Computer Architecture 2011 (CART'11)
Answers and Feedback

June 1st

Student number[1]: . . .
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.
There was some conflicting information about the names.

1 Introduction

This exam is divided into 6 sections that give a total of 143 points plus 9 bonus points. The mapping to the 7-scale grade is as follows:

$0 \dots 24$	$\rightarrow -3$
$25 \dots 41$	$\rightarrow 00$
$42 \dots 58$	$\rightarrow 02$
$59 \dots 84$	$\rightarrow 4$
$85 \dots 110$	$\rightarrow 7$
$111 \dots 127$	$\rightarrow 10$
$128 \dots 143$	$\rightarrow 12$

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.

The distribution of the marks is shown in table 1. 87 students were marked. There were a few typos in the exam and the most confusing one was spotted during the exam and you should have been told of it (pushl instead of popl). They were taken into account. Here they have been fixed.

2 Representing and Manipulating Information (36+2)

Exercise1: Let us consider integers represented on 8 bits.

1. **(4 pt)** Give the hexadecimal and decimal representations of the following binary numbers on 8 bits. Integers are **signed**. Hint: for decimal, it helps to write the formula, e.g., 00100010=0x22=2+2*16=34.

[1] No need to write your name.

Points	Students	Mark	Total
137 . . . 143+9	0	12	1
128 . . . 136	1		
120 . . . 127	2	10	4
111 . . . 119	2		
103 . . . 110	3	7	10
94 . . . 102	3		
85 . . . 93	4		
77 . . . 84	6	4	19
68 . . . 76	8		
59 . . . 67	5		
51 . . . 58	9	2	18
42 . . . 50	9		
34 . . . 41	10	0	15
25 . . . 33	5		
18 . . . 24	6	-3	20
9 . . . 17	8		
0 . . . 8	6		

Table 1: Marks.

0111 0010 = **0x72** = 7*16+2 = **114**

0011 0101 = **0x35** = 3*16+5 = **53**

1100 1101 = **0xCD** = -128 + 64+13 = **-51** (WRONG 205)

1111 1111 = **0xFF** = **-1**

Even though it was stated in bold that the integers were signed, many students missed that point.

2. **(4 pt)** Give the binary and hexadecimal representations of the following decimal numbers.

127 = **0x7F** = **0111 1111**

-128 = **0x80** = **1000 0000**

-1 = **0xFF** = **1111 1111**

15 = **0x0F** = **0000 1111**

There were some confusion in putting signs on the binary or hexadecimal representations, which we never did during the course. One point per correct line, or one point if all hexadecimal or decimal numbers were correct and the other column all wrong.

Exercise2: We want to store the integer 0xdeadbeef (32 bits) in memory. This will occupy 4 consecutive bytes.

1. **(1 pts)** Where in memory will each byte (in hexadecimal) be stored if a computer is using the little endian representation?

Address	0	1	2	3
Bytes	ef	be	ad	de

2. **(1 pts)** Where in memory will each byte (in hexadecimal) be stored if a computer is using the big endian representation?

Address	0	1	2	3
Bytes	de	ad	be	ef

Exercise3: Write C expressions, in terms of a variable `x`, for the following values. Your code should work for any word size $w \geq 8$. For reference, we show the result of evaluating the expressions for `x=0x87654321`, with $w = 32$. Use hexadecimal for the constants that you will need.

1. **(2 pts)** The least significant byte of `x`, with all other bits set to 0. [0x00000021].
   `x & 0xFF`
2. **(2 pts)** The least significant byte set to all 1s, and all other bytes of `x` left unchanged. [0x876543FF].
   `x | 0xFF`
   Small C errors such as || instead of | were tolerated. Mistakes on confusing bytes and bits were not.

Exercise4: Let us interpret an array of integers as an array of bits. Let us assume integers are 32 bits wide. We want to toggle bit `n` in that array, i.e., turn a 1 to a 0 and vice-versa. For example toggle bit 37 means toggle the 5^{th} bit of the second integer.

```
void toggle_bit (unsigned int *bits, unsigned int n)
{
  unsigned int pos   = n / (8*sizeof(int));
  unsigned int shift = n % (8*sizeof(int));
  bits[pos] ^= 1 << shift;
}
```

Listing 1: Function that toggles bit `n` in an array of bits.

1. **(5 pts)** Fill in the function in listing 1 without using any *if-statement*. Hint: use the xor operator `^`. (Bonus +1 pt) Do not hard-code the assumption that integers are on 32 bits.
   The bonus is obtained by using $8 * sizeof(int)$ *instead of 32.*

Exercise5: 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 2.

