Computer Architecture 2011 (CART'11)
Re-exam

August 11th

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 distribution of the marks was as follows:

Points	Students	Mark	Total
112 . . . 118+4	0	12	0
105 . . . 111	0		
98 . . . 104	0	10	0
91 . . . 97	0		
84. . . 90	0		
77 . . . 83	0	7	0
70 . . . 76	0		
63 . . . 69	1	4	5
56 . . . 62	1		
49 . . . 55	3		
42 . . . 48	5	2	12
35 . . . 41	7		
28 . . . 34	3	0	9
21 . . . 27	6		
14 . . . 20	1		
7 . . . 13	1	-3	2
0 . . . 6	0		

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?

$$D = -b_{w-1}2^{w-1} + \sum_{i=0}^{w-2} b_i 2^i$$

Also accepted: $D = b_0 2^0 + b_1 2^1 + \ldots b_{w-2}2^{w-2} - b_{w-1}2^{w-1}$.

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

10010010	=	**0x92**	= -128+16+2 = **-110**
00010110	=	**0x16**	= 16+4+2 = **22**
10001111	=	**0x8f**	= -128+15 = **-113**
11111110	=	**0xfe**	= -128+126 = **-2**

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

$$-2^{w-1} \ldots 2^{w-1} - 1$$

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

$$0 \ldots 2^{w} - 1$$

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?
   Binary number (human notation) = 00001001100110011111100110010110
   Hexadecimal = 0x0999f996
2. **(2 pts)** Consider a big-endian machine. What is the number you want to write to memory in binary and then hexadecimal formats?
   Binary number (human notation) = 01101001100111111001100110010000
   Hexadecimal = 0x699f9990

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.

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

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

The variables ux and uy (unsigned int) should have been specified but the exercise was clear enought that these were variables, not $u*x$ or $u*y$ given that these were C-expressions. You were supposed to be familiar with the exercise too.

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`
   False. Multiplication of positive numbers overflows to negative.
2. **(2 pts)** `x+y == uy+ux`
   True. Addition is the same bit-wise for signed and unsigned integers.
3. **(2 pts)** `x*~y + uy*ux == -x`
   where `~x` inverts all bits of `x`.
   True. ~y=-y-1 and * is distributive so x*~y+uy*ux==-x*y-x+uy*ux=-x*y-x+y*x==-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`
   False. Double to float loses precision.
2. **(2 pts)** `f == -(-f)`
   True. Invert twice the sign bit.
3. **(2 pts)** `(d + (double)x) - d == x`
   False. The term (double)x can be replaced by another d' or f, it does not matter, addition is not associative for floating point numbers.

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	Y	Y
associative	Y	N

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

   There is no difference! You can still lose precision with * between floating point numbers.

3 Digital Logic (3 pts)

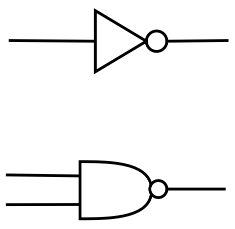


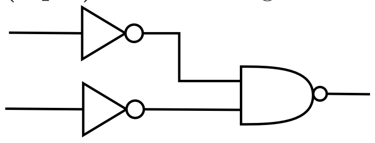
Figure 1: NOT (inverter) and NAND gates.

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 $not(a) = \neg a$.

$$or(a, b) = \neg(\neg a \wedge \neg b) = nand(not(a), not(b))$$

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?

   You have just pushed %ebp (offset 0) and the return address is offset 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`.

