Description of The Functionality of The Impulse Memory Controller

Lixin Zhang

UUCS-01-009

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

July 10, 2001

Abstract

This document describes the functionality and control flow models for each component of the Impulse main memory controller.

1 Background

1.1 KISS rule

Keep **I**t **S**imple and **S**tupid.

1.2 Impulse architecture

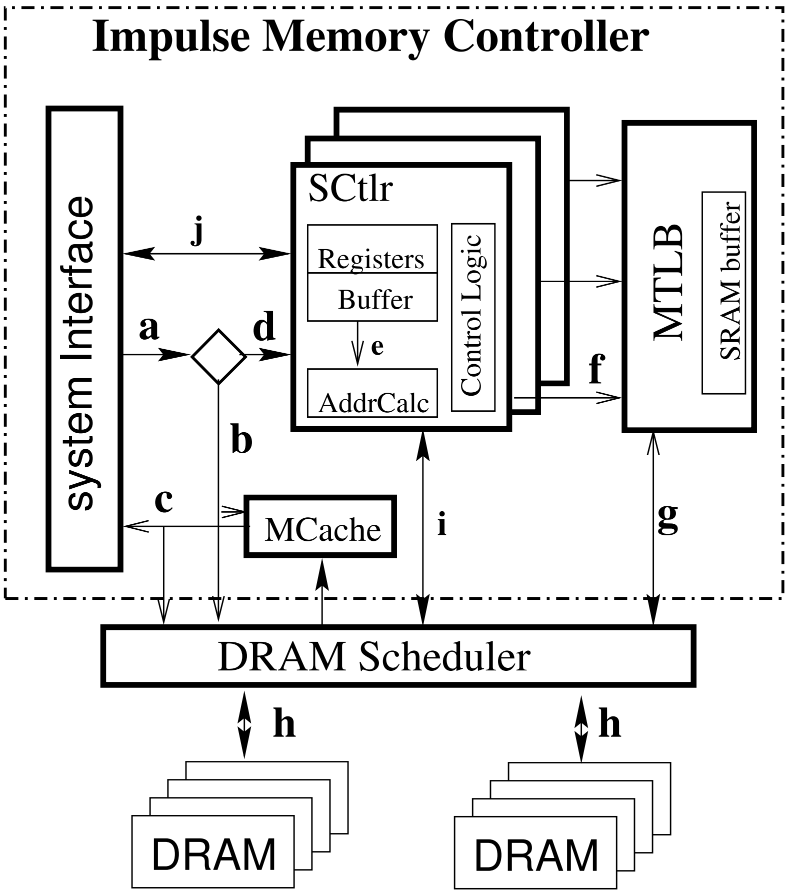


Figure 1: The internal architecture of the Impulse memory controller. The arrows indicate how data flows within an Impulse system.

Figure 1 shows the internal architecture of the Impulse memory controller, which includes the following components:

- a small number of *Shadow Controllers (SCtlr)*[1], each of which contains several *registers* to store remapping configuration information, a small *SRAM buffer* to scatter/gather data, a simple *ALU unit (AddrCalc)* to translate shadow addresses to pseudo-virtual addresses, and some *control logic* to control the flow;
- a *Memory Controller TLBs (MTLB)*, which are backed up by main memory and map pseudo-virtual addresses to physical DRAM addresses, along with a small number of buffers to hold page table entries fetched from DRAM;
- a *Memory Controller Cache (MCache)*, which buffers data prefetched from DRAM.

An address appearing on the system memory bus may be a real physical address or a shadow address (**a**). A real physical address is passed untranslated to the MCache(**b**). A shadow address must go through the matching shadow controller (**d**). The AddrCalc unit in the shadow controller translates the shadow address into a set of pseudo-virtual addresses using the configuration data stored in the control registers (**e**). These pseudo-virtual addresses are translated into real physical addresses by the MTLB (**f**). The real physical addresses then are passed to the MCache (**g**). If an access misses in the MCache, it is passed to the DRAM scheduler. The DRAM scheduler orders and issues the DRAM accesses (**h**) and sends the data back to the matching shadow controller (**i**) (for shadow addresses) or system interface (**c**) (for non-shadow addresses). Finally, the appropriate shadow controller assembles the data into a cache line and sends it to the system interface (**j**).

1.3 Assumptions and restrictions

We assume the Impulse memory controller is used in a system with the following features:

- 4K-byte base page, (maybe 16Kbyte later);
- 44-bit virtual address;
- 40-bit physical address;
- 128-byte L2 cache line.

Impulse applies the following restrictions.

[1]Shadow controller is what we used to call "shadow descriptor".

- Shadow address format:

39	38	37 ... 32	31 ... 0
1	**1**	**shaodw controller index**	

- Maximum size of each remapped virtual region: 16 Gbytes (2^{34}).
- Maximum shadow region for each shadow controller: 4 Gbytes (2^{32}).
- Any object to be scattered/gathered must meet the following requirements:
  - It must be no greater than a cache line[2] and no less than 4 bytes;
  - Its size must be a power of 2;
  - It can not cross cache-line boundary.
- The stride size must be a multiple of a cache line size.
- The starting virtual address of a remapped virtual region must be page-aligned.
- A shadow region must start from page boundary.
- The memory controller page table must be page-aligned too.

