| tval | XLEN | Value from appropriate utval/stval/vstval/mtval CSR. Field omitted for interrupts |

**Table 2.2:** Format 3 subformat 1 payload

### Subformat 2 - Context

This subformat is used to communicate the context and/or the timestamp, it is output when the context value changes.

Since both context and time are optional inputs, this packet is not emitted if context is not implemented.

The payload is organized as follows:

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 11 (sync): synchronisation. |
| subformat | 2 | 10 (context): Context change. |
| privilege | 2 | The privilege level of the new context. |
| time | XLEN | The time value. |
| context | TBD | The instruction context. |

**Table 2.3:** Format 3 subformat 2 payload

#### Subformat 3 - Support

This subformat is used to give supporting information to the decoder and it is sent when:

- Trace is enabled or disabled;
- The operating mode changes;
- One or more packets cannot be sent due to back-pressure from the communication channel infrastructure.

The payload is organized as follows:

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 11 (sync): synchronisation. |
| subformat | 2 | 11 (support): Supporting information for the decoder. |
| ienable | 1 | Indicates if the instruction trace encoder is enabled. |
| encoder_mode | 1 | Identifies trace algorithm Details and number of bits implementation dependent. Currently Branch trace is the only mode defined, indicated by the value 0. |
| qual_status | 2 | Indicates qualification status (no_change): No change to filter qualification (ended_rep): Qualification ended, preceding te_inst sent explicitly to indicate last qualification instruction (trace_lost): One or more instruction trace packets lost. (ended_ntr): Qualification ended, preceding te_inst would have been sent anyway due to an updiscon, even if it wasn't the last qualified instruction |
| ioptions | 3 | Values of all instruction trace run-time configuration bits. |
| denable | 1 | Indicates if the data trace is enabled (if supported). |
| dloss | 1 | One of more data trace packets lost (if supported). |
| doptions | TBD | Values of all data trace run-time configuration bits Number of bits and definitions implementation dependent. Examples might be - 'no data' Exclude data (just report addresses) - 'no addr' Exclude address (just report data) |

**Table 2.4:** Format 3 subformat 3 payload

### 2.2.2 Format 2 packet

This packet contains only an instruction and is output when the address of an instruction must be reported and there is no unreported branch information. By default it is used the differential address, that is computed as the difference between the address to send and last address sent in a packet. To send full address the full address mode must be enabled.

The payload is organized as follows:

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 10 (addr-only): differential address and no branch information. |
| notify | 1 | Differential instruction address. |
| updiscon | 1 | If the value of this bit is different from the MSB of address, it indicates that this packet is reporting an instruction that is not the target of an uninferable discontinuity because a notification was requested via trigger unit. |
| irreport | 1 | If the value of this bit is different from updiscon, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count. |
| irdepth | TBD | If the value of irreport is different from updiscon, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If irreport is the same value as updiscon, all bits in this field will also be the same value as updiscon. |

**Table 2.5:** Format 2 payload

### 2.2.3 Format 1 packet

This format is used to communicate to the encoder the branch information, and it is output if the branch information must be reported or when the address of an instruction must be reported and there has been at least one branch since the previous packet.

The payload for this specific format can be different and it depends on which there

is unreported branch or not.

The payload with branch map and address is organized as follows:

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 01 (diff-delta): includes branch information and may include differential address. |
| branches | 5 | Number of valid bits branch_map. The number of bits of branch_map is determined as follows: (cannot occur for this format) : 1 bit -3: 3 bits -7: 7 bits -15: 15 bits -31: 31 bits For example if branches = 12, branch_map is 15 bits long, and the 12 LSBs are valid. |
| branch_map | 31 | An array of bits indicating whether branches are taken or not. Bit 0 represents the oldest branch instruction executed. For each bit: : branch taken : branch not taken. |
| address | XLEN | Differential instruction address. |
| notify | 1 | If the value of this bit is different from the MSB of address, it indicates that this packet is reporting an instruction that is not the target of an uninferable discontinuity because a notification was requested via trigger unit. |
| updiscon | 1 | If the value of this bit is different from the MSB of notify, it indicates that this packet is reporting the instruction following an uninferable discontinuity and is also the instruction before an exception, privilege change or resync. |
| irreport | 1 | If the value of this bit is different from updiscon, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count. |
| irdepth | TBD | If the value of irreport is different from updiscon, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If irreport is the same value as updiscon, all bits in this field will also be the same value as updiscon. |

**Table 2.6:** Format 1 payload - address, branch map

The payload without branch map and address is organized as follows:

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 01 (diff-delta): includes branch information and may include differential address |
| branches | 5 | Number of valid bits in branch_map. The length of branch_map is determined as follows: : 31 bits, no address in packet -31: (cannot occur for this format) |
| branch_map | 31 | An array of bits indicating whether branches are taken or not. Bit 0 represents the oldest branch instruction executed. For each bit: : branch taken : branch not taken |

**Table 2.7:** Format 1 payload - no address, branch map

### 2.2.4 Format 0 packet

This format is used to communicate information in case of optional efficiency extensions. In version 2.0.3 only two extensions are defined:

- Reporting count of correctly predicted branches;
- Reporting jump target cache index.

None of the previous extensions is supported in this implementation, but for the sake of completeness the payload associated are reported.

#### Subformat 0

If branch prediction mode is supported and enabled, then there is a choice of whether to output a full branch map (via format 1), or a count of correctly predicted branches. The count format is used if the number of correctly predicted branches is at least 31. If there are 31 unreported branches (i.e. the branch map is full), but not all of them were predicted correctly, then the branch map will be output.

| Field name | Bits | Description |
| --- | --- | --- |
| format | 2 | 00 (opt-ext): formats for optional efficiency extensions. |
| subformat | 1 | 0 (correctly predicted branches). |
| branch_count | 32 | Count of the number of correctly predicted branches, minus 31. |
| branch_fmt | 2 | 00 (no-addr): Packet does not contain an address, and the branch following the last correct prediction failed. -11: (cannot occur for this format) |

**Table 2.8:** Format 0 subformat 0 payload - no address, branch count

| Field name | Bits | Description |
| --- | --- | --- |
| format | 2 | 00 (opt-ext): formats for optional efficiency extensions. |
| subformat | 1 | 0 (correctly predicted branches). |
| branch_count | 32 | Count of the number of correctly predicted branches, minus 31. |
| branch_fmt | 2 | 10 (addr): Packet contains an address. If this points to a branch instruction, then the branch was predicted correctly. (addr-fail): Packet contains an address that points to a branch which failed the prediction. ,01: (cannot occur for this format) |
| address | XLEN | Differential instruction address. |
| notify | 1 | If the value of this bit is different from the MSB of address, it indicates that this packet is reporting an instruction that is not the target of an uninferable discontinuity because a notification was requested via trigger unit. |
| updiscon | 1 | If the value of this bit is different from notify, it indicates that this packet is reporting the instruction following an uninferable discontinuity and is also the instruction before an exception, privilege change or resync. |
| irreport | 1 | If the value of this bit is different from updiscon, it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count. |
| irdepth | TBD | If the value of irreport is different from updiscon, this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If irreport is the same value as updiscon, all bits in this field will also be the same value as updiscon. |

**Table 2.9:** Format 0 subformat 0 payload - address, branch count

#### Subformat 1

If a jump target cache mode is supported and enabled, and the address to report following an updiscon is in the cache then the encoder can output the cache index using format 0, subformat 1. However, the encoder may still choose to output the differential address using format 1 or 2 if the resulting packet is shorter. This may occur if the differential address is zero, or very small.

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 00 (opt-ext): formats for optional efficiency extensions. |
| subformat | 1 | 1 (jump target cache). |
| index | TBD | Jump target cache index of entry containing target address. |
| branches | 5 | Number of valid bits in branch_map. The length of branch_map is determined as follows: : (cannot occur for this format) : 1 bit -3: 3 bits -7: 7 bits -15: 15 bits -31: 31 bits. For example if branches = 12, branch_map is 15 bits long, and the 12 LSBs are valid. |
| branch_map | 31 | An array of bits indicating whether branches are taken or not. Bit 0 represents the oldest branch instruction executed. For each bit: : branch taken : branch not taken |
| irreport | 1 | If the value of this bit is different from branch_map[MSB], it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count. |
| irdepth | TBD | If the value of irreport is different from branch_map[MSB], this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If irreport is the same value as branch_map[MSB], all bits in this field will also be the same value as branch_map[MSB]. |

**Table 2.10:** Format 0 subformat 1 payload - jump target index, branch map

| Field name | Bits | Description |
|---|---|---|
| format | 2 | 00 (opt-ext): formats for optional efficiency extensions. |
| subformat | 1 | 1 (jump target cache). |
| index | TBD | Jump target cache index of entry containing target address. |
| branches | 5 | Number of valid bits in branch_map. The length of branch_map is determined as follows: : (cannot occur for this format) : 1 bit -3: 3 bits -7: 7 bits -15: 15 bits -31: 31 bits. For example if branches = 12, branch_map is 15 bits long, and the 12 LSBs are valid. |
| irreport | 1 | If the value of this bit is different from branch_map[MSB], it indicates that this packet is reporting an instruction that is either: following a return because its address differs from the predicted return address at the top of the implicit_return return address stack, or the last retired before an exception, interrupt, privilege change or resync because it is necessary to report the current address stack depth or nested call count. |
| irdepth | TBD | If the value of irreport is different from branch_map[MSB], this field indicates the number of entries on the return address stack (i.e. the entry number of the return that failed) or nested call count. If irreport is the same value as branch_map[MSB], all bits in this field will also be the same value as branch_map[MSB]. |

**Table 2.11:** Format 0 subformat 1 payload - jump target index, no branch map

[PR] p. 33-43

## 2.3 Design

The modules defined are the following:

- *te_reg*, stores configuration data, produces clock signal for the other modules;
- *te_resync_counter*, counts packets emitted or clock cycles and asks for a resynchronisation packet;
- *te_branch_map*, counts the branches and keeps track if they were taken or not;
- *te_filter*, declares input blocks as "qualified" - they can processed by the next modules - based on user defined values read from te_reg;
- *te_priority*, determines which packet needs to be emitted and performs a simple

compression on the address that is put inside the payload;

- *te_packet_emitter*, populates the payload and can reset both the te_resync_-counter and te_branch_map.

