SET ASSOCIATIVE DATA TLB FOR SHAKTI C-CLASS PROCESSOR

A Project Report

submitted by

GURAJALA BHAVITHA CHOWDARY

in partial fulfilment of the requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY & MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.

JUNE 2022

THESIS CERTIFICATE

This is to certify that the thesis titled **SET ASSOCIATIVE DATA TLB FOR SHAKTI C-CLASS PROCESSOR**, submitted by **GURAJALA BHAVITHA CHOWDARY**, to the Indian Institute of Technology, Madras, for the award of the degree of **DUAL DEGREE**, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. V. Kamakoti
Research Guide
Professor
Department of Computer Science and Engineering
IIT Madras, 600 036

Prof. Nitin Chandrachoodan
Research Co-Guide
Associate Professor
Department of Electrical Engineering
IIT Madras, 600 036

Place: Chennai

Date: 16th June, 2022

ACKNOWLEDGEMENTS

I would like to express my gratitude towards my project guide, Prof. V. Kamakoti for giving me the opportunity to work on this project. His extensive knowledge and experience in this field has been a great source of inspiration. I would also like to acknowledge Prof. Nitin Chandrachoodan as the co-guide of this project and I am grateful for his support. I would also like to express my deepest gratitude to my project mentor, Neel Gala for his constant support throughout the course of my project.

I would like to thank my parents and friends for their encouragement and unwavering support throughout my education and research.

ABSTRACT

KEYWORDS: TLB; MMU; Virtualisation; Set associative mapping

Shakti C-class currently has a fully associative data TLB. This project explores the idea of a set associative TLB, that is flexible with the associativity of the mapping. The TLB is also designed to support multiple page sizes. The replacement policies explored are Round robin and Random replacement policies.

Source Code is in Bluespec System Verilog. Compilation and verilog Generation were done using BlueSpec Compiler. It is tested and then integrated with C-class.

ABBREVIATIONS

MMU	Memory Management Unit
TLB	Translation Lookaside Buffer
LRU	Least Recently Used
OS	Operating System
VPN	Virtual Page Number
BSV	Bluespec System Verilog
BSC	Bluespec System Compiler
LFSR	Linear Feedback Shift Register

CHAPTER 1

INTRODUCTION

C-Class is a member of the SHAKTI family of processors. It is an extremely configurable and commercial-grade 5-stage in-order core supporting the standard RV64GCSUN ISA extensions. C-Class now has separate fully associative I-TLB and D-TLB. This project explores the possibility of set associative data TLB for C-Class while also having the provision to apply fully associative mapping or direct mapping based on the number of ways and sets needed and inputting that to the code. The entire code is written in Bluespec System Verilog.

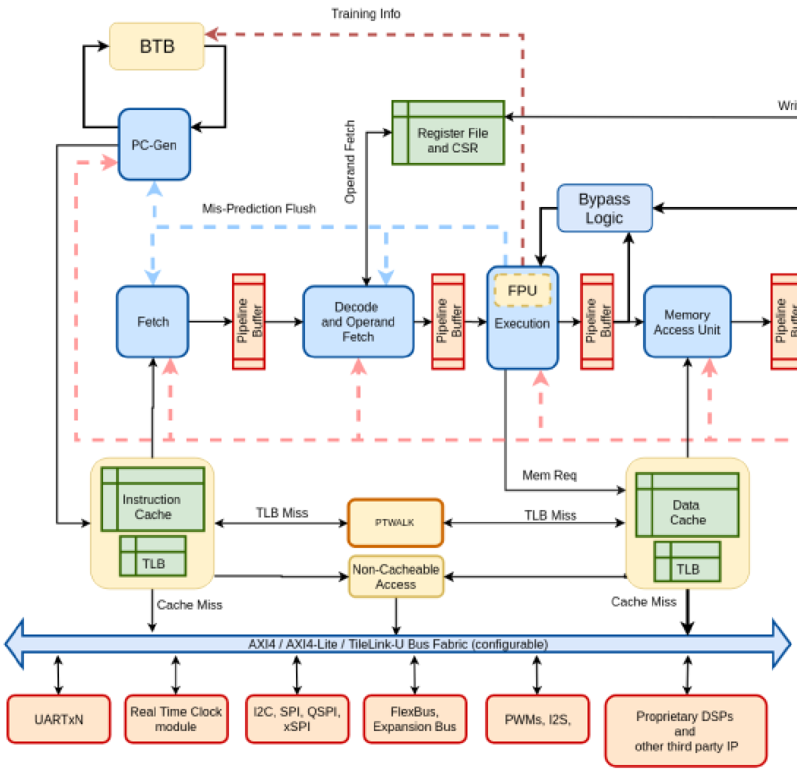

Figure 1.1: Block Diagram of Shakti C-class core

CHAPTER 2

Main components

2.1 Virtual Memory

Virtual memory, or virtual storage is a memory management technique that provides an idealized abstraction of the storage resources that are actually available on a given machine which creates the illusion to users of a very large memory.
Uses of Virtual Memory:

- Virtual Memory allows users to write code as if it has total unrestricted control of the entire memory range, regardless of the memory usage of any other program
- Virtual Memory also provides protection and isolation as it does to let malicious code touch the memory spaces of other running programs
- Virtual memory improves efficiency as it allows to undersubscribe or oversubscribe the memory space.