[2] In this document, a cache line means a line of the lowest cache level, or say a block from the system bus's point of view.

2 Shadow Controllers

2.1 Internal structure

Each shadow controller has equivalent functionality and supports all the remapping algorithms developed so far. Each shadow controller contains the following components:

- a small number of control registers to store configuration data;
- a small SRAM buffer to scatter/gather data;
- a cache-line-sized SRAM to store elements of the indirection vector in *scatter/gather mapping through an indirection vector*;
- an ALU unit to translate shadow addresses to pseudo-virtual addresses using the configuration data stored in the control registers;
- control logic.

2.1.1 Control registers

The control registers must be set with appropriate values before any corresponding shadow addresses reach the shadow controller. They are memory-mapped and set by the processor through *uncached store* operations. The number of control registers that different remapping algorithm requires is different. The following sections will describe the minimum configuration data needed by each remapping algorithm.

2.1.2 ALU unit

The ALU unit calculates pseudo-virtual addresses. It can perform only the following simple operations:

- addition: 32-bit + 32-bit $\Rightarrow$ 32-bit, without overflow detection;
- subtraction: 32-bit $-$ 32-bit $\Rightarrow$ 32-bit, with overflow detection;
- multiplication: 32-bit $\times$ 32-bit $\Rightarrow$ 32-bit, without overflow detection;

- shift: 32-bit << 4-bit, without overflow detection;
- masking: 32-bit & 32-bit ⇒ 32-bit.

2.1.3 Control Logic

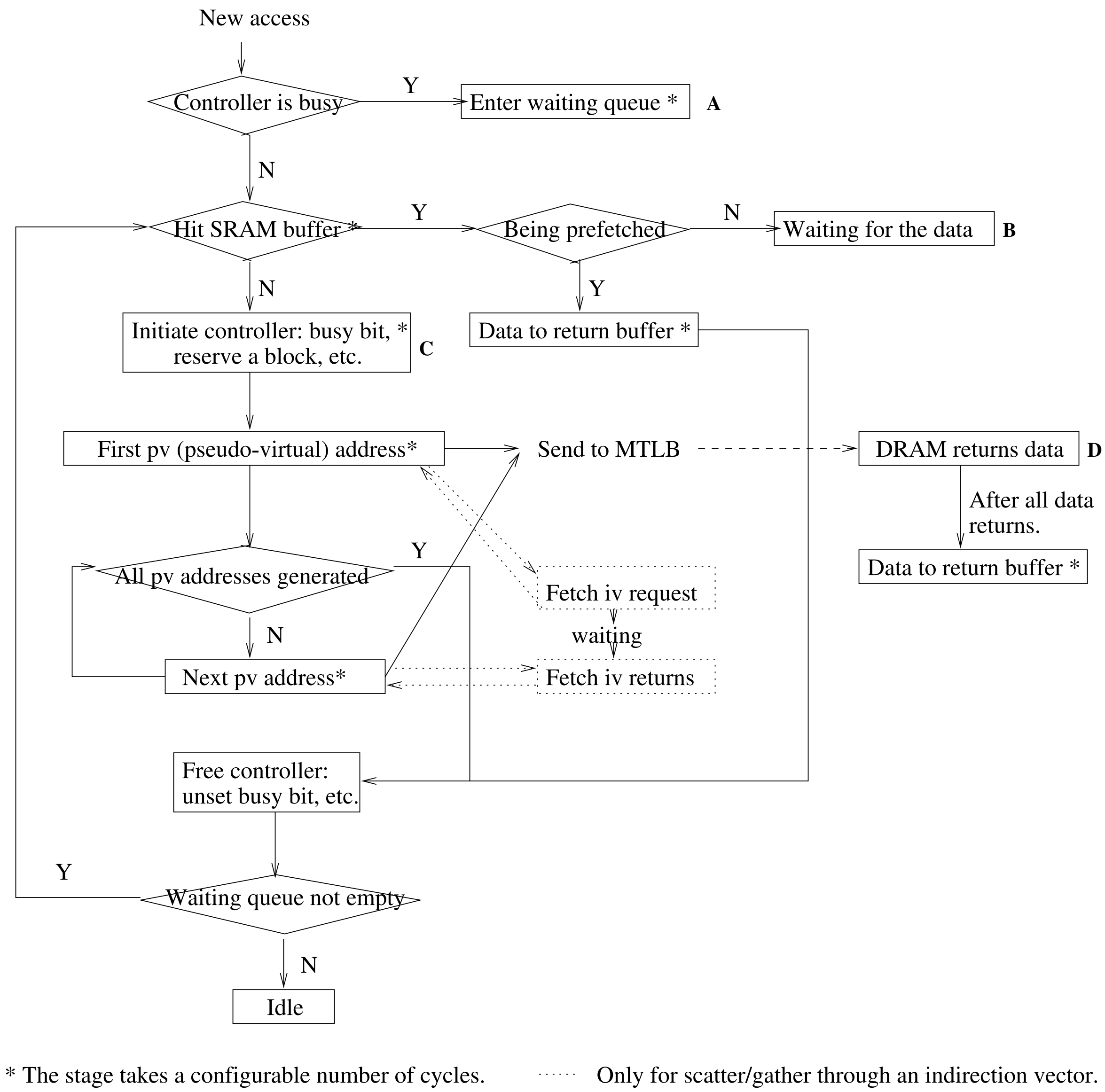


Figure 2: Flow model of shadow controller.

