(CS2.201a) Computer Systems Organization (Spring 2023) 01.04.2023

End-Semester Examination

Alloted time: 60 minutes **Marks: 32**

Instructions:

- There are four questions with sub-parts, printed over two sides of a sheet. Please check if your sheet has all the questions. Marks will be normalized by a factor of 10/32.
- Discussions amongst the students are not allowed. Any dishonesty shall be penalized heavily.
- Be clear in your arguments. Vague arguments shall not be given full credit.
- Please note that in class we discussed NASM x86 assembly and in assignments we discussed GNU x86 assembly. The main difference is in the order of operands. NASM uses destination first but GNU Assembly does not use destination as a first operand.

---

1. Consider a 5-bit floating-point representation based on the IEEE floating-point format, with one sign bit, two exponent bits ($k = 2$), and two fraction bits ($n = 2$).
   (a) Compute the bias.
   (b) What is the binary representation of $-\frac{31}{8}$ and $\frac{12}{56}$ in the above floating point representation.

   [2+3+3 marks]

2. Explain the given GNU x86 assembly snippet.

```
push   %ebp
mov    %esp,%ebp
mov    0x8(%ebp),%eax
# lea (%esi,%ebx,4), %edx corresponds to setting %edx to the address %esi+%ebx*4
lea    (%eax,%eax,4),%eax
lea    0xfffffff6(,%eax,4),%eax
mov    %ebp,%esp
pop    %ebp
ret
```

   [8 marks]

3. The 5 stages of the processor have the following latencies:

	Fetch	Decode	Execute	Memory	Writeback
a.	300ps	400ps	350ps	550ps	100ps
b.	200ps	150ps	100ps	190ps	140ps

   Assume that when pipelining, each pipeline stage costs 20ps extra for the registers between pipeline stages.

(a) Non-pipelined processor: what is the cycle time? What is the latency of an instruction? What is the throughput?

(b) Fully Pipelined processor: What is the cycle time? What is the latency of an instruction? What is the throughput?

(c) If you could split one of the pipeline stages into 2 equal halves, which one would you choose? What is the new cycle time? What is the new latency? What is the new throughput?

[2+3+3 marks]

4. (a) For a direct-mapped cache design with a 32-bit address and byte-addressable memory, the following bits of the address are used to access the cache:

	Tag	Index	Offset
a.	31-10	9-5	4-0
b.	31-12	11-6	5-0

For each configuration (a and b):

i. What is the cache block size (in words)?

ii. How many entries (blocks) does the cache have?

[1.5+1.5 marks]

(b) You have a 2-way set associative L1 cache that is 8KB, with 4-word cache lines. Writing and reading data to L2 takes 10 cycles. You get the following sequence of writes to the cache – each is a 32-bit address in hexadecimal:

0x1000
0x1004
0x1010
0x11c0
0x2000
0x21c0
0x3400
0x3404
0x3f00
0x2004
0x1004

i. How many cache misses occur with an LRU policy?

ii. How many cache misses occur with a FIFO policy?

iii. Would the miss-rate increase or decrease if the cache was the same size, but direct-mapped (in above case)? Explain.

[2+1+2 marks]

x86-64 Reference Sheet (GNU assembler format)

Instructions

Data movement

`movq` Src, Dest	Dest = Src
`movsbq` Src,Dest	Dest (quad) = Src (byte), sign-extend
`movzbq` Src,Dest	Dest (quad) = Src (byte), zero-extend

Conditional move

`cmove` Src, Dest	Equal / zero
`cmovne` Src, Dest	Not equal / not zero
`cmovs` Src, Dest	Negative
`cmovns` Src, Dest	Nonnegative
`cmovg` Src, Dest	Greater (signed >)
`cmovge` Src, Dest	Greater or equal (signed $\geq$)
`cmovl` Src, Dest	Less (signed <)
`cmovle` Src, Dest	Less or equal (signed $\leq$)
`cmova` Src, Dest	Above (unsigned >)
`cmovae` Src, Dest	Above or equal (unsigned $\geq$)
`cmovb` Src, Dest	Below (unsigned <)
`cmovbe` Src, Dest	Below or equal (unsigned $\leq$)

Control transfer

`cmpq` Src2, Src1	Sets CCs Src1 Src2
`testq` Src2, Src1	Sets CCs Src1 & Src2
`jmp label`	jump
`je label`	jump equal
`jne label`	jump not equal
`js label`	jump negative
`jns label`	jump non-negative
`jg label`	jump greater (signed >)
`jge label`	jump greater or equal (signed $\geq$)
`jl label`	jump less (signed <)
`jle label`	jump less or equal (signed $\leq$)
`ja label`	jump above (unsigned >)
`jb label`	jump below (unsigned <)
`pushq` Src	%rsp = %rsp 8, Mem[%rsp] = Src
`popq` Dest	Dest = Mem[%rsp], %rsp = %rsp + 8
`call label`	push address of next instruction, `jmp label`
`ret`	%rip = Mem[%rsp], %rsp = %rsp + 8

