Implementation of the Control Unit in the TRIPS Prototype Processor

Ramadass Nagarajan Robert G. McDonald Doug Burger Stephen W. Keckler
Computer Architecture and Technology Laboratory
Department of Computer Sciences
The University of Texas at Austin
cart@cs.utexas.edu - www.cs.utexas.edu/users/cart

Department of Computer Sciences
Technical Report TR-2006-34
The University of Texas at Austin

June 26, 2006

Abstract

Future processor microarchitectures will feature distributed hardware components communicating using on-chip interconnection networks. Managing the common execution state and controlling the operations of different components are important design challenges for performing distributed computation on such architectures. This paper describes the fine-grained control mechanisms used in the distributed microarchitecture of the TRIPS prototype processor. A set of master-slave protocols driven from a centralized unit and implemented atop point-to-point networks controls the overall execution in the processor. The protocols are latency tolerant and support a back-end capable of executing up to 16 instructions in each cycle.

1 Introduction

Hard power budgets, coupled with growing on-chip wire delays are causing processor architectures to become increasingly distributed. At the same time, stopping clock frequency growths are forcing architectures to expose and exploit higher levels of concurrency from applications. Modular designs are also being preferred for managing complexity and enhancing design productivity. Several microarchitectural solutions have been proposed to address these issues [1, 4, 7, 8]. Common to all of them are two key design principles. First, distributed computation on a single chip involves a number of simple processing elements (PEs). Second, on-chip interconnection networks transport fine-grained messages between the different PEs.

The TRIPS architecture is one such solution. It uses a microarchitecture that consists of both processor and on-chip memory components residing as nodes on a set of interconnection networks and communicating using well-defined protocols [1]. The microarchitecture adheres to the following design principles: a) use a small number of heterogeneous components, b) design for productivity through component reuse, and c) do not use global wires anywhere in the system. The processor consists of multiple heterogeneous tiles—execution units, register file banks, data and instruction cache banks—and connects them using a set of data and control micronetworks. Likewise, the on-chip memory system is composed from a set of memory banks residing on a switched network. The TRIPS prototype chip is one implementation of the TRIPS architecture. It consists of two processor cores and a shared 1MB non-uniform L2 cache [2]. Implemented in the 130nm IBM ASIC fabrication technology, the chip consists of over 170 million transistors on a 18 mm $\times$ 18 mm die area.

Wires are treated as a first class design elements throughout the TRIPS architecture and different protocols recognize and tolerate the latency of signal propagation through the microarchitecture. The TRIPS processor composes one large processor from different heterogeneous tiles. Consequently, the microarchitectural execution of a program involves all the tiles in the processor. For example, the register file banks are required for providing register values, the data cache banks for memory operations, and the execution units for executing instructions. All of these distributed tiles must be managed together—resources allocated and deallocated—for the successful execution of a program. This paper describes the control logic that performs these operations in the TRIPS processor.

A single master unit called the Global control Tile (GT) generates control signals and drives them to the other tiles (slaves) using the control networks. The GT tracks the execution state on behalf of the entire processor and initiates various control protocols such as fetch, execute, flush, and commit by sending signals on the control network. The slaves locally perform different control operations solely based on inputs from the network. The slaves operate independently of each other and the protocols obviate any global synchronization. The protocols are latency tolerant; control signal propagation through the microarchitecture for one operation can be fully overlapped with another operation.

The use of control networks and protocols to manage distributed computation offers several benefits. First, it avoids the use of global wires on control logic paths, making it more suitable for future technologies. Second, it enables scaling to more processing units; adding more units only introduces additional nodes on the network. Finally, it enables modularity, simplifying design entry and verification efforts. In addition, the TRIPS execution model provides a significant reduction in the amount of control state compared to conventional architectures, thus simplifying the GT implementation. By centralizing the management to a single unit, any sequencing restrictions between the control protocols are implemented only within the GT, thus simplifying the implementation of the slaves.