2.2 Block Diagram of Virtual Memory Organization

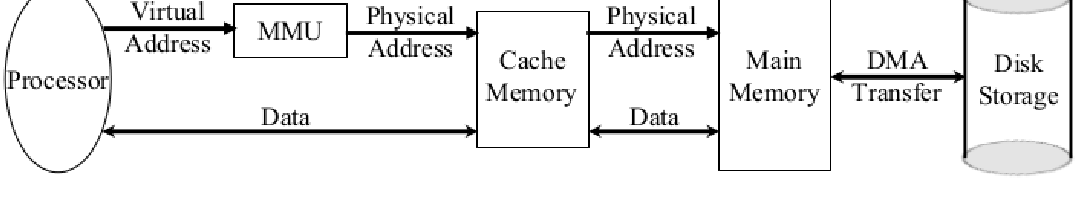

Figure 2.1: Block Diagram of Virtual memory management

2.3 MMU

A memory management unit (MMU) is a computer hardware component that handles all memory and caching operations associated with the processor.
MMU's operations can be divided into 3 categories:

- Hardware memory management, which oversees and regulates the processor's use of RAM (random access memory) and cache memory.
- OS (operating system) memory management, which ensures the availability of adequate memory resources for the objects and data structures of each running program at all times.
- Application memory management, which allocates each individual program's required memory, and then recycles freed-up memory space when the operation concludes.

2.4 Virtual memory address translation

The idea behind virtual memory is that physical memory is divided into fixed size pages. Pages are typically 512 to 8192 bytes, with 4096 being a typical value. Page size is virtually always a power of two. Pages frames are loaded into memory only when they are needed. Adjacent page frames are not necessarily loaded into adjacent pages in memory. At a point in time during the execution of a program, only a small fraction of the total pages of a process may be loaded.

It is possible that a process might access a page which is not in memory. The MMU will determine that this page is not loaded, and this will generate a page fault. A page fault is an interrupt. The handler for a page fault locates the needed page frame on the disk, copies it to a page in memory, and updates the page table, telling it where the page frame is loaded. Control then returns to the process.

In Fig. 2.1, we can see that the virtual address coming from the processor is converted to physical address by MMU.
The physical address obtained from processor is split into Virtual Page Number(VPN) and offset. VPN added to the page table base address is then used to compare with the

control bits of page table and the corresponding page frame is extracted and concatenated with offset to produce the physical address. This can be seen as a block diagram in the 2.2. This process of searching the page table for physical address is also called as Page Table Walk, also referred to as PTwalk henceforth.

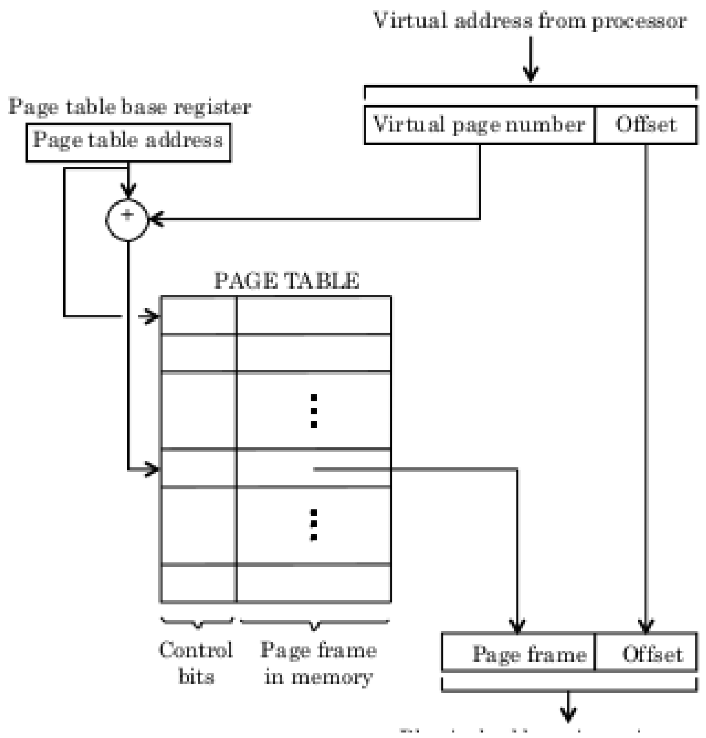

Figure 2.2: Virtual Address –> Physical Address

2.5 TLBs

Translation Lookaside Buffers(TLBs) are used for performance enhancement in this address translation discussed above. They are cache-like structures. They can be called address translation cache. These are based on the principles of temporal and spatial locality which most programs conform to. Most processes, no matter how large they are, are usually only executing steps in a small part of the text segment and accessing only a small fraction of the data during any given time short time period. This means that they are accessing only a small number of pages. The idea behind a TLB is that these pages are cached.