Figure 2 shows the control flow model of the shadow controller.

2.2 Supported remapping algorithms

Currently, the shadow controllers support the following types of remapping:

- no-copy superpage formation;
- strided mapping;
- no-copy page-color mapping;
- scatter/gather mapping through an indirection vector;
- transpose mapping.

The following sections describe the configuration data required by each type of remapping algorithm and how the configuration data is used to compute pseudo-virtual addresses.

2.2.1 No-copy superpage formation

This mapping creates superpages for disjoint physical pages. It maps one contiguous cache line in the shadow address space to one contiguous cache line in real physical memory.

Configuration parameters

Name	Bits	Description
map_type	8	DIRECT_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch
pref_count	16	prefetch distance, in bytes
saddr_start	32	bits 0 – 31 of starting shadow address
saddr_size	32	size of the remapped shadow region, in bytes
ptable_ptr	28	starting physical page of the MC page table

Address generation

First pseudo-virtual address[3] (Figure 3): (Assuming receiving shadow address *saddr*)
$saddr - saddr\_start$.

Next pseudo-virtual address: No.

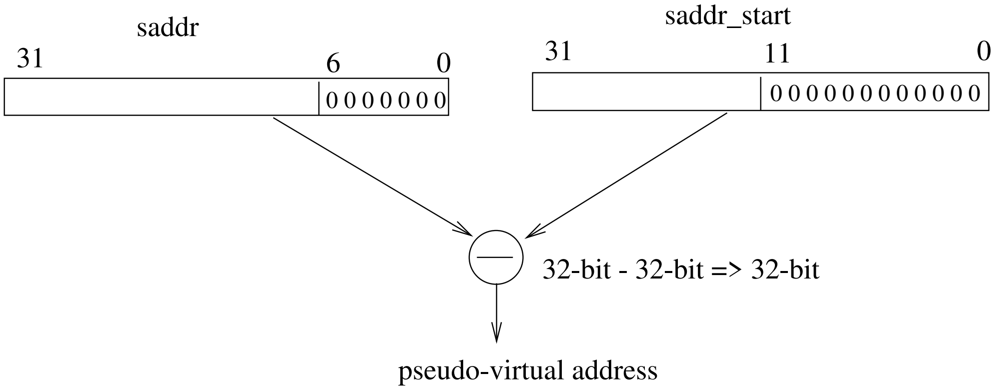


Figure 3: Computation of the first pseudo-virtual address for direct mapping.

2.2.2 No-copy page-color mapping

This mapping maps a virtual region to appropriate shadow regions so that data inside this virtual region goes to only the designated portion of a physically indexed cache.

An example

Figure 4 shows how this mapping is used. This example maps data structure **A** to the third quadrant of a physically-indexed L2 cache. The operating system first allocates a shadow address space four times of the L2 cache and then creates a page table in the CPU to map each quarter of **A** to an appropriate region in the allocated shadow address space, as shown by Figure 4. Assuming the allocated shadow address space is L2-cache-size-aligned, all the grey boxes in the shadow address space are mapped into the same portion of the L2 cache. Note that the white spaces in the shadow address space are wasted in this design. Since shadow addresses are not directly backed up by real physical memory, wasting them does not actually waste any real physical memory.

Configuration parameters

[3]Refer to **First pv address** and **Next pv address** stages in Figure 2.