The rest of the paper is organized as follows. Section 2 provides an overview of the TRIPS prototype chip, the microarchitecture, and details of the execution model. This section also identifies the different control operations required for managing the distributed execution on the microarchitecture. Section 3 provides an overview of the different logic blocks that comprise the control unit in the TRIPS processor. We

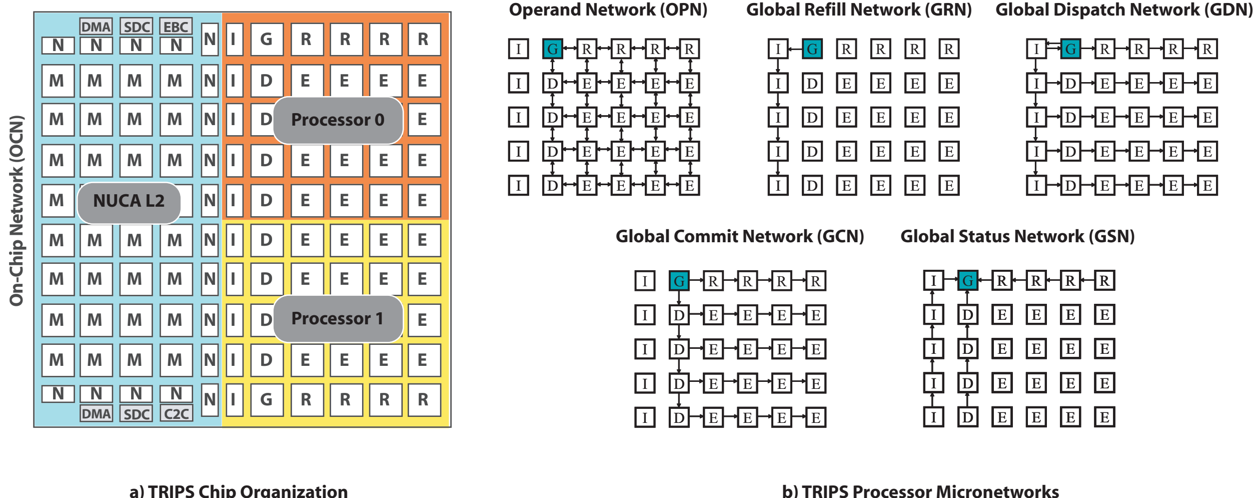

Figure 1: TRIPS Chip Overview.

present the details of the key control operations in Section 4 and conclude with a summary of our prototyping experience in Section 5.

2 TRIPS Microarchitecture

The TRIPS architecture is designed to address key challenges posed by future technologies—power efficiency, high concurrency, and adaptability to the demands of diverse applications [1]. Figure 1(a) depicts the schematic of the TRIPS prototype chip. It consists of two processor cores, and a secondary memory system, each of which is composed from smaller replicated hardware units connected by a set of micronetworks. In Figure 1a, the processor cores occupy the top right and bottom right quadrants, and the secondary memory system occupies the left half.

Each processor core is implemented using five different tiles, some of which are replicated. Each execution tile (ET) consists of an integer and floating point unit, a 64-entry reservation station, and is capable of executing one instruction in each cycle. Each register tile (RT) contains a portion of the architecture and physical register file. The data tiles (DT) and instruction tiles (IT) comprise the primary memory system for instruction and data respectively. The global control tile (GT) sequences the overall execution of a program. Each processor supports 16-wide out-of-order issue, 80KB of L1 instruction cache, 32KB of L1 data cache, and a window of 1024 in-flight instructions.

2.1 Micronetworks

Both the processor and L2 microarchitectures do not use any global wires. Each tile is small, typically of the order of $2 - 5\ mm^2$; therefore, connections within a tile use only local wires. A two-dimensional, worm-hole routed data network, also called the operand network (OPN), connects all tiles except the ITs. It is used for communicating data operands between the tiles. A set of control networks—GRN, GDN, GSN, and GCN—is responsible for transmitting all control signals that manage the overall execution in the processor. Unlike the OPN, there is no flow control on any of the control networks and consequently, no signal propagation stalls. Figure 1(b) depicts these networks. Each link on a network connects only the immediate neighbors and has a transmission latency of one cycle.