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

Listing 2: 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 > 0) || (x-1 < 0)`
   False. `INT_MIN <= 0 && INT_MIN-1 >= 0`
2. **(2 pts)** `(x < 0) || (-x <= 0)`
   True. `INT_MIN..-1 < 0` **and negations of** `0..INT_MAX` **do not overflow.**
3. **(2 pts)** `(x > 0) || (-x >= 0)`
   False. `INT_MIN <=0 && -INT_MIN < 0`
   A right answer needs some reasonable explanation to get full points.

Exercise6: In the C language right shifts are performed arithmetically for signed values and logically for unsigned values.

1. **(1 pt)** What is the difference between arithmetic and logical right shifts? Be precise and say which shift does what.
   Arithmetic shifts keep the sign bit, not the logical shifts.
   Any equivalent explanation that made sense was accepted.

Exercise7: In the Java language you have only signed integers.

1. **(Bonus +1 pt)** How do you do arithmetic and logical right shifts in Java?
   Arithmetic shifts: `x >> n`, logical shifts: `x >>> n`.
   Answers that did not specify which one was which were not accepted. In rare instances, other answers that made sense were also accepted.

Exercise8: 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)** `x == (int)(double) x`
   True. Int to double is exact, back to int preserves x.
2. **(2 pts)** `x == (int)(float) x`
   False. Int to float is not exact, e.g., `INT_MAX`.
3. **(2 pts)** `f == (float)(double) f`
   True. Float to double is exact, back to float preserves f.
4. **(2 pts)** `(f+d)-f == d`
   False. Addition is not associative for double or float, e.g. `d=1e100,f=1e-50`.

Exercise9: 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.

+	int	float
commutative	**Y**	**Y**
associative	**Y**	**N**

Any consistent way of answering (correctly) was accepted. Ticking, marking + - . . . . Unfortunately very few got this right even though it is VERY important. Most of you do not know what associativity and commutativity mean.

3 Program Encodings (28)

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*.
Exercise10: Assume the following values are stored at the indicated memory addresses and registers:

Address	Value	Register	Value
0x100	0xFF	%eax	0x100
0x104	0xAB	%ecx	0x1
0x108	0x13	%edx	0x3
0x10C	0x11		

1. **(3 pts)** Fill in the following table showing the byte values for the indicated operands. As a recall the general format for addressing is D(B,I,S) where D is the displacement (0 if

missing), B the base address (0 if missing), I (0 if missing) the index, and S the size for the offset (1 if missing). The address is then D+B+I*S.

Operand	Value
4(%eax)	**0xAB**
9(%eax,%edx)	**0x11**
0xFC(,%ecx,4)	**0xFF**

Not so many got all of them right even though the syntax was fully explained in the question.

Exercise11: 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 3. 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).

```
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
movl       %eax,  (%edx)
movl       %ebx,  (%ecx)
movl       %esi,  (%edi)
movl       %ebp,   %esp
pop        %ebp
ret
```

Listing 3: 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`.
That question was not at the low level detail of move this here and copy that there, but what the instructions actually do. Full points for all of them are obtained when mentioning the frame, of local stack, or something similar, otherwise -1 point if you were stuck at the low level details.

1. **(1 pt)** What is the register `%ebp` used for?
   To store the current frame pointer (base pointer).
2. **(2 pt)** What do the first `pushl` and `movl` instructions do?
   Save the previous frame, set the current frame to the current stack pointer.
3. **(2 pt)** What do the last `movl` and `popl` instructions do?
   Restore the stack pointer and the previous frame.
4. **(6 pts)** Write C code for `decode` that will have an effect equivalent to the assembly code of listing 3.
   One solution is to take names directly from the assembly. One could use int x, y, z too. There was a typo for zp, this was taken into account. You should dereference at least xp and yp and get the right decoding. Some of you tried it directly, e.g., `*xp = *zp; *yp = *xp; *zp = *yp;`, which is wrong because of data dependencies. You need at least one temporary variable.

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

```
{
  int ebx = *yp;
  int esi  = *zp;
  int eax = *xp;

  *yp = eax;
  *zp = ebx;
  *xp = esi;

  /* new *xp = old *zp *
   * new *yp = old *xp *
   * new *zp = old *yp */
}
```

Listing 4: Your implementation of `decode`.

5. **(4 pts)** Suppose that the function is called with the values of `%eax`, `%esi`, and `%edi`, respectively, for `xp`, `yp`, and `zp`. The assembly code to call the function is

```
pushl %edi
pushl %esi
pushl %eax
call decode
```

Suppose that `%esp=0x120` before calling `decode`. Fill in the written values in the stack after having executed the instructions to call the function and up to line 2 of listing 3.