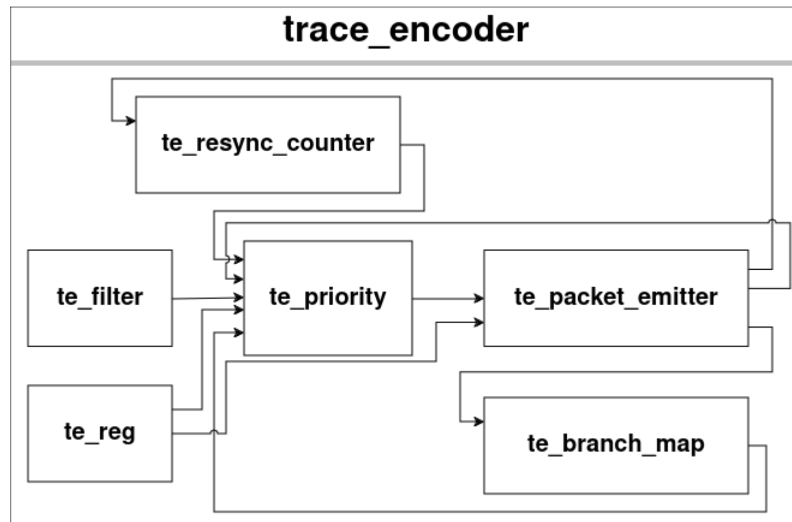


**Figure 2.1:** High level trace encoder architecture

### 2.3.1 te_reg

This module stores the user and non-user definable parameters.

The user definable parameters are the following:

- trace_activated, determines if the encoder is waiting for a first block to start tracing;
- nocontext, determines if the optional context input is used or not;
- notime, determines if the optional time input is used or not;
- encoder_mode, determines if the encoder is doing branch tracing or another type of tracing technique (right now only supported branch trace mode);
- configuration, determines in which of the 7 modes available in the E-Trace specification v2.0.3 the encoder is operating. Right now only delta_address - the address in payload is expressed as difference between the current address and the last one put in a payload - and full_address - the address in payload is the current one - modes are implemented;
- lossless_trace, determines if the TE stalls the hart if there is back-pressure from the transport interface;

- shallow_trace, determines if the branch map content has to be flushed after each packet emitted or only after a packet containing the branch map;
- All the parameters necessary to the te_filter module to perform filtering.

The clock_gated signal depends on the trace_activated signal, it is connected to all the other modules in order to have a different clock domain and to switch off the parts of the encoder when they are not necessary.

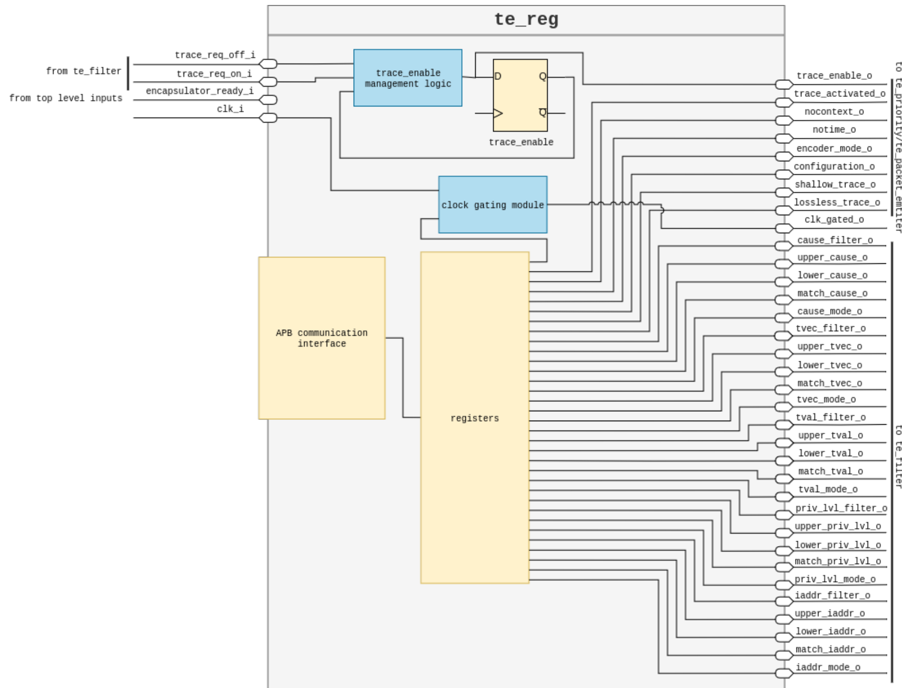


**Figure 2.2:** te_reg internal architecture

### 2.3.2 te_branch_map

The te_branch_map module serves the purpose of keeping track of branches. It does that by using a counter and an array called branch map. The counter counts the received branches and the branch_map saves if the branches are taken or not. According to the E-Trace specification, the counter counts up to 31 branches (5 bits long) and the branch map itself is a 31 bits long array that saves a 1 when the branch is not taken and a 0 when it is.

Whenever the counter reaches the value of 31 - branch map is full - the module asserts the is_full_o signal to request the generation of the associated packet (format 1) and it remains asserted until it receives the flush_i signal from the te_packet_-

emitter module that communicates the associated packet has been generated. This is done to keep the request for a packet active until it is satisfied.

The E-Trace specification does not say anything about how and when to handle the branches. In case of a multiple retirement CPU, the solution that makes more sense is to handle all of them in one cycle, this way in just one cycle the counter and branch map are updated.

The update of both branch counter and branch map is done in a combinatorial network. The first step is to load the number of branches and if they were taken or not left from the previous cycle, then the branches received in this cycle are counted and summed to the ones to serve. After that it is served what is possible and if the branch map is full - branch counter reached value 31 - all the remaining branches are stored to be served the next cycle. The info to keep track if a branch was taken or not are right shifted into an array, this way they will be served first in the following clock cycle.

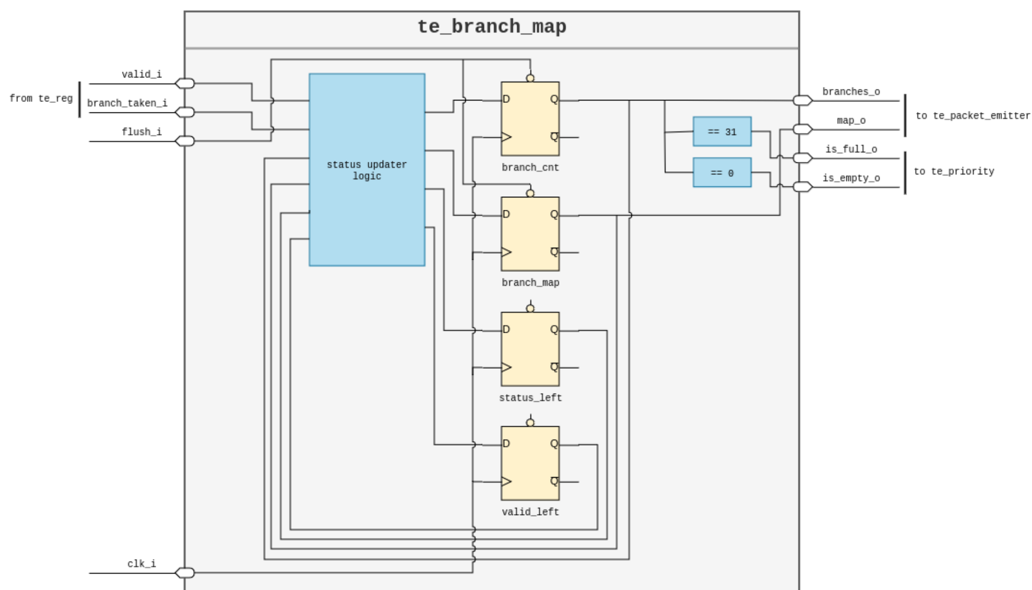


**Figure 2.3:** te_branch_map internal architecture

### 2.3.3 te_filter

The te_filter has the duty to filter input blocks such that it is possible to trace a specific portion of code, or specific discontinuities.

The E-Trace specification suggest an implementation that uses a pair of comparators and they can be arranged to select specific ranges for different parameters to be

traced. Furthermore, up to 3 run time selectable comparators can be used at a time.[PR] p. 29

This implementation is different from the one proposed by the E-Trace specification: it uses a comparator for each input and can be used as many comparators as they are available. This way it is possible to check if the value is in a range or matches with a specific one. This solution allows a better flexibity in terms of filtering.

**Figure 2.4:** Comparator internal architecture

In order to perform filtering it is necessary to enable the filter signal associated to the input port we want to filter and set the operative mode for each filter: range or match.

With the range mode is possible to set as qualified the inputs that are in a certain range. In match mode is possible to check if the inputs coincide with the value set. In case no filter is activated, all inputs are set as qualified.

The values used to filter are taken from the te_reg module.

