# Pasithea-1: An Energy-Efficient Self-Contained CGRA With RISC-Like ISA

Tobias Kaiser and Friedel Gerfers

Technische Universität Berlin, Chair of Mixed Signal Circuit Design, Germany
kaiser@tu-berlin.de

**Abstract.** This paper presents Pasithea-1, an energy-efficient coarse-grained reconfigurable array (CGRA) with RISC-like programming interface. In contrast to traditional RISC instruction sets, which are designed for centralized von Neumann architectures, it applies RISC principles to design an instruction set for energy-efficient CGRAs. Similar to dataflow and in-place processing architectures such as TRIPS and DiAG, Pasithea-1 can execute complex application code without external control. To demonstrate its programming, mechanisms and examples for dataflow, control flow, subroutine calls, coroutines and multi-threading are shown. A microarchitecture implementing the instruction set is presented. To reduce switching activity, its instructions, once fetched, remain in fabric, where they can be executed repeatedly without re-fetching. Pasithea-1 is compared against a minimal RISC system based on placed-and-routed designs. Using netlist simulation, energy and performance were compared. With a 10.1× energy reduction in the memory hierarchy and a 3.1× overall energy reduction, the described architecture surpasses the RISC system considerably in energy efficiency.

**Keywords:** Reconfigurable Computing · Coarse-Grained Reconfigurable Array · Instruction Set Architecture · Energy-Efficient Computing.

## 1 Introduction

Energy-efficient computing enables future low-power applications. In addition, it is key to high performance under thermal constraints. Although technological scaling continues to increase logic densities, it no longer improves energy efficiency proportionally [1]. New approaches in computer architecture could provide ways forward [2]. This paper explores a reconfigurable array as general-purpose alternative to von Neumann architectures. Despite requiring more silicon area, this approach can improve energy efficiency by reducing switching activity.

### 1.1 Reconfigurable Computing

In von Neumann computers, a central processing unit (CPU) fetches, decodes and executes a continuous stream of instructions. Coarse-grained reconfigurable arrays (CGRAs) break with this principle: Their machine code, also called configuration bitstream, is distributed to array elements and can remain largely

This version of the contribution has been accepted for publication after peer review but is not the Version of Record and does not reflect post-acceptance improvements, or any corrections. The Version of Record is available online at: http://dx.doi.org/[insert DOI]. Use of this Accepted Version is subject to the publisher's Accepted Manuscript terms of use: https://www.springernature.com/gp/open-research/policies/accepted-manuscript-terms.

stationary during execution. Array elements perform computations using local functional units and exchange data and control information to achieve complex behavior. CGRAs potentially require less energy than von Neumann architectures, as they do not require a sustained instruction stream.

Von Neumann processors execute instructions inherently in sequence. In CGRAs, spatial replication and coexistence of resources encourages parallel execution. While this inherent parallelism enables exceptional performance, it complicates their programming. General-purpose program code in high-level languages is founded upon an assumption of instruction (or statement) sequentiality. A shift away from this seems unlikely. This gap between software and CGRA hardware can be bridged by elaborate compiler techniques, as described by Cardoso et al. [3]. Despite successes, widespread replacement of von Neumann CPUs with CGRAs in general-purpose computing is currently seen as unlikely.

### 1.2 Related Work

The four-type classification shown in Table 1 covers pure von Neumann architectures, pure CGRA architectures and mixtures of both.

In Type II systems, CGRA configuration and von Neumann programming are separated in machine code. Thus, the programmer or compiler needs to divide programs into separate parts for the two sub-architectures. Examples for Type II systems are [9–12].

Type III systems are programmed using a von Neumann instruction set architecture (ISA) only. They execute some parts of the instruction stream natively on a von Neumann core; other parts are dynamically translated to CGRA configurations and offloaded to a CGRA. While this exposes a homogeneous ISA, the translation process limits the scope of CGRA execution and incurs additional hardware overhead. Examples for Type III systems are [13–16].

This paper focuses on Type IV systems: self-contained CGRAs that can run complex programs without supervising von Neumann CPU or a translation layer. Their homogeneous ISAs and microarchitectures make them compellingly simple. Table 2 compares Pasithea-1, which we propose in this paper, to four Type IV CGRAs and traditional RISC processors.

DiAG [4], a CGRA-like general-purpose processor, differs from the other approaches by implementing RISC-V, a von Neumann ISA, but spatially distributing its instructions to PEs for *in-place execution*. Once loaded, instructions remain in PEs for extended time periods and can be executed several times. This instruction reuse distinguishes DiAG from earlier in-place processors [17, 18].