If the requested address is present in the TLB, the search yields a match quickly and the retrieved physical address can be used to access memory. This is called a TLB hit.

If the requested address is not in the TLB, it is a miss, and the translation proceeds by looking up the page table in a process called a page table walk as discussed in the previous section. The page walk is time-consuming when compared to the processor speed, as it involves reading the contents of multiple memory locations and using them to compute the physical address. After the physical address is determined by the page walk, the virtual address to physical address mapping is entered into the TLB. The entire relationship between TLB, MMU and Cache can be seen in the 2.3 below:

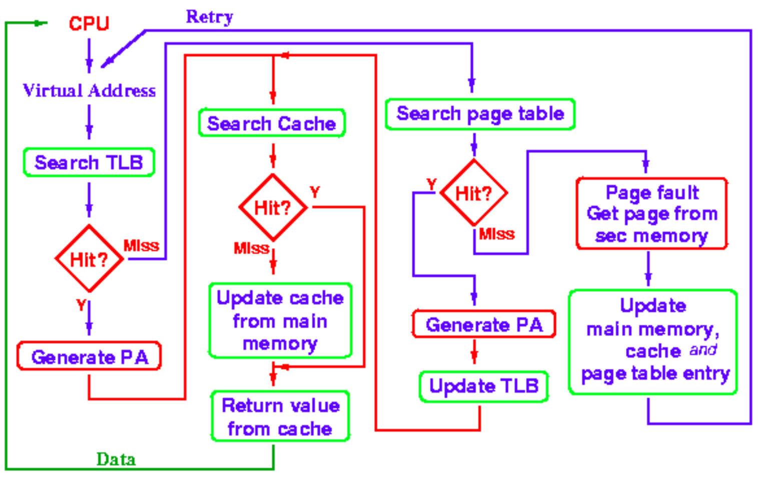

Figure 2.3: TLB operation

CHAPTER 3

Design of TLB

3.1 Associativity of TLB

The design aspects of TLB are important for the performance of processor.

- The bigger the size of TLB:
  - The higher the hit rate
  - The slower the response
  - The greater the expense
- The larger the page size
  - the size of the page table is inversely proportional to the page size
  - transferring larger pages to/from secondary storage is more efficient than transferring smaller pages
  - reduces the number of TLB misses.

The present TLB in Shakti C-Class processor is Fully Associative.

Fully Associative

Fully Associative Mapping refers to a technique of cache mapping that allows mapping of the main memory block to a freely available cache line. In case the data and information are fetched from memory, then they can be placed in an unused block of a cache. Hence, whenever a search is needed, comparison has to done

The major disadvantage of the fully associative implementation is the amount of hardware needed for the comparison increases in proportion to the cache size and hence, limits the capacity after a point.

N-way Set associative mapping

N-way set associative mapping restricts the entry of blocks into TLB lines. An N-way set associative mapping provides N blocks in each set where data mapping to that set might be found. Each memory address maps to a specific set, but it can map to any one of the N blocks in the set. When N=1, it is called direct mapping.

So each address is divided into Tag, Set and Block offset. The tag is only compared to the tags present in the corresponding set in TLB.

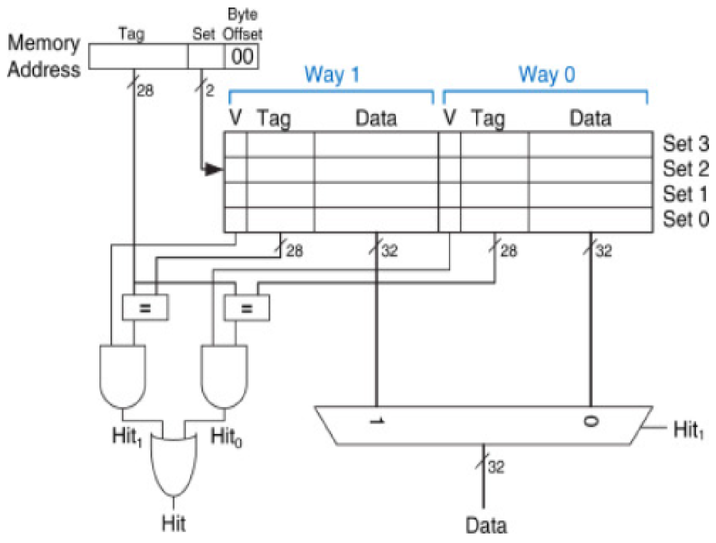

Figure 3.1: Set Associative Mapping

3.2 Separate TLBs for Different Page Sizes

TLBs must be fast and energy-efficient. Performance is desirable as TLBs reside on the critical datapath of pipelines. This means that they are usually built in a simple manner that meets timing constraints, with lookup and miss handling characteristics that are amenable to speed.