Notes: You do not have to use every line in the table. To write the value of, e.g., `%eax`, write `%eax` in the corresponding memory cell (of 32 bits). Be careful with `call`, you may want to write in plain English what you want to put in the stack.

Hint: You can cross-check with listing 3 to make sure you get it right.

Even though there was a hint for cross-check, very few made it. There was one possible ambiguity on how to interpret "call the function", as the C call (before the sequence push etc...) or at the actuall assembly call. Two solutions are therefor provided. The important point that was primarily missed was that the stack grows downward. Then you forgot the return address coming from the call instruction. Finally you placed it wrong, but that could have been crossed-checked (see hint).

Address	Memory	Ambiguity - acceptable
0x134		
0x130		
0x12C		
0x128		edi
0x124		esi
0x120		eax
0x11C	edi	return address
0x118	esi	ebp
0x114	eax	
0x110	return address	
0x10C	ebp	

6. **(1 pt)** What is the new value of `%esp`?
   0x10C. *Ambiguity: if "before calling decode" means "before call decode" instead of the C call that starts with the ASM pushl %edi, then* **0x118**.

Exercise12: Conditional jumps are very common in programs. If we consider the code in listing 5 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.

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

Listing 5: Implementation of `absdiff`.

```
push      %ebp
mov       %esp,    %ebp
movl   8(%ebp),   %edx   ;  Get x
movl  12(%ebp),   %eax   ;  Get y
cmpl       %eax,  %edx   ;  Compare x and y
jge    GE                ;  jump if x >= y
subl       %edx,  %eax   ;  result = y - x
jmp    END
GE:
subl       %eax,  %edx   ;  x = x - y
movl       %edx,  %eax   ;  result = x
END:
mov       %ebp,    %esp
pop       %ebp
ret
```

Listing 6: Your assembly code for absdiff using conditional jumps, jge solution.

```
push       %ebp
mov        %esp,    %ebp
movl    8(%ebp),  %edx   ;  Get x
movl   12(%ebp),  %eax   ;  Get y
cmpl       %eax,  %edx   ;  Compare x and y
jl    LT                 ;  jump if  x < y
subl       %eax,  %edx   ;  x = x − y
movl       %edx,  %eax   ;  result  = x
jmp END
LT:
subl       %edx, %eax    ;  result  = y − x
END:
mov        %ebp,   %esp
pop        %ebp
ret
```

Listing 7: Your assembly code for absdiff using conditional jumps, jl solution.

Two solutions based on either `jl` or `jge` are possible. Semantically equivalent variants where both were used with more jumps than necessary were also accepted. Common mistakes were to invert arguments (ignored for points), add unnecessary parenthesis (not specified in the question), or forget jumps.

1. **(6 pts)** Write the assembly code corresponding to this function using conditional jumps. You may want to use the instructions `cmpl`, `jge`, `jl`, `subl`, and `jmp` (although not necessarily all of them). The result of the function should be in `%eax` upon returning. You do not need to do memory access, though it is possible to get an acceptable solution that does it. Hint: this is may easier if you write pseudo-code or C using goto statements before you write your answer in assembly.

   `cmpl Y, X` compares `X` and `Y` and sets flags accordingly. To simplify, we'll consider only the right conditional jumps.
   `jge LABEL` jumps if `X >= Y` holds when comparing `X` and `Y`.
   `jl LABEL` jumps if `X < Y` holds when comparing `X` and `Y`.
   `subl X, Y` computes `Y = Y - X`.
   `jmp LABEL` jumps without condition.

Exercise13: Let us consider the structure declared as

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

1. **(3 pts)** Fill-in the assembly code in listing 8 that implements the function

   `int readrec(struct rec* p);`

   that should return `p->a[p->i+p->j]`. 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     (%edx), %eax          ;  eax = i
addl    4(%edx), %eax          ;  eax += j
movl    8(%edx,%eax,4),%eax ;  result
```

```

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

Listing 8: Your assembly code for returning `p->a[p->i+p->j]`.

That question was not taken so often though the offsets in the structure are easy to find. Full points were given even in the case of small mistakes. Partial points were given if you showed some understanding of the offsets, i.e. where data are located.

4 Y86 (28+6)

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

Exercise14: For these questions we will consider the simplified CPU model of the Y86. Its processing is separated into different stages as shown in table 2 for two instructions. The stages are using a restricted set of hardware registers (rA, rB, valA, . . .) to do the computations.

1. **(6 pts)** What do each of the computation stages do? The table given as an example is only an example. The question is about the meaning of the stages. You should not refer to valA, valB, etc . . .

