6.004 Computation Structures
Spring 2009

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.
Problem 1 (5 points): Quickies and Trickies

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

\[ \text{ADDC}(R0, 3*4+5, R1) \]

by

\[ \text{ADDC}(R0, 17, R1) \]

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 \text{BEQ} instruction resulting from the following assembly language snippet?

\[
. = 0x100 \\
\text{BEQ}(R31, \text{target}, R31) \\
\text{target: ADCDC}(R31, 0, R31)
\]

16-bit offset portion of above BEQ instruction: ________________

(D) Can every \text{SUBC} instruction be replaced by an equivalent \text{ADDC} instruction with the constant negated? If so, answer “YES”; if not, give an example of a \text{SUBC} instruction that can’t be replaced by an \text{ADDC}.

\text{SUBC}(…) instruction, or “YES”: ________________________________
Summary of Instruction Formats

Operate Class:

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>11</th>
<th>10</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>10xxxx</td>
<td>Rc</td>
<td>Ra</td>
<td>Rb</td>
<td>unused</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OP(Ra,Rb,Rc): \( \text{Reg}[\text{Rc}] \leftarrow \text{Reg}[\text{Ra}] \text{ op } \text{Reg}[\text{Rb}] \)

Opcodes: ADD (plus), SUB (minus), MUL (multiply), DIV (divided by)
AND (bitwise and), OR (bitwise or), XOR (bitwise exclusive or)
CMPEQ (equal), CMPLT (less than), CMPLE (less than or equal) [result = 1 if true, 0 if false]
SHL (left shift), SHR (right shift w/o sign extension), SRA (right shift w/ sign extension)

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>11xxxx</td>
<td>Rc</td>
<td>Ra</td>
<td>literal (two’s complement)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

OPC(Ra,literal,Rc): \( \text{Reg}[\text{Rc}] \leftarrow \text{Reg}[\text{Ra}] \text{ op SEXT(literal)} \)

Opcodes: ADDC (plus), SUBC (minus), MULC (multiply), DIVC (divided by)
ANDC (bitwise and), ORC (bitwise or), XORC (bitwise exclusive or)
CMPEQC (equal), CMPLTC (less than), CMPLEC (less than or equal) [result = 1 if true, 0 if false]
SHLC (left shift), SHRC (right shift w/o sign extension), SRAC (right shift w/ sign extension)

Other:

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>25</th>
<th>21</th>
<th>20</th>
<th>16</th>
<th>15</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>01xxxx</td>
<td>Rc</td>
<td>Ra</td>
<td>literal (two’s complement)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

LD(Ra,literal,Rc): \( \text{Reg}[\text{Rc}] \leftarrow \text{Mem}[\text{Reg}[\text{Ra}] + \text{SEXT(literal)}] \)
ST(Rc,literal,Ra): \( \text{Mem}[\text{Reg}[\text{Ra}] + \text{SEXT(literal)}] \leftarrow \text{Reg}[\text{Rc}] \)
JMP(Ra,Rc): \( \text{Reg}[\text{Rc}] \leftarrow \text{PC } + 4; \text{PC } \leftarrow \text{Reg}[\text{Ra}] \)
BEQ/BF(Ra,label,Rc): \( \text{Reg}[\text{Rc}] \leftarrow \text{PC } + 4; \text{if } \text{Reg}[\text{Ra}] = 0 \text{ then } \text{PC } \leftarrow \text{PC } + 4 + 4 \ast \text{SEXT(literal)} \)
BNE/BT(Ra,label,Rc): \( \text{Reg}[\text{Rc}] \leftarrow \text{PC } + 4; \text{if } \text{Reg}[\text{Ra}] \neq 0 \text{ then } \text{PC } \leftarrow \text{PC } + 4 + 4 \ast \text{SEXT(literal)} \)
LDR(label,Rc): \( \text{Reg}[\text{Rc}] \leftarrow \text{Mem}[\text{PC } + 4 + 4 \ast \text{SEXT(literal)}] \)

Opcode Table: (*optional opcodes)

<table>
<thead>
<tr>
<th>5:3</th>
<th>2:0</th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
<td>010</td>
<td>011</td>
<td>LD</td>
<td>ST</td>
<td>JMP</td>
<td>BEQ</td>
<td>BNE</td>
<td>LDR</td>
</tr>
<tr>
<td>100</td>
<td>ADD</td>
<td>SUB</td>
<td>MUL*</td>
<td>DIV*</td>
<td>CMPEQ</td>
<td>CMPLT</td>
<td>CMPLE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>101</td>
<td>AND</td>
<td>OR</td>
<td>XOR</td>
<td>SHL</td>
<td>SHR</td>
<td>SRA</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>ADDC</td>
<td>SUBC</td>
<td>MULC*</td>
<td>DIVC*</td>
<td>CMPEQC</td>
<td>CMPLTC</td>
<td>CMPLEC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>111</td>
<td>ANDC</td>
<td>ORC</td>
<td>XORC</td>
<td>SHLC</td>
<td>SHRC</td>
<td>SRAC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**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.

