

Computer Organization

Spring 2004 Test #2

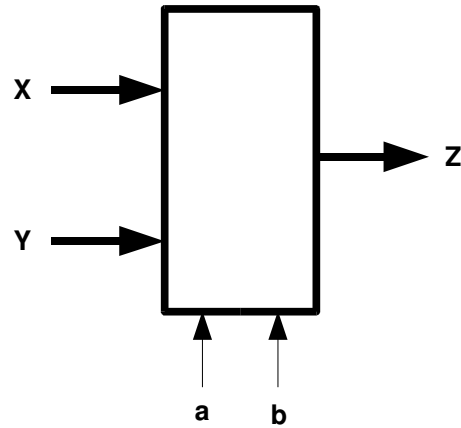
Name _____ Answer Key _____

There are 8 pages - make sure you have all of them.
Answer all questions - pay attention to the # of points for each question.
Don't leave anything blank - partial credit is always possible!

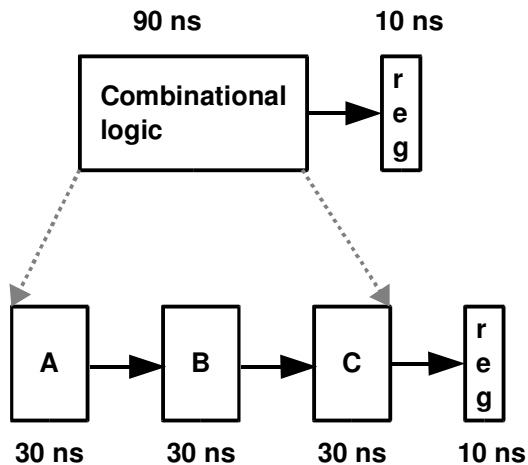
Question 1 (10 pts): We have a circuit (shown below) that has 2 word inputs labeled X and Y, and two single bit (boolean) inputs labeled a and b. The output (labeled Z) is determined as follows: If boolean inputs a and b are the same (either both 1 or both 0): the output is the value of X, otherwise the output is the value of the minimum of X and Y.

Express the computation done by this circuit using HCL.

```
int Z = [  
    (a&&b) : X;  
    (!a && !b) : X;  
    X<Y : X;  
    1: Y;  
];
```



Question 2 (10 pts): Using a sequential implementation, it takes a total of 100ns for each instruction, 90ns for the combinational logic to complete, and 10ns to store the results (in a register). We can break up the combinational logic in to 3 separate blocks named A,B and C, having delays 30 ns each :



Question 2a (5pts): What is the *throughput* (instructions/second) using the sequential implementation?

```
throughput      = 1 instructions/100ns
                = 1/100 * 10-9 seconds
                = 10 M Instructions/Second
```

Question 2b (5pts): We can create a pipelined version by inserting a single pipeline register between either blocks A and B or between blocks B and C (we are building a 2-stage pipeline, not 3!). What is the *throughput* (instructions/second) using the resulting 2 stage pipeline (assume that the pipeline register take 10ns).?

```
cycle length determined by the slowest stage in the
pipeline, so cycle len is 70ns. 2 cycles per
instruction (but we are always executing 2
instructions at a time):
throughput      = 2 instructions/140ns
                = 1/70 * 10-9 seconds
                = ~ 14 M Instructions/Second
```

Question 3 (10 pts): This question involves the Y86 code shown below and the 5 stage Y86 pipeline we discussed in class (stages Fetch, Decode, Execute, Memory, Writeback).

```
        mrmovl    100(%ecx), %eax
        irmovl    $20, %edx
        addl      %eax, %edx
        jne       foo
foo:     addl      $1, %esi
```

Question 3a (5 pts): Identify the *data* and *control* hazards in the above code. Identify the hazards any way you want (describe them, draw pictures, ...).