Large pages can dramatically reduce the frequency of TLB misses. There are, however, several reasons why typical page sizes have not increased. One of them being, larger pages result in larger working sets due to internal fragmentation, i.e., memory wasted due to the page size being larger than what the program needs. For improving TLB performance we use two page sizes and to require page sizes to be powers of two and pages to be aligned (i.e., a page of size B must be placed in virtual and physical memory at an address that is a multiple of B). If the large page size can be judiciously used, this method can make the TLB map more memory without a significant impact on working set size and no effect on minimum protection granularity.

However, supporting multiple page sizes also complicates the design of set-associative TLBs. The reason is that different page sizes require a different number of page offset bits. 4KB baseline pages require 12 page off- set bits, 2MB large pages require 21 page offset bits, while 1GB large pages require 30 page offset bits. To reduce conflict misses in set-associative TLBs, we use the least significant bits of the virtual page number as showin in 3.2.

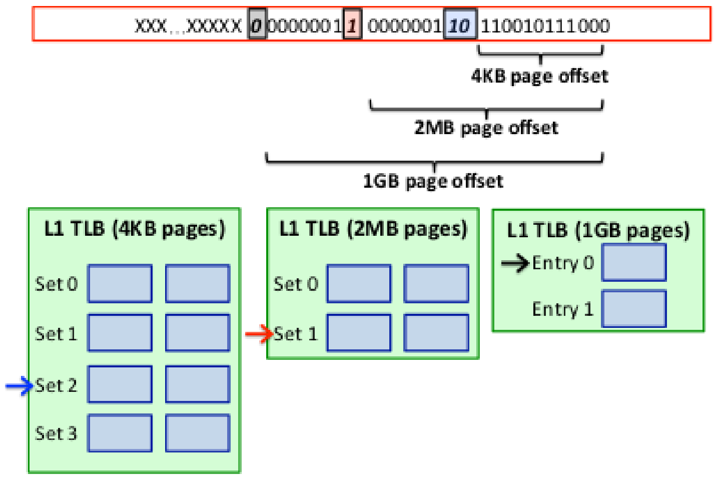

Figure 3.2: Multiple Page Sizes

As the page size cannot be identified by looking at the virtual address, a parallel lookup is done in all the L1 TLBs. A miss is reported if it's not found in any of the lookups. Then a page table walk is done and the page is placed in the appropriate TLB. Hardware in MMU aids in the determination of page size.

3.3 Replacement policies for TLB

When a Page table walk occurs, that page has to be placed in the respective TLB. If the set that the page belongs to is completely occupied, we have to replace an existing line. There are different policies for that, some policies used in this project are discussed below:

1. Round Robin Replacement policy

   Round-robin or cyclic replacement means there is a counter that cycles through the available ways and cycles back to 0 when it reaches the maximum number of ways.

2. Random Replacement policy

   A random number is generated and that page is replaced with the new page. LFSR is used to generate this random number in this code

3. LRU

   Least Recently Used (LRU) is to remove the least recenlty used page among the pages present and replace it with the new entry.

3.4 Testing

A test bench for bluespec was written and was used to tested some scenarios by changing the number of ways and sets, the code passed all the cases.

3.5 Integration with C-class

The code was integrated with C-class, macros were enabled for compiling and it was built successfully.

CHAPTER 4

Code Explanation

4.1 Defining interfaces and other variables

Defining the necessary interfaces, variables, vectors and registers for the code.
Interfaces, methods are defined for the following:

- Interfaces to send the core request
- When there's a tlb miss, sending and receiving the request from ptwalk and receive the response
- Methods to receive the current:
  - satp csr from the core
  - current privilege mode of operation
  - current values of the mstatus register

Each RegFile can be understood as a set, Vector of regfile indicates all the sets that make up the tlb. The arguments are given accordingly.

- pagesize1 refers to 4KB, takes the lower 12 bits of virtual address for page offset
- pagesize2 refers to 2MB, takes the lower 21 bits of virtual address for page offset

FIFOs are used for storing all the addresses and placing the requests. Variables defined and their explanation

- satp: lower bits of satp register(ppnsize) holds the ppn of root page table
- ASID: address space identifier (ASID) facilitates address-translation fences on a per-address-space basis
- For sv32, satp_mode=0, no translation, if satp_mode=1, Page-based 32-bit virtual addressing
- When MXR=0, only loads from pages marked readable (R=1) will succeed.
  When MXR=1, loads from pages marked either readable or executable (R=1 or X=1) will succeed.

- When SUM=0, S-mode memory accesses to pages that are accessible by U-mode (U=1) will fault.
  When SUM=1, these accesses are permitted.
- MPP represents mode(U or S)
- When MPRV=0, loads and stores behave as normal, using the translation and protection mechanisms of the current privilege mode.
  When MPRV=1, load and store memory addresses are translated and protected, and endianness is applied, as though the current privilege mode were set to MPP

4.2 sfence

Sfence rule will simply invalidate all the tlb entries. As we now have two tlbs, both of them have to be cleared. When sfence is fired, Each regfile in the vector is initiated to a vector of zeroes in both TLB with 4KB page size and 2MB page size. After the operation is done, sfence is deactivated again until further use

4.3 When the core sends a request

TLB lookup is done in parallel by extracting the tags, if it is a TLB miss, PTwalk is done. FIFOs are used to store addresses, an address is dequed from FIFO whenever needed.
The virtual address that needs to be looked up is dequed from the FIFO. This is used to extract the page offset and tag. Set tag is obtained by truncating the required number of bits in the address after removing the page offset bits. The other bits are taken to compare the addresses present in the set. The same is done in parallel for both the TLBs.

