Please enter your name and Athena login name in the spaces above; indicate where you want to pick up your quiz. Enter your answers in the spaces provided after each question. You can use the extra white space and the backs of the pages for scratch work.

**Problem 1. Cache Hit Rates (6 points)**

The following questions ask you to evaluate alternative cache designs using a specific pattern of memory references. Each of the caches under consideration has a total capacity of 8 (4-byte) words, with one word stored in each cache line. The cache designs under consideration are:

- **DM**: a direct-mapped cache.
- **S2**: a 2-way set-associative cache with a least-recently-used (LRU) replacement policy.

The questions below present a repeating sequence of addresses (expressed as decimal numbers) for memory reads. After the access to the last address on the list, the sequence starts over with the first address on the list. Keep in mind that byte addressing is used; addresses of consecutive words in memory differ by 4. Each question asks for the hit ratio for each cache. Answer by considering the steady-state hit ratio, i.e., the percentage of memory references that hit in the cache after the sequence has been repeated many times. Your answer can be expressed as a percentage or a ratio.

<table>
<thead>
<tr>
<th>Repeating address sequence</th>
<th>Hit ratio for cache DM</th>
<th>Hit ratio for cache S2</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A) 0, 8, 16, 24, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(B) 0, 4, 8, 12, 16, 20, 24, 28, 32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(C) 0, 4, 8, 12, 32, 36, 40, 44</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 2. Stack Detective (12 Points)

The following C program implements a function \( L(x, y) \) of two arguments, which returns an integer result. The assembly code for the procedure is shown on the right.

\[
\text{int } L(\text{int } x, \text{int } y) \{ \\
    \text{int } a = x/2; \\
    \text{if (} a == 0 \text{) return } y; \\
    \text{else return } ???; \\
\}
\]

The execution of the procedure call \( L(0xB00, 0) \) has been suspended just as the Beta is about to execute the \( \text{JMP}(LP) \) at the end of the procedure during one of the recursive calls to \( L \). A partial trace of the stack at the time execution was suspended is shown to the right below.

(A) (2 Points) Examining the assembly language for \( L \), what is the appropriate C code for ??? in the C representation for \( L \)?

\[ \text{C code for ???: } \quad \] ______________________________________________________________________

(B) (1 Point) If the \( \text{LD(BP,0,R0)} \) instruction at label \( yy: \) was rewritten as \( \text{LD(SP,NNN,R0)} \), what is the correct value for the literal \( NNN \)?

\[ \text{Value for NNN?: } \quad \] ______________________________________________________________________

(C) (4 Points) Please fill in the values for the blank locations in the stack dump shown on the right. Express the values in hex or write “---“ if value can’t be determined. Hint: Figure out the layout of \( L \)’s activation record and use it to identify and label the stack frames in the stack dump.

\[ \text{Fill in the blank locations with values (in hex!) or “---“} \]

(D) (5 Points) Determine the specified values at the time execution was suspended. Please express each value in hex or write “CAN’T TELL” if the value cannot be determined.

\[ \text{Value in R0 or “CAN’T TELL”: } 0x \] ________________

\[ \text{Value in BP or “CAN’T TELL”: } 0x \] ________________

\[ \text{Value in LP or “CAN’T TELL”: } 0x \] ________________

\[ \text{Value in SP or “CAN’T TELL”: } 0x \] ________________

\[ \text{Value in PC or “CAN’T TELL”: } 0x \] ________________

\[ \begin{array}{c}
0x02C0 \\
0x0002 \\
0x02C0 \\
0x0490 \\
0x0490 \\
0x0158 \\
0x00B0 \\
0x0004 \\
0x00B0 \\
0x0490 \\
0x016C \\
0x0058 \\
0x0005 \\
\end{array} \]

\[ \text{BP} \rightarrow \]
Problem 3. Memory Hierarchy and Cache Accesses (8 points)

(A) (1 point)

Assuming the memory hierarchy configuration shown above which consists of a level 1 (L1) cache with access time of 4 ns and hit rate of 0.6, and a level 2 (L2) cache with access time of 8 ns and hit rate of 0.9, and a main memory whose access time is 100 ns, determine the AMAT (average memory access time) for this system.

AMAT (ns): ______________

(B) (1 point) True or False. A 4 line 2-way set associative cache always has a better hit rate than a direct mapped cache with 8 lines.

TRUE      FALSE

For parts C-G, use the 2-line 2-way set associative cache with a block size of 2, shown below.

(C) (1 point) Which address bits are used for the offset, cache line index, and tag comparison?

Address bits used as offset (including byte offset): A[ _____ : _____ ]

Address bits used as cache line index: A[ _____ : _____ ]

Address bits used for tag comparison: A[ _____ : _____ ]

(D) (1 point) Give the address corresponding to the data 0x23 (line 0, data word B1) or write CAN’T TELL if there’s insufficient information to determine its address.