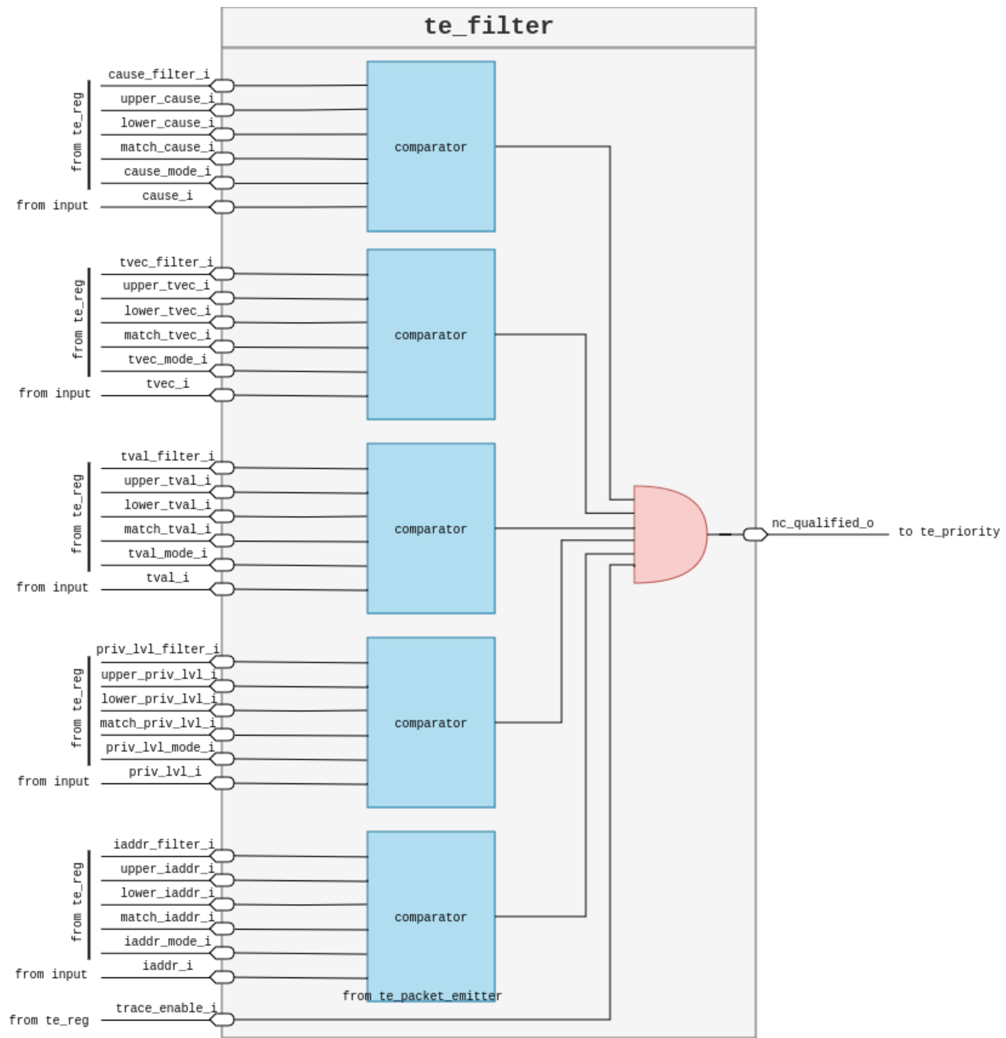


**Figure 2.5:** te_filter internal architecture

### 2.3.4 te_priority

This module implements the flow chart shown in the E-Trace specification as a combinatorial network, this way the packet type to issue is determined in - ideally - one clock cycle.

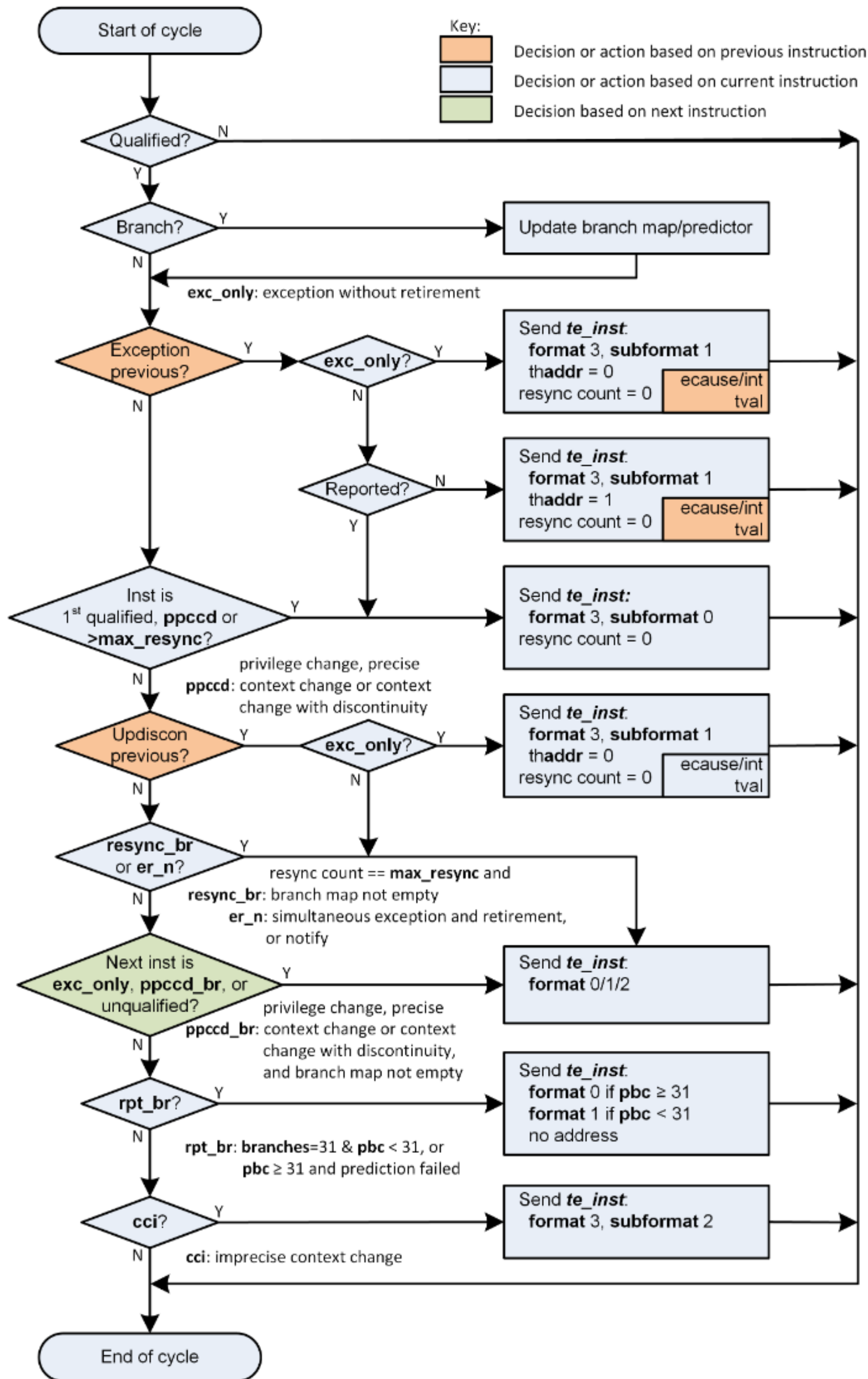


**Figure 2.6:** Flow chart that determines packet type

As the flow chart shows, packets are determined based on the three different states:

- Previous instruction
- Current instruction
- Next instruction

To obtain this three different periods is used a series of flip flop that store information read from inputs and delays it up to 2 cycles.

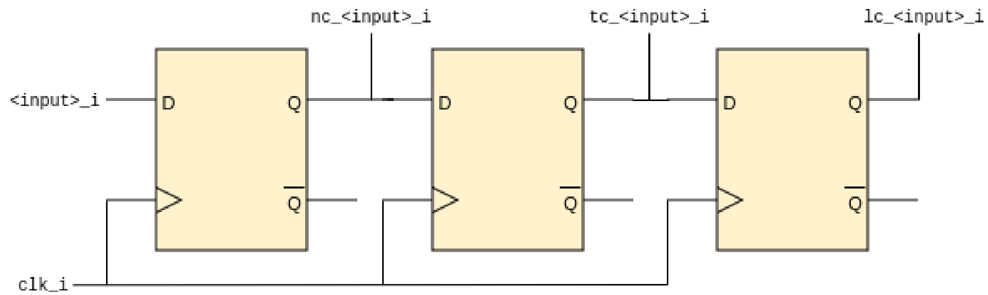


**Figure 2.7:** Logic to delay inputs

The E-Trace specification does not show the totality of the flow chart, since the determination of format 3 subformat 3 packets is just described in a paragraph before the flow chart and without giving any hints on where this part should be inside the flow chart. So, in this implementation it was chosen to put the network that determines if a format 3 subformat 3 packet needs to be issued before all the rest of the network.

This choice was made because the format 3 subformat 3 packets need to be output in the following scenarios: the TE is enabled or disabled, after the final qualified instruction has been traced, packets are lost. To prevent the request for packets that should not be issued, the if clause needs to be checked before the checks for all the other packets.

Another operation performed by the te_priority module is address compression. The E-Trace specification mentions the compression but never gives clear guidelines on how and when to do it. In this implementation was decided to compresse only the address - full or differential - contained in packet payloads. It was decided to compress only the address because it is easier and can reduce packet size.

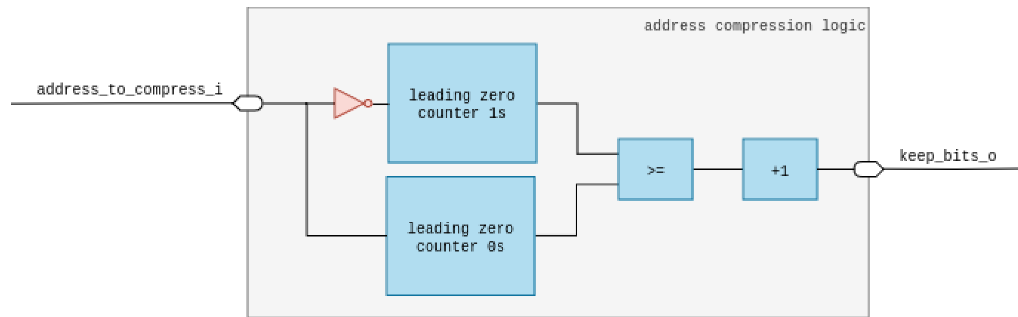


**Figure 2.8:** Address compression logic internal architecture

The idea to compress the address is trivial and consists in removing the most significant bits that are all 0s or 1s except for the last one. This last one bit is necessary to have a lossless compression because the address can be sign extended on the decoder side and the original address can be retrieved without losing any information.

