Final Exam

601.229 Computer Systems Fundamentals

Fall 2019
Johns Hopkins University
Instructors: Philipp Koehn and David Hovemeyer

13 December 2019

Complete all questions.

Use additional paper if needed.

Time: 120 minutes.

Name of student: reference solution

Q1. MIPS *20 points*

(a) [8 points] Consider the following sequence of MIPS instructions:

```
1:   add $t0, $t1, $t2
2:   add $t0, $t0, $t3
3:   sw $t0, 0($a0)
```

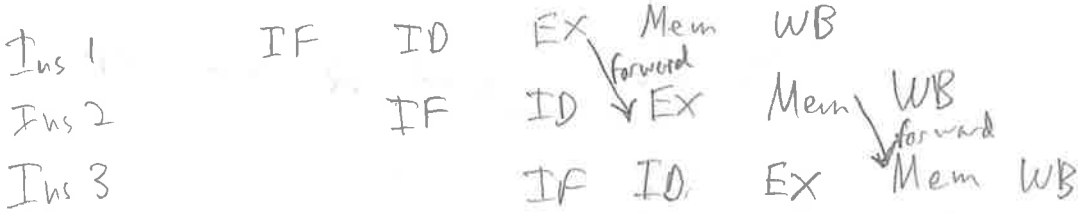
Assume that before the sequence is executed, the correct values of `$t1`, `$t2`, `$t3`, and `$a0` are immediately available in the register file.

Do any pipeline stalls occur in this sequence? If so, explain which stalls occur and why they occur. If there are no stalls, briefly explain why and mention any pipeline implementation details relevant for avoiding stalls for this sequence.

No stalls

Result of instruction 1 EX can be forwarded to instruction 2 EX

Result of instruction 2 must also be forwarded to Ins3 MEM because its WB happens at same time as Ins 3's Mem



(b) [8 points] Consider the following sequence of MIPS instructions:

```
1:   lw $t0, 0($a0)
2:   lw $t1, 4($a0)
3:   add $t2, $t0, $t1
4:   sw $t2, 8($a0)
5:   lw $t3, 0($a1)
6:   lw $t4, 4($a1)
7:   add $t5, $t3, $t4
8:   sw $t5, 8($a1)
```

Explain how to reorder instructions in order to avoid pipeline stalls, insofar as that is possible. Briefly justify your answer.

One solution — NO STALLS

```
lw $t0, 0($a0)
lw $t1, 4($a0)
lw $t3, 0($a1)
add $t2, $t0, $t1      ; operands forwarded
lw $t4, 4($a1)
sw $t2, 8($a0)
add $t5, $t3, $t4
sw $t5, 8($a1)         ; result forwarded
```

(c) [4 points] Briefly explain why MIPS uses separate caches for instructions and data.

eliminates structural hazard between IF (fetching instruction word) and Mem (loading / storing data value)

Q2. Caches *20 points*

(a) [6 points] On a system with 32 bit addresses, the data cache has 32 byte blocks, is 4-way set associative, and is 32,768 (2^{15}) bytes in size (excluding tags and metadata.)

Fill in the table below to indicate the ranges of address bits where the offset, index, and tag are located in an address, according to the cache parameters described above. Assume that bit 0 is the least significant bit of an address, and bit 31 is the most significant bit of the address.

Field	Range of bits
Address	0–31
Offset	0-4
Index	5-12
Tag	13-31

cache size = 2^{15}

blocks = $2^{15}/2^5 = 2^{10} = 1024$

$32 = 2^5$ bytes/block each set has 4 blocks,

$256 = 2^8$ sets so $1024/4 = 256$ sets

(b) [4 points] Consider the following C function:

```
void scaleMatrix(double matrix[ROWS][COLS], double fac) {
  for (int j = 0; j < COLS; j++)
    for (int i = 0; i < ROWS; i++)
      matrix[i][j] *= fac;
}
```

Assume that `ROWS` and `COLS` correctly describe the sizes of the first and second dimensions of the parameter `matrix`. Briefly explain (1) the performance issue with this function, and (2) how to change the code to fix the issue.

Perf issue: body of inner loop accesses elements in same column, but elements in same column aren't contiguous in memory (b/c 2D arrays in C are row-major)

Solution: reverse order of loops, ensures sequential access pattern

(c) [10 points] Complete the following table. For each address in the *Request* column, indicate the tags of cached blocks after handling the request. Addresses are 8 bits, blocks are 8 bytes, there are 4 sets, and the cache is 2-way set associative. All slots are initially empty. When a block is evicted, select the least-recently-used (LRU) block as the victim.

Request	Set 0		Set 1		Set 2		Set 3	
	Slot 0	Slot 1	Slot 0	Slot 1	Slot 0	Slot 1	Slot 0	Slot 1
	empty	empty	empty	empty	empty	empty	empty	empty
10110001					101			
10100011	101							
10011000							100	
00100100		001						
10011010	010						100	
01000111								
11110000						111		
00111011								001
10110100					101			
10010111						100		

Q3. x86-64 assembly, linking *20 points*

(a) [10 points] Write an x86-64 assembly language function called `swapInts` which swaps the values of two `int` variables. The C function declaration for this function would be

```
void swapInts(int *a, int *b);
```

Hints:

- Think about which registers the parameters will be passed in
- Think about what register(s) would be appropriate to use for temporary value(s)
- Consider that `int` variables are 4 bytes (32 bits), and use an appropriate operand size suffix

Important: Your function should follow proper x86-64 Linux register use conventions. Be sure to include the label defining the name of the function.

one solution

```
swapInts:
        movl (%rdi), %r10
        movl (%rsi), %r11
        movl %r10, (%rsi)
        movl %r11, (%rdi)
        ret
```

(b) [10 points] Consider the following x86-64 assembly language instruction:

```
callq some_function
```

Assume that `some_function` is not defined in the assembly language module containing this instruction. How is the address of `some_function` resolved (given that the assembler cannot know the eventual address of the function), and what are the roles of the assembler and linker in resolving the address? Explain briefly.