Mask is extracted and the value is used to extract the bits. The physical address is dependent on the mask value, takes the bits of vpn or ppn based on this. bits are taken from both of them and lower_pa is decided based on the mask bits. Physical address is obtained by concatenating lower _pa and page offset.

When there's a tlb miss, ptwalk lookup is instantiated

4.4 Replacement policies

The implementations of Round Robin and Random policies:

Randon replacement policy

Random generation is done using LFSR for 8 bits. The lower bits are now used for 4KB and upper bits are used for 2MB. If all lines are valid choose one to randomly replace.

Round Robin replacement policy

Round Robin replaces the dtlb lines in a sequential order. A vector is created in the size of the number of sets and it is incremented each time a replacement is made in the set. This parameter is used to update the entry in each set.

CHAPTER 5

CODE

```
interface Ifc_sa_dtlb;

  interface Put#(DTLB_core_request#(`vaddr)) put_core_request;
  // If it's a TLB hit, it is present in either one of the TLBs,
  paddr is different in both cases(4KB, 2MB)
  interface Get#(DTLB_core_response#(`paddr1))
  get_core_response1;
  interface Get#(DTLB_core_response#(`paddr2))
  get_core_response2;

  interface Get#(PTWalk_tlb_request#(`vaddr)) get_request_to_ptw;
  interface Put#(PTWalk_tlb_response#(TAdd#(`ppnsize,10),
  `varpages)) put_response_frm_ptw;

  method Action ma_satp_from_csr (Bit#(`vaddr) s);
  method Action ma_curr_priv (Bit#(2) c);
  method Action ma_mstatus_from_csr (Bit#(`vaddr) m);
  method Bool mv_tlb_available;
`ifdef perfmonitors
  method Bit#(1) mv_perf_counters;
`endif
endinterface

 /*doc:module: */
(*synthesize*)
(*conflict_free="put_response_frm_ptw_put,
put_core_request_put"*)
```

```
module mksa_dtlb#(parameter Bit#(32) hartid) (Ifc_sa_dtlb);


  // RegFile is indexed by the first argument and holds
  the inputs of second argument size

  Vector#( `setnum1, RegFile#( Bit#(`tlbindex), Bit#(`pagesize1))
  v_set_p1 <- replicateM(mkRegFile(unpack(0))));
  Vector#( `setnum2, RegFile#( Bit#(`tlbindex), Bit#(`pagesize2))
  v_set_p2 <- replicateM(mkRegFile(unpack(0))));

  /* doc:reg: reg to indicate which tlb the replacement is done
  in, 0 indicates 4KB sized TLB, 1 indicates 2MB TLB */
  Reg#(Bit#(1)) wr_tlbchoice <- mkReg(0);
  /* doc:reg: resgister to indicate which tlb index needs to
  be used for replacement*/
  Reg#(Bit#(`tlbindex) rg_tlbind <- mkReg(0);
  /*doc:reg: register to indicate which entry need to be
  filled/replaced
    There will be multiple lines with the same tlb index,
    max(setnum1, setnum2)=setnum1 is taken to indicate the number
  Reg#(Bit#(`setnum1)) rg_replace1 <- mkReg(0);
  Reg#(Bit#(`setnum2)) rg_replace2 <- mkReg(0);
  Vector#( `setnum1, Int#(`tlbindex)) setnum1 <-
  replicateM(mkReg(0));
  Vector#( `setnum2, Int#(`tlbindex)) setnum2 <-
  replicateM(mkReg(0));



  /*doc:reg: */
  Reg#(Bit#(`vaddr)) rg_miss_queue <- mkReg(0);
  /* mkSizedFIFO takes integer depth as an argument */
  /* depth- capacity of FIFO */
  FIFOF#(PTWalk_tlb_request#(`vaddr))
  ff_request_to_ptw <- mkSizedFIFOF(2);
  FIFOF#(LookUpResult) ff_lookup_result <- mkSizedFIFOF(2);
  FIFOF#(DTLB_core_response#(`paddr)) ff_core_response <-
  mkBypassFIFOF();

  // global variables based on the above wires
  // satp-supervisor address transalation and protection
  Bit#(`ppnsize) satp_ppn = truncate(wr_satp);
  Bit#(`asidwidth) satp_asid = wr_satp[`asidwidth - 1 +
  `ppnsize : `ppnsize ];
 `ifdef sv32
  Bit#(1) satp_mode = truncateLSB(wr_satp);
`else
  Bit#(4) satp_mode = truncateLSB(wr_satp);

`endif
  Bit#(1) mxr = wr_mstatus[19];
  Bit#(1) sum = wr_mstatus[18];
  Bit#(2) mpp = wr_mstatus[12 : 11];
  Bit#(1) mprv = wr_mstatus[17];

  /*doc:reg: register to indicate that a tlb miss is
  in progress*/
  Reg#(Bool) rg_tlb_miss <- mkReg(False);