For example, if the address to compress is 0000001100010 the 5 most significant bits can be removed and the address obtained is: 01100010. From this address is possible to retrieve the original one by sign extending the most significant bit - in this case 0 - to the original length.

The intention is to count the most significant 0s and 1s, this operation is performed by two leading zero counters modules: one that counts on the original address and one on the bitwise negated one (1s become 0s and the module can count them).

Than, it is chosen the most effective way to compress - removing 1s or 0s - and this value increased by one - this way last most significant 1 or 0 is kept for sign extension - is sent to the te_packet_emitter module that takes care of putting the right amount of bits of the address in the payload.

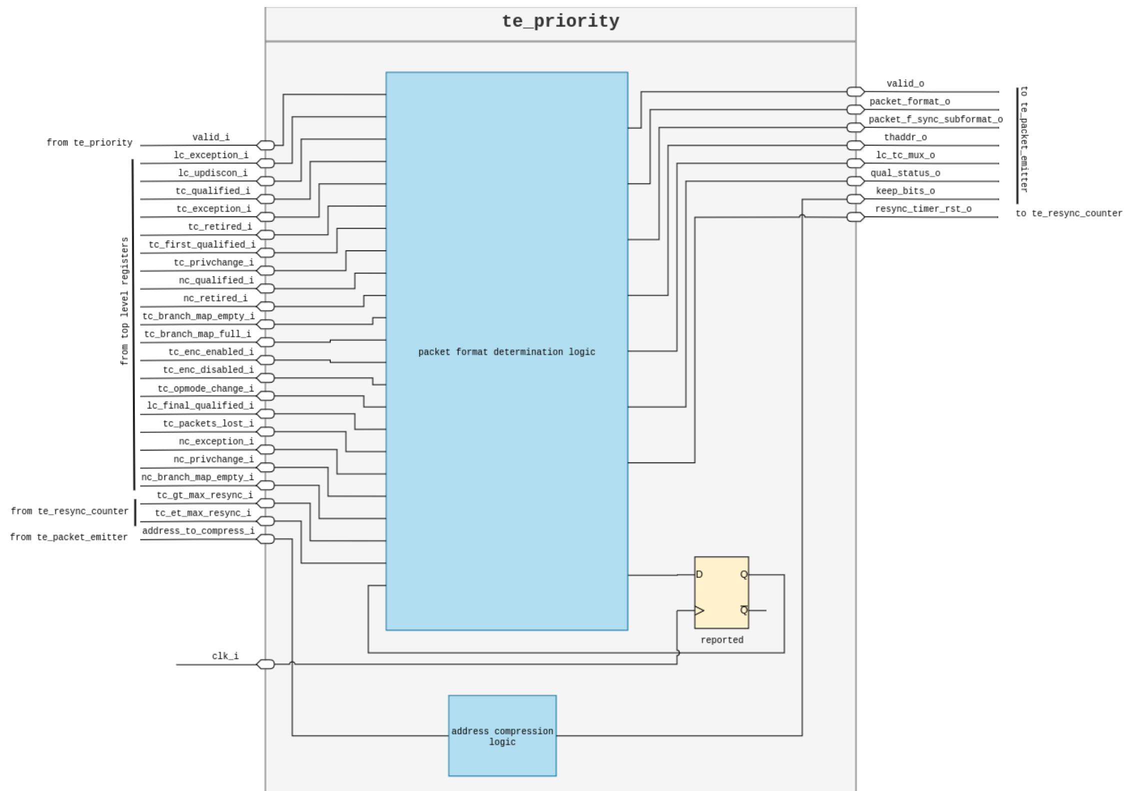


**Figure 2.9:** te_priority internal architecture

### 2.3.5 te_packet_emitter

The te_packet_emitter module takes care of populating the payload accordingly to the inputs received from te_priority - packet format and subformat - and sending them as outputs of the TE itself.

In order to place a compressed address inside the payload, the te_packet_emitter module sets as output the address to compress based on the payload to emit. In the same cycle it receives the number of bits to keep from the te_priority module. But System Verilog doesn't allow a dynamic array population, so a chunking operation is performed and the number of bits to keep is rounded up to the closest multiple of 8. This way is possible to insert the address but using less bits.

Another operation of compression is performed on the branch map. In this case is far simpler, because we already know the number of bits to keep thanks to the branch counter. Once again it is necessary to perform chunking to set the branch map inside the payload.

When the payload is composed, it is output along with its length in bytes and the

packet type. At this point is up to the encapsulator to make this data compliant with the communication protocol chosen.

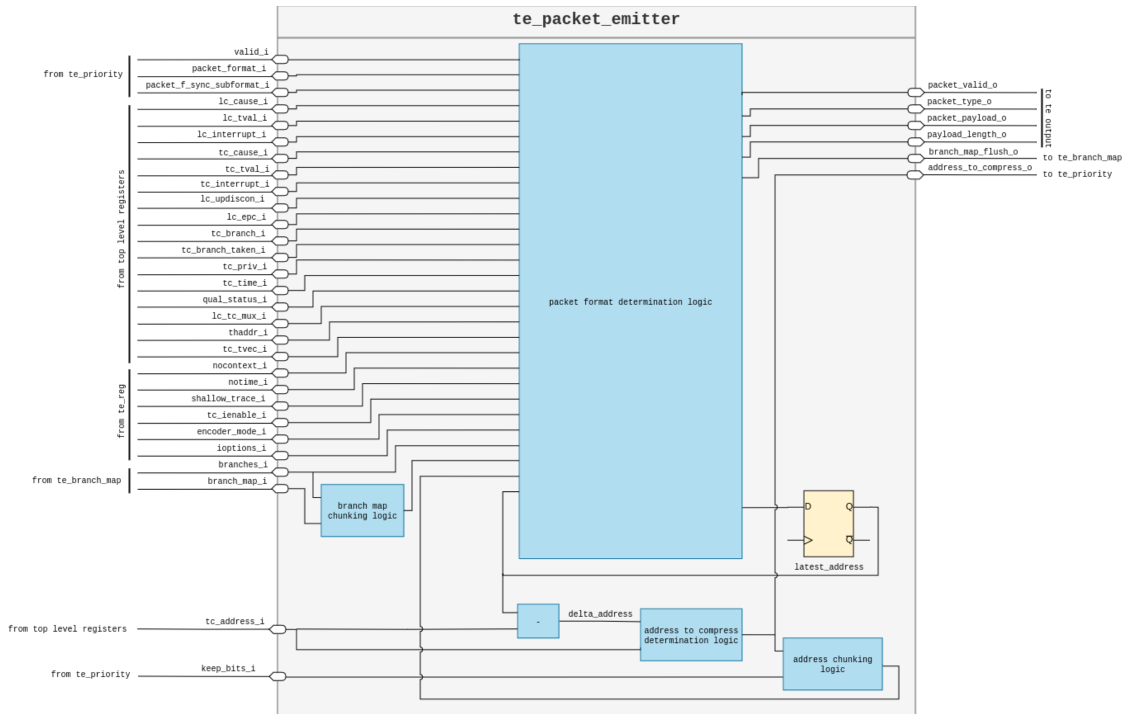


Figure 2.10: te_packet_emitter internal architecture

### 2.3.6 te_resync_counter

This module has the purpose to count the packets emitted or the clock cycles elapsed and communicates to the te_priority to request a resynchronisation packet.

The configurable parameters at instantiation are the following:

- N, the max number of packets emitted in one cycle;
- MODE, determines if the module counts packets emitted or clock cycles;
- MAX_VALUE, determines the value at which it assers the resync_max_o values.

As required by the te_priority flow chart, this module produces two different signals:

- et_resync_max_o when the counter has reached the defined max value;
- gt_resync_max_o when the counter has exceeded the defined max value, it remains asserted until the module receives a synchronous reset signal.

The te_resync_counter is designed to have a parametrizable number of valid ports, this way it can be adapted to scenarios in which are present N te_packet_emitter modules that may output up to N packets per cycle. The update is performed in a combinational network and so in one clock cycle the module status is up to date. To prevent counter overflow, the module is designed such that when it receives a number that would overflow the counter, it stops the count at MAX_VALUE+1 asserting the gt_resync_max_o signal.

The E-Trace specification leave to the designer the choice of what to do in case of a resynchronisation packet request pending and packets/cycles to count in that period. In this implementation when the counter has gt_resync_max_o asserted, all the packets/cycles to count are ignored since they belong to the time period preceding the resynchronisation.

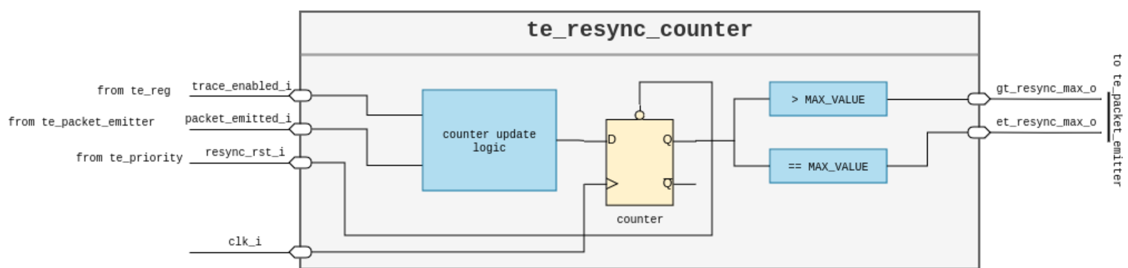


**Figure 2.11:** te_resync_counter internal architecture

## 2.4 Multiple retirement support

Nowadays, most of the RISC-V cores - excluding the embedded and IoT oriented ones - can retire up to N instructions per cycle. This means that the commit stage of the CPU might create up to N blocks to process for the TE per cycle.

