



UvA-DARE (Digital Academic Repository)

On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges

Poss, R.C.

Publication date
2012

[Link to publication](#)

Citation for published version (APA):

Poss, R. C. (2012). *On the realizability of hardware microthreading. Revisiting the general-purpose processor interface: consequences and challenges*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

Chapter 4

Machine model and hardware interface

Abstract

This chapter introduces the machine model and the machine interfaces offered by various implementations of hardware microthreading. We present the general concepts common to all implementations, then review how design choices may lead to different machines interfaces. We identify major “generations” of interfaces and introduce the low-level assembly languages available to program the various implementations.

Contents

4.1	Overview	60
4.2	Concepts	60
4.3	Semantics	63
4.4	Out of band control bits	71
4.5	Interactions with a substrate ISA	72
4.6	Faults and undefined behavior	75
4.7	Specific implementations and their interfaces	77
4.8	Assembler and assembly language	79
	Summary	80

4.1 Overview

This chapter is organized as follows:

- section 4.2 presents the general concepts underlying the machine model, and section 4.3 explores the various semantics that can be attached to these concepts in specific implementations;
- section 4.4 explains how the scheduling hints and thread termination events can be provided by programs;
- section 4.5 reviews how the introduction of hardware microthreading in an existing ISA interacts with the substrate ISA’s features;
- section 4.6 outlines potential behaviors for unplanned or erroneous situations;
- section 4.7 introduces actual implementations and their specific interfaces, and recognizes three major “generations” of implementations. Section 4.8 then reviews the concrete assembly languages for these implementations.

4.2 Concepts

- Regardless of the specific implementation choices, a hardware microthreaded architecture provides the following:

thread contexts, logical threads and thread programs.

Thread contexts are management structures in the Thread Management Unit (TMU) that define individually scheduled instruction streams in the microthreaded pipeline. They are analogous to the “threads” of [II11a, II11b] and existing threading APIs (e.g. POSIX [Ins95]), the “workers” of OpenMP [Ope08] or the “harts” of [PHA10].

Logical threads are the individual units of work defined by programs via the bulk creation event, using a common starting PC and a logical index range. The TMU maps logical threads sequentially over one or more thread contexts. Once activated, logical threads execute a *thread program* from the initial PC to completion. As such, logical threads are akin to the “blocks” of Grand Central Dispatch (GCD) [Sir09, App] or the “tasks” of OpenMP and Chapel [CCZ07].

Thread contexts are independently scheduled, and may interleave while they are active. However, there is no interleaving of logical threads within single thread contexts. Only logical threads are visible to programs; programs can control thread contexts only indirectly via the *placement* of work, discussed below.

thread program actions and synchronization.

Thread programs contain instructions that perform *actions* on the environment. Each instruction has some encoding which specifies its *input and output operands*. The encoding is assumed to derive from a RISC ISA. In particular, operands are encoded either as immediate *values* or as a register *name*, which is a *fixed* offset into a local storage space.

Compared to a conventional register machine, where this storage space would be general-purpose memory cells in hardware, i.e. physical registers, the microthreaded core maps instruction operands to *dataflow synchronizers*. These implement I-variables: a storage cell which may either contain a value (“full”) or be “waiting” on a value not yet available [ANP87]. These are then used to provide asynchrony between individual instructions, as a long-latency operation can now simply set its output operand to

“waiting” instead of actually stalling the processor. In other words, the microthreaded machine model subsumes existing instruction sets by substituting dataflow synchronizers for registers. This design was originally proposed in Denelcor’s HEP [Smi81] and Tera’s MTA [SCB⁺98] (later Cray’s XMT).

However, *memory outside of the core is not synchronizing*: the *indirect* storage accessible via “load” and “store” instructions behaves as regular data cells which always hold a value, i.e. do not synchronize. This is where the design diverges conceptually from the HEP and the MTA, both of which provide synchronization on the entire memory space. The rationale for this stems from the observation that negotiating synchronization is a communication activity, and that in general-purpose environments the software implementer is most competent to recognize computation patterns and organize communication. By restricting implicit synchronization to core-local structures, the machine interface defers the responsibility to organize non-local activities, including arbitrary patterns of inter-core synchronization, to explicit remote synchronizers read/write operations controlled by software.

synchronous vs. asynchronous operations.

The existence of dataflow synchronizers allows us to distinguish between:

- *synchronous* operations, which complete with a full output operand. These guarantee that any further instruction using the same operand as input will not suspend.
- *asynchronous* operations, which complete with a non-full operand in the issuing thread, while letting the operation run concurrently. These can be said to return a “future”¹ on their result.

bounded and unbounded operation latencies.

The existence of asynchronous behavior mandates a comment on operation latency, necessary to determine conditions to *progress* in executing programs.

In the proposed design we can distinguish between:

- operations with *bounded latency*: once the operation is issued, it is guaranteed to complete within a finite amount of time regardless of non-local system activity;
- operations with *unbounded latency*: *whether* the operation completes at all is dependent on non-local system activity, e.g. data-dependent branches in the control flow of other threads.

Intuitively, computation operations issued by programs should have a bounded latency, especially arithmetic and control flow. Yet some unbounded latencies become possible depending on the choice of semantics for concurrency management, discussed in section 4.3 below. In general, memory loads and stores, FPU operations, remote synchronizer reads/writes, and bulk creation *after* allocation of a creation context have a bounded latency; whereas requests to allocate new concurrency resources and requests to synchronize on termination of another thread may be indefinitely delayed by non-terminating threads, and these operations thus have an unbounded latency.

local wait continuations, remote wake ups.

The synchronizing storage is “local” to the core’s pipeline and its state is maintained by the flow of local instructions. In particular, *only local instructions can suspend on a*

¹The concept of “future” is introduced in [Hal85]: “The construct (**future** *X*) immediately returns a future for the value of the expression *X* and concurrently begins evaluating *X*. When the evaluation of *X* yields a value, that value replaces the future.”

dataflow synchronizer. This choice guarantees that wake-up events to waiting threads are always resolved locally upon writes to synchronizers.

Yet, synchronizers can be accessed remotely across the NoC. For example, a thread running on one core may write remotely to another core's synchronizers and wake up any threads waiting on it on that other core.

virtual mapping of the synchronization space, dataflow channels.

The fixed offsets in instruction operands do not have a static mapping to synchronizing storage; instead, the set of *visible dataflow synchronizers* can be *configured per thread* during bulk creation, and define a “window” on the synchronizing storage. In particular:

- if a synchronizer is visible from only one thread it is said to be “private,” or “local,” to that thread. Since values stored in it are subsequently readable by further instructions, it behaves functionally like a regular general-purpose register.
- two or more threads can map some of their register names to the same synchronizers. When this occurs, the configuration can be said to implement a *dataflow channel* between the threads: a “consumer” instruction in one thread will synchronize with a “producer” instruction in a different thread. Also, “consumer” instructions which read from a dataflow channel may have an unbounded latency as per the definition above.

bulk creation contexts.

