Advanced Computer Architecture
24 February 2014

A) A processor has the following cache hierarchy: 32KB, 2-way L1 I-cache, 32 KB 4-way L1 D-cache, 32-byte block; 1024 KB, 4-way associative, 64-byte block L2 cache; 4MB, 8-way associative, 64-byte block L3 cache. The latencies (disregarding virtual memory TLB) expressed in clock cycles are: 4 in L1, 10 in L2, 35 in L3. Copying a block from level 3 to level 2 costs 4 clocks, copying from L2 to l1 costs 2 clocks. All cache are pipelined, and the L3/ExtBUS interface handles a maximum of 4 outstanding misses.

a1) assuming a write through policy at the L1/L2 and L2/l3 interfaces and a write back policy at the L3/ExtBUS interface, show a sequence of operations that cause a block replacement in L3.

a2) a C compiler translates the following call to the malloc() function

*FM = (double *) malloc(nz * sizeof(double))*

using 0x00FBFB00 as the first byte of array element &FM[0]:
Choose a value for *nz* such that the L1 D-cache undergoes capacity miss during the execution of the following code fragment:

```
for (i=0; i<nz; i++) {&FM[i]= (double) i; }
```

a3) compute the number of hits and misses in each level of the cache hierarcy if the code fragment of a2) is executed with nz= 1024 (disregard I-cache).

B) A main memory of 8 GB is attached to the processor described in A) (clocked at 2,33 GHz), through a 64-bit bus. The memory is setup with SDRAM modules:

PC3-8500 | Unbuffered | Nonparity | 204-pin | 1066 MHz | DDR3 SDRAM

It consists of two interleaved banks (DIMM/0 and DIMM/1), each offering 32-bit words. Estimate the cost of a miss, assuming that the addressing of the memory interface takes 2 clock cycles, and that the activation costs 4 clock cycles.

C) The processor has 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 are 2 functional units (FUs) Int1-INT2 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 8-entry ROB, a pool of 4 Reservation Stations (RS) Rs1-4 shared among all FUs, 2 load buffers Load1-2, 2 store buffers Store1-2 (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) and then executed in the proper FU. FUs are *pipelined* (not the Fdiv one) and have the latencies quoted in the following table:

Int – 2	Fadd – 3
Fmolt – 5	Fdiv – 6

Further assumption

- Data caches are described in point A) and are assumed empty and invalidated.

c1) show state transitions for the instructions of the first two iterations of the following code fragment (assume the code is already loaded in the I-cache and a conventional L3 miss time of 120 clock cycles), highlighting conflicts, if any:

```
PC01 MOVI  R1, 0x00FBFB00  – set base address of X[0]
PC02 MOVI  R5, 1023        – set loop terminating condition
PC03 MOVI  R2, 0           -- initialize loop controlling variable
PC04 LD    F4,0(R1)        -- load X[i]
PC05 LD    F5,16(R1)       -- load into X[i+2]
PC06 FADD  F6,F4,F5
PC07 ST    F6,0(R1)
PC08 ADD   R2,R2,1         -- increase loop controlling variable
PC09 ADD   R1,R1,16        -- advance pointer into array by 2
PC10 BNEQ  R5,R2,PC04      -- testing for loop exit condition
```

c2) show ROB and buffer status at the issue of the PC04 in the second iteration;
c3) discuss the possible unrolling of the loop.
c4) estimate the run time of the code, and the speed up that can be obtained by
   i) raising the processor clock cycle by 10%
   ii) lowering miss penalty by 1%

Dynamic speculative execution
Decoupled ROB RS execution model

ISTRUCTION			INSTRUCTION STATE						
	n. ite	ROB pos	WO	RE	DI	EX	WB	RR	CO
PC01 MOVI R1, 0x00FBFB00									
PC02 MOVI R5, 1023									
PC03 MOVI R2, 0									
PC04 LD F4,0(R1)									
PC05 LD F5,16(R1)									
PC06 FADD F6,F4,F5									
PC07 ST F6,0(R1)									
PC08 ADD R2,R2,1									
PC09 ADD R1,R1,16									
PC10 BNEQ R5,R2,PC04									

	Load/store buffers							
	Busy	Op	Vj	Vk	ROB_j	ROB_k	ROB pos	Address
Load1								
Load2								
Store1								
Store2								

ROB_j ROB_k: 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 Entry#	Busy	Op	Status	L_j	L_k	Destination	Value
1							
2							
3							
4							
5							
6							
7							
8							

L_j, L_j : operands (use the following notation: V/0 for a Value that is ready, ROB_k for a value that will be produced at ROB position K) see example

	yes	operation		[3]/0	ROB_2	F4	

This "operation" has one operand ready read from ROB entry 3, and is waiting for a second operand that will be produced by ROB entry 2; it will eventually place its result in register F4.

$$\begin{tikzcd}
\text{QUEUE} \arrow[r] \arrow[rr, bend right] & \text{wo} \arrow[loop above] \arrow[r] & \text{RE} \arrow[loop above] \arrow[r] & \text{DI} \arrow[loop above] \arrow[r] & \text{EX} \arrow[loop above] \arrow[r] \arrow[rr, bend left] & \text{WB} \arrow[r] & \text{RR} \arrow[loop above] \arrow[r] & \text{CO}
\end{tikzcd}$$

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 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); up to two instructions are moved into two RS (if available) when 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.