**Table 1:** Classification of processors based on integration and role of CGRA

| | **Type I** | **Type II** | **Type III** | **Type IV** |
|---|---|---|---|---|
| **Primary unit** | von Neumann | von Neumann | von Neumann | **CGRA** |
| **Subordinated unit** | None | **CGRA** | **CGRA** | None |
| **ISA** | **homogeneous** | heterogeneous | **homogeneous** | **homogeneous** |

**Table 2:** Qualitative comparison of classic RISC and Type IV CGRA architectures

<table>
<tr><th></th><th>Classic RISC</th><th>DiAG [4]</th><th>Pasithea-1 (this work)</th><th>TRIPS [5] (EDGE [6])</th><th>Wavescalar [7]</th><th>PACT XPP [8]</th></tr>
<tr><td>Type</td><td>Type I (von Neumann)</td><td colspan="5">Type IV (self-contained CGRA)</td></tr>
<tr><td rowspan="2">ISA</td><td colspan="2">RISC</td><td colspan="4">custom, statically fragmented code</td></tr>
<tr><td colspan="6">← RISC resemblance</td></tr>
<tr><td>Local dataflow</td><td>central register file</td><td>register lanes</td><td>single shared bus</td><td>multiple regfiles</td><td>multiple shared buses + network</td><td>packet-oriented network</td></tr>
<tr><td rowspan="2">Local control flow</td><td colspan="3" rowspan="2">arbitrary</td><td colspan="3">dataflow-driven</td></tr>
<tr><td>no internal loops</td><td>arbitrary</td><td></td></tr>
<tr><td>Global dataflow</td><td colspan="2">central register file</td><td>fragment-level msg. passing</td><td>central register file</td><td>instr.-level msg. passing</td><td>FIFO buffers</td></tr>
<tr><td>Self-reconfig.</td><td>N/A</td><td colspan="4">implicit</td><td>explicit, low-level</td></tr>
</table>

Programming interfaces of TRIPS [5], Wavescalar [7] and PACT XPP [8] bear little resemblance to RISC. To allow running programs that exceed their fabric capacity, machine code of those architectures is statically divided into code fragments as smallest units of reconfiguration. In contrast to DiAG and RISC, their local control flow not explicitly encoded, but driven by dataflow.

### 1.3 This Work

To overcome challenges in CGRA programming and allow a wider range of applications to benefit from its merits, we propose rethinking instruction set design to bridge the gap between CGRA hardware and general-purpose software. We present Pasithea-1, an instruction set with a corresponding Type IV CGRA microarchitecture. Its two objectives are to allow easy programming and to surpass minimal RISC systems in energy efficiency by reducing instruction movement. Accordingly, the architecture is named Pasithea-1, after a mythical goddess of rest and relaxation.

RISC principles were used as guidance in the design process, leading to a simple and familiar programming interface. In establishing low-level instruction sequentiality, our ISA departs from typical CGRA programming. Being tailored for CGRA-based execution sets the Pasithea-1 ISA apart from other RISC instruction sets. Based on its hardware and software properties, Table 2 places Pasithea-1 between DiAG and TRIPS.

The Pasithea-1 ISA is laid out in Section 2. Its programming is explained in Section 3. The CGRA microarchitecture is presented in Section 4. Section 5 describes how the novel architecture was compared to a RISC system. Results are shown in Section 6. The paper finishes with a discussion in Section 7.

## 2 Instruction Set Architecture

Machine code for Pasithea-1 is structured in fragments, which are delimited by start and end markers shown in Figure 1a. Each fragment consists of up to 64 instructions, which comprise a primary instruction word (formats D and W) and optional prefixes (formats S and I). Figure 1 gives an ISA overview.

Section 2.1 describes the fragment instance mechanism, a level of indirection that makes fragments reentrant. Section 2.2 introduces local dataflow and control flow mechanisms, which link instructions within a fragment. Section 2.3 describes creation and termination of fragment instances and communication between them. Section 2.4 relates the ISA to RISC principles.

### 2.1 Fragment Instances

A fragment is a group of up to 64 machine instructions that can be loaded into a contiguous CGRA sub-array. A fragment instance (FI) *attaches runtime data to a fragment* and thus represents execution state of a sub-array. Multiple FIs of a single fragment can coexist at runtime. This enables fragment reentrancy, i.e. independent coexisting function calls executing the same fragment code.

Such a code instantiation mechanism is not found in RISC processors, as they do not attach runtime data to machine code, but maintain execution state in the CPU register file and in memory using a call stack.

Coexisting FIs are concurrent, unless communication primitives (Section 2.3) constrain event ordering. Depending on microarchitecture capabilities, such coexisting FIs can be executed in sequence or in parallel. Section 3.2 demonstrates how fragments can be used as subroutines, coroutines and threads.

### 2.2 Local Interaction with Target Instruction Pointers (TIPs)

Pasithea-1 instructions are designed for decentral execution in a CGRA rather than a CPU. They are executed sequentially. Like in RISC, machine code order and explicitly encoded branches determine the sequence of execution.

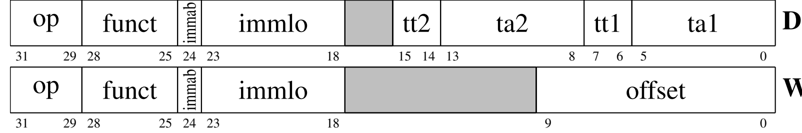
Instantiation attaches two 32-bit **operand registers**, *opA* and *opB*, to each instruction. Their values are used by *local* instructions as ALU inputs; *memory* and *global* instructions use *opA* and *opB* as address or data words for memory access or communication operations, details of which are specified in Figure 1b.