Programs use bulk creation of logical threads to define work. Yet bulk creation is itself a multi-phase *process* which involves e.g. reserving thread contexts in hardware and effecting the creation of logical threads (cf. section 3.3.5). This process itself has both input parameters and internal state. Its parameters are the initial PC, a logical index range, and potentially configuration information for the mapping of register names to synchronizers. Its internal state tracks which logical threads have been created and terminated over time.

Both the parameters and internal state must be kept live *until all logical threads have been created*, especially when some thread contexts must be reused for successive logical threads. As such they constitute collectively a *bulk creation context* which must be allocated, configured and managed.

dataflow synchronizer contexts.

During bulk creation, the TMU can allocate a subset of the synchronizing data storage to map it in the visible window of new threads. Conversely, synchronizers can be released when threads terminate if they are not mapped elsewhere. The set of dataflow synchronizers allocated to a group of bulk created threads thus forms their *dataflow synchronizer context*, a resource that must be managed jointly with the threads.

bulk synchronizers.

Programs use bulk synchronization to wait on termination of threads. As discussed in section 3.3.5 this implies space to store “what to do on termination” for the group of threads waited upon, and a semaphore. These *bulk synchronizers*, separate from the dataflow synchronizers, also constitute state which must be allocated, configured and managed.

automatic work placement and distribution.

When issuing bulk synchronizations, programs can define that the work is to be created either locally or remotely, and on either one or multiple cores. This *placement information* is provided early and serves to route the bulk creation event to the appro-

appropriate TMU on chip. When targeting multiple cores in one request, multiple TMUs cooperate to spread the logical index range over the cores.

Beyond the creation of threads, the TMUs also assist with the distribution of data. A single event can be sent after bulk creation to broadcast a value to the synchronizers of multiple cores, to serve as input to the logical threads.

binding of logical threads to cores.

In the proposed design, there is no mechanism provided to migrate the contents of management structures across cores after they are allocated. Since the state of synchronizers is not *observable* in software either, this implies that *logical threads cannot be migrated* to another core after they are created.

4.2.1 Families of logical threads

From the previous concepts, we can derive the notion of “*family*” to designate the set of logical threads that are created from a single bulk creation.

Like logical threads, families are defined by programs and *only exist indirectly* through the bulk creation contexts, thread contexts and bulk synchronizers allocated and managed for them in hardware. Although it merely designates an abstract concept, the term “family” forms a useful shorthand for the collective work performed by a bulk created set of logical threads. It is used with this meaning in the remainder of our dissertation and other literature about hardware microthreading.

4.3 Semantics

For the TMU to be *programmable*, some set of *semantics* must be associated to its state structures and their control events. Semantics establish the relationships between state structures and how state evolves at run-time in response to events. To define these semantics, previous research on hardware microthreading has explored three mostly orthogonal aspects:

- how to configure and trigger bulk creation and bulk synchronization (section 4.3.1);
- how to manage the mapping of logical threads to cores and thread contexts (section 4.3.2);
- how to manage the mapping of logical threads to dataflow synchronizers (section 4.3.3).

4.3.1 Bulk creation and synchronization

- The organization of bulk creation and synchronization determines primarily the relationship between bulk creation contexts, bulk synchronizers and thread contexts. We do not consider here how many thread contexts are allocated and how many logical threads are created over them, this will be covered in section 4.3.2.

Any choice of semantics must respect the following dependencies:

- a bulk creation context must exist before the allocation of thread contexts and the creation of logical threads starts;
- a bulk creation context must persist until all logical threads have been created;
- a bulk synchronizer must exist before:
 - the earliest point at which another thread may issue a request for bulk synchronization;

- the first logical thread is created;

whichever comes first;

- a bulk synchronizer must persist until after:
 - the latest point at which another thread may request bulk synchronization;
 - all threads have terminated and a requesting thread is notified;
 - the bulk synchronizer is explicitly released;

whichever comes last.

Then any choice of semantics must select the *interface* to use in programs, i.e. :

- what machine instruction(s) trigger these processes;
- what parameters can be provided and how;
- under which condition a bulk creation or bulk synchronization request synchronizes with the issuing thread;
- what “output” either of these requests returns to the issuing thread.

In our work, all implementations provide at least the following primitives:

allocation of a bulk creation context.

This takes as input some placement information, and performs the allocation of *both* a bulk creation context, a bulk synchronizer, a thread context and a set of synchronizers, on the target core(s). It is an asynchronous operation which produces a future on an *identifier to the bulk creation context* to the requesting thread. Its latency is discussed in section 4.3.1.1 below.

The reason why the allocation is combined is to guarantee that once the initial allocation succeeds, it will always be possible to create logical threads and synchronize on their termination afterwards.

configuration of bulk creation.

This takes as input an identifier to a bulk creation context and writes a value in the corresponding hardware structures, possibly remotely. It is an asynchronous operation with bounded latency and no result.

Implementations then differ in how they deal with allocation failures, and how they trigger bulk creation, synchronization and resource de-allocation. We discuss these further in sections 4.3.1.1 and 4.3.1.2.

4.3.1.1 Allocation failures

- We have found three main classes of interfaces that differ in how they deal with allocation failures:
 - in a *suspending* interface, allocation always suspends until resources become available. In these semantics, allocation always *appears* to succeed from the perspective of running threads but may have an unbounded latency.
 - in a *soft failure* interface, allocation failures cause a failure code to be reported as a value to the requesting thread with a bounded latency, for handling by the requesting thread program. This allows the thread program to opt for an alternate strategy, e.g. allocation with different placement parameters or serializing the work.

- in a *trapping* interface, allocation failures are handled within a bounded latency as a fault, and trigger a trap to be handled by a “fault handler” separate from (and possibly invisible to) the thread program.

We found implementations providing both suspending and soft failure interfaces, as discussed in chapter 10. The latter trapping interface is theoretical as of this writing, yet we believe it will become relevant in future work where issues of placement are managed by system software separate from application code.

4.3.1.2 Fused vs. detached creation

- We found two main classes of implementations that differ in how they trigger bulk creation and synchronization:

- in a *fused creation* interface, bulk creation, bulk synchronization and resource de-allocation are fused in a single asynchronous operation:

fused creation.

