Written Exam in HPPS—Theory Part
Example

January ?, 2021

Preamble

Solution

Disclaimer: The reference solutions in the following range from exact results to sketch solutions; this is entirely on purpose. Some of the assignments are very open, which reflects to our solutions being just one of many. Of course we try to give a good solution and even solutions that are much more detailed than what we expect you to give. On the other hand, you would expect to give more detailed solutions that our sketches. Given the broad spectrum of solutions, it is not possible to directly infer any grade level from this document. All solutions have been written in the in-lined space with this colour.

This is the *example* exam set for the (estimated) 3 hour theory part in HPPS, B2-2020/21. This document consists of 14 pages excluding this preamble; make sure you have them all. Read the rest of this preamble carefully. Your submission will be graded as a whole, on the 7-point grading scale, with external censorship.

- You can answer in either Danish or English.
- Remember to write your exam number on all pages.
- You do not have to hand-in this preamble.

Expected usage of time and space

The set is divided into sub-parts that each are given a rough guiding estimate of the size in relation of the entire set. However, your exact usage of time can differ depending on prior knowledge and skill.

Furthermore, all questions includes formatted space (lines, figures, tables, etc.) for in-line answers. Please use these as much as possible. The available spaces are intended to be large enough to include a satisfactory answer of the question; thus, full answers of the question does not necessarily use all available space.
If you find yourself in a position where you need more space or have to redo (partly) an answer to a question, continue on the backside of a paper or write on a separate sheet of paper. Ensure that the question number is included and that you in the in-lined answer space refers to it; e.g. write "*The [rest of this] answer is written on backside of/in appended page XX.*"

For the true/false and multiple-choice questions with one right answer give only one clearly marked answer. If more answers are given, it will be interpreted as incorrectly answered. Thus, if you change your answer, make sure that this shows clearly.

Exam Policy

This is an *individual*, open-book exam. You may use the course book, notes and any documents printed or stored on your computer and other device, but you may not search the Internet or communicate with others to answer the exam.

Errors and Ambiguities

In the event of errors or ambiguities in the exam text, you are expected to state your assumptions as to the intended meaning in your answer. Some ambiguities may be intentional.

1 Data representation and programming languages (about 25 %)

1.1 Numeric representations (about 10 %)

Numeric representations, 1.1.1:

Using a base-7 number system, having numbers in the range $[0-6]$, compute the following expressions:

1. $16_{b7} + 22_{b7}$
2. $555_{b7} + 123_{b7}$
3. $6_{b7} * 3_{b7}$

41

1011

24

Numeric representations, 1.1.2:

Answer the following questions with a 16-bit floating point number format that:

- has no sign bit (i.e. can only represent positive numbers)
- uses a 6 bit *signed* exponent
- uses a 10 bit *unsigned* significand
- Use significand as a fraction of the maximum significand

1. What is the largest possible number?
2. What is the smallest possible non-zero number?
3. What is a valid bit pattern for the number 1.0?
4. Are there more than one valid bit pattern for the number 1.0? If yes, provide an example.

$\frac{1023}{1024} * 2^{31} = 2.145.386.496$

$\frac{1}{1024} * 2^{-32} = 2,273736754 * 10^{-13}$

1536, $512/1024 * 2^1 = 11000000000$

Yes, there are 10:

1536, $512/1024 * 2^1 = 11000000000$,

2304, $256/1024 * 2^2 = 100100000000$,

3200, $128/1024 * 2^3 = 110010000000$,

4160, $64/1024 * 2^4 = 1000001000000$,

5152, $32/1024 * 2^5 = 1010000100000$,

6160, $16/1024 * 2^6 = 1100000010000$,

7176, $8/1024 * 2^7 = 1110000001000$,

8196, $4/1024 * 2^8 = 10000000000100$,

9218, $2/1024 * 2^9 = 10010000000010$,

10241, $1/1024 * 2^{10} = 10100000000001$

Numeric representations, 1.1.3:

For this question you are working with 7-bit signed and unsigned numbers. The signed numbers are using two's complement.

Fill in the blanks in this table.

Binary	Hex	Unsigned decimal	Signed decimal
0b001 1000	0x18	24	24
0b010 1010	0x2A	42	42
0b111 1000	0x78	120	-8
0b100 1000	0x48	72	-56

1.2 Array representation (about 10 %)

Array representation, 1.2.1: In this task you must rewrite multidimensional array indexes to flat array indexes for arrays of various shapes. For each array and index, provide the flat index for when the array is in row-major order and in column-major order, respectively.