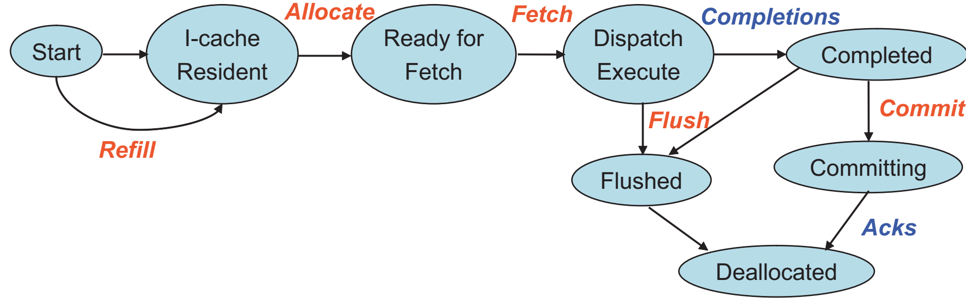

Figure 2: Different states of block execution.

2.2 Block Structure

The TRIPS ISA groups up to 128 instructions into a single TRIPS block. The ISA encodes each block in five 128-byte chunks, four of which are instruction chunks and one of them a header chunk that contains meta information about the block. The microarchitecture executes these blocks in a block-atomic fashion—every block is logically fetched, executed and committed as a single atomic unit. Exceptions, if any, are handled at block boundaries and are not instruction-precise. A block always emits a constant number of outputs—up to 32 registers, up to 32 stores and one branch output specifying the address of the next block.

2.3 Block Execution

The prototype processor supports an active execution window of up to eight of blocks. The execution resources are partitioned into eight slots called *frames* and each block executes in a separate frame. The processor can be configured to run in either single-threaded mode or simultaneous multi-threaded mode. In the single-threaded mode of operation, up to eight blocks belonging to the same thread can be in-flight simultaneously, seven of them speculatively. In the multi-threaded mode of operation, each thread can have up to two blocks in-flight, one of them speculatively. Control registers in the GT configure the processor into one of these two modes.

The execution of a single block involves several block-level operations: Figure 2 shows the different states in the lifetime of a block and the operations that induce the transitions between the states. A refill operation fills the instructions of a block into the I-cache from the secondary memory. After a free frame is allocated, the fetch and dispatch operations distribute the instructions to the execution units, where they execute in a dataflow fashion. Operand values are routed from one tile to another and any register and store outputs are routed to the RTs and DTs respectively. The block completes its execution after it has produced all of its outputs. A commit operation saves the architectural state modified the block. Any misspeculations will result in the flush of all state modified by the block.

Distributed execution of a single block requires solutions to two major challenges: a) controlling the operations of the distributed tiles, and b) managing the execution state of all in flight blocks. The GT implements the control logic functions required for both of these tasks. This logic is different from other distributed processors. Multi-core architectures such as Niagara [3], RAW [8] and SmartMemories [4] are examples of architectures that use a full processor, complete with register files and caches, as individual PEs on a distributed substrate. TRIPS uses smaller PEs to compose a full processor. Consequently, TRIPS requires fine-grained control mechanisms to keep all the components synchronized for correct operation.

The GT drives several protocols—refill, fetch, commit, and flush—to manage the distributed execution. It initiates a protocol by sending a signal on one of the control networks. Other tiles respond by performing the specified operation independently. For design simplicity and high performance, the implementation must satisfy a number of desired properties. First, any control state maintenance must attain a balance between

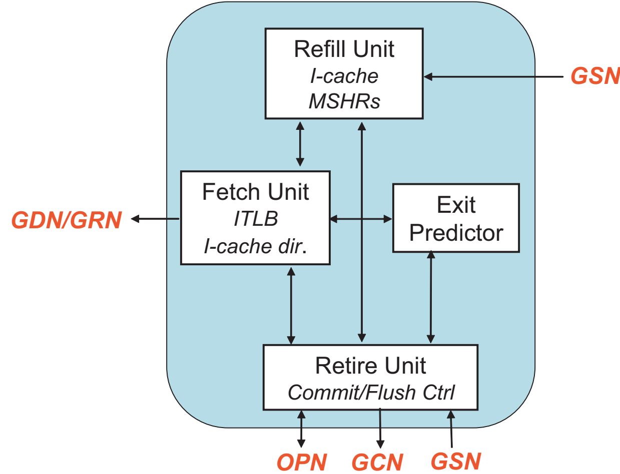

Figure 3: High-level organization of the GT.

centralization—for minimizing replication—and distribution—for maximizing concurrency. Second, since transporting signals across different tiles involves high latency, the control protocols must be latency tolerant. Different protocols must overlap their operations as much possible to maximize throughput. Finally, the protocols must be devised such that peak execution bandwidth of one instruction executing per tile in each cycle can be met.

