

CSE30 — Midterm

Yee

Fall '99

Name and Class Account Login: _____ Answer Key _____

There are a total of 17 questions on 17 pages. There are 102 points possible. It is unlikely that you will finish the entire exam. Wait until the instructor has passed out exams to everybody before you start. Advice: skim through the entire test to determine which of the problems you can solve quickly and work on those first, rather than getting stuck on a hard problem early and wasting too much of your time on it.

When you can start, you should first make sure that you have all the pages, and write your name and your login name on the first page, and your login name on the top of *all subsequent pages*. Pages of this exam will be separated and graded separately — if you fail to write your name at the top of a page, you will not receive credit for answers on that page. **Write clearly:** if we cannot read your handwriting or your pencil smudges, you will not properly get credit for your answers.

This exam is closed book. You are allowed a single sheet of notes. You may look at your *own* notes all you want. You may **not** look at anybody else's books, notes, exam, or otherwise obtain help from another human being, artificial intelligence, metaphysical entity, or space alien. If we see your eyeballs wandering, you will get a zero for the exam. If you must look away from your exam/notes to think, look up at the ceiling / into space or close your eyes.

No electronic computation aids are allowed.

Problem	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Total
Score																		
Possible	3	9	4	10	7	2	3	8	4	12	8	12	5	2	4	8	1	102

- 1 (Number representation) Given a number n represented as string of k digits $d_0, d_1 \dots d_{k-1}$ in base b , where $0 \leq d_j < b$ for $j = 0, \dots, k-1$, written as $n = d_{k-1}d_{k-2} \dots d_2d_1d_0_{(b)}$.

- 1: What is n written in a (base-free) mathematical notation (e.g., a summation).
- 2: Also write down the integer part of n/b^3 as a string of digits.

(3pt)

- 1: The number is

$$n = \sum_{i=0}^{k-1} d_i b^i$$

- 2: When this number is divided by b^3 , it is just $\lfloor n/b^3 \rfloor = d_{k-1}d_{k-2} \dots d_4d_3_{(b)}$, i.e., we omit the last three digits.

- 2 (Base Conversion) Perform the following base conversions. For bases larger than $16_{(10)}$, the individual digits are written as parenthesized base 10 numbers, e.g., $(17)(30)_{(72)} = 17_{(10)} \times 72_{(10)} + 30_{(10)}$.

- 1: $\text{CAFEF00D}_{(16)} = ?_{(2)}$ (note the zero digit "0" is not the letter "O".)
- 2: $37673335276_{(8)} = ?_{(16)}$
- 3: $2120102020211011220021122001_{(3)} = ?_{(27)}$ (use $[d_1][d_0]$ to write the base 27 digits to denote $d_1 * 27 + d_0$, where the d_i "digits" are in decimal)

(9pt, 3 each)

- 1: $\text{CAFEF00D}_{(16)} = 1100\ 1010\ 1111\ 1110\ 1111\ 0000\ 0000\ 1101_{(2)}$
- 2: $37673335276_{(8)} = 011\ 111\ 110\ 111\ 011\ 011\ 011\ 101\ 010\ 111\ 110_{(2)}$
 $= 0\ 1111\ 1110\ 1110\ 1101\ 1011\ 1010\ 1011\ 1110_{(2)}$
 $= \text{FEEDBABE}_{(16)}$
- 3: $2120102020211011220021122001_{(3)} = [2][15][11][6][22][4][24][7][17][1]_{(27)}$

- 3 (Micro-architecture) What is the purpose of a cache? Explain what it is, how it achieves its purpose, the expected speed-up, and what factors influence how well it achieves this.

(4pt)

A cache improves overall performance of the computer by transparently making memory accesses faster — most of the time. It is fast memory that is physically located closer to the processor, and contains copies of portions of main memory contents. When the processor attempts to access cached memory, a *cache hit* occurs and the access is fast, since the memory access does not have to travel to the DRAM over the system bus; when the processor attempts to access memory that has not been cached, a *cache miss* occurs and the cache forwards the access to the DRAM, saving (caching) a copy of the result for later use.

If p is the probability of a cache hit, then the expected memory access time is $p \cdot t_{\text{hit}} + (1 - p) \cdot t_{\text{miss}}$; typically caches are designed to have a very high p (depends on size and program mix), e.g., 0.99, and $t_{\text{hit}} \ll t_{\text{miss}}$, so including caches greatly improves overall performance of computers.

Factors that influence how well the cache speeds up programs include the size of the cache, the cache response times, and the program mix. By implementing the cache from faster (and more expensive) memory would improve the t_{hit} value. Vector programs would not benefit much from a data cache.