Data: third instruction needs `eax` and `edx` set from first two during it's decode stage, but the values have not yet been written to registers at that time. Also: `jne` needs condition codes set by the third instruction, but this does not happen in time

Control: No way to tell what instruction to fetch right after the `jne` instruction (no way to know what instruction is next!)

Question 3b (5 pts): Rewrite the above Y86 code with `nops` inserted that eliminate all the *data* hazards (with no unnecessary `nops`). You don't need to worry about eliminating any control hazards in your answer.

```
        mrmovl    100(%ecx), %eax
        irmovl    $20, %edx
        nop
        nop
        nop
        addl      %eax, %edx
```

Question 4 (35 pts): We want to add a new instruction to the Y86 sequential implementation. The new instruction will support setting any word in memory to the value 0. The instruction name will be `clear`. The single operand to the `clear` instruction is a memory location specified via the Y86 addressing mode (sum of a register and displacement). For example, assuming that register `%edx` holds the value `0x100`; the following instruction would set memory location `0x104` to the value 0:

```
clear 4(%edx)
```

Question 4a (20 pts): The new Y86 instruction named **clear** will be a six byte instruction as follows:

```
clear D(rB)
```

E0	rArB	D
-----------	-------------	----------

Where D is a 4 byte value. Fill out the following form to describe what needs to happen during each stage of this new instruction, a copy of the form for the push instruction is included for reference.

	pushl rA	clear
Fetch	<code>icode:ifun <- M₁[PC]</code> <code>rA:rB <- M₁[PC+1]</code> <code>valP <- PC+2</code>	<code>icode:ifun <- M₁[PC]</code> <code>rA:rB <- M₁[PC+1]</code> <code>valC <- M₄[PC+2]</code> <code>valP <- PC+6</code>
Decode	<code>valA <- R[rA]</code> <code>valB <- R[%esp]</code>	<code>valB <- R[rB]</code>
Execute	<code>valE <- valB + (-4)</code>	<code>valE <- valB + valC</code>
Memory	<code>M₄[valE] <- valA</code>	<code>M₄[valE] <- 0</code>
Write Back	<code>R[%esp] <- valE</code>	
PC update	<code>PC <- valP</code>	<code>PC <- valP</code>

Question 4b (15 pts): Modify the HCL expressions show below to support this new instruction. You can assume that the symbol ICLEAR is defined to represent the icode for this new instruction.

```
## Does fetched instruction require a regid byte?
bool need_regids = icode in { IRRMOVL, IOPL, IPUSHL, IPOPL,
                             IIRMOVL, IRMMOVL, IMRMOVL, ICLEAR };

## Does fetched instruction require a constant word?
bool need_valC = icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJXX, ICALL, ICLEAR };

## What register should be used as the A source?
int srcA = [
    icode in { IRRMOVL, IRMMOVL, IOPL, IPUSHL } : rA;
    icode in { IPOPL, IRET } : RESP;
    1 : RNONE; # Don't need register
];

## What register should be used as the B source?
int srcB = [
    icode in { IOPL, IRMMOVL, IMRMOVL, ICLEAR } : rB;
    icode in { IPUSHL, IPOPL, ICALL, IRET } : RESP;
    1 : RNONE; # Don't need register
];

## Select input A to ALU
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, ICLEAR } : valC;
    icode in { ICALL, IPUSHL } : -4;
    icode in { IRET, IPOPL } : 4;
];

## Select input B to ALU
int aluB = [
    icode in { IRMMOVL, IMRMOVL, IOPL, ICALL, IPUSHL, IRET, IPOPL, ICLEAR }
: valB;
    icode in { IRRMOVL, IIRMOVL } : 0;
];

## Select memory address
int mem_addr = [
    icode in { IRMMOVL, IPUSHL, ICALL, IMRMOVL, ICLEAR } : valE;
    icode in { IPOPL, IRET } : valA;
];

## Select memory input data
int mem_data = [
    icode in { IRMMOVL, IPUSHL } : valA;
    icode == ICALL : valP;
    icode == ICLEAR : 0;
];
```