- **Array size:** 3×9
  Index: $(2, 4)$
  Row-major flat index: $2 * 9 + 4 = 22$
  Column-major flat index: $4 * 3 + 2 = 14$
- **Array size:** $1 \times 2 \times 3$
  Index: $(0, 1, 2)$
  Row-major flat index: $(0 * 2 * 3) + (1 * 3) + (2) = 5$
  Column-major flat index: $(2 * 1 * 2) + (1 * 1) + (0) = 5$

Array representation, 1.2.2: Consider a sequence of 10 values stored consecutively in memory:

a	b	c	d	e	f	g	h	i	j

Write all possible two-dimensional arrays that could be represented by this sequence in memory. All dimensions must be at least one, and all elements a—j must occur in the array once:

Row-major order:

$$\begin{bmatrix} a & b & c & d & e & f & g & h & i & j \end{bmatrix} \quad \begin{bmatrix} a & b & c & d & e \\ f & g & h & i & j \end{bmatrix} \quad \begin{bmatrix} a \\ b \\ c \\ d \\ e \\ f \\ g \\ h \\ i \\ j \end{bmatrix}$$

Column-major order:

$$\begin{bmatrix} a & c & e & g & i \\ b & d & f & h & j \end{bmatrix} \quad \begin{bmatrix} a & f \\ b & g \\ c & h \\ d & i \\ e & j \end{bmatrix}$$

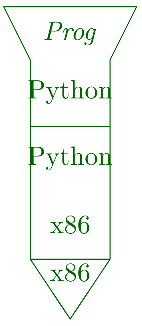
1.3 Tombstone diagrams (about 5 %)

Suppose that you have access to the following:

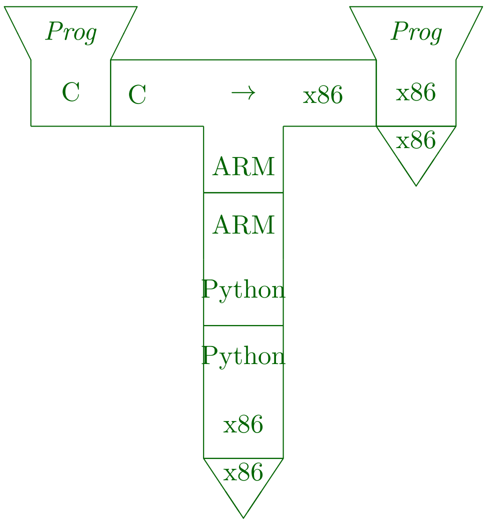
- an x86 machine,
- a compiler from C to x86 written in ARM
- an ARM interpreter written in Python
- a Python interpreter written in x86

Show Tombstone diagrams for the following cases:

1. How to execute a Python program:



2. How to execute a C program:



2 Caching and virtual memory (about 25 %)

2.1 Locality (about 5 %)

Consider the following program:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
    A[i*N+j] = A[i*N+j] * B[j*N];
```

Does it exhibit temporal and/or spatial locality? Can you optimise it to exhibit better locality?

The accesses to `A` exhibit spatial locality because each access is to an element adjacent to the one accessed previously. The `B` accesses *may* exhibit temporal locality if `N` is small. We can gain spatial locality by copying the part of `B` into a separate array before the loop:

```
double B_slice[N];
for (int i = 0; i < N; i++)
  B_slice[i] = B[i*N];
for (int i = 0; i < N; i++)
  for (int j = 0; j < N; j++)
   A[i*N+j] = A[i*N+j] * B_slice[j];
```

2.2 Caches (about 10 %)

Assume a byte-addressed machine with 8-bit addresses. The machine is equipped with a single L1-cache that is direct mapped and write-allocate, with a block size of 8 bytes. Total size of the data cache is 16 bytes.

Caches, 2.2.1: For each bit in the table below, indicate which bits of the address would be used for

- block offset (denote it with `O`),
- set index (denote it with `S`), and
- cache tag (denote it with `T`).

T	T	T	T	S	O	O	O
7	6	5	4	3	2	1	0

Caches, 2.2.2: Consider we are running the following matrix transpose function, transposing a 2 × 2 array, on the machine above:

```
void transpose(int dst[2][2], int src[2][2]) {
  for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
      dst[i][j] = src[j][i];
    }
  }
}
```

It is furthermore given that:

- The `src` array starts at address 0x40 and the `dst` array starts at address 0x50.
- Accesses to the `src` and `dst` arrays are the only sources of read and write misses, respectively. i and j are allocated in registers.
- The cache is initially cold and uses LRU-replacement.

For each element `row` and `col`, indicate whether each access to `src[row][col]` and `dst[row][col]` is a hit (`h`) or a miss (`m`). For example, reading `src[0][0]` is a miss as the cache is cold. Possibly explain your answer.

`dst` array	col 0	col 1
row 0	`m`	`h`
row 1	`m`	`m`

`src` array	col 0	col 1
row 0	`m`	`m`
row 1	`m`	`m`

The block size gives that two elements of the array is held in each cache line. The addresses are aligned such that that they hit same set, but there are two entries in each set. `src` are read column-wise and will thus have constant misses, while `dst` are written row-wise and will only have a miss in first write, but a set conflict in the last write.

Detailed, the order of addresses are `0x40, 0x50, 0x48, 0x54, 0x44, 0x58, 0x4C, 0x5C`.

The address `0x58` and `0x4C` have the same set bit.

2.3 Virtual Memory (about 10 %)

Virtual Memory, 2.3.1: Consider a system with the following properties:

- Memory is byte-addressed.
- Virtual addresses are 15 bits wide.
- Physical addresses are 14 bits wide.
- The page size is 32 bytes.
- The TLB is 3-way set associative with four sets and 12 total entries. Its initial contents are:

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	07	0	09	0D	0	11	51	0
1	07	2D	1	12	51	1	00	00	1
2	0A	00	0	09	01	1	10	34	1
3	10	A1	0	03	0D	0	10	A0	1

- The page table contains 12 PTEs:

VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid	VPN	PPN	Valid
12	12	1	01	02	1	02	01	1	31	33	1
41	01	1	09	17	1	02	33	0	0B	0D	0
00	00	1	10	21	0	13	32	0	21	43	1

Note that all addresses are given in hexadecimal. In the following questions, you are asked, for various virtual addresses, to show the translation from virtual to physical addresses in the memory system just described. *Hint: there is one TLB hit, one page table hit, and one page fault (not necessarily in that order). This should help you double-check your work.*

Virtual address: `0x0019`

1. Bits of virtual address

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	1

2. Address translation

Parameter	Value
VPN	0
TLB index	0
TLB tag	0
TLB hit? (Y/N)	N
Page fault? (Y/N)	N
PPN	0

3. Bits of phys. (if any)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	0	1	1	0	0	1

Virtual address: `0x0853`

1. Bits of virtual address

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	0	0	0	0	1	0	1	0	0	1	1

2. Address translation

Parameter	Value
VPN	42
TLB index	2
TLB tag	10
TLB hit? (Y/N)	Y
Page fault? (Y/N)	N
PPN	34

3. Bits of phys. (if any)

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	1	1	0	1	0	0	1	0	0	1	1

Virtual address: `0x1337`

1. Bits of virtual address

14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	0	0	1	1	0	0	1	1	0	1	1	1

2. Address translation

Parameter	Value
VPN	99
TLB index	1
TLB tag	26
TLB hit? (Y/N)	N
Page fault? (Y/N)	Y
PPN	

3. Bits of phys. (if any)

13	12	11	10	9	8	7	6	5	4	3	2	1	0

3 Computer networks (about 25 %)

3.1 Network properties (about 10 %)

Network properties, 3.1.1:

With the *network* layer from the OSI stack, answer the following questions, specific to the internet:

1. What is the responsibility of the *network* layer?
2. What adresses are used on the *network* layer?
3. What is the maximum number of addresses possible for the *network* layer?
4. How does the *network* layer handle package loss?

1. The network layer is responsible for routing packages between hosts.
2. The network layer uses IP addresses.
3. There are 2^{32} addresses for IPv4 and 2^{128} for IPv6.

The network layer does not provide any services for lost packages.

Network properties, 3.1.2:

For the TCP protocol, answer the following questions:

1. How does the *TCP protocol* handle package loss?
2. What is the purpose of sequence numbers in the *TCP protocol*?
3. What mechanism in the *TCP protocol* guards against network overloads?

1. The TCP protocol uses timers to detect lost packages and retransmit to recover.
2. Sequence numbers are used to ensure package are delivered in-order and without gaps.
3. The congestion control part of TCP throttles transmission rates if it detects network overload.

3.2 Programs on a network (about 15 %)

Programs on a network, 3.2.1: You are writing a program where you have concluded that for each 64-bit value transfered, you need to perform 600 floating point operations (FLOPs). The machine you are computing on, is listed as being able to compute with $10,000$ FLOPs pr second. You can send at most $2,000$ bytes in a package and the ISP guarantees a speed of $4,000$ bytes/s. Now compute the following:

1. How many FLOPs will be performed per package?
2. What is the maximum bandwidth the program can handle?
3. How many FLOPs would the machine need to be able to run the program at full bandwidth?

1. We can pack $2,000/8 = 250$ 64-bit values in a package, giving $250 * 600 = 150,000$ FLOPs.

2. We can handle at most $10,000/150,000 = 0.0667$ packages/s, or $0.0667 * 2000 = 133.33$ bytes/s

equivalent to appx. (1066 bits/s)

3. We can send $4000/8 = 500$ 64-bit values/s

which requires $500 * 600 = 300,000$ FLOPs/s for the program.

4 Concurrency and Parallelism (about 25 %)

4.1 Multiple Choice Questions (about 4 %)

In each of the following questions, you may put one or more answers. If needed use the lines to argue for your choices.

Multiple Choice Questions, 4.1.1: Which of the following operations are guaranteed to execute atomically in a multi-threaded program?

- ☐ **a)** `printf()`
- ☐ **b)** `write()`
- [✓] **c)** `pthread_mutex_lock()`
- ☐ **d)** `x = y` (when `x` and `y` are `int`)
- ☐ **e)** `exit(0)`
- ☐ **f)** `x++` (when `x` is `sig_atomic_t`)

4.2 Scaling with Amdahl's Law (about 3 %)

For this question we use Amdahl's Law to estimate speedup in latency. Suppose we have a program where 95% of the work is parallelisable. Assuming the rest can be fully parallelised without any overhead. Answer the following:

1. What is the speedup if we run it on a 4-processor machine?

3.48

2. What about with 128 processors?

17.41

3. What is the smallest number of processors that will give us a speedup of at least 8?

13

4. What is the highest achievable speedup, given an arbitrary number of processors?

20 (19 also acceptable)

4.3 Scaling with Gustafson's Law (about 3 %)

For this question we use Gustafson's Law to estimate speedup in latency. Suppose we have a program where 95% of the work is parallelisable. Assuming the rest can be fully parallelised without any overhead, and that the parallel workload is proportional to the amount of processors/threads we use:

1. What is the speedup if we run it on a 4-processor machine?

3.85

2. What about with 128 processors?

121.65

3. What is the smallest number of processors that will give us a speedup of at least 5?

6

4. What is the smallest number of processors that will give us a speedup of at least 30?

32

4.4 Parallelisation (about 15 %)

The following questions are about the C-like pseudocode below:

