Mid-term 3, 2013.

NAME Soln **NOTE: includes some corrections to questions.**

READ THIS

The purpose of exam questions is to gauge the degree to which you have absorbed the material covered. Be aware of each question's goal: try to demonstrate (any) relevant knowledge. Explain your thinking. You may insert comments and explanations that are off the topic. Some questions likely have errors. Comment on such difficulties: saying something like, "This question is stupid," is a good start. Select your questions: do the easier first; come back to the harder; do parts of a question. Partial and extra credit will be given liberally. Obviously, longer and harder problems get proportionally more credit.
Notation: "#" decimal number, "x" hex number; any other number is decimal.)

A user process runs on this LC3 system:
--- Virtual Memory (VM) with a TLB
--- 16-bit virtual and physical addresses, word addressable
--- 16-bit data words and instructions
--- 4k word pages and frames: x1000 words per page

The process's C code and assembly code is at right. The data segment includes a one-thousand word array, A. (Recall that ".BLKW 1000" is equivalent to 1,000 lines of ".FILL x0".) The VM map is shown below. Register values are shown just before the STR instruction executes.

Execution proceeds as follows:
--- Preamble: GDP <=== x5000; jump to "main".
--- main: R1 <=== #99
--- main: R2 <=== #6666
--- main: R3 <=== address of array A == (GDP + 2)
--- main: R3 <== R3 + R2 == (address of A[#6666])

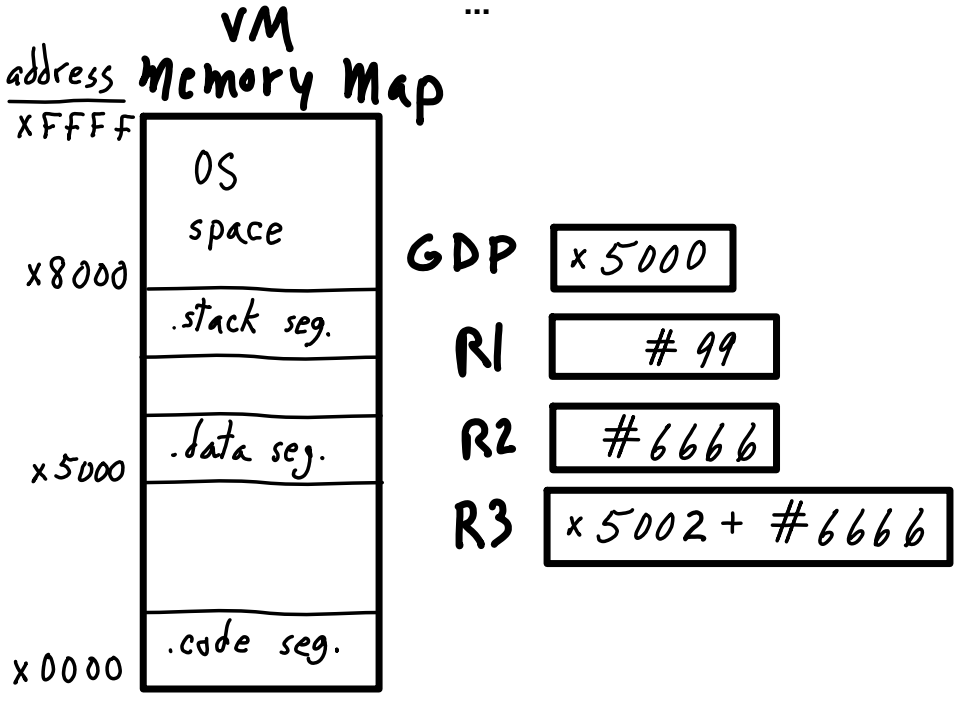
```
------- C code ----------------------
int A[1000];
int main(void) {
   A[6666] = 99;   <- array access.
   return( 0 );
}

------ assembly code: --------------
.DATA SEGMENT
 const_a:  .FILL #99        ;-- off = 0
 const_b:  .FILL #6666      ;-- off = 1
 A:        .BLKW #1000      ;-- off = 2
 const_c:  .FILL x0         ;-- off = 1002

.CODE SEGMENT
   main:
        LDR R1,  GDP,  #0
        LDR R2,  GDP,  #1
        ADD R3,  GDP,  #2
        ADD R3,  R3,  R2
        STR  R1,  R3,   #0
        ...
```

16-bit Page Table Entry: **[F#, (other), M, P]**
--- F#, 4-bit frame number
other PTE bits:
--- V (1: valid)
--- A (1: accessed)
--- D (1: dirty/modified)
--- L (1: locked in memory)
--- C (1: cache-able)
--- W (1: write permission)
--- U (1: user mode permission)
--- PID, 3-bit process ID
--- M (1: mapped, i.e., allocated)
--- P (1: page present in memory)

If P=0, the upper 8 bits are a disk page number.