Sol 1	Sol 2	Stack
0x100	0x110	zp
0x0fc	0x10c	yp
0x0f8	0x108	xp
0x0f4	0x104	ret adr
0x0f0	0x100	ebp

3. **(6 pts)** Write C code for `decode` that will have an effect equivalent to the assembly code of listing 2. *The function was wrongly declared to return int, that was a minor typo.*

```c
void decode(int *xp, int *yp, int *zp)
{
 int y = *yp;
 int z = *zp;
 int x = *xp;
 *yp += x;
 *zp = y;
 *xp += z;
}
```

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.

```c
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

;   Solution 1        ;     Solution 2      ;   Solution 3      ;   Solution 4
movl   %eax,%ecx   ; cmpl    %eax,%edx ; movl    %eax,%ecx  ;  movl   %eax,%ecx
subl   %edx,%eax   ; cmovge %eax,%ecx ; subl    %edx,%eax  ;  cmpl   %eax,%edx
subl   %ecx,%edx   ; cmovge %edx,%eax ; subl    %ecx,%edx  ;  cmovl  %edx,%ecx
cmpl  %ecx,8(%ebp); cmovge %ecx,%edx ; cmovge %edx,%eax ; cmovge %edx,%eax
cmovge %edx,%eax ; subl     %edx,%eax ;                          ; subl    %ecx,%eax

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].

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

Due to the pipeline architecture, the processor will either stall or need to predict the next instruction but if it gets it wrong it needs to undo the wrong branch.

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.

   There are several solutions. The least obvious (not hinted) that some found was to use the result flag of the last subtraction.

3. **(Bonus +1 pt)** You cannot use register `%ebx` directly. Why not?
   That is against the rules of what the caller or the callee should save. To be more precise the callee should save and restore %ebx.

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

movl 16(%edx), %eax     ;  p−>i after 4 ints  (4 bytes each)
addl  20(%edx), %eax    ;  add p−>j
andl  $3,  %eax         ;  do and 3,  forgetting  the $ is  ok
movl (%edx,%eax,4),%eax ; read at p + index∗4 (4 bytes  per  int )

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?
   It is doing a modulo 4 to make sure the index is not out-of-bound. In fact if the resulting index is negative, this will also work. The trick only works because 4 is a power of 2.

5 Y86 (15+1 pts)

Stage	OP rA,rB	`mrmovl` D(rB),rA
Fetch	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valP $\leftarrow$ PC+2	icode:ifun $\leftarrow$ M_1[PC] rA:rB $\leftarrow$ M_1[PC+1] valC $\leftarrow$ M_4[PC+2] valP $\leftarrow$ PC+6
Decode	valA $\leftarrow$ R[rA] valB $\leftarrow$ R[rB]	 valB $\leftarrow$ R[rB]
Execute	valE $\leftarrow$ valB OP valA set CC	valE $\leftarrow$ valB + valC
Memory		valM $\leftarrow$ M_4[valE]
Write back	R[rB] $\leftarrow$ valE	R[rA] $\leftarrow$ valM
PC update	PC $\leftarrow$ valP	PC $\leftarrow$ 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?
   The stage where it reads from the register file: decode.