On instantiation, each instruction initializes the operand register selected by *immab* with an **immediate value** and the other operand register with zero. By default, immediates are limited to $[-32 \,..\, 31]$. The I prefix extends this range to arbitrary 32-bit values.

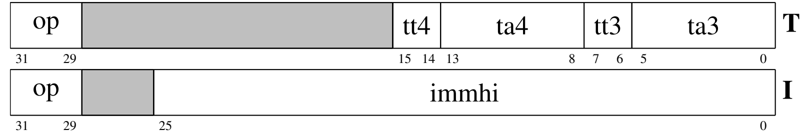
All D format instructions produce a result word on execution. To achieve local **dataflow**, this result can be written to operand registers of other local instructions. The originating instruction encodes each such data transfer as *target instruction pointer* (TIP). A TIP consists of a target address field ($taX$), which references one of the fragment's 64 instructions, and a target type field ($ttX$), whose encoding is shown in Figure 1d. Each D format instruction can directly

*Fragment start / end marker:*

*Primary instruction types:*




*Instruction prefixes:*




**(a)** Instruction formats

| Group | Mnemonic | Fmt. | Operation |
|---|---|---|---|
| Local | or, and, xor, add, sub slt, sltu, sll, srl, sra | D | res := ALU(opA, opB) |
| Memory | lw, lh, lb, lhu, lbu | D | res := Mem [opA + opB] |
| | sw, sh, sb | W | Mem [opA + offset] := opB |
| Global | recv *(receive)* | D | res := $\mathrm{Msg}_{\mathrm{self}}\,[MI]$ with $MI = (\mathrm{opA} + \mathrm{opB})_{2:0}$ |
| | send | W | $\mathrm{Msg}_{FIA}\,[MI]$ := opB with $FIA = (\mathrm{opA} + \mathrm{offset})_{31:6}$, $MI = (\mathrm{opA} + \mathrm{offset})_{2:0}$ |
| | inv *(invoke)* | D | invokes fragment at *addr* using *RMI* as return MI with $addr_{31:3} = (\mathrm{opA} + \mathrm{opB})_{31:3}$, $RMI = (\mathrm{opA} + \mathrm{opB})_{2:0}$; res := new FIA |
| | term *(terminate)* | W | terminates current FI |

**(b)** Operations encoded by mnemonics

| op | funct | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 |
| 000 | or | and | xor | add | sub | slt | sltu | sll | srl | sra |
| 001 | recv | lb | lh | lw | lbu | lhu | inv | | | |
| 010 | send | sb | sh | sw | term | | | | | |
| 011 | *S prefix* | | | | | | | | | |
| 100 | *I prefix* | | | | | | | | | |
| 101 | *fragment start / end marker* | | | | | | | | | |

**(c)** Mnemonics by opcode and funct code

| ttX | function | X |
|---|---|---|
| 10 | write opA | 1,2,3,4 |
| 11 | write opB | 1,2,3,4 |
| 00 | branch on zero | 1 |
| 01 | branch unless zero | 1 |
| 0x | no operation | 2,3,4 |

**(d)** Target type *ttX* (tt1, tt2, tt3, tt4) encoding

**Fig. 1:** Pasithea-1 ISA overview. Highlighted instructions are adopted from RISC-V.

encode two TIPs (tt1, ta1), (tt2, ta2). Use of the T prefix extends the possible number of TIPs per instruction to four.

By default, instructions are executed in machine code sequence. The first TIP can be used for **conditional branching**, as specified in Figure 1d. This mechanism supersedes dedicated control flow instructions.

### 2.3 Global Interaction of Fragment Instances

FIs exchange data by **message passing**. Unique 26-bit fragment instance addresses (FIAs) identify FIs throughout their life times. For receiving messages, each FI possesses eight message registers. Each register is identified by a three-bit message index (MI) and either holds a single 32-bit word or is marked as empty. The *send* instruction, shown in Figure 1b, sends *opB* to a message register of another FI. *opA* specifies FIA and MI of the target message register. Message registers already containing a value are overwritten. The *recv* instruction returns values received in local message registers. If a requested message register is empty, *recv* will block further FI execution and wait for message reception.

To **spawn new FIs**, fragments are invoked with *inv*, which takes a fragment address as input and returns the FIA of a newly created FI. The new FI automatically receives a *return handle* in message register 0. It can use this handle, which consists of the invoker FIA and a return MI (RMI) set by the invoker, to send return values to the invoker. The RMI can be used to assign return values of concurrent FIs to separate message registers. The *term* instruction terminates the current FI, invalidating its execution state and FIA.

### 2.4 What's the RISC?

Like RISC architectures, Pasithea-1 follows a scalar data model and sequential control flow model. Its adoption of RISC-V mnemonics [19] contributes to similarities in programming. It is furthermore influenced by key RISC concepts [20]:

1. Each instruction performs a simple operation, which can be executed easily without translating it to microinstructions.
2. Supporting concept 1, instructions only support one type of general-purpose operand by default: per-instruction operand registers.
3. Memory or message operands, less commonly needed than the general-purpose operand type, are only available through dedicated instructions.
4. The ISA is designed for hardware simplicity but leaves room for future hardware upscaling and optimizations that maintain ISA compatibility.