  /*doc:reg: register to indicate the tlb is undergoing an
  sfence*/
  Reg#(Bool) rg_sfence <- mkReg(False);

`ifdef perfmonitors
  /*doc:wire: */
  Wire#(Bit#(1)) wr_count_misses <- mkDWire(0);
`endif

  rule rl_fence(rg_sfence && !rg_tlb_miss &&
  !ff_lookup_result.notEmpty);
    for (Integer i = 0; i < `setnum1; i = i + 1) begin
      v_set_p1[i] <= replicateM(unpack(0));
    end
  rule rl_fence(rg_sfence && !rg_tlb_miss &&
  !ff_lookup_result.notEmpty);
    for (Integer i = 0; i < `setnum2; i = i + 1) begin
      v_set_p2[i] <= replicateM(unpack(0));
    end
    rg_sfence <= False;
    rg_replace <= 0;
    `logLevel( dtlb, 1, $format("DTLB[%2d]: SFencing Now",hartid))
  endrule

  rule rl_send_response(!rg_sfence);
    let lookup = ff_lookup_result.first;
    ff_lookup_result.deq;
    //4KB offset(12 bits)
    Bit#(12) page_offset1 = lookup.va[11 : 0];
    Bit#(`tlbindex) tag1 = truncate(lookup.va >> 12);
    Bit#(TAdd#(12, `tlbindex)) vpncomp =
    truncateLSB(lookup.va >> 12);
    //2MB offset(21 bits)
    Bit#(21) page_offset2 = lookup.va[20 : 0];
    Bit#(`tlbindex) tag2 = truncate(lookup.va >> 21);
    Bit#(TAdd#(21, `tlbindex)) vpncomp =
    truncateLSB(lookup.va >> 12);
    //Bit#(`vpnsize) fullvpn = truncate(lookup.va >> 12);
    Bit#(2) priv = mprv == 0?wr_priv : mpp;
    // `logLevel can be understood as print function
    `logLevel( dtlb , 1, $format("DTLB[%2d]: LookupResult:
    ",hartid ,fshow(lookup)))
    if(lookup.translation_done)begin
      ff_core_response.enq(DTLB_core_response
      {address: truncate(lookup.va), trap: lookup.trap ,
      cause: lookup.cause , tlbmiss: False});
    end
  else begin
     // page_fault is initiated as false
      Bool page_fault = False;
     // The cause is checked
      Bit#(`causesize) cause = lookup.access == 0
      ?`Load_pagefault : `Store_pagefault;
      // The lookup is assigned to a variable
      let pte = lookup.pte;
      Bit#(TSub#(`vaddr, `maxvaddr)) unused_va =
      lookup.va[`vaddr - 1 : `maxvaddr];
      // permissions are checked
      let permissions = pte.permissions;
      Bit#(TMul#(TSub#(`varpages ,1),`subvpn)) mask =
      truncate(pte.pagemask);
      Bit#(TMul#(TSub#(`varpages ,1),`subvpn)) lower_ppn =
      truncate(pte.ppn);
      Bit#(TMul#(TSub#(`varpages ,1),`subvpn)) lower_vpn =
      truncate(fullvpn);
      Bit#(TMul#(TSub#(`varpages ,1),`subvpn)) lower_pa
      =(mask&lower_ppn)|(~mask&lower_vpn);
      // msb bits of physical address extraction
      Bit#(`lastppnsize) highest_ppn = truncateLSB(pte.ppn);
      // the physical address is finally obtained after the
      trucation of the above things
    `ifdef sv32
  Bit#(`vaddr) physicaladdress = truncate({highest_ppn,
  lower_pa, page_offset});
`else
  Bit#(`vaddr) physicaladdress = zeroExtend({highest_ppn,
  lower_pa, page_offset});
`endif

  `logLevel( dtlb, 2, $format("DTLB[%2d]:
  mask:%h",hartid,mask))
  `logLevel( dtlb, 2, $format("DTLB[%2d]:
  lower_ppn:%h",hartid,lower_ppn))
  `logLevel( dtlb, 2, $format("DTLB[%2d]:
  lower_vpn:%h",hartid,lower_vpn))
  `logLevel( dtlb, 2, $format("DTLB[%2d]:
  lower_pa:%h",hartid,lower_pa))
  `logLevel( dtlb, 2, $format("DTLB[%2d]:
  highest_ppn:%h",hartid,highest_ppn))

  // check for permission faults
`ifndef sv32
  if(unused_va != signExtend(lookup.va[`maxvaddr-1]))begin
    page_fault = True;
  end
`endif
  // pte.a == 0 || pte.d == 0 and access != Load
  if(!permissions.a || (!permissions.d && lookup.access
  != 0))
  begin
    page_fault = True;
  end
  if(lookup.access == 0 && !permissions.r &&
  (!permissions.x || mxr == 0))
  begin// if not readable and not mxr executable
```

```
      page_fault = True;
    end
    if(priv == 1 && permissions.u && sum == 0)
    begin // supervisor accessing user
      page_fault = True;
    end
    if(!permissions.u && priv == 0)begin
      page_fault = True;
    end
    // for Store access
    if(lookup.access != 0 && !permissions.w)begin