Question 5 (15 pts): This question tests your knowledge of the stack and byte ordering. The following C code is compiled using gcc on an X86 machine (IA32 instruction set) running Linux:

```

int blah(int x) {
    char buf[4];
    gets(buf);
    printf("%s %x\n",buf,x);
    return x+1;
}

int main() {
    int y;
    y = blah(10);
    printf("y is %d\n",y);
}

```

Here is the corresponding machine/assembly code:

```

0804835c <blah>:
804835c:    55                push   %ebp
804835d:    89 e5            mov    %esp,%ebp
804835f:    83 ec 08        sub   $0x8,%esp
8048362:    8d 45 fc        lea   0xffffffff(%ebp),%eax
8048365:    50                push  %eax
8048366:    e8 11 ff ff ff  call  804827c <gets>
804836b:    ff 75 08        pushl 0x8(%ebp)
804836e:    8d 45 fc        lea   0xffffffff(%ebp),%eax
8048371:    50                push  %eax
8048372:    68 48 84 04 08  push  $0x8048448
8048377:    e8 20 ff ff ff  call  804829c <printf>
804837c:    8b 45 08        mov   0x8(%ebp),%eax
804837f:    40                inc   %eax
8048380:    c9                leave
8048381:    c3                ret

08048382 <main>:
8048382:    55                push   %ebp
8048383:    89 e5            mov    %esp,%ebp
8048385:    6a 0a            push  $0xa
8048387:    e8 d0 ff ff ff  call  804835c <blah>
804838c:    50                push  %eax
804838d:    68 51 84 04 08  push  $0x8048451
8048392:    e8 05 ff ff ff  call  804829c <printf>
8048397:    83 c4 10        add   $0x10,%esp
804839a:    c9                leave
804839b:    c3                ret

```

Note: leave is equivalent to the 2 instruction sequence:

```

movl %ebp,%esp
popl %ebp

```

Question 5(continued):

For the questions that follow, recall that:

- `gets` is a C library function that reads from standard input until a newline is found.
- IA32 machines are little endian.
- C strings are null terminated.
- Characters '0' through '9' have ASCII codes 0x30 through 0x39

Your answers will depend on the actual string typed in (that the function `gets` reads). The string typed in will be "01234567890123456789".

Assume we stop the program in function `blah()` right before execution of the `ret` instruction (at address 0x80048381). The `leave` instruction has already executed.

Question 5a (5 pts): What is the value in register `%ebp` ?

0x37363534

Question 5b (5 pts): What return address will be used by the `ret` instruction (where will `ret` jump to)?

0x31303938

Question 5c (5 pts): Using the same input as above, we run the program without stopping it. What will the output generated by the program be(what will be printed out)?

012345678901234567890 35343332
Segv...

Stack after call to `gets()`:

X	"2345"	0x35343332
return address	"8901"	0x31303938
old ebp	"4567"	0x37363534
buf	"0123"	0x33323130
?		
?		

Question 6 (20 pts): Given the definition of the function `cyclelen(int *a, int n)` shown below in C, write an IA32 assembly language subroutine (using the gcc calling conventions) that does the same computation. You must comment the code!

Remember that registers `ebx`, `esi` and `edi` are callee-save registers, so you must save/restore these if you use them!

```
int cyclelen(int *a, int n) {
    int len=0;
    int i=0;

    while (a[i]) {
        len ++;
        i = a[i];
    }
    return(len);
}
```

```
cyclelen:
    pushl   %ebp
    movl   %esp, %ebp
    xorl   %eax, %eax           # eax is len
    xorl   %ecx, %ecx          # ecx is i
    movl   8(%ebp), %edx       # edx is address of array a
loop:
    cmpl   $0, (%edx, %ecx, 4) # is a[i] == 0 ?
    je     done                # yes - goto done
    incl   %eax                # i++
    movl   (%edx, %ecx, 4), %ecx # ecx is now a[i]
    jmp    loop
done:
    leave
    ret
```