Problem 1. Hand-compile the following C fragments into Beta assembly language. You can assume that the necessary storage allocation for each variable or array has been done and that a UASM label has been defined that indicates the first storage location for that variable or array. All of the variables are stored in main memory (in the first 32k bytes of main memory so that they can be addressed by a 16-bit literal). You can also assume that all variables and arrays are C integers, i.e., 32-bit values.

A. ★ Explain what Beta assembly language instruction(s) are needed to load the value of a variable that has been allocated in the first 32k bytes of main memory (i.e., at an address less than 0x8000). How would your answer change if the variable was located at address outside this range (e.g., at address 0x12468).

If the storage for the variable is located at an address less than 0x8000, the 16-bit constant field of the LD instruction can hold the complete address. Note that the 16-bit constant is sign-extended, so our address has to fit in 15 bits. So LD(R31,addr,R0) would load the contents Mem[addr] into R0 assuming addr < 0x8000. For addresses >= 0x8000 the 16-bit constant field isn't large enough to hold the address. In these cases one could use the LDR instruction to load a 32-bit address into a register and the use LD to fetch the data:

```
vaddr:
    LONG(0x12468)
    ...
    LDR(vaddr,R0)    ; load address of variable into R0
    LD(R0,0,R0)      ; load Mem[address] into R0
```

B. ★ a = b + 3*c;

```
LD(c,R1)
SHLC(R1,1,R0)    ; 2*c
ADD(R0,R1,R0)    ; + c
LD(b,R1)
ADD(R1,R0,R0)
ST(R0,a)
```

C. ★ if (a > b) c = 17;

```
LD(a,R0)
LD(b,R1)
CMPLR((R0,R1,R0)
BT(R0,_L2)
CMOVE(17,R0)
ST(R0,c)

_L2:
```

D. if (sxt_short) { b = (b << 16) >> 16; }

```
LD(sxt_short,R0)
```
Problem 2. In block structured languages such as C or Java, the scope of a variable declared locally within a block extends only over that block, i.e., the value of the local variable cannot be accessed outside the block. Conceptually, storage is allocated for the variable when the block is entered and deallocated when the block is exited. In many cases, this means the compiler if free to use a register to hold the value of the local variable instead of a memory location.

Consider the following C fragment:

```c
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+1) sum += i;
}
```
A. Hand-compile this loop into assembly language, using registers to hold the values of the local variables "i" and "sum".

```
MOVE(R31,R2) ; R2 holds sum
ST(R2,sum)
MOVE(R31,R1) ; R1 holds i

_L5:
ADD(R2,R1,R2)
ADDC(R1,1,R1)
CMPLTC(R1,10,R0)
BT(R0,_L5)
ST(R2,sum)
```

B. Define a memory access as any access to memory, i.e., instruction fetch, data read (LD), or data write (ST). Compare the number of total number of memory accesses generated by executing the optimized loop with the total number of memory access for the unoptimized loop (part G of the preceding problem).

The unoptimized code has an 8 instruction loop that makes 4 data accesses; 10 loop iterations => 120 memory accesses. There are 4 additional memory accesses to initialize sum and i. Total = 124.

The optimized code has a 4 instruction loop that makes 0 data accesses; 10 loop iterations => 40 memory accesses. There are 6 additional memory accesses to initializes sum and i, and to store sum at the end of the loop. Total = 46.

C. Some optimizing compilers "unroll" small loops to amortize the overhead of each loop iteration over more instructions in the body of the loop. For example, one unrolling of the loop above would be equivalent to rewriting the program as

```
int sum = 0;
{ int i;
  for (i = 0; i < 10; i = i+2) { sum += i; sum += i+1; }
}
```

Hand-compile this loop into Beta assembly language and compare the total number of memory accesses generated when it executes to the total number of memory accesses from part (1).

```
MOVE(R31,R2) ; R2 holds sum
ST(R2,sum)
MOVE(R31,R1) ; R1 holds i

_L5:
ADD(R2,R1,R2)
ADDC(R1,1,R0)
ADD(R2,R0,R2)
ADDC(R1,2,R1)
CMPLTC(R1,10,R0)
BT(R0,_L5)
ST(R2,sum)
```

This code has a 6 instruction loop that makes 0 data accesses; 5 loop iterations => 30 memory accesses. There are 6 additional memory accesses to initializes sum and i, and to store sum at the end of the loop. Total = 36.
Problem 3.