## 3 Programming

Sections 3.1 and 3.2 describe programming of individual fragments and programming patterns for interaction between fragments, respectively.

### 3.1 Local Programming

Within fragments, TIPs encode data and control flow. Defining them manually by labeling and cross-referencing instructions turned out infeasible for low-level programming by hand. Therefore, an intermediate assembly-like language (IAL) has been devised. It allows instructions to read and write local variables, which start with a $ sign and mimic RISC's general-purpose registers. The following syntax is used for IAL instructions:

```
[L:] [goto L (if|ifnot)] [$var = ] op(opA, [offset,] opB)
```

The following steps translate an IAL fragment to machine code:

1. Compile-time expressions are resolved.
2. Unconditional jumps are transformed into conditionally branching *or* instructions with constant operands.
3. A control and data flow graph (CDFG) is constructed.
4. Ineffective data flow edges are removed from the graph. A data flow edge from instruction S to operand register opX of instruction T is ineffective, if there are no control flow paths from S to T that leave T.opX intact.
5. When instructions require more than four TIPs, excess TIPs are distributed to supplementarily inserted *or* instructions.
6. Machine code is generated. Dataflow edges and conditional branches are encoded as TIPs.

Table 3 shows an example fragment implementing Euclid's algorithm. Instructions 0 and 1 receive two input values, of which the greatest common divisor is to be calculated, from message registers 1 and 2. Instructions 2-5 form a loop, which is exited by a conditional jump from instruction 2 to instruction 6 once $p and $q are equal. Instructions 4 and 5 always branch to instruction 2 because their results are never zero. Instruction 6 reads the return handle (see

**Table 3:** Translation of IAL fragment *gcd*, which implements Euclid's algorithm, to Pasithea-1 machine code. Control flow: S=successor, C=conditional branch target, T=true (non-zero), F=false (zero); dataflow: A=write opA, B=write opB.

| | ↓ Source instruction | CDFG adjacency matrix | | | | | | | | | Fmt. | TIP 1 | TIP 2 | TIP 3 | TIP 4 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | `$p = recv(1, 0)` | | S | A | | | | | B | | D | 7.opB | 2.opA | - | |
| 1 | `$q = recv(2, 0)` | | | SB | | | | | | | D | 2.opB | - | - | |
| 2 | `loop: goto end ifnot $diff = sub($p, $q)` | | | | SA | A | B | C | | | D | F→6 | 4.opA | 5.opB | 3.opA |
| 3 | `goto p_lt_q if slt($diff, 0)` | | | | | S | C | | | | D | T→5 | - | - | |
| 4 | `goto loop if $p = or($diff, 0)` | | | CA | | | S | | B | | D | T→2 | 7.opB | 2.opA | - |
| 5 | `p_lt_q: goto loop if $q = sub(0, $diff)` | | | CB | | | | S | | | D | T→2 | 2.opB | - | |
| 6 | `end: $fia_ret = recv(0, 0)` | | | | | | | | SA | | D | 7.opA | - | - | |
| 7 | `send($fia_ret, 0, $p)` | | | | | | | | | S | W | - | | - | |
| 8 | `term()` | | | | | | | | | | W | - | | - | |
| | **Target instruction →** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | | | | | |

Section 2.3) and instruction 7 uses this handle to return the fragment's result value. Alongside the fragment's IAL source code, Table 3 shows the processed CDFG and resulting TIPs that will be encoded in machine code.

### 3.2 Global Programming

Typical programs comprise several hundred subroutines. Subroutines that are not too complex can be implemented using a single fragment. As programs commonly incur millions of subroutine calls per second [21], a streamlined calling mechanism is crucial for programmability and performance.

Figure 2a shows an example for subroutine calls. Like *gcd*, *gcd3* starts with receiving arguments and ends with returning a result. In between, it calculates the greatest common divisor of three numbers using two *gcd* subroutine calls: $\mathrm{gcd3}(p, q, r) = \mathrm{gcd}(\mathrm{gcd}(p, q), r)$. Subroutine calls are performed in three stages:

1. *Fragment invocation:* The requested subroutine fragment is invoked using *inv*. In addition to the fragment address, an RMI is specified and passed to the invoked fragment. Both calls in *gcd3* use an RMI of 1. *inv* returns a FIA, which will be used in the next two stages.
2. *Send arguments (optional):* The invoker sends argument values to the subroutine. By convention, the $n$-th argument is passed at $\mathrm{MI} = n$.
3. *Receive return values & synchronize control flow:* The subroutine's return value is obtained using a receive instruction. As *recv* waits until the return value is written, it also provides synchronization. A subroutine without return value needs to return a blank message for synchronization.

Figure 2b shows recursive subroutine calls, making use of reentrancy.

As described in Section 2.1, coexisting FIs are concurrent by default. Figure 2c shows an example of two concurrent *gcd* calls, which can be considered separate threads. Depending on the microarchitecture's capabilities, they can be executed in parallel (**thread-level parallelism**).

