Advanced Computer Architecture
31-01-2020

A) A processor embeds two cores that have private L1 and L2 caches; the L3 cache is shared. The caches obey the MESI protocol and have the following structure: 64KB, 4-way L1 I-cache and D-cache, each 32-byte block; 2048 KB, 8-way associative, 32-byte block L2 cache; 8MB, 8-way associative, 64-byte block L3 cache. The latencies (disregarding virtual memory TLB) expressed in clock cycles are: 2 in L1, 4 in L2, 8 in L3. Addresses are 48-bit long. Store operations are carried out with a write-back allocate policy. Assuming initially empty and invalidated cache lines throughout the hierarchy, consider the following memory accesses:

core 1) LD F1, 0000FFFFFFA0hex;

core 2) ST F2, 0000FFFFFFB0hex;

core 1) ST F3, 0000FFFFFFA0hex;

core 2) ST F3, 0000FFFFFFB0hex;

a1) show the blocks involved in each cache, and the associated MESI state after each instruction.

a2) considering the situation generated by the instructions above, show a set of instructions that cause a block replacement in L3.

a3) the processor is attached to a memory subsystem through a 128-bit bus, and memory is built using DDR4 chip modules described in the attached table; choose system configuration (bus clock, memory chips) to minimize the cost of a cache miss by taking into account the following constraints: i) all modules support a memory word of 8 bytes; ii) DDR4 modules up to 1866 included can be configured in bank mode with a maximum interleaving of 2, while other modules do not allow for interleaving; iii) memory addressing requires 2 bus clock cycles; iv) memory activation requires 4 bus clock cycles if interleaving is active, otherwise 1 bus clock cycle. The processor bus interface can sustain a maximum transfer rate of 16GB/s.

B) A 512x512 image of 3-channel pixels is stored as an array V[] containg in each element a struct POINT representing the 3-channel pixel composed by double X,Y,Z (see the following fragment of C-Language code).

#include <stdio.h>

#define LEN = 262144

struct point {

 double x; // 8 bytes

 double y; // 8 bytes

 double z; // 8 bytes

};

void main(int argc, int \*\*argv)

{

 int i = 0;

 // definition

 struct point v[LEN];

 // phase A: inizialization

 for (i = 0; i < LEN; i++)

 {

 v[i].x = rand();

 v[i].y = rand();

 v[i].z = rand();

 }

 // phase B: computation

 for (i = LEN-1; i >=0; i—-)

 v[i].x = (v[i].x + v[i].y + v[i].z) / 3.0;

 }

}

Assuming that V[] is allocated in memory at increasing addresses in big-endian mode starting with V[0].x at address 0000FFFFA000hex ,

B1) compute the number of misses in L3 only due to phase A
B2) compute the number of misses in L3 only due to phase B.

C) Each core of the processor described in A) is organized as a superscalar, 2-way pipeline, that fetches, decodes issues and retires (commits) bundles containing each 2 instructions. The front-end in-order section (fetch and decode) consists of 2 stages. The issue logic takes 1 clock cycle, if the instructions in the bundle are independent, otherwise it takes 2 clock cycles. The architecture supports dynamic speculative execution, and control dependencies from branches are solved when the branch evaluates the condition, even if it is not at commit. The execution model obeys the attached state transition diagram. There is a functional units (FUs) Int1 for integer arithmetic (arithmetic and local instructions, branches and jumps, no multiplication), one FU FAdd1 for floating point addition/subtraction, one FUs FMolt1 for floating point multiplication, and one FU for division, FDiv1.

There are 12 integer (R0-R11) and 12 floating point (F0-F11) registers. Speculation is handled through a 8-entry ROB, a pool of 4 Reservation Stations (RS) Rs1-4 shared among all FUs, 2 load buffers Load1-Load2, 1 store buffer Store1 (see the attached execution model): an instruction bundle is first placed in the ROB (if two entries are available), then up to 2 instructions are dispatched to the shared RS (if available) when they are ready for execution and then executed in the proper FU. FUs are *pipelined* and have the latencies quoted in the following table:

|  |  |
| --- | --- |
| Int - 2 | Fadd – 3 |
| Fmolt – 4 | Fdiv – 6 |

Further assumption

* The code is assumed to be already in the I-cache; data caches are described in point A) and are assumed empty and invalidated; the cost of a miss is conventionally set to 50.

c1) assuming write-back, allocate protocol for cache management, show state transitions for the instructions of the loop up to the moment when PC04 in the second iteration is issued:

PC01 ADDI R1,R0, 0000FFFFFFA0hex

PC02 ADDI R8,R0,1024

PC03 LD F1,0(R1)

PC04 LD F2,8(R1)

PC05 MULTF F3,F2,F1

PC06 ST F2,0(R1)

PC07 ADDF F4,F3,F1

PC08 ST F4,8(R1)

PC09 ADDI R1,R1,16

PC10 SUBI R8,R8,1

PC11 BNEZ R8,PC03

c2) show ROB, RS and buffer status at the issue of PC04 in the second iteration;

c3) assuming the code is executed in a standard, “Von Neumann” processor, no pipelining, memory accesses through the cache hierarchy of point A with L1 cache only (no L2, no L3), what would be the PCI of the code ?

c4) would it be better to introduce a L2 cache as in point A, or to reduce L1 latency from 2 to 1 clock cycle?

Give a properly argumented, quantitative answer.

Dynamic speculative execution

Decoupled ROB RS execution model

|  |  |  |  |
| --- | --- | --- | --- |
| **ISTRUCTION** |  |  | **INSTRUCTION STATE**  |
|  | **n.ite** | **ROBpos** | **WO** | **RE** | **DI** | **EX** | **WB** | **RR** | **CO** |
| PC01 ADDI R1,R0, 0000FFFFFFA0hex |  |  |  |  |  |  |  |  |  |
| PC02 ADDI R8,R0,1024  |  |  |  |  |  |  |  |  |  |
| PC03 LD F1,0(R1) |  |  |  |  |  |  |  |  |  |
| PC04 LD F2,8(R1)  |  |  |  |  |  |  |  |  |  |
| PC05 MULTF F3,F2,F1 |  |  |  |  |  |  |  |  |  |
| PC06 ST F2,0(R1) |  |  |  |  |  |  |  |  |  |
| PC07 ADDF F4,F3,F1 |  |  |  |  |  |  |  |  |  |
| PC08 ST F4,8(R1) |  |  |  |  |  |  |  |  |  |
| PC09 ADDI R1,R1,16 |  |  |  |  |  |  |  |  |  |
| PC10 SUBI R8,R8,1 |  |  |  |  |  |  |  |  |  |
| PC11 BNEZ R8,PC03  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

|  |  |
| --- | --- |
|  | Reservation station and load/store buffers |
| Busy | Op | Vj | Vk | ROBj | ROBk | ROB pos | Address |
| Rs1 |  |  |  |  |  |  |  |  |
| Rs2 |  |  |  |  |  |  |  |  |
| Rs3 |  |  |  |  |  |  |  |  |
| Rs4 |  |  |  |  |  |  |  |  |
| Load1 |  |  |  |  |  |  |  |  |
| Load2 |  |  |  |  |  |  |  |  |
| Store1 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |

ROBj ROBk: sources not yet available

ROB pos: ROB entry number where instruction is located

|  |  |
| --- | --- |
|  | Result Register status |
| Integer | R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 | R8 | R9 | R10 | R11 |  |  |
| ROB pos |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| state |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| Float. | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 |  |  |
| ROB pos |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| state |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |
| --- |
| Reorder Buffer (ROB) |
|  ROB Entry#  | Busy | Op | Status | Destination | Value |
| 0 |  |  |  |  |  |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |
| 5 |  |  |  |  |  |
| 6 |  |  |  |  |  |
| 7 |  |  |  |  |  |

