

Computer Organization

Fall 2003 Test #2

Name _____

There are 7 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 (20 pts): Given the definition of the function `vecsum(int *a, int n)` shown below in C, write a Y86 assembly language subroutine (using the gcc calling conventions) that does the same computation. Your assembly code can only include instructions from the Y86 instruction set. You must comment the code! Remember that registers `ebx`, `esi` and `edi` are callee-save registers.

```
int vecsum(int *a, int n) {
    int i,tot=0;

    for (i=0;i<n;i++)
        tot = tot + a[i];

    return(tot);
}
```

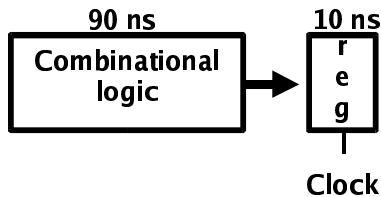
```
vecsum:    push    %ebp
           rrmovl  %esp,%ebp          # stack frame setup
           push   %ebx               # save ebx (callee save)

           irmovl  $0,$eax           # tot=0
           irmovl  $0,$edx           # i=0
loop:     mrmovl  0xC(%ebp),%ecx      # ecx = n
           subl   %edx,%ecx          # ecx = n-i
           je     done

           mrmovl  0x8(%ebp),%ecx    # ecx = a
           rrmovl  %edx,%ebx        # ebx = i
           addl   %ebx,%ebx         # ebx = 2*i
           addl   %ebx,%ebx         # ebx = 4*i
           addl   %ecx,%ebx         # ebx = &a[i]
           mrmovl (%ebx),%ebx       # ebx = a[i]
           addl   %ebx,%eax         # tot += a[i]
           movl   $1,%ebx           # i=i+1
           addl   %ebx,%edx         # i=i+1
           jmp    loop
done:     pop     %ebx               # restore ebx
           pop     %ebp
           ret
```

Question 2a (5 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). This means that the throughput will be 10 Million instructions/second. Assuming you switch to a 3 stage pipeline by splitting the combinational logic in to three equal parts, and all registers take 10ns to store results:

- How long will it take for a single instruction to execute in the pipelined implementation?
- Assuming the pipeline never stalls, what will the new throughput be (the number of instructions/second). Just show the calculation, you don't need to come up with a number.



Total time becomes $(30+10) * 3 = 120\text{ns}$

Throughput is $1,000,000,000/40$ instructions/second = 25 Million instructions/second

Question 2b (5 pts): Identify the *data* hazards in the following sequence of Y86 assembly instructions assuming the 5 stage pipeline we discussed in class (stages Fetch, Decode, Execute, Memory, Writeback). Identify the hazards any way you want (describe them, draw pictures, ...).

```
irmovl    $100, %eax
addl     %eax, %eax
movl     %eax, %ebx
```

eax from the irmovl instruction is not yet written when the add instruction goes through decode. The movl instruction will decode eax before either of the 2 previous instructions have written eax.

Question 2c (5 pts): Rewrite the above Y86 code with nops inserted that eliminate all the data hazards (with no unnecessary nops)

```
irmovl    $100, %eax
nop
nop
nop
addl     %eax, %eax
nop
nop
nop
movl     %eax, %ebx
```

Question 3 (40 pts): We want to add a new instruction to the Y86 sequential implementation. The new instruction will support an *indirect* unconditional jump named `jmpi`. The address to jump to is specified indirectly as a 4 byte absolute address. For example, assuming that memory location `0xaabbccdd` holds the value `0x012345678`, the following code will jump to memory location `$0x12345678`

```
jmpi *$0xaabbccdd
```

Question 3a (20 pts): The new Y86 instruction will named `jmpi` will be a five byte instruction as follows:

icode:ifun

E0	*Dest
-----------	--------------

Where `*Dest` is an immediate, 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	jmp_i
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>valC <- M₄[PC+1]</code>
Decode	<code>valA <- R[rA]</code> <code>valB <- R[%esp]</code>	
Execute	<code>valE <- valB + (-4)</code>	<code>valE <- valC + 0</code>
Memory	<code>M₄[valE] <- valA</code>	<code>valM <- M₄[valE]</code>
Write Back	<code>R[%esp] <- valE</code>	
PC update	<code>PC <- valP</code>	<code>PC <- valM</code>

Question 4b (20 pts): Modify the HCL expressions show below to support this new instruction. You can assume that the symbol IJMPI 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 };