**Coroutines** incorporate the full structural power of subroutines, but can pass control and data back and forth without relinquishing their local state [22]. Coroutine-like constructs are gaining popularity and are available in many modern programming languages such as Python, C++, Rust and Go. An example that implements coroutines in Pasithea-1 is shown in Figure 2d: The iterator subroutine *lfsr_test* receives a stream of bits from the generator subroutine *lfsr*, which implements a Galois linear-feedback shift register (LFSR). *lfsr_test* prints all received bits through a subroutine call to *print_bit*.

## 4 Microarchitecture

The microarchitecture consists of 128 processing elements (PEs), grouped in 8 tiles of 16 PEs each. Figure 3a shows an overview. Section 4.1 describes the fragment instance manager (FIM), which manages creation and termination of FIs and transfers FIs between fabric and memory. Section 4.2 describes execution of FIs on fabric. Section 4.3 describes storage of dormant FIs in memory.

```
1 fragment gcd3
2 $p = recv(1, 0)               } Receive
3 $q = recv(2, 0)               } arguments
4 $r = recv(3, 0)
5 $fia1 = inv(gcd+1, 0)         } Subroutine
6 send($fia1, 1, $p)            } call 1
7 send($fia1, 2, $q)
8 $x = recv(1, 0)
9 $fia2 = inv(gcd+1, 0)         } Subroutine
10 send($fia2, 1, $x)           } call 2
11 send($fia2, 2, $r)
12 $res = recv(1, 0)
13 $fia_r = recv(0, 0)          } Return result
14 send($fia_r, 0, $res)        } & terminate
15 term()
16 endfragment
```

**(a)** *gcd3:* subroutine call example

```
1 fragment qsort
2        $first = recv(1, 0)
3        $last = recv(2, 0)
4        goto end ifnot
                 slt($first, $last)
5        $piv = lbu($last, 0)
6        $p = sub($first, 1)
7        $j = or($first, 0)
8 loop:  $mj = lbu($j, 0)
9        goto noswap if slt($piv, $mj)
10       $p = add($p, 1)
11       $mp = lbu($p, 0)
12       sb($p, 0, $mj)
13       sb($j, 0, $mp)
14 noswap: $j = add($j, 1)
15       goto loop ifnot slt($last, $j)
16       $p_minus_1 = sub($p, 1)
17       $p_plus_1 = add($p_minus_1, 2)
18       $fia_qs1 = inv(qsort+1, 0)
29       send($fia_qs1, 1, $first)
30       send($fia_qs1, 2, $p_minus_1)
31       recv(1, 0)
32       $fia_qs2 = inv(qsort+2, 0)
33       send($fia_qs2, 1, $p_plus_1)
34       send($fia_qs2, 2, $last)
35       recv(2, 0)
36 end:  $fia_r = recv(0, 0)
37       send($fia_r, 0, 0)
38       term()
39 endfragment
```

**(b)** *qsort*: Quicksort implementation

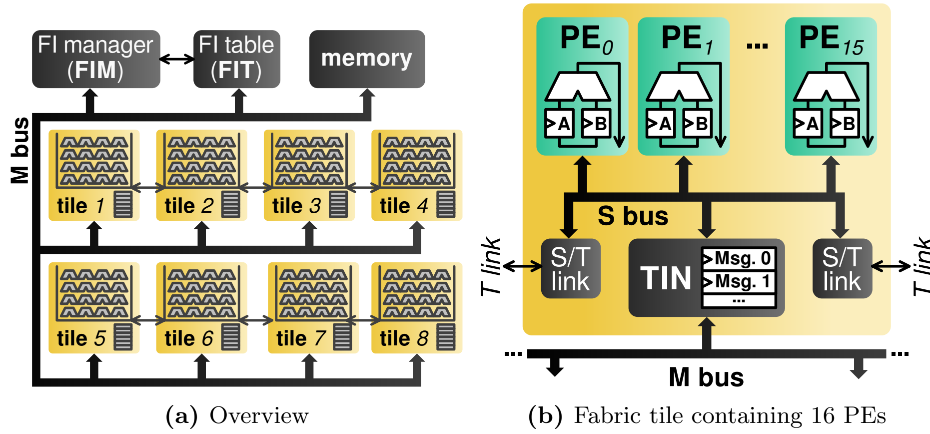
```
1 fragment gcd4mt
2 $p = recv(1, 0)               } Receive
3 $q = recv(2, 0)               } arguments
4 $r = recv(3, 0)
5 $s = recv(4, 0)
6 $fia1 = inv(gcd+1, 0)         } Launch
7 send($fia1, 1, $p)            } thread 1
8 send($fia1, 2, $q)
9 $fia2 = inv(gcd+2, 0)         } Launch
10 send($fia2, 1, $r)           } thread 2
11 send($fia2, 2, $s)
12 $x = recv(1, 0)              } Join
13 $y = recv(2, 0)              } threads 1 & 2
14 $fia3 = inv(gcd+1, 0)        } Subroutine
15 send($fia3, 1, $x)           } call
16 send($fia3, 2, $y)
17 $res = recv(1, 0)
18 $fia_r = recv(0, 0)          } Return result
19 send($fia_r, 0, $res)        } & terminate
20 term()
21 endfragment
```

