Advanced Computer Architecture
25-01-2016

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

A LD F1,-16(R1)

B ADDF F2,F1,F0

C ST F2,0(R1)

D ADD R1,R3,2

a1) list all dependencies;

B on A: raw (F1)

C on B: raw (F2)

D on A: war (R1)

D on C: war (R1)

a2) assuming this code fragment is executed in a statically scheduled pipeline with stages

IF|ID|INT |MEM|WB

 |A1 |A2|A3

 |M1 |M2|M3|M4|M5

and proper forwarding units, show a compiler schedule that minimizes the clock cycles required to complete execution (disregard caches and assume that MEM always takes 1 clock cycle).

This code fragment CANNOT be scheduled if the compiler has all registers already allocated. If some register is still un-allocated, it might be used to pre-compute the 0(R1) address for the ST F2,0(R1), provided this register is newly assigned after the ST, WITHOUT being read first. Assuming R2 is in such a condition:

unoptimized schedule optimzed schedule

LD F1,-16(R1) LD F1,-16(R1)

 nop ADD R2,R1,R0 (only if R2 is assigned after the ST without being read)

 ADDF F2,F1,F0 ADDF F2,F1,F0

 nop ADD R1,R3,2

 nop nop

 ST F2,0(R1) ST F2,0(R2)

 ADD R1,R3,2

B) A server with 128GB RAM is built using 16 processors that embed each four cores that have private L1 caches, while two L2 caches are shared by each couple of cores; the single L3 cache is shared by the four cores. The caches obey the MESI protocol and have the following structure: 16KB, 4-way L1 I-cache and D-cache, each 32-byte block; 512KB, 4-way associative, 64-byte block L2 cache; 8MB, 8-way associative, 128-byte block shared L3 cache. The latencies (disregarding virtual memory TLB) expressed in clock cycles are: 4 in L1, 8 in L2, 24 in L3. Addresses are 48-bit long.

A Ram controller manages the access to a shared bus from each processor. The MESI protocol is enforced throughout the memory hierarchy.

b1) assuming initially empty and invalidated cache lines throughout the hierarchy, consider the following memory accesses

core 1) LD F1, 0000FFFFA000hex;

core 2) LD F2, 0000FFFFA010hex;

core 3) ST F3, 0000FFFFA010hex

core 2) LD F2, 0000FFFFA000hex;

core 4) ST F4, 0000FFFFA000hex;

show the cache blocks involved throughout the memory hierarchy, the sequence of miss/hit in the hierarchy (assume writes are managed with allocate - write-back policy), and the blocks MESI state.

All accesses are properly aligned.

|  |  |  |
| --- | --- | --- |
|  | cache accesses | MESI status of involved blocks (b) |
|  | L1(1) | L2(1-2) | L3(1-4) | b L1(1) | b L1(2) | b L2(1-2) | b L1(3) | b L1(4) | b L2(3-4) | B L3(1-4) |
| core 1) LD F1, 0000FFFFA000hex | m | m | m | E |  | E |  |  |  | E |
| core 2) LD F2, 0000FFFFA010hex | m | h |  | S | S | S |  |  |  | S |
| core 3) ST F3, 0000FFFFA010hex | m | m | h | I | I | I | M |  | I | I |
| core 2) LD F2, 0000FFFFA000hex | m | m | m | S | S | S | S |  | S | S |
| core 4) ST F4, 0000FFFFA000hex | m | m | h |  | I | I |  | M | I | I |
|  |  |  |  |  |  |  |  |  |  |  |

b2) considering a shared array of 64-bit floating point elements X allocated in memory starting at address 0000FFFFA000hex and the following code fragment run on core 1)

*for (i=0,imax,i++) {X(i)=0.0};*

determine the value of *imax* that causes the first block replacement in the L1-D cache of core 1).

We disregard initially the cache blocks involved by reading variables *i*, *imax and the floting point constant 0.0* (they should be taken into account, nethertheless !!). The loop consists of a ST F0,0(R1) type instruction, where R1 is loaded with the address 0000FFFFA000hex that is aligned with the first element of the array and ALSO aligned with L3,L2 and L1 blocks.

000000000000000011111111111111111010|0000000|00000

R1 is increased by 8 at each iteration. Each array element takes 8 bytes, and each L1 D-cache block contains 4 elements. Successive array elements map into consecutive cache blocks that map themselves into consecutive ways and then into consecutive sets. So, the first set is filled up with the first 4x4=16 array elements. The first replacement occurs all 128 sets have been filled up, that is when 128x16=2048 elements have been stored. Stated otherwise, the replament happens when element *imax* has the same cache index of the first element of the array and a different tag, but the four ways of that set are occupied. This happens when the bit of position 12 changes from 0 to 1. So *imax=2048*.