This takes as input an identifier to a bulk creation context and an initial PC. After issue, the operation triggers the start of logical thread creation, and *also* requests bulk synchronization and resource release on termination of all logical threads. The request for bulk synchronization binds the output operand of the fused creation operation with the bulk synchronizer.

The operation thus produces a *future on termination* of the logical threads, with an unbounded latency. All the bound resources are de-allocated automatically after completion is signalled.

- in a *detached creation* interface, bulk creation, bulk synchronization and resource de-allocation become independent asynchronous operations:

creation.

This takes as input an identifier to a bulk creation context and an initial PC. After issue, the operation triggers the start of extra context allocation and logical thread creation. As soon as logical thread creation starts (which may be later than the issue time, due to network latencies), an identifier for the bulk synchronizer is returned to the issuing thread.

The creation operation thus produces a *future on the start of creation* with a bounded latency. The latency is bounded because the prior allocation guarantees that creation is always possible.

synchronization on termination.

This takes as input an identifier to a bulk synchronizer. After issue, the operation binds its output operand to the named bulk synchronizer.

It thus returns a *future on termination* of the logical threads with an unbounded latency.

de-allocation.

This takes as input an identifier to a bulk creation context. It has no output and completes with a bounded latency. The operation triggers either de-allocation of the resources if the work has completed, or automatic de-allocation on completion if the work has not completed yet.

This interface does not define the behavior when a synchronization request is issued after a de-allocation request (there is then a race condition between thread

termination and the bulk synchronization request); however, it guarantees that bulk synchronization is signalled before de-allocation if it was registered first.

4.3.2 Mapping of logical threads to cores and contexts

The design proposes to minimize the *necessary* amount of placement information to provide during allocation, while providing extra control to thread programs when desired. The placement information can specify “where” on the chip to allocate in terms of cores, then “how much” on each core to allocate in terms of thread contexts, then “how to spread” the logical index range over the selected thread contexts.

4.3.2.1 Mapping to cores

- To select the target core(s), the following parameters are available to programs:

inherit. Also called “default” placement in previous work. The placement information that was used to bulk create the issuing thread is reused for the new bulk creation request.

local. The bulk creation uses only the core where the issuing thread is running.

explicit. An explicit parameter is given by the program to target a named core, or cluster, on the chip. The format of explicit placements then depends on the implementation, although recent research has converged to provide a system-independent addressing scheme (cf. chapter 11).

These parameters are intended to control locality and independence of scheduling between application components.

4.3.2.2 Selection of thread contexts

- To select thread contexts, a single parameter called “block size” or “window size,” controls the *maximum number of thread contexts* to allocate *per core* during bulk creation. This parameter is optional; a program may choose to leave it unspecified, in which case the hardware will select a default value. This parameter bounds per-core concurrency, which is useful to control the working set size, and thus cache utilization, on each core. It can also help control utilization of the synchronizer space as explained in [CA88].

In the implementations we have encountered, an undefined block size selects *maximum concurrency*, i.e. allocate all thread contexts available on each core, or the number of logical threads to execute per core, whichever is lower. However, we are also aware of ongoing discussions to change this default to a value that selects the number of thread contexts based on the local cache sizes, to minimize contention. The choice of default behavior should thus be considered implementation-dependent.

4.3.2.3 Distribution of the logical indexes

- In the proposed interface, a requesting thread can specify an index range by means of *start*, *limit* and *step* parameters (side note 4.1). Compared to a potentially simpler interface which would only allow programs to set the number of logical threads, this interface enables direct iteration over data sets, in particular the addresses of array items in memory. These parameters are optional, and default to 0, 1, 1, respectively.

Side note 4.1: Logical index sequence.

The logical thread index sequence is defined from the *start*, *limit*, *step* parameters by using a counter initialized with *start* and increased with *step* increments until it reaches *limit*. *limit* is excluded from the sequence.

In the implementations we have encountered, no further control is provided to programs over the distribution of logical threads. The index space is spread evenly over all selected cores. On each core, thread contexts “grab” logical thread indexes on a first-come, first-served basis until the local index pool is exhausted. Again, we are aware of ongoing discussions to provide more control to programs to facilitate load balancing, a topic which we revisit in chapter 13. The interfaces available to programs to control logical distribution should thus be considered an implementation-dependent design choice, orthogonal to the general concepts of hardware microthreading.

In each logical thread created, the first synchronizer is pre-populated by the TMU with the value of the logical index.

4.3.3 Mapping of threads to synchronizers

As explained in section 3.3.3, the proposed architecture uses a dynamic mapping between operand addresses in instructions and the synchronization space. *How many* synchronizers are visible to each thread, and *which* synchronizers are visible, are both configurable properties of the thread context, fed into the pipeline for reading instruction inputs when each instruction is scheduled.

4.3.3.1 Interface design requirements

- The question then comes how to design an interface to configure this mapping. Here two forces oppose: on the one hand, maximum flexibility for software would mandate the opportunity for arbitrary mappings; on the other hand, the complexity of the mapping parameters impacts logic and energy costs in the hardware implementation.

Any trade-off when designing an interface must consider the following:

- for the design to be truly general-purpose, the opportunity must be given to programs to specify at least two private (non-shared) synchronizers per thread. This is the theoretical minimum necessary to carry out arithmetic and recursion within a thread;
- any configuration that shares synchronizers between threads should establish clear conditions for the eventual release of the synchronizers, lest their management becomes a source of tremendous complexity in compilers;
- a key feature of the design is the short latency bulk creation of logical threads over thread contexts. Any design which requires the hardware to modify the mapping of synchronizers between logical threads, or maintain heterogeneous mapping information across thread contexts, will impact the latency of thread creation.

Another aspect is the selection of *where the configuration is specified*. When considering only performance and implementation costs, the cheapest and most efficient choice is to let the thread that issues a bulk creation provide the mapping parameters explicitly. Yet this choice would be shortsighted. From a programmability perspective, a clear advantage of the architecture is the ability to compose separately written thread programs via bulk

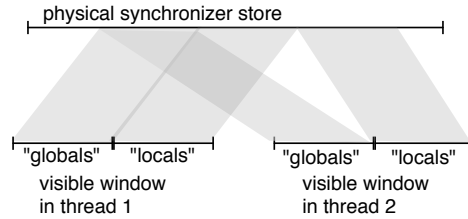


Figure 4.1: Example window mapping for two sibling threads with global and local synchronizers.

creation. The knowledge of how many synchronizers are visible, and whether and how they are shared between threads, is needed to assemble instruction codes. It thus belongs with the code generator for the threads being created, not the threads doing the creation. Therefore, an implementation should associate this configuration information with the thread programs that are the target of bulk creation somehow.

