CSE 30 — Final

Yee **Fall '96**

Name / Login: Answer sheet

DO NOT LOOK AT ANY OTHER PAGE OF THIS FINAL UNTIL THE INSTRUCTOR TELLS YOU TO START.

There are a total of 25 questions on 25 pages. When you are told you may start, you should first make sure that you have all the pages, and write your name and your login at the top of first page, and your login on the top of subsequent pages.

This test is open book, open notes. You may look at your *own* books and notes all you want. You may **not** look at anybody else's books or notes.

1. Define the `subz` instruction for the One-Instruction Computer. (2pts)

Ans:
The `subz` instruction takes three addresses as arguments:

```
subz A,B,C
```

It is equivalent to the following pseudo-code:

```
mem[A] = mem[A] - mem[B];
if (mem[A] != 0) {
        PC = PC + 1;
} else {
        PC = C;
}
```

2. Why do programmers confuse Halloween and Christmas? (1pt)

Ans:
Because Octal 31 is equal to Decimal 25.

---

3. What's the difference between general purpose registers and special purpose registers? Name a couple of each type for the MIPS. (2pts)

Ans:
General purpose registers may be used interchangeably by almost all instructions; special purpose registers may be used only with certain instructions and are sometimes used implicitly (e.g., $pc is automatically incremented). General purpose registers are: $a0, $a1, ..., $s0, etc. Special purpose registers include $hi and $lo.

---

4. The `spim` emulator extends the machine-level instruction set by providing pseudo-instructions such as the `la` and the `li` instruction which handles 32-bit constants. To what instruction sequence does `li` expand to when the constant does not fit in 16-bits? To what instruction does `li` expand to when the constant does fit in 16-bits? (2pts)

Ans:

```
li $n, <value>
```

expands to the two instruction sequence when `value` is a 32-bit constant:

```
lui $n,<high-16-bits>
ori $n,$n,<low-16-bits>
```

When `value` fits in 16-bits, it expands to:

```
ori $n,$zero,<16-bit-value>
```

---

5. What is *instruction bandwidth*? In what way do RISC and CISC processor design strategies differ about instruction bandwidth usage / requirements? (3pts)

Ans:
Instruction bandwidth is the rate at which the processor reads bytes of instructions from memory (instruction-cache and/or RAM). RISC designs use simpler-to-decode, longer instructions. And because the typical RISC instruction does "less" than the typical CISC instruction, more RISC instructions are required to do the same work. Thus, a RISC-coded version of a program would require a higher instruction bandwidth than the equivalent CISC-coded program to achieve the same overall run time.

6. What are the contents of registers \$a0, \$a1, \$v0, \$s0, and \$s1 after the following code fragment runs? Assume that the user typed in the key sequence 3 1 4 1 5 Return. What are the contents of memory at address `inbuf`, `inbuf+1`, `inbuf+2`, `inbuf+3`, ... `inbuf+255`? (3pts)

```
                    .data
inbuf:              .space 256
                    .text
                    ...
                    la $a0,inbuf
                    li $a1,256
                    li $v0,8
                    syscall
                    la $s0,inbuf
                    lb $s1,0($s0)
                    # values here
```

Ans:
\$a0 = `inbuf`, \$a1 = `256`, \$v0 = 8, \$s0 = `inbuf`, and \$s1 = ASCII value of the first character that the user typed in, which is '3' or 0x33.

At location `inbuf` will the ASCII value of the character 3, or 0x33. At location `inbuf+1` will be the character 1, or 0x31; at location `inbuf+2` will be 0x34; at location `inbuf+3` will be 0x31; at location `inbox+4` will be 0x35; at location `inbuf+5` will be a newline character, or 0xa; at location `inbuf+6` will be a null character to terminate the string, 0x0. The contents of `inbuf+7` through `inbuf+255` are unchanged from their previous values. (Depends on the elided code that preceded the syscall.)

7. If a C function (on a MIPS machine) calls another function that takes three arguments (`arg1`, `arg2`, and `arg3`), how will those arguments be passed? (3pts)

Ans:
The arguments will be passed in registers: `arg1` in `$a0`, `arg2` in `$a1`, and `arg3` in `$a2`. For non-word-sized arguments, see table D-13 in Kane & Heinrich.