   Fetch: **Fetch an instruction, its operands, and compute address of next instruction. Immediate arguments are read here.**
   Decode: **Read the values of the register operands.**
   Execute: **Execute an arithmetic instruction if needed and update the condition flags.**
   Memory: **Read from or write to memory.**
   Write back: **Update the register file.**
   PC update: **Update the program counter for the next instruction.**

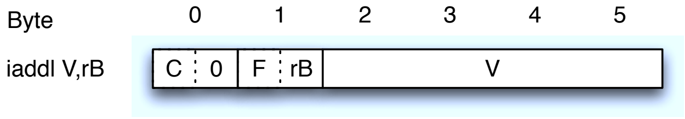
   You missed half of the fetch stage explanation most of the time. If you miss that you update the ***registers*** *in the write back stage, you lose the point. For memory you forgot read or write, but still got the point if you got the other one. For decode it was about reading from the registers. This was an easy question, given the tables as example.*

2. **(2 pts)** In table 2 the PC update is the same for both instructions. Obviously this is not always the case. Give an instruction for which this is not the case and explain what this instruction does (no need to give the computation stages). You should point out the difference in the PC update. The solution to that question is not unique, it is enough to pick one solution.

Conditional jumps jump to different target addresses depending on flags. The update is conditional.

3. **(Bonus +2 pts)** Give another solution and explain it. **The return instruction update the PC with a return address in the stack. The update depends on memory (the stack).**
Unexpected answers, such as, the special instruction `halt`, were accepted too.

Exercise15: It is common to add a constant value to a register. Unfortunately the instruction set of the Y86 requires first using an `irmovl`[2] instruction to set a register to the constant, and then an `addl` instruction to add this value to the destination register. Suppose we want to add a new instruction `iaddl` with the following format:



This instruction adds the constant value `V` to register `rB`. The values of the constants, i.e., C, 0, and F, in the encoding are irrelevant for the purpose of the exercise.

1. **(Bonus +1)** The value F is special. What is it used for? **It is used to fill in the generic format of the instruction and means no register argument.**
2. **(8 pts)** Describe the computations performed to implement this instruction.

Stage	`iaddl` V,rB

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

Decode	valB ← R[rB]
Execute	valE ← valB + valC set CC
Memory	
Write back	R[rB] ← valE
PC update	PC ← valP

You forgot `set CC` though this was in one of the examples given. In fact this should have been mentioned in the description of the stages.

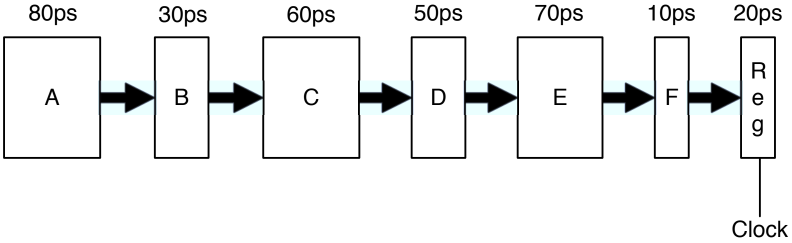
3. **(1 pt)** Why would a sequential implementation of our computation stages be inefficient?
Because the different stages are not active all the time of the execution (and the processor needs to wait for the completion of all of them for every instruction).
Any answer that showed the serial aspect of the execution was accepted.

Exercise16: To improve performance, a *pipeline* architecture is used instead of a sequential one.

1. **(1 pts)** How does it improve performance?
It improves the utilization of the computation stages (and several instructions can be executing at the same time).
Any answer that showed the parallel aspect of the execution was accepted.
2. **(2 pts)** Name two problems that need to be solved with pipelines.
Data dependencies, conditional jumps, load hazards, branch prediction...
Other answers that made sense, such as, added latency or balancing timing between the stages were also accepted.

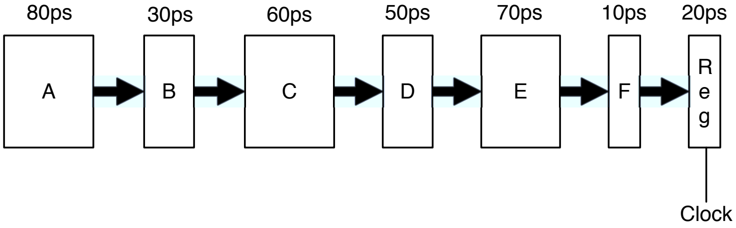
Exercise17: Suppose we analyze some combinational logic and determine that it can be separated into a sequence of six blocks, named A to F, having delays of 80, 30, 60, 50, 70, and 10 ps, respectively, illustrated as follows:

[2] Move immediate (constant) to register.