Assembler: emits relocation in object file as place holder for the currently-unknown address

Linker: has object code for all functions & data, including some_function, assigns addresses to them, and "fixes" all relocations to specify the assigned address

Q4. Virtual Memory *20 points*

(a) [8 points] In the 32 bit x86 architecture, addresses have 32 bits, pages are 4096 bytes, and the page tables defining an address space are structured as a tree, where

- the root node is the *page directory* containing an array of 1024 *page directory entries* containing the physical addresses of *page tables*, and
- each page table is an array of 1024 *page table entries* containing the physical addresses of physical memory pages.

The physical pages are the leaves of the tree. Note that $4096 = 2^{12}$ and $1024 = 2^{10}$.

What is the size of the portion of the overall virtual address space corresponding to a single page directory entry? Explain briefly.

address space is 2^{32} bytes
each PDE covers $1/1024 = 2^{-10}$ of addr. space

$2^{32} \cdot 2^{-10} = 2^{22}$ bytes

What is the size of the portion of the overall virtual address space corresponding to a single page table entry? Explain briefly.

each PDE controls one page-sized region of addr space, so 4096 bytes

(b) [6 points] Consider the following sequence of x86-64 assembly instructions:

```
1:    movq %rdi, %r10
2:    addq %rsi, %r10
3:    movq %r10, (%rdx)
4:    movq $1, %rax
```

Explain (1) which instruction could cause a page fault, and (2) which instruction control will return to once the page fault handler completes. (Assume that the page fault handler does not terminate the running program.)

1) Instruction 3, b/c it is a memory reference *

2) Control returns to instruction 3, so it can be re-executed with the referenced virtual page mapped to a physical page

note also that any instruction could cause a page fault on instruction fetch, if the page containing the instruction is not mapped

(c) [6 points] Briefly describe a situation in which a page fault will cause the OS kernel to both write data to disk and also load data from disk. Use specific details as appropriate.

1. process accesses unmapped page in a valid region of the address space
2. page fault handler notes that no free physical memory pages are available
3. OS kernel finds a "victim" page, and (assuming it's dirty) writes its data to disk (WRITE), and unmaps it from whatever addr. spaces it's mapped in
4. OS kernel reads new data from disk into the physical page (READ), e.g., from a file being accessed using memory-mapped I/O

Q5. Networks/threads *20 points*

(a) [8 points] Complete the following function called `chat_with_client`. It implements the server's role in the following network protocol:

- Clients will send messages consisting of 4 lower or upper case letters followed by a newline
- For each received client message, the server will send back a response in which each letter is converted to upper case
- As a special case, if the client sends `quit` followed by a newline, the session should be ended

Example session:

Client sends:	Server sends back:
`CSFi`	`CSFI`
`sDon`	`SDON`
`eYAY`	`EYAY`
`quit`	

Note that `clientfd` is a file descriptor referring to a TCP socket that can be used for both receiving data and sending data. You may assume the existence of `read_fully` and `write_fully` functions, which are like `read` and `write`, but will read and write the requested number of bytes if possible. You may assume that I/O routines will not fail. You may assume that messages sent by the client will be properly formed.

Continue on next page if necessary.

```
void chat_with_client(int clientfd) {
    int done = 0;  char buf[5];
    while (!done) {
        read_fully(clientfd, buf, 5);
        if (memcmp(buf, "quit\n", 5) == 0) { done = 1; }
        else {
            for (int i=0; i<4; i++) { buf[i] = toupper(buf[i]); }
            write_fully(clientfd, buf, 5);
        }
    }
}
```

[Continue code for Q5 part (a) here if necessary, and note that Q5 continues to parts (b) and (c) on the following pages]

(b) [8 points] Consider the following data type and functions:

```
typedef struct {
  int a;
  int b;
} IntPair;
```

struct pthread_mutex_t lock;

```
IntPair *intpair_create(int aval, int bval) {
  IntPair *pair = malloc(sizeof(IntPair));
  pair->a = aval;
  pair->b = bval;
  return pair;
}
```

pthread_mutex_init(&pair->lock, NULL);

```
void intpair_destroy(IntPair *pair) {
  free(pair);
}
```

pthread_mutex_destroy(&pair->lock);

```
void intpair_swap(IntPair *pair) {
  int tmp = pair->a;
  pair->a = pair->b;
  pair->b = tmp;
}
```

pthread_mutex_lock(&pair->lock);

pthread_mutex_unlock(&pair->lock);

```
void intpair_adjust(IntPair *pair, int adelta, int bdelta) {
  pair->a += adelta;
  pair->b += bdelta;
}
```

lock mutex

unlock mutex

```
int intpair_getsum(IntPair *pair) {
  int sum = pair->a + pair->b;
  return sum;
}
```

lock mutex

unlock mutex

Annotate the above code to show how to make instances of the `IntPair` type safe to access from multiple threads. (I.e., show where code should be inserted.)

(c) [4 points] The purpose of the Internet is pictures of cats. Draw a picture of a cat.

We hope you enjoyed CSF! Have a great break!