C programs follow the standard register usage convention, and any assembly language code that interfaces with C code must also.

8. What does the `jal` instruction do? What registers are affected? What is this instruction used for? (3pts)

Ans:
The `jal` instruction stores the address of the following instruction (actually instruction following the branch delay slot following the `jal`) in the `$ra` register, and then transfers control to the address label that is the argument to the `jal` instruction (i.e., the `$pc` is loaded with that address). The saving of the next instruction address is the "linking" part of the jump-and-link instruction; the instruction is used for calling a subroutine / function, which may then return to the caller by using the `jr $ra` instruction to return control to the caller.

9. What does word alignment mean? Why do RISC processors generally require that word loads and stores be word aligned? (3pts)

Ans:
Word alignment refers to placing data at addresses that end with 0, 4, 8, or c (in hex). Having data word aligned means that load and stores will use at most one bus transaction, since the data bus is a multiple of a word in size. If a load from memory permitted unaligned accesses, potentially two bus transactions would be needed, getting only part of the datum during each transaction.

RISC processors' alignment requirement forces the code to be more efficient, and simplifies the hardware design since detection of unaligned references can simply generate an exception — which is simple to do — instead of generating extra bus cycles and shifting data around to get at the data.

10. Do the following base conversions. You do *not* need to show intermediate results. (3pts)

(A) $\texttt{DEADBEEF}_{(16)} = ?_{(2)}$
(B) $53827_{(9)} = ?_{(3)}$
(C) $33653337357_{(8)} = ?_{(16)}$

Ans:

(A) $\texttt{DEADBEEF}_{(16)} = 1101\,1110\,1010\,1101\,1011\,1110\,1110\,1111_{(2)}$

(B) $53827_{(9)} = 12\,10\,22\,02\,21_{(3)}$

(C)

$$\begin{aligned} 33653337357_{(8)} &= 011\,011\,110\,101\,011\,011\,011\,111\,011\,101\,111_{(2)} \\ &= 1101\,1110\,1010\,1101\,1011\,1110\,1110\,1111_{(2)} \\ &= \texttt{DEADBEEF}_{(16)} \end{aligned}$$

11. What are stack frames? What are they used for? What are the function prologue and epilogue instructions? What do they do? (3pts)

Ans:
A stack frame is a data structure on the stack, which contains space to hold variables local to the function that constructed it and/or the original values of callee-saved registers (when that function needs to use those registers for its own purposes).

The function prologue code runs before the "real" body of the function executes; it sets up the stack frame. The epilogue code is the converse: it runs after the "real" body of the function code is done but prior to returning to the caller, and it tears down the stack frame, restoring the original value of various registers and deallocating the stack frame.

12. What is the stack memory and where are the places where it might be stored? (5pts)

Ans:
Stack memory contains a program's stack. It is part of a program's virtual address space, so it is typically stored in main memory. However, parts of it could be contained in the data cache, and parts of it could be paged out to disk.

13. What is *locality of reference*? Explain what it means relative to caches and virtual memory pages. (5pts)

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.

14. Give the equivalent MIPS code to the following C function:

```
int strncpy(char *dst, const char *src, unsigned int maxchars)
{
        /* note side-effect test/assignment */
        while (maxchars != 0 && *dst++ = *src++)
                --maxchars;
        /*
         * was ``*src'' instead of ``*src++'',
         * but typo corrected during final
         */
}
```

Note that this is the standard C library `strncpy` – if no `NUL` character is found after `maxchars` to terminate the source string, the destination string is not `NUL` terminated either. *You do not need to write a* `main` *function.* (5pts)

Ans:

```
                  .text
strncpy:          j test
loop:             subiu $a3,$a3,1
test:             beq $a3,$zero,done
                  lw $t0,0($a0)
                  addiu $a0,$a0,1
                  sw $t0,0($a1)
                  addiu $a1,$a1,1
                  bne $t0,$zero,loop
done:             jr $ra
```