- 4 (Number representation) Compute the two's complement of the following numbers stored in 16-bit registers:

1: 0x5141

2: 0x8576

Negate the following numbers stored in 16-bit registers:

3: 0xF35

4: 0xCAFE

5: 0x7FFF

In all 5 cases, mark which results would be interpreted as a negative number when interpreted as a 16-bit two's complement number.

(10pt, 2 each)

Taking the two's complement of a number is the same as negating it.

1: 0x5141 \rightarrow 0xAE6F (negative)

2: 0x8576 \rightarrow 0x7A8A

3: 0xF35 \rightarrow 0xF0CB (negative)

4: 0xCAFE \rightarrow 0x3502

5: 0X7FFF \rightarrow 0x8001 (negative)

- 5 (Number representation) Suppose you have a number in a 32-bit register, and its hexadecimal representation is 0x8000 0000. Is this number positive or negative when viewed as a two's complement number? What happens when you negate it? Is the result of the negation positive or negative when viewed as a two's complement number?

(7pt)

The number is negative when viewed as a two's complement number, since the high-order bit is set. The result from negating it is also 0x8000 0000. An overflow occurred during the negation, because the result can not be represented as a 32-bit two's complement number (it's too big). The result would be interpreted as the same as the original negative number.

- 6 What is the name of your favorite film?

(2pt)

This is a "freebie." Anything is fine. Mine is *Cassablanca*.

- 7 (One Instruction Computer) Define the `subz` instruction in pseudo-code.

(3pt)

`subz a,b,c`

is equivalent to the following C-like pseudo-code:

```
mem[a] = mem[a] - mem[b];
if (mem[a] == 0) {
    pc = c;
} else {
    pc = pc + 1;
}
```

- 8 (One Instruction Computer) Convert the following OIC program to hexadecimal, machine-code notation. Your translation must be acceptable to the oic program when run as

`oic foo.txt.oic foo.data.oic`

foo.masm:

```

                                .data 0x8000
A:                               .word 0xa
B:                               .word 0xb
C:                               .word 0xc
D:                               .word 0xd
out:                             .word 0x100
tmp:                             .word 0
one:                             .word 1
zero:                            .word 0
concept:                          .word -triple(tmp,tmp,done)

                                .text 0x0
main:                             subz out,out,next
                                subz out,A,next
                                subz out,B,next
                                subz out,C,next
                                subz D,zero,done
                                subz D,one,main
done:                             subz tmp,tmp,done

```

(8pt)

Scoring: 3 points for the address assignment; 4 points for generating the machine code; 3 points for getting the -triple right.

foo.txt.oic:

```

0x0                               ;                               .text 0x0
800480040001                      ; 0x0   main:          subz out,out,next
800480000002                      ; 0x1                               subz out,A,next
800480010003                      ; 0x2                               subz out,B,next
800480020004                      ; 0x3                               subz out,C,next
800380070006                      ; 0x4                               subz D,zero,done
800380060000                      ; 0x5                               subz D,one,main
800580050006                      ; 0x6   done:         subz tmp,tmp,done

```

foo.data.oic:

```

0x8000                            ;                               .data 0x8000
00000000000a                      ; 0x8000 A:          .word 0xa
00000000000b                      ; 0x8001 B:          .word 0xb
00000000000c                      ; 0x8002 C:          .word 0xc
00000000000d                      ; 0x8003 D:          .word 0xd
0000000000100                     ; 0x8004 out:        .word 0x100
000000000000                      ; 0x8005 tmp:        .word 0
000000000001                      ; 0x8006 one:        .word 1
000000000000                      ; 0x8007 zero:       .word 0
7ffa7ffa7ffa                      ; 0x8008 concept:   .word -triple(tmp,tmp,done)

```

9 (Macro Assembly) What's the difference between using macros and subroutines?

(4pt)

Macros expand “in place” — the macro bodies take the place of each invocation. Unlike subroutines, no “call” sequence is needed, so the use of macros is very efficient in terms of execution time. They do, however, use up more space in memory. For each subroutine, there's only one copy of it in memory. Calling a subroutine is more expensive from the point of view of execution time, but cheaper from the point of view of instruction memory space consumed.

Using subroutines also permits the implementation of recursive algorithms directly. Macro assembly languages do not allow this, since the recursion depth is input dependent, and the macro would just recursively expand indefinitely, until all of memory is consumed by the macro body.

(This page intentionally left blank.)

