Mid-term Exam, 2012 spring

NAME _____

READ THIS, IT IS IMPORTANT

Exam questions are not easy to write. Their purpose is to guage the degree to which you have absorbed the lessons of the material covered and are able to put them to use. Testing one's ability to calculate numerical values or other mechanical problem solving does not much further that purpose; unfortunately, other ways of asking questions are hard to invent. Here's the point: Be aware of the goal of each question and try to demonstrate your relevant knowledge. Be aware that the problems presented may not be well designed to elicit your understanding. Explain your thinking, adding whatever additions you feel best allow me to understand what you know and understand. If the exam is not allowing you to do that, you may even insert comments and explanations that are off the given topic. Some questions likely have flaws: poor or incorrect wording, mistaken values, missing information, contradictions, or presentation that is confusing or leads one's thinking astray. Comment on such difficulties if you encounter them. Saying something like, "This question is stupid," is a good start. Anything that helps me learn what I need to know is valuable to both of us.

Adapt your test taking strategy to the exam: I try to write long exams with a wide range of difficulty. I do not expect them to be completed. Instead, I give you a selection to choose from. Do the easier ones first. If you still have time, come back to the harder ones. You can choose to do only part of a question, especially if answering is taking a lot of time. Partial and extra credit will be given liberally. Obviously, longer and harder problems get proportionally more credit than shorter and easier ones.

Question. More than fundamental principles, historical accident and precedent have determined the nature of computer systems as we know them today. What forces have acted in the past or are currently affecting future directions in system designs and capabilities? How are these forces changing? What new directions might become attractive that were not before? Here is a list of key terms to work with:

Heirarchical design, standard interfaces, abstractions, monopolization
Commonality, generality, customization, reconfigurabilty, design complexity
Productivity, performance, cost (acquisition and operation), energy, power
Yield, learning curve, economy of scale, fixed costs (mask sets, silicon foundry capability)
Printing versus assembly, copying, digital representation
Price versus time curves, latency versus bandwidth
Marketability, supporting legacy program code, application portability
Large-scale infrastructure (communications, cloud computing)
Personal devices, embedded devices, functionality reward versus expense
Miniaturization, device scaling, switching energy, leakage current, heat density
Switching speed, voltage and frequency scaling,
Virtual machines, simulation, portability, migration

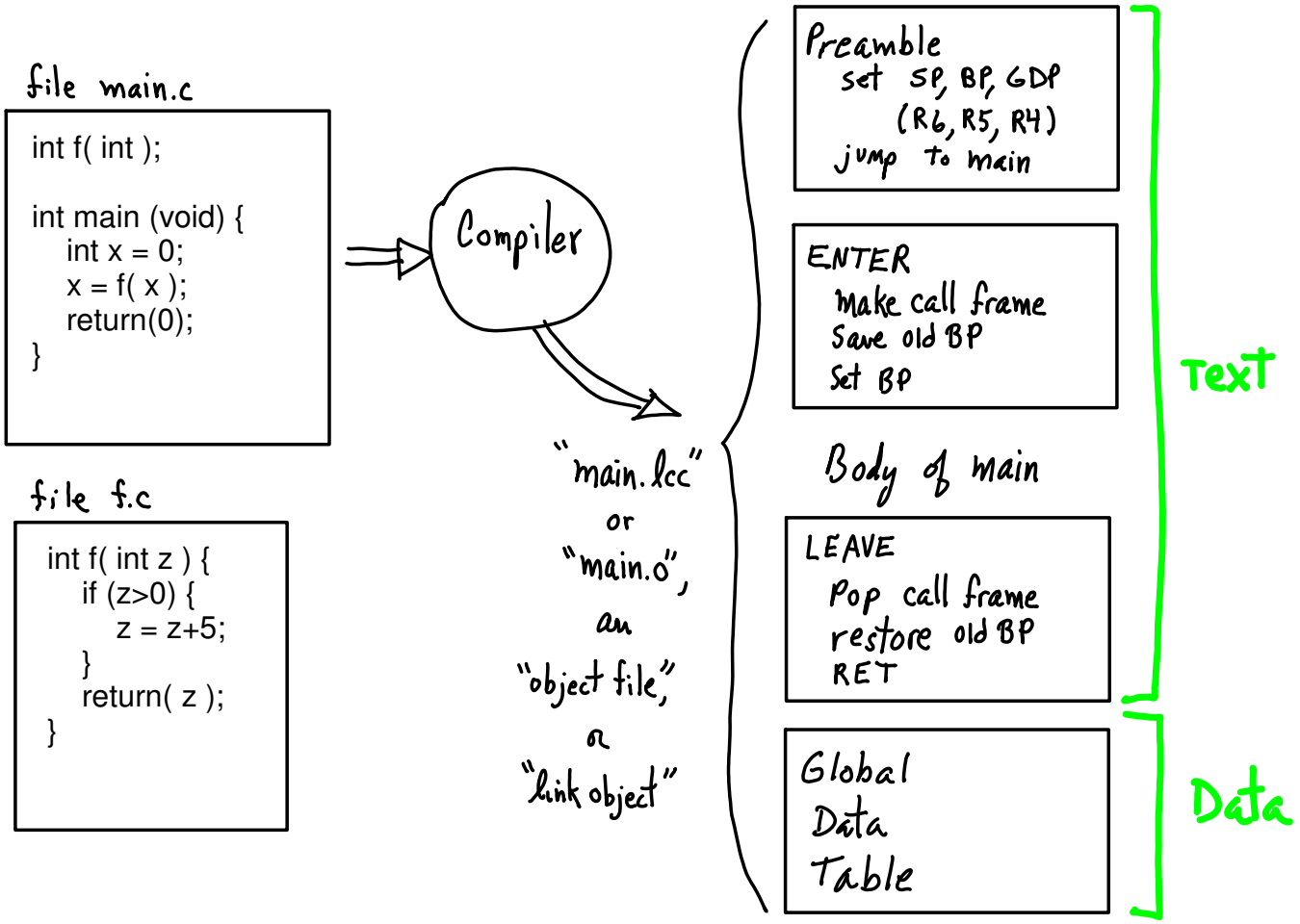
There are too many "right" answers to state a single response as the correct one for this question. We can start by looking to the fact that legacy code has been a very big influence on what designs are accepted in the market. This is generally the issue of standardized, abstract interfaces, which the PC is an example. To the extent these exist, markets and products grow around them, the cost of products goes down, and the direction of development is driven by these two effects. Start up costs of design overhead for chip manufacture dictates that chip design effort will go into the common denominator. However, the emergence of personal devices, with non-standardized components, and their need for specialization to accomodate adequate processing power at low energy costs, means that an opening has emerged for new and different architectures to develop.

This is also an effect of virtualization. Because server farms have to be heterogeneous to some degree, and virtual machines are commonly used, the need for standardization at the micro-architecture level is less important as legacy code and code written for legacy architectures can continue to be supported in an environment which is not uniform. This also suggests an opening up of paths of development for computer architectures.

Another influence on future developments in architecture come from the increased importance of basic physical limitations on computing hardware. As the improvement rate for performance and the cost/performance ratio has been flattening, power density and clocking rates have leveled off as well. The improvement in performance is now dependent on the clever use of parallelism. While we can expect numbers of cores to keep increasing per chip, this cannot continue indefinitely. In particular, making use of multi-core and general parallelism is constrained by 1) the complexity of writing correct, highly parallel programs that achieve performance gains, and 2) bottlenecks such as the memory bandwidth disparity between processors and memory devices. In particular, the latter is acerbated by the increase in on-chip parallelism.

Finally, markets for computing devices are broadening. For instance, network switches are alone a significant influence on developments in computing machinery. Embedded devices are ubiquitous and heterogeneous. This suggests a fracturing into many avenues of development, each with different parameters. Watching the changes driven by the influences of economy of commonality and their contradictates from the influences of market disparity will be an interesting story to watch unfold.

The LC3 represents a standard, basic CPU. Compiling C code for the LC3 produces simple assembly language code, which is translated to LC3 machine code. Separate compilation produces files that must be linked. Linking can be done at the time the load object file is produced, and also at runtime when the loader loads the program to memory. Certain standard pieces of code are inserted: PREAMBLE, ENTER, and LEAVE. These do standard actions to conform to C or OS protocols. Below is shown a main() and a schematic of its translation to an assembly code link object. OS convention maps programs to memory by segments: code, data, and stack. Here, the data segment immediately follows the code segment.



The C code for f() is separately translated similarly. Addresses of functions and data constants (such as "0" above) are located in the Global Data Table, and the preamble sets the GDP (R4) to point to it. The two object files are linked to produce the load object file. Here is a command that does linking:

```
lcc main.lcc f.lcc
```

The output is "main.obj", an LC3 load object module containing the machine code generated from both files. Linking merges the two global data tables. Link objects typically include symbol tables for all symbols referenced so that references in one object to a location in another object can be resolved.

Load editing adjusts addresses as needed to correspond to the memory locations where the executable load object's segments are loaded at runtime. Load objects contain tables indicating all addresses and offset fields that must be relocated. The LC3 does not have virtual memory; so, all addresses are physical.

Q. On the diagram at right, draw the memory map at runtime for the program above. Assume f() has been called but has not returned.

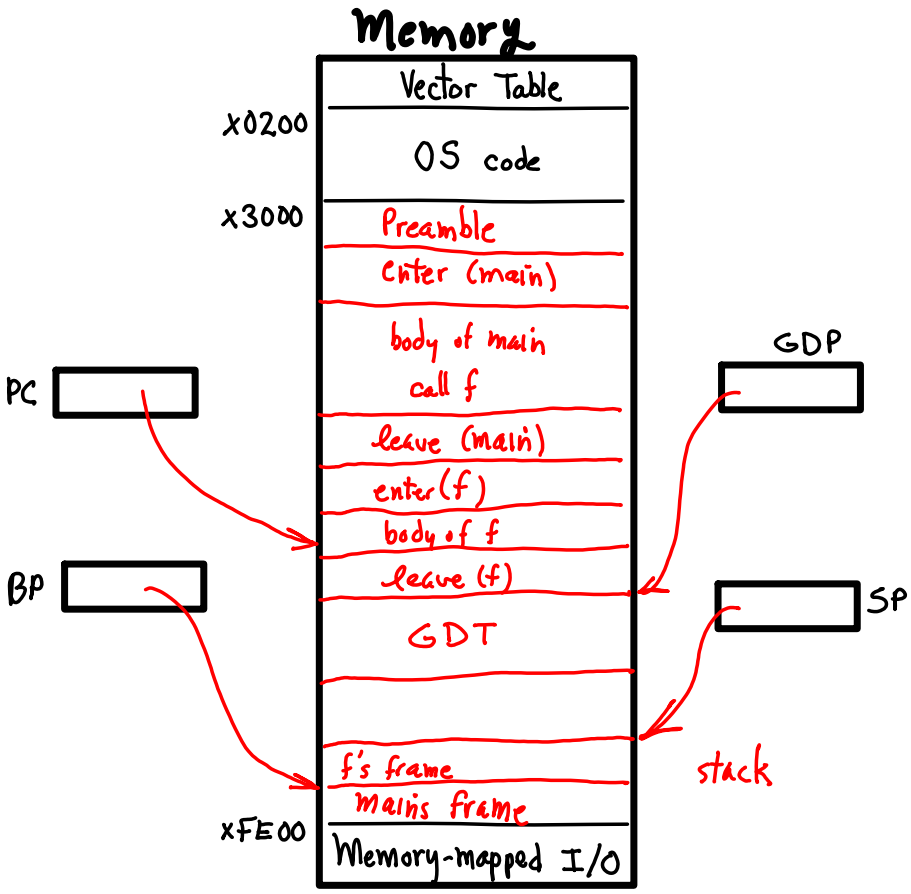
Indicate locations of the following:
--- preamble
--- ENTER and LEAVE code blocks
--- the global data table (constants and function pointer variables)
--- call frames (arguments, local variables, return value, and return address)

Assumptions:
--- this is a user program
--- the preamble sets SP = xFE00
--- OS area starts at x0200
--- user area starts at x3000
--- memory-mapped I/O area starts at xFE00.

Draw arrows to show where the PC, SP, BP, and GDP point.

*Make any reasonable approximation for the exact layout of the global data table, call frames, and the location of the BP.

*Explain any details or assumptions.