DI

EX

WB

RR

RE

CO

wo

QUEUE

**Decoupled execution model for bundled (paired) instructions**

The state diagram depicts the model for a dynamically scheduled, speculative execution microarchitecture equipped with a Reorder Buffer (ROB) and a set of Reservation Stations (RS). The ROB and RSs are allocated during the ISSUE phase, denoted as RAT (Register Alias Allocation Table) in INTEL microarchitectures, as follows: a bundle (2 instructions) if fetched from the QUEUE of decoded instructions and ISSUED if there is a free pair of consecutive entries in the ROB ( head and tail of the ROB queue do not match); a maximum of two instructions are moved into the RS (if available) when all of their operands are available. Access memory instructions are allocated in the ROB and then moved to a load/store buffer (if available) when operands (address and data, if proper) are available .

**States** are labelled as follows:

WO: Waiting for Operands (at least one of the operands is not available)

RE: Ready for Execution (all operands are available)

DI: Dispatched (posted to a free RS or load/store buffer)

EX: Execution (moved to a load/store buffer or to a matching and free UF)

WB: Write Back (result is ready and is returned to the Rob by using in exclusive mode the Common Data Bus CDB)

RR: Ready to Retire (result available or STORE has completed)

CO: Commit (result is copied to the final ISA register)

**State transitions** happen at the following events:

 *from* QUEUE *to* WO: ROB entry available, operand missing

*from* QUEUE *to* RE: ROB entry available, all operands available

*loop at* WO: waiting for operand(s)

*from* WO *to* RE: all operands available

*loop at* RE: waiting for a free RS or load/store buffer

*from* RE *to* DI: RS or load/store buffer available

*loop on* DI: waiting for a free UF

*from* DI *to* EX: UF available

*loop at* EX: multi-cycle execution in a UF, or waiting for CDB

*from* EX *to* WB: result written to the ROB with exclusive use of CDB

*from* EX *to* RR: STORE completed, branch evaluted

*loop at* RR: instruction completed, not at the head of the ROB, or bundled with a not RR instruction

*from* RR *to* CO: bundle of RR instructions at the head of the ROB, no exception raised

**Resources***Register-to-Register* instructions hold resources as follows:

ROB: from state WO (or RE) up to CO, inclusive;

RS: state DI

UF: EX and WB

*Load/Store* instructions hold resources as follows:

ROB: from state WO (or RE) up to CO, inclusive;

Load buffer: from state WO (or RE) up to WB

Store buffer: from state WO (or RE) up to EX (do not use WB)

**Forwarding**: a write on the CDB (WB) makes the operand available to the consumer in the same clock cycle. If the consumer is doing a state transition from QUEUE to WO or RE, that operand is made available; if the consumer is in WO, it goes to RE in the same clock cycle of WB for the producer.

**Branches**: they compute Next-PC and the branch condition in EX and optionally forward Next-PC to the “in-order” section of the pipeline (Fetch states) in the next clock cycle. They do not enter WB and go to RR instead.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Standardname** | **Memoryclock (MHz)** | **I/O busclock (MHz)** | **Datarate (**[**MT/s**](https://en.wikipedia.org/wiki/Transfer_%28computing%29)**)** | **Modulename** | **Peak transferrate (MB/s)** | **Timings,CL-tRCD-tRP** | **CAS latency(ns)** |
| DDR4-1600J\*DDR4-1600KDDR4-1600L | 200 | 800 | 1600 | PC4-12800 | 12800 | 10-10-1011-11-1112-12-12 | 12.513.7515 |
| DDR4-1866L\*DDR4-1866MDDR4-1866N | 233.33 | 933.33 | 1866.67 | PC4-14900 | 14933.33 | 12-12-1213-13-1314-14-14 | 12.85713.92915 |
| DDR4-2133N\*DDR4-2133PDDR4-2133R | 266.67 | 1066.67 | 2133.33 | PC4-17000 | 17066.67 | 14-14-1415-15-1516-16-16 | 13.12514.06315 |
| DDR4-2400P\*DDR4-2400RDDR4-2400U | 300 | 1200 | 2400 | PC4-19200 | 19200 | 15-15-1516-16-1618-18-18 | 12.513.3315 |