Q. Assume the user's segments are as follows:
--- code: 3k instructions
--- data: 1003 words
--- stack: 1k words

each is less than a 4k page.
So, 1 page for each segment.

The system does not allocate pages that would be empty. In the page table at right, fill in the M bit, (0: not-mapped, 1: mapped).
Only the user's portion of the PT is shown; the OS portion is irrelevant. Explain each non-zero entry.

By the VM map,
code is in P# = 0,
data is in P# = 5,
stack is in P# = 7

These are the mapped pages (mapped to disk or to frames).

PT (M: mapped)

Page #	frame #	M	
0	1	1	
1		0	
2		0	
3		0	
4		0	
5	2	1	
6		0	
7	3	1	

frames avail: 1-E
choose 1, 2, 3

Q. All the user's pages have been accessed and are in memory. In the PT above, for each mapped page, put in a frame number. Choose any physical frame that is available. Physical frames xF and x0 are reserved for OS use.

frames 0 and F not available: 1-E are.
Choose frames 1, 2, 3 for pages 0, 5, 7.

TLB

Page #	frame #	V	
0	1	1	
5	2	1	
7	3	1	
		0	

Every page ref'd: PTE for each in TLB
FA: order does not matter

Q. No other process has evicted any of the process's TLB entries. The TLB is fully-associate. Given your PT entries above, fill in the TLB above. When the STR instruction executes, will there be a TLB miss? Why or why not?

No TLB miss for instr. fetch: P# = 0 has its PTE in TLB.
Data access is for address in R3 = x5002 + *6666 — what is this in hex?

*6666 = 6(1024) + e < 7k
= 6k + e
e ≈ 500 < k

7k → 2^10
111 0000000000
x1 C 0 0

x5002
+x1C00
x6C02

⇒ Page # 6 → un-mapped page, PTE not present in TLB ⇒ TLB miss

Q. The TLB-miss exception handler will send a "SIGSEG" signal to a process that tries to reference an un-mapped page. This will cause the process to abort, issuing the error, "segmentation fault." Why will the process above be aborted?

The STR instruction references an address in an un-mapped page (#6).
Handler will abort the process.

Continuing with the above example, the process "forks" a child process by duplicating itself: all memory pages are copied, and a new PT is allocated to the child. For instance, suppose the above processes code included,

```
proc = fork();
if (proc == 0 ) { exec( "foo" ); }
else            { wait( proc ); }
```

The parent's code pages are copied to the child, and both child and parent execute the same code after the fork(), except the OS gives the child a return value of 0 for fork(), while the parent gets the child's PID. The child here then overwrites its code segment with code from file "foo" and executes that instead.

However, to save time, the OS can create the child process without allocating and copying: it copies the parent's PT to the child process. This is called "lazy" forking. If and when a page is written by the child, then the OS copies the page and updates the child's PT. For instance, exec() here would cause the code page to be copied, and then overwritten. The OS detects the need to copy by initializing all child PT entries w/ W = 0. The exception caused by attempting to write will allow the OS to do the lazy copying.

Q. As described above, the process forks a child, and the OS uses "lazy" forking. Show the child's initial PT below. The child's PT is 0010 (x2), the parent's is 0001 (x1).