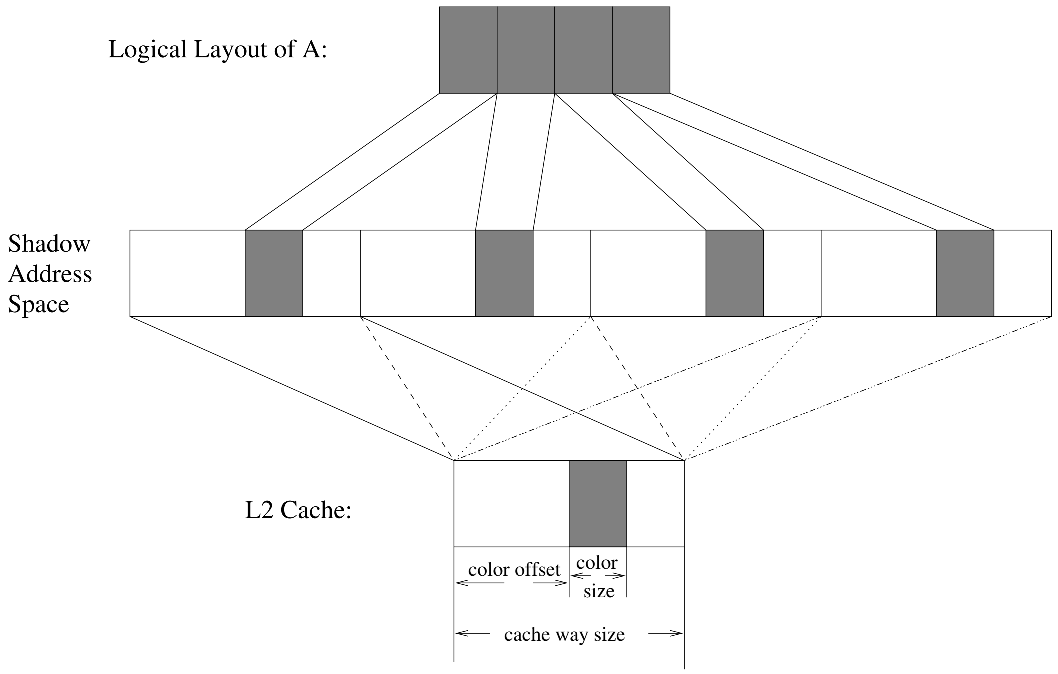


Figure 4: Map A into the third quadrant of L2 cache

Name	Bits	Description
map_type	8	PAGECOLOR_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch
pref_count	16	prefetch distance, in bytes
saddr_start	32	bits 0 – 31 of the starting shadow address
saddr_size	32	size of the remapped shadow region, in bytes
color_size	32	size of the designated color, in bytes.
way_size	32	cache blocking factor (size/associativity), in bytes
color_offset	32	offset of the color in a cache way, in bytes
ptable_ptr	28	starting physical page of the MC page table

Address generation

First pseudo-virtual address (Figure 5):

$$(saddr - saddr\_start)/waysize \times colorsize +$$
$$(saddr - saddr\_start)\%waysize - coloroffset.$$

Next pseudo-virtual address: No.

Although the mathematic formula seems complicated, Figure 5 shows how easily the translation can be done in hardware. Note that both *way_size* and *color_size* must be a power of two multiple of base pages. The figure shows operations that can possibly be performed in parallel at parallel positions. It is up to hardware design team to decide whether to actually perform them in parallel.

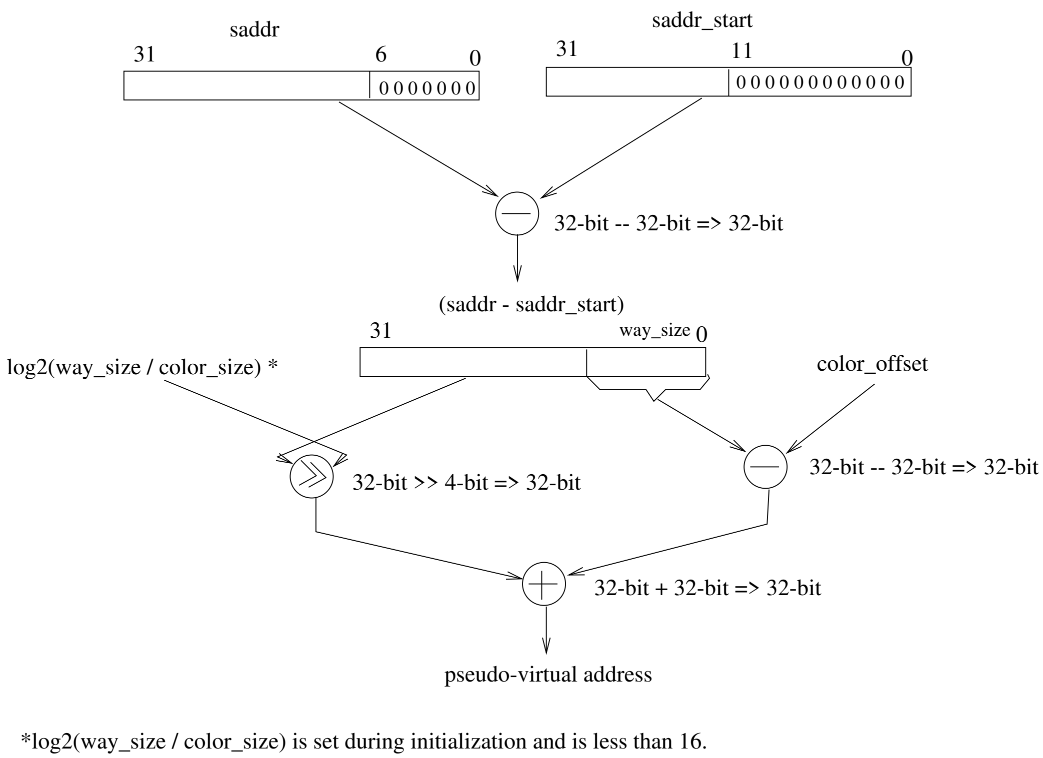


Figure 5: Computation of the first pseudo-virtual address for page-color mapping.