Q. Consider the function f(). It has an "if" statement which generates the code at right. The ADD is part of the evaluation of the if's conditional expression. R4 is the GDP, and the third entry in the gobal data table is,

```
.FILL   lc3_L2_f
```

What is the code between the BR and the L6 label doing? Is this the IF's code that is executed when the conditional expression evaluates to TRUE or to FALSE?

*Note that "L6" and "lc3_L2_f" are compiler-generated labels and there are no branches or jumps after lc3_L2_f except for RET.

*Recall that the GDP is R4.

```
...
ADD R7, R7, R3
BRn L6
ADD R7, R4, #2
LDR R7, R7, #0
JMP R7
L6
...
lc3_L2_f
...
RET
```

The if-statement's conditional expression is "if(z >0)". The ADD is doing (R7 + R3). We can guess that one register has the variable z in it and the other has 0 in it. So, (z ≥0) is true if the BRn is not taken (seems to be a bug, as the conditional would be ≥, not >). So, the next bit of code is doing z = z+5. However, ADD R7, R4, #2 is loading the address of GDT's 3rd entry into R7. The next LDR loads R7 with the address of the code at label lc3_L2_f, and then JMP R7 jumps there. So, the code at L6 is skipped. That results in RET. So perhaps this is the False branch instead.

Q. During linking and/or load editing, would the offset value of the BR instruction need to be adjusted? What about the global data ".FILL   lc3_L2_f"? (Recall that load editing takes place after assembly, while for LC3's lcc, linking takes place using the assembly language output of the compiler.) Explain.

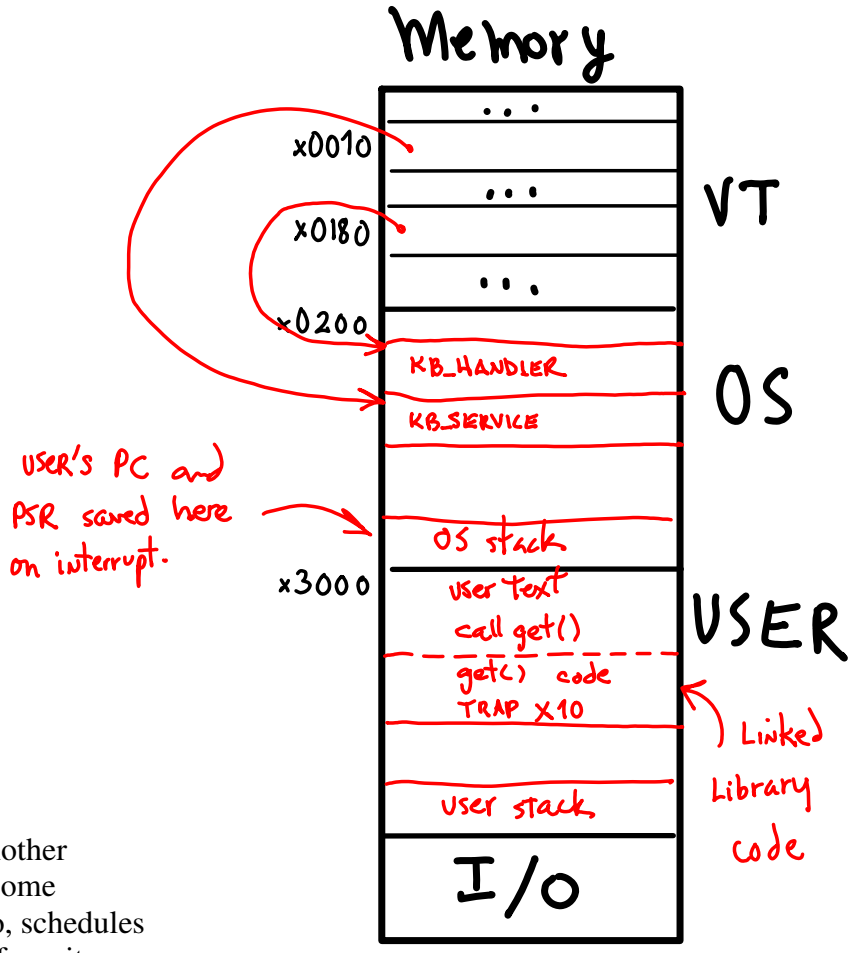
The branch uses PC-relative addressing and works correctly w/o recalculating the offset value for BR. Link-time or load-time relocation has no effect on the offset's value.

The value assigned to the label lc3_L2_f was determined at link time and is an absolute address. At load-time, this value may need adjustment if the code is not loaded where the linker assumed it would be.

Joe is working on the OS for the LC3, and has written a device driver for the keyboard. He has also written a service routine that user programs can call to get keyboard input data. The service routine is envoked with "TRAP x10". The interrupt vector for the keyboard is x0180. The service routine is called by a library function "get()" that is linked with the user's C code. Joe's code is shown below (mostly just the comments):

```
KB_INIT
    ;--- write VT for KB_SERVICE
    ;--- write VT for KB_HANDLER
    ;--- enable keyboard interrupts
        RET
KB_HANDLER
    ;--- get keyboard data
    ;--- insert in buffer
    ;--- set Ready flag variable
        TRAP x12
    ;--- enable keyboard interrupts
        IRT
KB_SERVICE
    ;--- while (1)
    ;---    check Ready flag
    ;---    if Ready
    ;---        get data from buffer
    ;---        put data in R0 as return value
               RET
    ;---    else
              TRAP x11
    ;--- end-while
```

*TRAP x11 jumps to a scheduler that starts another program running. TRAP x12 checks whether some program is waiting for keyboard data, and if so, schedules it to be reloaded and run as if it was returning from its TRAP x11 call.



Q. In the diagram above, show memory areas for the OS and user stacks. Indicate where in memory the KB_HANDLER and KB_SERVICE code is located. Add arrows from the VT to indicate where their VT entries point. Show where the get() routine is in memory after linking. Show where a call to get() would be. Indicate where the "TRAP x10" instruction would be in memory. Indicate where the user's PC and PSR are saved when an interrupt occurs.

Q. Suppose the LC3 were altered so that TRAP executed just like an interrupt: the mode is changed to kernel mode, the SP is switched if the mode was user mode, the PC and PSR are pushed onto the stack, and then a jump is made through the corresponding entry in the VT. What little bit of Joe's code needs to be fixed?

