Computer Architecture 2011 (CART'11)
Re-exam

August 11^{th}

Please write your student number on every page of the exam. If you need to add additional pages, please indicate how many and write your student number on each of them. **You should answer on the exam sheets**, there is space for it.

1 Introduction

This exam is divided into 7 sections that give a total of 118 points plus 4 bonus points. The mapping to the 7-scale grade is as follows:

$$\begin{array}{rl} 0 \ldots 20 & \to -3 \\ 21 \ldots 34 & \to 00 \\ 35 \ldots 48 & \to 02 \\ 49 \ldots 69 & \to 4 \\ 70 \ldots 90 & \to 7 \\ 91 \ldots 104 & \to 10 \\ 105 \ldots 118 & \to 12 \end{array}$$

You will find in appendix (last page) the list of the first powers of 2 that may help you for the whole exam. The questions are weighted in points in function of their difficulty or importance. As a hint, questions requiring to write some code that give x points can be solved with x lines of code.

2 Representing and Manipulating Information (30 pts)

Exercise1: Let us consider integers represented on w bits in the general case and 8 bits for the examples.

1. **(3 pts)** Consider a signed binary number represented on a w-bit word. The number is a string of w bits denoted b_i with lower bits starting from 0. What is the formula in function of b_i (and w) that gives the decimal number D represented by such a binary number?

2. **(4 pt)** Give the hexadecimal and decimal representations of the following binary numbers on 8 bits. Integers are **signed**.

10010010 = 0x =

00010110 = 0x =

10001111 = 0x =

11111110 = 0x =

3. **(2 pts)** What is the range of representable numbers for signed integers represented on w bits?

4. **(2 pts)** What is the range of representable numbers for unsigned integers represented on w bits?

Exercise2: In the old days fonts where nothing more than individual bitmaps for every character. In other words, a binary image with 0 and 1 shows how to display characters. We want to store an hexadecimal number in a bitmap and of course endianness matters. Suppose we want to represent the letter A as follows:

```
0110
1001
1001
1111
1001
1001
1001
0000
```

This is sequence of 32 consecutive bits that must appear exactly in this order on screen.

1. **(2 pts)** Consider a little-endian machine. What is the number you want to write to memory in binary and then hexadecimal formats?

2. **(2 pts)** Consider a big-endian machine. What is the number you want to write to memory in binary and then hexadecimal formats?

Exercise3: Assume we are running code on a 32-bit machine using two's-complement arithmetic for signed values. The variables are declared and initialized as in listing 1.

```
int x = foo();    /* Some arbitrary value. */
int y = bar();    /* Some arbitrary value. */
```

Listing 1: Declarations for `x` and `y`.

For each of the following C expressions, either (1) argue that it is true (it evaluates to 1) for *all* values of `x` and `y`, or (2) give values of `x` and `y` for which it is false (evaluates to 0). You can use a power of two or the constants `INT_MIN` and `INT_MAX` in your answers.

1. **(2 pts)** `(x*x) >= 0`

2. **(2 pts)** `x+y == uy+ux`

3. **(2 pts)** `x*~y + uy*ux == -x`
   where `~x` inverts all bits of `x`.

Exercise4: Assume variables `x`, `f`, and `d` are of type `int`, `float`, and `double`, respectively. Their values are arbitrary, except that neither `f` nor `d` equals $+\infty$, $-\infty$, or *NaN*. For each of the following C expressions, either argue that it will always be true (evaluates to 1) or give a value for the variables such that it is not true (evaluates to 0). You can give the values in decimal or hexadecimal.

1. **(2 pts)** `d == (double)(float)d`

2. **(2 pts)** `f == -(-f)`

3. **(2 pts)** `(d + (double)x) - d == x`

Exercise5: Commutativity and associativity are important properties on operators that matter for programmers and compilers.

1. **(2 pt)** Fill in the table to indicate if the $*$ operator is commutative and associative for integers and floating point numbers. Use Y for yes and N for no in the table.

$*$	int	float
commutative		
associative		

2. **(1 pt)** Considering communitativity and associativity and not the operator itself, what's the difference with the + operator?

3 Digital Logic (3 pts)

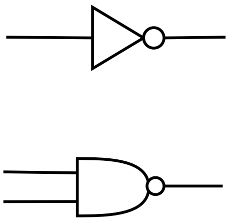


Figure 1: NOT (inverter) and NAND gates.

