시스템프로그램 기말시험 (2020학년도 2학기, 12/17/2020)

학과		학번		학년		이름		

(1) Function **f1** has the following prototype, and gcc generates the following assembly code for the function. Fill in the C code of **f1** (in 1 expression).

```
double f1 (double w, int x, float y, long z)
{
   return _________________;
}
```

```
f1:
   vcvtsi2ss     %edi,%xmm2,%xmm2
   vmulss        %xmm1,%xmm2,%xmm1
   vunpcklps     %xmm1,%xmm1,%xmm1
   vcvtps2pd     %xmm1,%xmm2
   vcvtsi2sdq    %rsi,%xmm1,%xmm1
   vdivsd        %xmm1,%xmm0,%xmm0
   vsubsd        %xmm0,%xmm2,%xmm0
   ret
```

(2) Consider the following functions:

```
long min(long x, long y) { return x < y ? x : y; }
long max(long x, long y) { return x < y ? y : x; }
void incr(long *xp, long v) { *xp += v; }
long square(long x) { return x * x; }
```

The following three code fragments call these functions:

```
A. for (i = min(x, y); i < max(x, y); incr(&i, 1))
       t += square(i);
B. for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
       t += square(i);
C. long low = min(x, y);
   long high= max(x, y);
   for (i = low; i < high; incr(&i, 1))
       t += square(i);
```

Assume x equals 10 and y equals 50. Fill in the following table indicating the number of times each of the four functions is called in code fragments A~C.

	Code	min	max	incr	square
①	A				
②	B				
③	C				

⑶ Suppose we wish to write a function to evaluate a polynomial, where a polynomial of degree **n** is defined to have a set of coefficients a_0, a_1, a_2, ... , a_n. For a value **x**, we evaluate the polynomial by computing the equation 4.1 or 4.2.

$$a_0 + a_1 \cdot x + a_2 \cdot x^2 + \cdots + a_n \cdot x^n \quad (3.1)$$

$$a_0 + x \cdot (a_1 + x \cdot (a_2 + \cdots + x \cdot (a_{n-1} + a_n \cdot x) \cdots)) \quad (3.2)$$

These evaluations can be implemented by the following functions, **poly1** and **poly2**, having as arguments an array of coefficients **a**, a value **x**, and the polynomial degree **deg** (the value **n** in Equation 3.1 and 3.2). Answer the following questions.

```
double poly1(double a[], double x, long deg)
{
   long i;
   double result = a[0];
   double xpwr = x;
   for (i = 1; i <= deg; i++) {
      result += a[i] * xpwr;
      xpwr = x * xpwr;
   }
   return result;
}
```

```
double poly2(double a[], double x, long deg)
{
   long i;
   double result = a[deg];
   for (i = deg-1; i >= 0; i--)
      result = a[i] + x * result;
   return result;
}
```

① For degree **n**, how many additions and how many multiplications does the functions perform? (Answer for additions and multiplications on double data)

[poly1]
[poly2]

② On the reference machine, with arithmetic operations having the latencies of integer addition 1, integer multiplication 3, float/double addition 3, and float/double multiplication 5, what is the expected value of CPE for the two functions? Explain how these CPEs arise based on the data dependencies formed between iterations of the functions.

[poly1]
[poly2]

③ Explain how the function **poly1** runs faster than **poly2**, even though it requires more operations.

(4) The following table gives the parameters for a number of different caches. For each cache, fill in the missing fields in the table. Recall that **m** is the number of physical address bits, **C** is the cache size (number of data bytes), **B** is the block size in bytes, **E** is the associativity, **S** is the number of cache sets, **t** is the number of tag bits, **s** is the number of set index bits, and **b** is the number of block offset bits.

Cache	m	C	B	E	S	t	s	b
①	32	4,096	4	4				
②	32	2,048	8	2				
③	32	1,024	32	1				

(5) This problem concerns basic cache lookups.

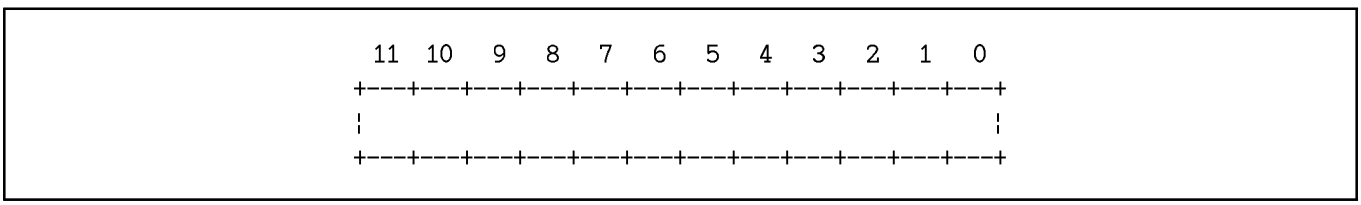
- The memory is byte addressable.
- Memory accesses are to 1-byte words (not 4-byte words).
- Physical addresses are 12-bits wide.
- The cache is 4-way set associative, with a 2-byte block size and 32 total lines.

