CSE30 — Final

Yee **Fall '97**

Name / Login: Answer Key

There are a total of 26 questions on 23 pages. There are 98 points possible from the questions. **Writing your login clearly on every page is worth two additional points.** If we have problems putting your exam back together because your written logins at the top of each of the pages are hard to read, *you lose these points*. Yes, like in grade school, you get points for penmanship.

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 at the top of 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 open book, open notes, but *not* open people (no scalpels, please). You may look at your *own* books and 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, 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 into space.

No electronic computation aids are allowed.

1. **[Base Conversion]** Given a string of digits $d_{k-1}d_{k-2}\ldots d_2d_1d_0$ in a certain base b, where $0 \le d_j < b$ for $j = 0, \ldots, k-1$ (usually written as $d_{k-1}d_{k-2}\ldots d_2d_1d_0\,_{(b)}$), what is the corresponding integer? (1) Write this in a mathematical notation. (2) Also write down this number multiplied by b^2 as a string of digits in base b.
(2pts)

Ans:

(1) The number is

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

and (2) b^2 times this number is just $d_{k-1}d_{k-2}\ldots d_2d_1d_0 00_{(b)}$.

---

2. **[Base Conversion]** Perform the following base conversions.

1: $\mathtt{82743}_{(9)} = ?_{(3)}$

2: $\mathtt{BEEF\,DEAD}_{(16)} = ?_{(8)}$

(3pts, 1.5 each)

Ans:

1: $\mathtt{82743}_{(9)} = \mathtt{22\,02\,21\,11\,10}_{(3)}$

2:

$$\begin{aligned}
\mathtt{BEEF\,DEAD}_{(16)} &= \mathtt{1011\,1110\,1110\,1111\,1101\,1110\,1010\,1101}_{(2)} \\
&= \mathtt{10\,111\,110\,111\,011\,111\,101\,111\,010\,101\,101}_{(2)} \\
&= \mathtt{27673757255}_{(8)}
\end{aligned}$$

---

3. [**Micro-architecture**] What are the differences between registers and cache memory? Explain.
(2pts)

Ans:

Registers are very fast memory that may be operated upon directly by instructions. Typically on RISC machines, ALU operations may only be performed on registers — the instruction encoding allows 2 or 3 registers to be named. This necessarily implies that there are very few registers compared to other kinds of memory, since the names of 2 or 3 registers must fit inside of an instruction word (along with the op code and immediate constants).

Cache memory are also very fast, but this memory is not directly addressable. Instead, they are accessed as a side effect of accessing main memory (RAM), and the cache operates transparently to speed up loads (`lw`) and stores (`sw`) (the latter only if it is a write-back cache).

4. [**Micro-architecture**] Suppose we are building a computer system around a processor "core" that is capable of running at 500 MHz. The processor cycle time is $\frac{1}{500\text{MHz}}$ or 2nS (nanoseconds). Suppose we can build a data cache that takes less than 2 nS to respond to a read request, so a cache hit would not slow down the processor at all, and a main memory that takes 100 nS or 50 cycles to similarly respond, forcing the processor to stall for those 50 cycles. We are considering a range of cache sizes in our design. (1) Suppose we chose a cache size that would give a cache hit rate of 99%; what is the average memory access time? (2) Suppose we save some money and reduce the size of the cache so that the hit rate would now be just 90%. What is the average memory access time now? (3) Suppose we save yet more money, and trim the cache so that the cache hit rate would only be 80%. What is the new average memory access time? (These numbers provide a rough estimate of the performance of the designs.)
(3pts)

Ans:

If p is the probability of a cache hit, then the expected memory access time is $p \cdot t_{\text{hit}} + (1-p) \cdot t_{\text{miss}}$. (1) At a hit rate of 99%, the expected time is $0.99 \times 2 + 0.01 \times 100 = 1.98 + 1 = 2.98$ nS. (2) At a hit rate of 90%, the expected access time is $0.9 \times 2 + 0.1 \times 100 = 1.8 + 10 = 11.8$ nS. (3) At a hit rate of 80%, the expected access time is $0.80 \times 2 + 0.2 \times 100 = 1.6 + 20 = 21.6$ nS.

