Grade:

Computer Principles (Spring 2018)

Final Exam

Beihang University

2018/06/27

Your Name:

Your Student No.:

Notes:

Only two cheat sheets are allowed.

Calculator is allowed but computer and mobile phones are NOT allowed during exam.

Please write your answers on the answer sheets. Do NOT write answers directly on the exam paper.

Remember to write your name and student No. on each page of the answer sheets.

Problem 1 (10 points)

Unsigned Integers

If we have an n-digit unsigned numeral $d_{n-1}d_{n-2}\ldots d_0$ in radix (or base) r, then the value of that numeral is $\sum_{i=0}^{n-1} r^i d_i$, which is just fancy notation to say that instead of a 10's or 100's place we have an r's or r^2's place. For the three radices binary, decimal, and hex, we just let r be 2, 10, and 16, respectively.

Recall also that we often have cause to write down unreasonably large numbers, and our preferred tool for doing that is the IEC prefixing system:

Ki (Kibi) $= 2^{10}$	Mi (Mebi) $= 2^{20}$	Gi (Gibi) $= 2^{30}$	Ti (Tebi) $= 2^{40}$
Pi (Pebi) $= 2^{50}$	Ei (Exbi) $= 2^{60}$	Zi (Zebi) $= 2^{70}$	Yi (Yobi) $= 2^{80}$

1.1 We don't have calculators during exams, so let's try this by hand

1. Convert the following numbers from their initial radix into the other two common radices:
   (a) 0b10010011
   (b) 0xD3AD
   (c) 63
   (d) 0b00100100
   (e) 0xB33F
   (f) 0
   (g) 39
   (h) 0x7EC4
   (i) 437
2. Write the following numbers using IEC prefixes:
   - 2^{16}
   - 2^{27}
   - 2^{43}
   - 2^{36}
   - 2^{34}
   - 2^{61}
   - 2^{47}
   - 2^{58}
3. Write the following numbers as powers of 2:
   - 2 Ki
   - 512 Ki
   - 16 Mi
   - 256 Pi
   - 64 Gi
   - 128 Ei

Problem 2 (10 points)

Memory Management

1. Match the items on the left with the memory segment in which they are stored. Answers may be used more than once, and more than one answer may be required.

1. Static variables	A. Code
2. Local variables	B. Static
3. Global variables	C. Heap
4. Constants	D. Stack
5. Machine Instructions	
6. Result of malloc()	
7. String Literals	

2. Write the code necessary to properly allocate memory (on the heap) in the following scenarios

   1. An array `arr` of k integers

   2. A string `str` containing p characters

   3. An $n \times m$ matrix `mat` of integers initialized to zeros

3. What is wrong with the C code below?

```
int* pi = malloc(314 * sizeof(int));
if(!raspberry) {
  pi = malloc(1 * sizeof(int));
}
return pi;
```

4. Write code to prepend (add to the start) to a linked list, and to free/empty the entire list.
   `struct ll_node { struct ll_node* next; int value; }`

`void prepend(struct ll_node** lst, int val)`	`void free_ll(struct ll_node** lst)`

Note: `*lst` *points to the first element of the list, or is* `NULL` *if the list is empty.*

Problem 3 (10 points)

RISC-V Intro

1. Assume we have an array in memory that contains `int* arr = {1,2,3,4,5,6,0}`. Let the value of `arr` be a multiple of 4 and stored in register `s0`. What do the snippets of RISC-V code do? Note that these snippits all run immediately after each other (snippet a) runs, then snippet b) and so on).

```
a) lw t0, 12(s0)

b) slli t1, t0, 2
   add t2, s0, t1
   lw  t3, 0(t2)
   addi t3, t3, 1
   sw t3, 0(t2)

c) lw t0, 0(s0)
   xori t0, t0, 0xFFF
   addi t0, t0, 1
```

2. What are the instructions to branch to `label` on each of the following conditions? The only branch instructions you may use are `beq` and `bne`.

`s0 < s1`	`s0 <= s1`	`s0 > 1`

Problem 4 (10 points)

RISC-V Instruction Formats

1 Overview

Instructions in RISC-V can be turned into binary numbers that the machine actually reads. There are different formats to the instructions, based on what information is need. Each of the feilds above is filled in with binary

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

that represents the information. Each of the registers takes a 5 bit number that is the numeric name of the register (i.e. zero = 0, ra = 1, s1 = 9). See your reference card to know which register corresponds to which number.
I type instructions fill the immediate into the code. These numbers are signed 12 bit numbers.

2 Exercises

1. Expand `addi s0 t0 -1`

2. Expand `lw s4 5(sp)`

3. Write the format name of the following instructions:

   (a) `jal`
   (b) `lw`
   (c) `beq`
   (d) `add`
   (e) `jalr`
   (f) `sb`
   (g) `lui`

Problem 5 (10 points)

RISC-V Calling Conventions

1. How do we pass arguments into functions?

2. How are values returned by functions?

3. What is `sp` and how should it be used in the context of RISC-V functions?

4. Which values need to saved before using `jal`?

5. Which values need to be restored before using `jalr` to return from a function?

Problem 6 (10 points)

RISC-V Addressing

1 Overview

- We have several **addressing modes** to access memory (immediate not listed):
  (a) **Base displacement addressing:** Adds an immediate to a register value to create a memory address (used for lw, lb, sw, sb)
  (b) **PC-relative addressing:** Uses the PC and adds the immediate value of the instruction (multiplied by 2) to create an address (used by branch and jump instructions)
  (c) **Register Addressing:** Uses the value in a register as a memory address (jr)

2 Exercises

1. What is range of 32-bit instructions that can be reached from the current PC using a branch instruction?

2. What is the range of 32-bit instructions that can be reached from the current PC using a jump instruction?

3. Given the following RISC-V code (and instruction addresses), fill in the blank fields for the following instructions (you'll need your RISC-V green card!).

```
0x002cff00: loop: add t1, t2, t0          |________|________|________|________|________|__0x33__|
0x002cff04:       jal ra, foo             |____________________________|________________|__0x6F__|
0x002cff08:       bne t1, zero, loop      |________|________|________|________|________|__0x63__|
...
0x002cff2c: foo:  jr ra                 ra=___________________
```

Problem 7 (10 points)

Single Cycle CPU Design

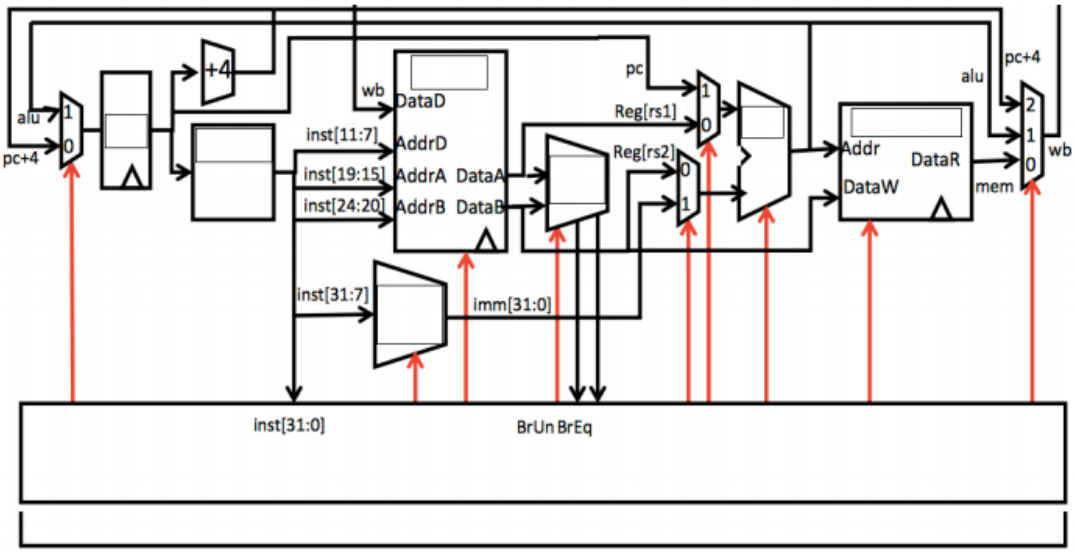
Here we have a single cycle CPU diagram. Answer the following questions:

1. Name each component.

2. Name each datapath stage and explain its functionality.

Stage	Functionality

3. Provide data inputs and control signals to the next PC logic.



Single Cycle CPU Control Logic

Fill out the values for the control signals from the previous CPU diagram.

Instrs.	Control Signals									
	BrEq	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add										
ori										
lw										
sw										
beq										
jal										

(Put an X if the signal doesn't matter)

Clocking Methodology

- The input signal to each state element must stabilize before each rising edge.
- Critical path: Longest delay path between state elements in the circuit.
- If we place registers in the critical path, we can shorten the period by reducing the amount of logic between registers.

Single Cycle CPU Performance Analysis

The delays of circuit elements are given as follows:

Stage	IF	ID	EX	MEM	WB
Delay	200	100	200	200	100

1. Mark the stages the following instructions use and calculate the time to execute.

Instruction	IF	ID	EX	MEM	WB	Total
add						
lw						
sw						
xori						
beq						
jal						

2. Which instruction(s) exercises the critical path?

3. What is the fastest you could clock this single-cycle datapath?

4. Why is a single cycle CPU inefficient?

5. How can you improve its performance? What is the purpose of pipelining?

Problem 8 (10 points)

Putting It All Together

Instruction	C0	C1	C2	C3	C4	C5	C6			
1. addi t0, s0, -1	IF	REG	EX	MEM	WB					
2. and s2, t0, a0		IF	REG	EX	MEM	WB				
3. sw s2, 100(t0)			IF	REG	EX	MEM	WB			
4. beq s0, s3, label				IF	REG	EX	MEM	WB		
5. addi t2, x0, x0					IF	REG	EX	MEM	WB	

Given the RISC-V code above and a pipelined CPU with no forwarding, how many hazards would there be? What types are each hazard? Consider all possible hazards from all pairs of instructions.

How many stalls would there need to be in order to fix the data hazard(s)? What about the control hazard(s)?

Problem 9 (10 points)

Tag Index Offset

When working with caches, we have to be able to break down the memory addresses we work with to understand where they fit into our caches. There are three fields:

Tag - Used to distinguish different blocks that use the same index - Number of bits: leftovers
Index - The set that this piece of memory will be placed in - Number of bits: $\log_2$(# of indices)
Offset - The location of the byte in the block - Number of bits: $\log_2$(size of block)

In the following diagrams, each blank box represents 1 byte (8 bits) of data. All of memory is byte addressed.

Direct mapped Cache

1. Let's say we have a 8192KiB cache with an 128B block size, how many bits are in tag, index, and offset? What parts of the address of 0xFEEDF00D fit into which sections?

	Tag	Index	Offset
Number of bits			
Bits of address			

2. Now fill in the table below. Assume that we have a write-through cache, so the number of bits per row includes only the cache data, the tag, and the valid bit.

Address size (bits)	Cache Size	Block Size	Tag Bits	Index Bits	Offset Bits	Bits per row
16	4KiB	4B				
32	32KiB	16B				
32			16	12		
64	2048KiB			14		1068

Cache Hits and Misses

1 Finding Hits and Misses

Assume we have a direct-mapped byte-addressed cache with capacity 32B and block size of 8B. Of the 32 bits in each address, which bits do we use to find the index of the cache to use?
Classify each of the following byte memory accesses as a cache hit (H), cache miss (M), or cache miss with replacement(R). It is probably best to try drawing out the cache before going through so that you can have an easier time seeing the replacements in the cache. The following white space is to do this:

1. 0x00000004
2. 0x00000005
3. 0x00000068
4. 0x000000C8
5. 0x00000068
6. 0x000000DD
7. 0x00000045
8. 0x00000004
9. 0x000000C8

2 The 3 C's of Misses

3 types of cache misses:

I. Compulsory: First time you ask the cache for a certain block. A miss that must occur when you first bring in a block. Reduce compulsory misses by having a longer cache lines (bigger blocks), which bring in the surrounding addresses along with our requested data. Can also pre-fetch blocks beforehand using a hardware prefetcher (a special circuit that tries to guess the next few blocks that you will want).

II. Conflict: Occurs if you hypothetically went through the ENTIRE string of accesses with a fully associative cache and wouldn't have missed for that specific access. Increasing the associativity or improving the replacement policy would remove the miss.

III. Capacity: The only way to remove the miss is to increase the cache capacity, as even with a fully associative cache, we had to kick a block out at some point.

Note: There are many different ways of fixing misses. The name of the miss doesn't necessarily tell us the best way to reduce the number of misses.

Classify each M and R above as one of the 3 misses above.

Problem 10 (10 points)

Floating Point

The IEEE 754 standard defines a binary representation for floating point values using three fields:

- The *sign* determines the sign of the number (0 for positive, 1 for negative)
- The *exponent* is in **biased notation** with a bias of 127
- The *significand* is akin to unsigned, but used to store a fraction instead of an integer.

The below table shows the bit breakdown for the single precision (32-bit) representation:

Sign	Exponent	Significand
1 bit	8 bits	23 bits

There is also a double precision encoding format that uses 64 bits. This behaves the same as the single precision but uses 11 bits for the exponent (and thus a bias of 1023) and 52 bits for the significand.

How a float is interpreted depends on the values in the exponent and significand fields:

For normalized floats:

Exponent	**Significand**	**Meaning**
0	Anything	Denorm
1-254	Anything	Normal
255	0	Infinity
255	Nonzero	NaN

$$\text{Value} = (-1)^{\text{Sign}} \times 2^{(\text{Exponent} - \text{Bias})} \times 1.\text{significand}_2$$

For denormalized floats:

$$\text{Value} = (-1)^{\text{Sign}} \times 2^{(\text{Exponent} - \text{Bias} + 1)} \times 0.\text{significand}_2$$

Exercises

1. How many zeroes can be represented using a float?

2. What is the largest finite positive value that can be stored using a single precision float?

3. What is the smallest positive value that can be stored using a single precision float?

4. What is the smallest positive normalized value that can be stored using a single precision float?

5. Convert the following numbers from binary to decimal or from decimal to binary:

0x00000000 8.25 0x00000F00 39.5625 0xFF94BEEF -∞