initially, child's PT is copy of parent's, points to same frames

child's PT

Page #	frame #	PID	W
0	1	2	0
1			
2			
3			
4			
5	2	2	0
6			
7	3	2	0

TLB

Page #	frame #	PID	
0	1	1	
5	2	1	
7	3	1	
0	4	2	

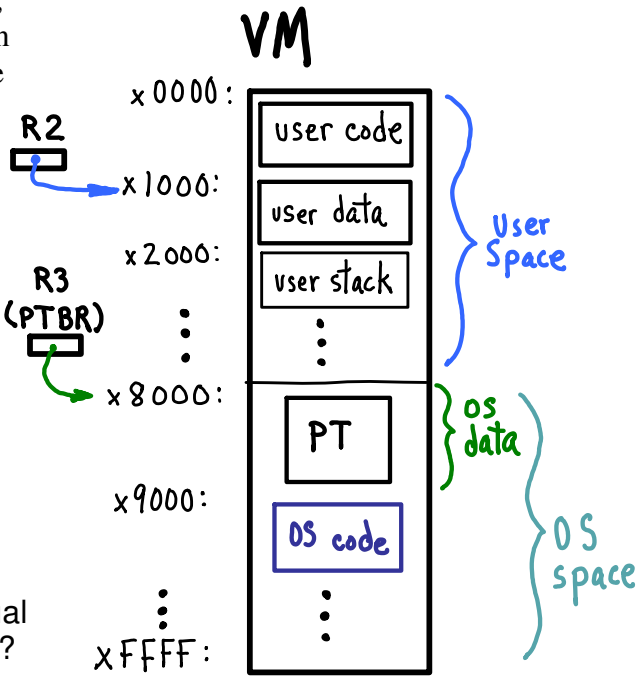
frames 4-E are available for copy.

exec() overwrites page # 0, causing copy to new frame before writing. Child's PT is correspondingly updated.

Q. Assume the TLB is as you showed in the above questions. The child does 'exec("foo")'. Show the TLB content just after exec() completes. Pick any available frame to use as the frame for the copied page. Assuming no other pages are in memory other than shown in the TLB, would the OS choose a frame for this copy that would cause a page fault (either to write back, or later read in)? Why or why not?

There are lots of free frames available (any frame except 0, 1, 2, 3, F). No need to kick out some page and pay the cost of writing out a modified page, or kick out any resident page and later cause a page miss.

The processor fetches a user program instruction and tries to execute it, PC = x0022. The VM map is shown at right, physical memory is shown below. The PT is in virtual page x8, which is mapped to physical frame xA. Note, for clarity's sake, PT entries include page numbers.