To perform a complete and correct tracing it is necessary to keep track of all the commit ports of a CPU and to achieve that the E-Trace specification mentions in the trace interface chapter which inputs need to be replicated.

Since a block is associated to a special instruction, in some CPU where multiple special instructions are retired in a cycle, there's the need to process more blocks in the same cycle. In order to do that, the following inputs are replicated for the number of blocks that can be processed in the same cycle:

- itype
- iaddr
- iretire

- ilastsize

[PR] p. 20

To cope with this two different scenarios, it was decided to define two slightly different architectures for the TE.

### 2.4.1 Multiple retirement branches only

This architecture is designed to cope with the case of a multiple retirement CPU that can output only one uninferable discontinuity or up to N branches. This means the TE can only output up to one packet per clock cycle because branches do not directly require the output of a packet, unlike other discontinuities such as JALR that is considered an uninferable discontinuity.

In this first case the modification consists in the replication of the filter module. This is necessary to perform filtering on all the input ports. The branch map is also adapted to update its status in one cycle and be ready to process other branches in the following cycle.

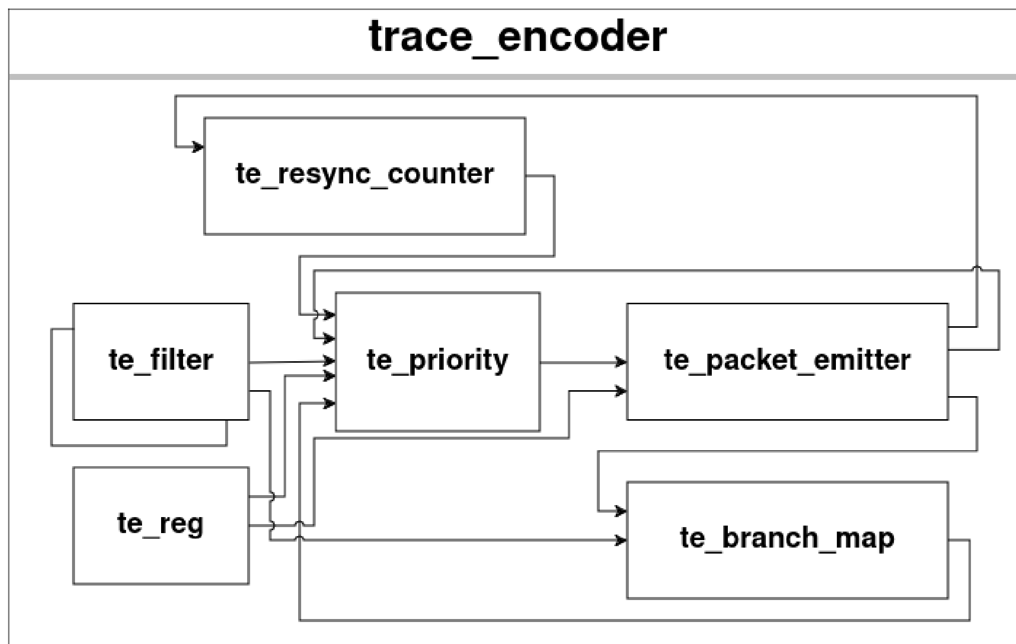


**Figure 2.12:** Trace encoder internal architecture for up to N branches

Other modifications are not necessary because the number of packets to emit is the same as the single retirement case.

### 2.4.2 Multiple retirement not only branches

This architecture is designed to cope with the scenario of a CPU that can output up to N uninferable discontinuities. This means the TE can output up to N packets, since uninferable discontinuities directly require a packet.

The second scenario requires the replication of the modules that determine and assemble the packet itself: te_filter, te_priority, te_packet_emitter. This is necessary because up to N packets can be emitted and so for each packet are necessary the modules that generates the packet.

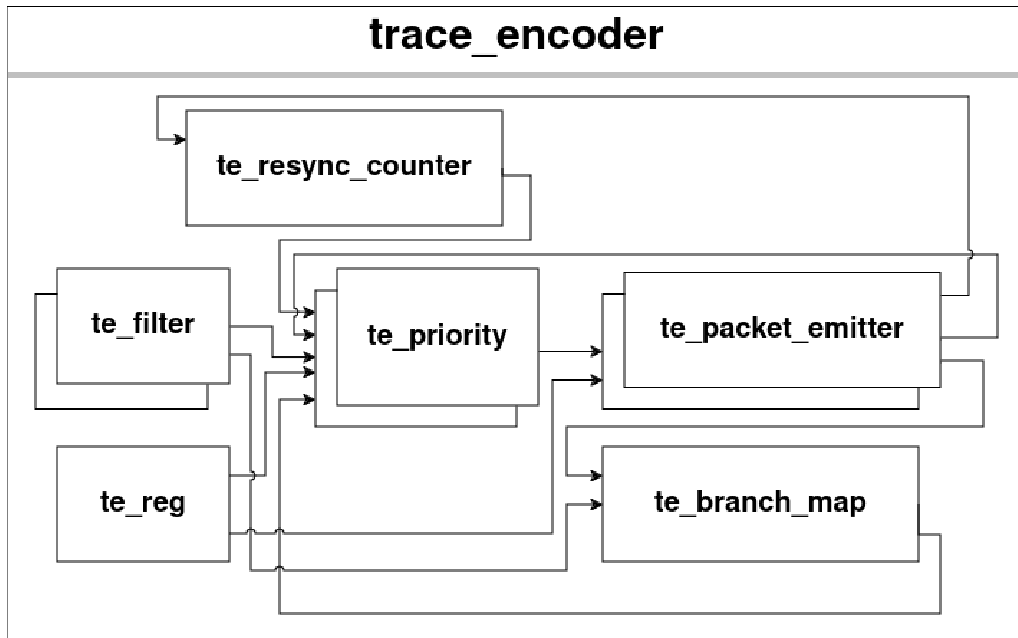


**Figure 2.13:** Trace encoder internal architecture for up to N discontinuities

Since up to N packets can be emitted, the te_resync_counter module needs to be adapted similarly to the te_branch_map in order to update its status in just one cycle.

In this scenario, the te_resync_counter outputs and the branch_map_full_o signal from te_branch_map module are connected to only the first te_priority module. This is necessary to avoid the generation of N identical - resync or format 1 - packets in the same cycle, but just one.

# Chapter 3

# CVA6 Trace Encoder Connector

According to the *E-Trace specification*, the mandatory inputs the *Trace Encoder* takes in are the following:

- *itype*
- *cause*
- *tval*
- *priv*
- *iaddr*
- *iretire*
- *ilastsize*

No RISC-V CPU generates the aforementioned signals, necessitating an *interface module* that converts these CPU signals into a format comprehensible by the TE.

The E-Trace specification does not address this module, leaving its design to the discretion of the designer.

The module that has been implemented is studied specifically for the CVA6 CPU; however, it can be easily adapted for use with the Snitch core. Given it is designed to work as interface for CVA6, the cva6_te_connector can manage N commit ports and process up to N instructions committed per cycle.

This module captures the instructions executed by the core, along with exceptions and interrupts, and generates the necessary inputs for the TE.

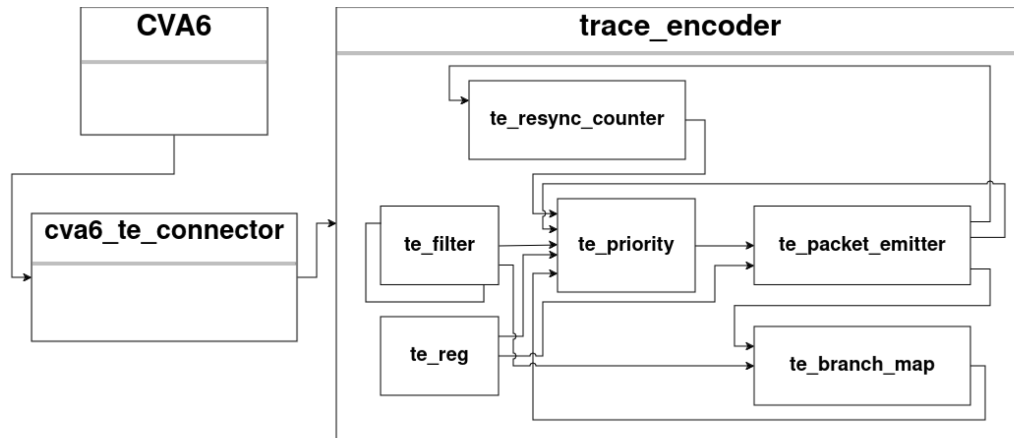


**Figure 3.1:** Whole system architecture including cva6_te_connector

The main idea is to process one instruction at a time and generate the blocks along with their information. This processing is done *by instruction* and not by cycle, because the blocks can be split along multiple cycle and so an FSM is the best solution to tackle this problem.

Since a CPU can retire up to N instructions, this module is structured to be easily parametrizable and adaptable to different CPU configurations.

In order to determine the info for the blocks to output, the following data are required for each commit port:

- If an instruction is committed;
- If the instruction is compressed;
- The PC value of the instruction;
- The operation type of the instruction;
- If the instruction is a branch;
- If the branch is taken or not.

Then, the following common information is required:

- If an exception or interrupt occurs;
- The cause and tval associated with interrupt and exception.

## 3.1 CVA6 core

CVA6 is a 6-stage, single-issue, in-order CPU which implements the 64-bit RISC-V instruction set. It fully implements I, M, A and C extensions as specified in Volume I: User-Level ISA V 2.3 as well as the draft privilege extension 1.10. It implements three privilege levels M, S, U to fully support a Unix-like operating system.

