November 2020

A Chained Barrel Processor

N/A

A Chained Barrel Processor

ABSTRACT

A barrel processor implements a technique to interleave a set of B instruction streams in a round-robin manner, such that a given thread/stream has an instruction slot once every B cycles. The barrel approach delivers throughput in a way that is efficient in circuit area and power, but suffers from the constraint that a given thread of execution can issue only once every N cycles, limiting single-threaded performance. This disclosure describes a hybrid architecture and microarchitecture that improves single-threaded performance while preserving the bandwidth advantages for multithreaded applications.

KEYWORDS

- Barrel processor
- Multithreading
- Reduced instruction set computer
- RISC
- Single instruction, multiple data
- SIMD
- Replicated barrel operation
- Chained operation
- Execution pipeline

BACKGROUND

The “end of Moore’s law,” the point at which improvements in semiconductor density can be relied upon to deliver continuous exponential improvements in price/performance, is no longer a hypothetical future. This has many consequences for computing. Architectural and

microarchitectural techniques for improving performance and efficiency are therefore of great importance.

A barrel processor is an architectural technique to interleave a set of B instruction streams in a round-robin manner, such that a given thread/stream has an instruction slot once every B cycles. The functioning of a barrel processor can be thought of as being similar to a Gatling gun. The fact that the next instruction for a given thread is not issued for B cycles simplifies pipeline design. Functional unit timing is relaxed, many bypass paths are no longer required, and some stalls become impossible. This technique has been leveraged at scales from embedded DSPs, such as the Qualcomm Hexagon to supercomputers, such as the Cray MTA.

Cycle	Slot			
	0	1	2	3
0	LD R0,[R4](R7)			
1		MUL R3,R1,R0		
2			LD R0,[R4](R7)	
3				ADD R4,R4,1
4	LD R1,[R5](R7)			
5		ADD R2,R3,R2		
6			LD R1,[R5](R7)	
7				ADD R5,R5,1
8	ST R0,[R4](R6)			
9		SUB R4,R4,1		
10			ST R0,[R4](R6)	
11				SUB R2,R2,1
12	ST R1,[R5](R6)			
13		ST (R5),R2		
14			ST R1,[R5](R6)	
15				BNZ R2,-7

Fig. 1: The distribution of instruction issue cycles across a four-slot barrel processor

Fig. 1 illustrates the distribution of instruction issue cycles across a four-slot (B=4) barrel processor. There are four interleaved instruction streams executing in parallel, with the instruction stream for each slot operating on its own set of registers and memory maps. The barrel approach delivers throughput in a way that is efficient in circuit area and power, but

suffers from the constraint that a given thread of execution can issue only once every N cycles, limiting single-threaded performance. This is an acceptable trade-off for some applications, but certainly not for all applications.

DESCRIPTION

This disclosure describes a hybrid architecture and microarchitecture that improves single-threaded performance while preserving the bandwidth advantages for multithreaded applications.

The techniques described herein define *chains* as replicated barrel operations with a single thread of control. The pipeline can enter and leave a mode where each instruction issued by a designated physical thread context is mirrored for other subsequent slots around the barrel, a form of single instruction, multiple data (SIMD) parallelism. A chained operation is one where a single thread of execution sets up the issue and execution pipeline for the following S barrel slots, in the place of their normal, independent fetch and issue logic. The instruction sequences normally driving the slots are suspended while the slot is chained. The memory map/translation lookaside buffer (TLB) entries used for the initiating slot is used by the others while chaining is in effect. Control flow branches are taken only by the slot that initiated the chaining.

Borrowing the register file of a given slot with a chained operation from another slot creates a state management problem that can be handled in several ways. In processors performing register renaming, it may be possible to assign new physical registers for use during the chained operations and restore the previous mapping afterwards. At the simpler end of the implementation spectrum, software can perform the function. There can be a software convention that only a specific subset of registers can be used in chained sequences, and these

are only used in a chained state, or at some cost in overhead cycles, software context save-and-restore can be performed.

