CMU 18-447 Introduction to Computer Architecture, Spring 2015
HW 4: SIMD, VLIW, Static Scheduling, Caching, Virtual Memory

Instructor: Prof. Onur Mutlu
TAs: Kevin Chang, Rachata Ausavarungnirun, Albert Cho, Jeremie Kim, Clement Loh

Assigned: Wed., 2/25, 2015
Due: **Wed., 3/18, 2015 (Midnight)**
Handin: autolab

1 Vector Processing [15 points]

Consider the following piece of code:

```
for (i = 0; i < 100; i ++)
  A[i] = ((B[i] * C[i]) + D[i])/2;
```

(a) Translate this code into assembly language using the following instructions in the ISA (note the number of cycles each instruction takes is shown next to each instruction):

Opcode	Operands	Number of Cycles	Description
LEA	Ri, X	1	Ri ← address of X
LD	Ri, Rj, Rk	11	Ri ← MEM[Rj + Rk]
ST	Ri, Rj, Rk	11	MEM[Rj + Rk] ← Ri
MOVI	Ri, Imm	1	Ri ← Imm
MUL	Ri, Rj, Rk	6	Ri ← Rj x Rk
ADD	Ri, Rj, Rk	4	Ri ← Rj + Rk
ADD	Ri, Rj, Imm	4	Ri ← Rj + Imm
RSHFA	Ri, Rj, amount	1	Ri ← RSHFA (Rj, amount)
BRcc	X	1	Branch to X based on condition codes

Assume one memory location is required to store each element of the array. Also assume that there are 8 registers (R0 to R7).

Condition codes are set after the execution of an arithmetic instruction. You can assume typically available condition codes such as zero, positive, and negative.

Solution:

```
MOVI    R1, 99     // 1 cycle
LEA     R0, A      // 1 cycle
LEA     R2, B      // 1 cycle
LEA     R3, C      // 1 cycle
LEA     R4, D      // 1 cycle
LOOP:
LD      R5, R2, R1 // 11 cycles
LD      R6, R3, R1 // 11 cycles
MUL     R7, R5, R6 // 6 cycles
LD      R5, R4, R1 // 11 cycles
ADD     R8, R7, R5 // 4 cycles
RSHFA   R9, R8, 1  // 1 cycle
ST      R9, R0, R1 // 11 cycles
ADD     R1, R1, -1 // 4 cycles
BRGEZ   R1  LOOP   // 1 cycle
```

How many cycles does it take to execute the program?

Solution:
$5 + 100 \times 60 = 6005$

(b) Now write Cray-like vector assembly code to perform this operation in the shortest time possible. Assume that there are 8 vector registers and the length of each vector register is 64. Use the following instructions in the vector ISA:

Opcode	Operands	Number of Cycles	Description
LD	Vst, #n	1	Vst ← n (Vst = Vector Stride Register)
LD	Vln, #n	1	Vln ← n (Vln = Vector Length Register)
VLD	Vi, X	11, pipelined	
VST	Vi, X	11, pipelined	
Vmul	Vi, Vj, Vk	6, pipelined	
Vadd	Vi, Vj, Vk	4, pipelined	
Vrshfa	Vi, Vj, amount	1	

Solution:

```
LD      Vln, 50
LD      Vst, 1
VLD     V1,  B
VLD     V2,  C
VMUL    V4,  V1,  V2
VLD     V3,  D
VADD    V6,  V4, V3
VRSHFA V7,  V6, 1
VST     V7,  A

VLD     V1,  B + 50
VLD     V2,  C + 50
VMUL    V4,  V1,  V2
VLD     V3,  D + 50
VADD    V6,  V4, V3
VRSHFA V7,  V6, 1
VST     V7,  A + 50
```

How many cycles does it take to execute the program on the following processors? Assume that memory is 16-way interleaved.

(i) Vector processor without chaining, 1 port to memory (1 load or store per cycle)

Solution:

The third load (VLD) can be pipelined with the add (VADD). However as there is just only one port to memory and no chaining, other operations cannot be pipelined.