4.3.3.2 Common semantics

- We could find the following consensus throughout implementations:
 - *The configuration information immediately precedes the first instruction of a thread program.* This answers the argument above, and provides good locality of access to the creation process.
 - *Thread programs can specify a number of private synchronizers.* This answers the generality argument above. The fixed number is then allocated for each participating thread context and is reused by all successive logical threads running on these contexts. These synchronizers are dubbed “local” in the remainder of our dissertation.
 - *Thread programs can specify a number of synchronizers to become visible from all threads.* This is meant to provide common data to all threads. That fixed number of synchronizers is mapped onto the visible window of all participating thread contexts on that core. These synchronizers are henceforth dubbed “globals”; they are “global” to all logical threads created from a single bulk creation. We also call them “global dataflow channels” when they are used strictly for broadcasting.

An example is given in fig. 4.1.

4.3.3.3 Heterogeneous sharing patterns

- Besides the “global” pattern introduced above, prior work has explored other forms of synchronizer sharing between threads. The implementations we have used have proposed the following features, in various combinations:

“shared” synchronizers between adjacent thread contexts.

With this feature, thread programs can specify a number of extra synchronizers to share between neighbouring contexts during bulk creation. This creates the overlap pattern illustrated in fig. 4.2. This feature is coordinated by the TMU so that logical thread indexes are distributed accordingly, and a point-to-point communication chain

Side note 4.2: Purpose and motivation of “shared” synchronizers.

This feature was originally proposed to explore whether bulk created microthreads are a suitable alternative to dependent sequential loops, when the loop-carried dependency has stride 1. We revisit this in section 13.8.

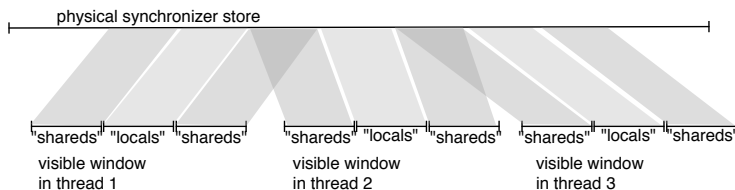


Figure 4.2: Example window mapping for three sibling threads with local and shared synchronizers.

is created in the logical thread order. The mapping is heterogeneous because the “border” thread contexts do not necessarily share synchronizers (cf. below).

When this pattern is used only to express a forward-only dependency chain, we call the synchronizers “shared dataflow channels.” When necessary, from the perspective of a given thread we distinguish further between “D” synchronizers which are shared with the preceding context, and “S” synchronizers shared with the succeeding context.

“hanging” vs. “separated” global synchronizers.

In the “separated” variant, the global synchronizers of new threads are freshly allocated from the synchronizing storage and initialized to the “empty” state upon bulk creation, on each participating core. A dedicated NoC message is then available to programs to broadcast a value to the participating cores explicitly.

In the “hanging” variant, if the core where the thread issuing a bulk creation is also a target of the creation, then on that core the global synchronizers of new threads are not freshly allocated; instead, the window of the new threads is configured to map to some *existing* private synchronizers of the thread that issued the bulk creation. Here, the interface allows a creating thread to indicate which of its local synchronizers to use as a base offset for the created windows.

Then, if there are more cores participating, fresh synchronizers are allocated on the remaining cores. The values stored in the synchronizers on the first core are automatically broadcasted to the other cores during bulk creation.

The “hanging” variant results in higher utilization of the synchronizing store on the first core. It was historically the first implemented. We can see it was primarily designed for single-core systems and creates a strong resource dependency between the creating thread and the created threads. The “separated” variant is more flexible and homogeneous when multiple cores are involved.

We illustrate the difference between “hanging” globals and the “separated” alternative in fig. 4.3.

“hanging” vs. “separated” shared synchronizers.

As with “separated” globals above, with “separated” shares the shared synchronizers of new threads are freshly allocated and initialized to “empty.” The “leftmost” synchronizers in the first thread context may then be subsequently written asynchronously by the issuing thread using a dedicated NoC message.

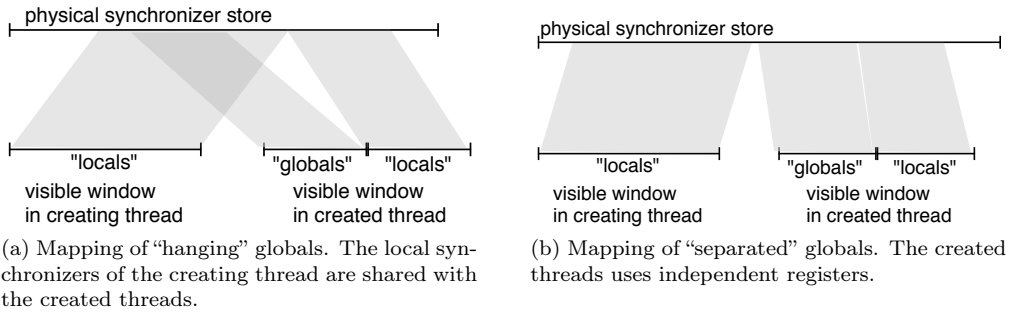


Figure 4.3: Alternatives for mapping global synchronizers.

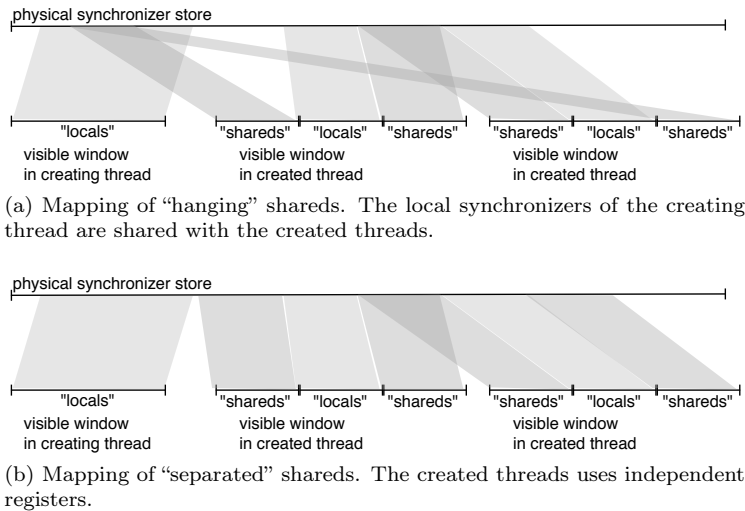


Figure 4.4: Alternatives for mapping shared synchronizers.

With “hanging” shares, as with “hanging globals” above, a special case exists if the core where a bulk creation is issued from is also participating. In this case the shared synchronizers of the border contexts are not freshly allocated from the synchronizing storage; instead, their visible windows are configured to map to some existing private synchronizers of the issuing thread.