C) The server RAM is built using 2x8GB DDR3 1600 chips attached to the 64-bit external bus of the processors running at 2.6GHz. The external bus allows a maximum transfer rate of 8 GB/sec.

c1) how many chips are necessary for the whole RAM?

Assuming a naive chip layout, 128GB/16GB=8

c2) is the memory subsystem (bus+memory chips) well balanced?

NO, because the DDR3-1600 chips allow for a 800MGz clock, while the external bus transfer rate of 8 GB/sec on a 8-byte bus complies with 8G/8=1G transfers per second that are sustained by a 500MHz clock (double date rate transfer). The bus is the limiting sub-unit.

c3) the DDR chips allow reading out a 16-byte memory word in a single activation; assuming that addressing takes 2 clock cycles and that memory activation takes 2 clocks if the interleaving factor is 2, 3 clocks for an interleaving of 3, and 5 for an interleaving of 4, choose the memory organization that minimizes a cache miss, and compute the cost of the miss in processor clock cycles.

Taddr is 2, independently of the memory logical organization; Ttransfer also is 128 (L3 block) over 8 (bus width) over 2 (double date rate)=8; the logical organization changes the number of activations as follows:

Natt=sup [L3 block/(memory word\* interleaving factor)]

so we have

Tatt x Natt=sup [L3 block/(memory word\* interleaving factor)]

2 x sup[128/(16\*2)]=8

3 x sup[128/(16\*3)]=9

5 x sup[128/(16\*4)]=10

So the minimum is for and interleaving factor 2, and the total cost of the miss is

Taddr + Natt x Tatt + Ttrasf = 2 + 8 + 8= 18 (bus clock cycles)

CPU clock/bus clock=2.66 GHz/0.5 Ghz= 5,32

Total cost in CPU clock cycles = sup[5,32 x 18]=96

c4) assuming that *imax* in code fragment is 1024, determine the number of hit and of misses in each

cache hierarchy if the code is run in isolation in core 1), with initially empty and invalidated D-caches;

The pattern of cache accesses has a period of 16, because the array is properly aligned and stored in consecutive memory areas. 16 iterations use a whole L3 block (16 x 8 bytes), so there are 64 such cycles.

The accesses are as follows

L1 m h h h m h h h m h h h m h h h

L2 m h m h

L3 m h

This pattern is repeated 64 times

L1 miss= 4 x 64 L1 hit= 12 x 64

L2 miss= 2 x 64 L2 hit= 2 x 64

L3 miss= 1 X 64 L3 hit= 1 x 64

c5) compute the best possible estimate of the clock cycles required to complete the execution of the code fragment with imax=1024;

Let us compute the clock cycles for the ST instructions, by taking into account cache latencies Lat1,Lat2, Lat3 and miss cost M as estimated in point c3:

64((Lat1+Lat2+Lat3+M)+12xLat1+2x(Lat1+Lat2)+(Lat1+Lat2+Lat3))

 m m m M h m h m m h

that is

64((4+8+24+96)+12x4+2x(4+8)+(4+8+24))=64(132+48+24+36)= 64x240=15360

We can safely assume that the execution of the other instructions in loop is totallo hidden by the memory hierarchy latencies, so that this is a sound estimated of the loop elapsed time in cpu clock.

D) Each core of the processor 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 arithmetics (arithmetic and local instructions, branches and jumps, no multiplication, 2 FUS FAdd1-Fadd2 for floating point addition/subtraction, a 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 6-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 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 *blocking* and have the latencies quoted in the following table:

|  |  |
| --- | --- |
| Int - 2 | Fadd – 3 |
| Fmolt – 5 | 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.

d1) assuming a write-back protocol for cache management, show state transitions for the instructions of the first iteration of the following code fragment, that scans the 1024 floating elements, each 8-byte, of array X[] and replaces X[i] with X[i]+A (R0 is always 0).

PC01 OR R1,R0,00000FFFFA000hex -- set base address of X[0]

PC02 OR R5,R0,1023dec –- set loop terminating condition

PC03 OR R3,R0,R0 -- initialize loop controlling variable

PC04 LD F0,0(R1) -- load X[i]

PC05 ADDF F2,F1,F0 -- new value for X[i]=X[i]+A (A is stored in F1)

PC06 ST F2,0(R1) -- written back

PC07 ADD R3,R3,1 -- increase loop controlling variable

PC08 ADD R1,R1,8 -- advance pointer into array X

PC09 BL R5,R3,PC04 -- testing for loop exit condition

d2) show ROB, RS and buffer status at the issue of PC04 in the second iteration.

