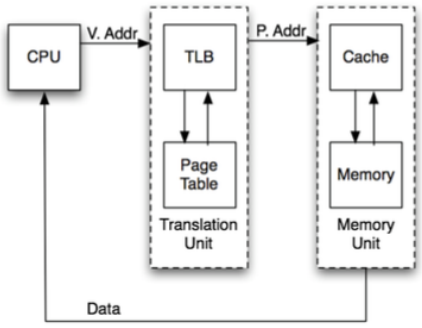
Virtual Memory

Virtual Address (VA): What your program uses

Virtual Page Number (VPN)	Page Offset

Physical Address(PA): What actually determines where in memory to go

Physical Page Number (PPN)	Page Offset



Translation Unit

TLB

V	Tag =VPN	*Page Table Entry*		
		D	Permission Bits	PPN
…	…	…	…	…

Page Table

VPN	V	D	Permission	PPN
0				
1				
2				
…				
n				

Memory Unit

Cache

V	Tag	Data

Memory

0x0…	…	…	…	0xF…F

Protection Fault: The page table entry for a virtual page has permission bits that prohibit the requested operation

Page Fault: The page table entry for a virtual page has its valid bit set to false. The entry is not in memory.

1) The specs for a MIPS machine's memory system that has one level of cache and virtual memory are:

- 1MiB of Physical Address Space
- 4GiB of Virtual Address Space
- 4KiB page size
- 16KiB 8-way set-associative write-through cache, LRU replacement
- 1KiB Cache Block Size
- 2-entry TLB, LRU replacement

The following code is run on the system, which has no other users and process switching turned off. Assume that the page table can hold 11 amounts of pages. **To make things "easier," pretend that the compiled binary for the following program does not require a page to be implemented for questions (a-g).** `malloc` should return address 0x100000 (you should be "block-aligned" and "page-aligned."

```
#define NUM_INTS 8192                          // This many ints...
int *A = (int *)malloc(NUM_INTS * sizeof(int));
int i, total = 0;
for(i = 0; i < NUM_INTS; i += 128) A[i] = i;
for(i = 0; i < NUM_INTS; i += 128) total += A[i];  // SPECIAL
```

a) What is the T:I:O bit breakup for the cache (assuming byte addressing)? T:_9_I:_1_O:_10_

b) What is the VPN : PO bit breakup for VM (assuming byte addressing)? _20_:_12_

c) What is the PPN : PO bit breakup for PM (assuming byte addressing)? _8_:_12_

d) How many page faults can occur in the **worst-case scenario** before the "SPECIAL" for loop?

Well, in the worst case scenario, the TLB is empty/flushed prior to the process and the page table is empty. Ignoring any possibility that the compiled binary code takes up any page, we know that our array takes up 8192 * 4 bytes = 2^{15} bytes, which is equivalent to 2^3 *2^{12} bytes, which is equal to 8 pages. Thus, we would have 8 page faults in the worst case.

For the following questions, only consider the line marked “SPECIAL”. However, the current state of the process should be whatever is expected after the prior `for` loop. Your answer can be a fraction:

e) Calculate the hit percentage for the cache

1/2 = 50%. Our cache is 2^{14}B 8-way set associative. We know that A is 8192 * 4 bytes = 2^{15} bytes, which is the equivalent to 2 cache sizes. We also know that we are block and page aligned. We read from A[i] once. If we look at accesses in the first block, we can see that there is a single miss. We then move to i = 128, which is 512 bytes away and proceed to hit. Of these two memory accesses, there is one hit. This rate repeats for all blocks.

f) Calculate the hit percentage for the TLB

7/8 = 87.5%. We know that one page is 2^{12} Bytes. As seen in c), we are incrementing by 512 bytes, which means that there are $2^{12}/2^9=2^3$ page accesses per page. Of these accesses, we miss once to load in the page from disk to memory and then proceed to hit the other 7 times.

g) Calculate the page hit percentage for the page table

100% (look at the for loop before SPECIAL).

h) How would having the compiled binary for this program take up one page-sized amounts of data affect our problems for the "SPECIAL" loop? What about for part d?

We can potentially have one TLB page designated for code and the other for all memory accesses of our array (our TLB has 2 entries). For part d, we'd have one extra page fault however because our page table would not originally contain the page where our code is located.

i) What would happen to our TLB performance if **`malloc`** perhaps returned an address that was not "page aligned" and instead was for instance, 0x0FFFFC? What about our page fault count for part d?

Malloc would not have returned a page-aligned address. Our array is 8 page sizes long. If it was not page-aligned, we could potentially address it in such a way that it spanned 9 different pages in virtual memory, and in turn required three different mappings to physical

memory. This could reduce our hit percentage for the TLB, and also increase our page fault count in part d.

I/O

3)

Operation	Definition	Pro	Con
Polling	Forces the hardware to wait on ready bit (alternatively, if timing of device is known – the ready bit can be polled at the frequency of the device). It basically means manually checking the ready bit regularly.	PRO: -easy to write -poll handler does not have excessively high overhead -deterministic -doesn't require additional hardware	Infeasible on hardware with fast transfer rates that is actually rarely ready (e.g. Ethernet card receiver)
Interrupts	Hardware fires an "exception" when it becomes ready. CPU changes $PC to execute code in the interrupt handler when this occurs.	PRO: -Necessary for fast devices that are rarely ready. Good for: Fast devices - Hard drives, Network cards	-nondeterministic when interrupt occurs -interrupt handler has some overhead (saves all registers), meaning polling can actually be faster for slow, often ready devices such as mice

4) Memory Mapped I/O: Certain memory addresses correspond to registers in I/O devices and not normal memory.

0xFFFF0000 – Receiver Control:
Lowest two bits are interrupt enable bit and ready bit.

0xFFFF0004 – Receiver Data:
Received data stored at lowest byte.

0xFFFF0008 – Transmitter Control
Lowest two bits are interrupt enable bit and ready bit.

0xFFFF000C – Transmitter Data
Transmitted data stored at lowest byte.

Write MIPS code to read a byte from the receiver and immediately send it to the transmitter.

```
                lui $t0 0xffff
receive_wait: #poll on ready of receiver
              lw $t1 0($t0)
              andi $t1 $t1 1
              beq $t1 $zero receive_wait
              lb $t2 4$t0) #load data
```

```
transmit_wait: #poll on ready of transmitter
               lw $t1 8($t0)
               andi $t1 $t1 1
               beq $t1 $zero transmit_wait
               #write to transmitter
               sb $t2 12($t0)
```