# CSE30 — Final

**Yee** **Fall '98**

Name and last two letters of login: Answer Key

There are a total of 18 questions on 12 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 and tells you to start before opening the exam. *Gratuitous 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 some problem early and wasting too much of your time on it. Also note how much each problem is worth when you're optimizing the use of your time.*

When you can start, you should first make sure that you have all the pages, and write your name and your login name at the top of first page, and **PRINT IN CAPITAL LETTERS the last two letters of your login name on the top of all subsequent pages**. This exam will be unstapled and the pages graded separately — if you fail to write the last two letters of your login 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** get credit for your answers.

This exam is closed book. You are allowed only the Larus handout and two sheets of notes. You may look at your *own* notes all you want. You may **not** look at anybody else's notes, exam, or otherwise obtain help from another human being, artificial intelligence, metaphysical entity, or space alien. Please refrain from using any ESP abilities that you may have. 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 or 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 | 18 | **Total** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Score** | | | | | | | | | | | | | | | | | | | |
| **Possible** | 3 | 3 | 6 | 5 | 6 | 4 | 5 | 5 | 1 | 8 | 10 | 4 | 8 | 8 | 5 | 10 | 10 | 1 | 102 |

1. [**Number representation**] Given a number $n$ represented as string of $k$ digits $d_{k-1}, d_{k-2}, \ldots, d_0$ in base $b$, where $0 \le d_j < b$ for all $j = 0, \ldots, k-1$, 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 (i.e., a summation of the variables). (2) Also write down the integer part of $n/b^2$ as a string of digits in whatever base you like.
(3pt)

Ans:

(1) The number is

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

(2) It is easiest to continue using base $b$. The result is just $\lfloor n/b^2 \rfloor = d_{k-1}d_{k-2}\ldots d_2\,_{(b)}$, i.e., we omit the last two digits.

2. [**Number representation**] Given a number $n \ge 0$ in a MIPS machine register `$t0`, how do we efficiently compute (1) $n$ mod 16? (2) $\lfloor \frac{n}{32} \rfloor$ (the integer part of the division)?
(3pt)

Ans:

(1) For $n$ mod 16, we use the instruction `and $t1, $t0, 15`, and
(2) for $\lfloor \frac{n}{32} \rfloor$, we use the instruction `srl $t1, $t0, 5`.

3. [**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: $\texttt{CAFEBEEF}_{(16)} = ?_{(2)}$

2: $(53)(13)(28)_{(81)} = ?_{(3)}$

3: $27673757255_{(8)} = ?_{(16)}$

(6pt, 2 each)

Ans:

1: $\texttt{CAFEBEEF}_{(16)} = 1100\,1010\,1111\,1110\,1011\,1110\,1110\,1111_{(2)}$

2: $(53)(13)(28)_{(81)} = 1222\,0111\,1001_{(3)}$

3:

$$\begin{aligned} 27673757255_{(8)} &= 010\,111\,110\,111\,011\,111\,101\,111\,010\,101\,101_{(2)} \\ &= 0\,1011\,1110\,1110\,1111\,1101\,1110\,1010\,1101_{(2)} \\ &= \texttt{BEEFDEAD}_{(16)} \end{aligned}$$

4. [**Assembler Directives**] What does the `.align` MIPS assembly directive do?
(5pts)

Ans:

The `.align` directive aligns memory. It takes a numeric argument $n$, and forces the next data element to start at an address that is divisible by $2^n$. Equivalently, the address of the next data element will have all zeros in the low order $n$ bits.

5. [**One Instruction Computer**] Covert the following OIC program to hexidecimal, machine-code notation. Your translation must be acceptable to the `oic` program when run as

% `oic -e 0x100` *yourfile.oic*

You may write the answer next to the listing if that helps.

```
A:      .equ 0x0
B:      .equ 0x1
C:      .equ 0x2
D:      .equ 0x3
out:    .equ 0x4
        .text
        .org 0x100
main:   subge out,out,next
        subge out,A,next
        subge out,B,next
        subge C,zero,ldone
        subge C,one,main
ldone:  subge D,zero,done
        subge D,one,main
done:   subge tmp,tmp,done
tmp:    .word 0
one:    .word 1
zero:   .word 0
```

(6pt)

Ans:

```
0x100           ;                .org 0x100
0x000400040101  ; 0x100  main:   subge out,out,next
0x000400000102  ; 0x101          subge out,A,next
0x000400010103  ; 0x102          subge out,B,next
0x0002010a0105  ; 0x103          subge C,zero,ldone
0x000201090100  ; 0x104          subge C,one,main
0x0003010a0107  ; 0x105  ldone:  subge D,zero,done
0x000301090100  ; 0x106          subge D,one,main
0x010801080107  ; 0x107  done:   subge tmp,tmp,done
0x000000000000  ; 0x108  tmp:    .word 0
0x000000000001  ; 0x109  one:    .word 1
0x000000000000  ; 0x10a  zero:   .word 0
```

6. [**Macro Assembly**] What are the differences between using macros and using subroutines? Which is better? For what occasions?