---

5. [**Number representation**] Suppose you had a computer with 48-bit words (e.g., our One Instruction Computer). When using the two's complement representation, what is the largest number that you can represent in a word? What is the smallest? Write these as simple mathematical expressions.
(2pts)

Ans:

The largest is $2^{47} - 1$. The smallest is -2^{47}.

---

6. [**One Instruction Computer**] Convert the following `subz` program fragment to its hexadecimal representation for our OIC simulator.

```
   subz t,t,next
   subz c,c,next
   subz t,a,next
   subz t,b,next
   subz c,t,next
   subz t,t,this
a: 0xc0de4beef
b: 0x314159265358
c: 0
t: 0
```

Assume that the first instruction will be in memory location 0. You do not need to add comments to this code, but you should be explicit about how you assigned addresses and how you arrived at your result. (Give the address of each instruction.)
(2pts)

Ans:

```
     9       9       1   ; 0       subz t,t,next
     8       8       2   ; 1       subz c,c,next
     9       6       3   ; 2       subz t,a,next
     9       7       4   ; 3       subz t,b,next
     8       9       5   ; 4       subz c,t,next
     9       9       5   ; 5       subz t,t,this
   0xc  0x0de4  0xbeef   ; 6   a:  0xc0de4beef
0x3141  0x5926  0x5358   ; 7   b:  0x314159265358
     0       0       0   ; 8   c:  0
     0       0       0   ; 9   t:  0
```

7. [**Operating System**] Why do most modern operating systems forbid self-modifying code?
(2pts)

Ans:

Disallowing self-modifying code permits the sharing of the code segment when several different processes are running from the same original program image. Furthermore, self-modifying code interact badly with Harvard architecture machines, i.e., those with separate instruction and data caches: I-caches are normally read-only, and so can be implemented using simpler circuitry and can be denser than the D-cache. In this case, the I-cache's copy of memory would not be synchronized with the D-cache, where the dynamically modified copy would reside.

8. [**Architecture / Operating Systems**] What is *locality of reference*? Explain what it means relative to caches and virtual memory pages.
(2pts)

Ans:

Locality of reference refers to the fact that programs tend to use certain portions of memory more often than others, typically depending on which part of the program is running at the moment. This applies both to instruction memory as well as data memory. Thus, caches exploit this phenomenon by automatically keeping frequently-accessed memory contents in faster cache memory in a transparent fashion. Virtual memory also exploit this phenomenon at a coarser granularity and lower absolute speeds; instead of cache memory versus physical memory, virtual memory uses physical memory to hold the frequently accessed data, and the less frequently accessed data gets written out to disk.

9. [**Assembly Language**] What are leaf functions? Give a concise definition, and explain how it affects the manner in which the assembly language for it is written.
(3pts)

Ans:

A leaf function is a function that calls no functions – it is a node with no outbound edges in the call graph. Such a property can sometimes simplify code generation: the `$ra` register need not be saved, and often the entire stack frame may be omitted if the caller-saved registers suffice for the leaf function's use.

10. [**Stack Frames**] An explicit frame pointer is sometimes not used – the Larus handout gives both `$fp` and `$s8` as names (aliases) for `$30`. In what case **must** a frame pointer be used?
(4pts)

Ans:

A frame pointer must be used when compiling code that uses the `alloca` memory allocator. `alloca` allocates memory that is automatically freed when the function calling `alloca` returns, and this is accomplished by simply moving the stack pointer to allocate the temporary storage. In this case, local variables must be located relative to the frame pointer and not the stack pointer, since the stack pointer will have moved by a possibly input-dependent amount after the call to `alloca`.

11. [**MIPS Pseudo-instructions**] Into what real MIPS instruction sequence does the pseudo-instruction

```
sw $t0, array_addr($t1)
```

expand into?
(4pts)

Ans:

The array `array_addr` is a 32-bit value, and the MIPS assembler will expand it into the sequence:

```
lui $at, UPPER(array_addr)
addu $at, $at, $t1
sw $t0, LOWER(array_addr)($at)
```

which is a three instruction sequence.