Processing the first 50 elements takes 346 cycles as below

```
| 1 | 1 | 11 | 49 | 11 | 49 | 6 | 49 |
                                | 11 | 49 | 4 | 49 | 1 | 49 | 11 | 49 |
```

Processing the next 50 elements takes 344 cycles as shown below (no need to initialize Vln and Vst as they stay at the same value).

```
| 11 | 49 | 11 | 49 | 6 | 49 |
                      | 11 | 49 | 4 | 49 | 1 | 49 | 11 | 49 |
```

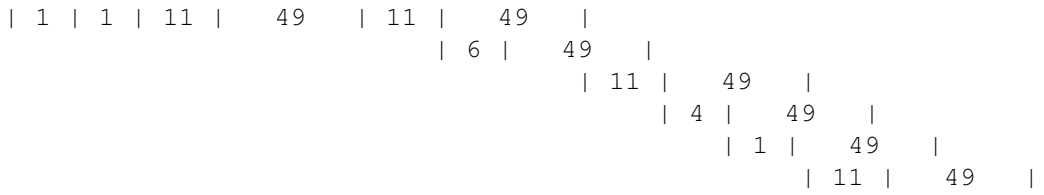
Therefore, the total number of cycles to execute the program = 690 cycles

(ii) Vector processor with chaining, 1 port to memory

Solution:

In this case, the first two loads cannot be pipelined as there is only one port to memory and the third load has to wait until the second load has completed. However, the machine supports chaining, so all other operations can be pipelined.

Processing the first 50 elements takes 242 cycles as below



Processing the next 50 elements takes 240 cycles (same time line as above, but without the first 2 instructions to initialize Vln and Vst).

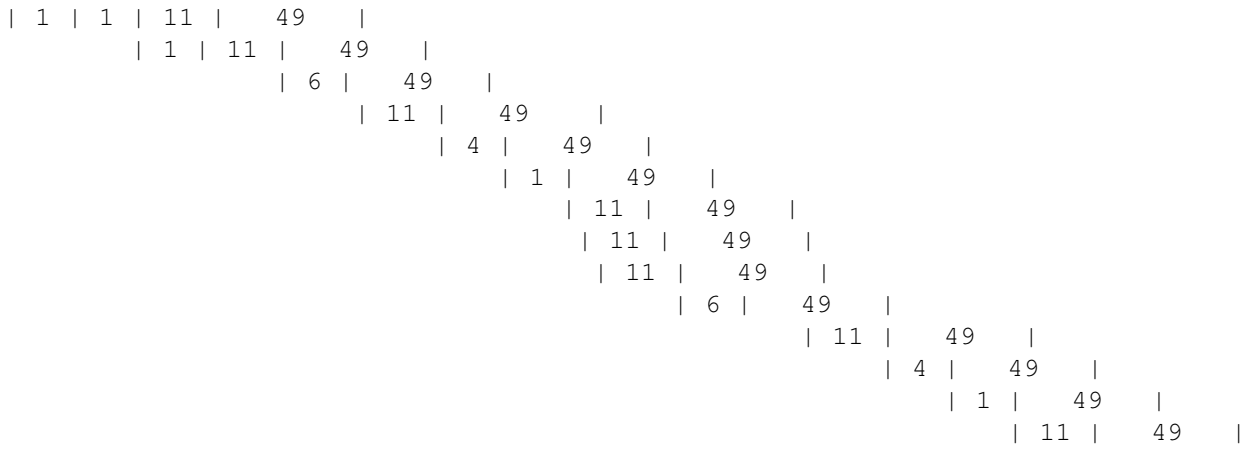
Therefore, the total number of cycles to execute the program = 482 cycles

(iii) Vector processor with chaining, 2 read ports and 1 write port to memory

Solution:

Assuming an in-order pipeline.

The first two loads can also be pipelined as there are two ports to memory. The third load has to wait until the first two loads complete. However, the two loads for the second 50 elements can proceed in parallel with the store.



Therefore, the total number of cycles to execute the program = 215 cycles

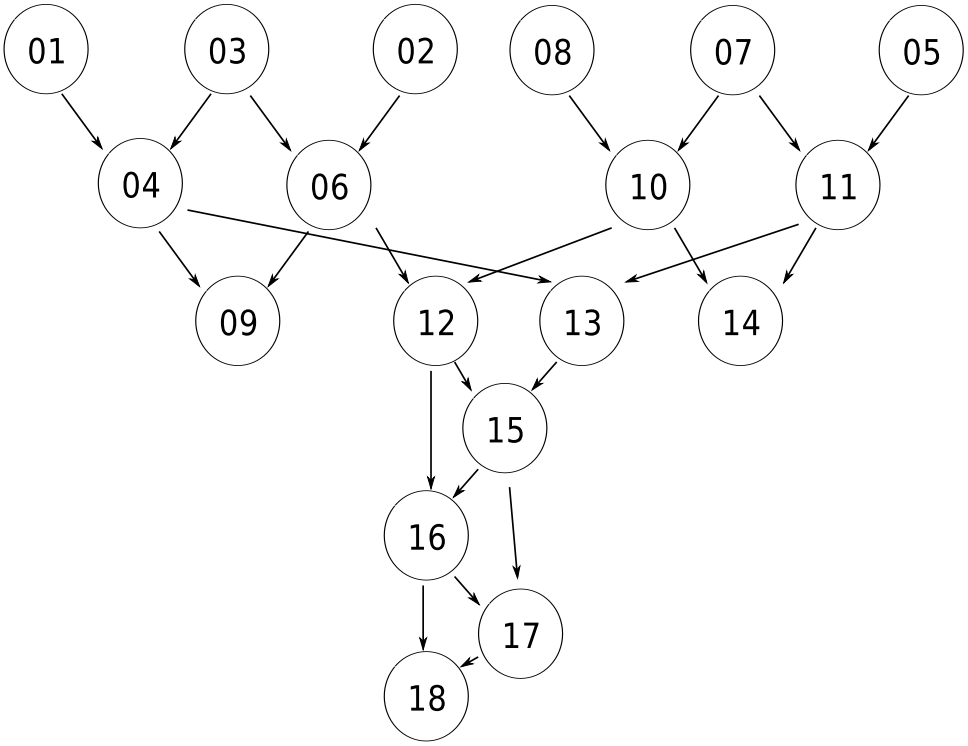
2 VLIW [15 points]

You are using a tool that transforms machine code that is written for the MIPS ISA to code in a VLIW ISA. The VLIW ISA is identical to MIPS except that multiple instructions can be grouped together into one *VLIW instruction*. Up to N MIPS instructions can be grouped together (N is the machine width, which depends on the particular machine). The transformation tool can reorder MIPS instructions to fill VLIW instructions, as long as loads and stores are not reordered relative to each other (however, independent loads and stores can be placed in the same VLIW instruction). You give the tool the following MIPS program (we have numbered the instructions for reference below):

```
(01) lw   $t0 ← 0($a0)
(02) lw   $t2 ← 8($a0)
(03) lw   $t1 ← 4($a0)
(04) add  $t6 ← $t0, $t1
(05) lw   $t3 ← 12($a0)
(06) sub  $t7 ← $t1, $t2
(07) lw   $t4 ← 16($a0)
(08) lw   $t5 ← 20($a0)
(09) srlv $s2 ← $t6, $t7
(10) sub  $s1 ← $t4, $t5
(11) add  $s0 ← $t3, $t4
(12) sllv $s4 ← $t7, $s1
(13) srlv $s3 ← $t6, $s0
(14) sllv $s5 ← $s0, $s1
(15) add  $s6 ← $s3, $s4
(16) add  $s7 ← $s4, $s6
(17) srlv $t0 ← $s6, $s7
(18) srlv $t1 ← $t0, $s7
```

(a) Draw the dataflow graph of the program Represent instructions as numbered nodes (01 through 18), and flow dependences as directed edges (arrows).

Solution:



(b) When you run the tool with its settings targeted for a particular VLIW machine, you find that the resulting VLIW code has 9 VLIW instructions. What minimum value of N must the target VLIW machine have?

Solution:

$N = 3$ (see VLIW program below). If $N = 2$, then the VLIW program must have at least 11 MIPS instructions, and the number of VLIW instructions either stays the same or decreases as width is increased by one MIPS instruction.

(c) Write the MIPS instruction numbers (from the code above) corresponding to each VLIW instruction, for this value of N. When there is more than one MIPS instruction that could be placed into a VLIW instruction, choose the instruction that comes earliest in the original MIPS program.

Solution:

	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.
VLIW Instruction 1:	01	02	03							
VLIW Instruction 2:	04	05	06							
VLIW Instruction 3:	07	08	09							
VLIW Instruction 4:	10	11								
VLIW Instruction 5:	12	13	14							
VLIW Instruction 6:	15									
VLIW Instruction 7:	16									
VLIW Instruction 8:	17									
VLIW Instruction 9:	18									

(d) You find that the code is still not fast enough when it runs on the VLIW machine, so you contact the VLIW machine vendor to buy a machine with a larger machine width N. What minimum value of N would yield the maximum possible performance (i.e., the fewest VLIW instructions), assuming that all MIPS instructions (and thus VLIW instructions) complete with the same fixed latency and assuming no cache misses?

Solution:

$N = 6$. This is the maximum width of the dataflow graph and results in 7 VLIW instructions (see below). If $N = 5$, then the VLIW program will instead have 8 VLIW instructions. Increasing N further does not allow any more MIPS instructions to be parallelized in wider VLIW instructions.

(e) Write the MIPS instruction numbers corresponding to each VLIW instruction, for this optimal value of N. Again, as in part (c) above, pack instructions such that when more than one instruction can be placed in a given VLIW instruction, the instruction that comes first in the original MIPS code is chosen.

Solution:

	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.	MIPS Inst. No.
VLIW Instruction 1:	01	02	03	05	07	08				
VLIW Instruction 2:	04	06	10	11						
VLIW Instruction 3:	09	12	13	14						
VLIW Instruction 4:	15									
VLIW Instruction 5:	16									
VLIW Instruction 6:	17									
VLIW Instruction 7:	18									
VLIW Instruction 8:										
VLIW Instruction 9:										

(f) A competing processor design company builds an in-order superscalar processor with the same machine width N as the width you found in part (b) above. The machine has the same clock frequency as the VLIW processor. When you run the original MIPS program on this machine, you find that it executes slower than the corresponding VLIW progra m on the VLIW machine in part (b). Why could this be the case?

Solution:

Concurrently fetched instructions can be dependent in a superscalar processor, requiring bubbles in the pipeline to be processed. A VLIW code translator can reorder instructions to minimize such bubbles.

Note that the superscalar processor is in-order in this question.

(g) When you run some other program on this superscalar machine, you find it runs faster than the corresponding VLIW program on the VLIW machine. Why could this be the case?

Solution:

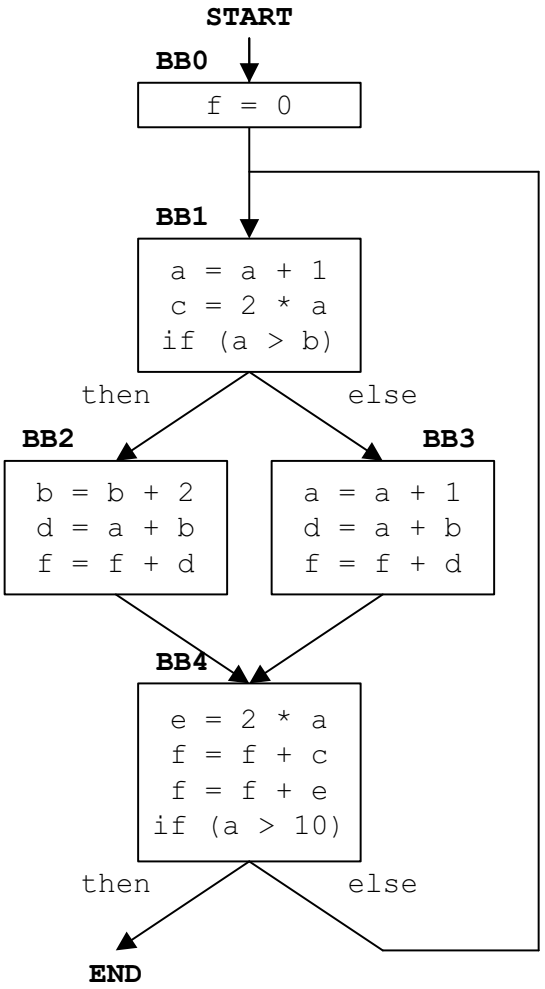
VLIW code must have explicit NOPs; the superscalar processor does not require these NOPs. Higher code density results in a higher I-cache hit rate and lower required fetch bandwidth.