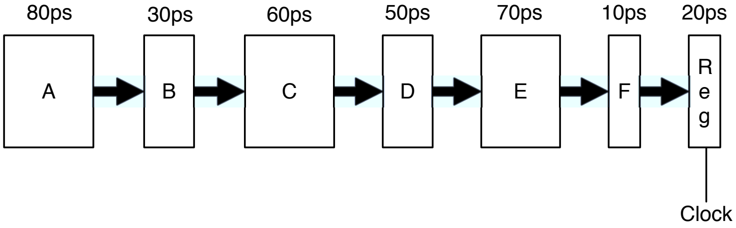


We can create pipelined versions of this design by inserting pipeline registers between pairs of these blocks. Different combinations of pipeline depth (how many stages) and maximum throughput arise, depending on where we insert the pipeline registers. Assume that a pipeine register has a delay of 20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

1. **(2 pt)** On the figure, you notice a clock connected to the output (hardware) register "Reg". Explain.
   Register updates are synchronous and happen when the clock signal rises.
2. **(3 pts)** Inserting a single register gives a two-stage pipeline. Where should the register be inserted to maximize throughput? What would be the throughput[3] and latency[4]? **C-D**

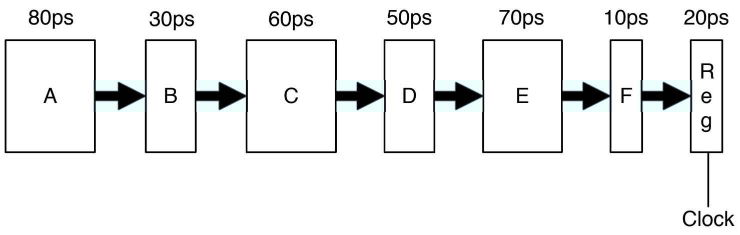
   Throughput: $1/(190 * 10^{-12})$ instructions/s
   Latency: 340ps
3. **(3 pts)** Where should two registers be inserted to maximize the throughput of a three-stage pipeline? What would be the throughput and latency? **B-D,D-E**

   Throughput: $1/(130 * 10^{-12})$ instruction/s
   Latency: 360ps

   Surprisingly, very few of you know what latency and throughput/bandwidth are. I thought I was clear on that, my mistake. The throughput, instruction per second, comes from the slowest part of the pipeline that acts as a bottleneck (counting the register too, as said in the question). The latency is the total time for executing one instruction, it's the time to completion.
4. **(Bonus +3 pts)** Where should three registers be inserted to maximize the throughput of a four-stage pipeline? What would be the throughput and latency? **A-B,C-D,D-E**

[3] Give the expression for the throughput, e.g., $\frac{1}{320*10^{-12}}$ instructions per seconds (IPS).
[4] Do not forget the (inserted registers) taking 20ps each.



Throughput: $1/(110 * 10^{-12})$ instructions/s
Latency: 380ps

5 RAM and Memory Hierarchy (15)

Exercise18: Different RAM technologies are used together in computers.

1. **(1 pt)** What is SRAM memory and where is it used?
   Static RAM (random access memory), used in caches.
2. **(1 pt)** What is DRAM memory and where is it used?
   Dynamic RAM, used for main memory.
3. **(3 pts)** Give three differences between SRAM and DRAM.
   SRAM consumes more power, has higher bandwidth, lower latency, and costs more than DRAM.
   Obvious answers, e.g., they have different names, were not accepted.

Exercise19: The principle of locality is the fundamental reason why modern computers work in practice.

1. **(2 pts)** What is temporal locality?
   Reuse of the same data later.
2. **(2 pts)** What is spatial locality?
   Use of (different) data closely located, typically successive reads or writes.
   *Most of you had the intuition for the next question, i.e., locality, but you do not know exactly what it means. The temporal aspect (in time) is to access **later** the same thing. The spatial aspect (space, address) is to access something **close**.*

```
int sum1(int a[M][N]) {
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];

  return sum;
}

int sum2(int a[M][N]) {
  int i, j, sum = 0;

  for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
      sum += a[i][j];

  return sum;
}
```

Listing 9: Different implementations of sums.

3. **(2 pts)** If we consider the implementations of listing 9 that compute the sums of all elements in a matrix in two different ways, which one is better and why?
   The better one is sum1 because it uses spatial locality.
4. **(4 pts)** Modern architectures implement a *memory hierarchy.* Illustrate it with a drawing[5]. Give two main characteristics of the type of memory when you go to the top or the bottom of the hierarchy?
   Pyramid with: register > cache > main memory > disk.
   Characteristics: speed, size, cost.
   Many of you did not give the characteristics as asked and lost half the points.