The TRAP service routine, KB_SERVICE, exits w/ RET. This will need to be changed to IRT. Traps do not use stack-based arguments, usually, so even though stacks are swapped, this should not affect functionality.

Q. Suppose also that memory protection is implemented so that user code cannot access OS or I/O memory for read, write, or execute. Could OS service routines return values to user programs larger than would fit into a few registers? If it is possible, suggest how it could be done. If it is not possible, suggest an alteration to the LC3 that would accomodate passing large return values.

Because kernel mode has full access to memory, the service routines have permission to write into user space. User code will need to provide a buffer for I/O in its own space, on the stack or in the Data area, and pass its address as an argument to the service routine.

Q. The "get()" routine is written in assembly language, but conforms to the C conventions for calling a function: it gets its arguments from the stack and sends its return value back to the caller via the stack, repositioning the SP and BP before returning. However, when it calls TRAP x10, it does not use the C stack discipline, nor does the TRAP x10 code. For instance, the TRAP x10 return value is sent to get() via R0. However, TRAP x10 could have been written to conform with the C stack discipline: arguments and return values might be sent via the stack. C code could generate the usual C function call code but instead of using JSSR use TRAP for the jump. For instance,

```
#define KB_SERVICE 0x10
y = sys_call( KB_SERVICE);
```

might generate the appropriate code if we had the compiler recognize sys_call() as different from the usual function call and use TRAP where it would normally use JSSR. Do you see any problem with this scheme?

The user stack would need to get syscall()'s return value. If TRAP is like INT, it switches to kernel stack. The TRAP routine would have to put its return value on user's stack instead of its own kernel stack. But it wouldn't know where the user's SP points to: user SP is saved in a special register, savedUSP, which cannot be read. If TRAP does not work like INT, but acts like JSR (as it does in LC3), then this is not a problem.

Two load-store architecture machines, M1 and M2, have the following characteristics:

	M1	M2	
CR	4 GHz	3/2 GHz	
CPI-*alu*	3/2	1	ALU operations
CPI-*br*	5	3	Branches
CPI-*mem*	20	7	Loads and Stores

The CPIs are averaged over all instructions in each class. Given an execution trace of any program P, let a be the number of ALU operate instructions in P's trace. Let b be the number of branch instructions and m the number of memory access instructions in the trace. Assume a, b, and m are non-zero.

Q. Determine the characteristics of an execution trace in terms of a, b, and m, such that M2 out performs M1. What sort of a program might P be? That is, what kind of job would have these characteristics? Hint: use the basic processor performance equation and the speed-up formula of M1 versus M2 to find an expression relating a, b, and m.

$1 < S_{M2\text{-}M1} = \frac{T_{M1}}{T_{M2}} = \frac{n\,(a(3/2) + b(5) + m(20))(1/4\,GHz)}{n\,(a(1) + b(3) + m(7)(1/3/2\,GHz)} = \frac{3a/2 + 5b + 20m}{a + 3b + 7m}(3/8)$

$\Rightarrow\ 8\,(a + 3b + 7m) < 3(3a/2 + 5b + 20m)$

$7/2\,a + 9b < 4m \approx a + 2b < m$

memory ops predominate $\Rightarrow$ I/O bound job

Branching $< \frac{1}{2}m$ (no sort, e.g.)

Looks like less than 1 op per read, if all reads, or 2 per R-W pair: Streaming?

Two machines, M1 and M2, implement the same ISA:

Machine	CPI-A	CPI-B	CPI-C	CR
M1	1	2	4	1.6 GHz
M2	2	3	3	2.0 GHz

$\overline{CPI}_1 = (0.6(1) + 0.3(2) + 0.1(4)) = 1.6$

$\overline{CPI}_2 = (0.6(2) + 0.3(3) + 0.1(3)) = 2.4$

There are n instructions in trace T: 60% are class A, 30% are class B, and 10% are class C.

Q. What are the MIPS ratings for both machines for T? Recall MIPS = n / (time of execution) expressed in millions of instructions per second. What is the speed-up of M1 w.r.t. M2?

$T_1 = n\,\overline{CPI}_1\,(1/CR_1) = n(1.6)\,1/(1.6\,G)$ $\quad MIPS_1 = (n/10^6)/(T_1 = n(1.6)(\frac{1}{1.6})(1/10^9)) = 10^3$

$T_2 = n(2.4)(\frac{1}{2G})$ $\quad MIPS_2 = \frac{n}{10^6}/(n(2.4)(1/2G)) = 10^3/1.2$

$S_{1\text{-}2} = T_2/T_1 = \frac{MIPS_1}{MIPS_2} = 10^3/(10^3/1.2) = 1.2$

Q. Suppose we want to make the slower machine as fast as the faster machine, but we can only improve the execution time of one class of instruction. Find the minimum execution improvement necessary for one class of instruction. That is, for which machine and which class of instruction should we improve performance and by how much, such that the least improvement is needed? What is the new CPI for this class?

$$S_{2\text{-}1} = T_1/T_2 = \frac{n\,\overrightarrow{CPI_1}\,(1/CR_1)}{n\,CPI_2\,(1/CR_2)} = \frac{1.6\,(1/1.6)}{(0.6\,(2/x) + 0.3\,(3/y) + 0.1\,(4/3))(1/2)} = 2/(1.2/x + 0.6/y + 0.4/z)$$

where x, y, z are speedups for A, B, C. Biggest factor is $(1.2/x)$: speedup class A.

$$\Rightarrow \quad 2/(1.2/x + 0.6/1 + 0.4/1) \geq 1 \qquad \Rightarrow \quad 2 \geq (1.2/x + 0.9 + 0.3) = 1.2(1/x + 1)$$

$$\Rightarrow (2/1.2) - 1 \geq 1/x \quad \Rightarrow x \geq \frac{1.2}{0.8} = 3/2 \quad : \text{make M2's class A go 50\% faster}$$

Q. Suppose we can augment the ISA with MMX instructions that can operate on multiple, short, operands in parallel 10 times faster than without the MMX instructions. What sort of instruction mix would result in a speed-up of 2 for M1? For M2? That is, given x% of instructions can be executed as MMX operations, find x that gives a speed-up of 2.