12. **[MIPS instruction set]** There are "`u`" versions of the `lb` and `lh` load instructions, `lbu` and `lhu`. Why isn't there a "`u`" version of the `lw` instruction? Did the MIPS architects forget?
(3pts)

Ans:

The "`u`" instructions are unsigned load instructions. There are unsigned versions for the byte and half-word loads because when loading unsigned non-32-bit data values into a 32-bit register, the upper bits need to be zeroed in order for the newly-loaded register contents to be interpreted as the same value. The non-unsigned load instructions sign-extends the smaller data value, and so negative numbers would still be interpreted as negative.

Since the `lw` instruction completely loads a word, having a signed/unsigned distinction makes no sense.

13. [**Reverse Polish Notation**] The calculator program that we built used reverse polish notation (RPN). Convert the following normal infix arithmetic expression into a keystroke sequence for the calculator, where the numbers are entered in the same order as in the infix expression.

```
5 * (2 + 3) - 2
```

(2pts)

Ans:

In RPN, the expression would be

5 *enter*
2 *enter*
3 *enter*
+ *enter*
* *enter*
2 *enter*
- *enter*

14. [**Translating C to assembly**] Translate the following C code into an equivalent series of MIPS assembly language instructions. You may assume that the C variables are in the correspondingly named registers. Indicate where the code that precedes the loop, the code that comprises the body of the loop, and the code that follows the loop would be located in your equivalent MIPS code. Efficiency matters.

```
code that precedes loop
for (t0 = 0; (t0 > t1) && (t2 != t3); t0 += 4 ) {
        loop body ... may change any variable
}
code that follows loop
```

(5pts)

Ans:

```
        code that precedes loop
        li $t0,0
        b test
loop:   loop body ... may change any variable
        addiu $t0,$t0,4
test:   ble $t0,$t1,done
        bne $t2,$t3,loop
done:   code that follows loop
```

15. **[Recursion]** Translate the following C function into MIPS assembly. As usual, obey all the standard register usage conventions.

```
int ack(int i,int j)
{
     if (i == 1) return 1 << j;
     if (j == 1) return ack(i-1,2)
     return ack(i-1,ack(i,j-1));
}
```

(5pts)

Ans:

```
        .globl ack
ack:    sub $sp, $sp, 16
        sw $ra, 4($sp)
        sw $a0, 8($sp)
        sw $a1, 12($sp)
        bne $a0, 1, L1
        li $v0, 1
        asl $v0, $v0, $a1
        b L2
L1:     bne $a1, 1, L3
        sub $a0, $a0, 1
        li $a1, 2
        jal ack
        b L2
L3:     sub $a1, $a1, 1
        jal ack
        move $a1, $v0
        sub $a0, $a0, 1
        jal ack
L2:     lw $ra, 4($sp)
        lw $a0, 8($sp)
        lw $a1, 12($sp)
        add $sp, $sp, 16
        jr $ra
```

16. [**Number representation / MIPS assembly**] Suppose you had an unsigned number N in the register `$t0`. Give the optimal sequence of instructions to compute (1) N mod 64, placing the result back in `$t0`. Do the same for (2) $N \div 128$ (integer division, truncating towards zero), (3) $9 \times N$, and (4) N^{17}.

By "the optimal sequence" I mean the fastest execution time – assume that the `mult` instruction requires 12 cycles to complete, and that the `div` instruction requires 35 cycles to complete. **Do not** use any pseudo-instructions (the dagger "†" instructions in Larus). You may use the other `$t` registers for any scratch space that you need.
(8pts, 2 each)

Ans:

(1) Since 64 is an exact power of 2, this is most efficiently computed using bit masks:

```
andi $t0, $t0, 0x3f
```

This requires only a single instruction.

(2) 128 is also an exact power of 2, so we use:

```
srl $t0, $t0, 7
```

This requires only one instruction.

(3) We express 9 as $8 + 1$, and use the sequence:

```
sll $t1, $t0, 3
addu $t0, $t0, $t1
```

using two instructions and one scratch register, avoiding the `mult` instruction.