6 Cache Memory (15)

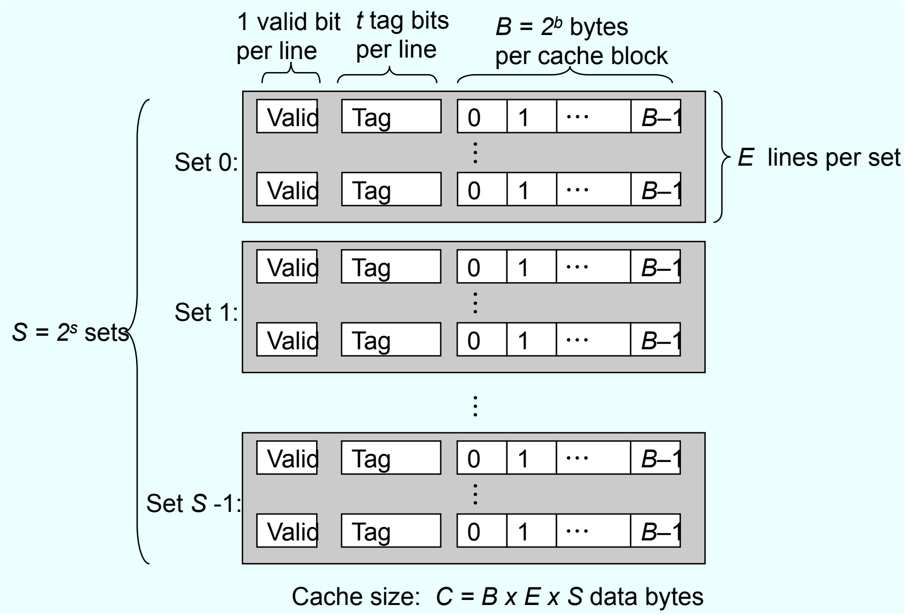


Figure 1: General organization of caches.

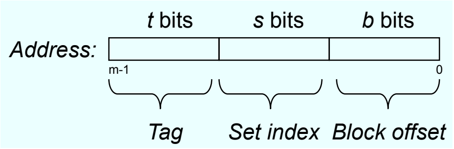


Figure 2: Physical addressing.

Exercise20: Caches are organized in general into sets of lines where each line stores a block of bytes as shown in Figure 1. A cache is determined by its parameters S, E, and B.
All the information for solving the subsequent questions was in the figure.

1. **(1 pt)** What does it mean to have an N-way associative cache?
   To have N lines per set.
2. **(1 pt)** What is a fully associative cache?
   A cache with one set.

[5] Use at least 4 different types of levels in your illustration. You can use three levels of cache but they count as one.

3. **(1 pt)** What is a direct mapped cache?
   A cache with one line per set.
4. **(1 pt)** What is the valid bit used for?
   To mark the data to be valid or not.
   This question was obvious and could have been removed. You managed to get it wrong though, by mixing it with the role of the tag.
5. **(1 pt)** The addresses are split with a sequence of t, s, and b bits for the different field. Why not use the sequence t,b,s instead?
   That would break spatial locality.
   Given an address, we can choose these bits as we want, if we want to. This would be not very smart but that's the point of the question. Any similar answer was accepted, as well as the common bits used in virtual and physical addresses, which could have been more obvious.
6. **(3 pts)** The following table gives the parameters for a number of different caches. For each cache, determine the number of cache sets (S), tab bits (t), set index bits (s), and block offset bits (b). In the table m is the number of physical address bits. Hint: rewrite C, B, and E as powers of two.

m	C	B	E	S	t	s	b
32	1024	4	1	$2^8 = 256$	**22**	**8**	**2**
32	1024	8	4	$2^5 = 32$	**24**	**5**	**3**
32	1024	32	32	$2^0 = 1$	**27**	**0**	**5**

Exercise21: 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).

1. **(2 pts)** Indicate which bits would be used to determine the cache block offset (CO), the cache set index (CI), and the cache tag (CT).
   CT: 8, CI: 3, CO: 2.

Exercise22: To benchmark memory systems, an experiment known as the memory mountain can be done. Figure 3 shows the result obtained on a Core i7.