(4pt)

---

Ans:

Macros expand "in place" — the macro bodies take the place of each invocation (1). Unlike subroutines, no "call" sequence is needed, so the use of macros is very efficient (1). They do, however, use up more space in memory (1). 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.

Subroutines also permit the implementation of recursive algorithms directly (1). 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.

---

7. [**RISC and CISC**] If RISC processor instructions are simpler and, in order to perform some computation, more instructions must be executed, why would running a program on a RISC processors be faster than on a CISC processor?

(5pts)

---

Ans:

The processor design for supporting those simpler instructions is also simpler, so RISC processors can run at a higher clock rate than CISC processors (3). Furthermore, the number of instructions per cycle can be higher with simpler, easier-to-pipeline instructions (2). So even though the total number of instructions that need to be executed is higher, the faster execution of the individual instructions more than compensate, resulting in faster overall execution time.

---

8. [**Register usage convention**] What is the MIPS calling convention and register usage convention? (5pts)

Ans:

When making a function call, the `$t` are caller-saved (1pt). The called function may overwrite their contents as needed. The `$a` registers (1) and the `$v` registers (1) are similarly caller-saved. The `$s` registers are callee-saved (1), so that the calling function may rely on those values remaining unchanged across the call. The `$sp` and `$fp` registers are also similarly callee-saved. The `$ra` register holds the return address of a call. Leaf function need not do anything special, but non-leaf functions must save their `$ra` values prior to calling another routine. (1 if any of these are discussed.)

9. [**Multiple choice**] When you go to bed at night, you

1: give thanks that you're a Computer Science major,

2: give thanks that you're not an English major,

3: worry about whether your multi-threaded program will deadlock,

4: wonder what is "deadlock",

5: think about how to speed up your latest program,

6: wonder how does "bit-slicing" works,

7: worry about your CSE 30 grade,

8: wish you had a BFG2000 to use on your professor, or

9: other (specify).

(1pts)

Ans:

Anything is okay.

10. [**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 preceeds 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. *Efficiency is very important.* You can use any of the `$t` registers not already mentioned to hold temporary values.

```
int t0, t1, *t2, t3;
code that precedes loop
for (t0 = 0, t2 = &globalIntArray[0], t3 = 0; t0 <= t1; t0 += 2, t2++) {
     t3 += 96 * *t2;
}
code that follows loop
```

(8pt)

Ans:

```
        code that precedes loop
        move $t0, $zero
        la $t2,globalIntArray
        move $t3, $zero
        b test                 # test at bottom (4pts)
loop:   lw $t9, 0($t2)
        sll $t8, $t9, 6        # 2^6 = 64
        sll $t9, $t9, 5        # 2^5 = 32
        add $t9, $t9, $t8      # 64 + 32 = 96 (5pts)
        add $t3, $t3, $t9
        add $t0,$t0,2
        add $t2,$t2,4
test:   ble $t0,$t1,loop
        code that follows loop
```

11. [**Stack Frames**] Write the MIPS assembly language equivalent for the following function:

```
int flab(unsigned int n)
{
      if (n <= 3) return 2 * n;
      else return 3 * flab(n-1) + flab(n-2) + flab(n-3);
}
```

(10pt)

Ans:

```
flab:       sub $sp, $sp, 16
            sw $fp, 4($sp)
            add $fp, $sp, 16
            sw $ra, -8($fp)
            bgt $a0, 3, rec_flab
            sll $v0, $a0, 1
            b flab_done
rec_flab:   sw $a0, 0($fp)
            sub $a0, $a0, 1
            jal flab
            sll $v1, $v0, 1
            add $v0, $v0, $v1
            sw $v0, -4($fp)
            lw $a0, 0($fp)
            sub $a0, $a0, 2
            jal flab
            lw $a0, -4($fp)
            add $v0, $v0, $a0
            sw $v0, -4($fp)
            lw $a0, 0($fp)
            sub $a0, $a0, 3
            jal flab
            lw $a0, -4($fp)
            add $v0, $v0, $a0
flab_done:  lw $ra, -8($fp)
            lw $fp, 4($sp)
            add $sp, $sp, 16
            jr $ra
```

(5 for getting the stack frames right, 5 for getting the recursion right.)

12. [**Pipelines**] Give the stages of the MIPS R2000 pipeline, describe what they are, and give details on what would occur in the various staged when executing the

```
lw $t0, 12($t1)
```

instruction.

(4pts)

---

Ans:

The MIPS R2000 pipeline has 5 stages. They are IF, RD, EX, MEM, and WR. The IF or Instruction Fetch stage fetches the instruction from cache and starts the decoding process. In the RD cycle, the instruction's register argument are read from the register file – latched as inputs to the ALU. In the example instruction's case, the contents of `$t1` is sent to the ALU, as well as the offset value of 12. In the EX stage, the ALU operation occurs. In the example instruction, the addition of 12 and the contents of `$t1` occurs to compute the address from which the memory load will occur. In the MEM stage, memory reads and writes occur. In the example, the ALU's result is sent to the cache as a memory address, which responds with the word value stored there. In the WR stage, the result obtained from the cache is written to the register file as `$t0`. (2 for naming the stages, 2 for describing what occurs where.)