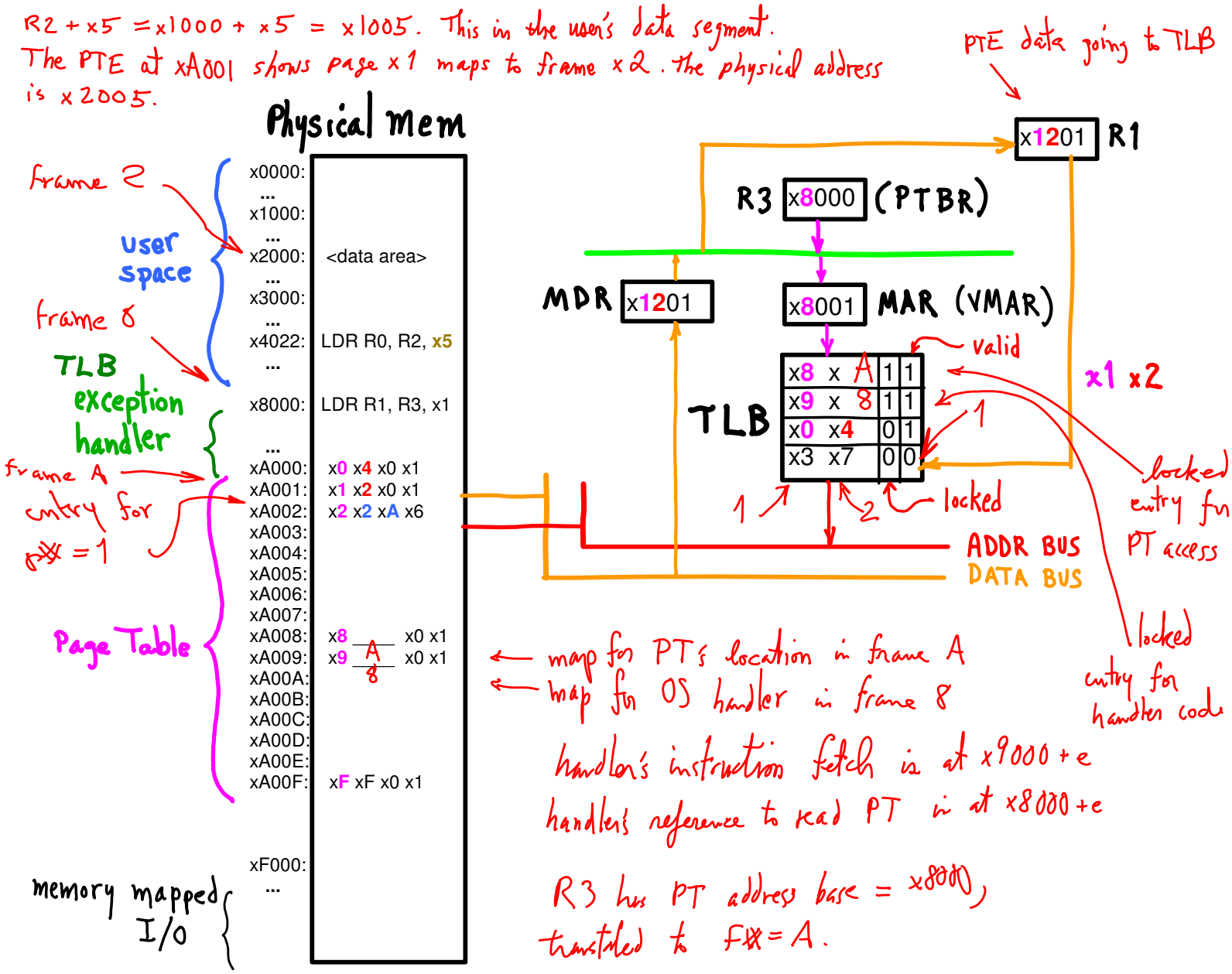


Q. What is the instruction's virtual page number? What is the physical address of its PTE? To which physical frame is that page mapped? What is the instruction's physical memory location? What is the instruction?

The instruction's P# = x0. Its PTE entry is in the PT at offset = 0, at physical address xA000. Page x0 is mapped to frame x4. The instruction is at physical address x4022. It is LDR R0, R2, x5.

Q. The instruction at x4022 is making a memory reference to address R2 + x5. The content of R2 is shown at right. What virtual address is being referenced? What segment is that reference to? What does physical address does that virtual address map to?

R2 + x5 = x1000 + x5 = x1005. This in the user's data segment. The PTE at xA001 shows page x1 maps to frame x2. the physical address is x2005.



Q. Assuming the TLB contents are as shown above when the instruction is executed, why does this memory reference cause a TLB miss? Show the PTE content that the TLB-miss exception handler will read to update the TLB.

The PTE for virtual page x1 is not in the TLB. The address cannot be translated. The PTE is at xA001 and is [x1 x2 x0 x1].

Q. The TLB-miss causes an instruction fetch from the OS code segment. Which virtual page is the OS code in, according to the VM map above? Assume the first instruction in the OS code segment is the first instruction of the TBL-miss handler. The jump to the TLB-miss handler loads what virtual address to the PC? Looking at physical memory, from which physical address is the instruction fetched? What is the instruction? Complete the TLB entry and the PTE above for this instruction fetch translation.

OS code is in Page x9. The first instruction is at virtual address x9000, which would be loaded to the PC. The physical address is x8000. The instruction there is LDR R1, R3, x1. The entries to fill in are [x9 x8 x0 x1].

Q. The instruction, "LDR R1, R3, x1" is the first instruction of the TLB-miss handler. R3 is the Page Table Base Register (PTBR), and is pointing to the virtual address of the current PT. What virtual address does this instruction read from? Looking at physical memory, what is the base address of the PT? Fill in the TLB entry and the PTE for this translation. What physical address is read from by this instruction? What content is at that physical address?

The virtual address referenced is x8001. The PT is physically at xA000. Virtual page x8 maps to physical frame xA. The entries are [x8 xA x0 x1]. The physical address referenced is xA001. The content is the PTE for page x1.

Q. The MDR is loaded by execution of that instruction, and then R1 is loaded, as shown above. The content of R1 is a PTE that the TLB-miss handler needs to load into the TLB. Assume subsequent handler instructions write R1 into the TLB as shown. The TLB-miss handler exits by restarting the instruction that missed: The PC is set to x0022. Now that the TLB has been updated, will the instruction miss in the TLB on instruction fetch? Why or why not? Will the instruction suffer a TLB miss for data access? Why or why not?