We don't know which class MMX applies to, but suppose x% of all can be MMX.

$$S^1_{new\text{-}old} = T_{old}/T_{new} = \frac{W_{mmx}/V + W_{non}/V}{W_{mmx}/10V + W_{non}/V} = \frac{xW + (1-x)W}{xW/10 + (1-x)W} = \frac{1}{x/10 + (1-x)} = 2$$

$$\Rightarrow 10 = 2(10 - 9x) = 20 - 18x \Rightarrow x = 10/18 \approx 1/2 \quad \text{Same goes for } S^{1/2}_{new\text{-}old}.$$

Q. We wish to compare the performance of two machines, X, Y. We have 3 benchmark programs with data, b1, b2, b3. We have execution times for each benchmark on each machine and on a reference machine W, (T-x-1, T-x-2, T-x-3, T-y-1, T-y-2, T-y-3, T-w-1, T-w-2, T-w-3). Give an expression for each machine that sums up its performance relative to the reference machine. Show that the ratio of the two summary measures is a summary of relative speed-ups of the two machines. Suppose W is particularly slow for b2. Would your comparison change if we used reference machine W2 which runs b2 four times faster?

$$m_x = \sqrt[3]{\left(\frac{T\text{-}w\text{-}1}{T\text{-}x\text{-}1}\right)\left(\frac{T\text{-}w\text{-}2}{T\text{-}x\text{-}2}\right)\left(\frac{T\text{-}w\text{-}3}{T\text{-}x\text{-}3}\right)} \qquad \frac{m_x}{m_y} = \sqrt[3]{\frac{T\text{-}y\text{-}1 \cdot T\text{-}y\text{-}2\, T\text{-}y\text{-}3}{T\text{-}x\text{-}1 \cdot T\text{-}x\text{-}2 \cdot T\text{-}x\text{-}3}} = \sqrt[3]{S^{b_1}_{x\text{-}y} \cdot S^{b_2}_{x\text{-}y} \cdot S^{b_3}_{x\text{-}y}}$$

$$m_y = \sqrt[3]{\left(\frac{T\text{-}w\text{-}1}{T\text{-}y\text{-}1}\right)\left(\frac{T\text{-}w\text{-}2}{T\text{-}y\text{-}2}\right)\left(\frac{T\text{-}w\text{-}3}{T\text{-}y\text{-}3}\right)}$$

Because T-w-2 cancels, it makes no difference.

We run program P on two different machines, a 1-core processor and an 8-core processor, with the following results. CPI_*i* is the average cycles per instruction for a single core on an *i*-core processor:
CPI_1 = 1.2, CPI_8 = 1.8

The number of instructions executed per core on an *i*-core processor is *n_i*:
*n*_1 = 10, *n*_8 = 1.3 (in Giga-instructions)

We can adjust the voltages and clock rates:
CR = 3 GHz at v = 1 Volt, or CR = 1/2 GHz at v = 1/2 Volt

Power consumption is = 5(v * v) CR (1/GHz) per core:
15 Watts at 3 GHz, 5/8 Watt at 1/2 GHz.

Q. Find the time to run P and the total energy consumed in all four cases: (1-core, 3 GHz; 8-core, 3 GHz; 1-core, 1/2 GHz; 8-core, 1/2 GHz). Recall, energy = power * time. What is the maximum ratio of energy consumption to get the job done. What is the maximum speed-up of the fastest versus the slowest? Which configuration offers the best trade-off? Hint: use the basic processor performance equation to get the execution time.

$T_{1\text{-}1V} = n_1 CPI_1 (1/3\,GHz) = (10G)(1.2)(1/3)10^{-9} = 4\,s$

$T_{1\text{-}1/2V} = n_1 CPI_1 (1/(1/2)\,GHz) = (10)(1.2)(2) = 24\,s$

$T_{8\text{-}1V} = n_8 CPI_8 (1/3)10^{-9} = (1.3)(1.8)(1/3) = \frac{13}{10}\frac{6}{10}\,s = \frac{78}{100}\,s$

$T_{8\text{-}1/2V} = n_8 CPI_8 (2)\,10^{-9} = (1.3)(1.8)(2) = 6\,(78/100)\,s$

$E_{1\text{-}1V} = 4s\,(15W) = 60\,J$

$E_{1\text{-}1/2V} = 24s\,(5/8\,W) = 15\,J$

$E_{8\text{-}1V} = \frac{78}{100}s\,(15W)8\,J = \frac{39}{5\cdot 10}(3.5)2\cdot 4 = \frac{39\cdot 12}{5} \approx 96\,J$

$E_{8\text{-}1/2V} = 6\left(\frac{78}{100}\right)\left(\frac{5}{8}W\right)8 = 2\cdot 3\cdot\frac{39}{5\cdot 10}\frac{5}{8}8 = \frac{3\cdot 39}{5} \approx 24\,J$

$S_{max} \cong \frac{24}{80/100} = \frac{240}{8} = 30 = S_{(8\text{-}1V)\text{-}(1\text{-}1/2V)}$

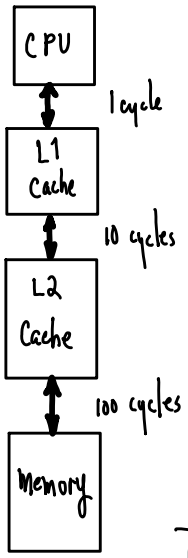
$\frac{E_{max}}{E_{min}} \cong \frac{96\,J}{15\,J} = \frac{4\cdot 28}{15} \approx 4(2) = 8 = E_{(8\text{-}1V)}/E_{(1\text{-}1/2V)}$

Best trade-off $\Rightarrow$ 8-core, 1/2 V $\quad T \cong 6(80/100) = 6(4/5) = 24/5 \approx 5s$

$S_{(8\text{-}1/2V)\text{-}(1\text{-}1/2V)} = \frac{T_{(1\text{-}1/2V)}}{T_{(8\text{-}1/2V)}} \cong \frac{24}{(24/5)} = 5$ times faster

$\frac{E_{(8\text{-}1/2V)}}{E_{(1\text{-}1/2V)}} \cong \frac{24\,J}{15\,J} = 1\,9/15$, or only about 60% more energy

A load-store machine M has a two-level cache heirarchy. On an L1 hit, instruction fetch completes in time for a non-memory instruction (not load/store) to be fetched and complete execution in one cycle. For a load/store instruction two requests are sent to L1 sequentially, one for fetch and one for data access. If both hit in L1, execution takes two cycles. If L1 misses, the first cycle detects the miss, and then 10 cycles are required to detect a hit or miss in L2. If L2 hits, then L1 completes the access in one more cycle. But if L2 misses, memory access takes 100 cycles, then 10 cycles later L2 sends a ready signal to L1, and L1 then completes the access in one more cycle. The CPU stalls until all accesses are complete.