1. **(2 pts)** Describe the memory mountain experiment.
   Read or write some amount of data (2k, 4k, . . .) repeatedly in strided access pattern (1, 2, . . .), and measure throughput (or bandwidth).
   This question was about a simple description of the experiment, not the result (shown in the graph).
2. **(2 pts)** Explain the ridges of temporal locality and the slopes of spatial locality.
   Ridges: when the data set does fit in the cache (L1, L2, L3), performance drops since we go down in the memory hierarchy. Slopes: when the strides increase, we lose in spatial locality, which degrades performance.
3. **(1 pt)** At stride 1, the throughput is kept flat through L2 and L3, and stays high even when going through memory. Explain.
   This comes from prefetching.
   *The point of this question was missed. It should have been bonus. The point was that the throughput was kept almost **flat** through the different levels of memory, so this is does not come from temporal locality (data do not fit in the cache), but is related to spatial locality, though there is something special to go through all levels like this. That's prefetching.*

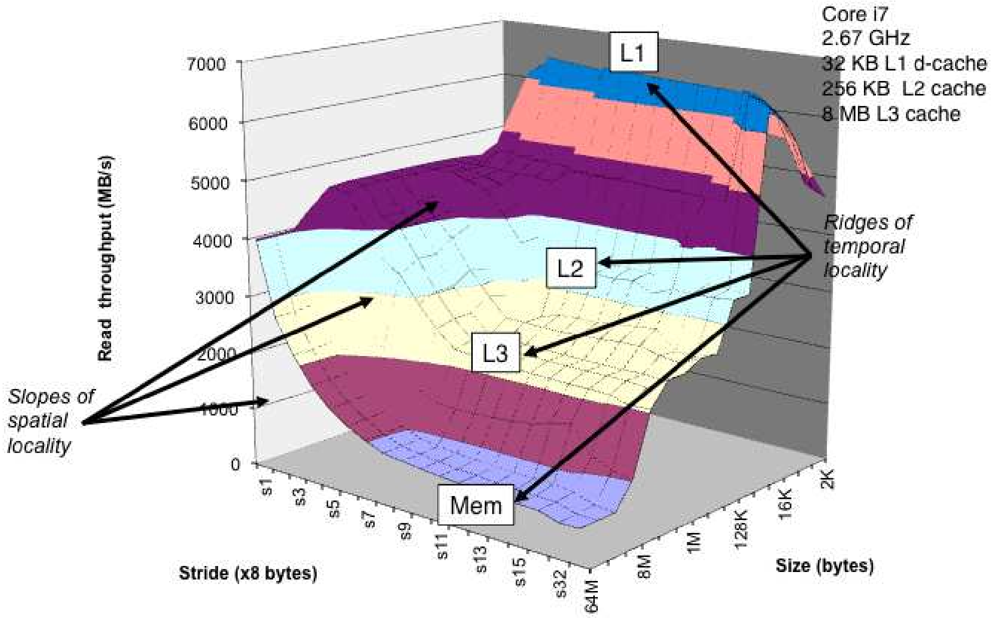


Figure 3: The memory mountain for the Core i7.

7 Virtual Memory (21+1)

Exercise23: 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. **(2 pts)** Name two benefits of using virtual addressing.
   Abstraction for programmer, protection between processes, dynamic/flexible use of resources...
   Any other sensible answer was accepted.
2. **(4 pts)** Given a 32-bit virtual address space and a 24-bit physical address, determine the number of bits in the virtual page number (VPN), virtual page offset (VPO), physical page number (PPN), and physical page offset (PPO) for the following page sizes P. Hint: rewrite P as a power of 2.

P	No. VPN bits	No. VPO bits	No. PPN bits	No PPO bits
1 KB	22	10	14	10
2 KB	21	11	13	11
4 KB	20	12	12	12
8 KB	19	13	11	13

Exercise24: Figure 4 shows a simple memory system with a 4-way associative TLB with 16 entries and a direct mapped cache with 16 lines and 4 bytes per line (block size).

1. **(1 pt)** What does TLB stand for?
   Translation look-aside buffer.

This is one of the rare important abbreviations to know, given the central and vital role of the TLB in virtual memory systems.