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?
   The stage where it reads from memory: memory. *That question was ambiguous and fetch stage was also accepted.*
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.
   add: 1st byte=(icode:ifun), 4 bits each, 2nd byte=(rA:rB), 4 bits each.
   mrmovl D(rB),rA: 1st byte=(icode:ifun), 4 bits each, 2nd byte=(rA:rB), 4 bits each, 4 last bytes=offset D.
4. **(1 pt)** The `ret` instruction used to return from function calls is not good for pipelining. Why not?
   The return address that tells the processor where to go next is on the stack and the processor needs to access memory 1st to know where to go next. It cannot even guess where that is.
5. **(Bonus +1)** What is the common trick to solve that problem?
   The processor can keep a calling stack internally (that is not in memory) and read it instead of the stack. Later it would need to confirm when the stack is read that the address was good.
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.
   You need to use the ALU twice: to compute the address and to compute the arithmetic operation. You can't do that with this architecture on the same

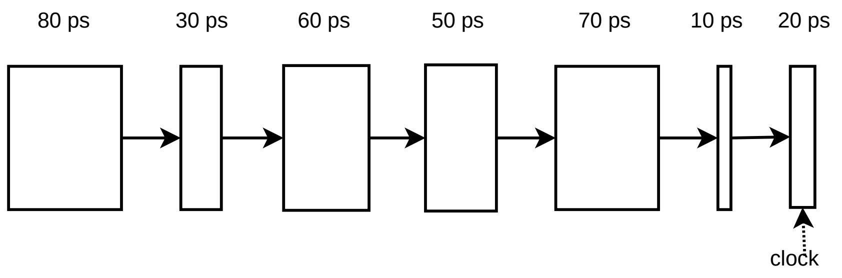
instruction. *It was also accepted if you said that you needed to use the execution stage after the memory stage, or anything similar that made sense.*

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]?
   Throughput.
2. **(1 pt)** What is degraded for every instruction?
   Latency.
3. **(2 pts)** Name two problems that need to be solved with pipelines.
   Dependencies, prediction, "hazards", latencies, balance the pipeline, etc . . . take your pick.

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: 1/(100 ps) = 10^{10} inst. per second.
   Latency: 80+20+30+20+60+20+50+20+70+10+20 = 400ps.

   The bottleneck is at 80 (+20 for the additional register). 30+60 > 80, add a stage after 30, 60+50 > 80, add one after 60, 50+70 > 80, add one after 50, 70+10=80, keep them together.

   I noticed the answers to the practice problem are messed up for latencies.

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?
   They change state at the frequency of the processor (loop of inverters basically) whereas DRAM is refreshed at a much lower frequency.
2. **(2 pts)** On the other hand SRAM memories are faster than DRAM memories. Give two reasons that explain why.
   SRAM runs at a higher frequency, sits near the processor (lower latency, higher bandwidth), and does not have to wait for extra latencies caused by refresh cycles that DRAM has.

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

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

1. **(2 pts)** What is temporal locality?
   Locality in time: what is read now is read again later.
2. **(2 pts)** What is spatial locality?
   Locality in space: what is read here means the data nearby is read (now or) soon.