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

int wfps(int i, int n) { // current index in STR
    int c = STR[i]; // next character
    int new_n = 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.
}
```

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

6.004 Spring 2009 - 2 of 5 - Quiz #3
Scratch copies of code and memory snippet for Problem 2:

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

int wfps(int i, int n) { // current index in STR
    int c = STR[i]; // LPARENs to balance
    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!
xxxxxx; } // MYSTERY CODE!
    return wfps(i+1, new_n); // and recurse.
}
```

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

wfps: PUSH(LP)
PUSH(LP)
MOVE(SP, BP)
ALLOCATE(1)
PUSH(R1)  // Memory
LD(BP, -12, R0)  // address: Data
188: 7
MULC(R0, 4, R0)  
18C: 4A8
LD(R0, STR, R1)  
190: 0
ST(R1, 0, BP)  
194: 0
BNE(R1, 0, BP)  
198: 458
MULC(R0, 4, R0)  
1A0: D4
LD(BP, -16, R0)  
1A4: D8
CMPEQC(R0, 0, R0)  
1AC: 1
LD(BP, -16, R0)  
1B0: 3B8
CMPEQC(R1, 1, R0)  
1B4: 1A0
BF(R0, rpar)  
1BC: 1
LD(BP, -16, R0)  
1C0: 0
ADDC(R0, 1, R0)  
1C4: 2
BR(par)  
1C8: 3B8
CMPEQC(R1, 1, R0)  
1CC: 1B8
BF(R0, rpar)  
1D0: 2
LD(BP, -12, R0)  
1D4: 2
ADDCC(R0, -1, R0)  
SP->1D8: 0
ADDCC(R0, -1, R0)  
BP->1D0: 2
ADDCC(R0, -1, R0)  
BP->1D0: 2
ADDCC(R0, -1, R0)  
BP->1D0: 2
PUSH(R0)  
BP->1D0: 2
ADDCC(R0, -1, R0)  
BP->1D0: 2
ADDCC(R0, -1, R0)  
BP->1D0: 2
PUSH(R0)  
BP->1D0: 2
BR(rtn)
```

Scrat
ch copies of code and memory snippet for Problem 2:

```assembly
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!
xxxxxx; } // MYSTERY CODE!
    return wfps(i+1, new_n); // and recurse.
}
```
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.

\[
\begin{array}{cccccccccccc}
\hline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\hline
\end{array}
\]

(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”: ______________

(C) (1 point) Is the value of the variable \(\text{new\_n}\) 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 \(\text{new\_n}\), or “NO”: ______________

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

\[
\begin{array}{cccccccccccc}
\hline
\hline
\hline
\hline
\end{array}
\]

(give missing C code shown as \(xxxxx\))
Scratch copies of code and memory snippet for Problem 2:

```c
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.
}
```

```assembly
STR: . = .+4*100
wfps: PUSH(LP)
      PUSH(BP)
      MOVE(SP, BP)
      ALLOCATE(1)
      PUSH(R1)
      LD(BP, -12, R0) 188: 7
      MULC(R0, 4, R0) 18C: 4A8
      LD(R0, STR, R1) 190: 0
      ST(R1, 0, BP) 194: 0
      BNE(R1, 0, BP) 198: 458
      CMPEQC(R0, 0, R0) 19C: D4
      ALLOCATE(2)
      CMPEQC(R0, 0, R0) 1A0: 1
      RETURN
more: CMPEQC(R0, 1, R0) 1B8: 2
      BF(R0, rpar) 1BC: 1
      LD(BP, -16, R0) 1C0: 0
      ADDC(R0, 1, R0) 1C4: 2
      BR(par) 1C8: 3B8
      deallocates
rpar: LD(BP, -16, R0) 1CC: 1B8
      BEQ(R0, rtn) 1D0: 2
      ADDC(R0, -1, R0) 1D4: 2
par:  PUSH(LP)
      BR(rtn)
```
Problem 2 continued again:

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

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

(E) (1 point) What are the arguments to the \textit{most recent} active call to \texttt{wfps}?

Most recent arguments (HEX): \texttt{i=\_\_\_\_\_\_; n=\_\_\_\_\_}\n
(F) (1 point) What are the arguments to the \textit{original} call to \texttt{wfps}?

Original arguments (HEX): \texttt{i=\_\_\_\_\_\_; n=\_\_\_\_\_}\n
(G) (1 point) What value is in \texttt{R0} at this point?

Contents of \texttt{R0} (HEX): ________

(H) (1 point) How many parens (left and right) are in the string stored at \texttt{STR} (starting at index 0)? Give a number, or “\textsc{can’t tell}” if the number can’t be determined from the given information.

Length of string, or “\textsc{can’t tell}”: ________________

(I) (1 point) What is the hex address of the instruction tagged \texttt{par} : ?

Address of \texttt{par} (HEX): ________

(J) (1 point) What is the hex address of the \texttt{BR} instruction that called \texttt{wfps} originally?

Address of original call (HEX): ________
Control logic:

<table>
<thead>
<tr>
<th></th>
<th>OP</th>
<th>OPC</th>
<th>LD</th>
<th>ST</th>
<th>JMP</th>
<th>BEQ</th>
<th>BNE</th>
<th>LDR</th>
<th>ILOP</th>
<th>IRQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALUFN</td>
<td>F(op)</td>
<td>F(op)</td>
<td>A=B</td>
<td>A=B</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>A</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>WERF</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>BSEL</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>WDSEL</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>--</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>WR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>RA2SEL</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>PCSEL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>Z</td>
<td>~Z</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>ASEL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>--</td>
<td>--</td>
<td>1</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>WASEL</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>--</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
**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:

\[
\text{LDX}(Ra, Rb, Rc) \quad \text{// Load, double indexed}
\]
\[
\begin{align*}
EA & \leftarrow \text{Reg}[Ra] + \text{Reg}[Rb] \\
\text{Reg}[Rc] & \leftarrow \text{Mem}[EA] \\
\text{PC} & \leftarrow \text{PC} + 4
\end{align*}
\]

\[
\text{MVZC}(Ra, \text{literal}, Rc) \quad \text{// Move constant if zero}
\]
\[
\begin{align*}
\text{If } \text{Reg}[Ra] & = 0 \text{ then } \text{Reg}[Rc] \leftarrow \text{SEXT(literal)} \\
\text{PC} & \leftarrow \text{PC} + 4
\end{align*}
\]

In the following table, $\phi$ 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 $\phi$ as appropriate.

<table>
<thead>
<tr>
<th>Instr</th>
<th>ALUFN</th>
<th>WERF</th>
<th>BSEL</th>
<th>WDSEL</th>
<th>WR</th>
<th>RA2SEL</th>
<th>PCSEL</th>
<th>ASEL</th>
<th>WASEL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\phi$</td>
<td>$\phi$</td>
<td>0</td>
<td>0</td>
<td>$\phi$</td>
<td>2</td>
<td>$\phi$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\phi$</td>
<td>1</td>
<td>$\phi$</td>
<td>0</td>
<td>$\phi$</td>
<td>$Z$</td>
<td>$\phi$</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>LDX</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A+B</td>
<td>$Z$</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$\phi$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

(Complete the above table)

**END OF QUIZ!**

(phew!)
**Convenience Macros**

We augment the basic $\beta$ instruction set with the following macros, making it easier to express certain common operations:

<table>
<thead>
<tr>
<th>Macro</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEQ(Ra, label)</td>
<td>BEQ(Ra, label, R31)</td>
</tr>
<tr>
<td>BF(Ra, label)</td>
<td>BF(Ra, label, R31)</td>
</tr>
<tr>
<td>BNE(Ra, label)</td>
<td>BNE(Ra, label, R31)</td>
</tr>
<tr>
<td>BT(Ra, label)</td>
<td>BT(Ra, label, R31)</td>
</tr>
<tr>
<td>BR(label, Rc)</td>
<td>BEQ(R31, label, Rc)</td>
</tr>
<tr>
<td>BR(label)</td>
<td>BR(label, R31)</td>
</tr>
<tr>
<td>JMP(Ra)</td>
<td>JMP(Ra, R31)</td>
</tr>
<tr>
<td>LD(label, Rc)</td>
<td>LD(R31, label, Rc)</td>
</tr>
<tr>
<td>ST(Rc, label)</td>
<td>ST(Rc, label, R31)</td>
</tr>
<tr>
<td>MOVE(Ra, Rc)</td>
<td>ADD(Ra, R31, Rc)</td>
</tr>
<tr>
<td>CMOVE(c, Rc)</td>
<td>ADDC(R31, c, Rc)</td>
</tr>
<tr>
<td>PUSH(Ra)</td>
<td>ADDC(SP, 4, SP)</td>
</tr>
<tr>
<td></td>
<td>ST(Ra, -4, SP)</td>
</tr>
<tr>
<td>POP(Rc)</td>
<td>LD(SP, -4, Rc)</td>
</tr>
<tr>
<td></td>
<td>SUBC(SP, 4, SP)</td>
</tr>
<tr>
<td>ALLOCATE(k)</td>
<td>ADDC(SP, 4*k, SP)</td>
</tr>
<tr>
<td>DEALLOCATE(k)</td>
<td>SUBC(SP, 4*k, SP)</td>
</tr>
</tbody>
</table>