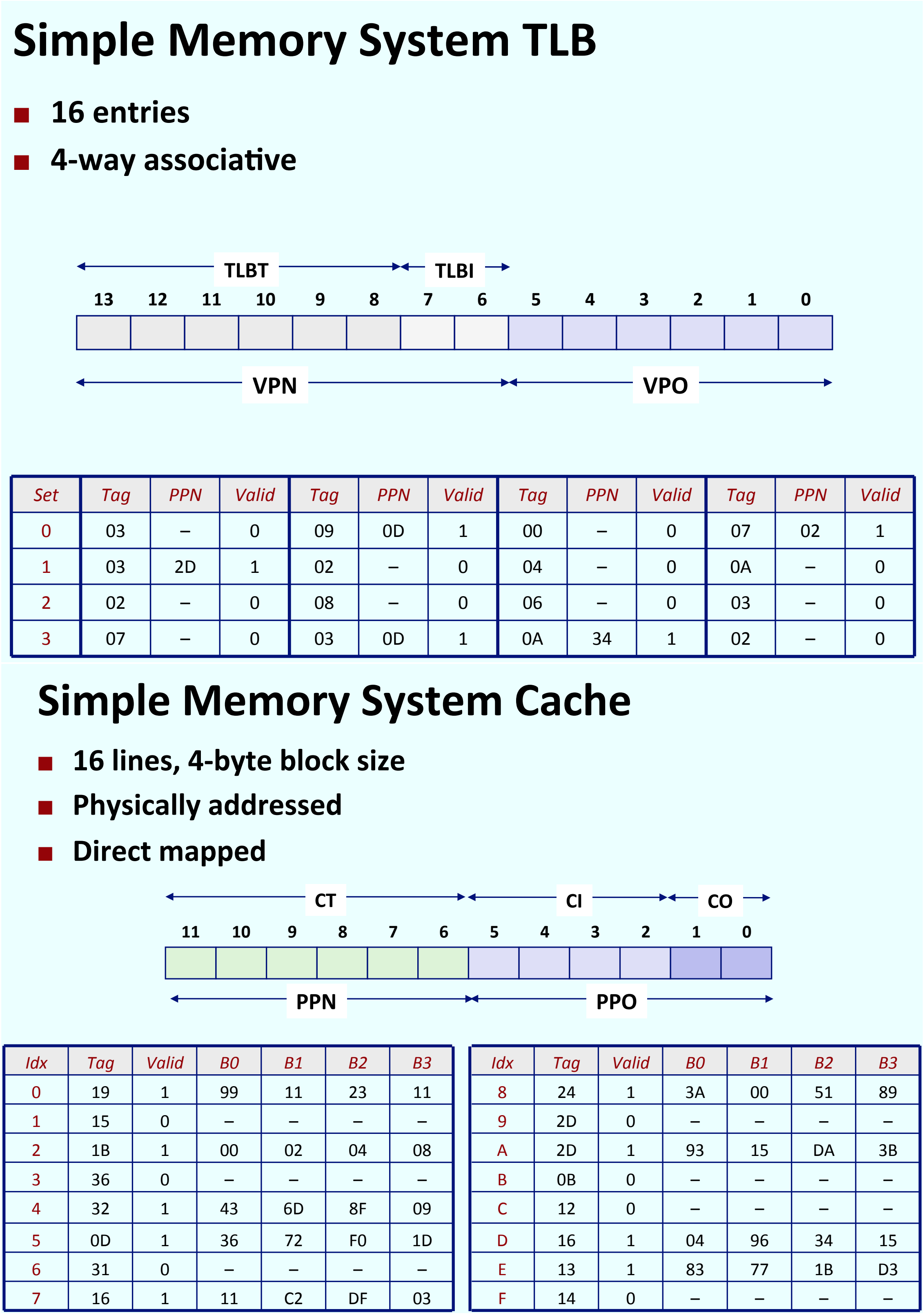
2. **(1 pt)** What is the TLB used for?
   To cache some entries of the mapping VPN to PPN.
   The TLB is nothing but a cache. It is a special one that sits before the cache (that is addressed with physical addresses).
3. **(1 pt)** Why is important to have a TLB?
   Because the translation virtual to physical address has to be done for every memory access.
   You got the point for any other sensible answer though none of you got this extremely important point that I hammered on the board during the lecture.
4. **(Bonus +1 pt)** As you notice, CI+CO matches PPO that matches VPO. This is no coincidence. How does the memory system exploit this?
   The hardware can start a cache lookup before it knows the physical address.
   Some of you managed to give other sensible answers and got the point.
5. **(3x4 pts)** Translate the virtual addresses `0x03D5`, `0x0B8E`, `0x0021` into their physical addresses according to Figure 4. Indicate if you have hits, misses, or page faults. If you cannot continue with the information in the table, indicate what the memory system will do. Use the physical address to access the corresponding byte in the cache. Indicate if you have a cache hit or not. Indicate what the memory system will do if you cannot read the data. Answer by clearly enumerating the different stages.

   0x03D5: TLBT=3, TLBI=3, VPN=0xF, VPO=0x15.
   TLB hit PPN=0x0D, valid.
   CT=0D, CI=5, CO=1.
   Cache hit, B1=0x72.

   0x0B8E: TLBT=0xB, TLBI=2, VPN=0x2E, VPO=0x0E.
   TLB miss. Page fault.
   CT=?, CI=3, CO=2, both invalid.

   0x0021: TLBT=0, TLBI=0, VPN=0, VPO=0x21.
   TLB hit, invalid.
   CT=?, CI=8, CO=1.
   Cache hit if CT is 0x24.

   This was done during the lecture. I changed the block offset so you had to read the 2nd byte instead of the 1st, that was all.



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

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