As above, the “hanging” arrangement was the first designed, with single-core execution in mind. We illustrate the difference between “hanging” shares and the “separated” alternative in fig. 4.4.

4.3.3.4 Layout of the visible window

- Once a program has determined *which* synchronizers are visible, a choice exists of *where* in the visible window they should be mapped. For example, if a thread program configures e.g. 4 local synchronizers and 7 globals, it is possible to map offsets 0 to 3 to the locals, and offsets 4 to 10 to the globals. It is also possible to map offsets 0 to 6 to the globals, and offsets 7 to 10 to the locals.

We can identify this *mapping order*, using a string of the form “X-Y-Z...”, where X, Y, Z... indicate the type of synchronizer mapped and the order in the sequence indicates the order of mapping in the visible window. For example, some early implementations used the order G-D-L-S, meaning that global synchronizers were mapped first (if any), followed by the synchronizers shared with the previous context, followed by the local synchronizers, followed by the synchronizers shared with the next context.

- From the hardware designer’s perspective, this choice seems neutral. However, here we discovered an extra requirement from the programmability perspective. Indeed, considering a common software service that uses only local synchronizers (say, a routine to allocate heap memory), it proves useful to be able to reuse the same code for this service from multiple thread contexts with different numbers of “global” and “shared” synchronizers. However, the address of synchronizers are statically encoded in the instruction codes; this implies that *the mapping of local synchronizers must be identical regardless of the number of other synchronizer types*. To satisfy this, we have required that local synchronizers are always mapped first in the visible windows of threads. Following our requirement the implementations have been subsequently aligned to the order L-G-S-D.

4.4 Out of band control bits

- Introduced in section 3.2.1, the instruction *control bits* indicate to the fetch unit when to *switch* threads and when to *terminate execution* of a thread. These bits must be defined for every instruction processed through the pipeline; there are three possible design directions to program them.

The direction least intrusive for existing software, that is, *able to preserve an existing code layout from a pre-existing ISA*, is also the most expensive in hardware: maintain two PCs on the fetch unit, where one refers to code and the other to the control bits in a separate area in memory. In this approach, the I-cache is shadowed by a cache for control bits, or “C-cache.” Thread creation events should specify both the initial PC for instruction and for control bits. Alternatively, the TMU can read the control bit PC by looking up thread metadata at a fixed offset from the initial instruction PC. The fetch unit must further place threads on the waiting list on either an I-cache miss, or a C-cache miss. The extra complexity of this approach comes from the synchronization of waiting threads: upon I-cache refills, a thread can only migrate from the “waiting” state to “active” if the C-cache hits, and *vice-versa*.

The most contrasting approach is to make the control bits part of the machine instruction encoding, for example using two leading bits in the instruction format. With variably sized instruction words, an existing ISA can be extended this way, however with any fixed-width ISA (such as found in RISC designs) this approach requires to redesign the ISA because some existing instructions may have no unused bits left. In the latter case, this approach is also the most intrusive as it prevents reusing existing assembly code and compilers as-is.

The implementations we have encountered opt instead for an intermediate approach: interleave control bits and machine instructions. In this approach, each group of N instructions is immediately preceded by $2N$ control bits. To ensure that the fetch unit can always read the control bits, these must be present in the same I-cache line as the controlled instructions. This in turn constrains the minimum I-cache line size. With 32-bit instruction formats, the “sweet spot” which incurs no wastage of space in I-lines while preserving a size power of two lies at $N = 15$, with a minimum I-cache line size of 64 bytes. We detail the optimal parameters in Appendix B.

Side note 4.3: Thread switching not specified by control bits.

Besides where specified by control bits, thread switches are also incurred automatically for instructions laying on the boundary of I-cache lines. This avoids missing the I-cache during fetch (side note 3.3), but also helps guarantee fairness of scheduling between active threads. Switches are also incurred upon branches for fairness. Yet, for efficiency it is recommended to still place a “switch” annotation on the branch instruction, as this will ensure the next pipeline slot is filled by an instruction from another thread and avoids a one cycle pipeline bubble if the branch is taken.

Side note 4.4: Switching and thread termination as instructions.

Another approach may consider using explicit instructions for thread switching and termination. An instruction can signal a control event prior to the instruction that immediately succeeds it. This requires embedding the few gates necessary to decode the instruction within the fetch stage itself. Alternatively, an instruction can signal the event for the end of the next cycle. In this case, another instruction that immediately follows would still be executed before the event is effected.

In either case, this solution incurs a pipeline bubble at every control event: the instruction occupies an idle cycle that does not contribute to useful computations. It is thus only interesting if switching and thread termination are uncommon. This solution is thus undesirable with fine-grained interleaving, e.g. where threads replace inner loop bodies and where memory loads incur switching, such as with the proposed design.

To program this latter approach, the assembler program must emit a control word at regular intervals between instructions. This is covered below in section 4.8.

4.5 Interactions with a substrate ISA

The proposed architecture concepts can be applied either to new RISC cores with a dedicated ISA, or *extend* existing RISC cores, and thus to pre-existing ISAs. In the latter case, the ISA is a *substrate* upon which the extensions are built. When working with a pre-existing ISA, its semantics interact with the proposed micro-architecture. We list the most notable possible interactions below.

4.5.1 Delayed branches

- Delayed branches, as found in the MIPS, SPARC and OpenRISC² ISAs, have been designed for in-order, single-threaded pipelines to reduce the cost of control hazards: the instruction(s) immediately succeeding a branch execute(s) regardless of whether the branch is taken; the corresponding positions in the instruction stream are called *delay slots*. With “branch and link” instructions, where the PC of the first instruction after the branch is saved to another register, the first address *after* the delay slot(s) is used.

When an architecture using delay slots evolves to use out-of-order execution, e.g. via the introduction of superscalar execution or pipeline multithreading, the “next instruction” in instruction stream order may not immediately succeed the branch in the pipeline anymore, and extra logic must be introduced to ensure that this instruction completes even when the branch is taken. With hardware multithreading, this implies maintaining a separate “next PC” register for every thread.

²<http://opencores.org/or1k>

4.5.2 Predication

- Predication, as found in the ARM and IA-64 ISAs, is another feature designed to reduce the cost of control hazards: each instruction is predicated on shared *condition codes* updated by special instructions.

When an architecture using predication evolves to use out-of-order execution, extra logic must be introduced to ensure that the ordering of tests to the condition codes follow the ordering of updates. With hardware multithreading, this further implies that separate condition codes must be maintained for each thread.