In the following tables, all numbers are given in hexadecimal. The contents of the cache are as follows:

4-way Set Associative Cache																
Index	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1	Tag	Valid	Byte 0	Byte 1
0	29	0	34	29	87	0	39	AE	7D	1	68	F2	8B	1	64	38
1	F3	1	0D	8F	3D	1	0C	3A	4A	1	A4	DB	D9	1	A5	3C
2	A7	1	E2	04	AB	1	D2	04	E3	0	3C	A4	01	0	EE	05
3	3B	0	AC	1F	E0	0	B5	70	3B	1	66	95	37	1	49	F3
4	80	1	60	35	2B	0	19	57	49	1	8D	0E	00	0	70	AB
5	EA	1	B4	17	CC	1	67	DB	8A	0	DE	AA	18	1	2C	D3
6	1C	0	3F	A4	01	0	3A	C1	F0	0	20	13	7F	1	DF	05
7	0F	0	00	FF	AF	1	B1	5F	99	0	AC	96	3A	1	22	79

① The box below shows the format of the physical address. Indicate (by labeling the diagram) the fields that would be used to determine the following:

- `B` (The block offset within the cache line)
- `S` (The cache set index)
- `T` (The cache tag)



② For the given physical address, indicate the cache entry accessed and the cache byte value returned in hex by filling in the following table. Indicate whether a cache miss occurs. If there is a cache miss, enter "Miss" for "Cache Byte returned".

- Physical address: **D93**

	Parameter	Value	Binary address
①	Byte offset		⑥
②	Cache set index		
③	Cache tag		
④	Cache hit? (Y/N)		
⑤	Cache byte returned		

(6) Suppose you are given the task of improving the performance of a program consisting of three parts A, B, and C. Part A requires 30% of the overall run time, part B requires 10%, and part C requires 60%. You determine that for $1000 you could either speed up ① part A by a factor of 3 or ② part C by a factor of 2. Which choice would maximize performance? Explain your answer in detail.

(7) Fill in the blanks of the following sentences. Choose your answer in the Term-Box below.

○ An optimization technique, called (①), involves identifying a computation that is performed multiple times (e.g., within a loop), but such that the result of the computation will not change. We can therefore move the computation to the earlier section of the code that does not get evaluated as often.

○ With (②) execution, the processor begins fetching and decoding instructions at where it predicts the branch will go, and even begins executing these operations before it has been determined whether or not the branch prediction was correct. In this mode, the operations are evaluated, but the final results are not stored in the program registers or data memory until the processor can be certain that these instructions should actually have been executed.

○ After the disk controller receives the read command from the CPU, it translates the logical block number to a sector address, reads the contents of the sector, and transfers the contents directly to main memory, without any intervention from the CPU. This process, whereby a device performs a read or write bus transaction on its own, without any involvement of the CPU, is known as (③).

○ When we write a word **w** that is already cached (a write hit), we can update the word **w** in the cache first and defers updating the copy of **w** in the next lower level of the hierarchy as long as possible, by writing the updated block to the next lower level only when it is evicted from the cache by the replacement algorithm. This policy is called (④).

○ Static linkers such as the Linux (⑤) program take as input a collection of relocatable object files and command-line arguments and generate as output a fully Jinked executable object file that can be loaded into memory and run.

○ (⑥) is a special type of relocatable object file that can be loaded into memory and linked dynamically, at either load time or run time.

○ Compilers and assemblers generate code and data sections that start at address 0. The linker (⑦) these sections by associating a memory location with each symbol definition, and then modifying all of the references to those symbols so that they point to this memory location.

○ Modern systems compile the code segments of shared modules so that they can be loaded anywhere in memory without having to be modified by the linker. With this approach, a single copy of a shared module's code segment can be shared by an unlimited number of processes. Code that can be loaded without needing any relocations is known as (⑧).

[Term-Box]

position independent code	pipelined	ar	speculative	reentrant code
relocates	ld	direct memory access	loads	memory-mapped I/O
code motion	write-back	Linkable object file	write-through	Shared object file

①	②	③	④	⑤
⑥	⑦	⑧	⑨	⑩