2.2.3 Stride mapping

This mapping creates dense cache lines from data items whose virtual addresses are distributed in a uniform stride.

Configuration parameters

Name	Bits	Description
map_type	8	STRIDE_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch
pref_count	18	prefetch distance, in bytes
saddr_start	32	bits 0 – 31 of the starting shadow address
saddr_size	32	size of the remapped shadow region, in bytes
stride_size	16	stride size, in bytes
object_size	12	object size, in bytes
object_count	32	number of objects
object_offset	12	offset of the required object in a stride
ptable_ptr	28	starting physical page of the MC page table

Address generation

First pseudo-virtual address (Figure 6):
$(saddr - saddr\_start)/object\_size \times stride\_size + object\_offset.$

Next pseudo-virtual address:
previous_one $+ stride\_size.$

2.2.4 Scatter/Gather mapping using an indirection vector

It packs dense cache lines from array elements according to an indirection vector.

Configuration parameters

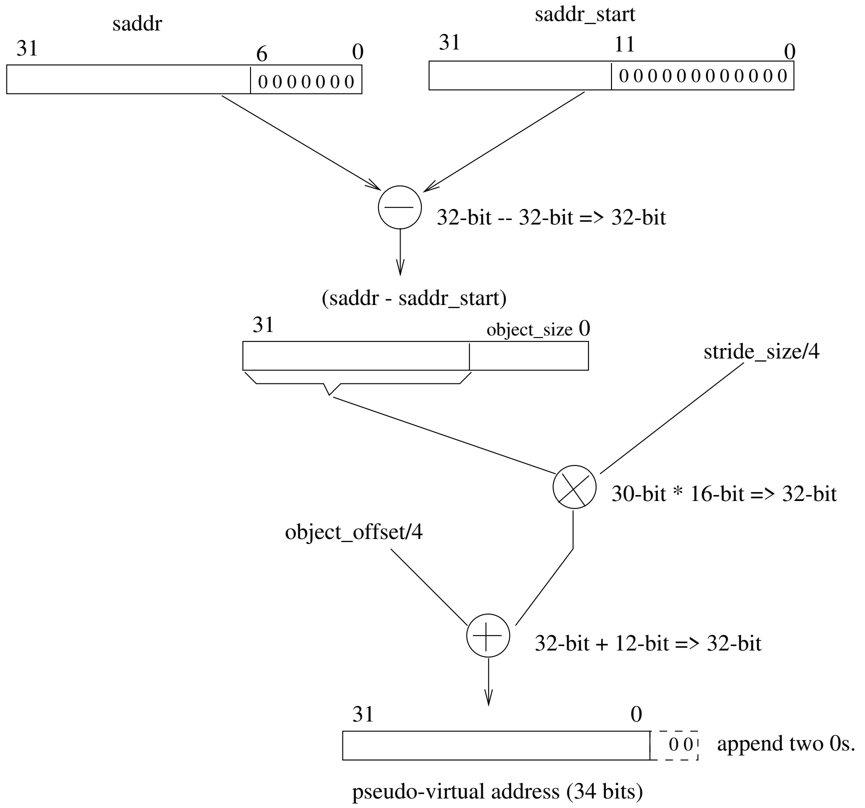


Figure 6: Computation of the first pseudo-virtual address for stride mapping.

Name	Bits	Description
map_type	8	INDIRVECTOR_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch
pref_count	16	prefetch distance, in bytes
saddr_start	32	bits 0 – 31 of the starting shadow address
saddr_size	32	size of the remapped shadow region, in bytes
object_size	12	object size, in bytes
object_count	32	number of objects.
iv_paddr	28	starting physical page of the indirection vector
iv_elemsize	3	element size of the indirection vector
iv_objcount	32	number of objects in the indirection vector
fortran_sub	1	C style or Fortran style array subscript
ptable_ptr	28	starting physical page of the MC page table

Address generation

First pseudo-virtual address: (Figure 7)

$$index = (saddr - saddr\_start)/object\_size;$$
$$(iv[index] - fortran\_sub) \times object\_size.$$

Next pseudo-virtual address:

$$(iv[++index] - fortran\_sub) \times object\_size).$$

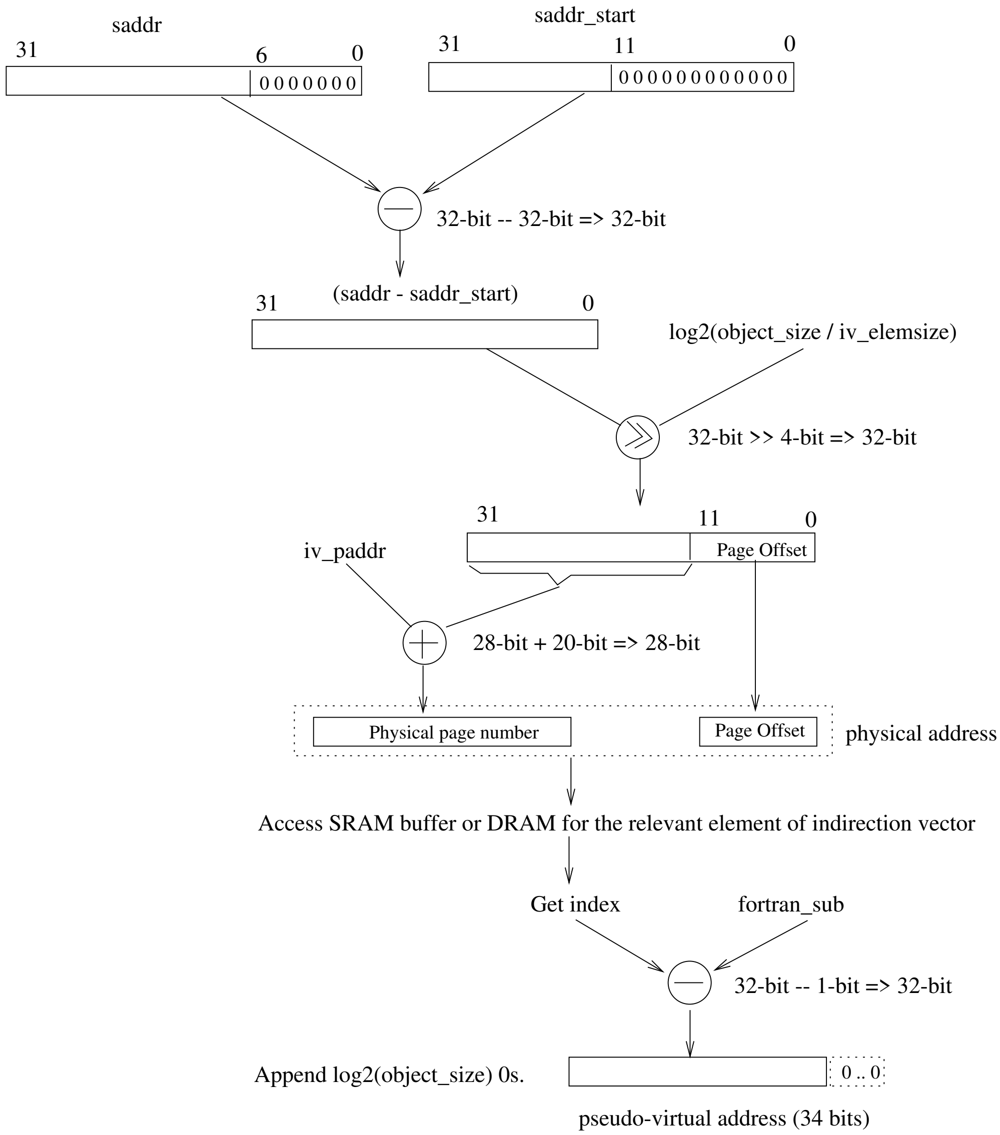


Figure 7: Computation of the first pseudo-virtual address for scatter/gather through an indirection vector.

2.2.5 Transpose mapping

This type of mapping creates the transpose of a two-dimensional matrix by mapping element *[j][i]* of the transposed matrix to element *[i][j]* of the original matrix.

Configuration parameters

Name	Bits	Description
map_type	8	TRANSPOSE_MAPPING
pref_info	2	prefetch forward, or backward, or no prefetch.
pref_count	16	prefetch distance, in bytes
saddr_start	32	bits $0-31$ of the starting shadow address
saddr_size	32	size of the remapped shadow region, in bytes
elem_size	12	size (a power of 2) of an array element, in bytes
row_size	32	size of each row, in bytes
row_num	32	number of rows in the array. Must be a power of 2
ptable_ptr	28	starting physical page of the MC page table.

Address generation

First pseudo-virtual address (Figure 8):

$offset = (saddr - saddr\_start)/elem\_size;$

$offset\%row\_num \times row\_size + offset/row\_num \times elem\_size.$

Next pseudo-virtual address:

previous_one $+row\_size.$

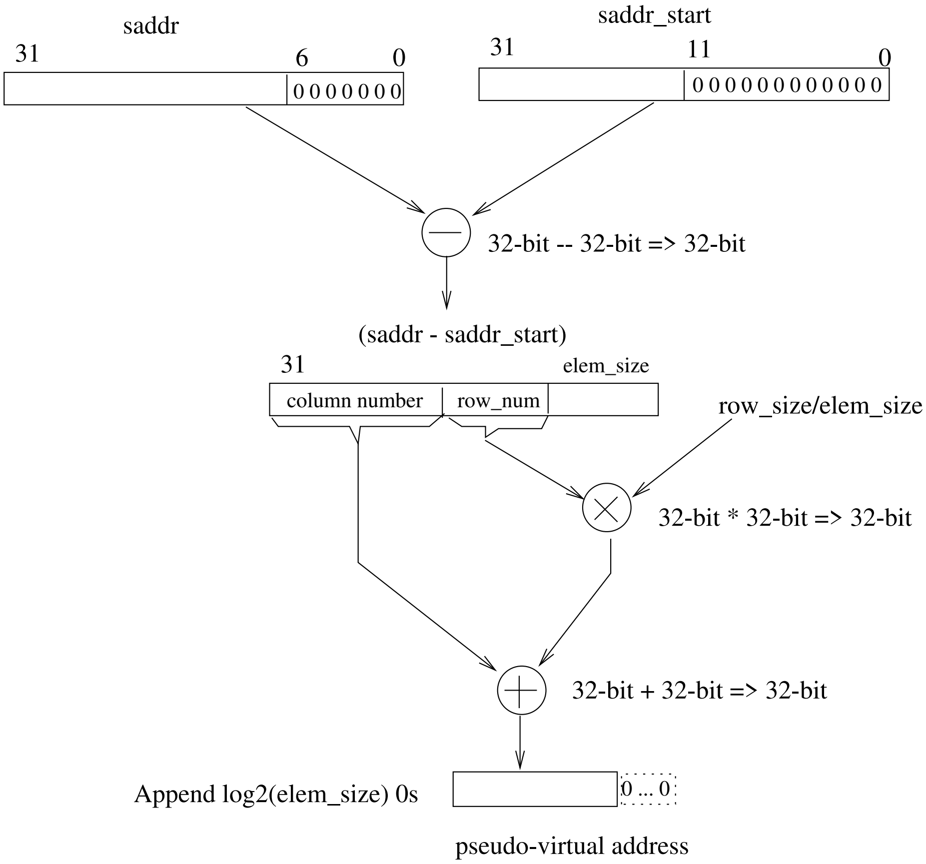


Figure 8: Computation of the first pseudo-virtual address for transpose mapping.

3 Memory Controller TLB

The MTLB is responsible for the mapping from pseudo-virtual addresses to physical DRAM addresses.

3.1 Architecture

When an application issues an Impulse system call, the operating system creates a dense, flat page table to store the pseudo-virtual-to-physical translations of the data structure being remapped. We refer to this page table as the *memory controller page table*. Each 4-byte entry of the memory controller page table has the following format:

valid (1)	ref (1)	modify(1)	fault(1)	frame (28)

The *valid* bit indicates whether this mapping is valid. The *reference* bit indicates whether a page has been referenced. This bit is set on the first MTLB miss for the page. The *modify* bit indicates whether a page has been written. This bit is set on the first write reference for the page. The *fault* bit indicates whether the page is in main memory. The *frame* is physical page number. Assuming 40-bit physical address and 4kilobyte page size, *frame* has 28 bits.

In the simulator, the MTLB has configurable size and associativity, uses a Not Recently Used (NRU) replacement policy, and has a one-cycle access latency. Each entry of the MTLB has the following format:

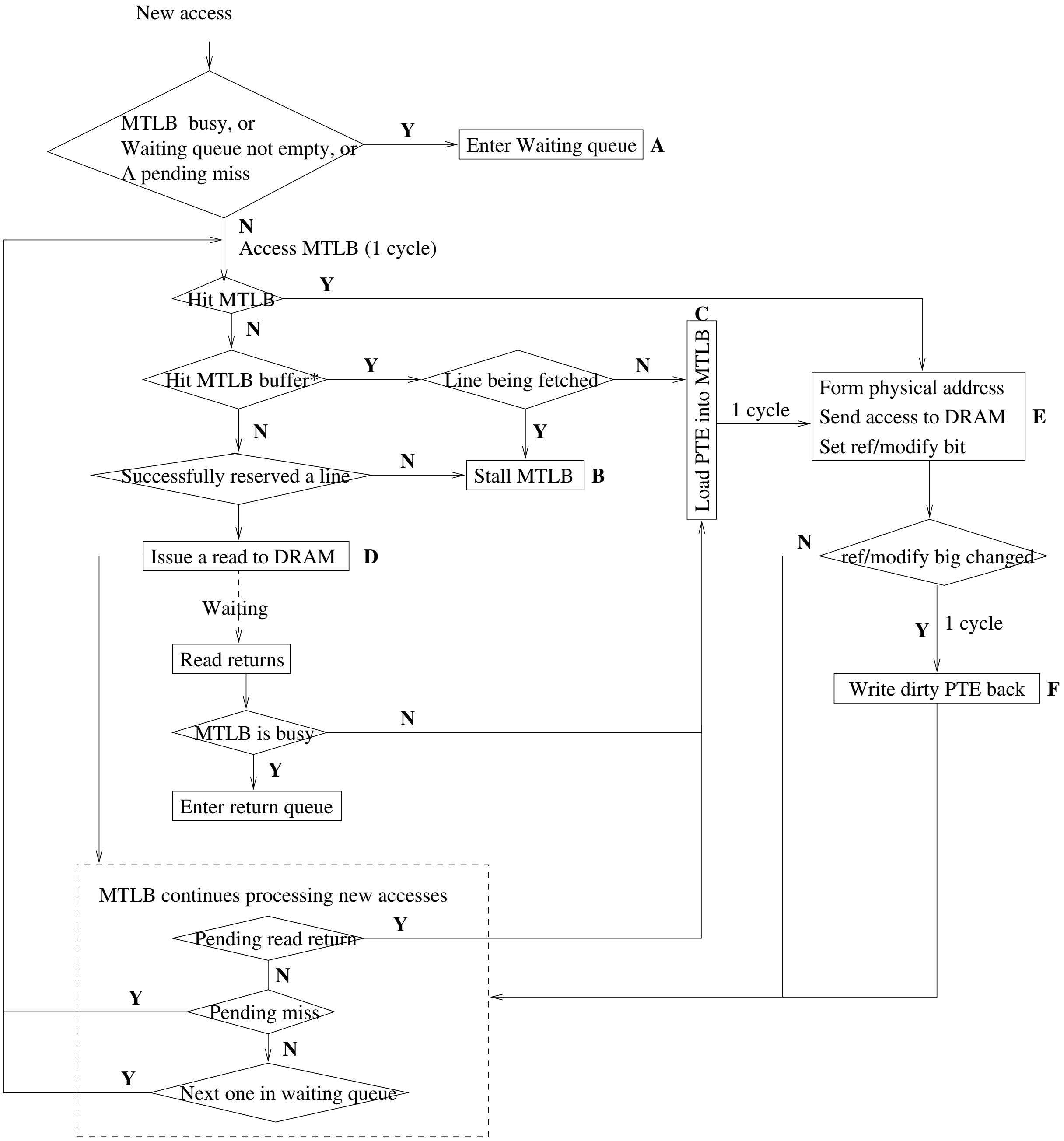
valid (1bit)	locked (1bit)	tag (22bits)	refcount (2-4bits)	PTE (4 bytes)

