Advanced Computer Architecture
Semptember 2014 – first test

A) A processor embeds four cores that have private L1 and L2 caches, and a shared L3 cache. The caches obey the MESI protocol and have the following structure: 32KB, 2-way L1 I-cache, 32 KB 4-way L1 D-cache, each 32-byte block; 1024 KB, 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, 10 in L2, 25 in L3. Addresses are 48-bit long.

a1) assuming initially empty and invalidated cache lines throughout the hierarchy, consider the following memory accesses
core 1) LD F1, $00000FFFF0000_{hex}$;
core 2) LD F1, $00000FFFF0010_{hex}$;
core 3) ST F1, $00000FFFF0010_{hex}$
and show the cache blocks involved throughout the memory hierarchy.
a2) show the status of each cache block involved at the end of the sequence of memory operations, according to the MESI protocol.

B) The processor runs at 2.6GHz, and the 64-bit external bus allows a maximum transfer rate of 16 GB/sec.
The external RAM is realized with DDR3-1066 chips and is logically organized with 4 banks, each capable of delivering a 16-bit word. Addressing the memory subsystems requires two bus cycles, and activating a memory row requires 3 bus clock cycles. Assuming burst transfer mode from DDRAM, estimate the cost of a miss.

C) 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 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-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* (not the Fdiv one) 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 that computed at point B.

c1) assuming a write-back protocol for cache management, show state transitions for the instructions of the first iteration of the following code fragment, that transforms the 1024 floating elements, each 8-byte, of array X[], into 512 floating point elements in array Y[], highlighting conflicts, if any:

```
PC01 MOVI  R1,00000FFFF0000hex -- set base address of X[0]
PC02 ADDI  R2,R1,8192dec       -- set base address of Y[0]
PC03 MOVI  R5,1023             -- set loop terminating condition
PC04 MOVI  R3,0                -- initialize loop controlling variable
PC05 LD    F0,0(R1)            -- load X[i]
PC06 LD    F1,8(R1)            -- load X[i+1]
PC07 ADDF  F2,F0,F1            -- X[i]+X[i+1]
PC08 ST    F2,0(R2)            -- store into Y[i]
PC09 ADD   R3,R3,2             -- increase loop controlling variable
PC10 ADD   R1,R1,16            -- advance pointer into array X
PC11 ADD   R2,R2,8             -- advance pointer into array Y
PC12 BL    R5,R3,PC05          -- testing for loop exit condition
```

c2) show ROB, RS and buffer status at the issue of PC05 in the second iteration;

D) Show a c-like version of the assembly code fragment and answer the following questions:
d1) does the code allow for unrolling ? give a detailed explanation;
d2) imagine to modify the code for parallel execution (e.g. with openMP); which part of the code can be actually parallelized? what is the maximum speed-up can that be obtained in the processor ?

E) Estimate the CPI of the algorithm.

Dynamic speculative execution
Decoupled ROB RS execution model

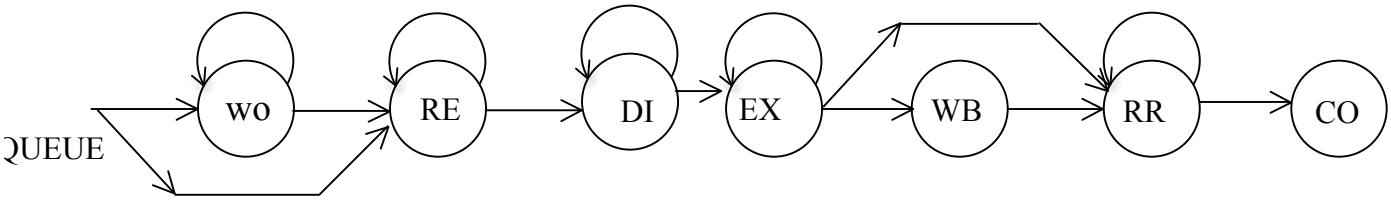
ISTRUCTION			INSTRUCTION STATE						
	n. ite	ROB pos	WO	RE	DI	EX	WB	RR	CO
PC01 MOVI R1,00000FFFF0000$_{hex}$									
PC02 ADDI R2,R1,8192$_{dec}$									
PC03 MOVI R5,1023									
PC04 MOVI R3,0									
PC05 LD F0,0(R1)									
PC06 LD F1,8(R1)									
PC07 ADDF F2,F0,F1									
PC08 ST F2,0(R2)									
PC09 ADD R3,R3,2									
PC10 ADD R1,R1,16									
PC11 ADD R2,R2,8									
PC12 BL R5,R3,PC05									

	Reservation station and load/store buffers							
	Busy	Op	Vj	Vk	ROB_j	ROB_k	ROB pos	Address
Rs1								
Rs2								
Rs3								
Rs4								
Load1								
Load2								
Store1								

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)					
ROB Entry#	Busy	Op	Status	Destination	Value
1					
2					
3					
4					
5					
6					
7					
8					



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); a maximum of two instruction 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.