3 GT Implementation

The GT implements all of its logic functions using four major sub-units: the fetch unit, refill unit, retire unit, and the exit predictor. Figure 3 shows the high level organization of these sub-units. In this section, we provide detailed descriptions of some these sub-units. We also compare each sub-unit with their counterparts in conventional processors and provide the rationale behind their designs.

3.1 Fetch Unit

The fetch unit consists of a TLB (Translation-Lookaside Buffer) and a directory of the blocks that are resident in the I-cache. In addition, it contains the program counters (PC) for each thread and control registers that are used to configure the execution of each block.

3.1.1 I-cache Directory

Table 1 provides an overview of how different portions of a block are cached by the ITs. The instructions of a single block are striped across all of the ITs. For example, IT0 caches chunk 0 of a block and IT1 caches

Tags for cached blocks	GT
chunk 0 instructions	IT0
chunk 1 instructions	IT1
chunk 2 instructions	IT2
chunk 3 instructions	IT3
chunk 4 instructions	IT4

Table 1: Storage of a block in the I-cache.

V	Valid block
L	LRU information
PTAG	Physical tag of the block’s address
H	Meta information for the block

Table 2: An entry in the I-cache directory.

V	Valid refill
S	Set in the cache being refilled
W	Way in the set being refilled
TID	Thread corresponding to the refill
PTAG	Physical tag of the block's address
F	Refill already flushed/cancelled
C	Refilled completed
Ca	Block L1 cacheable or not
H	Meta information for the refilled block

Table 3: State tracked for each pending refill.

chunk 1 of the same block. The I-cache directory contains a listing of all blocks that are currently resident in the I-cache. Table 2 provides a description of an entry in the directory. The directory consists of 128 entries, organized in a 2-way set-associative fashion. Each entry identifies a unique cached block and also stores a portion of the meta information associated with the block. The directory is virtually indexed and entries are evicted and replaced in a LRU fashion.

The I-cache directory is similar to the tag array in conventional caches. In the TRIPS processor, the GT maintains a single array on behalf of all the ITs. An alternate design could maintain the tag array as part of each IT. Since a single block is striped across all ITs and each of them operate in a distributed fashion, this approach would require special hardware to keep the tag arrays consistent. A centralized directory provides a consistent view of the cached blocks and avoids scenarios where portions of a block are not present in the I-cache. The tag array in each IT can be eliminated, thus simplifying the implementation in both the GT and ITs.

3.1.2 Instruction TLB

A set of sixteen registers provide the translations of virtual addresses of blocks to physical addresses. Each register defines the size and read/execute access attributes of up to sixteen memory segments. The minimum size of a memory segment is 64KB and the maximum size is 1TB. Instruction memory segments may be marked as uncacheable in the L1. A block in such a segment will never be filled into the I-cache. A miss in the ITLB or an access protection violation will result in an exception being generated. Similar to the I-cache directory, implementing the ITLB inside the GT avoids redundant implementation in the ITs.

3.2 Refill Unit

The refill unit maintains the status of pending I-cache refills. The TRIPS processor supports up to four outstanding refills, but at most one per thread. Table 3 shows the state that the GT tracks for each pending refill. The state includes information such as the I-cache set and the way being refilled, whether the refill has completed or not, and the meta header information for the block being refilled. The pending refill state in the GT is similar to the I-cache MSHR (Miss Status Handling Register) state in conventional processors.

3.3 Retire Unit

The retire unit consists of the retirement table which tracks the execution state of all blocks in flight. It is also responsible for initiating the flush, commit, and deallocation of the blocks in flight. Table 4 shows the details of the state maintained for each block. Most of this state is updated locally by the GT, when it

V	Valid block
O	Oldest block in thread
Y	Youngest block in thread
BADDR	Virtual address of the block
PADDR	Predicted address of the next block
RADDR	Actual resolved address of the next block
RC	Registers completed
SC	Stores completed
BC	Branch completed
RCOMM	Registers committed
SCOMM	Stores committed
E	Exception in block
F	Block already flushed

Table 4: State tracked for each block in the retirement table.

starts various block-level operations. The rest is updated when the GT receives notifications on the control networks from other tiles.