The architecture is the following:

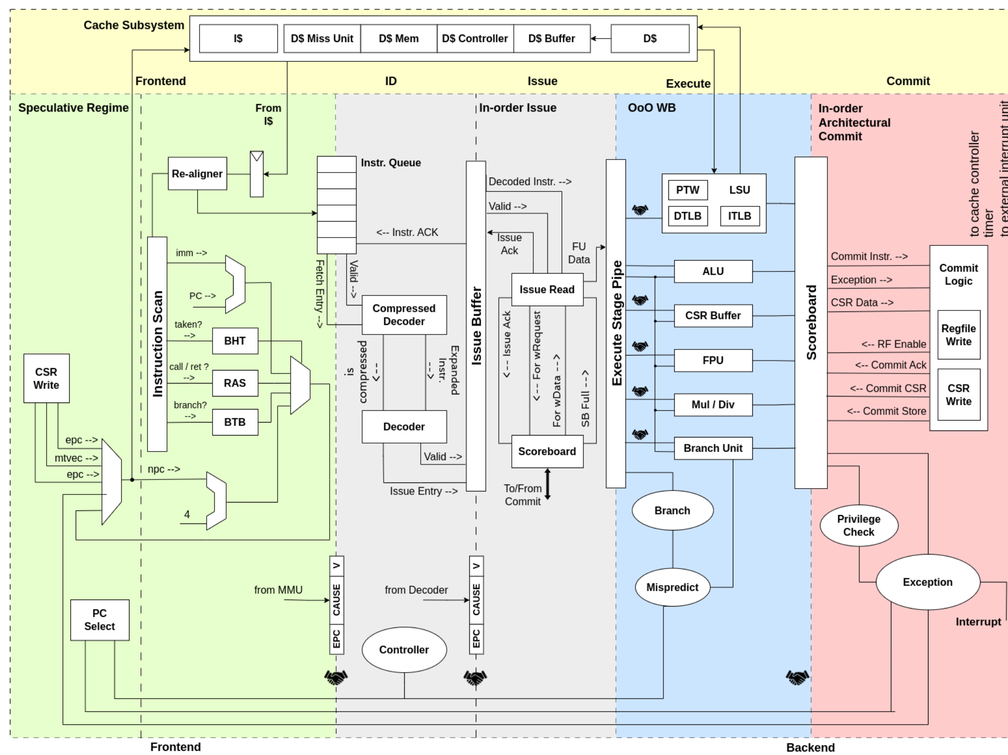


**Figure 3.2:** CVA6 internal architecture

[Gro24]

The commit stage is to most interesting to us, because it is the final stage of the pipeline in which the instructions are completed and results are committed. The commit stage of CVA6 can output up to N instructions per cycle and N is set as 2 as default and the tracer must be able to track all the commit ports of the CPU to obtain a correct trace.

By using the output of the commit stage it is possible to extrapolate all the necessary information to do a proper branch trace.

As we'll see in the next portion of the document, these signals are necessary to

determine the blocks and their fields.

## 3.2 Design

On a high level, this module takes the instructions committed by the CPU and stores them inside FIFOs; then these instructions are fed into an *FSM* that computes the parameters for each block and then outputs them.

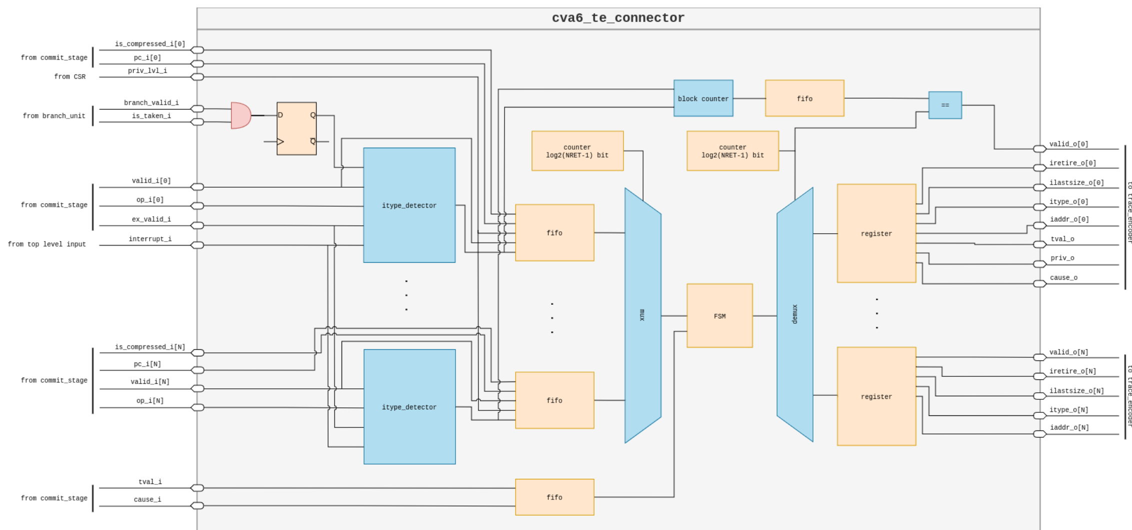


**Figure 3.3:** cva6_te_connector internal architecture

### 3.2.1 itype_detector

A fundamental part of the design is the itype_detector, whose objective is to determine the itype of the instruction. This is done by checking the operation associated with a committed instruction and checking if an exception or interrupt occurs in the same cycle. The branch-associated inputs are stored in a register because in CVA6 simulated waveforms these signals were asserted N cycles before the instruction commitment.

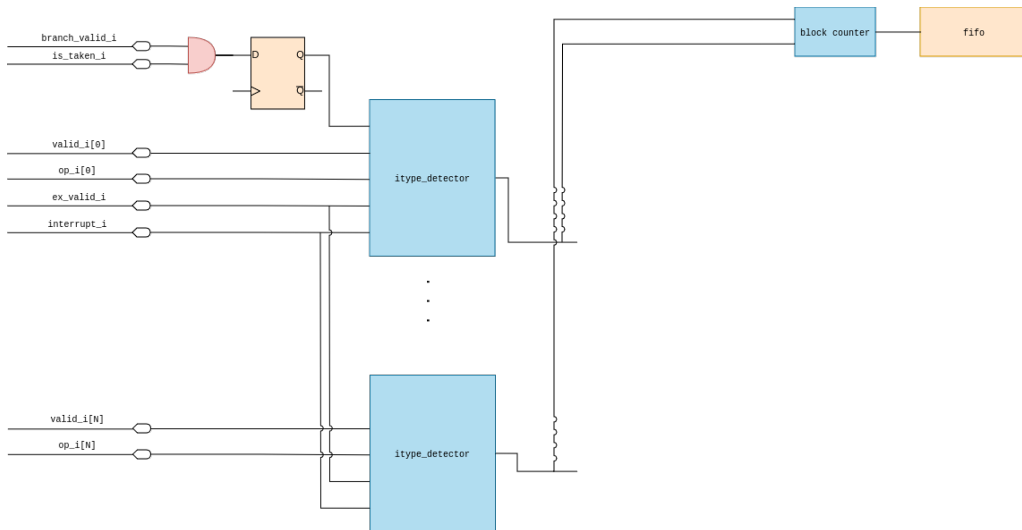


**Figure 3.4:** itype_detector associated logic

The itype determination is done by a combinational network, so the result is ideally available in one cycle.

After the itype of the input instructions is detected, another combinational network counts the number of blocks that will be output in one cycle by counting the number of itypes different from 0. This value is stored in a FIFO to save the result until the previous N blocks are output.

### 3.2.2 Serialization

Since more instructions can be committed than are processed, a FIFO is used for each commit port to work as an elastic buffer and store information without stalling the rest of the pipeline.

Then, instructions need to be fed inside the FSM that determines the block fields, using a counter that operates a multiplexer, which each cycle sends to the FSM values stored in a different FIFO.

If an exception or interrupt is encountered, the counter outputs only the first FIFO to prevent sending multiple exception or interrupt itypes to the FSM. This behavior is caused by exception and interrupt signals being connected to all itype_detectors.

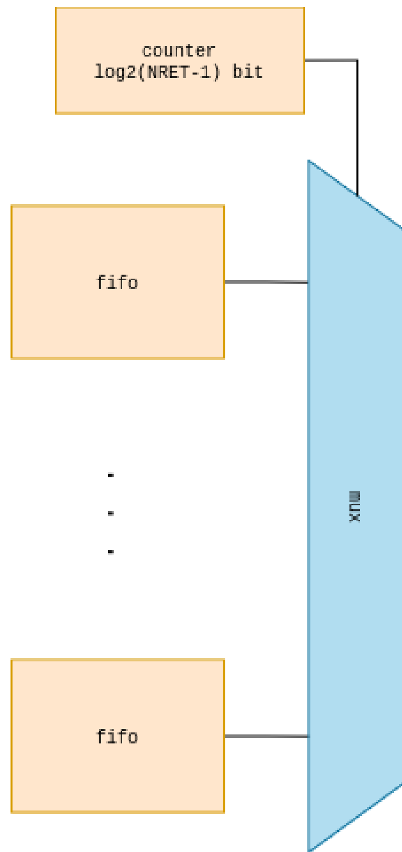


**Figure 3.5:** Serialization logic

### 3.2.3 Finite State Machine

The FSM is the core of this module, processing one instruction at a time, and when a block is ready, it outputs the fields, which are stored in a register if necessary.

The FSM has two states:

- *idle*, where the starting values are set;
- *count*, where the final parameters are set.