3 Code Optimizations [20 points]

Assume an **in-order**, **non-pipelined** machine. The ISA for this machine consists of only the following five instructions that never generate exceptions. In this syntax, x is a register, whereas y and z are either registers or constants. Note that this machine is able to execute branches and jumps **instantaneously**.

Instruction Name	Syntax	Cycles
Move	$x = y$	1
Addition	$x = y + z$	2
Multiplication	$x = y * z$	5
Branch-If-Greater-Than	`if` $(y > z)$	0
Jump	–	0

Consider the following assembly program that is written in the machine's ISA. For any initial values of registers a and b, the purpose of the program is to compute the final value and store it in register f. After the program terminates, it is considered to have executed correctly if and only if the values stored in registers a, b, and f are correct.



(a) How many cycles does the machine take to execute the assembly program, when a and b are initialized to 0 and 1, respectively?

In this program, the two registers a and b are racing to become larger than the other. `If` ($a > b$) `then` b is incremented so that it can "catch up" to a; `else` a is incremented so that it can "catch up" to b. However, this game of "leap frog" is rigged in favor of a, because a is guaranteed to be incremented by 1 at every iteration of the loop. Therefore, when b is trying to catch up, it can only close the gap by 1, even though b itself is incremented by 2. On the other hand, when a is trying to catch up, it always closes the gap by 2. To give b more turns to catch up, the majority of the loop iterations execute `BB2`, rather than `BB3`.

Initially: (a, b) = (0, 1)

Iteration #1:
After BB1: (1, 1)
Before BB4: (2, 1)

Iteration #2:
After BB1: (3, 1)
Before BB4: (3, 3)

Iteration #3:
After BB1: (4, 3)
Before BB4: (4, 5)

Iteration #4:
After BB1: (5, 5)
Before BB4: (6, 5)

Iteration #5:
After BB1: (7, 5)
Before BB4: (7, 7)

Iteration #6:
After BB1: (8, 7)
Before BB4: (8, 9)

Iteration #7:
After BB1: (9, 9)
Before BB4: (10, 9)

Iteration #8:
After BB1: (11, 9)
Before BB4: (11, 11)
Exit loop.

Each iteration executes 8 instructions: 6 additions and 2 multiplications. Per-iteration execution time: 6*2 + 2*5 = 22 cycles. Initializing f takes 1 cycle.

Answer: 8-iterations * 22-cycles/iteration + 1-cycle = 177 cycles.

You decide to make simple optimizations to the program to reduce its execution time. The optimizations involve only **removing**, **modifying**, and/or **moving** some of the already existing instructions. However, there are two restrictions: you may **not** move instructions out of the loop and you may **not** add completely new instructions.

(b) Show the optimized assembly program.

```
BB0:
f = 0

BB1:
a = a + 1
c = a + a /* CHANGE MULTIPLICATION TO ADDITION */
if (a > b)

BB2:
b = b + 2
~~d = a + b~~ /* MOVE */
~~f = f + d~~ /* MOVE */

BB3:
a = a + 1
~~d = a + b~~ /* MOVE */
~~f = f + d~~ /* MOVE */

BB4:
d = a + b /* MOVE */
f = f + d /* MOVE */
e = a + a /* CHANGE MULTIPLICATION TO ADDITION */
f = f + c
f = f + e
if (a > 10)
```

(c) How many cycles does the machine take to execute the optimized assembly program, when a and b are initialized to 0 and 1?

Each iteration executes 8 instructions: 8 additions.
Per-iteration execution time: 8*2 = 16 cycles.

Answer: 8-iterations * 16-cycles/iteration + 1-cycle = 129 cycles.

After learning about *superblocks*, you decide to optimize the program even further to reduce its execution time. In order to form the superblock(s), assume that you run the program once beforehand when a and b are initialized to 0 and 1. During this profile run, if a branch is biased in either direction by more than **60%**, it is included in the superblock(s). However, there are two restrictions: you may **not** move instructions out of the loop and you may **not** unroll the loop.

(d) Show the superblock-optimized assembly program. **Circle** the superblock(s).

```
BB0:
f = 0

BB1:
a = a + 1
c = a + a
if (a > b)

BB2:
b = b + 2

BB2-TAIL:
d = a + b
f = f + d
~~e = a + a~~ /* REMOVE UNNECESSARY COMPUTATION */
f = f + c
f = f + c /* SUBSTITUTE c for e */
if (a > 10)

BB3:
a = a + 1

BB3-TAIL:
d = a + b
f = f + d
e = a + a
f = f + c
f = f + e
if (a > 10)
```

The superblock consists of `BB1`, `BB2`, and `BB2-TAIL`. Recall from (a) that the branch is biased towards BB2, not BB3.

