CSE 30 — Midterm

Yee **Fall '96**

Name / Login: Answer sheet

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

There are a total of 21 questions on 22 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. How do you add two numbers using the One Instruction Computer? Suppose we had the following declaration, where `A` and `B` are the input values, and `sum` should hold the result. (You may add more declarations as needed.)

```
                .data
A:              .word x         # caller initializes
B:              .word y         # caller initializes
sum:            .space 1        # return value goes here
```

Ans:

```
scr:            .space 1
                .text
                subz scr,scr,next
                subz scr,A,next
                subz scr,B,next
                subz sum,sum,next
                subz sum,scr,next
```

2. How many general purpose registers are there in the MIPS processor?

Ans:
32

---

3. What's the difference between general purpose registers and special purpose registers? Name some of each for the MIPS.

Ans:
General purpose registers may be used interchangeably by almost all instructions; special purpose registers may be used only with certain instructions. General purpose registers are: $a0, $a1, ..., $s0, etc. Special purpose registers include $hi and $lo.

---

4. What's the maximum displacement for a conditional branch instruction for the MIPS?

Ans:
The displacement field is 16 bits, which is treated as a signed integer. Because instructions are word aligned, the displacement is shifted left 2 bits, then signed extended before being added to the current PC value (address of instruction after the branch). Thus, the target address must be in the range $\texttt{pc} - 2^{17}, \ldots, \texttt{pc} + 2^{17} - 1$.

5. The `spim` emulator extends the machine-level instruction set by providing pseudo-instructions such as `la` and an `li` instruction which handles 32-bit constants. To what instruction sequence does `li` expand to?

Ans:

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

expands to the two instruction sequence:

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

6. What is the difference between an instruction cache and a data cache? Why do processor designers use separate caches for instruction and data rather than a unified cache?

Ans:
The instruction cache is read-only; the processor will never write to it. The data cache is read-write.

Separating instruction from data allows the designer to make use of the read-only property of the I-cache to make its circuitry denser. This means that the total cache size can be larger for a given amount of silicon.

7. 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 1 2 Return. What are the contents of memory at address `inbuf`, `inbuf+1`, `inbuf+2`, `inbuf+3`, `inbuf+4`, and `inbuf+5`?

```
                  .data
inbuf:            .space 256
                  .text
                  ...
                  la $a0,inbuf
                  li $a1,256
                  li $v0,8
                  syscall
                  la $s0,inbuf
loop:             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.

At location `inbuf` will the ASCII value of the character 1, or 0x31. At location `inbuf+1` will be 0x32; at location `inbuf+2` will be the newline character, 0xa; and at location `inbuf+3` will be a null character to terminate the string, 0x0. The contents of `inbuf+4` and `inbuf+5` are unchanged from their previous values.

8. Describe the conventions regarding the usage of the a, s, t, and v registers.

Ans:
The a registers are used to pass arguments to a function; the s registers are the callee-saved temporary registers; the t registers are the temporary registers which are not saved across function calls; the v registers are used for function return values.

9. How is the $ra register used?

Ans:
The $ra register is used to hold the return address to the caller. The jal instruction saves the return address in this register before passing control to the called function; called functions use jr $ra to return control to the caller.

10. What does word alignment mean? Why do RISC processors generally require that word loads and stores be word aligned?

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.

11. Do the following base conversions. You do *not* need to show intermediate results.

(A) $\texttt{3A8C}_{(16)} = ?_{(2)}$
(B) $\texttt{35613}_{(9)} = ?_{(3)}$
(C) $\texttt{10110110110}_{(2)} = ?_{(8)}$

Ans:

(A) $\texttt{3A8C}_{(16)} = \texttt{0011 1010 1000 1100}_{(2)}$

(B) $\texttt{35613}_{(9)} = \texttt{10 12 20 01 10}_{(3)}$

(C) $\texttt{10 110 101 010}_{(2)} = \texttt{2666}_{(8)}$

12. What is the difference between registers and main memory?

Ans:
Registers are very fast memory in the processor; main memory is slower and is on the system bus. There are only a few registers (32 general purpose registers on the MIPS), but there could be lots of memory (many millions of words). Registers are named by fields in the instructions directly, and instructions are designed to be able to manipulate registers very quickly; words in main memory are accessed via their memory address, and such access is limited to load and store instructions which copies from memory to registers and vice versa.

13. Is the instruction sequence on the left faster or is the instruction sequence on the right faster?

`li $t0,42`	or	`sll $v0,$a0,5`
`mult $t0,$a0`		`sll $t0,$a0,3`
`mflo $v0`		`add $v0,$v0,$t0`
		`sll $t0,$a0,1`
		`add $v0,$v0,$t0`

Ans:
The second, shift-and-add sequence is faster, since it completes in 5 clock cycles; the first version uses the `mult` instruction, which requires 10-18 cycles to complete by itself.

14. What are stack frames? What are they used for? What are the function prologue and epilogue instructions? What do they do?

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 stack frame.

15. Give the equivalent MIPS code to the following:

```
int strlen(char *cp)
{
        int count = 0;

        while (*cp++)
                ++count;
        return count;
}
```

Ans:

```
                .text
strlen:         move $v0,$zero    # count = 0
L0:             lb $t0,0($a0)     # *cp++
                add $a0,$a0,1
                beqz $t0,L1
                add $v0,$v0,1
                j L0
L1:             jr $ra
```

16. Write a MIPS assembly language function `eval` to evaluate a polynomial $p(x) = \sum_{i=0}^{d} a_i x^i$ at a given point z. 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` should be $p(z)$.

Ans:

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

---

17. What is pipelining? How does it affect performance?

Ans:
Pipelining separates the execution of an instruction into several smaller subtasks — IF, RD, ALU, MEM, WB — that may be executed sequentially, so that functional units that perform these smaller tasks may operate in parallel, so that the next instruction may be started after the current instruction moves from the first subtask to the second.

Pipelining does nothing for the latency of results, but provides increased throughput. In the MIPS R2000/3000 pipeline, each instruction requires at least 5 cycles to complete (make its way through the pipeline); by overlapping execution of pipelined instructions, instead of completing one instruction every 5 cycles, the processor completes one instruction per cycle.

18. What does the following function do? Describe what it does and give the equivalent C code.

```
                .text
fn:             sll $t0,$a2,2
                add $t0,$a1,$t0
again:          beqz $a2,done
                sub $t0,$t0,4
                lw $t1,0($t0)
                sw $t1,0($a0)
                add $a0,$a0,4
                sub $a2,$a2,1
                j again
done:           jr $ra
```

Ans:
This function does an order reversing memory copy of words. The first argument is the address of the source buffer, the second argument is the address of the destination, and the third argument is the number of words to copy. The order of the words in the destination area will in reverse from the original order. The equivalent C code is:

```
void fn(int *src, noalias int *dst, noalias int count)
{
        dst += dst + count;
        while (count) {
                *--dst = *src++;
                --count;
        }
}
```

19. Write a MIPS assembly program to implement the following C program. Your program must follow the structure of the C code and use recursion.

```
int fn(int i)
{
        if (i <= 0) return 0;
        return i + fn(i-2);
}
```

Ans:

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

                    blez $a0,simple
                    sw $a0,-4($fp)
                    sub $a0,$a0,2
                    jal fn
                    lw $t0,-4($fp)
                    add $v0,$v0,$t0
                    j done

simple:             li $v0,0

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

Recall the `putnum` function:

```
/* precondition:  base > 1, val >= 0 */
void    putnum(int     val,
               int     base)
{
        int     div,rem;

        div = val / base;
        rem = val % base;
        if (div != 0) putnum(div,base);
        if (rem < 10) {
                putchar('0' + rem);
        } else {
                putchar('A' + rem - 10);
        }
}
```

The `putnum` function outputs `val` as a string of "digits" in base `base`, with A–Z representing the values 10–36. The function does not output a trailing newline character.

20. Prove that the `putnum` function works correctly when given input that satisfies the preconditions.

Ans:

The proof is inductive. We assume that the preconditions hold, that is, $\texttt{val} \geq 0$, and that $\texttt{base} \geq 2$.

First, we prove the base case, showing that `putnum` outputs correct when $\texttt{val} < \texttt{base}$. When $\texttt{val} < \texttt{base}$, $\texttt{div} = 0$ and $\texttt{rem} = \texttt{val}$ holds. Thus, no recursive call is made, and the output is a single digit as decided by the `if/else` construct.

Next, we prove the induction step. Assume that `putnum` outputs correctly when $\texttt{val} < \texttt{base}^k$; we need to show that `putnum` works correctly for $\texttt{val} : \texttt{val} < \texttt{base}^{k+1}$. Without loss of generality we may assume that $\texttt{base}^k \leq \texttt{val} < \texttt{base}^{k+1}$, since if $\texttt{val} < \texttt{base}^k$ the inductive assumption tells us that `putnum` will generate the correct output. Since $\texttt{base}^k \leq \texttt{val} < \texttt{base}^{k+1}$, `div` must satisfy $\texttt{base}^{k-1} \leq \texttt{div} < \texttt{base}^k$, and `rem` must satisfy $0 \leq \texttt{rem} < \texttt{base}$. Since $\texttt{div} > 0$, we make a recursive call, which by our induction hypothesis will successfully output `div` as a base `base` digit string. Next, `rem` is *appended* to the output as another base `base` digit.

This output is correct: the induction hypothesis tells us that the output of the recursive call is a base `base` digit string for the parameter, so that digit string's value must be `div`. Appending a digit in base `base` to that output effectively multiplies that output by `base` and adds the value of the new digit, so the value of the output string is $\texttt{base} \times \texttt{div} + \texttt{rem}$, which is our original `val`. Thus we've proven the induction for the next step, and since both the induction step and the base case works, this function must output the correct digit string.

21. (Extra credit, 10pts) Show how to determine if an unsigned number is greater than another one using the One Instruction Computer.

Ans:
The following code compares `A` with `B` and transfers control to the label `greater` if `A` > `B`, and to the label `not_greater` otherwise.

```
                  .data
A:                .word 10
B:                .word 9
Acopy:            .word 0
Bcopy:            .word 0
scr:              .word 0
one:              .word 1
                  .text
gt:               subz Acopy,Acopy,next
                  subz Bcopy,Bcopy,next
                  subz scr,scr,next
                  subz scr,A,next
                  subz Acopy,scr,next
                  subz scr,scr,next
                  subz scr,B,next
                  subz Bcopy,scr,next
                  subz Bcopy,A,not_greater
                  subz Bcopy,Bcopy,next
                  subz Bcopy,scr,next
gtloop:           subz Bcopy,one,greater
                  subz Acopy,one,not_greater
                  subz scr,scr,gtloop
```