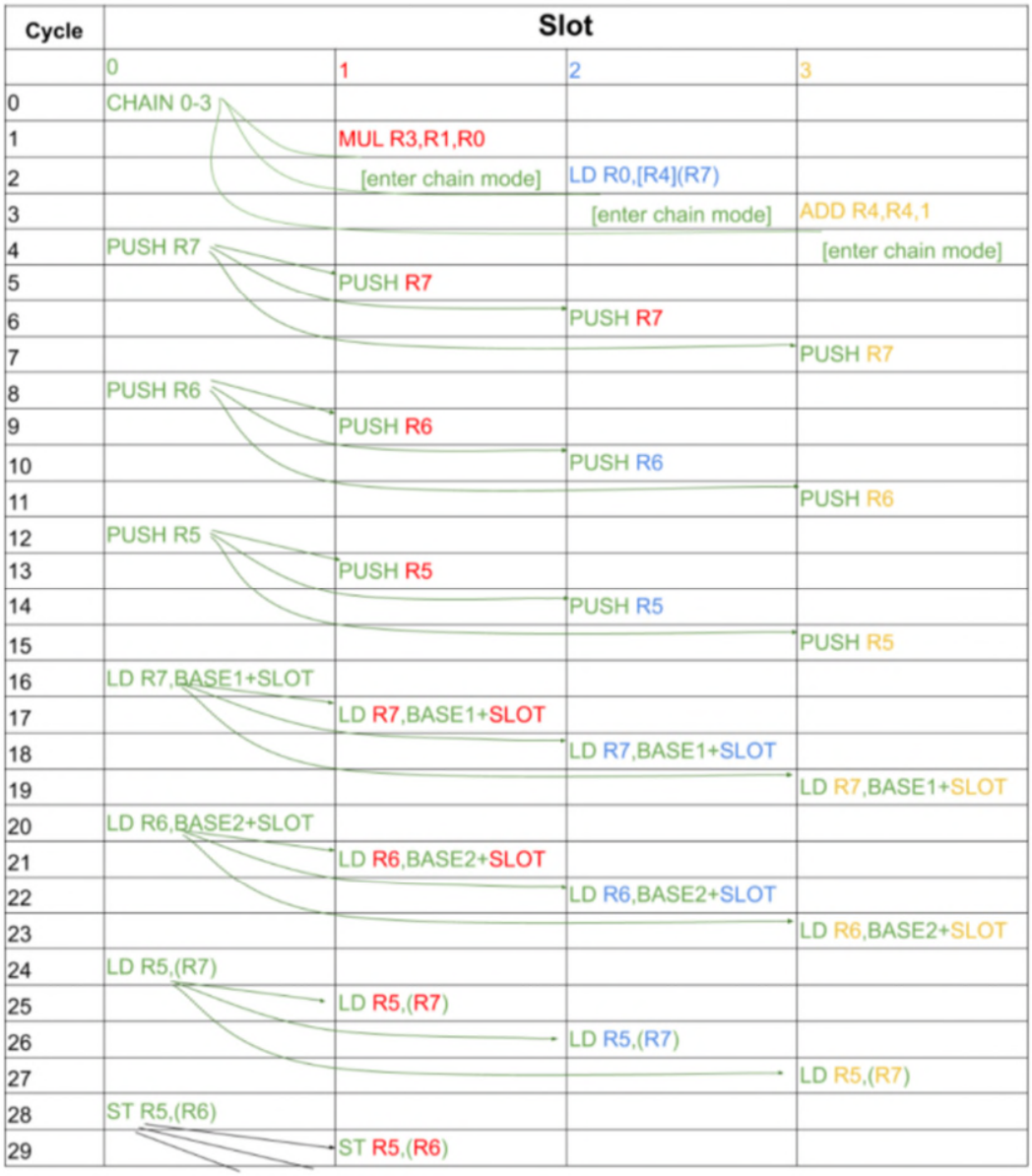

Fig. 2: Instruction issue sequences across four slots as slot 0 puts the processor into a chained state

Fig. 2 illustrates the instruction issue sequences across 4 slots as slot 0 puts the processor into a chained state. The “CHAIN” instruction has 4 cycles before its effects can be visible to the instruction stream of the initiating slot, and the instructions from the original threads running on

slots 1-3 can issue and execute normally. After that, each instruction from slot 0 is issued locally, then issued on the other slots, in sequence, operating on the register file of each respective slot, as indicated by their color.

In the reduced instruction set computer (RISC) example above, modeling the beginning of the chained copy operation, the controlling slot issues PUSHes of all the registers that will be used in the chained code. With good compiler technology, this may be unnecessary on the lead thread itself; however, there is no way of knowing what the other slots are doing with the needed registers. The next step in the example is to set up source and destination address registers. These need to be different for each slot. This is handled by an element of per-slot architectural state, a SLOT value that can be used explicitly or implicitly to manage chain parallelism. In a copy, SLOT may be used to assign a chained slot to copy every Bth word, every Bth cache line, or every Bth page. The limit bandwidth of a chained copy is 1 word every 2 operations (a load and a store), versus 1 word every 2B operations in a standard barrel processor.