- 10 (Macro Assembly) Expand the macros in the following macro assembly program, giving “pure” assembly code. Do *not* convert to machine code. Use the space on the next sheets if needed.

```
move:      .macro  dst,src
           subz   tmp,tmp,next
           subz   tmp,src,next
           subz   dst,dst,next
           subz   dst,tmp,next
           .endmacro

entry:     .macro  name
           .word  0          # ret instruction immediately before entry
name:      .endmacro
exit:     .macro  entry
           subz   zero,zero,entry-1
           .endmacro

call:     .macro  entry
           move   entry-1,RetInst
           subz   zero,zero,entry
retpt:    # on following instruction
           .data
RetInst:  .word   triple(zero,zero,retpt)
           .text
           .endmacro

end:      .macro
end_done: subz   zero,zero,end_done
           .endmacro

           .data
main_out: .word  0
main_p1a: .word  0x1a
main_p1b: .word  0x1b
main_p2a: .word  0x2a
main_p2b: .word  0x2b
mult_prod: .word  0      # mult output
mult_a:   .word  0      # mult input 1
mult_b:   .word  0      # mult input 2
mult_sum: .word  0
mult_i:   .word  0
zero:     .word  0
negone:   .word  -1
tmp:      .word  0

           .text
main:     move   mult_a,main_p1a
           move   mult_b,main_p1b
           call   mult
           move   main_out,mult_prod
           move   mult_a,main_p2a
           move   mult_b,main_p2b
           call   mult
           subz   main_out,mult_prod,next
           end
           ; continues on next page
```

```

        entry    mult
        subz    mult_sum,mult_sum,next
        subz    mult_i,mult_i,next
        subz    mult_i,mult_a,next
        subz    mult_i,zero,mult_done
mult_loop: subz    mult_sum,mult_b,next
        subz    mult_i,negone,mult_done
        subz    zero,zero,mult_loop
mult_done: subz    mult_prod,mult_prod,next
        subz    mult_prod,mult_sum,next
        exit    mult

```

(12pt)

grading: basic macro expansion (all macros expanded fully) 6 pts; local labels handled properly, including multiple expansions of the retpt triples etc, 6 pts.

```

        .data
main_out: .word 0
main_p1a: .word 0x1a
main_p1b: .word 0x1b
main_p2a: .word 0x2a
main_p2b: .word 0x2b
mult_prod: .word 0      # mult output
mult_a:    .word 0      # mult input 1
mult_b:    .word 0      # mult input 2
mult_sum:  .word 0
mult_i     .word 0
zero:      .word 0
negone:    .word -1
tmp:       .word 0

        .text
main:    subz    tmp,tmp,next
        subz    tmp,main_p1a
        subz    mult_a,mult_a,next
        subz    mult_a,tmp,next
        subz    tmp,tmp,next
        subz    tmp,main_p1b,next
        subz    mult_b,mult_b,next
        subz    mult_b,tmp,next
        subz    tmp,tmp,next
        subz    tmp,RetInst_00,next
        subz    mult-1,mult-1,next
        subz    mult-1,tmp,next
        subz    zero,zero,mult

retpt_00:
RetInst_00: .word  triple(zero,zero,retpt_00)
        .text
        subz    tmp,tmp,next
        subz    tmp,mult_prod,next

```

```

                                subz    main_out,main_out,next
                                subz    main_out,tmp,next
                                subz    tmp,tmp,next
                                subz    tmp,main_p2a,next
                                subz    mult_a,mult_a,next
                                subz    mutl_a,tmp,next
                                subz    tmp,tmp,next
                                subz    tmp,main_p2b,next
                                subz    mult_b,mult_b,next
                                subz    mult_b,tmp,next
                                subz    tmp,tmp,next
                                subz    tmp,RetInst_01,next
                                subz    mult-1,mult-1,next
                                subz    mult-1,tmp,next
                                subz    zero,zero,mult

retpt_001:
                                .data
RetInst_01:                    .word    triple(zero,zero,retpt_001)
                                .text
                                subz    main_out,mult_prod
end_done_000:                  subz    zero,zero,end_done_000

                                .word    0
mult:                          subz    mult_sum,mult_sum,next
                                subz    mult_i,mult_i,next
                                subz    mult_i,mult_a,next
                                subz    mult_i,zero,mult_done
mult_loop:                     subz    mult_sum,mult_b,next
                                subz    mult_i,negone,mult_done
                                subz    zero,zero,mult_loop
mult_done:                    subz    mult_prod,mult_prod,next
                                subz    mult_prod,mult_sum,next
                                subz    zero,zero,mult-1
```


- 11 (One Instruction Computer) Write an oic assembly language program to compute $\sum_{i=1}^N i$ where N is stored in memory location 0x8000, and the result is placed in location 0x8001. The program should start at location 0x0.