**(c)** *gcd4mt:* multi-threading example

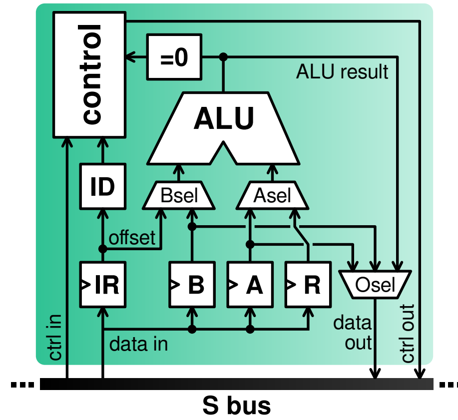
```
1 fragment lfsr_test
2        $fia_lfsr = inv(lfsr + 1, 0)
3        $i = 20
4 loop:  $bit = recv(1, 0)
5        $fia_print = inv(print_bit+1, 0)
6        send($fia_print, 1, $bit)
7        recv(1, 0)
8        send($fia_lfsr, 1, 1)
9        goto loop if $i = sub($i, 1)
10       term()
11 endfragment
12 fragment lfsr
13       $fia_r = recv(0,0)
14       $lfsr = 1
15 loop: $bit = and($lfsr, 1)
16       send($fia_r, 0, $bit)
17       $lfsr = srl($lfsr, 1)
18       goto skip ifnot or($bit, 0)
19       $lfsr = xor($lfsr, 46080)
20 skip: goto loop if recv(1,0)
21       term()
22 endfragment
```

**(d)** *lfsr:* coroutine example

**Fig. 2:** Example code for Pasithea-1




**(a)** Overview **(b)** Fabric tile containing 16 PEs




**(c)** Single processing element (PE)

**Fig. 3:** Pasithea-1 microarchitecture

### 4.1 Fragment Instance Management

The FIM allocates a fixed-size memory frame for every FI. It manages a pool of unused frames as linked list. Frame memory is only used when the corresponding FI is dormant, i.e. not present on fabric (see Section 4.3). Frame base addresses also serve as fragment instance addresses (FIAs).

When a FI is terminated, its machine code is retained in fabric as *residual fragment*. When a fragment is invoked and a matching residual fragment is available, the FIM reuses it, omitting instruction fetching.

By default, new FIs are created on fabric. If fabric occupancy of other FIs and residual fragments prevents this, new FIs are created in memory.

When all FIs on fabric are waiting for messages (*recv*) from dormant FIs, a stall is detected. This triggers the following **scheduling routine**: A dormant FI from the ready queue (Section 4.3) is restored to fabric. To make space for

the dormant FI, residual fragments are cleared. If this does not suffice, waiting FIs are evicted to memory. Least-recently used (LRU) policies are employed for clearing of residual fragments and eviction of waiting FIs.

### 4.2 Tiles & PEs: Fragment Instances On Fabric

Every PE can hold one instruction and its corresponding execution state. A variable number of tiles is required per FI: For FIs of 16 or fewer instructions, a single tile suffices. For FIs of up to 32 or 64 instructions, two or four tiles are linked together using the *T link*, which joins S buses of adjacent tiles. As FIs enter and leave fabric, tiles are linked and unlinked on-the-fly. A similar approach has previously been used in EDGE architectures [23].

FIs communicate with other FIs, the FIM and memory using tile interface nodes (TINs), which contain the message registers and connect to the global M bus. When messages are sent to FIs on fabric, target FIAs are translated to tile indices by the fragment instance table (FIT), which the FIM manages.

Figure 3b shows one tile. Within a tile or a group of T-linked tiles, the S bus connects all PEs and the TIN. The S bus is a shared 32-bit wide data bus with additional meta-data and control signals. It allows the active instruction to send output data to the TIN or up to two local PEs. Independent of this data flow capability, the S bus also features signals for passing control flow to a PE or the TIN.

Figure 3c shows the individual PE. Its instruction register (IR) can hold a primary instruction and optional prefixes. *opA* and *opB*, defined in the ISA, are held in the A and B registers. Results of memory and global operations are received by the TIN and forwarded through the R register of the initiating PE.

During FI loading, the memory unit sends the fragment's machine code to the TIN, which forwards it to local PEs through the S bus. If the FI was previously dormant, its memory representation is restored. The TIN starts execution once all instructions are fetched and their execution states are restored if necessary.

By default, local instructions (Figure 1b) take one cycle to execute. Branches add no latency. An S prefix adds one additional cycle of latency. Global and memory instructions incur additional latencies.

### 4.3 Dormant Fragment Instances

When an FI waiting for a message is evicted from fabric, its execution state is transferred to its allocated memory frame. This includes all operand register values, message register values, the awaited MI, a marker for the currently active instruction and the fragment code address. Operand register values are stored using a compressed format, which encodes operand registers equal to zero or equal to their immediate value using compact four-bit codes.