A. ★ Hand-assemble the following Beta assembly language program:

\[
\begin{align*}
\text{I} &= 0x5678 \\
\text{B} &= 0x1234 \\
\text{LD}(\text{I}, \text{R0}) \\
\text{SHLC}(\text{R0}, 2, \text{R0}) \\
\text{LD}(\text{R0}, \text{B}, \text{R1}) \\
\text{MULC}(\text{R1}, 17, \text{R1}) \\
\text{ST}(\text{R1}, \text{B}, \text{R0}) \\
\text{I} &= 0x5678 \\
\text{B} &= 0x1234 \\
\text{LD}(\text{R31}, \text{I}, \text{R0}) &= 011000 00000 11111 0101 0110 0111 1000 = 0x601F5678 \\
\text{SHLC}(\text{R0}, 2, \text{R0}) &= 111100 00000 00000 0000 0000 0010 = 0xF0000002 \\
\text{LD}(\text{R0}, \text{B}, \text{R1}) &= 011000 00001 00000 0001 0010 0011 0100 = 0x60201234 \\
\text{MULC}(\text{R1}, 17, \text{R1}) &= 110010 00001 00001 0000 0000 0001 0001 = 0xC8210011 \\
\text{ST}(\text{R1}, \text{B}, \text{R0}) &= 011001 00001 00000 0001 0010 0011 0100 = 0x64201234
\end{align*}
\]

B. What C statement might have been compiled into the code fragment above?

\[B[I] = B[I] * 17;\]

Problem 4. Hand-assemble the following Beta branch instructions into their binary representation:

A. foo: BR(foo) [recall that BR(label) = BEQ(R31,label,R31)]

\[
\begin{align*}
\text{BEQ} & \quad \text{R31} \quad \text{R31} \quad \text{offset} = -1 \\
011101 & \quad 11111 \quad 11111 \quad 1111 \quad 1111 \quad 1111
\end{align*}
\]

B. BR(bar)

bar:

\[
\begin{align*}
\text{BEQ} & \quad \text{R31} \quad \text{R31} \quad \text{offset} = 0 \\
011101 & \quad 11111 \quad 11111 \quad 0000 \quad 0000 \quad 0000 \quad 0000
\end{align*}
\]

C. ★ foo = 0x100

. = 0x1000

BF(R17,foo,R31)

\[
\begin{align*}
\text{BEQ} & \quad \text{R31} \quad \text{R17} \quad \text{offset} = (0x100 - 0x1004)/4 = 0xFC3F \\
011101 & \quad 11111 \quad 10001 \quad 1111 \quad 1100 \quad 0011 \quad 1111
\end{align*}
\]
D. Explain why PC-relative branch addressing is a good choice for computers like the Beta that can encode only a "small" constant in each instruction.

Branches are used to implement conditional and looping constructs (e.g., if, while, for). So most branch targets are just a few instructions away. With PC-relative addressing, we can reach targets 32767 instructions before or 32768 instructions after the branch, independently of the actual absolute address of the branch. So used as an offset, the 16-bit constant can accommodate most branch targets even for very large programs. Used as an absolute address, branch targets would be constrained to be in the first 32K of memory.

E. Suppose a different computer could encode an arbitrary 32-bit constant in an instruction (using, e.g., a variable-length instruction encoding). Would PC-relative addressing still make sense? Why?

Even if a complete absolute address could be encoded in an instruction, a PC-relative address has the advantage that it represents "position independent code" -- that is, an entire instruction sequence (including local branches) can be moved to another location in memory without changing the branch offsets. If branches included absolute addresses, these fields would have to be changed to reflect the location of the code.

Problem 5.

A. True or false: The Beta SUBC opcode could be eliminated since every SUBC instruction can be replaced an equivalent ADDC instruction.

False: SUBC(Rx,0x8000,Rx) subtracts -32768 from Rx. The ADDC equivalent would add 32768 to Rx, but we can't express that constant in the signed, 16-bit constant field provided in the Beta instruction format.

B. What is the binary representation for the Beta instruction SUBC(R17,12,R22)?

```
SUBC   R22   R17          12
110001 10110 10001 0000 0000 0000 1100 = 0xC6D1000C
```

C. A certain TA wants to know what would happen if the Beta as implemented in the lab executed 0xEDEDEDED as an instruction. What does happen?

```
0xEDEDEDED = 111011 01111 01101 1110 1101 1110 1101
The opcode field corresponds to an illegal instruction opcode which causes the beta to take a trap (saving the PC+4 of the offending instruction in the XP register) and set the PC to ILLOP.
```

D. Suppose that the Beta instruction BR(error) were assembled into memory location 0x87654. Assuming that the instruction works as intended (i.e., when executed, control is transferred to the first instruction in the error routine), which of the following is the best statement about the possible values for the symbol "error"?