(8pt)

```

                                .data  0x8000
N:                               .word  0          # will be overwritten
output:                          .word  0
i:                               .word  0
zero:                            .word  0
negone:                          .word -1
                                .text  0x0
                                subz   output,output,next
                                subz   i,i,next
                                subz   i,N,done
loop:                            subz   output,i,next
                                subz   i,negone,done
                                subz   zero,zero,loop
done:                            subz   i,i,done
```

- 12 (One Instruction Computer) Write an oic macro assembly language function to copy memory from one (fixed) memory region starting at the label `src` to another (fixed) region starting at the label `dst`. The number of elements to copy is given in memory location `N`. The value at `N` is non-negative. If you wish to use macros, you must define them in your answer. Your function must work even if called multiple times.

(12pt)

```
.data
tmp:      .word 0
remain:   .word 0
negone:   .word -1
zero:     .word 0
firstload: .word -triple(tmp,src,next)
clearstore: .word -triple(dst,dst,next)
firststore: .word -triple(dst,tmp,next)
bumpload:  .word -triple(0,1,0)
bumpclear: .word -triple(1,1,0)
bumpstore: .word -triple(1,0,0)
.text
src2dstRet: .word 0          # goto caller
src2dst:    subz remain,remain,next
            subz remain,N,next
            subz remain,zero,src2dstRet
            subz load,load,next
            subz load,firstload,next
            subz cleardst,cleardst,next
            subz cleardst,clearstore,next
            subz store,store,next
            subz store,firststore,next
loop:      subz tmp,tmp,next
load:      .word 0          # subz tmp,src+i,next
cleardst:  .word 0          # subz dst+i,dst+i,next
store:     .word 0          # subz dst,tmp,next
            subz remain,negone,src2dstRet
            subz load,bumpload,next
            subz cleardst,bumpclear,next
            subz store,bumpstore,next
            subz tmp,tmp,loop
```

- 13 (MIPS) What are the MIPS `t` and `s` registers used for? In what way are they different from each other?

(5pt)

Both the `t` and `s` registers are for temporaries. The `t` registers are *caller-saved* registers, since by convention a subroutine is allowed to use them, so the caller must save their contents if they are needed. The `s` registers are *callee-saved* registers; a routine can call a subroutine and expect that these registers' contents will be preserved when the subroutine returns — the callee will save their contents prior to using these registers and restore them before returning.

- 14 (RISC and CISC) Give an example of a processor with a RISC architecture and an example of processor with a CISC architecture.

(2pt)

grading: MIPS Rxxx, Motorola/IBM/Apple PowerPC, Compaq/DEC Alpha are all RISCs. Intel x86, Motorola 68000, VAX, ... are all CISCs.

The MIPS architecture is a RISC, and the R2000 is an implementation of that architecture; a 486, Pentium, Pentium II are processors that implements the x86 (or IA-32) architecture, which is a CISC architecture.

- 15 (Converting C to MIPS assembly) Convert the following C code to MIPS assembly. You may assume that the C variables are in the correspondingly named registers. Indicate where the code that precedes the loop, the code that comprise the body of the loop, and the code that follows the loop would be located in your equivalent MIPS code (as indicated by the C comments). Efficiency matters.

```
int    t0, t1, *t2;

/* code that precedes loop */
for (t0 = 0, t2 = &globalIntArray[0]; t0 <= t1; t0++, t2++) {
    /* loop body */
}
/* code that follows loop */
```

(4pt)

```
    # code that precedes loop
    li $t0,0
    la $t2,globalIntArray
    b test
loop: # loop body
    add $t0,$t0,1
    add $t2,$t2,4
test: ble $t0,$t1,loop
done: # code that follows loop
```

- 16 (Stack Frames) Write the MIPS assembly language equivalent for the following function:

```
int funnyfact(int n)
{
    if (n <= 2) return 1;
    else return n * funnyfact(n-2);
}
```

(8pt)

```
funnyfact:    subu $sp, $sp, 12
             sw $fp, 4($sp)
             addu $fp, $sp, 12
             sw $ra, -4($fp)
             bgt $a0, 2, rec_fib
             li $v0, 1
             b funnyfact_done

rec_funnyfact:
             sw $a0, 0($fp)
             subi $a0, $a0, 2
             jal funnyfact
             lw $a0, 0($fp)
             mul $v0, $v0, $a0

funnyfact_done:
             lw $ra, -4($fp)
             lw $fp, 4($sp)
             addu $sp, $sp, 12
             jr $ra
```

- 17 Write your name and class account legibly on all the pages.
(1pt)