The retirement table is similar to the reorder buffer (ROB) in conventional processors. However, this table does not track the status of individual instructions. It has only one entry for each block, thus containing far fewer entries than a conventional ROB.

3.4 Exit Predictor

The exit predictor predicts the address of the next block to execute from a single thread. It uses both local and global history information and employs a tournament-style prediction similar to the Alpha 21264 predictor [5]. The predictor state amounts to a total of 74 Kbits and together, they sustain competitive accuracies compared to the Alpha 21264 predictor.

The predictor performs three major operations—*predict, update and repair*. Predict provides a prediction for the next block. Update modifies the predictor tables with the information from a committing block. Repair corrects any predictor state modified by incorrect speculation. The predict and update operations each consume three processor cycles, while the repair consumes two cycles. None of the operations are overlapped with any other. A detailed discussion of the predictor implementation is beyond the scope of the paper.

3.5 Physical Implementation

The TRIPS chip implemented using the 130nm IBM ASIC fabrication technology uses more than 170 million transistors in a chip area of 3.1 mm^2. Each processor core occupies 29% of the overall chip area. The GT occupies roughly 2% of each processor, in a 3.4 mm $\times$ 0.9 mm area. The predictor tables consume nearly 50% of all the logic cells in the GT. The fetch and retire sub-units consume 19% and 11% of the logic cells respectively. The OPN router inside the GT occupies 14% of the GT area. The refill unit and other miscellaneous logic consume the rest.

4 Block Operations

This section provides a detailed description of four block-level operations: a) I-cache fills, b) block fetch and dispatch, c) block commit, and d) block flush. We describe how the GT logic blocks simplify the

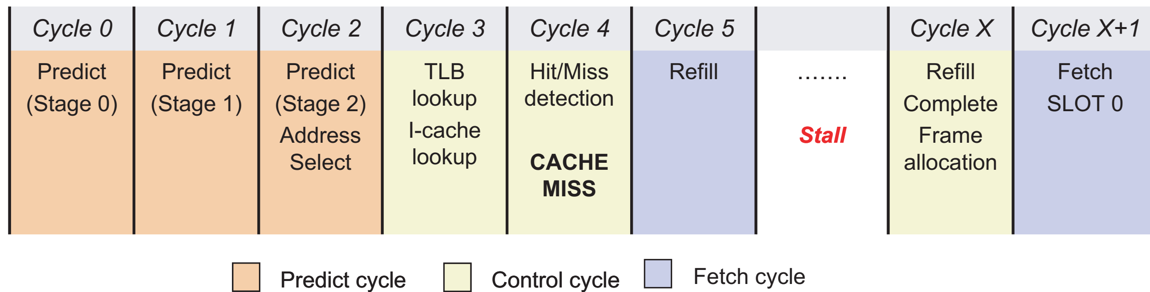

Figure 4: Refill Pipeline.

implementation and also discuss design limitations and alternatives.

4.1 I-cache Fills

The I-cache fill of a block happens in two steps – *fill* and *update*. In the fill step, the instruction bits are fetched from the secondary cache and buffered in a structure called the fill buffer. In the update step, the instruction bits are read from the fill buffer and written into the I-cache banks. The refill protocol performs the fill operation. The fetch protocol described in the next section performs the update operation.

Figure 4 depicts the different events during the execution of the refill protocol. It begins with the GT sending the physical address of the block on the GRN interface (cycle 5). During the preceding cycles (0–2), the GT computes the address of the block to refill. The GT performs a TLB translation, a lookup of the I-cache directory, and detects a miss in cycles 3 and 4. It begins the refill operation in cycle 5. Each IT subsequently receives the refill command and independently launches several transactions with the secondary memory to fetch its chunk of the block. When the fill operation completes in all the ITs, they notify the GT using the GSN. After the GT receives such a completion message from the ITs, the entire refill operation is marked as completed.