4.5.3 Register classes

- Some ISAs rely on *separate register classes* for the operands and targets of certain instructions. Most processor designs use different register classes for integer and floating-point instructions; then ARM also has a separate class for “return addresses,” used implicitly by branch and link instructions, and SPARC has “status registers,” including the Y register used as implicit operand for multiplies and divides.

With hardware multithreading, separate instances of these registers must be available in each thread. Moreover, if these registers can become the target of long-latency operations, such as the Y register in SPARC, the ordering of dependent instructions cannot be controlled by the dataflow scheduling from section 3.2.1 unless these registers are also equipped with dataflow state bits.

4.5.4 Multiple register windows per thread

- We have outlined the role of sliding register windows in section 3.3.3. More generally, some ISAs define multiple register windows *per thread* and conditions to “switch” the instruction stream from one to another. For example, ARM defines that a separate register window is used upon traps, syscalls and interrupts.

With the introduction of hardware multithreading, different instances of the register windows must be available to each thread to avoid saving and restoring the registers to and from memory during context switches. The number of windows *per thread* does not change as the desired number of hardware threads increases, and becomes the growth factor for the register file. However, mere duplication would be wasteful, as only one window is active per instruction and per thread. Here two optimizations are possible.

When the ISA specifies that extra register windows are used only for asynchronous events, such as in ARM, it is possible to serialize all the asynchronous events from all threads over a smaller set of registers. In this scenario, if a trap handler is ongoing for one thread, a trap from another thread would suspend until the trap register window becomes available. While this approach may suggest reduced parallelism, it is quite appropriate for trap and syscall entry points which usually require atomic access to the core’s resources (e.g. to fetch exception statuses).

If the ISA defines explicit instructions to switch register windows, it is possible to replace these instructions by explicit loads and stores to exchange the registers to memory. This can be done either statically by substitution in a compiler (as we did, cf. Appendix H.9), or dynamically by a dedicated process in hardware. This trades the complexity of a larger register file for overhead in switching windows within threads, which can be tolerated using dataflow scheduling (section 3.2.2).

ISA	MIPS	SPARC	Alpha	ARM	PowerPC	OpenRISC
Delay slots	yes	yes	no	no	no	yes
Predication	no	no	no	yes	no	no
Register classes	2	3	2	3	2-6	2
Status register as implicit operand	no	yes	no	yes	yes	yes
Register windows	no	yes (8+)	no	yes (6)	no	no
Max. operands per instruction (input/output)	2/1	4/2 [†]	2/1	3/2 [‡]	2/1	2/1

[†] `std` has 4 inputs and 1 output; `ldd` has 2 inputs and 2 outputs.

[‡] dual multiplies have 3 inputs; long multiplies have 2 outputs.

Table 4.1: Features of existing general-purpose RISC ISAs.

4.5.5 Atomic transactions

- With RISC-style split-phase atomic transactions, for example the Load-link, Store-conditional (LL/SC) instruction pairs found in MIPS, PowerPC, Alpha and ARM, the semantics can be kept unchanged when the core is extended with multithreading, with no extra complexity. However, the rate of transaction rollbacks may increase significantly if the working set of all local threads does not fit in the L1 cache.

In contrast, if the ISA features single instruction atomics, in particular Compare-and-Swap (CAS) or atomic fetch-and-add found in most other architectures than the four already named, the memory interface must be extended with a dedicated asynchronous FU for atomics. Indeed, single-instruction atomics require locking the L1 line for the duration of the operation, and without an asynchronous FU, any subsequent load would cause the pipeline to stall until the CAS instruction is resolved.

4.5.6 Summary of the ISA interactions

This section has outlined the interactions between features of the substrate ISA and the introduction of hardware microthreading. Of common pre-existing general-purpose ISAs (table 4.1), the Alpha ISA is the one that requires the least added complexity when introducing microthreading. In our research, we have used implementations derived from both the Alpha and SPARC substrate ISAs:

- The original Alpha ISA uses neither delay slots, nor predication, nor sliding register windows, and uses only two symmetric register classes for integer and floating-point instructions, with no implicit operands in instructions and only one window per thread. It does not define CAS instructions but does feature LL/SC.
- The original SPARC ISA uses delay slots and sliding register windows, but no predication. It has three register classes, with implicit operands in integer multiply and divide (the Y register). It also defines CAS instructions, and 8 sliding windows per thread.

4.6 Faults and undefined behavior

- To increase understanding of the proposed interface, we found it useful to explore the “negative space” of the semantics. That is, provide information about situations that “should not occur” but still may occur due to programming errors, malicious code, etc. From the hardware interface perspective, we can *specify* two types of reactions upon an erroneous or unplanned situation in hardware.

If we specify that a situation constitutes a *fault*, we mean that it constitutes an error recognized by the implementation, and where execution is guaranteed to not progress past the fault. For example, after a fault occurs, either the entire system stops execution, or a *fault handler* is activated to address the situation and decide an alternate behavior.

If we specify *undefined behavior*, we mean a situation where the behavior of an implementation is left undefined, that is, an implementation may or may not test for the situation and execution may or may not progress past the situation. The characteristic of undefined behavior is *absence of knowledge about the behavior*, in particular that *no further knowledge about the state of the system can be derived from past knowledge after the situation occurs*.

We detail the situations we have found most relevant to programmability and validation in the following sections.

4.6.1 Invalid synchronizer accesses

4.6.1.1 Unmapped synchronizers

If an instruction uses a synchronizer address that is not mapped in the visible window (e.g. address “8” when there are only 4 synchronizers mapped), two behaviors are possible:

- either an implementation guarantees that such situations will always read the value zero as if it was a valid literal operand, or
- the situation constitutes a fault.

We found implementations exhibiting both these behaviors.

4.6.1.2 Stale states and values in local synchronizers

- Upon bulk creation, “local” synchronizers are freshly allocated for each thread context (section 4.3.3.2). Yet an implementation may choose to not reset the dataflow state after allocation. If that is the case, the synchronizer stays in the state it was from a prior use by another group of threads. It may contain a value, or it may be in the “empty” state. Therefore, in a new thread, if an instruction reads from a local synchronizer *before it has been written to* by another instruction from the same thread, two behaviors are possible:

- either the instruction reads from the previous state unchanged, and may read a stale value or deadlocks the thread if the operand was in the “empty” state; or
- the implementation guarantees that the value zero is read from any “empty” local synchronizers.

This aspect is crucially important when generating code. Indeed, if the thread program transfers control to a subroutine defined separately, and that subroutine subsequently spills some callee-save synchronizers to memory, this may cause the thread to suspend (and deadlock) if the synchronizers were still in the “empty” state. To avoid this situation, a code

generation would thus need to explicitly clear all local synchronizers to a “full” state prior to calling any subroutine. With the alternate implementation that reads zero from “empty” local synchronizers, this initialization is not necessary.