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

## Select input A to ALU
int aluA = [
    icode in { IRRMOVL, IOPL } : valA;
    icode in { IIRMOVL, IRMMOVL, IMRMOVL, IJMPI } : 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 } : valB;
    icode in { IRRMOVL, IIRMOVL, IJMPI } : 0;
];

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

## What address should instruction be fetched at
int new_pc = [
    icode == ICALL : valC;
    icode == IJXX && Bch : valC;
    icode == IRET : valM;
    icode == IJMPI : valM;
    1 : valP;
];
```

Question 4(15 pts): This question tests your knowledge of the stack and byte ordering. The following C code is compiled on an X86 machine (IA32 instruction set) running BSD:

```

void foo(char *x) {
    int buf[1];
    strcpy((char *)buf, x);
}

void callfoo() {
    foo("abcdefghi");
}

```

Here is the corresponding machine/assembly code:

```

0804848c <foo>:
804848c:    55                push   %ebp
804848d:    89 e5            mov    %esp,%ebp
804848f:    83 ec 18        sub   $0x18,%esp
8048492:    83 c4 f8        add   $0xffffffff8,%esp
8048495:    8b 45 08        mov   0x8(%ebp),%eax
8048498:    50                push  %eax
8048499:    8d 45 fc        lea  0xffffffffc(%ebp),%eax
804849c:    50                push  %eax
804849d:    e8 ae fe ff ff  call  8048350 <strcpy>
80484a2:    83 c4 10        add   $0x10,%esp
80484a5:    c9                leave
80484a6:    c3                ret

080484a8 <callfoo>:
80484a8:    55                push   %ebp
80484a9:    89 e5            mov    %esp,%ebp
80484ab:    83 ec 08        sub   $0x8,%esp
80484ae:    83 c4 f4        add   $0xffffffff4,%esp
80484b1:    68 07 85 04 08  push  $0x8048507
80484b6:    e8 d1 ff ff ff  call  804848c <foo>
80484bb:    83 c4 10        add   $0x10,%esp
80484be:    c9                leave
80484bf:    c3                ret

```

Question 4 (continued): Here are some notes that may be helpful in answering the questions below:

`strcpy(char *dst, char *src)` copies the string at address `src` (including the terminating `'\0'` character) to address `dst`. It does not check the size of the destination buffer.

Recall that x86 machines are Little Endian.

You will need to know the hex values of the following characters:

Character	Hex value	Character	Hex value
'a'	0x61	'f'	0x66
'b'	0x62	'g'	0x67
'c'	0x63	'h'	0x68
'd'	0x64	'i'	0x69
'e'	0x65	'\0'	0x00

Now consider what happens on when `callfoo` calls `foo` with the input string `"abcdefghi"`.

Question 4a (6 pts): List the contents of the following memory locations immediately after `strcpy` returns to `foo`. Each answer should be an unsigned 4-byte integer expressed as 8 hex digits.

`buf[0] = 0x64636261`

`buf[1] = 0x68676665`

`buf[2] = 0x08040069`

Question 4b (6 pts): Immediately before the `ret` instruction at address `0x080484a6` executes, what is the value of the frame pointer register `%ebp`?

`%ebp = 0x0x68676665`

Question 4c (6 pts): Immediately after the `ret` instruction at address `0x080484a6` executes, what is the value of the program counter register `%eip`?

`%eip = 0x0x08040069`

Question 5 (10 pts): Consider the following C code to convert a string to all upper case characters:

```
void upcase(char *s) {
    int i;

    for (i=0;i<strlen(s);i++) {
        if ((s[i]>='a') && (s[i]<='z') ) {
            s[i]+='A'-'a';
        }
    }
}
```

Question 5a (5 pts): Show an improved version of the code that will run faster (on average) that does not involve loop unrolling (it must still result in the same exact modification to the string s).

```
void upcase(char *s) {
    int i;
    int len = strlen(s);

    for (i=0;i<len;i++) {
        if ((s[i]>='a') && (s[i]<='z') ) {
            s[i]+='A'-'a';
        }
    }
}
```

Question 5b (5 pts): Show an improved version of the code that will run faster (on average) that does involve loop unrolling (it must still result in the same exact modification to the string s).

```
void upcase(char *s) {
    int i;
    int len = strlen(s);

    for (i=0;i<len;i+=2) {
        if ((s[i]>='a') && (s[i]<='z') ) {
            s[i]+='A'-'a';
        }
        if ((s[i+1]>='a') && (s[i+1]<='z') ) {
            s[i+1]+='A'-'a';
        }
    }
}

Assumes that the string length is divisible by
2.
```