```c
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. *You could not identify clearly the different localities for the following questions. The question stated to do so for the elements of each of the arrays. There was no comparison of which one was better.*

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.

c[i][j] is written repeatedly in the k loop, that's temporal locality.
a[i][k] is read row-wise (consecutive elements on the row), that's spatial locality.
b[k][j] is read column-wise, no locality here, that's strided access of size P.

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.
   c[i][j] is written row-wise, spatial locality.
   a[i][k] is read repeatedly, temporal locality.
   b[k][j] is read row-wise, spatial locality.

7 Cache Memory (14+1 pts)

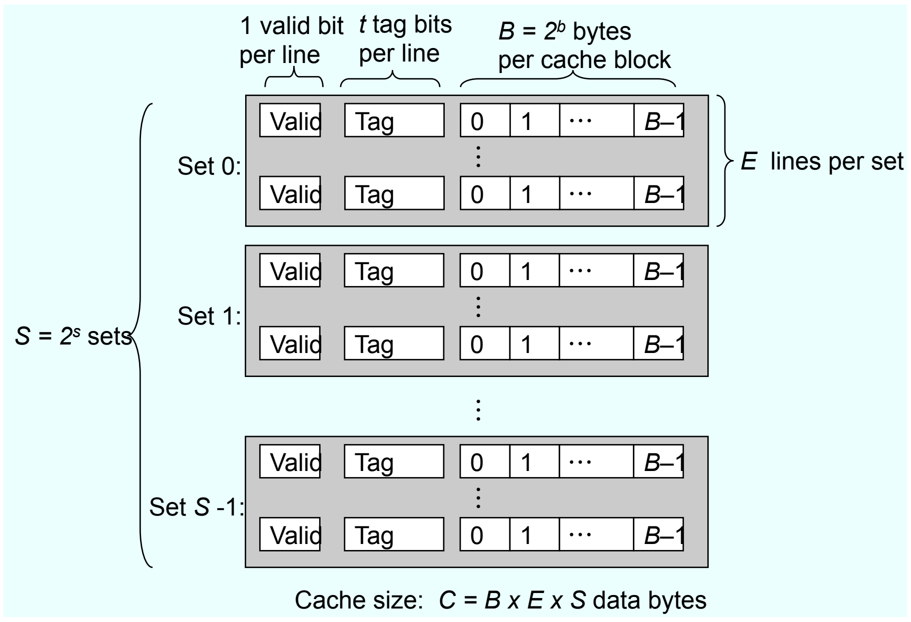


Figure 2: General organization of caches.

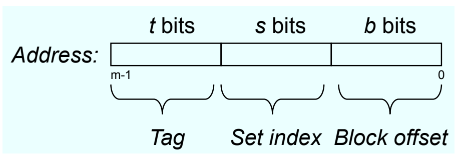


Figure 3: Physical addressing.

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?
   As the name suggests, it evicts the data that was least recently used (oldest untouched/unread). It is good because it respects temporal locality.
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)?

Write-back writes data back to memory (from the cache) only when the data is evicted. Write-through writes the data to memory (and through the cache, so to speak) everytime it is updated. Write-back is better because it minimizes traffic on the bus.

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).
   B=4 means we need 2 bits for CO, S=8 means we need 3 bits for CI, the rest is for CT (8 bits).
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` Parameter	Value	Address `0x0DD5` Parameter	Value
Block offset (CO)	0x0	Block offset (CO)	0x1...
Set index (CI)	0x5	Set index (CI)	0x5
Tag (CT)	0x71	Tag (CT)	0x6E
Hit? (Y/N)	Y	Hit? (Y/N)	N
Byte returned	0x0B	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?
   Cache size = 8x8 bytes, can store 8 doubles, max N = 2.
4. **(1 pt)** What is pre-fetching?
   Reading the next consecutive data in advance, before they are actually needed.
5. **(Bonus 1pt)** Find a drawback of pre-fetching[3].
   The memory bus can be overloaded needlessly if the data are not used.

[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?

   The hardware accelerates translations from virtual to physical addresses and when it cannot do it it hands over to the OS. The OS handles exceptions raised by the hardware, populates the TLB, kills processes when needed, and configures the MMU. The MMU checks for unauthorized reads and writes as well.
   Answers that said that the OS was managing the VM (or something similar) could give you points as well.

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).