Exercise6: It is cheaper in terms of transistors to use NAND gates to implement other gates. Figure 1 shows the NOT and NAND gates.

1. **(2 pts)** Rewrite $or(a, b) = a \vee b$ in function of $nand(a, b) = \neg(a \wedge b)$ and $not(a) = \neg a$.

2. **(1 pts)** Make an OR gate from NAND and NOT gates.

4 Program Encodings (25+2 pts)

Reminder The general registers on IA32 are `%eax`, `%ebx`, `%ecx`, `%edx`, `%esi`, `%edi`, `%esp`, and `%ebp`. We adopt the same convention used by `gcc` for assembly, in particular for the order of the arguments. The syntax for the `mov` instruction is `mov` *src,dst*.
Exercise7: A function that has the following prototype

`void decode(int *xp, int *yp, int *zp);`

has its implementation compiled into assembly code. The code is given in listing 2. Assume we are in 32-bit mode on some Intel compatible CPU. As a reminder `movl X,Y` will copy the value of `X` to `Y`, where `X` **or** `Y` (not both) can be a memory reference (in the syntax given in the previous exercise) and the other argument a register (or a constant for `X` only). The operation `addl X,Y` will add the values of `X` to `Y`.

```
pushl     %ebp
movl      %esp,    %ebp
movl   8(%ebp),   %edi   ; xp
movl  12(%ebp),   %edx   ; yp
movl  16(%ebp),   %ecx   ; zp
movl    (%edx),   %ebx
movl    (%ecx),   %esi
movl    (%edi),   %eax
addl      %eax,  (%edx)
movl      %ebx,  (%ecx)
addl      %esi,  (%edi)
movl      %ebp,   %esp
pop       %ebp
ret
```

Listing 2: Assemby code of `decode`.

Parameters `xp`, `yp`, and `zp` are stored at the memory locations with offsets 8, 12, and 16, respectively, relative to the address in the register `%ebp`. In addition, when calling the function with three expressions, e.g., `decode(`*expr1,expr2,expr3*`)`, *expr3* is evaluated and pushed first, then *expr2*, and *expr1*.

1. **(2 pts)** Explain why the first offset is 8. In other words what is located at offsets 0 and 4?

2. **(6 pts)** Draw the relevant part of the stack as seen by this function from its first `movl` instruction. Start your stack at address `0x100` to clearly show how you map your stack in memory and indicate where the stack would "grow", should you have more `push`.

3. **(6 pts)** Write C code for `decode` that will have an effect equivalent to the assembly code of listing 2.

```
int decode(int *xp, int *yp, int *zp)
{











}
```

Listing 3: Your implementation of `decode`.

Exercise8: Conditional jumps are very common in programs. If we consider the code in listing 4 that computes $|x - y|$, the return statement depends on a comparison between x and y. Note the C statement *cond ? p : q* evaluates to p if *cond* is true else q.

```
void absdiff (int x, int y)
{
    return (x < y) ? y - x : x - y;
}
```

Listing 4: Implementation of `absdiff`.

```
push      %ebp
mov       %esp,    %ebp
movl   8(%ebp),    %edx   ;  Get x
movl  12(%ebp),    %eax   ;  Get y















mov       %ebp,    %esp
pop       %ebp
ret
```

Listing 5: Your assembly code for absdiff without conditional jumps.

1. **(2 pts)** Normally the corresponding assembly code would involve comparing `x` and `y`, and then make a conditional jump to perform the right subtraction. Such a conditional jump

that depends on some arbitrary value is bad for the processor. Explain why[1].

2. **(5 pts)** One way to improve performance is to evaluate both subtractions and use a conditional move. Write the assembly code corresponding to this solution using conditional move(s) in listing 5.
   You may use one or two conditional moves. You may use `%ecx` as an additional register. You will need the instruction `cmpl`, and possibly `cmovge` or `cmovl`. The result of the function should be in `%eax` upon returning.

   `cmpl Y, X` compares `X` and `Y` and sets flags accordingly.
   `cmovge A,B` copies `A` to `B` if `X >= Y` after the previous compare.
   `cmovl A,B` copies `A` to `B` if `X < Y` after the previous compare.
   `subl X, Y` computes `Y = Y - X`.
   Note: the conditional moves depend on flags that are changed by `sub` as well.

