22nd SCSE – Past Year Paper Solution (2021 – 2022 Semester 1)
CE/CZ 3001 – Advanced Computer Architecture

1 **(a)** Let $t_{original}$ be x.

Enhancement E1 applies to 20% of original instructions and have speedup factor of 8.

- $t_{E1} = t$ for portion of application affected by enhancement + t for portion of application not affected by enhancement
  $= \frac{0.2x}{8} + 0.8x = 0.825x$

Enhancement E2 is as effective as E1 and applies to 35% of original instructions.

- $t_{E2} = t_{E1} = 0.825x$
- $0.825x = \frac{0.35x}{Speedup} + 0.65x$
- $Speedup = \frac{0.35x}{0.825x - 0.65x} = \mathbf{2}$

The concern of the supervisor is according to Amdahl's Law, speedup via parallelism is limited by the component of the application that cannot be enhanced.

Using formula for upper limit of enhancement

$$\lim_{Speedup \to \infty} \left(\frac{1}{\frac{Fraction\ of\ application\ sped\ up}{Speedup} + Fraction\ of\ unaffected\ application} \right)$$

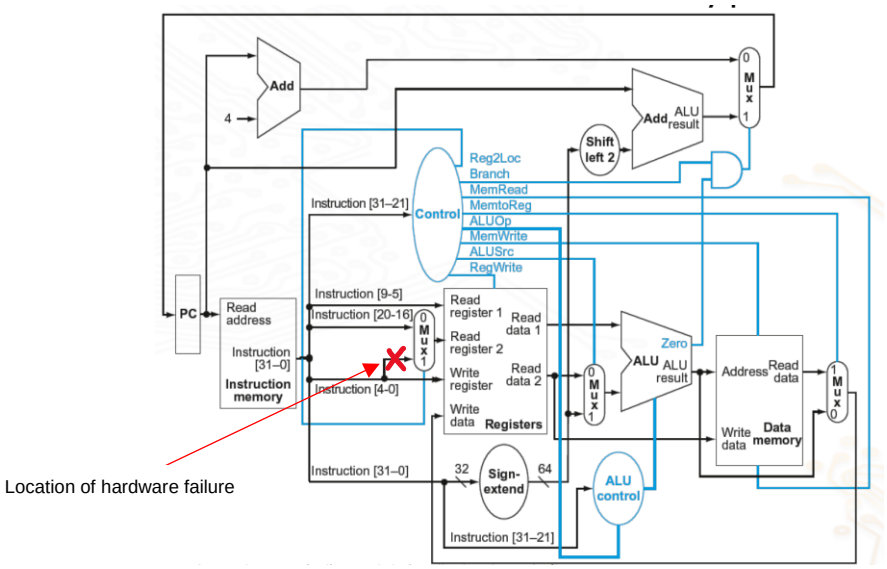
$$= \frac{1}{Fraction\ of\ unaffected\ application}$$

- For E1:
  - Speedup is limited by a factor of 100/20 = 5
  - Upper limit of enhancement = $\frac{1}{0.8} = 1.25$
- For E2:
  - Speedup is limited by a factor of 100/35 = 2.857
  - Upper limit of enhancement = $\frac{1}{0.65} = 1.538$

Comparing both enhancements, E1 have a higher speedup limiting factor and lower upper limit for its enhancement as speedup factor approaches infinity, hence making E2's speedup effect more noticeable despite having a much lower speedup factor than E1.

(b) LDUR X0, [X1, #8] – Fetch value stored at memory address (value of reg X1 + 8) and store it in reg X0.

CBZ X0, #8 – If value in reg X0 is == 0, branch to address of value of current PC + 8.



opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

For LDUR X0, [X1, #8]

Instruction is writing value stored in memory to register. Looking at the instruction format, bit 0 to 4 is Rt which is writing to register using the write register port of the registers file, and bit 5 to 9 is Rn which is reading from register. Since the data path from the instruction memory to the write register port does not need to go through the hardware failure just before the MUX of read register 2 port for the instruction to function correctly, LDUR will have no issues executing.

opcode	address	Rt
8 bits	19 bits	5 bits

For CBZ X0, #8

Instruction is reading the value of the register for a conditional branch. Looking at the instruction format, Rt which is bit 0 to 4 is the bits used to instruct the register file which registers to read from. This Rt is different from the above LDUR instruction as for LDUR, it is a write operation while for CBZ, it is a read operation. Since CBZ is reading from the register file and the data path for Instruction [4-0] to Read Register 2 Port of the Register File is broken, this instruction will have trouble executing properly as the instruction will not be able to specify to the register file which register's value to read from.

Note: Instruction format can be obtained from the exam's paper appendix section.

(c) Initial PC value = 0x10FC

opcode	address
6 bits	26 bits

Unconditional branch only have 26 bits for immediate addressing.