15. Write a MIPS assembly language function `eval_deriv` to evaluate the derivative of a given polynomial $p(x) = \sum_{i=0}^{d} a_i x^i$ at a given point z. The derivative of $p(x)$ is given by $p'(x) = \sum_{i=1}^{d} i a_i x^{i-1}$.
The code should take three parameters: `$a0` is the degree of the polynomial (d), `$a1` is the address of an array of words containing the coefficients of the polynomial, highest degree coefficent at the first word, next highest degree coefficient at the next word, etc (address \$a1 contains a_d, address \$a1+4 contains a_{d-1}, ..., address \$a1+4d+4 contains a_0), and `$a2` containing the value of z. The return value of `eval_deriv` should be $p'(z)$. *You do not need to write a* `main` *function.* (5pts)

Ans:
We can write

$$p'(x) = \sum_{i=1}^{d} i a_i x^{i-1}$$

as

$$p'(x) = \{[\{[(da_d)\, x + (d-1)\, a_{d-1}]\, x\} + ...2a_2]\, x + a_1\}$$

and eliminate the exponentiation of z:

```
                    .text
eval_deriv:         li $v0,0
                    beq $a0,$zero,edone
eloop:              lw $v1,0($a1)
                    mult $v1,$a0
                    mflo $v1
                    add $v0,$v0,$v1
                    sub $a0,$a0,1
                    beq $a0,$zero,edone
                    mult $v0,$a2
                    add $a1,$a1,4
                    mflo $v0
                    j eloop
edone:              jr $ra
```

16. Write a MIPS assembly program to implement the following C program. Your program must follow the structure of the C code and use recursion. Also, give the standard name for this mathematical function. (6pts)

```
int fn(unsigned int i)
{
        if (i <= 1) return 1;
        return i * fn(i-1);
}
```

Ans:

```
fn:             sub $sp,$sp,12
                sw $fp,4($sp)
                addu $fp,$sp,12
                sw $ra,0($fp)

                bgtu $a0,1,nope
                li $v0,1
                j done

nope:           sw $a0,-4($fp)
                subu $a0,$a0,1
                jal fn
                lw $t0,-4($fp)
                multu $v0,$v0,$t0

done:           lw $ra,0($fp)
                lw $fp,4($sp)
                addu $sp,$sp,12
                jr $ra
```

This is the factorial function.

17. Give the structured assembly language equivalents of the following C language control flow constructs (6pts, 2 each):

1.
```
do {
        ...
} while (expr);
```
2.
```
while (expr) {
        ...
}
```
3.
```
for (expr1; expr2; expr3) {
        ...
}
```

Be sure to indicate where and how the various expressions and elided loop body code is evaluated. You may assume that the result of the expressions are in the `$s` registers (use `$s0` for `expr`, and `$s1`, `$s2`, and `$s3` for `expr1`, `expr2`, and `expr3` as needed).

Ans:
For the `do ... while (expr);` loop, the equivalent assembly code is

```
top:        ... # loop body
            # expr evaluation
            bnez $s0, top
```

For the `while` loop, the equivalent assembly code is

```
            j test
top:        ... # loop body
test:       # expr evaluation
            bnez $s0,top
```

For the `for` loop, the equivalent assembly code is

```
            # expr1 evaluation
            j test
top:        ... # loop body
            # expr3 evaluation
test:       # expr2 evaluation
            bnez $s2,top
```

18. Given the C boolean expression:

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

Apply DeMorgan's Theorem and distribute the logical negation through to the variables `A`, `B`, `C`, and `D`. (4pts)

Ans:

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

19. Vector addition $\vec{x} + \vec{y}$ is defined as the element-wise sum of the vectors, i.e., for k dimensional vectors $\vec{x}$ and $\vec{y}$, if $\vec{x} = (x_1, x_2, \ldots, x_k)$ and $\vec{y} = (y_1, y_2, \ldots, y_k)$, then $\vec{x} + \vec{y} = (x_1 + y_1, x_2 + y_2, \ldots, x_k + y_k)$. (The vectors must be of the same length.) C code to compute the sum of two vectors is

```
void vec_add(int *result, int *x, int *y, int dim)
{
        int i;

        for (i = 0; i < dim; i++) {
                *result++ = *x++ + *y++;
        }
}
```