3. **(Bonus +1 pt)** You cannot use register `%ebx` directly. Why not?

Exercise9: Let us consider the structure declared as

```
struct rec {
  int a[4];
  int i;
  int j;
};
```

1. **(4 pts)** Fill-in the assembly code in listing 6 that implements the function

   `int readrec(struct rec* p);`

   that should return `p->a[(p->i+p->j) & 3]`. Hint: it may help to write down the offsets to access the data before trying to write the code. You will want to use the `addl` instruction. You can use it as `addl D(adr),X` for adding the number at address `adr` with some offset (or displacement) `D` to `X`.

```
push    %ebp
mov     %esp,    %ebp
movl    8(%ebp), %edx   ;  Get p









mov     %ebp,    %esp
pop     %ebp
ret
```

Listing 6: Your assembly code for returning `p->a[p->i+p->j]`.

2. **(Bonus +1pt)** The logical and `& 3` may look strange but it makes the function safe. What is its meaning?

[1] That question is on computer architecture, not the program encoding.

5 Y86 (15+1 pts)

Stage	OP rA,rB	`mrmovl` D(rB),rA

Fetch	icode:ifun ← M_1[PC]
	rA:rB ← M_1[PC+1]
	
	valP ← PC+2
		icode:ifun ← M_1[PC]
		rA:rB ← M_1[PC+1]
		valC ← M_4[PC+2]
		valP ← PC+6

Decode	valA ← R[rA]
	valB ← R[rB]
		
		valB ← R[rB]

Execute	valE ← valB OP valA set CC	valE ← valB + valC
Memory		valM ← M_4[valE]
Write back	R[rB] ← valE	R[rA] ← valM
PC update	PC ← valP	PC ← valP

Table 1: Computation stages for an arithmetic operator OP and the memory to register move `mrmovl`.

Exercise10: For these questions we will consider the simplified CPU model of the Y86. Its processing is separated into different stages as shown in table 1 for two instructions. The stages are using a restricted set of hardware registers (rA, rB, valA, . . .) to do the computations.

1. **(1 pts)** Typically these stages would be implemented in a pipeline architecture. Suppose that an instruction is stalled because it needs the value of a particular register that is not ready yet. At which stage will it be stalled and why?

2. **(1 pts)** Suppose that an instruction is stalled because it needs to read memory and the memory architecture needs a few more cycles to answer. At which stage will the instruction be stalled and why?

3. **(2 pts)** Deduct the byte encoding of the two instructions shown in the table 1 (you can take `OP=add` as an example if you want). As a remainder M_i stands for a memory reference of size i bytes, and the D in for `mvmovl` stands for displacement.

4. **(1 pt)** The `ret` instruction used to return from function calls is not good for pipelining. Why not?

5. **(Bonus +1)** What is the common trick to solve that problem?

6. **(2 pts)** The X86 has more complex instructions than the Y86, in particular it can add a value in memory to a register. If you try to encode such a `madd D(rB),rA` instruction, you will fail with the Y86 architecture. Based on table 1 explain why.

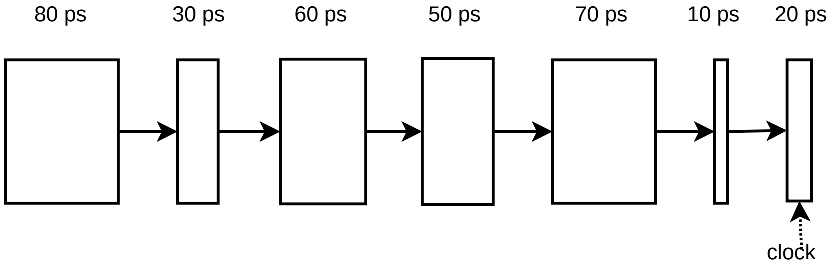
Exercise11: To improve performance, a *pipeline* architecture is used instead of a sequential one. This is done by inserting temporary registers between pipeline stages and trying to run all stages in parallel.

1. **(1 pt)** What is improved[2]?
2. **(1 pt)** What is degraded for every instruction?
3. **(2 pts)** Name two problems that need to be solved with pipelines.

Exercise12: Suppose we analyze some combinational logic and determine that it can be separated into a sequence of six blocks, named A to F, having delays of 80, 30, 60, 50, 70, and 10 ps, respectively.