The *valid* bit indicates whether or not this mapping is valid. The *locked* bit indicates whether or not this entry is reserved for an ongoing write-back transaction. A *tag* is formed by a pseudo-virtual page number and the index number of the shadow controller that generated this pseudo-virtual address. The *refcount* records the total number of references to the page. It is used to implement the NRU replacement policy when the MTLB is not direct-mapped.

A small buffer inside the MTLB is used to cache page table entries loaded from physical memory. Each MTLB miss checks this buffer first before sending a fill request to DRAM. If an MTLB miss hits in the buffer, it only takes one extra cycle to load the translation into the MTLB. If it misses in the buffer, the MTLB generates a fill request to load a cache line worth of page table entries from physical memory.

3.2 Flow model

Figure 9 shows the control flow model of the MTLB.



* MTLB and its buffer are accessed in parallel.

Figure 9: Flow model of the MTLB.

4 Memory Controller Cache

4.1 MC-based prefetching

An important feature of Impulse is its supporting for prefetching at the memory controller – MC-based prefetching. The MC-based prefetching preloads data from DRAM into a modest SRAM cache (so-called MCache) in the MMC.

When MC-based prefetching is turned on, each access first checks the MCache for a match. If it hits in the MCache, the Impulse MMC can quickly move the requested data to the system interface without going through a full DRAM access (which contributes the majority of a memory latency). MC-based prefetching is very important for shadow accesses. Each shadow access must go through the shadow controller, which may take from several cycles to hundreds of cycles. It is crucial for the Impulse MC to start loading shadow data as early as possible to hide the cost of remapping.

The MC-based prefetching currently implements a simple **next-line sequential prefetching algorithm**. A prefetching transaction is issued in the following two situations:

- When a prefetched line is hit by a demanding request, prefetch the next line;
- When a request misses in the MCache, fetch the requested line and prefetch the next line;

4.2 MCache organization

The MCache uses a FIFO replacement policy. The MCache is quite different from the CPU caches. Since the most frequently used data should reside in the CPU caches, data in the MCache should normally not be used frequently. Replacement policies such as LRU and NRU simply do not work well with the MCache. Previous experiments show that FIFO outperforms LRU in all programs that we have tested.

In general, the MCache has the following features:

- Physically indexed and physically tagged;
- Configurable size and set-associativity;
- FIFO replacement policy;

- Write-invalidate protocol. Since writes invalidates the matched data in the MCache, data in the MCache can never be dirty, which means that conflict victims can simply be discarded.

Each MCache line has the following format:

used (1bit)	state (1bit)	pref (1bit)	physical tag (25bits)	data (128)

The *used* bit indicates whether or not this line is in use. The *state* bit indicates the state of this line — either *Fetching* or *Valid*. *Fetching* means that the data is being fetched right now but has not returned from physical memory. After the fetched data has returned, the *state* bit will be changed to *Valid*. The *pref* bit indicates whether or not it is a prefetched line never being used. When a non-prefetch access hits a prefetched line, this bit is cleared. The *tag* is formed by extracting bits 7 – 31 of a shadow address.

To avoid generating duplicate DRAM accesses when a cache line being fetched is also requested by a processor, a line is reserved and its tag is set when a transaction is issued. If an access needs a line that an ongoing transaction is fetching, it will hit in the MCache and wait for the return of the desired data. A line reserved for an ongoing transaction will not be victimized before the requested data returns. If all lines that a transaction can use are occupied by other ongoing transactions, this transaction will stall the shadow controller pipeline if it is a normal transaction, or be discarded if it is a prefetch transaction.