Q. Given clock rate CR how much time is required to execute a non-memory instruction that hits in L1? How much time to execute a memory instruction that hits in L1 for both fetch and data? What is the L1 hit time per access?

T_h = 1 cycles ($1/CR$) $T_{h/h}$ = 1 + 1 cycles ($1/CR$)

Per access, 1 cycle per hit.

Q. How much time is required for a non-memory instruction to complete execution if it misses in L1 but hits in L2? If it misses in L2? Write each as a sum of individual components.

T_{mh} = 1 + 10 + 1 cycles ($1/CR$) T_{mm} = 1 + 10 + 100 + 10 + 1 cycles ($1/CR$)

Q. What is the miss penalty for an access (instruction fetch or data) that misses in L1 and hits in L2 (aka, L2 hit time)? What is the miss penalty for an access to L2 that misses?

$T_{1,penalty}$ = 11 cycles $T_{2,penalty}$ = 110 cycles

Penalty time is the extra time we have to spend to access the next lower level. Every instruction has to hit L1. If we miss L1 and hit in L2 we pay an extra 10+1 cycles. An L2 miss incurs that expense plus another 100+10 cycles.

Q. Give an expression for average instruction execution time in terms of the quantities above and miss rates for L1 and L2.

There are several ways to look at this. An intuitive approach is as follows:

n instructions, T total execution time $\Rightarrow$ $\bar{T} = T/n$ is average time per instruction.

Some are load/store, the rest are not: $n = (\%LS)n + (1-\%LS)n$, $\%LS$ = fraction that are load/store. $T = \sum_{LS} T_i + \sum_{non\text{-}LS} T_i$. Let's tackle the second term: $m = (\%LS)n$

$e = (1-\%LS)n$ are the number of LS instructions and non-LS instructions.

$$\sum_{\text{non-LS}} T_i = \underbrace{e(HR_1)}_{\text{number that hit in L1}}(T_1) + \underbrace{e(MR_1)(HR_2)}_{\text{miss in L1 and hit in L2}}(T_1 + T_2 + T_1) + \underbrace{e\,MR_1MR_2}_{\text{miss both}}(T_1 + T_2 + T_m + T_2 + T_1)$$

(T1: L1 hit time; T1 + T2 + T1: L1 miss + L2 hit + L1 hit; T1 + T2 + Tm + T2 + T1: L1 miss, L2 miss, mem, L2 hit, L1 hit)

$$\frac{1}{e}\sum_{\text{non-LS}} T_i = (1-MR_1)(T_1) + MR_1(1-MR_2)(2T_1 + T_2) + MR_1MR_2(2T_1 + 2T_2 + T_m)$$

$$= T_1 + (-MR_1T_1) + 2MR_1T_1 + MR_1T_2 - MR_1MR_2T_1 - MR_1MR_2T_2$$

$$+ 2MR_1MR_2T_1 + 2MR_1MR_2T_2 + MR_1MR_2T_m$$

$$= T_1 + MR_1T_1 + MR_1T_2 + MR_1MR_2T_2 + MR_1MR_2T_m$$

$$= T_1 + MR_1(T_1 + T_2 + MR_2(T_2 + T_m))$$

$$= \underbrace{T_1}_{\text{every instruction has to hit L1}} + MR_1(\underbrace{T_{1,\text{penalty}}}_{\text{added time for hitting L2}} + MR_2(\underbrace{T_{2,\text{penalty}}}_{\text{added time for mem access}}))$$

We've just derived the cache performance equation. We can use it to deal with the LS instructions. Because there are two accesses per instruction, we multiply by 2.

$$\frac{1}{m}\sum_{LS} T_i = \sum_{\text{non-LS}}\left(T_{\text{fetch+exec}_i} + T_{\text{data-access}_i}\right) = 2\left(T_1 + MR_1(T_{1,\text{penalty}} + MR_2(T_{2\,\text{penalty}}))\right)$$

$$T = \sum_{\text{non-LS}} T_i + \sum_{LS} T_i = (e+2m)\left(T_1 + MR_1(T_{1,\text{penalty}} + MR_2(T_{2\,\text{penalty}}))\right)$$

$$(e+2m) = n((1-\%LS) + 2(\%LS)) = n(1+\%LS)$$

$$\overline{T} = T/n = (1+\%LS)\left[1 + MR_1(11 + MR_2(110))\right]$$

Machine M has a split L1 cache and unified L2. Running program P, L1's miss rate is MR_1 = 5% (combined) and L2's miss rate is MR_2 = 1%. Loads and stores account for 30% 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*, cache *i* 's hit time is T_*i*, and memory's access time is T*m*.

Q. How many memory accesses occur? State the result in terms of the parameters given. How many hit in L1? How many miss L1?

n instructions $\Rightarrow$ (n fetches) + ($30\%\ n$ load or store data) $\Rightarrow$ $1.3n$ total memory accesses

(L1 hits) = $(1.3n)(95\%)$ (L1 misses) = $(1.3n)(5\%)$

Q. What is the total execution time, ignoring all stalls? How many clock cycles?

n instructions $\Rightarrow$ $n \cdot T_1$ cycles = $n \cdot T_1 \cdot (CR)$

Q. Of the accesses that miss in L1, how many miss in L2? How much main memory access stall time results? How many stall cycles?

(L1 misses = $(1.3n)(5\%)$)(1%) = $(1.3n)(5\%)(1\%)$ = (L2 misses)

(L2 miss penalty) = (L2 misses) T_m = $(1.3n)(5\%)(1\%)\ T_m$

$\Rightarrow (1.3n)(5\%)(1\%)T_m(CR)$

Q. How much stall time is attributable to L2 access, ignoring main memory access stalls? How many cycles?

all L1 misses must pay an L2 access stall equal to L_2 hit time = T_2.