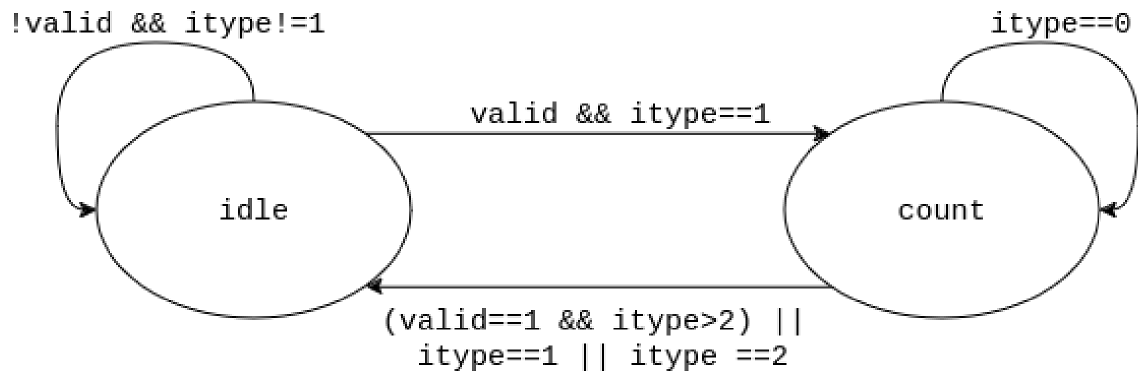


**Figure 3.6:** FSM states chart

More precisely, in the *idle* state:

- iaddr is set as the address of the itype == 0 instruction;
- iretire is incremented by 1 or 2, depending on the compressed input;
- ilastsize is set in case the first instruction is a special instruction; in the scenario in which in the clock cycle of the special instruction no instruction is committed, the ilastsize value is read from register.

The FSM transitions to count state if the instruction is a standard one. Otherwise, it populates the block fields and remains in this state, waiting for the next block to begin.

In *count* state:

- iretire is incremented;
- ilastsize is set in case of an exception or interrupt without a committed instruction;
- privilege level is set.

The FSM stays in this state while committed standard instructions are read.

In both states, the cause and tval fields are populated only if there's an interrupt or exception; otherwise, they remain 0.

### 3.2.4 Deserialization

Since up to N blocks can be output per cycle, the cva6_te_connector module parallelizes the blocks produced by the FSM. A demultiplexer operated by a counter stores each block emitted by the FSM inside a register.

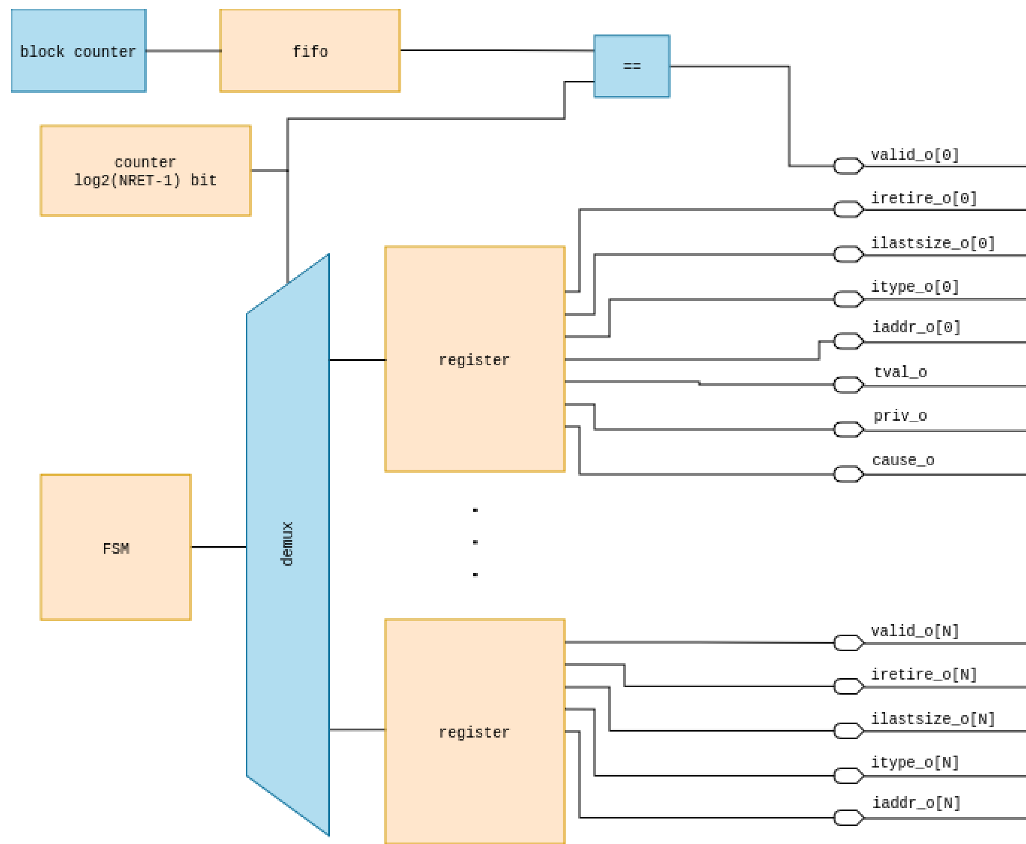


**Figure 3.7:** Deserialization logic

When all the blocks are ready - checked by comparing the value in the block number FIFO to the counter value - the valid signal associated with the module output is asserted, and data can be read by the TE.

# Chapter 4

# Evaluation

Once the design is complete it makes sense to try and synthesize it on an FPGA and see what is the impact of the two previous modules on the SoC area usage.

The parameters to consider are the following:

- timings, the change in these values shows how much the TE and cva6_te_-connector influence the working frequency of the SoC;
- area utilization, it is fundamental to compute how much silicon is necessary to perform tracing.

## 4.1 Testing platform

The platform chosen for the integration testing is the *Alsaqr SoC*. This platform was chosen due to the presence of two CVA6 cores and its ease of use in case of adding new modules to the design.

A System-on-Chip (SoC) is an integrated circuit that consolidates the core components of a computing system onto a single chip. This typically includes a processor, input/output interfaces, and other specialized hardware modules used to accelerate specific workloads. The design of an SoC aims to optimize performance, power efficiency, and size. By combining multiple functionalities into a single chip, SoCs reduce cost, increase reliability, and improve system integration compared to traditional multi-chip designs.

The Alsaqr SoC is divided in three main parts:

- Host Domain, it contains the two CVA6 cores, the AXI interconnect and the I/0 ports; it mainly works as the orchestrator of the whole SoC;

- Secure Subsystem Domain, it conatains all the modules designed for encryption and decryption, it is used for the security part of the SoC;
- Cluster Domain, it contains cores designed for acceleration like RI5CY and RedMule.

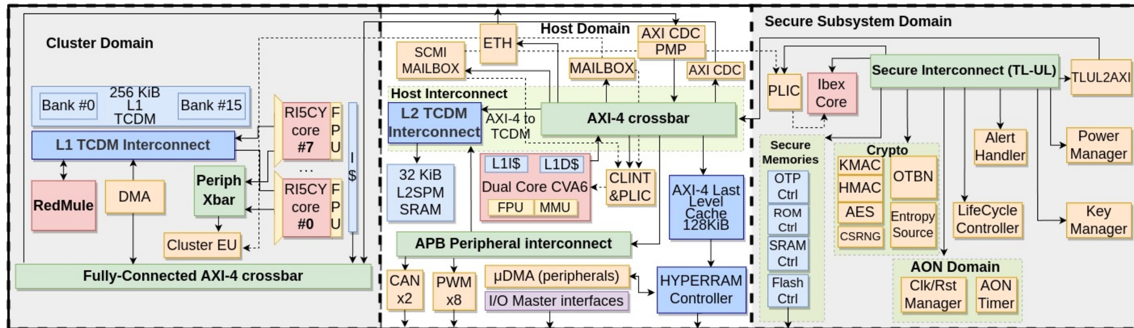


**Figure 4.1:** Alsaqr SoC architecture

Since the cva6_te_connector is designed to be connected to a CVA6 core, the only part of the Alsaqr SoC necessary to perform the testing is the *Host Domain*, that contains the two CVA6 cores. The Secure System Domain and the Cluster Domain are both excluded from the synthesis.

The target platform for the synthesis is the Xilinx VCU118, this board features the Virtex UltraScale+ XCVU9P-L2FLGA2104E FPGA. From now on, the term "FPGA" is referred to this specific FPGA model. The results are strictly related to the technlogy used in the FPGA and the synthesis for another target might lead to different area occupations and timings.

## 4.2 Timing

An important parameter to consider in order to determine the critical path is the *negative slack*. The negative slack is the difference between the time available for a signal to travel through a circuit path and the time necessary for the signal to reach its destination.

If this value is negative, it means the FPGA can not operate at the specified clock frequency. To make the negative slack positive and therefore having the FPGA working at the specified clock frequency the circuit can be modified by the developer - reducing the combinatorial logic between registers - or the synthesizer can arrange the logic on different positions inside the FPGA with a trade-off in terms of area usage.

The Worst Negative Slack is the lower value of the negative slack, this value is related to a specific path between registers: the critical path. The length of this path and its associated travel time determines the maximum clock frequency at which the FPGA can operate.[Har24]

The modules of the trace encoder are not inside the critical path and so the addition of both the te_cva6_connector and the trace encoder does not influence the operative frequency of Alsaqr Host Domain.

## 4.3 Area utilization

The area utilization is compared is compared with respect to two other hardware components:

- The Ariane core;
- The Host Domain block of the SoC.

### 4.3.1 Ariane core

Compare the area utilization to the one used by the Ariane core makes sense, because a TE is necessary for each core and in case of synthesis it is a necessary parameter to consider.