A. it depends on the first instruction in the error routine.
B. it can have any 32-bit value
C. it can have any 32-bit value that is a multiple of 4
D. it is a multiple of 4 in the range 0x7F658 to 0x8F654 inclusive.
E. it is a multiple of 4 in the range 0x67658 to 0xA7654 inclusive.
F. none of the above
A branch instruction in which the branch is taken will multiply the sign-extended 16-bit literal field by 4 and add it to PC+4.

\[
\text{if literal} = 0x8000 \text{ then} \\
\text{new PC} = 0x87654 + 4 - (8000 \times 4) = 0x67658 \\
\text{if literal} = 0x7FFF \text{ then} \\
\text{new PC} = 0x87654 + 4 + (7FFF \times 4) = 0xA7654
\]

**Problem 6.** The Meta is a processor similar to the Beta, except that the data paths have been modified to accommodate the addition of a new Subtract One and Branch instruction:

**Usage:** \(\text{SOB(Ra, label, Rc)}\)

**Operation:**
- \(\text{literal} = \left(\frac{\text{OFFSET(label)} - \text{OFFSET(current inst)}}{4}\right) - 1\)
- \(\text{PC} = \text{PC} + 4\)
- \(\text{EA} = \text{PC} + 4 \times \text{SEXT(literal)}\)
- \(\text{Reg[Rc]} = \text{Reg[Ra]} - 1\)
- \(\text{if (Reg[Ra] - 1) \neq 0 \text{ then PC} = \text{EA}}\)

As with branches in the Beta, the binary encoding of the SOB instruction places the low-order 16 bits of the "literal" value in the low-order 16 bits of the instruction. The designers of the Meta implementation have used the Meta's ALU to perform the subtraction.

A. Suppose R1 contains the value 1. How will executing \(\text{SOB(R1, label, R31)}\) change register R1 and the PC?

R1 is unchanged since the destination register (Rc) of the example SOB instruction is R31. \(\text{Reg[R1]} - 1 = 0\), so the branch is *not* taken and so the PC will point to the instruction following the SOB.

B. Consider the following instruction sequence:

```
loop: \text{ADD(R1, R2, R3)} \\
\text{SOB(R4, loop, R4)}
```

Assuming the ADD instruction is placed in location 0x108 of memory, what are the contents of the low-order 16 bits of the SOB instruction?

Actually we don't need to know the address of the ADD instruction to answer the question since the SOB instruction (like all Beta branches) uses PC-relative addressing. Remembering that the branch offset is computed from the PC of the instruction following the SOB, the correct contents of the offset field is \(-2 = 0xFFFF\).

C. A schematic for the adder circuitry in the ALU of the Meta is shown below:
What would be the correct values for OP\[2:0\] in order to perform a subtract (i.e., SUM = A - B)?

\[
\text{SUM} = A - B = A + (\neg B + 1).
\]
Setting OP2 = 1 and OP1 = 0 selects \(\neg B\) as the XB input to the 32-bit add, setting OP0 = 1 asserts the carry in for the low-order bit of the 32-bit add and hence provides the required "+1". So OP\[2:0\] = 0b101.

D. What would be the correct values for OP\[2:0\] in order to perform the decrement needed for the SOB instruction (i.e., SUM = A - 1)?

If OP\[2:0\] = 0b110, the XB input is set to (B or \(\neg B\)) = all ones, the two's complement representation for -1. Carry-in should be set to 0.

E. Is it possible to use the logic above to do an increment (i.e., SUM = A+1)?

Yes, OP\[2:0\] = 0b001, setting XB to 0 and the carry-in to 1.