- Value is then multiplied by 4 (Non-Rotate Shift left 2 before the ALU, refer to data path diagram above)

Since value is in 2's complement, maximum value for immediate addressing is 0x01FF FFFF, minimum value is 0xFE00 0000. Multiplying by 4 gives 0x07FF FFFC and 0xF800 0000 respectively

Using initial PC value,

- Max possible branch value = **0x0800 10F8.**
- Min possible branch value = **0xF800 10FC.**
- Hence range is **0xF800 10FC to 0xFFFF FFFF and 0x0000 0000 to 0x0800 10F8.**

2 **(a)** Data hazard – Either source or destination register of an instruction are not available at the time expected in the pipeline, causing a pipeline stall.

- Eg. LDUR X1, [X0, #0] then ADDI X1, #8: Instruction ADDI needing to wait for LDUR instruction to update X1 before executing.

Structural hazard – When instructions require the use of a given hardware resource at the same time, leading to stalling in pipeline due to limited hardware resource.

- Eg. When CPU requires the memory bus, but DMA controller is currently using the memory bus.

Control hazard - Stalling in the pipeline either due to a delay in obtaining the results of a conditional/unconditional branching or due to a delay in the availability of an instruction

- Eg. CBZ X1, Loop then ORR X2, X3, X4: instruction ORR will have to wait for the results of the CBZ instruction before executing in the pipeline, causing a stall.

(b) **(i)** Full data forwarding is allowed, hence once value from a previous instruction is ready (at Execute stage), it can be brought forward earlier for instructions that need the value.

For branching, PC value is updated with branch target address at Execute Stage, provided by question. Hence next fetch will happen after it.

Reservation Table: Circled in red are dependencies brought forward by data forwarding.

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13
LDUR X0, [X7, #0]	F	D	E	M	W								
LDUR X1, [X8, #0]		F	D	E	M	W							
XOR X2, X1, X0			F	D	N	E	M	W					
STUR X2, [X7, #0]				F	N	D	E	M	W				
SUBI X7, X7, #8						F	D	E	M	W			
SUBI X8, X8, #16							F	D	E	M	W		
CBZ X8, finish								F	D	E	M	W	
Next cycle/B loop									N	N	F	D	E

Total of 3 stalls each cycle.

Total cycles = X8 / 16 = 0x1100 / 16 = $\frac{4352}{16}$ = **272 cycles**

Each cycle contains 7 instructions, 10 cycles.

Steady-state CPI (infinite loops)
= Total cycles in a loop / Total instructions in a loop
= $\frac{7+3}{7}$
= **1.428**

(ii) This question's solution made heavy references to part (i). Nothing has changed on the dependencies and the cause of the stall cycles. There is still data forwarding.

- SUBI X8, X8, #16 is brought forward as only LDUR X1, [X8, #0] needs the old value of X8.
- This solution assumes we can reorder the instructions as only 1 of the processors can execute D-type instructions only.

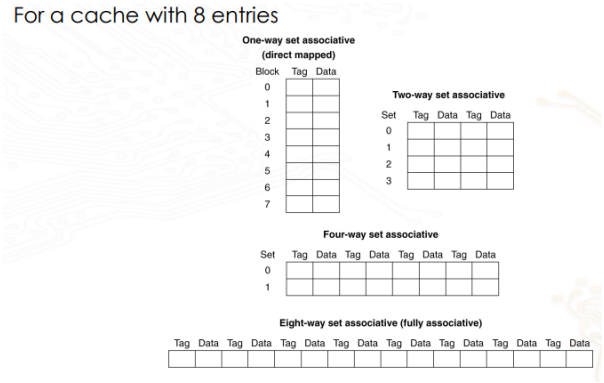
Cycle	Way-1 (LDUR/STUR only)	Way-2
1	LDUR X0, [X7, #0]	NOP
2	LDUR X1, [X8, #0]	SUBI X8, X8, #16
3	NOP	NOP
4	NOP	XOR X2, X1, X0
5	STUR X2, [X7, #0]	SUBI X7, X7, #8
6	NOP	CBZ X8, finish
7	NOP	NOP
8	NOP	NOP
9	NOP/Next Cycle	B loop

Hence, steady-state CPI = Total Cycles in a loop / Total instructions in a loop
= $\frac{8}{7}$ = **1.143**

(iii) Branch prediction or delayed branching could potentially cut stalls by 2, lowering the number of cycles taken per loop to 6.

Loop unrolling is another method to reduce stalls in the 2-way superscalar pipeline as it reduces control hazard and potentially data hazards through reordering of instructions.

3 (a)



Direct-mapped cache - Implements a mapping scheme that results in main memory blocks being mapped in a modular fashion to certain specific sets.

Full associative cache – Allows the main memory block to be placed anywhere in the cache.

Set associative cache – A combination of both direct-mapped and full associative. Compared to Direct-Mapped, Set associative can store more than 1 memory blocks in a set.

(b) (i) From Question, L1 is an eight-way set associative cache containing total of 1024 cache blocks.

- No. of L1 cache indexes = 1024 / 8 = 128
- Hence, no. of Y bits needed = $\log_2(128)$ = **7 bits.**

(ii) Given size of main memory = 2GB = 2^{31} B, physical address will need to have 31-bits of addressing space. Therefore X = 31.