Give an efficient, loop unrolled version of this code in either MIPS assembly or C. (6pts)

Ans:
In C, the code is:

```
void vec_add(int *result, int *x, int *y, int dim)
{
        register int i;

        switch (dim & 0x3) {
        case 3:  *result++ = *x++ + *y++;
        case 2:  *result++ = *x++ + *y++;
        case 1:  *result++ = *x++ + *y++;
        }
        for (dim >>= 2; dim > 0; --dim) {
                result[0] = x[0] + y[0];
                result[1] = x[1] + y[1];
                result[2] = x[2] + y[2];
                result[3] = x[3] + y[3];
                result += 4; x += 4; y += 4;
        }
}
```

20. Apply strength reduction to give an equivalent C or MIPS assembly function to the following C code:

```
void print_table(int n)
{
        int i;

        for (i = 0; i < n; i++) {
                printf("%d %d %d %d\n",i,i*i,i*i*i,i*i*i*i);
        }
}
```

Your equivalent code should not use the multiplication operator. Hint: Pascal's Triangle is:

```
1 1 1 1 1
1 2 3 4
1 3 6
1 4
1
```

(6pts)

Ans:

```
void print_table(int n)
{
        int i, i2, i3, i4;

        for (i = i2 = i3 = i4 = 0; i < n; i++) {
                printf("%d %d %d %d\n",i,i2,i3,i4);
                i4 += (i3<<2) + (i2<<2)+(i2<<1) + (i<<2) + 1;
                i3 += (i2<<1)+i2 + (i<<1)+i + 1;
                i2 += (i<<1) + 1;
        }
}
```

21. In the following code:

```
int vec_norm1(int len, int *vec) {
        int sum = 0, i;

        for (i = 0; i < len; i++) {
                if (vec[i] >= 0)
                        sum += vec[i];
                else
                        sum -= vec[i];
        }
        return sum;
}
```

the loop invariant – at the test – is $\texttt{sum} = \sum_{k=0}^{\texttt{i}-1} |\texttt{vec}[k]|$. Prove that this is the invariant, and use it to prove that this code correctly computes the sum of the absolute value of the vector elements. (In mathematics, this is called the L_1 norm.) (7pts)

Ans:
We note that the invariant holds at the beginning when $\texttt{i} = \texttt{sum} = 0$, and since $\texttt{i} - 1 < 0$, the sum is zero also. This is the base case. Assuming that the invariant holds at $\texttt{i} = i_0$, we want to show that it holds after one pass through the body of the loop. Before entering the body, we have

$$\texttt{i} = i_0$$

and

$$\texttt{sum} = \sum_{k=0}^{i_0-1} |\texttt{vec}[k]|$$

We modify sum so it is incremented, and $\texttt{sum}' = \texttt{sum} + |\texttt{vec}[i_0]|$, so $\texttt{sum}' - |\texttt{vec}[i_0]| = \texttt{sum} = \sum_{k=0}^{i_0-1} |\texttt{vec}[k]|$, and

$$\begin{aligned} \texttt{sum}' &= |\texttt{vec}[i_0]| + \sum_{k=0}^{i_0-1} |\texttt{vec}[k]| \\ &= \sum_{k=0}^{i_0} |\texttt{vec}[k]| \end{aligned}$$

Next, we have $\texttt{i}' = i_0 + 1$, so $\texttt{sum}' = \sum_{k=0}^{\texttt{i}'-1} |\texttt{vec}[k]|$, and the invariant holds to the next iteration of the loop (inductive case).