---

Problem 7. A local junk yard offers older CPUs with non-Beta architectures that require several clocks to execute each instruction. Here are the specifications:

<table>
<thead>
<tr>
<th>Model</th>
<th>Clock Rate</th>
<th>Avg. clocks/Inst.</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>40 Mhz</td>
<td>2.0</td>
</tr>
<tr>
<td>y</td>
<td>100 Mhz</td>
<td>10.0</td>
</tr>
<tr>
<td>z</td>
<td>60 Mhz</td>
<td>3.0</td>
</tr>
</tbody>
</table>

You are going to choose the machine which will execute your benchmark program the fastest, so you compiled and ran the benchmark on the three machines and counted the total instructions executed:

- x: 3,600,000 instructions executed
- y: 1,900,000 instructions executed
- z: 4,200,000 instructions executed

A. Based on the above data which machine would you choose?

Total execution time:

- x: \((3,600,000 \text{ insts})(2 \text{ clocks/inst})(25 \text{ ns/inst}) = 0.18 \text{ seconds}\)
- y: \((1,900,000 \text{ insts})(10 \text{ clocks/inst})(10 \text{ ns/inst}) = 0.19 \text{ seconds}\)
- z: \((4,200,000 \text{ insts})(3 \text{ clocks/inst})(16.67 \text{ ns/inst}) = 0.21 \text{ seconds}\)
Problem 8. Kerry DeWay is proposing to add a "Load Constant" instruction LDC(const,Rx) to the Beta instruction set. LDC loads the 32-bit constant const in register Rx. She can't convince the hardware team to implement LDC directly and consequently plans to define it as a macro. She is considering the following alternative implementations:

[1] .macro LDC(const,Rx) {
    LD(.+8,Rx)
    BR(.+8)
    LONG(const)
}

[2] .macro LDC(const,Rx) {
    PUSH(R17)
    BR(.+8,R17)
    LONG(const)
    LD(R17,0,Rx)
    POP(R17)
}

[3] .macro LDC(const,Rx) {
    ADDC(R31,const >> 16,Rx)
    SHLC(Rx,16,Rx)
    ADDC(Rx,const & 0xFFFF,Rx)
}

Kerry tries each definition on a few test cases and convinces herself each works fine. The Quality Assurance team isn't so sure and complains that Kerry's LDC implementations don't all work for every choice of register (Rx), every choice of constant (const), and every choice of code location.

A. Evaluate each approach and decide whether it works under all circumstances or if it fails, indicate that it misbehaves for certain choices of Rx, const or code location.

[1] fails if the code is located so that the LD instruction is at, e.g., address 0x7FFC since we can't represent .+8 = 0x8004 in the 16-bit literal field of the LD instruction.

[2] fails for LDC(const,R17) since the POP(R17) at the end of the macro restores the old value of R17, wiping out the constant we just loaded.

[3] fails for any const which has bit 15 set (e.g., 0x8000) since the final ADDC will sign-extended its literal field, adding 0xFFFF to the high half of Rx.

Problem 9. Which of the following Beta instruction sequences might have resulted from compiling the following C statement?

X ran the benchmark the fastest.
int x[20], y;
y = x[1] + 4;

A. LD (R31, x + 1, R0)
   ADDC (R0, 4, R0)
   ST (R0, y, R31)

   Not this one. If x[0] is stored at location x, x[1] is stored at location x + 4 since x[] is an integer array and each integer takes one word (4 bytes).

B. CMOVE (4, R0)
   ADDC (R0, x + 4, R0)
   ST (R0, y, R31)

   Not this one. The second instructions adds the address of x[1] to R0, not the contents of x[1].

C. LD (R31, x + 4, R0)
   ST (R0, y + 4, R31)

   Not this one. This stores x[1] in the location following the one word of storage allocated for y.

D. ★ CMOVE (4, R0)
   LD (R0, x, R1)
   ST (R1, y, R0)

   Not this one. This implements y[1] = x[1].

E. ★ LD (R31, x + 4, R0)
   ADDC (R0, 4, R0)
   ST (R0, y, R31)

   Yes!

F. ★ ADDC (R31, x + 1, R0)
   ADDC (R0, 4, R0)
   ST (R0, y, R31)

   Not this one. The ADDC instruction loads the address of x plus 1 into R0.

Problem 10. An unnamed associate of yours has broken into the computer (a Beta of course!) that 6.004 uses for course administration. He has managed to grab the contents of the memory locations he believes holds the Beta code responsible for checking access passwords and would like you to help discover how the password code works. The memory contents are shown in the table below:

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents (in hexadecimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xC05F0008</td>
</tr>
<tr>
<td>0x104</td>
<td>0xC03F0000</td>
</tr>
</tbody>
</table>
A. Reconstruct the Beta assembly code that corresponds to the binary instruction encoding shown above. If the code sequence contains branches, be sure to indicate the destination of each branch.