Dynamic speculative execution

Decoupled ROB RS execution model

|  |  |  |  |
| --- | --- | --- | --- |
| **ISTRUCTION** |  |  | **INSTRUCTION STATE**  |
|  | **n.ite** | **ROBpos** | **WO** | **RE** | **DI** | **EX** | **WB** | **RR** | **CO** |
| PC01 OR R1,R0,00000FFFFA000hex |  | **1** |  | **1** | **2** | **3-4** | **5** | **6-9** | **10** |
| PC02 OR R5,R0,1023dec  |  | **2** |  | **1** | **2-5** | **6-7** | **8** | **9** | **10** |
| PC03 OR R3,R0,R0  |  | **3** |  | **2** | **3-8** | **9-10** | **11** | **12-104** | **105** |
| PC04 LD F0,0(R1)  |  | **4** | **2-4** | **5** | **6** | **7-102** | **103** | **104** | **105** |
| PC05 ADDF F2,F1,F0  |  | **5** | **3-102** | **103** | **104** | **105-107** | **108** | **109-114** | **115** |
| PC06 ST F2,0(R1)  |  | **6** | **4-107** | **108** | **109** | **110-113** | **-** | **114** | **115** |
| PC07 ADD R3,R3,1  |  | **1** |  | **11** | **12** | **13-14** | **15** | **16-115** | **116** |
| PC08 ADD R1,R1,8  |  | **2** |  | **11** | **12-15** | **16-17** | **18** | **19-115** | **116** |
| PC09 BL R5,R3,PC04  |  | **3** |  | **106** | **107** | **108-109** | **-** | **110-116** | **117** |
| PC10 ISTRA  |  | **4** |  | **cancelled at 110** |
|  |  |  |  |  |  |  |  |  |  |

The Int FU is blocked up to WB included

PC04 in the second iteration receives the NextPC address in clock 110, takes 4 clock cycles for L1-Icache hit (first stage) plus a further clock for the second (decode) stage, so it is issues at time 116 (the ROB is full till clock 115 included, by the way)

|  |  |
| --- | --- |
|  | 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 |  | 2 |  | 1 |  |  |  |  |  |  |  |  |  |  |
| state |  | W |  | W |  |  |  |  |  |  |  |  |  |  |
| Float. | F0 | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 |  |  |
| ROB pos | 4 |  | 5 |  |  |  |  |  |  |  |  |  |  |  |
| state | B |  | B |  |  |  |  |  |  |  |  |  |  |  |

|  |
| --- |
| Reorder Buffer (ROB) |
|  ROB Entry#  | Busy | Op | Status | Destination | Value |
| 1 | yes | PC07 ADD R3,R3,1  | CO | R3 | [R3]+1 |
| 2 | yes | PC08 ADD R1,R1,8  | CO | R1 | [R1]+1 |
| 3 | yes | PC09 BL R5,R3,PC04  | RR |  | PC04 |
| 4 | yes | PC04 LD F0,0(R1)  | RE | F0 | Mem([R1]) |
| 5 | no |  |  |  |  |
| 6 | no |  |  |  |  |

clock 116

second iteration

|  |
| --- |
| Reorder Buffer (ROB) |
|  ROB Entry#  | Busy | Op | Status | Destination | Value |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 | yes | PC09 BL R5,R3,PC04  | CO |  | PC04 |
| 4 | yes | PC04 LD F0,0(R1)  | DI | F0 | Mem([R1]) |
| 5 | yes | PC05 ADDF F2,F1,F0  | WO | F2 | [ROB5]+[F1] |
| 6 | no |  |  |  |  |

clock 117 (PC05 enters 1 clock after PC04 because of the dependency)

second iteration

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 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 DI up to WB

Store buffer: from state DI 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.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Standard name** | **Memory clock****(MHz)** | **Cycle time****(ns)** | **I/O bus clock****(MHz)** | **Data rate****(**[**MT/s**](http://en.wikipedia.org/wiki/Transfer_%28computing%29)**)** | **Module name** | **Peak transfer rate****(MB/s)** |
| DDR3-800DDDR3-800E | 100 | 10 | 400 | 800 | PC3-6400 | 6400 |
| DDR3-1066EDDR3-1066FDDR3-1066G | 133⅓ | 7 1⁄2 | 533⅓ | 1066⅔ | PC3-8500 | 8533⅓ |
| DDR3-1333F\*DDR3-1333GDDR3-1333HDDR3-1333J\* | 166⅔ | 6 | 666⅔ | 1333⅓ | PC3-10600 | 10666⅔ |
| DDR3-1600G\*DDR3-1600HDDR3-1600JDDR3-1600K | 200 | 5 | 800 | 1600 | PC3-12800 | 12800 |
| DDR3-1866J\*DDR3-1866KDDR3-1866LDDR3-1866M\* | 233⅓ | 4 2⁄7 | 933⅓ | 1866⅔ | PC3-14900 | 14933⅓ |
| DDR3-2133K\*DDR3-2133LDDR3-2133MDDR3-2133N\* | 266⅔ | 3 3⁄4 | 1066⅔ | 2133⅓ | PC3-17000 | 17066⅔ |

DDR3 standard JEDEC specification (source Wikipedia)