The loop terminates when $\texttt{i} = \texttt{len}$, so

$$\begin{aligned} \texttt{sum} &= \sum_{k=0}^{\texttt{i}-1} |\texttt{vec}[k]| \\ &= \sum_{k=0}^{\texttt{len}-1} |\texttt{vec}[k]| \end{aligned}$$

which is what we wanted to show. $\square$

---

22. What is multithreading? How does it affect performance? (7pts)

Ans:
Multithreading provides several threads of control, where each thread is its own, independet register set. The contents of the registers within each set change according to instructions fetched by the `$pc` in that set, and main memory, which is shared among all the threads of a process, are also modified accordingly. When kernel threads are used, operations that block one thread will cause the kernel to context switch to another runnable thread, thus not wasting CPU cycles. Furthermore, if the code is ran on a multiprocessor, the kernel will automatically use the extra processor(s) to run available runnable threads. These effects improve the performance of the program. There are, however, aspects of using threads that degrade performance: to ensure correctness, mutual exclusion via locks, semaphores, monitors, etc are needed to control access to shared data structures, and the mutual exclusion mechanism adds overhead. The overall effect on performance depends on the available parallelism in the problem, the number of real processors available, and the mutual exclusion overhead.

23. What is virtual memory? How is it different from physical memory? How are the two related? What does virtual memory do for the program / programmer? (7pts)

Ans:

Virtual memory is the mechanism by which the amount of physical memory (aka main memory, or RAM-resident memory) is abstracted away. Pages of "physical" memory, typically 4 kilobytes in size, are "mapped" into the virtual address space as needed; when a process accesses memory, it is using virtual addresses which are translated into physical addresses by special address translation hardware. The system can provide more virtual memory than there is physical memory: pages of physical memory are *paged out*, or copied to a hard disk, when the operating system needs to use the physical memory for something else, e.g., data for another virtual page, and the OS can change the virtual-to-physical translation so that that same page of physical memory will appear elsewhere in the processes's (or another process's) address space.

When the original data is needed by the process again, a *page-fault exception* occurs, where the operating system gains control and finds a free physical page into which the data from disk is read (*page-in*), and that physical page made to appear in the right place in the faulting process's address space before that process is allowed to continue.

This means that programmers do not have to know precisely how much memory is available on a given machine: their programs can run on machines with different memory configurations without change. The only effect would be the actual runtime, which depends on whether the resident set of the program fits into the amount of memory physically available and on the actual amount of locality of reference.

24. The following code counts the number of 1s in the binary number in register `$v0` (some machines have a built-in `popcount` instruction that is equivalent). It counts up to 32 (maximum number of 1 bits in a 32-bit word) in only 28 instructions (the first 4 `li` counts as two instructions each). Why/how does it work? (7pts)

```
li $t0, 0x55555555
srl $v1,$v0,1
and $v0, $v0, $t0
and $v1, $v1, $t0
addu $v0, $v0, $v1

li $t0, 0x33333333
srl $v1,$v0,2
and $v0, $v0, $t0
and $v1, $v1, $t0
addu $v0, $v0, $v1

li $t0, 0x0f0f0f0f
srl $v1,$v0,4
and $v0, $v0, $t0
and $v1, $v1, $t0
addu $v0, $v0, $v1

li $t0, 0x00ff00ff
srl $v1,$v0,8
and $v0, $v0, $t0
and $v1, $v1, $t0
addu $v0, $v0, $v1

li $t0, 0xffff
srl $v1,$v0,16
and $v0, $v0, $t0
addu $v0, $v0, $v1
```

Ans:
The code performs several additions in parallel in the `addu` instructions. The first `addu` adds 16 pairs of one-bit numbers in 16 2-bit-wide registers, and the second `addu` adds 8 pairs of 2-bit numbers in 8 4-bit-wide registers, etc.

25. Give a faster `popcount` sequence. (7pts) (Hint: recall the sample solutions for assignment 2 on printing the ASCII code of input characters.)

Ans:
The following table-lookup version is faster:

```
andi $t1,$t0,0xff
lb $v0,pop($t1)
srl $t1,$t0,8
andi $t1,$t1,0xff
lb $t2,pop($t1)
addu $v0,$v0,$t2
srl $t1,$t0,16
andi $t1,$t1,0xff
lb $t2,pop($t1)
addu $v0,$v0,$t2
srl $t1,$t0,24
andi $t1,$t1,0xff
lb $t2,pop($t1)
addu $v0,$v0,$t2
```

The table `pop` is a 256-entry array of bytes containing the number of one bits in the index.