6.004 Computation Structures
Spring 2009

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

6.004 Computation Structures
Spring 2009

Quiz #3: April 10, 2009

Name Solutions	*Athena login name*	*Score* Avg: 20.1

NOTE: Reference material and scratch copies of code appear on the backs of quiz pages.

Problem 1 (5 points): **Quickies and Trickies**

(A) (2 points) A student tries to optimize his Beta assembly program by replacing a line containing

ADDC(R0, 3*4+5, R1)

by

ADDC(R0, 17, R1)

Is the resulting binary program smaller? Does it run faster?

(circle one) Binary program is SMALLER? yes … [✓ no]

(circle one) FASTER? yes … [✓ no]

(B) Which of the following best conveys Church's thesis?

C1: Every integer function can be computed by some Turing machine.
C2: Every computable function can be computed by some Turing machine.
C3: No Turing machine can solve the halting problem.
C4: There exists a single Turing machine that can compute every computable function.

(circle one) Best conveys Church's thesis: C1 … [✓ C2] … C3 … C4

(C) What value will be found in the low 16 bits of the **BEQ** instruction resulting from the following assembly language snippet?

```
        . = 0x100
        BEQ(R31, target, R31)
target: ADDC(R31, 0, R31)
```

16-bit offset portion of above BEQ instruction: 0x0000

(D) Can every **SUBC** instruction be replaced by an equivalent **ADDC** instruction with the constant negated? If so, answer **"YES"**; if not, give an example of a **SUBC** instruction that can't be replaced by an **ADDC**.

SUBC(…) instruction, or "YES": SUBC(Ra, 0x8000, Rc)
(Ra can be any of R0–R31;
Rc can be any of R0–R30.)

Problem 2. (13 points): **Parentheses Galore**

The **`wfps`** procedure determines whether a string of left and right parentheses is well balanced, much as your Turing machine of Lab 4 did. Below is the code for the **`wfps`** ("well-formed paren string") procedure in C, as well as its translation to Beta assembly code. This code is reproduced on the backs of the following two pages for your use and/or annotation.

```
int STR[100];                  // string of parens

int wfps(int i,                // current index in STR
         int n)                // LPARENs to balance
{ int c = STR[i];              // next character
  int new_n;                   // next value of n
  if (c == 0)                  // if end of string,
    return (n == 0);           //   return 1 iff n == 0
  else if (c == 1)             // on LEFT PAREN,
    new_n = n+1;               //    increment n
  else {                       // else must be RPAREN
    if (n == 0) return 0;      // too many RPARENS!
    xxxxx; }                   // MYSTERY CODE!
  return wfps(i+1, new_n);     // and recurse.
}
```

`wfps` expects to find a string of parentheses in the integer array stored at **`STR`**. The string is encoded as a series of **32-bit integers** having values of

1 to indicate a left paren,
2 to indicate a right paren, or
0 to indicate the end of the string.

These integers are stored in consecutive 32-bit locations starting at the address **`STR`**.

`wfps` is called with two arguments:

1. The first, **`i`**, is the index of the start of the part of **`STR`** that this call of **`wfps`** should examine. Note that indexes start at 0 in C. For example, if **`i`** is 0, then **`wfps`** should examine the entire string in **`STR`** (starting at the first character, or **`STR[0]`**). If **`i`** is 4, then **`wfps`** should ignore the first four characters and start examining **`STR`** starting at the fifth character (the character at **`STR[4]`**).
2. The second argument, **`n`**, is zero in the original call; however, it may be nonzero in recursive calls.

`wfps` returns 1 if the part of **`STR`** being examined represents a string of balanced parentheses if **`n`** additional left parentheses are prepended to its left, and returns 0 otherwise.

Note that the compiler may use some simple optimizations to simplify the assembly-language version of the code, while preserving equivalent behavior.

The C code is incomplete; the missing expression is shown as **`xxxx`**.

```
STR:   . = .+4*100

wfps:  PUSH(LP)
       PUSH(BP)
       MOVE(SP, BP)
       ALLOCATE(1)
       PUSH(R1)

       LD(BP, -12, R0)
       MULC(R0, 4, R0)
       LD(R0, STR, R1)
       ST(R1, 0, BP)
       BNE(R1, more)

       LD(BP, -16, R0)
       CMPEQC(R0, 0, R0)

rtn:   POP(R1)
       MOVE(BP, SP)
       POP(BP)
       POP(LP)
       JMP(LP)

more:  CMPEQC(R1, 1, R0)
       BF(R0, rpar)
       LD(BP, -16, R0)
       ADDC(R0, 1, R0)
       BR(par)

rpar:  LD(BP, -16, R0)
       BEQ(R0, rtn)
       ADDC(R0, -1, R0)

par:   PUSH(R0)
       LD(BP, -12, R0)
       ADDC(R0, 1, R0)
       PUSH(R0)
       BR(wfps, LP)
       DEALLOCATE(2)
       BR(rtn)
```

Problem 2 continued:

(A) (3 points) In the space below, fill in the binary value of the instruction stored at the location tagged '**`more:`**' in the above assembly-language program.

1 1 0 1 0 0	0 0 0 0 0	0 0 0 0 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

(fill in missing 1s and 0s for instruction at more:)

(B) (1 point) Is the value of the variable **`c`** from the C program stored in the local stack frame? If so, give its (signed) offset from **BP**; else write "**NO**".

Stack offset of variable `c`, or "NO": BP+0

(C) (1 point) Is the value of the variable **`new_n`** from the C program stored in the local stack frame? If so, give its (signed) offset from **BP**; else write "**NO**".

Stack offset of variable `new_n`, or "NO": NO or BP+8

(D) (2 points) What is the missing C source code represented by **`xxxxx`** in the given C program?

```
new_n = n - 1
```

(give missing C code shown as `xxxxx`)

The original intention of this problem was that the variable **`new_n`** was not stored on the local stack frame (unlike the variable **`c`**, which did have a space specifically allocated on the stack for it). However, it is true that when **`wfps`** makes a recursive call, it pushes the value of **`new_n`** onto the stack, so we ended up also accepting BP+8 as an answer.

Problem 2 continued again:

The procedure **`wfps`** is called from an external procedure and its execution is interrupted during a recursive call to **`wfps`**, just prior to the execution of the instruction labeled '**`rtn:`**'. The contents of a region of memory are shown to below on the left. At this point, **SP** contains 0x1D8, and **BP** contains 0x1D0.

NOTE: All addresses and data values are shown in hexadecimal.

```
    188:     7
    18C:   4A8
    190:     0
    194:     0
    198:   458
    19C:    D4
    1A0:     1
    1A4:    D8
    1A8:     1
    1AC:     1
    1B0:   3B8
    1B4:   1A0
    1B8:     2
    1BC:     1
    1C0:     0
    1C4:     2
    1C8:   3B8
    1CC:   1B8
BP->1D0:     2
    1D4:     2
SP->1D8:     0
```

(E) (1 point) What are the arguments to the *most recent* active call to **`wfps`**?

Most recent arguments (HEX): `i`=__2__**; `n`=**__0__

(F) (1 point) What are the arguments to the *original* call to **`wfps`**?

Original arguments (HEX): `i`=__0__**; `n`=**__0__

(G) (1 point) What value is in **R0** at this point?

Contents of R0 (HEX): __0__

(H) (1 point) How many parens (left and right) are in the string stored at **`STR`** (starting at index 0)? Give a number, or "**CAN'T TELL**" if the number can't be determined from the given information.

Length of string, or "CAN'T TELL": __CAN'T TELL__

(I) (1 point) What is the hex address of the instruction tagged **`par:`**?

Address of `par` (HEX): __39C__

(J) (1 point) What is the hex address of the **`BR`** instruction that called **`wfps`** originally?

Address of original call (HEX): __454__

Problem 3 (7 Points): **Beta control signals**

Following is an incomplete table listing control signals for several instructions on an unpipelined Beta. You may wish to consult the Beta diagram on the back of the previous page and the instruction set summary on the back of the first page.

The operations listed include two existing instructions and two proposed additions to the Beta instruction set:

```
LDX(Ra, Rb, Rc)                    // Load, double indexed
        EA ← Reg[Ra] + Reg[Rb]
        Reg[Rc] ← Mem[EA]
        PC ← PC + 4

MVZC(Ra, literal, Rc)              // Move constant if zero
        If Reg[Ra] == 0 then Reg[Rc] ← SEXT(literal)
        PC ← PC + 4
```

In the following table, **φ** represents a "**don't care**" or unspecified value; **Z** is the value (0 or 1) output by the 32-input NOR in the unpipelined Beta diagram. Your job is to complete the table by filling in each unshaded entry. In each case, enter an opcode, a value, an expression, or **φ** as appropriate.

Instr	ALUFN	WERF	BSEL	WDSEL	WR	RA2SEL	PCSEL	ASEL	WASEL
JMP	φ	1	φ	0	0	φ	2	φ	0
BEQ	φ	1	φ	0	0	φ	Z	φ	0
LDX	A+B	1	0	2	0	0	0	0	0
MVZC	A+B	Z	1	1	0	φ	0	0	0

(Complete the above table)

END OF QUIZ!
(phew!)