CSE30 — Final

Yee **Fall '99**

Name and Class Account Login: Answer Key

There are a total of 15 questions on 12 pages. There are 100 points possible. It is unlikely that you will finish the entire exam. Wait until the instructor/proctors has passed out exams to everybody before you start. The questions are not in any particular order. 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 two sheets 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	**Total**
Score																
Possible	3	9	12	5	2	10	5	2	7	15	8	8	5	8	1	100

1 (Number representation) Given a number n represented as string of k digits $d_0, d_1 \ldots d_{k-1}$ in base b, where $0 \leq d_j < b$ for $j = 0, \ldots, k-1$, written as $n = d_{k-1}d_{k-2}\ldots d_2 d_1 d_0\,_{(b)}$.

1: What is n written in a (base-free) mathematical notation (e.g., a summation).

2: Also write down $n \times b^3$ as a string of digits.

(3pts)

1: The number is

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

2: When this number is multiplied by b^3, it is just $n \times b^3 = d_{k-1}d_{k-2}\ldots d_1 d_0 000_{(b)}$, i.e., we add three trailing zeros.

2 (Base Conversion) Perform the following base conversions.

1: $\texttt{11111110111010111111000000001011}_{(2)} =?_{(16)}$

2: $\texttt{24000330013}_{(8)} =?_{(16)}$

3: $\texttt{55555555}_{(16)} =?_{(8)}$

(9pts, 3 each)

1:
$$\begin{aligned}\texttt{11111110111010111111000000001011}_{(2)} &= \texttt{1111 1110 1110 1011 1111 0000 0000 1011}_{(2)}\\ &= \texttt{FEEBF00B}_{(16)}\end{aligned}$$

2:
$$\begin{aligned}\texttt{24000330013}_{(8)} &= \texttt{10 100 000 000 000 011 011 000 000 001 011}_{(2)}\\ &= \texttt{1010 0000 0000 0001 1011 0000 0000 1011}_{(2)}\\ &= \texttt{A001B00B}_{(16)}\end{aligned}$$

3:
$$\begin{aligned}\texttt{55555555}_{(16)} &= \texttt{0101 0101 0101 0101 0101 0101 0101 0101}_{(2)}\\ &= \texttt{01 010 101 010 101 010 101 010 101 010 101}_{(2)}\\ &= \texttt{125 2525 2525}_{(8)}\end{aligned}$$

3 Suppose we number the bits of a 32-bit word in the usual way, i.e., the least significant bit is b_0 and the most significant bit is b_{31}. Give an efficient MIPS instruction sequence to compute $oddbits = b_1 + b_3 + \ldots + b_{31}$, and $evenbits = b_0 + b_2 + \ldots + b_{30}$ of input word `$a0`. You may use any of the `t` registers for scratch computation. After your MIPS instruction sequence runs, the value *evenbits* should be in register `$v0`, and the value *oddbits* should be in register `$v1`.

(12pts)

```
li      $t0,0x55555555
srl     $t1,$a0,1
and     $t2,$a0,$t0     # only the 16 even bits
and     $t1,$t1,$t0     # only the 16 odd bits

li      $t0,0x33333333
srl     $t3,$t2,2
srl     $t4,$t1,2
addu    $t2,$t2,$t3
addu    $t1,$t1,$t4
and     $t2,$t2,$t0     # 8 four-bit sub-registers
and     $t1,$t1,$t0     # possible values are 0,1,2

li      $t0,0x0f0f0f0f
srl     $t3,$t2,4
srl     $t4,$t1,2
addu    $t2,$t2,$t3
addu    $t1,$t1,$t4
and     $t2,$t2,$t0     # 4 8-bit sub-registers
and     $t1,$t1,$t0     # 0-4

li      $t0,0x00ff00ff
srl     $t3,$t2,4
srl     $t4,$t1,2
addu    $t2,$t2,$t3
addu    $t1,$t1,$t4
and     $t2,$t2,$t0     # 8 4-bit sub-registers
and     $t1,$t1,$t0     # 0-8

li      $t0,0xffff
srl     $t3,$t2,8
srl     $t4,$t1,8
addu    $t2,$t2,$t3
addu    $t1,$t1,$t4
and     $v0,$t2,$t0
and     $v1,$t1,$t0
```

4 (Micro-architecture) What does the Memory Management Unit (MMU) do?
(5pts)

The MMU translates virtual addresses to physical addresses. It also enforces memory protection, so that processes can only access their own memory: one process cannot access the memory belonging to another process.