The centralized I-cache tags and the refill protocol allows the GT to control the operation of the ITs effectively. Occasionally, the GT may chose to discard a block saved in the fill buffers without updating the I-cache. It does so by simply not initiating an update step. By modifying the entries in the I-cache directory, the GT may evict an already cached block to accommodate a new block. The centralized tags help maintain consistency in the IT data banks. At the same time, the ITs can operate independently without any explicit synchronization.

An alternate design for the refill protocol might merge the fill and update steps of a refill and eliminate the need for fill buffers. However, branch mispredictions often result in refills that do not correspond to any legal block. These spurious refills pollute the I-cache and evict other blocks that are currently in the working set of the program. Occasionally, branch mispredictions result in correct prefetching refills. However, we observed that the pollution resulting from spurious refills outweigh the benefits of occasional serendipitous refills.

4.2 Block Fetch

The GT initiates a fetch protocol to distribute the instructions from the IT data banks to the execution units. Figure 5 shows the different events during the fetch protocol. The GT allocates a free frame for the block and begins the fetch protocol by issuing a command on the GDN. The command includes the address of the block and the frame identifier allocated for the block. In addition, the GT also instructs the ITs to perform the update step of a refill operation if the fetch resulted from a preceding refill.

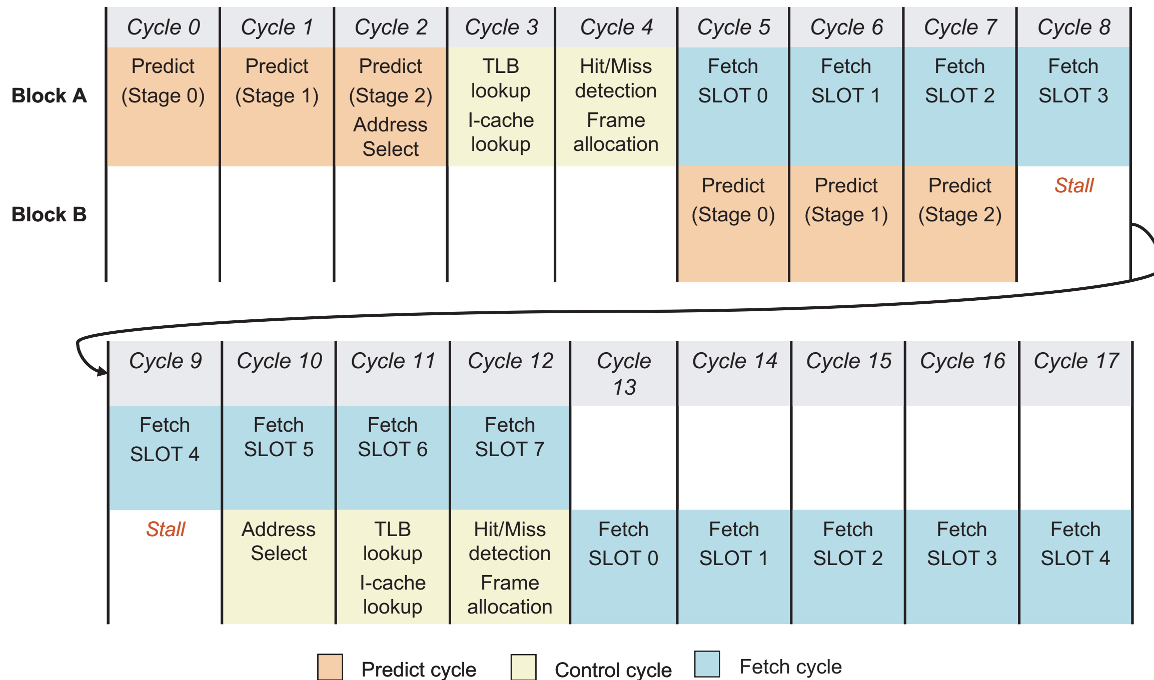

Figure 5: Fetch Pipeline.

The GDN transports up to 128 bits of instructions during each cycle. Since each instruction chunk consists of 128-bytes, initiating the fetch and dispatch for an entire block consumes eight cycles. Following the eight cycles, the GT can initiate the fetch and dispatch for the next block. Figure 5 shows how the fetches for two blocks are pipelined. The GT consumes eight cycles to initiate the fetch of the first block, starting from cycle 5. In parallel, a prediction is made and the fetch of the next block is set up during the cycles 5–12. The fetch of the second block starts at cycle 13.