(L2 access time) = (L1 misses) T_2 = $(1.3n)(5\%)T_2$

(L2 access cycles) = $(1.3n)(5\%)T_2\ (CR)$

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

$T = T_{exec} + (T_{stalls} = T_{L2\text{-}access} + T_{mem\text{-}access})$

$= n \cdot T_1 + (1.3n)(5\%)T_2 + (1.3n)(5\%)(1\%)T_m = n\left(T_1 + (1.3)\{(5\%)T_2 + (5\%)(1\%)T_m\}\right)$

avg. CPI = $\frac{\#\text{cycles}}{\#\text{instructions}} = \frac{T(CR)}{n} = \left[T_1 + (1.3)\{(5\%)T_2 + (5\%)(1\%)T_m\right]CR$

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?

$$CPI_{new} = \left[\frac{T_1}{2} + (1.3)\left\{(5\%)\frac{T_2}{2} + (5\%)(1\%)T_m\right\}\right](2\,CR)$$

$$= \left[T_1 + (1.3)\left\{(5\%)T_2 + (5\%)(1\%)2T_m\right\}\right](CR) = CPI_{old} + (5\%)(1\%)T_m\,CR$$

Because 2nd term is $5 \cdot 10^{-4} T_m$ ($\times 10^9$), $\times$ small integer (2?), T_m is on the order of 100ns, and CPI is small integer (2?) overall effect is $\frac{1/2}{2} = 25\%$ increase in CPI.

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

$$S = \frac{T_{old}}{T_{new}} = \frac{n\,CPI_{old}\,(1/CR)}{n\,CPI_{new}\,(1/2CR)} = \frac{CPI_{old}\;2}{CPI_{old}\,(1.25)} = 2\left(\frac{4}{5}\right) = 8/5 = 1\,3/5,\ 60\%\ \text{faster}$$

Q. Assuming some reasonable relationships between T_1 and T_2 and T_m, what qualitatively is the effect of the improvement? What does Amdahl's Law suggest in regard to which aspect of performance should be improved? Supposing CR keeps doubling every two years?

$T_2 = 10\,T_1 \quad T_m = 10\,T_2$

$$S = \frac{1 + (1.3)\{5\%\,(10) + (5\%)(1\%)(100)\}}{1/2 + (1.3)\{5\%\,(5) + (5\%)(1\%)(100)\}} = \frac{1 + (1.3)\{0.55\}}{1/2 + (1.3)\{0.3\}} = \frac{1 + 0.55 + 0.15 + 0.015}{0.5 + 3.9}$$

$$= \frac{1.715}{8.9} \approx \frac{17}{9} = 1\,8/9$$

$\approx$ doubled

$$T = 1 + (1.3)(0.05)10 + (1.3)(0.05)$$

$$= 1 + 0.65 + 0.065$$

↖ L1 performance is biggest % of work: speed-up L1 has biggest effect

$$T = \frac{1}{2^n} + \frac{0.65}{2^n} + (1.3)(0.05)$$

$$\approx (1 + 0.7)/2^n + 0.07$$

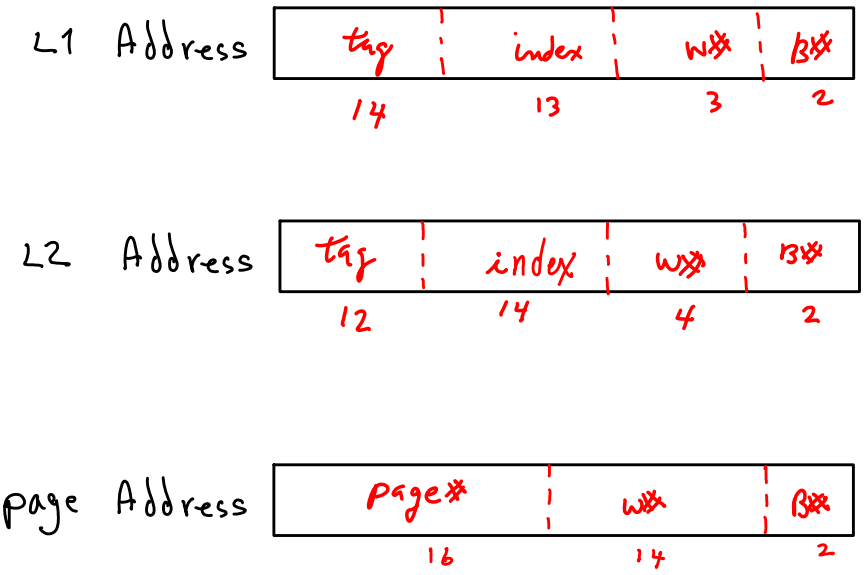
$$= (1.7)/2^n + 0.07$$

When $\frac{1.7}{2^n} < 0.07$ mem bandwidth is most important, or

$2^n > \frac{1.7}{7/100} = \frac{170}{7} \approx 24$. So, after 5 doublings of CR, need to focus on mem

Machine M has 32-bit virtual and physical addresses, 32-bit words, and memory is byte addressable. Pages are 64 kB. L1 is a 256-kB direct mapped cache and L2 is a 4-MB 4-way set-associative cache. Cache blocks are 8 and 16 words for L1 and L2, respectively.

Q. In the address bit-field diagrams below, divide the address bits into fields for byte number within a word, word number within a cache block, and tag, showing the number of address bits in each bit field as it applies to L1, L2, and virtual pages.



$256\,kB\left(\frac{block}{32B}\right) = \frac{2^8 2^{10}}{2^5} = 2^{13}$ blocks

$\Rightarrow$ 13 index bits

8 words = 3 bit w#

4B words $\rightarrow$ 2 bit B#

32 − 18 = 14-bit tag

$4MB\left(\frac{1}{4\ ways}\right) = 1MB/way\ (DM)$

$1MB\left(\frac{1\ block}{64B}\right) = \frac{2^{20}}{2^6} = 2^{14}$ blocks

$\Rightarrow$ 14 index bits

$\left(\frac{64B}{block}\right)\left(\frac{w}{4B}\right) \Rightarrow 2^4$ word/block