Arithmetic operations

`leaq` Src, Dest	Dest = address of Src
`incq` Dest	Dest = Dest + 1
`decq` Dest	Dest = Dest − 1
`addq` Src, Dest	Dest = Dest + Src
`subq` Src, Dest	Dest = Dest − Src
`imulq` Src, Dest	Dest = Dest * Src
`xorq` Src, Dest	Dest = Dest ^ Src
`orq` Src, Dest	Dest = Dest | Src
`andq` Src, Dest	Dest = Dest & Src
`negq` Dest	Dest = − Dest
`notq` Dest	Dest = ∼ Dest
`salq` *k*, Dest	Dest = Dest $\ll k$
`sarq` *k*, Dest	Dest = Dest $\gg k$ (arithmetic)
`shrq` *k*, Dest	Dest = Dest $\gg k$ (logical)

Addressing modes

- **Immediate**
  $val Val
  val: constant integer value
  `movq $7, %rax`
- **Normal**
  (R) Mem[Reg[R]]
  R: register R specifies memory address
  `movq (%rcx), %rax`
- **Displacement**
  D(R) Mem[Reg[R]+D]
  R: register specifies start of memory region
  D: constant displacement D specifies offset
  `movq 8(%rdi), %rdx`
- **Indexed**
  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+D]
  D: constant displacement 1, 2, or 4 bytes
  Rb: base register: any of 8 integer registers
  Ri: index register: any, except %esp
  S: scale: 1, 2, 4, or 8
  `movq 0x100(%rcx,%rax,4), %rdx`

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long (4 bytes)
`q`	quad (8 bytes)

Condition codes

CF	Carry Flag
ZF	Zero Flag
SF	Sign Flag
OF	Overflow Flag

Integer registers

%rax	Return value
%rbx	Callee saved
%rcx	4th argument
%rdx	3rd argument
%rsi	2nd argument
%rdi	1st argument
%rbp	Callee saved
%rsp	Stack pointer
%r8	5th argument
%r9	6th argument
%r10	Scratch register
%r11	Scratch register
%r12	Callee saved
%r13	Callee saved
%r14	Callee saved
%r15	Callee saved

CS107 x86-64 Reference Sheet

Common instructions

`mov src, dst`	dst = src
`movsbl src, dst`	byte to int, sign-extend
`movzbl src, dst`	byte to int, zero-fill
`cmov src, reg`	reg = src when condition holds, using same condition suffixes as jmp
`lea addr, dst`	dst = addr
`add src, dst`	dst += src
`sub src, dst`	dst -= src
`imul src, dst`	dst *= src
`neg dst`	dst = -dst (arith inverse)
`imulq S`	signed full multiply R[%rdx]:R[%rax] <- S * R[%rax]
`mulq S`	unsigned full multiply same effect as **imulq**
`idivq S`	signed divide R[%rdx] <- R[%rdx]:R[%rax] mod S R[%rax] <- R[%rdx]:R[%rax] / S
`divq S`	unsigned divide - same effect as **idivq**
`cqto`	R[%rdx]:R[%rax] <- SignExtend(R[%rax])
`sal count, dst`	dst <<= count
`sar count, dst`	dst >>= count (arith shift)
`shr count, dst`	dst >>= count (logical shift)
`and src, dst`	dst &= src
`or src, dst`	dst |= src
`xor src, dst`	dst ^= src
`not dst`	dst = ~dst (bitwise inverse)
`cmp a, b`	b-a, set flags
`test a, b`	a&b, set flags
`set dst`	sets byte at dst to 1 when condition holds, 0 otherwise, using same condition suffixes as jmp
`jmp label`	jump to label (unconditional)
`je label`	jump equal ZF=1
`jne label`	jump not equal ZF=0
`js label`	jump negative SF=1
`jns label`	jump not negative SF=0
`jg label`	jump > (signed) ZF=0 and SF=OF
`jge label`	jump >= (signed) SF=OF
`jl label`	jump < (signed) SF!=OF
`jle label`	jump <= (signed) ZF=1 or SF!=OF
`ja label`	jump > (unsigned) CF=0 and ZF=0
`jae label`	jump >= (unsigned) CF=0
`jb label`	jump < (unsigned) CF=1
`jbe label`	jump <= (unsigned) CF=1 or ZF=1
`push src`	add to top of stack Mem[--%rsp] = src
`pop dst`	remove top from stack dst = Mem[%rsp++]
`call fn`	push %rip, jmp to fn
`ret`	pop %rip

Condition codes/flags

ZF	Zero flag
SF	Sign flag
CF	Carry flag
OF	Overflow flag

Addressing modes

Example source operands to **mov**

Immediate

```
mov $0x5, dst
```

$val
source is constant value

Register

```
mov %rax, dst
```

%R
R is register
source in %R register

Direct

```
mov 0x4033d0, dst
```

0xaddr
source read from Mem[0xaddr]