The distributed fetch protocol provides significantly higher fetch bandwidth compared to conventional processors. Directing the fetch from the GT obviates the need for frame management at every tile. Since a new frame is required for executing every block, managing the free list of frames in a distributed fashion and keeping them in sync requires additional hardware mechanisms. Managing the free list in the GT and propagating the allocated identifier along with every fetch reduces the complexity in other tiles.

The implementation tightly couples the predictor operations and the fetch protocol operations in one single pipeline. In steady state, the three cycles for predict and three cycles for update can fully overlap with the 8 cycles of fetch required for one block. Thus there are no bubbles in the fetch pipeline, enabling a new block fetch every eight cycles. This offers a peak fetch rate of 16 instructions per cycle (128 instruction / 8 cycles) matching the peak execution rate of the processor. Occasionally, the predictor update operation may delay the predict operation causing bubbles in the fetch pipeline. For example, in Figure 5, an update operation starting in cycle 5, could delay the predict operation for the second block until cycle 8. The fetch of block B will not start until cycle 14, introducing a bubble in the pipeline.

An alternate design could have completely decoupled the prediction pipeline from the fetch pipeline using a fetch target buffer [6]. That design offers two advantages. First, multiple refills can be initiated well ahead of a fetch, offering prefetching benefits. Second, stalls in the predict pipeline are less likely to affect the fetch pipeline. Implementing this design requires additional block management in the fetch unit. During design time, the extra complexity did not appear to be worth the benefits.

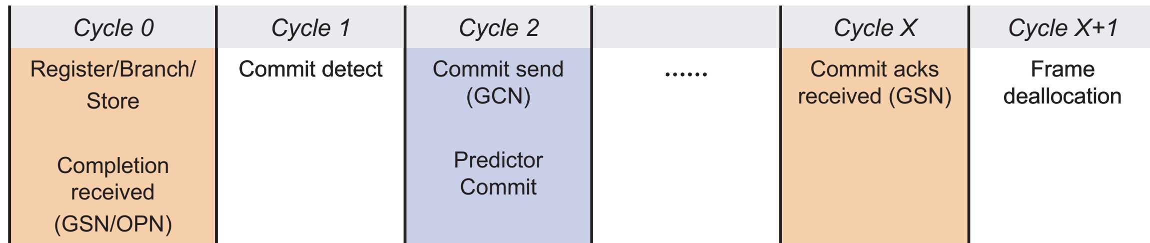

Figure 6: Commit Pipeline.

4.3 Block Retirement

The retire unit detects when a block has completed execution, and initiates the flush and commit of the blocks. It is also responsible for deallocating the frame allocated for executing a block.

4.3.1 Completion Detection

A block completes its execution if it has produced all of its outputs. The RTs track the register outputs, while the DTs track the store outputs. A branch instruction sends its result to the GT using the OPN. When the RTs detect that all registers have been produced, they inform the GT by sending a message on the GSN. The GT receives one message on behalf of the entire block and subsequently updates its retirement table. The GT detects the completion of the store similarly. When all outputs have been produced, the GT marks the block as completed. Detecting completion in such a distributed and hierarchical fashion avoids the complexity of sending and tracking multiple messages to the GT to detect completion.

4.3.2 Flush Operation

The GT flushes the execution of a block if any misspeculations or exceptions are detected. Exceptions are reported and detected along with the completion messages. A flush operation begins with the GT issuing a flush message on the GCN, which propagates to the RTs, DTs and the ETs. Each tile responds by invalidating all state corresponding to the flushed blocks. The GT invalidates all the state corresponding to the flushed blocks in the retirement table and stops pending fetches for flushed blocks.

4.3.3 Commit Operation

A block may be committed if it has completed its executed without any exceptions and if all previous blocks have been committed. Figure 6 shows the different events of a commit operation. It begins with the GT issuing a commit message on the GCN. The RTs and the DTs begin the commit operation and when they are have completed that operation, they send a message to the GT using the GSN. The GT deallocates the block, after it has received the acknowledgements from both the RT and the DT. These acknowledgements are required because, the number of outputs and therefore, the commit latency varies for each block.