From Figure Q3, since L1 Tag bits = 16 and we know L1 Cache Index bits from Q3b(i) which is 7, we can deduce that the L1 Block Offset bits = 31 – 16 – 7 = 8 bits.

Hence, it is possible to know the block size of L1 cache given the size of the main memory. L1 block size = 2^8 = **256 Bytes.**

(iii) From Question, L2 is a two-way set associative cache.
From Figure Q3, L2 Cache Index bits is 11.
From part(i), we know that L1 index Offset bits is 8 bits.

- Therefore, Block Offset bits = 31 – 7 – 16 = 8 bits.

Block Size = 2^8 Bytes
Total bits for valid, LRU, dirty and TAG per block = 3 + (31 – 8 – 11) = 15 bits

Size of cache = (Block Size + Bytes used for LRU, dirty, TAG, valid) * No. of Blocks
= $(2^8 + \frac{15}{8}) * 2^{12}$
= $2^{20} + 15 * 2^9$
= **1031.5 KB**

(iv) From Question, miss Rate for L1 and L2: 2% and 1% respectively.
Time taken for access to L1, L2 and main memory: 2, 25, 200 cycles respectively.

<u>Without L2</u>
Avg. Memory Access Time
= Probability of L1 hit * Time to access L1 + Probability of L1 miss * Time to access main memory
= (1 - 0.02)(2) + (0.02)(200)
= 5.96 cycles

<u>With L2</u>
Avg. Memory Access Time
= Probability of L1 hit * Time to access L1 + Probability of L1 miss * (Probability of L2 hit * Time to access L2 + Probability of L2 miss * Time to access main memory)
= (1 – 0.02)(2) + (0.02) [(1 – 0.01)(25) + (0.01)(200)]
= 2.495 cycles

Speedup when L2 cache is added
= Perf with L2 / Perf without L2
= Time without L2 / Time with L2
= 5.96 / 2.495
= **2.389**

4 **(a)** Flynn's classification system
SISD – Single instruction, single data stream
SIMD – Single instruction, multiple data stream
MISD – Multiple instruction, single data stream
MIMD – Multiple instruction, multiple data stream

GPU belongs to SIMD as it operates on 1 instruction to process a lot of data, which is applicable for processing data for displays.

(b) __global__ keyword is used to declare a specific function to be run on an external device instead of the host's CPU, most of the time a GPU. This will cause the function to be compiled on the external device.

__shared__ keyword is used when an external device, usually a GPU, have a segment in the code whereby data in the device's memory is required to be shared or communicated among the multiple threads in a block. This will tell the external's device compiler that the variables with __shared__ keyword declared has values that are not limited to only the executing thread itself, but rather the whole memory block.

(c) kernel_A<<<4,128>>> will launch with 4 blocks of 128 threads, with a total of 512 threads. The compiler will execute the code defined under __global__ function kernel_A with these threads.

Each block is then executed by any available streaming multiprocessor (SM) in the GPU either concurrently or sequentially assigned by the runtime system. Blocks cannot split among multiple SMs. The block is then further partitioned into groups of 32 threads known as a warp. All threads in a warp must execute the same instruction but with different data. Each warp is then scheduled by a warp scheduler for execution and only 1 warp can execute at any one time on a CUDA core. The need for a warp scheduler is because often, the number of threads assigned is more than the amount of CUDA cores available.

Due to the limited amount of hardware resources, the number of registers and amount of memory used per thread is capped in a SM. The more resources a thread requires, the lesser the number of thread that can simultaneously reside in the SM, reducing parallelism.

To identify the individual threads during execution, we use variables blockDim.x, blockIDx.x and threadIdx.x which when used in combination (blockIdx.x * blockDim.x + threadIdx.x), can return the ID of the thread the program is executing on currently.

(d) For function *kernel_B,* int n and int x are passed into the function by value. They are not integer pointers.

- Hence, only 1 instance of n and x are passed into the function and the same value of n and x is repeated for all the threads that will be executing this function. This means the results from the threads execution will also be the same.

- Also, by passing x as an integer instead of an integer pointer, the program will not allocate the same memory slot as the original x whose value is being passed into the function. This means the thread could not return the results properly and once the function finish executing, the final value of x from the thread execution will be deleted as it does not share the same memory slot as the original.

- Even if int n and int x are passed as pointers, they are not indexed, meaning the first issue still persists.

Overall, the function does not make use of the multi-threading function of the GPU. To fix this, int n and int x should be declared as an integer pointer in the function parameters so that the results from the thread execution can be passed back. They also should be indexed according to the thread they are running on through the use of blockIdx.x * blockDim.x + threadIdx.x so that each thread will have different data streams.

Note: I am not too sure about my solution to this question, please take it with a pinch of salt. There are also different ways to answer this question.

Solver: Foo Jun Ming (fooj0026@e.ntu.edu.sg)