5 (One Instruction Computer) Define the `subz` instruction in pseudo-code.
(2pts)

```
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;
}
```

6 (Macro Assembly) Expand the macros in the following macro assembly program, giving "pure" assembly code. Do *not* convert to machine code.

```
                .data
one:            .word   1
zero:           .word   0
a:              .word   15
b:              .word   20
c:              .word   7
d:              .word   99

foo:            .macro  a,b,c
again:          subz a,b,next
                subz c,one,done
                subz zero,zero,again
done:
                .endmacro

                .text
                foo     b,c,a
                foo     d,c,b
done:           subz    zero,zero,done
```

(10pts)

```
                .data
one:            .word   1
zero:           .word   0
a:              .word   15
b:              .word   20
c:              .word   7
d:              .word   99

                .text
again0:         subz    b,c,next
                subz    a,one,done0
                subz    zero,zero,again0
done0:
again1:         subz    d,c,next
                subz    b,one,done1
                subz    zero,zero,again1
done1:
done:           subz    zoer,zero,done
```

7 (One Instruction Computer) Write an `oic` assembly language program to compute $\sum_{i=1}^{N} 2 \times i$ where N is stored in memory location `0x8000`, and the result is placed in location `0x8001`. The program should start at location `0x0`.

(5pts)

```
                .data   0x8000
N:              .word   0       # input value; 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    output,i,next
                subz    i,negone,done
                subz    zero,zero,loop
done:           subz    i,i,done
```

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

(2pts)

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.

9 (Efficiency) When should a loop be unrolled?

```
int     t0, t1, *t2;

/* code that precedes loop */
for (t0 = 0; t0 < N; t0++) {
        /* loop body */
}
/* code that follows loop */
```

(7pts)

If the loop body is short, then the loop control overhead, as a fraction of the total execution time for one iteration, is large. In this case, if efficiency requirements dictate, the loop should be unrolled so that the loop control overhead is amortized over the execution time of several copies of the loop body.

If the loop body is long or if the number of iterations is typically small anyway, then unrolling won't help much.

(Loop unrolling should be one of the later steps in the optimization process: first improve the algorithm, then if still more efficient code is needed, apply common subexpression elimination and constant folding if the compiler can't do it, and loop unrolling.)

10 (Efficiency) Write a MIPS assembly language function that is functionally equivalent to the following C function. Make it as efficient as you can; you should *not* just do a verbatim translation.

```
void    wordcopy(unsigned int   *dst,
                 unsigned int   *src,
                 int            count)
{
        int     i;

        for (i = 0; i < count; i++) {
                dst[i] = src[i];
        }
}
```

(15pts)