    // if not readable and not mxr  executable
      page_fault = True;
    end
    if(lookup.tlbmiss)begin
      rg_miss_queue <= lookup.va;
      ff_request_to_ptw.enq(PTWalk_tlb_request{address :
      lookup.va, access : lookup.access });
      ff_core_response.enq(DTLB_core_response{address
      : ?, trap  : False, cause : ?, tlbmiss  : True});
    end
    else begin
      `logLevel( dtlb, 0, $format("DTLB[%2d]: Sending PA:%h
      Trap:%b",hartid, physicaladdress, page_fault))
      `logLevel( dtlb, 0, $format("DTLB[%2d]: Hit in
      TLB:",hartid,fshow(pte)))
      ff_core_response.enq(DTLB_core_response{address
:

      truncate(physicaladdress), trap : page_fault,
      cause     : cause,    tlbmiss  : False});
    end
  end
endrule
```

```
interface put_core_request = interface Put
  method Action put (DTLB_core_request#(`vaddr) req)
  if(!rg_sfence);

    `logLevel( dtlb, 0, $format("DTLB[%2d]: received req: "
    ,hartid,fshow(req)))

    //4KB offset(12 bits)
    Bit#(12) page_offset1 = lookup.va[11 : 0];
    // tag is obtained by truncating the address after
    removing the page offset bits
    Bit#(`tlbindex) tag1 = truncate(lookup.va >> 12);
    // The other bits are taken to compare the addresses
    present in the block
    Bit#(TAdd#(12, `tlbindex)) vpncomp1 =
    truncateLSB(lookup.va >> 12);
    //2MB offset(21 bits)
    Bit#(21) page_offset2 = lookup.va[20 : 0];
    // tag is obtained by truncating the address after
    removing the page offset bits
    Bit#(`tlbindex) tag = truncate(lookup.va >> 21);
    // The other bits are taken to compare the addresses
    present in the block
    Bit#(TAdd#(21, `tlbindex)) vpncomp2 =
    truncateLSB(lookup.va >> 21);
    //Bit#(`vpnsize) fullvpn = truncate(lookup.va >> 12);

    /*doc:func: */
    function Bool fn_vtag_match (VPNTag t);
      return t.permissions.v && (({'1,t.pagemask} & fullvpn)
     == t.vpn) && (t.asid == satp_asid || t.permissions.g);
    endfunction
```

```
Bit#(`vaddr) va = req.address;
Bit#(`causesize) cause = req.cause;
Bool trap = req.ptwalk_trap;
Bool translation_done = False;
//checking in the first tlb
for(i <= 1; i <= `setnum1; i <= i + 1)begin
let hit_entry1 = find(vpncomp1,
readVReg(v_set_p1[i][tag1]));
if(hit_entry1) break
end
//checking in the second tlb
if(!hit_entry1) begin
for(i <= 1; i <= `setnum2; i <= i + 1)begin
let hit_entry2 = find(vpncomp2,
readVReg(v_set_p2[i][tag2]));
end
end
Bool tlbmiss = !isValid(hit_entry1 || hit_entry2);
VPNTag pte = fromMaybe(?,hit_entry);
Bit#(TSub#(`vaddr, `paddr)) upper_bits =
truncateLSB(req.address);
Bit#(2) priv = mprv == 0?wr_priv : mpp;
translation_done = (satp_mode == 0 || priv == 3 ||
req.ptwalk_req || req.ptwalk_trap);
if(!trap && translation_done)begin
   trap = |upper_bits == 1;
   cause = req.access == 0? `Load_access_fault:
   `Store_access_fault;
end

if(req.sfence && !req.ptwalk_req)begin
  rg_sfence <= True;
```

```
    end
    else begin
      ff_lookup_result.enq(LookUpResult{va: va, trap: trap,
      cause: cause, translation_done: translation_done,
      tlbmiss: tlbmiss, pte: pte, access: req.access});
    end

    if(req.sfence)
      rg_tlb_miss <= False;
    else if(rg_tlb_miss && req.ptwalk_trap)
      rg_tlb_miss <= False;
    else if(!translation_done && !req.ptwalk_req) begin
      rg_tlb_miss <= tlbmiss;
    `ifdef perfmonitors
      wr_count_misses <= pack(tlbmiss);
    `endif
    end

  endmethod
endinterface

  module mkreplace#(parameter Bit#(2) alg)(Ifc_replace#(sets,way
provisos(Add#(a__, TLog#(ways), 4));

//defining vectors needed for round robin
Vector#( `waynum4KB, Int#(`tlbindex)) waynum4KB <-
replicateM(mkReg(0));
Vector#( `setnum2, Int#(`tlbindex)) setnum2 <-
replicateM(mkReg(0));

//valueof, that takes a size type and gives the
corresponding Integer value
let v_ways = valueOf(ways);
let v_sets = valueOf(sets);
//compile time assertion by staticAssert