---

13. [**Threads**] What are the differences between kernel and user-level threads? Define what they are and give their advantages and disadvantages.

(8pts)

---

Ans:

Kernel threads are virtual processors provided by the operating system. Context switches are "automatic" (2). Kernel threads can run simultaneously on several physical processors (2). The user of kernel threads does not normally need to do anything to time share the processor if the number of available physical processors is less than the number of threads. The kernel handles pre-emption.

User threads are also called coroutines or cooperative threads. The processor time is shared among the threads, but typically threads must explicitly call a yield routine to give up the processor; if a user thread blocks, e.g., due to I/O, all threads block (2). This means that programs using user-level threads cannot effectively make use of a multi-processor system. User threads are cheaper during context switches, since not all registers need to be saved — since using the yield routine would follow the standard register usage convention, the caller-saved registers need not be saved/restored across context swithces (2).

---

14. [**Threads**] Why are locks used even when non-preemptive user-level threads are used? (8pts)

Ans:

Even though non-preemptive user-level threads can provide mutual exclusion by simply not yielding, this causes problems. When no yielding is allowed for mutual exclusion, the lack of sharing of the CPU means that there is no concurrency (4). By using locks, threads that need exclusion can still yield, providing greater concurrency among the threads, and yet still maintain correctness (4).

(Additionally, there are modularity concerns, since a thread that should not yield due to exclusion cannot call routines without knowing whether those routines will yield.)

15. [**Efficiency**] What is the best technique for speeding up programs? Why? (5pts)

Ans:

Use a better algorithm (3). Using a better algorithm can speed up programs a lot more than by other techniques (2): a factor of 100 to 10,000 is often achievable when an algorithm change results in a runtime of $O(n \log n)$ instead of $O(n^2)$. Other techniques such as constant folding, common subexpression elimination, dead code elimination, loop unrolling, bit-slicing, etc. can each account for only a factor of 2 to 10.

16. [**Bit manipulation**] Suppose $gp contains the address of the following table:

```
addr_in_gp:  .word 0x0000FFFF
             .word 0xFF00FF00
             .word 0x00FF00FF
             .word 0xF0F0F0F0
             .word 0x0F0F0F0F
             .word 0x33333333
             .word 0xCCCCCCCC
             .word 0x55555555
             .word 0xAAAAAAAA
             .word 0xFFFF0000
```

Write an *efficient* version of the `popcount` function in MIPS assembly. Assume that multiply and remainder calculations are too expensive. You are not required to use this table of values. (Hint: the entire function can be done in 22 single-cycle instructions, including the return, if all accesses hit in the cache.)

(10pts)

Ans:

```
popcount:  srl $v0, $a0, 1
           lw $t0, 28($gp)         # 0x55555555
           and $v0, $v0, $t0       # basic shift
           and $a0, $a0, $t0       # mask,
           add $a0, $a0, $v0       # and add (8)

           srl $v0, $a0, 2
           lw $t0, 20($gp)         # 0x33333333
           and $v0, $v0, $t0
           and $a0, $a0, $t0
           add $a0, $a0, $v0

           srl $v0, $a0, 4
           add $a0, $a0, $v0       # masking after shift/add (1)
           lw $t0, 16($gp)         # 0x0F0F0F0F
           and $a0, $a0, $t0

           srl $v0, $a0, 8
           add $a0, $a0, $v0
           lw $t0, 8($gp)          # 0x00FF00FF
           and $a0, $a0, $t0

           srl $v0, $a0, 16
           add $v0, $a0, $v0
           andi $v0, $v0, 0xFFFF   # using and immediate (1)

           jr $ra
```

17. [**Bit Manipulation**] Suppose we have 32 pairs of unsigned 2-bit numbers packed into four registers: `$t1` and `$t0` contain the high and low order bits of the first numbers in the 32 pairs, and `$t3` and `$t2` contain the high and low order bits of the second numbers in the 32 pairs. Give the MIPS assembly language instructions that computes the pairwise sums of these numbers, so that the bits of the results are stored in `$t6`, `$t5`, and `$t4` in decreasing order of significance. You may use any of the other `$t` registers for scratch values. *Your code must correctly compute the sum. Efficiency is also very important.*
(10pts)

Ans:

The bit sliced operations are:

```
and $t5, $t0, $t2
xor $t4, $t0, $t2
and $t6, $t5, $t1
xor $t5, $t5, $t1
and $t7, $t5, $t3
xor $t5, $t5, $t3
or $t6, $t6, $t7
```

Here, `$t7` is a scratch register used to handle the carry out from the additions of the high order bits. We know that bits in the same bit position in `$t6` and `$d7` cannot be set simultaneously, since the largest numbers being added are 3 and 3.

18. [**Extra Credit?**] If a genie grants you one wish, what would you wish for?
(1pts)

Ans:

I'd wish for a genie who would grant me more wishes! (Anything is fine.)