You don't have to unroll the loop 8 times as in this sample solution. This gives a per-element cost of $20\frac{\texttt{count}}{8} = 2.5\texttt{count}$. The simple, non-unrolled solution has a per-element cost of about 7`count`.

```
                .data
jtab:           .word   L7,L6,L5,L4,L3,L2,L1,L0
                .text
wordcopy:       and     $t0,$a2,7
                srl     $t0,$t0,2
                lw      $t0,jtab($t0)
                jr      $t0
L7:             lw      $t0,0($a1)
                sw      $t0,0($a0)
                addu    $a1,$a1,4
                addu    $a0,$a0,4
L6:             lw      $t0,0($a1)
                sw      $t0,0($a0)
                addu    $a1,$a1,4
                addu    $a0,$a0,4
L5:             lw      $t0,0($a1)
                sw      $t0,0($a0)
                addu    $a1,$a1,4
                addu    $a0,$a0,4
L4:             lw      $t0,0($a1)
                sw      $t0,0($a0)
                addu    $a1,$a1,4
                addu    $a0,$a0,4
L3:             lw      $t0,0($a1)
                sw      $t0,0($a0)
                addu    $a1,$a1,4
                addu    $a0,$a0,4
L2:             lw      $t0,0($a1)
                sw      $t0,0($a0)
                addu    $a1,$a1,4
                addu    $a0,$a0,4
L1:             lw      $t0,0($a1)
                sw      $t0,0($a0)
```

```
                addu    $a1,$a1,4
                addu    $a0,$a0,4
L0:             srl     $a2,3
                beq     $a2,$zero,done
again:          lw      $t0,0($a1)
                sw      $t0,0($a0)
                lw      $t1,4($a1)
                sw      $t1,4($a0)
                lw      $t2,8($a1)
                sw      $t2,8($a0)
                lw      $t3,12($a1)
                sw      $t3,12($a0)
                lw      $t4,16($a1)
                sw      $t4,16($a0)
                lw      $t5,20($a1)
                sw      $t5,20($a0)
                lw      $t6,24($a1)
                sw      $t6,24($a0)
                lw      $t7,28($a1)
                sw      $t7,28($a0)
                addu    $t1,$t1,32
                addu    $t0,$t0,32
                subu    $a2,$a2,1
                bgt     $a2,$zero,again
done:           jr      $ra
```

11 (Stack Frames) Write the MIPS assembly language equivalent for the following C function:

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

(8pts)

```
                # frame: fp, funny(n-2), ra, n
funny:          subu    $sp, $sp, 16
                sw      $fp, 4($sp)
                addu    $fp, $sp, 16
                sw      $ra, -4($fp)
                bgt     $a0, 2, rec_fib
                li      $v0, 1
                b       funny_done
rec_funny:
                sw      $a0, 0($fp)
                subu    $a0, $a0, 2
                jal     funny
                sw      $v0, -8($fp)
                lw      $a0, 0($fp)
                subu    $a0, $a0, 1
                jal     funny
                lw      $a0, -8($fp)
                mul     $v0, $v0, $a0
                lw      $a0, 0($fp)
                mul     $v0, $v0, $a0
funny_done:
                lw      $ra, -4($fp)
                lw      $fp, 4($sp)
                addu    $sp, $sp, 16
                jr      $ra
```

12 (Efficiency) Give an efficient MIPS implementation of the following C expression:

```
v0 = 24 * s0 + 9 s1;
```

(8pts)

```
sll     $t0,$s0,4      # 16 s0
sll     $t1,$s0,3      # 8 s0
addu    $v0,$t0,$t1    # 24 s0
sll     $t0,$s1,3      # 8 s1
addu    $v0,$v0,$t0
addu    $v0,$v0,$s1
```

13 Design questions.

1: How would an implementation of `bitblt` change if the source and destination regions were not allowed to overlap?

2: How would an implementation of `bitblt` change if the display was 24-bit true color instead of black-and-white bitmapped? (A true-color display uses a word (4 bytes) per pixel for intensity values of the red, green, and blue color components, expressed as three separate 8-bit values, one for each of the first 3 bytes; the fourth byte is ignored.)

(5pts)

1: There wouldn't be a need to divide up into the four major cases to decide which "direction" to do the copying. Shifting still

2: There wouldn't be a need for any shifting to align pixel values within words, since every pixel is word aligned.

14 Describe the MIPS R2000 pipeline and what happens in each pipeline stage for the instruction `lw $t0,-4($fp)`.

(8pts)

IF: instruction fetch; ID: instruction decode – figure out it is a load word, send immediate displacement -4 to ALU, tell the register file to send the value in `$fp` to the ALU; EX: execution – the ALU adds the -4 and the value from `$fp`; MEM: memory operation – the computed address (ALU's output) is sent to the cache to perform the memory load; WR: write register – the returned memory word is written to `$t0`.

15 Write your class account legibly on all the pages.

(1pt)