4.6.1.3 Writing to a non-full synchronizer

- If an instruction is issued with an output operand containing a future on the result of a previous asynchronous operation from the same thread (e.g. the target of a prior memory load, or the future of a family termination due to bulk synchronization), the following behaviors are possible:
 - undefined behavior: the hardware may let the instruction execute and overwrite the continuation of the pending result with a new value; when this happens, any threads that were waiting on that result will never be rescheduled, and a fault may or may not occur when the operation completes and the asynchronous response handler does not find a continuation in the synchronizer; or
 - the instruction suspends until the target becomes full and can be written to.

Undefined behavior can be avoided within procedure bodies when compiling programs with clear use-def relationships (as is the case in C), but is more insidious between procedure calls. Indeed, when transferring control between subroutines, a subroutine may spill some callee-save synchronizers to memory, and restore them from memory before returning control. This means that these subroutines end by issuing some memory loads but not waiting on the result. When the caller resumes, it may then choose to not reuse the previous value of the callee-save synchronizer, and instead write a new result to it. Without support from the hardware (second alternative above), a code generator must ensure that all memory loads at the end of a procedure are eventually waited upon, by introducing dummy “drain” instructions.

4.6.2 Invalid bulk creation parameters

- An invalid program counter during bulk creation (e.g. to non-readable memory) triggers the same behavior as a control flow branch to an invalid address, whatever this behavior is specified to be in the substrate ISA.

Concurrency management faults are then signalled in the following situations:

- the control word (section 4.4) contains invalid bit values;
- a concurrency management operation is provided an input that is not of the right *type*, e.g. the parameter to a create operation is not the identifier to a bulk creation context;
- the *order* of operations in the protocol is violated, e.g. the bulk creation request is sent twice on the same bulk creation context.

A more insidious situation comes from data-dependent behavior in bulk creation. In particular, if an invalid index range is provided, for example with the start value greater than the limit, then this yields undefined behavior in all the implementations we have encountered. Some let the bulk creation process start creating logical threads and never terminate, some others terminate bulk creation when the largest index value is reached. These ought to be signalled as faults, yet we should recognize that detecting the condition would add extra steps in the critical path of bulk creation, increasing its latency.

4.6.3 Deadlock detection and handling

- With the proposed primitives, deadlocks should only occur as a result of thread programs using the primitives explicitly in an order that creates a deadlock situation, or from the exhaustion of available concurrency resources. For example, a thread may create another logical thread which reads from an input dataflow channel, and then wait on termination of that logical thread without providing a source value on the channel. While these situations are avoidable “by contract” with a programmer or a higher-level code generator, they may nonetheless occur in a running system.

The result is one or more thread contexts that are suspended and never resume execution. The deadlock can then propagate (other threads waiting on termination recursively) or stay isolated. In either case the allocated thread contexts are never released and cannot be reused. This constitute leakage of hardware resources.

The implementations we have encountered do not provision any support for detecting and handling resource leakage due to deadlocks when it occurs. At best, a system-level deadlock detection may be available, which signals to an external operator that the entire system is idle yet there are some threads suspended.

In [JPvT08], the authors propose that resources (core clusters, communication channels) are *leased* to program components for contractual amounts of time or maximum memory capacities. If the contract is violated (e.g. due to a timeout or quota excess), the resource is forcibly reclaimed by the system. The existence of such mechanisms would provide a way to address the deadlock situations mentioned above; however they are not (yet) available in the implementations we worked with.

4.7 Specific implementations and their interfaces

- During our work we have been exposed to multiple implementations with different design choices for the various aspects covered above. Some of these implementations are listed in table 4.2. We can classify these implementations mostly into six categories along two axes:

1st, 2nd and 3rd generation ISAs. The first generation corresponds to the characteristics captured in the UTLEON3 implementation on FPGA, eventually published in [DKKS10, DKK⁺12]. The second generation introduces detached creation (cf. section 4.3.1.2), removes “hanging” mappings of synchronizers (cf. section 4.3.3.3) and thus makes new threads fully independent from creating threads; this simplifies code generation as we discuss later in sections 6.3.4 and 6.3.5. The third generation codifies placement across arbitrary cluster of cores, we discuss this further in chapter 11.

SPARCV8 (32-bit) vs. Alpha (64-bit). The SPARC-based implementations extend the SPARCV8 ISA with microthreading, but do not include support for traps, delay slots, sliding register windows, dataflow scheduling on the status registers (including Y), nor CAS. Sliding windows are replaced by explicit spills and restores (cf. Appendix H.9). The Alpha-based implementations extend the Alpha 21264 ISA with microthreading, but do not include support for traps, PALcodes nor LL/SC.

We illustrate these six major implementations in fig. 4.5. The figure also shows how the assembler and linker have evolved with the ISA generations.

Name	Substrate ISA	Allocation failure modes	Creation style	Placement style	Mapping style for "globals"	Mapping style for "shares"	Window layout
MGSim v1 200807	Alpha	suspend only	fused	local, explicit (local cluster)	hanging	hanging	G-S-L-D
MGSim v1 200902	Alpha, SPARCv8	suspend only	fused	local, explicit (local cluster)	hanging	hanging	G-S-L-D
MGSim v1 200904	Alpha, SPARCv8	suspend only	fused	inherit, local, explicit (named static clusters)	hanging	hanging	G-S-L-D
MGSim v1 200909	Alpha, SPARCv8	suspend, soft fail	fused	inherit, local, explicit (named static clusters)	hanging	hanging	G-S-L-D
MGSim v2 201004	Alpha, SPARCv8	suspend, soft fail	detached	inherit, local, explicit (named static clusters)	separated	separated	G-S-L-D
MGSim v2 201005	Alpha, SPARCv8	suspend, soft fail	detached	inherit, local, explicit (named static clusters)	separated	separated	L-G-S-D
MGSim v3 201103	Alpha, SPARCv8	suspend, soft fail	detached	inherit, local, explicit (arbitrary clusters)	separated	separated	L-G-S-D
UTLEON3 <201003	SPARCv8	suspend only	fused	inherit only	hanging	hanging	G-S-L-D
UTLEON3 201003	SPARCv8	soft fail only	fused	inherit only	hanging	hanging	G-S-L-D
UTLEON3 201011	SPARCv8	soft fail only	fused	inherit only	hanging	hanging	L-G-S-D
utc-ptl	N/A	trapping	detached	local, explicit (local cluster)	hanging	hanging	N/A
dutc-ptl	N/A	trapping	detached	local, explicit (nodes on network)	hanging	hanging	N/A
hlsim <201107	N/A	trapping	detached	local, explicit (local cluster)	hanging	hanging	N/A
hlsim 201107	N/A	suspend, soft fail	detached	inherit, local, explicit (arbitrary clusters)	separated	separated	N/A

Table 4.2: Characteristics of various implementations.