$\rightarrow$ 4-bit w#

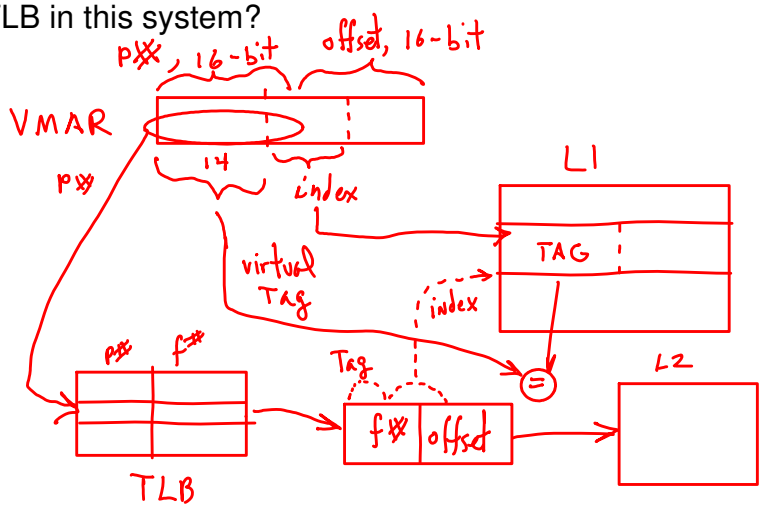
32 − 20 = 12-bit tag

64 kB page $\Rightarrow 2^6 \cdot 2^{10}$ B/page $\Rightarrow$ 16 bits for page offset, 16-bit page#.

$\Rightarrow$ words are 4B $\Rightarrow$ 2 bit B# field

$\Rightarrow$ 14 bits for word#

Q. Comment on the feasibility of using virtual indexing in this system. (Recall, virtual indexing indexes into the caches before address translation.) Is there a performance problem w.r.t. to the TLB in this system?



Because page# overlaps 2 index bits, physical indexing results in an L1 delay until after TLB translation is done. Virtual indexing solves this problem. In that case, if L1 is physically tagged, the entire frame# must be used for the tag as the 2 physical index bits are not the same as the virtual index bits, and a partial frame# would not uniquely identify which block was present. Virtual tagging works and L1 access is fast. L2 can be physically indexed and tagged.

Q. Comment very briefly on the performance tradeoffs of each cache feature.

1. larger cache blocks — More spatial locality => lower miss rate; large block => higher latency replacement

2. more cache levels — lower miss penalty; more complexity + higher miss rates

3. more associativity — lower miss rate; slower access; more complex; more power

4. virtual tagging — faster L1 hit time + parallel translation; synonyms

5. larger total cache size w/ larger blocks

   Lower miss rate in spatially local code + more efficient (pipelined) block reads from memory; loss of efficiency for temporally local code + longer miss penalty + more collisions.

6. larger total cache size w/ smaller blocks

   Lower miss rate spatially local + shorter miss penalty + less collisions; less efficient memory access

7. write buffering w/ a searchable buffer — shorter write-miss penalty + no delay for reads that hit in buffer; more complexity and energy.

8. write-back — Lower memory bandwidth + more efficient memory access; more complexity

9. no-allocate/write-through — Faster writes + improved miss rate if active read blocks get hits that written block wouldn't get.

10. larger pages — Lower page miss rate + smaller page tables; longer replacement latency + more internal fragmentation.

11. PID fields in caches and TLBs — Less overhead flushing caches and TLB entries on context switch; more complexity and energy for PID matching + more space.

12. word-level valid bits within cache blocks — For write-back, faster write miss (no need to load block)+ less load penalty (on write-back).

13. valid bits in TLB entries — Provides for simple cache flushing.

A system uses an inverted page table on a machine with 64-bit virtual addresses. The inverted page table is hashed into, and if there is a miss (page number not found), the page number is re-hashed with a second hash function and the table probed again. If that fails, another rehash is applied, and so on. If after *k* re-hashes the page number is still not found, the table must be linearly searched. Each entry refers to a specific frame: e.g., page number 10 in entry 3 means that page 10 is in frame 3. A "free" bit in an entry indicates whether the frame contains a valid page.

If the requested page is not in memory, a second table indicating the page number and its corresponding disk address must be read to fetch the page from disk.

Q. How many entries in the inverted table if physical memory is 32 GB and pages are 4MB?

$32\,GB = 2^5 \cdot 2^{30}$

$4\,MB = 2^2 2^{20}$

$32GB\left(\frac{frame}{4\,MB}\right) = \frac{2^{35}}{2^{22}} = 2^{13}$ frames $\Rightarrow 2^{13}$ entries in PT.

$= 8k$ entries

Q. How big is the table? (Round the size of an entry up to the nearest integer number of bytes.)

Each entry has a virtual page#. 64 b address $\Rightarrow 2^{64}$ B virtual memory @ 4 MB/page

$\frac{2^{64}}{2^{22}} = 2^{42}$ pages $\Rightarrow$ 42 bit page# (+ 22 bit page offset)

$\Rightarrow$ 6 B entrys $\Rightarrow (8k\ PT\ entries)\left(\frac{6B}{entry}\right) = 6.8\,kB = 48\,kB$

Q. How many levels would be required for a multi-level page table scheme (not inverted), assuming each sub-table at any level fits in a single page?

Entries contain frame number $\rightarrow 2^{13}$ frames $\Rightarrow$ 13-bit frame# $\approx$ 2B

$\left(\frac{4\,MB}{page}\right)\left(\frac{entry}{2B}\right) = 2M$ entries per page.

n levels $\Rightarrow (2M)^n$ lowest-level entries $= 2^{42}$ pages

$\Rightarrow 2^{21 \cdot n} = 2^{42} \Rightarrow n = 2$

Q. How big is the disk map per process? That is, each process has its own pages, and each has its own map from page numbers to disk addresses for pages that are not in memory but on disk. (NB--Give a maximum, minimum, and some expected size for the table.)

Total (max) pages per process $= 2^{42} \Rightarrow 2^{42}$ entry disk map.

Suppose disk access unit is 1 kB, disk is 128 GB $\Rightarrow 128\,G/k = 128\,M$ disk addresses,

$\Longrightarrow$ disk address = 27 bits $\approx$ 4B/entry $\Rightarrow 2^{42}(4B)$ disk map $= 2^{14}2^{30}\,B = 16\,k$ GB

Min pages = 1 $\Rightarrow$ 1-entry map = 4B.

Avg.? 128 GB? $\Rightarrow 2^7 2^{30}/4\,MB = \frac{2^{37}}{2^{22}} = 2^{15}$ pages $\Rightarrow 32k(4B) = 128k\,B$ disk map.