(4) We express 17 as $16 + 1$, so $N^{17} = N^{16} \times N = (((N^2)^2)^2)^2 \times N$:

```
mult $t0,$t0
mflo $t1
mult $t1,$t1
mflo $t1
mult $t1,$t1
mflo $t1
mult $t1,$t1
mflo $t1
mult $t1,$t0
mflo $t0
```

using 10 instructions of which 5 are multiplies.

17. [**Logic**] Given the C boolean expression:

```
!(((A || !B) && C) || !(D && E))
```

apply DeMorgan's Theorem and distribute the logical negation all the way through to the variables `A`, `B`, `C`, `D`, and `E`. Fully parenthesize the expression.
(3pts)

Ans:

```
(((!A && B) || !C) && (D && E))
```

18. [**Logic and assembly**] Translate the following C code fragment to an equivalent sequence of MIPS instructions. Assume that the C variables are in the correspondingly named registers.

```
code which precedes
if ((t0 && t1) || (t2 && t3)) {
      true arm
} else {
      false arm
}
code which follows
```

Clearly show where the instructions for the *italicized code* would go. Efficiency matters — no unconditional branches are needed.
(5pts)

Ans:

```
      code which precedes
      beqz $t0, L1
      bnez $t1, L2
L1:   beqz $t2, L3
      beqz $t3, L3
L2:   true arm
      b L4
L3:   false arm
L4:   code which follows
```

19. [**Translating C to assembly**] Translate the following C code into an equivalent sequence of MIPS instructions using a jump table. You may assume that the C variables are in the correspondingly named registers. You may use any of the other `$t` registers for scratch.

code that precedes the switch
```
switch (t0 & 0x7) {
case 0:  t1 = 5; break;
case 1:  t1 = 8; break;
case 2:  t1 = 2; break;
case 3:  t1 = 3; break;
case 4:  t1 = 0; break;
case 5:  t1 = 12; break;
case 6:  t1 = 1; break;
case 7:  t1 = 17; break;
}
```
code that follows the switch

(5pts)

Ans:

```
        .text
        code that precedes the switch
        and $t1,$t0,0x7
        sll $t1, $t1, 2
        lw $t1, jtbl($t1)
        jr $t1
case0:  li $t1, 5
        b end_switch
case1:  li $t1, 8
        b end_switch
case2:  li $t1, 2
        b end_switch
case3:  li $t1, 3
        b end_switch
case4:  li $t1, 0
        b end_switch
case5:  li $t1, 12
        b end_switch
case6:       li $t1, 1
             b end_switch
case7:       li $t1, 17
             b end_switch
end_switch:  code that follows the switch
             .data
jtbl:        .word case0
             .word case1
             .word case2
             .word case3
             .word case4
             .word case5
             .word case6
             .word case7
```

20. [**Translating C to assembly**] Translate the following C code into an equivalent series of MIPS instructions using table lookup. You may assume that the C variables are in the correspondingly named registers. You may use any of the other `$t` registers for scratch.

code that precedes the table lookup
```
{
    static unsigned char fntbl[] = {
        5,8,2,3,0,12,1,17,
    };
    t1 = fntbl[t0];
}
```
code that follows the table lookup

(5pts)

Ans:

```
        .text
        code that precedes the table lookup
        and $t1,$t0,0x7
        lb $t1, fntbl($t1)
        code that follows the table lookup
        .data
fntbl:  .byte 5
        .byte 8
        .byte 2
        .byte 3
        .byte 0
        .byte 12
        .byte 1
        .byte 17
```

21. [**Code Optimization**] Suppose you need to speed up a program. By profiling the program, you've identified the handful of routines that are consuming most of the running time. What are the techniques that we went over in class you might apply?
(4pts)

Ans:

The first and most dramatic speed-up can be achieved by improving the algorithm used. Strength reduction should then be applied to algebraically replace expensive arithmetic operations with equivalent cheaper ones. Next, common subexpression elimination and constant propagation can remove unnecessary repeated computation. Side-effect-free functions with a small range of inputs can be replaced with table lookup. Loop unrolling can also be applied, to amortize loop control overhead over many more instructions.

