Advanced Computer Architecture
21-02-2018

A) A processor running at 2.3 GHz 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: 32KB, 4-way L1 I-cache and D-cache, each 32-byte block; 1024KB, 4-way associative, 64-byte block L2 cache; 8MB, 8-way associative, 128-byte block L3 cache. The latencies (disregarding virtual memory TLB) expressed in clock cycles are: 2 in L1, 6 in L2, 12 in L3. Addresses are 48-bit long. Write operations are managed with a write-back, allocate policy. Assuming initially empty and invalidated cache lines throughout the hierarchy, consider the following memory accesses:

core 2) LD F2, 0000FFFFA00Ahex; (8-byte load)

core 2) ST F3, 0000FFFFA010hex; (8-byte store)

core 1) ST F1, 0000FFFFA0A0hex; (8-byte store)

core 1) ST F1, 0000FFFFA000hex; (8-byte store)

a1) show the blocks involved in each cache, and the associated MESI state after each instruction;
a2) show a sequence of operations that causes a block replacement in the L2 cache in core 1 (assuming empty, invalidated caches, and no intervening cache access in core 2).

B) An array v[] contains in each element a struct POINT composed by a float X and a float Y (see the following fragment of C-Language code).

#include <stdio.h>

#define LEN 1024

struct point {

 float x; // 8 bytes

 float y; // 8 bytes

};

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

{

 int i = 0;

 // definition

 struct point v[LEN];

 // inizialization

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

 {

 v[i].x = rand();

 v[i].y = rand();

 }

};

The array is shared between two threads, T1 running in core1, T2 running in core2. The array has been properly allocated and initialized, with v[o].x stored at address 0000FFFFA0A0hex; after this initialization phase all caches have been invalidated. At some point T1 and T2 start executing the following code

 // T1 works on even numbered elements

 for (i = 0; i <=1022; i= i+2)

 v[i].x = v[i].x \* v[i].y;

 v[i].y = (v[i].x + v[i].y) / 2;

 }

 // T2 works on odd numbered elements

 for (i = 1; i <=1023; i= i+2)

 v[i].x = v[i].x \* v[i].y;

 v[i].y = (v[i].x + v[i].y) / 2;

 }

B1) Estimate the number of hit and miss in each cache;
B2) Assuming that float multiplication take 4 clock cycles to execute, what speed-up would you expect by making the floating point unit complete execution in 2 clock cycles?

B3) What would be the speed-up if the cores were driven by a 3.0 GHz clock, while keeping unaltered floating point unit latencies (float multiplication 4 clock cycles, float addition 3 clock cycles) ?
Warning: make any reasonable assumption to answer B2 and B3.

B4) Is it possible to unroll the loop executed by each thread? Discuss in detail.

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 unit (FUs) Int1 for integer arithmetics (arithmetic and local instructions, branches and jumps, no multiplication), 2 FUs FAdd1-Fadd2 for floating point addition/subtraction and shift, 1 FU FMolt1 for floating point multiplication, and a FU for division, FDiv1.

There are 12 integer (R0-R11) and 12 floating point (F0-F11) registers. Speculation is handled through a 10-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* (except for the float division unit, which is blocking) and have the latencies quoted in the following table:

|  |  |
| --- | --- |
| Int - 2 | Fadd – 3 |
| Fmolt – 4 | Fdiv – 8 |

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 30.

c1) assuming a write-back protocol for cache management, show state transitions for all instructions in the first iteration and instructions up to PC05 included in the second iteration :

PC01 ADDI R1,R0, 0000FFFFA0A0hex

PC02 ADDI R3,R0, OFF

PC03 LD F1,0(R1)

PC04 LD F2,8(R1)

PC05 MULTF F2,F1,F2

PC06 ADDI R1,R1,16

PC07 FRSH F2,F2,2 -- float right shift by 2 (executed in a Fadd FU)

PC08 LD F3,0(R1)

PC09 LD F4,8(R1)

PC10 ADDF F3,F3,F4

PC11 FRSH F3,F3,2 -- float right shift by 2 (executed in a Fadd FU)

PC12 ST F2,-16(R1)

PC13 ST F3,0(R1)

PC14 BLE R1,R3,PC03 - Branch on R1<=R3

c2) chose the constant OFF in such a way that this loop scans through all 1024 elements of the
“array of struct” v[];

c3) show ROB, RS and buffer status at the issue of PC05 in the second iteration;

c4) discuss the effect (on this piece of code) of a BTB with 128 entries in the IF stage of the in-order pipeline.

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, 0000FFFFA0A0hex |  |  |  |  |  |  |  |  |  |
| PC02 ADDI R3,R0, OFF  |  |  |  |  |  |  |  |  |  |
| PC03 LD F1,0(R1)  |  |  |  |  |  |  |  |  |  |
| PC04 LD F2,8(R1) |  |  |  |  |  |  |  |  |  |
| PC05 MULTF F2,F1,F2 |  |  |  |  |  |  |  |  |  |
| PC06 ADDI R1,R1,16 |  |  |  |  |  |  |  |  |  |
| PC07 FRSH F2,F2,2  |  |  |  |  |  |  |  |  |  |
| PC08 LD F3,0(R1) |  |  |  |  |  |  |  |  |  |
| PC09 LD F4,8(R1) |  |  |  |  |  |  |  |  |  |
| PC10 ADDF F3,F3,F4 |  |  |  |  |  |  |  |  |  |
| PC11 FRSH F3,F3,2  |  |  |  |  |  |  |  |  |  |
| PC12 ST F2,-16(R1) |  |  |  |  |  |  |  |  |  |
| PC13 ST F3,0(R1) |  |  |  |  |  |  |  |  |  |
| PC14 BLE R1,R3,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 |  |  |  |  |  |
| 8 |  |  |  |  |  |
| 9 |  |  |  |  |  |

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 couple 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.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** | **15** | **16** | **17** | **18** | **19** | **20** | **21** | **22** | **23** | **24** | **25** | **26** | **27** |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |