CSE30 — Midterm

Yee **Fall '98**

Name / Login: Answer Key

There are a total of 16 questions on 14 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 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 closed book. You are allowed a single sheet 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.

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 (i.e., a summation). (2) Also write down the integer part of n/b as a string of digits.
(4pt)

Ans:

(1) The number is

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

(2) When this number is divided by b, it is just $\lfloor n/b \rfloor = d_{k-1}d_{k-2}\ldots d_2 d_1\,_{(b)}$, i.e., we omit the last digit.

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

2: $47284_{(9)} = ?_{(3)}$

3: $(69)(3)(27)_{(81)} = ?_{(3)}$

4: $2011221112_{(3)} = ?_{(27)}$

5: $2767357255_{(8)} = ?_{(16)}$

(15pt, 3 each)

Ans:

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

2: $47284_{(9)} = 11\,21\,02\,22\,11_{(3)}$

3: $(69)(3)(27)_{(81)} = 2120\,0010\,1000_{(3)}$

4: $2\,011\,221\,112_{(3)} = (2)(4)(25)(14)_{(27)}$

5:

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

3. [**Micro-architecture**] What is the purpose of a cache? Explain what it is, how it achieves its purpose, and what factors influence how well it achieves this.
(5pt)

Ans:

A cache improves overall performance of the computer by transparently making memory accesses faster most of the time. It is fast memory that is located closer to the processor, and contains copies of portions of main memory contents. When the processor attempts to access cached memory, a *cache hit* occurs and the access is fast, since the memory access does not have to travel to the DRAM over the system bus; when the processor attempts to access memory that has not been cached, a *cache miss* occurs and the cache forwards the access to the DRAM, saving (caching) a copy of the result for subsequent use.

If p is the probability of a cache hit, then the expected memory access time is $p \cdot t_{\mathrm{hit}} + (1-p) \cdot t_{\mathrm{miss}}$; typically caches are designed to have a very high p (depends on size and program mix), e.g., 0.99, and $t_{\mathrm{hit}} << t_{\mathrm{miss}}$, so including caches greatly improves overall performance of computers.

Factors that influence how well the cache speeds up programs include the size of the cache, the cache response times, and the program mix. By implementing the cache from faster (and more expensive) memory would improve the t_{hit} value. Vector programs would not benefit much from a data cache.

4. What is the name of your favorite film?
(2pt)

Ans:

Anything is fine. Mine is *Cassablanca*.

5. [**Number representation**] Compute the two's complement of the following numbers stored in 16-bit registers:

1: `0x5141`

2: `0x8576`

Negate the following numbers stored in 16-bit registers:

3: `0xF35`

4: `0xCAFE`

In all 4 cases, mark which results would be interpreted as a negative number when interpreted as a 16-bit two's complement number.
(8pt, 2 each)

Ans:

Taking the two's complement of a number is the same as negating it.

1: `0x5141` → `0xAEBF` (negative)

2: `0x8576` → `0x7A8A`

3: `0xF35` → `0xF0CB` (negative)

4: `0xCAFE` → `0x3502`

---

6. [**Number representation**] Suppose you have a number in a 32-bit register, and its hexidecimal representation is 0x8000 0000. Is this number positive or negative when viewed as a two's complement number? What happens when you negate it? Is the result of the negation positive or negative when viewed as a two's complement number?
(7pt)

Ans:

The number is negative when viewed as a two's complement number, since the high-order bit is set. The result from negating it is also 0x8000 0000. An overflow occurred during the negation, because the result can not be represented as a 32-bit two's complement number (it's too big). The result would be interpreted as the same as the original negative number.

---

7. [**One Instruction Computer**] Define the `subge` instruction.
(3pt)

Ans:

```
subge 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;
```

8. [**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*

```
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 out,C,next
        subge D,zero,done
        subge D,one,main
done:   subge tmp,tmp,done
tmp:    .word 0
one:    .word 1
zero:   .word 0
```

(7pt)

Ans:

```
0x100           ;               .org 0x100
0x000400040101  ; 0x100  main:  subge out,out,next
0x000400000102  ; 0x101         subge out,A,next
0x000400010103  ; 0x102         subge out,B,next
0x000400020104  ; 0x103         subge out,C,next
0x000301090106  ; 0x104         subge D,zero,done
0x000301080100  ; 0x105         subge D,one,main
0x010701070106  ; 0x106  done:  subge tmp,tmp,done
0x000000000000  ; 0x107  tmp:   .word 0
0x000000000001  ; 0x108  one:   .word 1
0x000000000000  ; 0x109  zero:  .word 0
```

9. [**Macro Assembly**] What's the difference between using macros and subroutines? (4pt)

Ans:

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

10. [**Macro Assembly**] Expand the macros in the following macro assembly program. Do *not* convert to machine code. Use extra space on next sheet if needed.

```
zero:   .macro loc
        subge loc, loc, next
        .endmacro
move:   .macro src, dst
        subge dst, dst, next
        subge tmp, tmp, next
        subge tmp, src, next
        subge dst, tmp, next
        .endmacro
call:   .macro entrypt, retpt
        subge retpt, retpt, next
        subge retpt, L0, next
        subge tmp, tmp, entrypt
        .data
L0:     .word neg(triple(tmp,tmp,L1))
        .text
L1:
        .endmacro
add2:   .macro val, var
        subge tmp, tmp, next
        subge tmp, val, next
        subge var, tmp, next
        .endmacro
A:      .equ 0
B:      .equ 1
C:      .equ 2
D:      .equ 3
X:      .equ 0x200
Y:      .equ 0x201
prod:   .equ 0x202
mult:   .equ 0x203
rmult:  .equ 0x223
        .org 0x100
main:   zero sum
        move A, X
        move B, Y
        call mult, rmult
        move prod, sum
        move C, X
        move D, Y
        call mult, rmult
        add2 prod, sum
quit:   subge tmp, tmp, quit
sum:    .word 0
tmp:    .word 0
```

(9pt)

Ans:

```
A:       .equ 0
B:       .equ 1
C:       .equ 2
D:       .equ 3
X:       .equ 0x200
Y:       .equ 0x201
prod:    .equ 0x202
mult:    .equ 0x203
         .org 0x100
main:    subge sum, sum, next
         subge X, X, next
         subge tmp, tmp, next
         subge tmp, A, next
         subge X, tmp, next
         subge Y, Y, next
         subge tmp, tmp, next
         subge tmp, B, next
         subge Y, tmp, next
         subge rmult, rmult, next
         subge rmult, apple, next
         subge tmp, tmp, mult
         .data
apple:   .word neg(triple(tmp,tmp,orange))
         .text
orange:
         subge sum, sum, next
         subge tmp, tmp, next
         subge tmp, prod, next
         subge sum, tmp, next
         subge X, X, next
         subge tmp, tmp, next
         subge tmp, C, next
         subge X, tmp, next
         subge Y, Y, next
         subge tmp, tmp, next
         subge tmp, D, next
         subge Y, tmp, next
         subge rmult, rmult, next
         subge rmult, pear, next
         subge tmp, tmp, mult
         .data
pear:    .word neg(triple(tmp,tmp,peach))
         .text
peach:
         subge tmp, tmp, next
         subge tmp, prod, next
         subge sum, prod, next
quit:    subge tmp, tmp, quit
sum:     .word 0
tmp:     .word 0
```

11. [**One Instruction Computer**] Write an `oic` assembly language program to square a non-negative number. The input number is at location 0, The output should be at location 1. The program should start at location 0x100.
(8pt)

Ans:

```
; negsum = 0;
; for (negx = -N; negx < 0; negx++) negsum -= N;
; out = -negsum;
N:       .equ 0
out:     .equ 1
         .org 0x100
main:    subge negsum, negsum, next
         subge negx, negx, next
         subge negx, N, next          ; negx = -N
         subge tmp, tmp, test
loop:    subge negsum, N, next        ; negsum -= N
         subge negx, neg1, next       ; negx++
test:    subge tmp, negx, done        ; 0 - negx >= 0 or 0 <= negx
         subge tmp, tmp, loop
done:    subge out, out, next
         subge out, negsum, next
endloop  subge tmp, tmp, endloop      ; end of program
negx:    .word 0
negsum:  .word 0
tmp:     .word 0
neg1:    .word -1
```

12. [**One Instruction Computer**] Write an `oic` assembly language program to copy memory from one array (`src`) to another (`dst`). The number of elements to copy is given in memory location `N`. (8pt)

Ans:

```
            subge i, i, next
            subge i, N, next            ; i = -N
            subge i, zero, done         ; i - 0 >= 0 or i >= 0
loop:       subge dst, dst, next        ; move macro
            subge tmp, tmp, next        ; but addrs changes
l2:         subge tmp, src, next        ; as pgm executes
l3:         subge dst, tmp, next
            subge loop, loopMod, next   ; modify above code
            subge l2, l2Mod, next
            subge l3, l3Mod, next
test:       subge i, negone, done       ; i = (i + 1) >= 0
            subge tmp, tmp, loop
done:       subge tmp, tmp, done
i:          .word 0
tmp:        .word 0
loopMod:    .word neg(triple(1,1,0))
l2Mod:      .word neg(triple(0,1,0))
l3Mod:      .word neg(triple(1,0,0))
negone:     .word -1
zero:       .word 0
```

13. [**MIPS**] What are the MIPS `t` and `s` registers used for? In what way are they different from each other? (5pt)

Ans:

Both the `t` and `s` registers are for temporaries. The `t` registers are *caller-saved* registers, since by convention a subroutine is allowed to use them. The `s` registers are *callee-saved* registers; a routine can call a subroutine and expect that these registers' contents will be preserved when the subroutine returns.

14. [**RISC and CISC**] Give an example of a processor with a RISC architecture and an example of processor with a CISC architecture.
(3pt)

Ans:

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.

15. [**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 matters.

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

(5pt)

Ans:

```
        code that precedes loop
        li $t0,0
        la $t2,globalIntArray
        b test
loop:   loop body
        add $t0,$t0,2
        add $t2,$t2,4
test:   ble $t0,$t1,loop
done:   code that follows loop
```

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

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

(9pt)

Ans:

```
fib:        sub $sp, $sp, 16
            sw $fp, 4($sp)
            add $fp, $sp, 16
            sw $ra, -8($fp)
            bgt $a0, 1, rec_fib
            li $v0, 1
            b fib_done
rec_fib:    sw $a0, 0($fp)
            sub $a0, $a0, 1
            jal fib
            sw $v0, -4($fp)
            lw $a0, 0($fp)
            sub $a0, $a0, 2
            jal fib
            lw $a0, -4($fp)
            add $v0, $v0, $a0
fib_done:   lw $ra, -8($fp)
            lw $fp, 4($sp)
            add $sp, $sp, 16
            jr $ra
```