The instruction fetch will be translated by the TLB entry [x0 x4 x0 x1], because it is still in the TLB. The data access will be translated because the TLB entry [x3 x7 x0 x0] has been replaced with [x1 x2 x0 x1]. So, there won't be a TLB miss this time.

Q. Suppose the OS switches processes. Suppose the new process's PT is in virtual page xD and in physical frame xE. Can the OS simply reload the PTBR's content to xD000 and continue? That is, if the OS loads the PTBR using the instruction "ADD R3, R7, #0" where R7 contains xD000, will the next OS instruction after that be fetched correctly? The next few instruction will presumably cause the OS to jump to new process's code segment.

Because the OS part of every PT is identical, all OS mappings are preserved on a context switch. There will be TLB misses, but the handler code is still mapped correctly by the TLB's locked entries.

Machine M has a split L1 cache and unified L2. Running program P, L1's miss rate is $MR_1 = 1/16$ (combined) and L2's miss rate is $MR_2 = 1/4$. Loads and stores account for 1/3 of instructions executed. All misses (loads, stores, and instruction fetches) cause CPU stalls. If both fetch and data access hit in L1, instruction execution time is L1's hit time. L1's fetch and data accesses run in parallel, but L2 processes requests from both serially. The number of instructions executed is n, M's clock rate is CR, L1's hit time is T_1, L2's is T_2 and memory's access time is T_m.

Q. In total, how many memory accesses occur (fetches and data R/W)? State the result in terms of the parameters given. Of these, how many hit in L1? How many miss L1?

n instr → n fetches $n/3$ in LD/ST $n(1+1/3)$ mem accesses $= K$

$K\,1/16$ = # misses $K\,15/16$ = # hits

Q. If all miss penalties were 0, what would be the total execution time? How many clock cycles?

T_1/instr execution $T_1 \cdot CR$ = cycles per instr

$n \cdot T_1 \cdot CR$ = total cycles

Q. In total, many memory accesses (fetches and data R/W) miss in L2? How much main memory access stall time results? How many stall cycles?

→ ((K/16 L1 misses)(1/4) = # miss in L2) $\cdot T_m$ = stall time,

→ $\frac{K}{16}(\frac{1}{4}) T_m \cdot CR$ stall cycles.

Q. For an L1 miss that hits in L2, how much stall time is attributable to the L2 access? How many cycles?

Time to wait for L2 hit = T_2, $T_2 \cdot CR$ stall cycles

Q. What is P's total running time? How many cycles? What is M's average CPI?

Total time = (n instr)(T_1, L1 hit time) + $n \cdot MR_1 (1 - MR_2)$(T_2, L2 hit time)

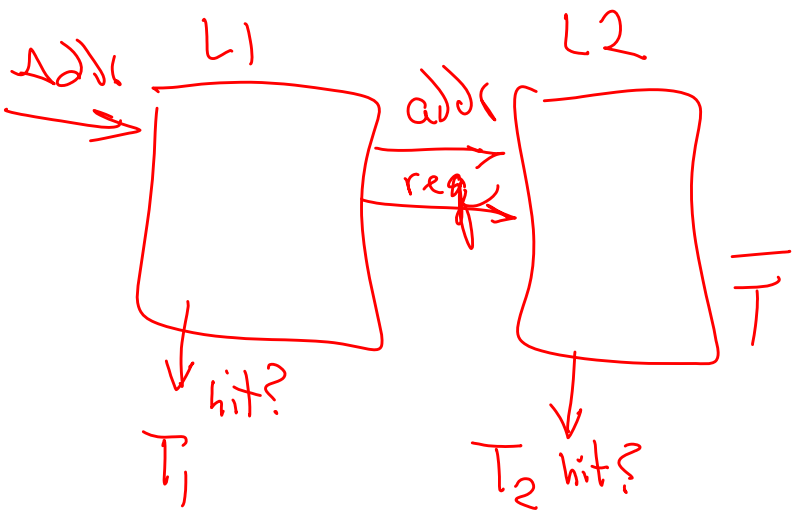
+ $n \cdot MR_1 (MR_2 ($(T_2, L2 hit time + T_m, L2 miss penalty$))$