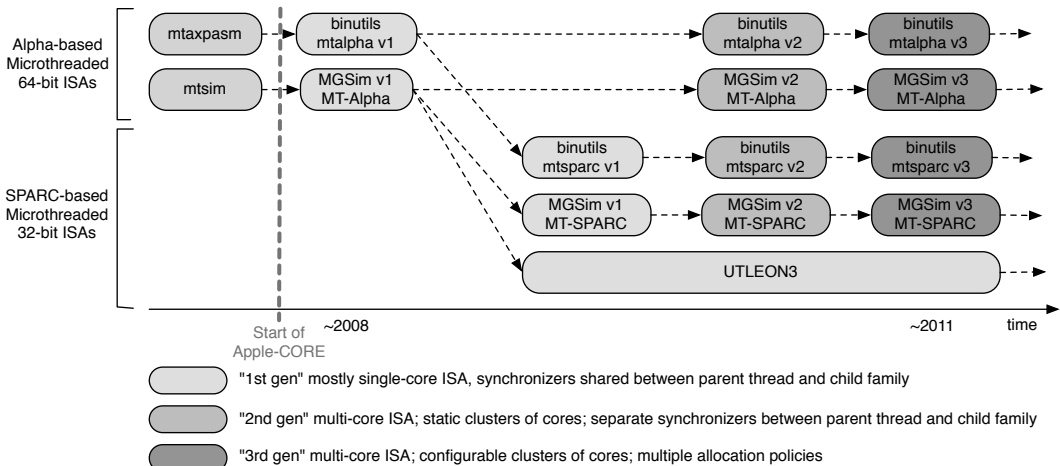


Figure 4.5: History of the hardware platform implementations.

4.8 Assembler and assembly language

The assembler program translates textual machine code into binary code suitable for execution. Any given assembler, and more importantly its *assembly language*, are thus dependent on the specific choice of instruction encoding and the design choices described earlier.

During our work we have been exposed to multiple assemblers, developed outside of our own research. They were all derived from the GNU assemblers for the GNU/Linux/Alpha (`alpha-linux-gnu`) and GNU/Linux/SPARC (`sparc-linux-gnu`) targets.

4.8.1 Common features

- The following features exist across all implementations:
 - the new mnemonics “`swch`” and “`end`” are recognized. The assembler computes the control bits (section 4.4) by associating the value “switch during fetch” to instructions immediately followed by `swch`, and the value “switch in fetch and end thread” to instructions immediately followed by `end`.
 - the new directive “`.registers`” is recognized. This should be used at the start of a thread program and produces the synchronizer mapping configuration discussed in section 4.3.3. The numerical arguments specify the number of local synchronizers per thread, the number of “shared” synchronizers shared between adjacent contexts, and the number of “global” synchronizers shared by all contexts. There are different arguments for the different register classes, for example integer and FP registers on Alpha.
 - the format of *register names* is extended to provide relatively numbered aliases to the different types of synchronizers in the visible window. For example, the name “`$13`” on the Alpha on an implementation with a G-S-L-D layout (section 4.3.3.4) is translated by the assembler to the operand offset $G + S + 3$, where G and S are the number of “global” and “shared” synchronizers declared earlier with “`.registers`”. The formats `$lN`, `$gN`, `$sN`, `$dN`, `$lfN`, `$gfN`, `$sfN`, `$dfN` (Alpha), `%t1N`, `%tgN`, `%tsN`, `%tdN`, `%tlfN`, `%tgfN`, `%tsfN`, `%tdfN` (SPARC) are recognized: “l” stands for “local”; “g” for “global”, and “s” and “d” for the “S” and “D” parts of “shared” synchronizer mappings (section 4.3.3.3). The SPARC variant uses a “`%t`” prefix for aliases to avoid ambiguity with the native SPARC name “`%g0`”.

4.8.2 Extra instructions

- The instruction mnemonics available depend on the choice of interface for bulk creation and synchronization (section 4.3.1). In all implementations we can find the following:
 - “`allocate`,” which expands to instructions that allocate a bulk creation context. There are different forms depending on the supported failure modes (section 4.3.1.1) and placement options (section 4.3.2.1).
 - “`setstart`,” “`setlimit`,” “`setstep`,” “`setblock`.” These expands to instruction that configure the parameters of bulk creation (sections 4.3.2.2 and 4.3.2.3).

Further instructions depend on:

- the creation style, discussed in section 4.3.1.2:

- interfaces with fused creation provide a combined “**create**.” (The Alpha ISA further distinguishes between “**cred**” and “**crei**” for immediate/direct and relative/indirect PCs specifications).
- interfaces with detached creation distinguish between “**create**” for bulk creation, “**sync**” to request bulk synchronization, and “**release**” to request de-allocation of the resources.
- whether synchronizers are “hanging,” discussed in section 4.3.3.3:
 - interfaces with “hanging” synchronizers must specify at which offset of the visible window of the issuing thread the sharing of hanging synchronizers should start. Here there are two variants; either there is no interface and the hanging always starts at the first “local” synchronizer (e.g. on UTLEON3); or an instruction “**setregs**” exists to explicitly configure the offset (e.g. on MGSim v1).
 - for interfaces without “hanging” synchronizers, explicit “**put**” and “**get**” instructions are available to communicate values between an issuing thread and the created threads.
- We have constructed a detailed specification of the exact formats and encodings of the various instructions on the Alpha and SPARC ISA variants of “MGSim v3” and “UTLEON3 201111,” reproduced in Appendix D.

4.8.3 Program images

- The implementations use either flat memory images or ELF [Com95] images to load program code and data into memory. The GNU linker has been modified prior to our work to produce these from object code created by the assemblers introduced above.

Summary

- The machine interface of the proposed architecture subsumes an existing ISA by substituting general-purpose registers with dataflow synchronizers. In addition to this, new primitives and extra semantics are added to the existing ISA to control the Thread Management Unit (TMU) in hardware and thus provide control over concurrency management to programs. To illustrate, we provide in Appendix C an example concrete program for a specific implementation.
- Due to a diversity of possible implementation choices, the specific machine interfaces of given implementations may differ slightly in their semantics. For example a fundamental distinction, visible to programs, exists between “fused” and “detached” bulk creation. We have highlighted the different areas where implementation choices impact the machine interface, and characterized the available implementations by their specific choices in these areas.