We can create pipelined versions of this design by inserting pipeline registers between pairs of these blocks. Different combinations of pipeline depth (how many stages) and maximum throughput arise, depending on where we insert the pipeline registers. Assume that a pipeline register has a delay of 20 ps.

1. **(4 pts)** What is the minimum number of stages that would yield a design with the maximum achievable throughput? Describe this design, its throughput, and its latency.



Throughput:
Latency:

6 RAM and Memory Hierarchy (13 pts)

Exercise13: Different RAM technologies are used together in computers.

1. **(1 pt)** SRAM memories consume more energy than DRAM memories. Why?
2. **(2 pts)** On the other hand SRAM memories are faster than DRAM memories. Give two reasons that explain why.

Exercise14: The principle of locality is *the fundamental* reason why modern computers work in practice.

1. **(2 pts)** What is temporal locality?

[2] Answer concretely, e.g., not *performance*.

2. **(2 pts)** What is spatial locality?

```
void mult1(double a[M][N], double b[N][P], double c[M][P])
{
  int i, j, k;

  for (i = 0; i < M; i++)
  {
    for (j = 0; j < P; j++)
    {
      c[i][j] = 0
      for (k = 0; k < N; ++k)
      {
        c[i][j] += a[i][k]*b[k][j];
      }
    }
  }
}

void mult2(double a[M][N], double b[N][P], double c[M][P])
{
  int i, j, k;
  double sum;

  for (i = 0; i < M; i++)
  {
    for (j = 0; j < P; j++)
    {
      c[i][j] = 0;
    }
    for (k = 0; k < N; ++k)
    {
      for (j = 0; j < P; j++)
      {
        c[i][j] += a[i][k]*b[k][j];
      }
    }
  }
}
```

Listing 7: Different implementations of matrix multiplication.

Exercise15: Let us consider the implementations of listing 7 that compute the matrix multiplication $c = a * b$. The actual algorithm is irrelevant here, both implementations will yield the same result.

1. **(3 pts)** Considering the inner loop of `mult1`, explain the (temporal and/or spatial) locality for accessing the elements of each of the arrays.

2. **(3 pts)** Considering the inner loop of `mult2`, explain the (temporal and/or spatial) locality

for accessing the elements of each of the arrays.

7 Cache Memory (14+1 pts)

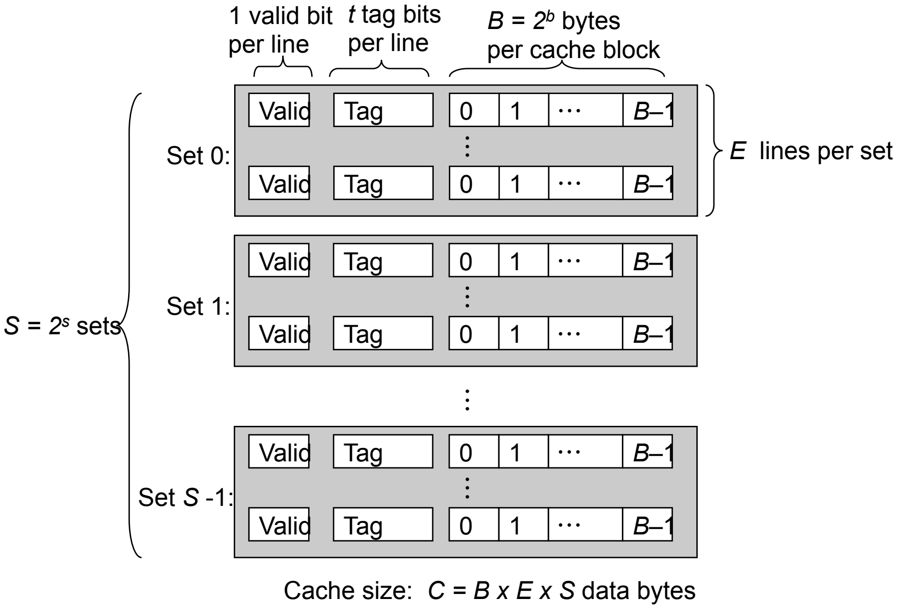


Figure 2: General organization of caches.