(e) How many cycles does the machine take to execute the superblock-optimized assembly program, when a and b are initialized to 0 and 1?

5 out of 8 iterations take BB2. Those iterations execute 7 instructions: 7 additions. Execution time: 7*2 = 14 cycles.

3 out of 8 iterations take BB3. Those iterations execute 8 instructions: 8 additions. Execution time: 8*2 = 16 cycles.

Answer: 5-iterations * 14-cycles/iteration + 3-iterations * 16-cycles/iteration + 1 = 119 cycles.

(f) If you had used *traces* to optimize the program instead of superblocks, would the execution time increase, decrease, or stay the same compared to (e)? Choose one and explain briefly why.

Increase. Requires fix-up code.

(g) If you had used *hyperblocks* to optimize the program instead of superblocks, would the execution time increase, decrease, or stay the same compared to (e)? Choose one and explain briefly why.

Increase. Cannot perform the optimizations done with superblocks since we do not know the outcome of the predicated instructions. Also, useless instructions from one of the branch paths are executed due to predication.

4 Caching [15 points]

Below, we have given you four different sequences of addresses generated by a program running on a processor with a data cache. Cache hit ratio for each sequence is also shown below. Assuming that the cache is initially empty at the beginning of each sequence, find out the following parameters of the processor's data cache:

- Associativity (1, 2 or 4 ways)
- Block size (1, 2, 4, 8, 16, or 32 bytes)
- Total cache size (256 B, or 512 B)
- Replacement policy (LRU or FIFO)

Assumptions: all memory accesses are one byte accesses. All addresses are byte addresses.

Sequence No.	Address Sequence	Hit Ratio
1	0, 2, 4, 8, 16, 32	0.33
2	0, 512, 1024, 1536, 2048, 1536, 1024, 512, 0	0.33
3	0, 64, 128, 256, 512, 256, 128, 64, 0	0.33
4	0, 512, 1024, 0, 1536, 0, 2048, 512	0.25

Solution:
Cache block size - 8 bytes
For sequence 1, only 2 out of the 6 accesses (specifically those to addresses 2 and 4) can hit in the cache, as the hit ratio is 0.33. With any other cache block size but 8 bytes, the hit ratio is either smaller or larger than 0.33.
Therefore, the cache block size is 8 bytes.

Associativity - 4 For sequence 2, blocks 0, 512, 1024 and 1536 are the only ones that are reused and could potentially result in cache hits when they are accessed the second time. Three of these four blocks should hit in the cache when accessed for the second time to give a hit rate of 0.33 (3/9).

Given that the block size is 8 and for either cache size (256B or 512B), all of these blocks map to set 0. Hence, an associativity of 1 or 2 would cause at most one or two of these four blocks to be present in the cache when they are accessed for the second time, resulting in a maximum possible hit rate of less than 3/9. However, the hit rate for this sequence is 3/9. Therefore, an associativity of 4 is the only one that could potentially give a hit rate of 0.33 (3/9).

Total cache size - 256 B
For sequence 3, a total cache size of 512 B will give a hit rate of 4/9 with a 4-way associative cache and 8 byte blocks regardless of the replacement policy, which is higher than 0.33. Therefore, the total cache size is 256 bytes.

Replacement policy - LRU
For the aforementioned cache parameters, all cache lines in sequence 4 map to set 0. If a FIFO replacement policy were used, the hit ratio would be 3/8, whereas if an LRU replacement policy were used, the hit ratio would be 1/4. Therefore, the replacement policy is LRU.

5 Virtual Memory [10 points]

An ISA supports an 8-bit, byte-addressable virtual address space. The corresponding physical memory has only 128 bytes. Each page contains 16 bytes. A simple, one-level translation scheme is used and the page table resides in physical memory. The initial contents of the frames of physical memory are shown below.