```
float A[2*M];
for (int i = 0; i < N; i++) {
    A[0] = N;
    for (int k = 1; k < 2*M; k++) {
        A[k] = sqrt(A[k-1] * i * k);
    }
    for (int j = 0; j < M; j++) {
        B[i+1, j+1] = B[i, j] * A[2*j  ];
        C[i,   j+1] = C[i, j] * A[2*j+1];
    }
}
```

4.4.1 (about 2 %)

Explain why in the code above, neither the outer loop (of index `i`) nor the inner loops (of indices `k` and `j`) are parallel.

Every `i`-iteration writes to the same elements of `A`. The `k`-loop has a RAW dependency on `A`. The `j`-loop has RAW dependency on `C`.

4.4.2 (about 3 %)

Explain why is it safe to privatise array A and show the code after privatisation.

Each `i`-iteration completely overwrites all of `A`, so no value is propagated from one iteration to the next

(that is, there are no RAW dependencies)

```
for (int i = 0; i < N; i++) {
    float A[2*M];
    A[0] = N;
    for (int k = 1; k < 2*M; k++) {
        A[k] = sqrt(A[k-1] * i * k);
    }
    for (int j = 0; j < M; j++) {
        B[i+1, j+1] = B[i, j] * A[2*j  ];
        C[i,   j+1] = C[i, j] * A[2*j+1];
    }
}
```

4.4.3 (about 5 %)

Once privatised, explain why is it safe to distribute the outermost loop across the `A[0] = N;` statement and across the other two inner loops. Perform the loop distribution and show the result, while remembering to perform array expansion for `A`.

This is safe because there are no dependency cycles.

```
float A[N][2*M];
for (int i = 0; i < N; i++) {
    A[i][0] = N;
}
for (int i = 0; i < N; i++) {
    for (int k = 1; k < 2*M; k++) {
        A[i][k] = sqrt(A[i][k-1] * i * k);
    }
}
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        B[i+1, j+1] = B[i, j] * A[i][2*j  ];
        C[i,   j+1] = C[i, j] * A[i][2*j+1];
    }
}
```

4.4.4 (about 5 %)

Use direction vectors to explain which loops in the resulting three loop nests are parallel.

The first loop nest has no dependencies, because the only data that it writes (`A`) is indexed with a distinct index for every iteration. So this nest is parallel.

The second nest has a RAW dependency on `A` with direction vector `[=,<]`, which means only the outer loop is parallel.

The third loop nest has no dependencies on `A`, because there are no writes. It has RAW dependencies on `B` with directions `[<, <]` and on `C` with `[=,<]`. Hence this nest is not parallel.