Indirect

```
mov (%rax), dst
```

(%R)
R is register
source read from Mem[%R]

Indirect displacement

```
mov 8(%rax), dst
```

D(%R)
R is register
D is displacement
source read from Mem[%R + D]

Indirect scaled-index

```
mov 8(%rsp, %rcx, 4), dst
```

D(%RB,%RI,S)
RB is register for base
RI is register for index (0 if empty)
D is displacement (0 if empty)
S is scale 1, 2, 4 or 8 (1 if empty)
source read from:
Mem[%RB + D + S*%RI]

CS107 x86-64 Reference Sheet

Registers

`%rip`	Instruction pointer
`%rsp`	Stack pointer
`%rax`	Return value
`%rdi`	1st argument
`%rsi`	2nd argument
`%rdx`	3rd argument
`%rcx`	4th argument
`%r8`	5th argument
`%r9`	6th argument
`%r10,%r11`	Callee-owned
`%rbx,%rbp, %r12-%15`	Caller-owned

Instruction suffixes

`b`	byte
`w`	word (2 bytes)
`l`	long /doubleword (4 bytes)
`q`	quadword (8 bytes)

Suffix is elided when can be inferred from operands. e.g. operand `%rax` implies `q`, `%eax` implies `l`, and so on

Register Names

64-bit register	32-bit sub-register	16-bit sub-register	8-bit sub-register
%rax	%eax	%ax	%al
%rbx	%ebx	%bx	%bl
%rcx	%ecx	%cx	%cl
%rdx	%edx	%dx	%dl
%rsi	%esi	%si	%sil
%rdi	%edi	%di	%dil
%rbp	%ebp	%bp	%bpl
%rsp	%esp	%sp	%spl
%r8	%r8d	%r8w	%r8b
%r9	%r9d	%r9w	%r9b
%r10	%r10d	%r10w	%r10b
%r11	%r11d	%r11w	%r11b
%r12	%r12d	%r12w	%r12b
%r13	%r13d	%r13w	%r13b
%r14	%r14d	%r14w	%r14b
%r15	%r15d	%r15w	%r15b

NASM Intel x86 Assembly Language Cheat Sheet

Instruction	Effect	Examples
Copying Data		
mov *dest,src*	Copy src to dest	mov eax,10 mov eax,[2000]
Arithmetic		
add *dest,src*	dest = dest + src	add esi,10
sub *dest,src*	dest = dest – src	sub eax, ebx
mul *reg*	edx:eax = eax * reg	mul esi
div *reg*	edx = edx:eax **mod** reg eax = edx:eax ÷ reg	div edi
inc *dest*	Increment destination	inc eax
dec *dest*	Decrement destination	dec word [0x1000]
Function Calls		
call *label*	Push eip, transfer control	call format_disk
ret	Pop eip and return	ret
push *item*	Push item (constant or register) to stack. I.e.: esp=esp-4; memory[esp] = item	push dword 32 push eax
pop *[reg]*	Pop item from stack and store to register I.e.: reg=memory[esp]; esp=esp+4	pop eax
Bitwise Operations		
and *dest, src*	dest = src & dest	and ebx, eax
or *dest,src*	dest = src | dest	or eax,[0x2000]
xor *dest, src*	dest = src ^ dest	xor ebx, 0xffffffff
shl *dest,count*	dest = dest << count	shl eax, 2
shr *dest,count*	dest = dest >> count	shr dword [eax],4
Conditionals and Jumps		
cmp *b,a*	Compare b to a; must immediately precede any of the conditional jump instructions	cmp eax,0
je *label*	Jump to label if b == a	je endloop
jne *label*	Jump to label if b != a	jne loopstart
jg *label*	Jump to label if b > a	jg exit
jge *label*	Jump to label if b ≥ a	jge format_disk
jl *label*	Jump to label if b < a	jl error
jle *label*	Jump to label if b ≤ a	jle finish
test *reg,imm*	Bitwise compare of register and constant; should immediately precede the jz or jnz instructions	test eax,0xffff
jz *label*	Jump to label if bits were **not** set (“zero”)	jz looparound
jnz *label*	Jump to label if bits **were** set (“not zero”)	jnz error
jmp *label*	Unconditional relative jump	jmp exit
jmp *reg*	Unconditional absolute jump; arg is a register	jmp eax
Miscellaneous		
nop	No-op (opcode 0x90)	nop
hlt	Halt the CPU	hlt

Instructions with no memory references must include ‘byte’, ‘word’ or ‘dword’ size specifier.
Arguments to instructions: Note that it is not possible for **both** src and dest to be memory addresses.

Constant (decimal or hex):	10 or 0xff	Fixed address:	[200] or [0x1000+53]
Register:	eax bl	Dynamic address:	[eax] or [esp+16]

32-bit registers: eax, ebx, ecx, edx, esi, edi, ebp, esp (points to first used location on top of stack)
16-bit registers: ax, bx, cx, dx, si, di, sp, bp
8-bit registers: al, ah, bl, bh, cl, ch, dl, dh