Frame Number	Frame Contents
0	Empty
1	Page 13
2	Page 5
3	Page 2
4	Empty
5	Page 0
6	Empty
7	Page Table

A three-entry translation lookaside buffer that uses Least Recently-Used (LRU) replacement is added to this system. Initially, this TLB contains the entries for pages 0, 2, and 13. For the following sequence of references, put a circle around those that generate a TLB hit and put a rectangle around those that generate a page fault. What is the hit rate of the TLB for this sequence of references? (Note: LRU policy is used to select pages for replacement in physical memory.)

References (to pages): 0, 13, 5, 2, 14, 14, 13, 6, 6, 13, 15, 14, 15, 13, 4, 3.

Solution:
References (to pages): (0), (13), 5, 2, [14], (14), 13, [6], (6), (13), [15], 14, (15), (13), [4], [3].

TLB Hit Rate = 7/16

(a) At the end of this sequence, what three entries are contained in the TLB?

Solution:
4, 13, 3

(b) What are the contents of the 8 physical frames?

Solution:
Pages 14, 13, 3, 2, 6, 4, 15, Page table

6 Page Table Bits [5 points]

(a) What is the purpose of the "reference" or "accessed" bit in a page table entry?

Solution:

To aid page replacement.

(b) Describe what you would do if you did not have a reference bit in the PTE. Justify your reasoning and/or design choice.

Solution:

Pick a random page to replace when a page fault occurs.

(c) What is the purpose of the dirty or modified bit in a page table entry?

Solution:

To enable writeback of only dirty pages (rather than all pages) to disk.

(d) Describe what you would do if you did not have a modified bit in the PTE. Justify your reasoning and/or design choice.

Solution:

Write back all pages to disk.

Alternative answer: the OS could map all pages read-only by default. On a page-fault due to a write, if the program has permission to change the page, the operating system remaps the page as read-write and also knows that the page has become dirty.

7 Virtual Memory - Optional(From past exam..)

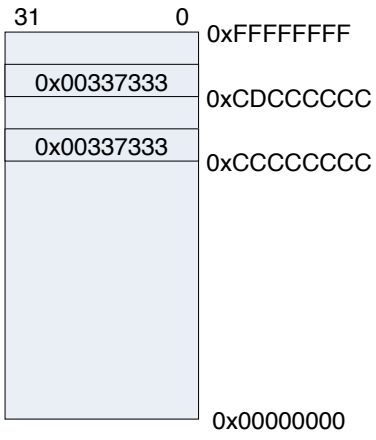
A dedicated computer architecture student (like you) bought a 32-bit processor that implements paging-based virtual memory using a single-level page table. Being a careful person, she also read the manual and learnt that

- A page table entry (PTE) is 4-bytes in size.
- A PTE stores the physical page number in the least-significant bits. Unused bits are zeroed out.
- The page table base address (PTBA) is page-aligned.
- If an instruction attempts to modify a PTE, an exception is raised.

However, the manual did not mention the page size and the PTBA. The dedicated student then wrote the following program to find out the missing information.

char *ptr = 0xCCCCCCCC;
∗ptr = 0x00337333;

The code ran with no exceptions. The following figure shows relevant locations of the physical memory **after** execution.



Using these results, what is the PTBA of this machine?

Let n = log2(page size).

Maximum possible number of unused bits in a PTE is 10. This means n $\leqslant$ 10.

0xC = 0b1100

If n % 4 == 0 ⇒ PTE offset ends with 0b110000, PTBA is not page aligned.
If n % 4 == 1 ⇒ PTE offset ends with 0b11000, PTBA is not page aligned.
If n % 4 == 2 ⇒ PTE offset ends with 0b1100, PTBA is aligned.
If n % 4 == 3 ⇒ PTE offset ends with 0b110, PTBA is not page aligned.

This leaves 10, 6, 2 as the only possible values of n.

Since all relevant memory locations are shown, the write must have occured to one of the two entries. However, neither n = 2 or or n = 6 results in a physical address that maps to the two locations.

This leaves n = 10.

Now suppose if 0xCDCCCCCC was PTE. Translating to physical address leads to 0xCDCCCCCC again. This would have triggered an exception.

This means 0xCCCCCCCC was the PTE, and the value was written to 0xCDCCCCCC.

Using this information, PTBA = 0xCCCCCCCC - (0x333333<<2) = 0xCC000000.

What is the page size (in bytes) of this machine? Write your answer in the form 2^n.

Since n = 10, The page size is 2^{10}.