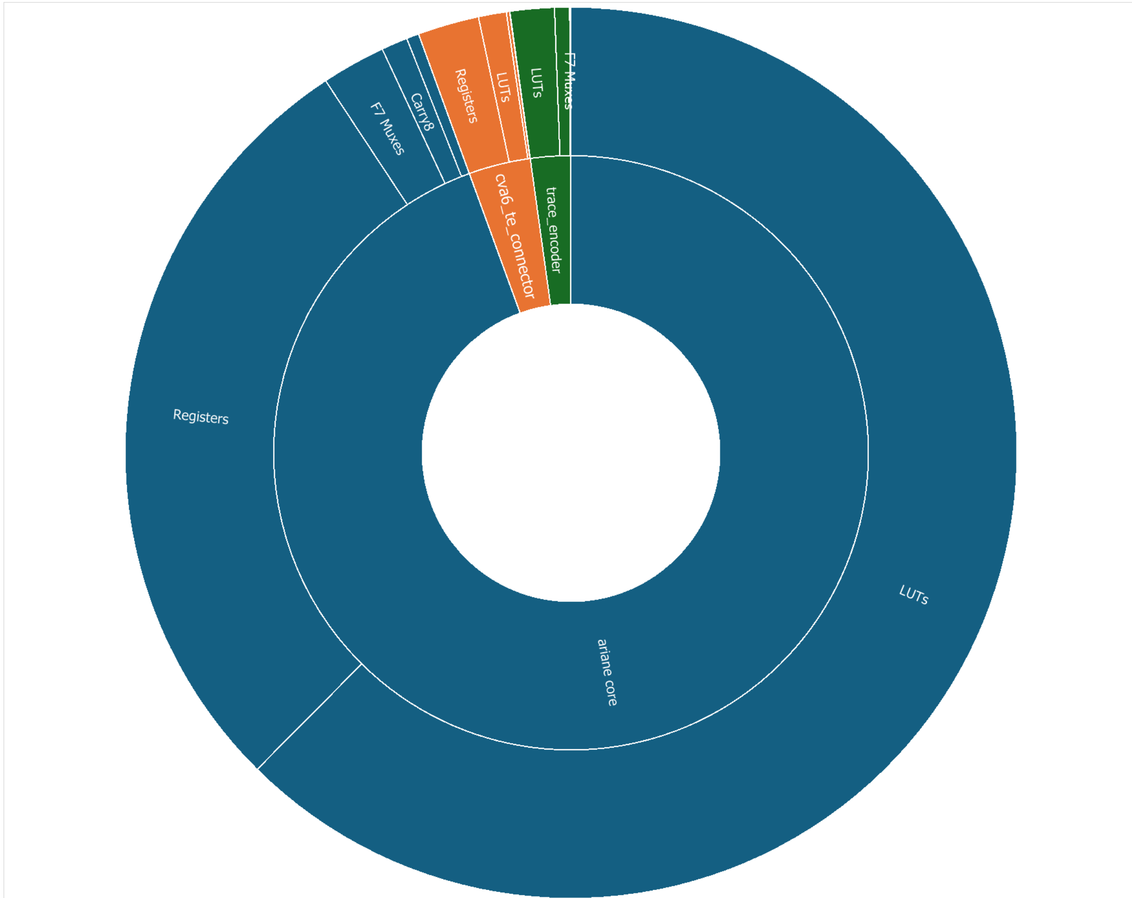


**Figure 4.2:** Area utilization with respect to an Ariane core

The TE and the cva6_te_connector combined occupy 0.21% of the total FPGA area; an Ariane core occupies 5.06% of the total FPGA area. This means the whole TE system occupies around 4.15% of the area occupied by the core to trace.

### 4.3.2 Alsaqr Host Domain

If the Host Domain of the Alsaqr SoC is considered, the area occupied by the TE and cva6_te_connector is even more negligible with respect to the area occupied by the whole SoC.

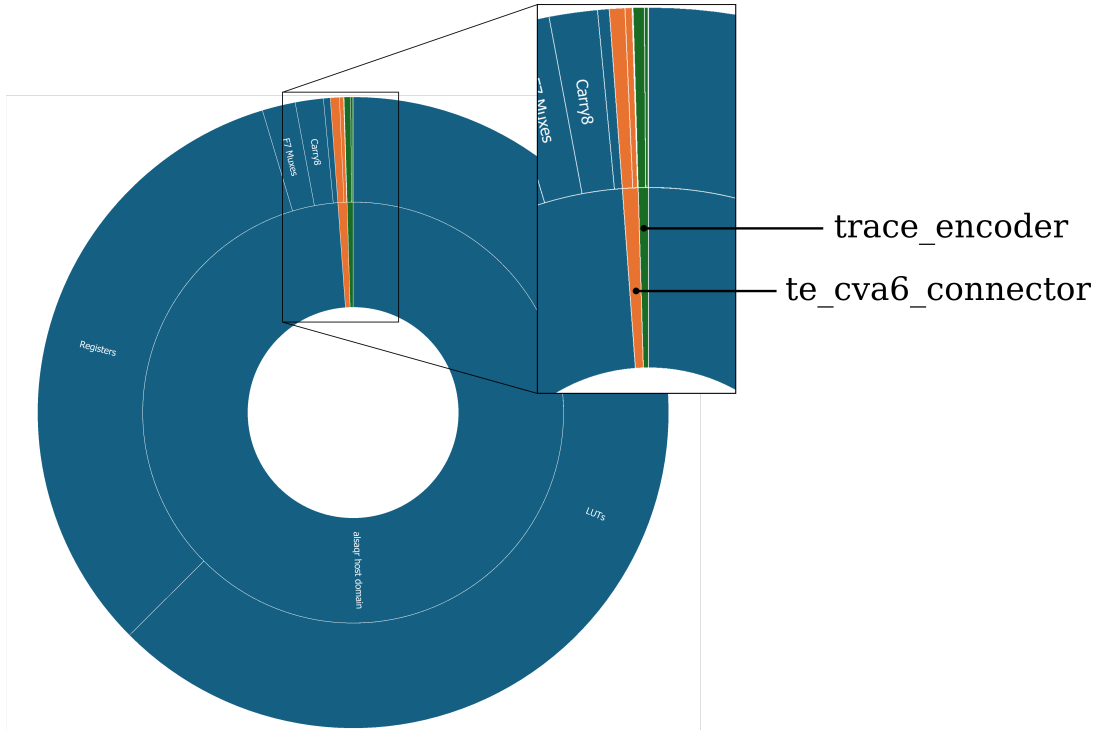


**Figure 4.3:** Area utilization with respect to the Alsaqr Host Domain

The small orange and green segments are the cva6_te_connector and the TE respectively. The Host Domain of Alsaqr SoC occupies the 24.33% of the available FPGA area, meanwhile the cva6_te_connector and TE occupies the 0.21% of it. This means the two modules necessary to trace a CVA6 core require 0.86% of the whole Alsaqr Host Domain area suggesting a very negligible area impact with respect to the SoC.

## 4.4 Benchmarking

As the specification name suggests, the objective of this tracing is being efficient and a good metric to evaluate efficiency is the compression rate. The compression rate considers the ratio between the raw data - in this case the instructions committed - and the compressed data - the packets emitted by the TE - and determines how much info can be saved.

The formula of the compression rate in percentage is the following:

$$Compression\ rate\ (\%) = (1 - \frac{Uncompressed\ data\ size}{Compressed\ data\ size}) * 100$$

The tests used to determine this metric are some of the already available inside the Alsaqr repository.

The compression rate is computed considering the two parameters in bits, so the number of committed instructions is multiplied by 32 - the length of a RISCV instruction - and the total length is multiplied by 8 to convert it from bytes to bits.

The formula for the compression rate becomes:

$$Compression\ rate = (1 - \frac{Committed\ instructions\ *\ 32}{Total\ packets\ bytes\ *\ 8}) * 100$$

Here the results for each test:

| Test name | Committed instructions | Packets emitted | Total packet bytes | Compression rate % |
|---|---|---|---|---|
| axi_hyper_fibonacci | 17576 | 56 | 176 | 99.75 |
| backend_test | 331 | 50 | 162 | 87.76 |
| bypass_cva6_dco | 16500 | 89 | 305 | 99.54 |
| can | 351 | 58 | 179 | 87.25 |
| dhrystone | 2370 | 46 | 149 | 98.43 |
| fp16_matmul | 22894 | 90 | 273 | 99.70 |
| fp16-vec_matmul | 23331 | 94 | 287 | 99.69 |
| gpios | 4962 | 125 | 479 | 97.59 |
| gpios_all | 6126 | 285 | 1231 | 94.98 |
| hello | 654 | 72 | 250 | 90.44 |
| hello_culsans | 772 | 73 | 360 | 88.34 |
| hyperbus | 14585 | 75 | 253 | 99.57 |
| kmeans | 936 | 57 | 186 | 95.03 |
| l1_test | 16784 | 62 | 197 | 99.71 |
| llc_spm_test | 323 | 49 | 160 | 87.62 |
| mbox_test | 1388 | 131 | 463 | 91.66 |
| mm | 16784 | 62 | 197 | 99.71 |
| sb_macl_444 | 23221 | 83 | 244 | 99.74 |
| sb_macl_844 | 20273 | 82 | 242 | 99.70 |
| timer | 956 | 216 | 565 | 85.22 |

**Table 4.1:** Info for each test

From the previous results it is possible to compute the average compression rate:

$$Compression\ rate_{AVG} = 95.07\%$$

This means the packets emitted reduced the data usage - with the respect to the raw code - an average of 95.07%. For example if the instructions committed has a size of 100kB, the resulting packets required to cover that portion of code have a size of around 5kB.

# Bibliographic references


[ARM24] ARM. Learn the architecture - understanding trace. https://developer.arm.com/documentation/102119/0200/What-is-trace-, 2024.

[Gro24] OpenHW Group. Cva6 risc-v cpu. https://github.com/openhwgroup/cva6, 2024.

[Har24] Lance Harvie. How to perform static timing analysis (sta) for correct fpga operation. https://runtimerec.com/how-to-perform-static-timing-analysis/, 2024.

[PR] Gajinder Panesar and Iain Robertson. *Efficient Trace for RISC-V v2.0.3.*