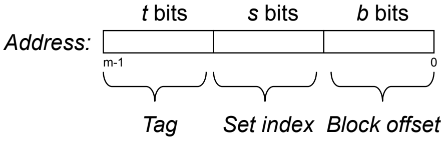


Figure 3: Physical addressing.

Exercise16: Caches are organized in general into sets of lines where each line stores a block of bytes as shown in Figure 2. A cache is determined by its parameters S, E, and B.

1. **2 pts** When choosing which piece of data to evict from the cache, the LRU (least recently used) policy is used. What does it do and why is it a good policy?

2. **(3 pts)** When a piece data is updated with a new value, it needs to be written to memory. There is a choice on *when* to do that. Two different policies for that are write-back and

write-through. What does each of them do and which one is better (and why)?

Exercise17: Assume that the memory is byte addressable with addresses on 13 bits, you can access bytes individually, and the cache is two-way associative (E=2), with a 4-byte block size (B=4) and eight sets (S=8) (see Fig. 2). The contents of the cache are as follows, with all numbers given in hexadecimal notation.

Set	Tag	Valid	Byte0	Byte1	Byte2	Byte3	Tag	Valid	Byte0	Byte1	Byte2	Byte3
0	09	1	86	30	3F	10	00	0	-	-	-	-
1	45	1	60	4F	E0	23	38	1	00	BC	0B	37
2	EB	0	-	-	-	-	0B	0	-	-	-	-
3	06	0	-	-	-	-	32	1	12	08	7B	AD
4	C7	1	06	78	07	C5	05	1	40	67	C2	3B
5	71	1	0B	DE	18	4B	6E	0	-	-	-	-
6	91	1	A0	B7	26	2D	F0	0	-	-	-	-
7	46	0	-	-	-	-	DE	1	12	C0	88	37

1. **(1 pts)** Indicate which bits would be used to determine the cache block offset (CO), the cache set index (CI), and the cache tag (CT).

2. **(6 pts)** Suppose a program running on this machine references the 1-byte word at address `0x0E34`. Indicate the cache entry accessed and the cache byte value returned in hexadecimal. Indicate whether a cache miss occurs. If there is a cache miss, write "-" for the byte returned. Repeat for the memory address `0x0DD5`. Hint: You will have to decompose the addresses according to the bit fields of the previous question.

Address `0x0E34`		Address `0x0DD5`	
Parameter	Value	Parameter	Value
Block offset (CO)	0x...	Block offset (CO)	0x...
Set index (CI)	0x...	Set index (CI)	0x...
Tag (CT)	0x...	Tag (CT)	0x...
Hit? (Y/N)	...	Hit? (Y/N)	...
Byte returned	...	Byte returned	...

3. **(1 pt)** Suppose we want to store a matrix of NxN doubles (8 bytes per double) in this cache. How big can N be?

4. **(1 pt)** What is pre-fetching?

5. **(Bonus 1pt)** Find a drawback of pre-fetching[3].

[3] Apart from additional transistors to implement it.

8 Virtual Memory (18 pts)

Exercise18: Virtual addressing is used instead of physical addressing in running programs. Virtual addresses need to be translated at some point in time to physical addresses to access data.

1. **(4 pts)** The virtual memory *system* consists of a software part (the operating system) and a hardware part (the MMU). What is the role of each?