4.4 Summary

A control signal generated at the GT incurs multiple cycles of latency to propagate to all the tiles. Table 5 provides the minimum latencies for a few block events. For example, the minimum latency for executing the last instruction assigned to the farthest ET (bottom right corner) is 20 cycles. If the result of this instruction is a register output, that result takes a minimum of four cycles to reach an RT. Notifying the completion of the register outputs to the GT consumes up to four additional cycles. Combining all events for a single

Event	Latency
Dispatch of first instruction to nearest ET	4
Dispatch of last instruction to farthest ET	17
Execution of first instruction in nearest ET	7
Execution of last instruction in farthest ET	20
Commit in nearest RT/DT	20
Commit in farthest RT/DT	24
Deallocation	32

Table 5: Minimum latencies for a few block events. Latencies are measured in processor cycles from the start of the fetch protocol in the GT.

block, the overall lifetime of block from start to deallocation could involve a significant number of control signal propagation cycles. Hiding these cycles is important for attaining high throughput.

In the TRIPS prototype implementation, all control protocols are pipelined. Except for fetch, a new operation for other protocols can be started in every cycle. The fetch for a second block can be initiated eight cycles after the first block. The fetch of a new block following a flush can be started three cycles after the flush. Such pipelining amortizes the latency of the control protocols across multiple tiles and blocks. For example, while one tile is performing the flush of speculative blocks, another could perform the commit of the non-speculative block, and a third tile could dispatch the instructions of a new block all in the same cycle.

5 Conclusion

Distributed microarchitectures with on-chip networks are likely to be prevalent in the future. Control of the distributed hardware units and management of the common execution state will be an important component of the design. The TRIPS prototype processor implements fine-grained control using a set of master-slave protocols. During the implementation of these protocols, we learned a few important lessons:

- **Separation of the control networks:** In the first design, we implemented all of the control protocols using a single shared network. However, we observed that the contention for the network among the different protocols wasted the execution bandwidth of the processor. In fact, it was not possible to match the peak execution rate of the processor. Consequently, in the final design we implemented each protocol with a separate network at the cost of additional wiring between the tiles. Since the protocols were well-defined and fairly simple, migrating to a new implementation did not involve a significant redesign effort.

- **Predictor/Fetch coupling:** For design simplicity, we chose to couple the predictor and fetch operations in one single pipeline. This introduced bubbles in the fetch pipeline, limiting the fetch rate of the processor. Future implementations must choose to decouple the two pipelines.

- **Reducing flush overhead:** Reducing the penalty of a flush, i.e, the latency to start a new fetch after a flush was important. The GT implementation could not reduce this latency to any further than three cycles. Further reductions required additional ports in the TLB and the I-cache directory and their associated complexity.

The GT logic paths were such that detecting a flush and subsequently clearing the retirement table state was a single cycle operation. These paths were the dominant critical paths in the GT for meeting cycle time. However, none of the GT paths appear among the top timing critical paths in the entire processor.

Our experience shows that even on a distributed substrate, the control operations can be accomplished with relatively simple protocols. The techniques demonstrated in this paper are viable design solutions for future distributed microarchitectures.

References

[1] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the End of Silicon with EDGE architectures. *IEEE Computer*, 37(7):44–55, July 2004.

[2] C. Kim, D. Burger, and S. W. Keckler. An adaptive, non-uniform cache structure for wire-delay dominated on-chip caches. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 211–222, October 2002.

[3] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[4] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz. Smart memories: A modular reconfigurable architecture. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 161–171, June 2000.

[5] N. Ranganathan, R. Nagarajan, D. Jimenez, D. Burger, S. W. Keckler, and C. Lin. Combining hyperblocks and exit prediction to increase front-end bandwidth and performance. Technical Report TR-02-41, Department of Computer Sciences, The University of Texas at Austin, September 2002.

[6] G. Reinman, T. M. Austin, and B. Calder. A scalable front-end architecture for fast instruction delivery. In *Proceedings of 26th Annual International Conference on Computer Architecture*, pages 234–245, May 1999.

[7] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. WaveScalar. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–302, December 2003.

[8] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2):25–35, 2002.