$= n\left(T_1 + (1/16)(3/4) T_2 + (1/16)(1/4)(T_2 + T_m) \right) = \#$

cycles $= \# \cdot CR$

CPI $= \frac{1}{n} \# \cdot CR = \left(T_1 + \frac{3}{64} T_2 + \frac{1}{64}(T_2 + T_m)\right) \cdot CR$

miss L1, Hit in L2



$T = T_1 + T_2 \approx 0 = T_1$

$\overline{T} = T_{hit} + T_{miss}$

$= HR(T_{hit}) + (1-HR)T_{miss}$

$(1-HR)(T_{hit} + T_m)$

$= HR(T_{hit}) + (1-HR)T_{hit} + (1-HR)T_m$

$= T_{hit} + (1-HR)T_m$

miss penalty

Q. Suppose memory access time does not improve while processor improvements double the clock rate. Assume L1 and L2 access times are also halved. Show an expression for the new CPI in terms of the old clock rate. What effect does this have on average CPI?

$$CPI_{new} = \left(\frac{T_1}{2} + \frac{3}{64}\left(\frac{T_2}{2}\right) + \frac{1}{64}\left(\frac{T_2}{2} + T_m\right)\right)(2 \cdot CR)$$

$$= \left(T_1 + \frac{3}{64}T_2 + \frac{1}{64}(T_2 + 2T_m)\right) \cdot CR$$

adds $T_m CR_{old}$ to CPI, makes CPI worse.

Q. Show the speed-up of the new processor relative to the unimproved version.

$$S = T_{old}/T_{new} = \frac{CPI_{old}\,(1/CR_{old})}{CPI_{new}\,(1/CR_{new})} = \frac{\left[T_1 + \frac{3}{64}T_2 + \frac{1}{64}(T_2 + T_m)\right] CR\,(1/CR)}{\left[T_1 + \frac{3}{64}T_2 + \frac{1}{64}(T_2 + 2T_m)\right] CR\,(1/2\,CR)}$$

$$=$$

Q. Assuming $T_2 = 10\,T_1$ and $T_m = 10\,T_2$, what qualitatively is the effect of the improvement? What does Amdahl's Law suggest in regard to which aspect of performance should be improved? What if CR keeps doubling every two years?

$$T_m = 100\,T_1$$

$$S' = \frac{T_1 + \frac{3}{64}(10\,T_1) + \frac{1}{64}(10\,T_1 + 100\,T_1)}{T_1 + \frac{3}{64}(10\,T_1) + \frac{1}{64}(10\,T_1 + 200\,T_1)}(2) = 2\left(\frac{1 + 30/64 + 110/64}{1 + 30/64 + 210/64}\right)$$

$$\approx \left(\frac{1 + 1/2 + 2}{1 + 1/2 + (2+2)}\right)2$$

The memory speed is reducing the effect of CR and cache speedup. If cache and CR performance improves more, the effect is that memory speed is dominating performance increase: $T_m \to k\,T_m$

$$S' \approx \frac{3\frac{1}{2}}{3\frac{1}{2} + k}(k) \quad \text{as } k \to \infty \Rightarrow 3\frac{1}{2} \text{ is Amdahl's result.}$$

Need to improve memory performance to allow CR improvement to have an impact.

Q. Comment very briefly (~5 words each) on the performance tradeoffs of each cache feature.

larger cache blocks

pro: better at exploiting spatial locality, efficient pipelined transfer
con: more collisions, block transfer overhead (latency)

more cache levels

pro: lower overall miss penalty
con: complexity (area, energy, design)

more associativity

pro: lower miss rate, fewer collisions
con: complexity, LRU algorithms, not SRAM

virtual tagging

pro: faster hit time
con: synonyms, aliases ===> complexity

larger total cache size w/ larger blocks

pro: better exploitation of spatial locality
con: block transfer overhead, area

larger total cache size w/ smaller blocks

pro: less collisions, less transfer latency
con: less efficient transfers, area

write buffering w/ a searchable buffer

pro: no stalls for writes
con: complexity

write-back

pro: less memory traffic
con: LRU overhead, memory coherence

larger pages

pro: more efficient page transfers, better TLB coverage, more spatial locality
con: more latency, more internal fragmentation

PID fields in caches and TLBs

pro: no flushing, less misses, less latency for process start
con: lower hit rate, complexity