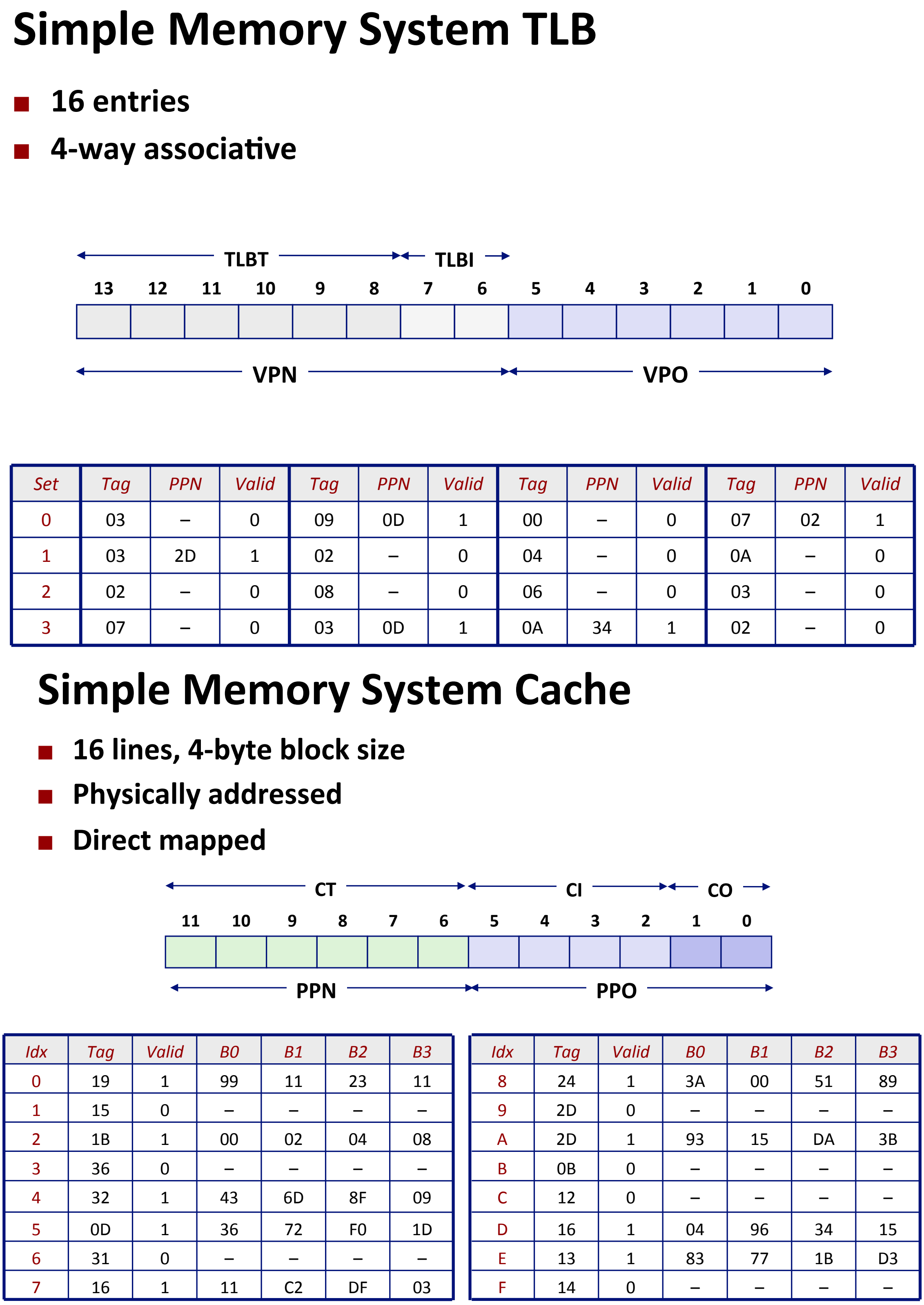
Exercise19: Figure 4 shows a simple memory system with a 4-way associative TLB (translation look-aside buffer) with 16 entries and a direct mapped cache with 16 lines and 4 bytes per line (block size).

1. **(1 pt)** The TLB is a special cache. What does it cache?

2. **(1 pt)** What is the page size of this memory system?

3. **(12 pts)** For the virtual addresses `0x0936`, `0x0369`, and `0x069f`: translate them into their physical addresses according to the memory system of Figure 4, indicate when you have a TLB or cache hit/miss, and read the corresponding byte whenever possible. If you have a TLB miss, you can still make a look-up in the cache, in which case you should indicate a PPN that would be needed to get a cache hit. If you cannot read any data, indicate what the memory system[4] will do. Enumerate clearly each stage of the translation.

[4] That includes the OS and the MMU.



Simple Memory System TLB

- 16 entries
- 4-way associative

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

Simple Memory System Cache

- 16 lines, 4-byte block size
- Physically addressed
- Direct mapped

Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Figure 4: Simple memory system.

A First Powers of 2

$$\begin{aligned}
2^0 &= 1 \\
2^1 &= 2 \\
2^2 &= 4 \\
2^3 &= 8 \\
2^4 &= 16 \\
2^5 &= 32 \\
2^6 &= 64 \\
2^7 &= 128 \\
2^8 &= 256 \\
2^9 &= 512 \\
2^{10} &= 1024 = 1k \\
2^{11} &= 2048 = 2k \\
2^{12} &= 4096 = 4k
\end{aligned}$$