Advanced Computer Architecture  
24-01-2017

A) A processor embeds two cores that have private L1 caches; the L2 and L3 caches are shared. The caches obey the MESI protocol and have the following structure: 32KB, 2-way L1 I-cache and D-cache, each 16-byte block; 512KB, 4-way associative, 32-byte block L2 cache; 4MB, 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. Assuming initially empty and invalidated cache lines throughout the hierarchy, consider the following memory accesses:

core 1) LD F1, 0000FFFFA008hex; (64-bit load into F1)

core 2) LD R2, 0000FFFFA000hex; (32-bit load into R2)

core 1) ST F1, 0000FFFFA000hex; (64-bit store)

core 2) ST R3, 0000FFFFA010hex; (32-bit store)

a1) show the blocks involved in each cache, and the associated MESI state after each instruction, under a write-back policy.

a2) show a set of instructions that causes a block replacement in the L2 cache.

B) Let us consider the following code fragment of compiler-unscheduled instructions:

ADDI R1,R0,0000FFFFF000hex -- R1 set to base address 0000FFFFF000hex

loop: LD F1,0(R1)

MULTF F2,F1,F0 -- F0 contains a pre-loaded float constant

MULTF F2,F2,F2

ST F2,0(R1)

LD F3,8(R1)

ADDF F5,F3,F1

ST F5,8(R1)

ADD R1,R1,16

BLI R1,000100000000hex,loop -- compare immediate, branch on less

assuming this code fragment is executed in a statically scheduled pipeline with stages

IF|ID|INT |MEM|WB

|A1 |A2|

|M1 |M2|M3|M4|M5

and proper forwarding units, assuming that branch instructions take a decision in stage MEM, assuming that all LD hit in L1 D-cache with a latency of 1 clock cycles, and that ST hits with a latency of 2 clock cycles

b1) show a compiler schedule that minimizes the clock cycles required to complete execution;

b2) compute the CPI of the execution of this kernel;

b3) estimate the speed-up that can be obtained by reducing the latency of the FLOAT multiply unit from 5 to 3 cloack cycles.

C) Each core of the processor described in A) is organized as a superscalar, 3-way pipeline, that fetches, decodes issues and retires (commits) bundles containing each 3 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 arithmetics (arithmetic and local instructions, branches and jumps, no multiplication), 2 FUs FAdd1-Fadd2 for floating point addition/subtraction, two FUs FMolt1-Fmult2 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 8-entry ROB, a pool of 4 Reservation Stations (RS) Rs1-4 shared among all FUs, 1 load buffers Load1, 1 store buffer Store1 (see the attached execution model): an instruction bundle is first placed in the ROB (if three 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 60.

c1) assuming a write-back protocol for cache management, show state transitions for the instructions of the first two iterations of the loop in point B) :

PC01 ADDI R1,R0,0000FFFFF000hex

PC02 LD F1,0(R1)

PC03 MULTF F2,F1,F0

PC04 MULTF F2,F2,F2

PC05 ST F2,0(R1)

PC06 LD F3,8(R1)

PC07 ADDF F5,F3,F1

PC08 ST F5,8(R1)

PC09 ADD R1,R1,16

PC10 BLI R1,000100000000hex,PC02

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

c3) assume the L3 cache is doubled in size either i) by doubling its block size to 128B , or ii) by doubling its index set while retaining a block-size of 64B. Estabilish which is best for the loop above.

Dynamic speculative execution

Decoupled ROB RS execution model

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **ISTRUCTION** |  |  | **INSTRUCTION STATE** | | | | | | |
|  | **n. ite** | **ROB pos** | **WO** | **RE** | **DI** | **EX** | **WB** | **RR** | **CO** |
| PC01 ADDI R1,R0,0000FFFFF000hex |  |  |  |  |  |  |  |  |  |
| PC02 LD F1,0(R1) |  |  |  |  |  |  |  |  |  |
| PC03 MULTF F2,F1,F0 |  |  |  |  |  |  |  |  |  |
| PC04 MULTF F2,F2,F2 |  |  |  |  |  |  |  |  |  |
| PC05 ST F2,0(R1) |  |  |  |  |  |  |  |  |  |
| PC06 LD F3,8(R1) |  |  |  |  |  |  |  |  |  |
| PC07 ADDF F5,F3,F1 |  |  |  |  |  |  |  |  |  |
| PC08 ST F5,8(R1) |  |  |  |  |  |  |  |  |  |
| PC09 ADD R1,R1,16 |  |  |  |  |  |  |  |  |  |
| PC10 BLI R1,000100000000hex,PC02 |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Reservation station and load/store buffers | | | | | | | |
| Busy | Op | Vj | Vk | ROBj | ROBk | ROB pos | Address |
| Rs1 |  |  |  |  |  |  |  |  |
| Rs2 |  |  |  |  |  |  |  |  |
| Rs3 |  |  |  |  |  |  |  |  |
| Rs4 |  |  |  |  |  |  |  |  |
| Load1 |  |  |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  |  |
| 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 (3 instructions) if fetched from the QUEUE of decoded instructions and ISSUED if there is a free triple 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.