Addresses of dormant FIs cannot be resolved by the FIT. Messages addressed to them are delivered through the FIM, which writes them to the frame memory of the targeted FI. If the delivered MI matches its awaited MI, the FI is tagged as ready and added to the ready queue, a linked list managed by the FIM.

### 4.4 Memory Subsystem

A 256-bit wide memory bus connects the system to main memory. For instruction fetching, a single 256-bit instruction buffer is used. Load/store accesses utilize a $4 \times 256$-bit fully associative data cache with LRU policy. The purpose of instruction buffer and data cache is not to increase latency and bandwidth but to increase energy efficiency by reducing the number of memory accesses.

## 5 Evaluation Methodology

We compared Pasithea-1 with a RISC-based reference system to evaluate energy efficiency, performance and area. Table 4 lists the used benchmark programs.

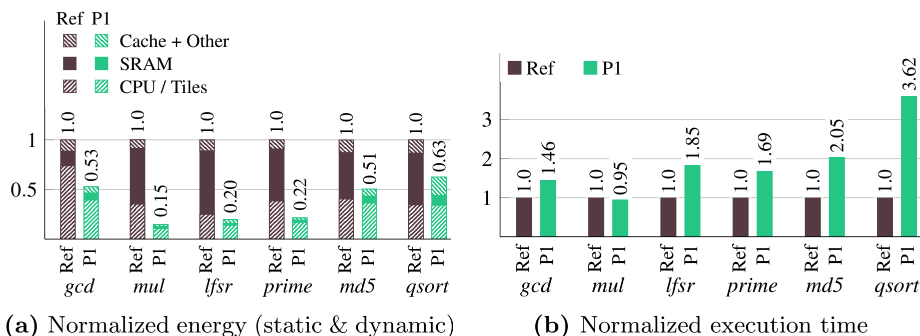
The open-source RISC-V core Ibex [24, 25], configured for 32-bit integer and compressed instructions (RV32IC), was used as reference system CPU. The reference system integrates Ibex with the instruction buffer, data cache and memory of Pasithea-1. Instruction fetching using the instruction buffer and data cache hits require no wait cycles. Data cache misses incur one wait cycle.

SystemVerilog register-transfer-level models of Pasithea-1 and the RISC-V reference system were used as basis for physical implementation. Both systems use hierarchical clock gates to reduce switching activity. A 256 kB SRAM serves as main memory for both systems.

Both systems were synthesized, placed and routed in GlobalFoundries 22 nm FD-SOI CMOS [26]. For this purpose, Synopsys Design Compiler and IC Compiler II were used. Ultra-low-leakage standard cells based on high-$V_{\mathrm{T}}$ transistors were used to minimize the impact of static power.

**Table 4:** Benchmarks. (For Pasithea-1, all benchmarks were written in IAL.)

| | | | | | | Pasithea-1 | | RISC-V | |
|---|---|---|---|---|---|---|---|---|---|
| **Name** | **Description** | **Data mem.** | **Subroutines** | **Recursion** | **Coroutines** | **Code Size** | **Instrs. / Frag.** | **Code Size** | **Impl. in** |
| *gcd* | Euclid's algorithm | | | | | 44 | 9 | 18 | asm |
| *mul* | shift-and-add multiply | | | | | 40 | 9 | 28 | asm |
| *lfsr* | Galois LFSR & coroutines | | ✓ | | ✓ | 132 | 10, 9, 7 | 104 | asm |
| *prime* | primality test | | ✓ | | | 360 | 42, 9, 20 | 178 | asm + C |
| *md5* | Message Digest 5 | ✓ | | | | 448 | 30, 59 | 246 | C |
| *qsort* | Quicksort | ✓ | ✓ | ✓ | | 204 | 27, 15 | 86 | C |




**(a)** Normalized energy (static & dynamic) **(b)** Normalized execution time

**Fig. 4:** Benchmarking results for reference system (Ref) and Pasithea-1 (P1)

## 6 Results

Based on timing-annotated netlist simulation and vector-based power analysis of the placed-and-routed designs, Figure 4a compares energy use of Pasithea-1 with that of the reference system. Static power accounted for 0.6 % and 4.4 % of total power for the reference system and Pasithea-1, respectively. Geometric means over the presented benchmarks reveal a 10.1× improvement of energy efficiency in the memory hierarchy and a 3.1× overall improvement.

Static timing analysis revealed maximum clock frequencies of 40 MHz for Pasithea-1 and 52 MHz for the reference system. Figure 4b compares execution times of the benchmarks at maximum clock speeds. In execution time, the reference system outperforms Pasithea-1 by a factor of 1.78. This can be attributed to latencies in global (FIM) and memory operations, which were not optimized during design.

Excluding SRAM, logic cell areas were 8932 $\mu m^2$ for the reference system and 244,121 $\mu m^2$ (27.3× larger) for Pasithea-1. This area disparity shrinks to 1.8× when the SRAM ($\sim$ 300,000 $\mu m^2$) is included.

## 7 Discussion

As shown in Section 6, Pasithea-1 achieves extraordinarily low switching rates and surpasses the RISC reference in energy efficiency considerably. Its easy low-level programming interface, demonstrated in Section 3, promises to make this energy efficiency available to a wide range of applications. A compiler backend for Pasithea-1 is currently under development and will enable further research into the viability and tradeoffs of using CGRA for general-purpose computing.

Clock frequency and per-cycle performance are weaknesses of the presented CGRA. Per-cycle performance was not optimized for in the presented microarchitecture and can likely be improved in future design iterations. To compete with general-purpose von Neumann processors in single-thread performance, higher clock frequencies must be supported using faster logic cells. Naively speeding up

all cells, e.g. by replacing them with lower-$V_T$ versions, would incur an unacceptably large static power penalty. To achieve higher performance while maintaining energy efficiency, techniques such as fine-grained body biasing could be used to dynamically provide speed where needed and reduce static power in other parts of the design.

With growing array sizes, we predict tile-tile and tile-memory data transfers to dominate energy efficiency and performance, likely encouraging dynamic optimization of tile and cache allocation. In the long term, further scaling of the presented approach could allow large fractions of complex programs to remain largely stationary in CGRA fabric.

**Acknowledgements** We thank GlobalFoundries for access to their 22 nm FD-SOI technology and EUROPRACTICE for providing design tools.

## References

1. Taylor, M.B.: A Landscape of the New Dark Silicon Design Regime. IEEE Micro 33(5), 8–19 (2013)
2. Patterson, D.: The Future of Computer Architecture. A white paper prepared for the Computing Community Consortium committee of the Computing Research Association (2008)
3. Cardoso, J.M.P., *et al.*: Compiling for reconfigurable computing. ACM Computing Surveys 42(4), 1–65 (2010)
4. Wang, D.K., and Kim, N.S.: DiAG: a dataflow-inspired architecture for general-purpose processors. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. ACM (2021)
5. Sankaralingam, K., *et al.*: Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. (2003)
6. Burger, D., *et al.*: Scaling to the end of silicon with EDGE architectures. Computer 37(7), 44–55 (2004)
7. Swanson, S., *et al.*: The WaveScalar architecture. ACM Transactions on Computer Systems 25(2), 1–54 (2007)
8. Baumgarte, V., *et al.*: PACT XPP — A Self-Reconfigurable Data Processing Architecture. The Journal of Supercomputing 26(2), 167–184 (2003)
9. Gobieski, G., *et al.*: SNAFU: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In: 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA). IEEE (2021)
10. Ozaki, N., *et al.*: Cool Mega-Arrays: Ultralow-Power Reconfigurable Accelerator Chips. IEEE Micro 31(6), 6–18 (2011)
11. Venkatesh, G., *et al.*: Conservation cores. ACM SIGPLAN Notices 45(3), 205–218 (2010)
12. Mishra, M., *et al.*: Tartan: evaluating spatial computation for whole program execution. ACM SIGOPS Operating Systems Review 40(5), 163–174 (2006)

13. Lysecky, R., *et al.*: Warp Processors. ACM Transactions on Design Automation of Electronic Systems 11(3), 659–681 (2006)
14. Watkins, M.A., *et al.*: Software transparent dynamic binary translation for coarse-grain reconfigurable architectures. In: 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA). IEEE (2016)
15. Souza, J.D., *et al.*: A Reconfigurable Heterogeneous Multicore with a Homogeneous ISA. In: Proceedings of the 2016 Conference on Design, Automation & Test in Europe. DATE '16, pp. 1598–1603. EDA Consortium, Dresden, Germany (2016)
16. Brandalero, M., *et al.*: Multi-Target Adaptive Reconfigurable Acceleration for Low-Power IoT Processing. IEEE Transactions on Computers 70(1), 83–98 (2021)
17. Henry, D., *et al.*: The Ultrascalar processor-an asymptotically scalable superscalar microarchitecture. (1999)
18. Gunadi, E., and Lipasti, M.H.: CRIB: consolidated rename, issue, and bypass. ACM SIGARCH Computer Architecture News 39(3), 23–32 (2011)
19. Waterman, A., and Asanović, K.: The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2019121. Tech. rep., RISC-V Foundation (2019)
20. Séquin, C.H., and Patterson, D.A.: Design and Implementation of RISC I. In: Proc. Advanced Course on VLSI Architecture (1982)
21. Spivey, J.M.: Fast, accurate call graph profiling. Software: Practice and Experience 34(3), 249–264 (2004)
22. De Moura, A.L., and Ierusalimschy, R.: Revisiting coroutines. ACM Transactions on Programming Languages and Systems 31(2), 1–31 (2009)
23. Kim, C., *et al.*: Composable Lightweight Processors. In: 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), pp. 381–394. IEEE (2007)
24. Schiavone, P.D., *et al.*: Slow and steady wins the race? A comparison of ultra-low-power RISC-V cores for Internet-of-Things applications. (2017)
25. lowRISC Contributors: Ibex RISC-V Core, (2022). https://github.com/lowrisc/ibex
26. Carter, R., *et al.*: 22nm FDSOI technology for emerging mobile, Internet-of-Things, and RF applications. In: 2016 IEEE International Electron Devices Meeting (IEDM). IEEE (2016)