Cycle	Slot			
	0	1	2	3
0	ST R5,(R6)			
1		ST R5,(R6)		
2			ST R5,(R6)	
3				ST R5,(R6)
4	SUB R4,R4,1			
5		SUB R4,R4,1		
6			SUB R4,R4,1	
7				SUB R4,R4,1
8	BGEZ R4,-8			
9		[NOOP]		
10			[NOOP]	
11				[NOOP]
12	POP R4			
13		POP R4		
14			POP R4	
15				POP R4
16	POP R5			
17		POP R5		
18			POP R5	
19				POP R5
20	POP R6			
21		POP R6		
22			POP R6	
23				POP R6
24	UNCHAIN			
25				
26		[exit chain mode]		
27			[exit chain mode]	
28	LD R0,(R1)			[exit chain mode]
29		ADD R2,R3,R2		
30			LD R1,[R5](R7)	
31				ADD R5,R5,1
32	ST R0,[R4](R6)			
33		SUB R4,R4,1		
34			ST R0,[R4](R6)	
35				SUB R2,R2,1

Fig. 3: The end of a chained loop of instructions

At the end of a chained loop of instructions, if there is no further need for chaining, the process is reversed, as illustrated in Fig. 3, which illustrates how this might be done in the hypothetical RISC-like implementation. When the loop completes, chained POP operations restore the registers used for the chained work, and an explicit UNCHAIN operation is

performed. The slots can start issuing the next instructions in their own thread starting with the slot after the UNCHAIN; however, this example assumes that an extra cycle is required to restart independent issuance of instructions.

Architectural constructs

The following are certain architectural constructs required for a chained barrel processor:

- An instruction or event that establishes a chained mode of operation. This can be an explicit instruction, an implicit property of a more complex instruction, or an implicit property of an exception.
- Explicit or implicit state which distinguishes execution slots, to enable the division of work. Existing “node ID” or “CPU Number” architectural state may be used for this. The state could, in principle, be maintained by software in memory, if memory maps are per-slot. Software can guarantee a read-only mapping to a distinct value at a well-known address.
- An instruction or event that terminates the chained mode of operation. This is the inverse operation of the above instruction that establishes a chained mode of operation. It can be an explicit instruction, an implicit property of a more complex instruction, or an implicit property of the return from an exception.

The explicit CHAIN/UNCHAIN instruction in the example is an explicit RISC-like mechanism that is simple to implement and useful for demonstration. These chained barrel processor concepts can be applied to implementations of various instruction set architectures.

Exceptions

Chaining introduces an exception case that is not otherwise present in a barrel processor. It is possible for a slot that has been chained to follow another to take a value (floating point) or address (page fault) exception that would not have occurred on the initiating slot. One way to handle this is to have the initiating slot take any exceptions from any chained slot, but suspend execution (chained or otherwise) of the chained slots on entry to the exception handler. Depending on the exception model and system requirements, mechanisms may be needed for a trusted OS kernel or agent to query and modify the registers of one slot from another. This need not be efficient, as it is only needed for exception diagnosis and fixup.

Security isolation

While there are many application execution environments where all slots of a barrel processor are used by threads of the same process, with the same access controls; per-thread memory maps are necessary for some programming models. A clean architecture can enable the security model of isolation of both memory and register values between users, thus, between slots. This is a source of overhead and complexity that is not central to the chained barrel concept, but which needs to be dealt with for broad applicability of the concept.

There are two basic security models to consider. One where chaining is available to arbitrary, otherwise unprivileged programs, and one where only trusted code can initiate chaining. If only trusted code can perform the equivalent of a CHAIN/UNCHAIN sequence, it can be trusted to not make illicit copies of the register data of other slots, and no additional hardware mechanisms other than those described need to be applied. If the initiation of chaining is to be enabled for arbitrary, untrusted software in a multi-user environment, further

architectural features may be required. To show that the problem is solvable, an example is described below; an exhaustive exploration is outside the scope of this document.

One option is to break the chained/unchained state transition into further substates. Instead of going directly into the state where the collection of register files is operating on the memory map of the chain-initiating slot, a sort of “PRECHAIN” operation can be performed to send the chained slots into a state where:

- instructions issued to the chain-initiating slot are sent to the other slots, as above, but
- the memory map of the slot is retained, and
- when a store (or push) operation is performed on a register, that register is cleared or otherwise scrambled so as to make its previous value unavailable.

The chaining prologue does a PRECHAIN, after which the PUSHes or stores of the working registers go into the prior process memory map. The values are in memory not mapped to the initiating slot, and have been purged from the register file. With this done, a fully chained state can be enabled, e.g., with a CHAIN instruction, putting all the slots in the same memory map as the initiating slot. At the end of the chained section, a reverse process can be applied, a DECHAIN instruction that enters the intermediate memory mapping state, enabling POP or load operations sent to the chained slots to restore the previous register values invisibly to the initiator, after which UNCHAIN releases the chained slots to their prior instruction stream.

CONCLUSION

This disclosure describes a hybrid computer architecture and microarchitecture for barrel processors that improves single-threaded performance while preserving the bandwidth advantages for multithreaded applications.