staticAssert(alg==0 || alg==1 || alg==2,"Invalid replacement
Algorithm");
//Random algorithm
if(alg == 0)begin
  LFSR#(Bit#(8)) random <- mkLFSR_8();
  Reg#(Bool) rg_init <- mkReg(True);
  //the rule to initiate LFSR
  rule initialize_lfsr(rg_init);
    random.seed(1);
    rg_init<=False;
  endrule

// The idea is to use the lower bits for 4KB dtlb and upper
bits for 2MB dtlb
method ActionValue#(Bit#(TLog#(ways))) line_replace
(Bit#(TLog#(sets))
        index, Bit#(ways) valid);
    if (&(valid)==1 && &(dirty)==1)begin
            replace = random.value();
            if(clog2(tag+1)==12) begin
             //extracting bits needed for 4KB from the random
             number generated
                    waynum4KB = replace[#Add(`waynum4KB, -1):0];
                    v_set_p1[waynum4KB][rg_replace1] <= tag;

            end
            if(clog2(tag+1)==12) begin
                    waynum2MB =
                    replace[#Add(`waynum2MB,#Add(`waynum4KB,
                    -1)):`waynum4KB];
                    v_set_p1[waynum2KB][rg_replace1] <= tag;
            end
            return truncate(rg_replace);
    end
    // if any line is not valid
    else if(&(valid)!=1) begin
      Bit#(TLog#(ways)) temp=0;
      for(Bit#(TAdd#(1,TLog#(ways)))
      i=0;i<fromInteger(v_ways);i=i+1) begin
        if(valid[i]==0)begin
          temp=truncate(i);
        end
      end
     return temp;
    end
  endmethod
  //method to update the LFSR with the next value
  method Action update_set (Bit#(TLog#(sets)) index,
  Bit#(TLog#(ways)) way)if(!rg_init);
    random.next();
  endmethod
  method Action reset_repl;
    random.seed(1);
  endmethod
end
else if(alg== 1)begin // RRBIN
  Vector#(sets,Reg#(Bit#(TLog#(ways)))) v_count <-
  replicateM(mkReg(fromInteger(v_ways-1)));
method ActionValue#(Bit#(TLog#(ways))) line_replace
(Bit#(TLog#(sets))
        index, Bit#(ways) valid);

  /* Bool alldirty = (&valid == 1 && &dirty == 1);
  Bool somedirty = (&valid == 1 && &dirty != 1);
  Bool nonedirty = (&valid == 1 && |dirty == 0);  */

    if(clog2(tag+1)==12) begin
            // the replacement is done based on the size and
            tag which is given to rg_replace1
            v_set_p1[waynum4KB][rg_replace1] <= tag;
    end
    if(clog2(tag+1)==21) begin
            // the replacement is done based on the size and
            tag
            which is given to rg_replace2
            v_set_p2[setnum2][rg_replace2] <= tag;
            // rg_replace1[setnum2] <= rg_replace1[setnum2]
            + 1;
            //rg_replace2 <= rg_replace2 + 1;
    end

endmethod
method Action update_set (Bit#(TLog#(sets)) index,
Bit#(TLog#(ways)) way);
    if(clog2(tag+1)==12) begin
            // Based on the tag, if the block in one set is
            replaced now, the following block with the same
            tag is replaced next
            waynum4KB[rg_replace1] <= waynum4KB[rg_replace1]
            + 1;
    end
    if(clog2(tag+1)==21) begin
            // Based on the tag, if the block in one set is
            replaced now, the following block with the same
            tag is replaced next
            waynum4KB[rg_replace2] <= waynum4KB[rg_replace2]
```

```
                + 1;
        end
      endmethod
      method Action reset_repl;
        for(Integer i=0;i<rg_replace1;i=i+1)
          waynum4KB[rg_replace1]<=fromInteger(v_ways-1);
        for(Integer i=0;i<rg_replace2;i=i+1)
          waynum4KB[rg_replace2]<=fromInteger(v_ways-1);
      endmethod
    end
endmodule
```

CHAPTER 6

REFERENCES

1. Caches MMU repository: https://gitlab.com/shaktiproject/uncore/caches_mmu
2. Shakti C-class: https://c-class.readthedocs.io/en/latest/index.html
3. Architectural and Operating System Support for Virtual Memory by By Abhishek Bhattacharjee, Daniel Lustig · 2017
4. Set Associative Mapping: https://www.sciencedirect.com/topics/computer-science/set-associative-cache