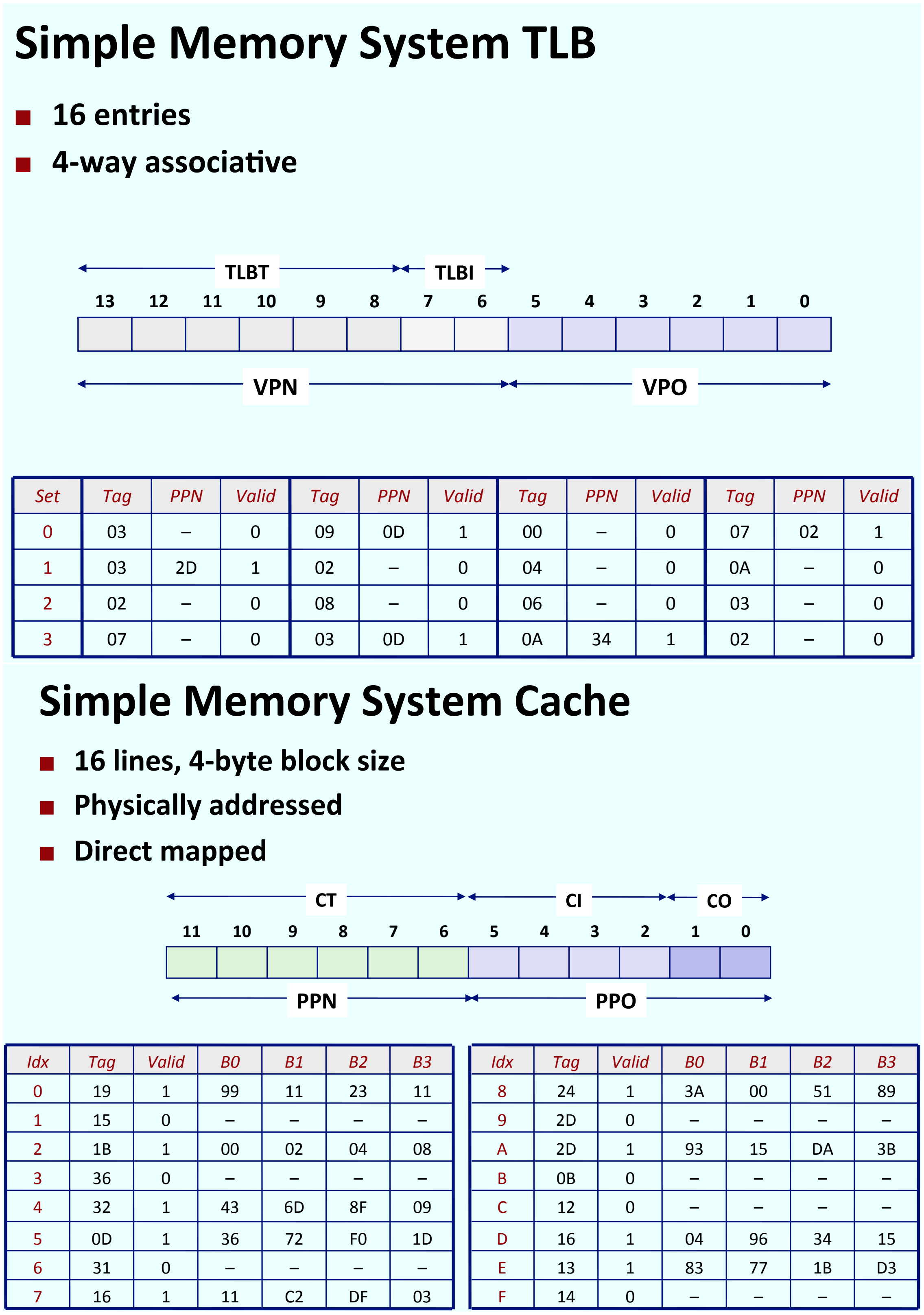
1. **(1 pt)** The TLB is a special cache. What does it cache?
   It caches physical addresses.
2. **(1 pt)** What is the page size of this memory system?
   It is $2^6 = 64$ bytes.
3. **(12 pts)** For the virtual addresses `0x0936`, `0x0369`, and `0x069f`: translate them into their physical addresses 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.

   0x936 = 00 1001 0011 0110
   VPN:VPO = 001001.00 : 1101.10
   TLBI=0 so PPN=0D=CT, tag 09 hit
   CI=D tag 16, miss.

   0x369 = 00 0011 0110 1001
   VPN:VPO = 000011.01 : 1010.01
   TLBI=1 so PPN=2D=CT, tag 03 hit
   CI=A tag 2D, hit $\rightarrow$ read 15.

   0x69f = 00 0110 1001 1111
   VPN:VPO = 000110.10 : 0111.11
   TLBI=2 so PPN=? tag 6 invalid, we get a page fault.
   CI=7 wants tag 16 for hit, so would read 03.

[4]That includes the OS and the MMU.



Simple Memory System TLB

- 16 entries
- 4-way associative

TLBT
TLBI
13 12 11 10 9 8 7 6 5 4 3 2 1 0
VPN
VPO

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

CT
CI
CO
11 10 9 8 7 6 5 4 3 2 1 0
PPN
PPO

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

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