22. [**Code Optimization**] Speed up the following code:

```
#define N (1024*1024) /* or some other very large number */
...
     extern int array[N];
     int i;

     for (i = 0; i < N; i++) {
          array[i] = i * i * i * i;
     }
```

You may write the faster version in either C or MIPS assembly.
(7pts)

Ans:

```
#define N (1024*1024) /* or some other very large number */
...
     extern int array[N];
     int i, i2, i3, i4;

     for (i = i2 = i3 = i4 = 0; i < N; ) {
          array[i] = i4;
          i4 += (i3 << 2) + (i2 << 2) + (i2 << 1) + (i << 2) + 1;
          i3 += (i2 << 1) + i2 + (i << 1) + i + 1;
          i2 += (i << 1) + 1;
          i += 1;
     }
```

23. [**Code Optimization**] Speed up the following code and translate it into MIPS assembler:

```
/* assume nelt is usually large */
void init_array(int array[], int nelt)
{
      int t0;

      for (t0 = 0; t0 < nelt; t0++) {
           switch (t0 % 4) {
           case 0:  array[t0] = 1; break;
           case 1:  array[t0] = 5; break;
           case 2:  array[t0] = 425; break;
           case 3:  array[t0] = -37; break;
           }
      }
}
```

(7pts)

Ans:

```
init_array:  andi $t1, $a1, 0xfffffffc
             sll $t1, $t1, 2
             addu $t1, $a0, $t1
             li $t2, 1
             li $t3, 5
             li $t4, 425
             li $t5, -37
             move $t6, $a0
             b L1
L2:          sw $t2, 0($t6)
             sw $t3, 4($t6)
             sw $t4, 8($t6)
             sw $t5, 12($t6)
             add $t6, $t6, 16
L1:          blt $t6, $t1, L2
```

```
     and $t1, $a1, 0x3
     beq $t1, 0, L3
     beq $t1, 1, L4
     beq $t1, 2, L5
     sw $t3, 0($t6)
     add $t6, $t6, 4
L5:  sw $t4, 0($t6)
     add $t6, $t6, 4
L4:  sw $t5, 0($t6)
     add $t6, $t6, 4
L3:  jr $ra
```

24. **[Threads]** What are threads? What are the differences between kernel and user-level threads? Explain the pros and cons.
(5pts)

Ans:

Threads are virtual CPUs. These are complete register sets — and multiple threads in a single process may execute in completely different places in the program (i.e., the PC for the virtual CPUs have unrelated values). In kernel threads, the operating system kernel context switches among them. If one kernel thread blocks, e.g., due to an I/O operation, the kernel will know to switch to another runnable thread; additionally, on a multi-CPU machine, different threads can execute on different CPUs. User threads are lighter weight: fewer registers need to be saved, and context switching does not involve address space changes (no need to context switch to kernel mode due to a timer interrupt and then back to user mode). User level (or coroutine) threads, however, are not preemptive and if one coroutine thread blocks on an I/O operation, all the threads will also stop as a side effect.

25. **[Threads]** Explain what is mutual exclusion, and give a concrete example of why it is important.
(5pts)

Ans:

Mutual exclusion is preventing more than one thread from accessing some data at one time. The example given in class was adding a new element as the head of a linked list:

```
struct elt {
      struct elt *next;
      int num;
} *head;

void add_elt(struct elt *new_elt)
{
      elt->next = head;
      head = elt;
}
```

If two threads try to add to the list at the same time, and the first thread pauses (e.g., is context switched out) after the `elt->next = head;` statement, the second thread could start and complete a call to `add_elt` before the first thread continues; when it does, it will overwrite `head`, making the element added by the second thread unreachable.

To ensure mutual exclusion, locks or semaphores may be used to prevent two threads from being in a critical region simultaneously.

26. [**Extra credit (?)**] Computer programmers often confused Halloween and Christmas, because Oct 31 (October / Octal 31) is equal to Dec 25 (December / Decimal 25). How would you devise a mnemonic so you won't get confused when you become a more experienced programmer?
(0pts)

Ans:

Use this page (front and back) for overflow. Clearly mark which problem's answer is being overflowed on both this page and the page containing the original question.