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
<th>Opcode</th>
<th>Rc</th>
<th>Ra</th>
<th>Rb</th>
<th>Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xC05F0008</td>
<td>110000</td>
<td>00010</td>
<td>11111</td>
<td></td>
<td>ADDC(R31, 0x8, R2)</td>
</tr>
<tr>
<td>0x104</td>
<td>0xC03F0000</td>
<td>110000</td>
<td>00001</td>
<td>11111</td>
<td></td>
<td>ADDC(R31, 0x0, R1)</td>
</tr>
<tr>
<td>0x108</td>
<td>0xE060000F</td>
<td>111000</td>
<td>00011</td>
<td></td>
<td>00000</td>
<td>ANDC(R0, 0xF, R3)</td>
</tr>
<tr>
<td>0x10C</td>
<td>0xF0210004</td>
<td>111100</td>
<td>00001</td>
<td>00001</td>
<td></td>
<td>SHLC(R1, 0x4, R1)</td>
</tr>
<tr>
<td>0x110</td>
<td>0xA4230800</td>
<td>101001</td>
<td>00001</td>
<td>00011</td>
<td>00001</td>
<td>OR(R3, R1, R1)</td>
</tr>
<tr>
<td>0x114</td>
<td>0xF4000004</td>
<td>111101</td>
<td>00000</td>
<td>00000</td>
<td></td>
<td>SHRC(R0, 0x4, R0)</td>
</tr>
<tr>
<td>0x118</td>
<td>0xC4420001</td>
<td>110001</td>
<td>00010</td>
<td>00001</td>
<td></td>
<td>SUBC(R2, 0x1, R2)</td>
</tr>
<tr>
<td>0x11C</td>
<td>0x77E20002</td>
<td>011101</td>
<td>11111</td>
<td>00010</td>
<td></td>
<td>BEQ(R2, 0x128) *</td>
</tr>
<tr>
<td>0x120</td>
<td>0xF0211000</td>
<td>011101</td>
<td>11111</td>
<td>11111</td>
<td></td>
<td>BEQ(R3, 0x128) **</td>
</tr>
<tr>
<td>0x124</td>
<td>0xA4230800</td>
<td>101001</td>
<td>00001</td>
<td>00011</td>
<td></td>
<td>not an opcode</td>
</tr>
<tr>
<td>0x128</td>
<td>0x605F0124</td>
<td>011000</td>
<td>00010</td>
<td>11111</td>
<td></td>
<td>LD(0x0124, R2)</td>
</tr>
<tr>
<td>0x12C</td>
<td>0x90211000</td>
<td>100100</td>
<td>00001</td>
<td>00001</td>
<td>00010</td>
<td>CMPEQ(R1, R2, R1)</td>
</tr>
</tbody>
</table>

* The literal in instruction 0x11C is 0x2, so the corresponding label in Beta assembly is
  
  \[
  \text{PC} + 4 + 4\times\text{literal} = 0x11C + 4 + 4\times2 = 0x128
  \]

** In instruction 0x120, SEXT(literal) = -7, so the corresponding label in Beta assembly is

  \[
  \text{PC} + 4 + 4\times\text{literal} = 0x120 + 4 + 4\times(-7) = 0x124 - 0x01C = 0x108
  \]

B. Further investigation reveals that the password is just a 32-bit integer which is in R0 when the code above is executed and that the system will grant access if R1 = 1 after the code has been executed. What "passnumber" will gain entry to the system?

Let's analyze this assembly by translating it to pseudo-code:

```
R2 = 8; /* R2 is used as a counter */
R1 = 0;

loop:  R3 = R0 & 0xF;  /* R3 stores the current low nibble of R0 */
       R1 = R1 << 4;
       R1 = R3 | R1;
       R0 = R0 >> 4;
       R2 = R2 - 1;
       if R2 == 0 goto done;
       goto loop;
```
data: 0xA4230800

done: LD(data, R2);
    if (R1 == R2)
        R1 = 1;
    else
        R1 = 0;

We can see that the code shifts R1 left by a nibble (4 bits) and ors it with the low nibble (R3) of the user's entered password (R0). It then shifts the user's password right by a nibble and loops back to the beginning. It does this a total of 8 times. The net effect is to reverse the order of the nibbles in R0 and to store this into R1. The result is then compared to 0xA4230800. Therefore, in order for the entered password to be accepted, it must be the nibble-reversed version of 0xA4230800.

Thus, the "passnumber" required to enter is 0x0080324A