CAN’T TELL of address of 0x23 data (0x): ______________

(E) (1 point) Would a store to address 0x71C hit or miss?

HIT      MISS
(F) (1 point) Suppose that the CPU executes a ST instruction writes the value 7 into location 0x71C. Fill in all changed entries for the cache in the unfilled cache below.

For parts G and H compute the line number and tag for each data access. Indicate if the access will result in a hit or a miss. Provide the value returned by the LD, or enter CAN’T TELL if you can’t tell from the information provided. (1 point each)

<table>
<thead>
<tr>
<th>Access</th>
<th>Line number</th>
<th>Tag</th>
<th>Hit? (y or n)</th>
<th>Value returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>(G) LD from address 0x528</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(H) LD from address 0x430</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Problem 4. Multiway branches (7 points)

Multiway branches change a program’s control flow by matching a variable against a set of possible values. In C, multiway branches can be expressed using a switch statement of a sequence of if statements. Figures 1 and 2 below show two equivalent implementations using each approach, and Figure 3 shows the Beta code that our compiler produces.

```
switch (x) {
  case 0: A(); break;
  case 1: B(); break;
  case 2: C(); break;
  case 3: D(); break;
}
```

**(A) (1 point)** Using the sequence-of-comparisons implementation in Figure 3, how does the number of instructions required to implement a multiway branch grow with the number of possible values N?

Number of instructions (exact number or order-of notation): ____________

To improve performance, we want to add a multiway branch instruction to the Beta ISA:

```
MBR(Ra, literal)
PC ← PC + 4 + 4*SXT(literal)*Reg[Ra]
```

This instruction adjusts the PC by an offset that depends both on the value of register Ra and the 16-bit sign-extended literal. The purpose of the literal is to allow for multiple instructions to handle each possible value of Ra. Figures 4 and 5 illustrate MBR with two examples. In Figure 4, MBR’s literal is 1, so if R1 is 0, the next instruction will be ADDC; if R1 is 1, the next instruction will be SUBC; and so on. In Figure 5, MBR’s literal is 2, so if R1 is 0, the next instruction will be ADDC; if R1 is 1, the next instruction will be ORC; and so on.

```
MBR(R1, 1)
ADDC(R0, 1, R2)
SUBC(R0, 1, R2)
ORC(R0, 1, R2)
SHLC(R0, 1, R2)
```

**(A) (1 point)** Using the sequence-of-comparisons implementation in Figure 3, how does the number of instructions required to implement a multiway branch grow with the number of possible values N?
(B) (2 points) Write assembly code equivalent to that of Figure 3 (i.e., for N=4) using MBR to reduce the number of instructions executed. Note that variable $x$ may be larger than $N-1$. You must handle this case correctly by continuing execution immediately after your code. Also note that A, B, C, and D are conventional procedures, and will return control to the instruction that follows the calling instruction.

(C) (2 points) Extend the single-cycle Beta implementation to implement the MBR instruction. First, specify the datapath muxes where you wish to add an extra input. Second, draw the combinational logic that you want to add to execute MBR. Third, specify the control signals for the MBR instruction.

1. Datapath muxes that need an extra input:

2. Additional logic:
   - Specify inputs and outputs using same names as in the unpipelined Beta diagram.
   - You can use standard combinational logic blocks, such as gates, muxes, adders, shifters, and multipliers.
   - You do not need to reuse existing datapath elements (e.g., the ALU).

3. Control signals for MBR (use don’t cares where appropriate):

<table>
<thead>
<tr>
<th>Instr.</th>
<th>ALUFN</th>
<th>ASEL</th>
<th>BSEL</th>
<th>MOE</th>
<th>MWR</th>
<th>PCSEL</th>
<th>RA2SEL</th>
<th>WASEL</th>
<th>WDSEL</th>
<th>WERF</th>
</tr>
</thead>
<tbody>
<tr>
<td>MBR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
(D) (2 points) Although multiway branches are common, adding instructions to accelerate them is not worth it. The table (or array) \texttt{jtab} below has the locations of methods A, B, C, and D in sequence. Write assembly code equivalent to that of Figure 3 that uses \texttt{jtab} to transfer control to the right procedure. Your implementation should execute a constant number of instructions regardless of the number of choices. (This implementation of a multiway branch is called a \textit{jump} table.)

\begin{verbatim}
jtab:  LONG(A)  
      LONG(B)  
      LONG(C)  
      LONG(D)  
\end{verbatim}

Problem 5. Broken Betas (7 points)

A team of Harvard MBAs recently decided to get rich quick by manufacturing Beta processors. Things have not gone well. They have designed and built several Betas, but each of them has a major design error. They hire you to salvage their processors by modifying the compiler.

In each of the questions below, you are given a Beta with a design flaw and an instruction that is broken as a result of this flaw. Write assembly code that performs the same functionality as the given instruction but whose functioning is not affected by the design flaw. Assume the following:

- You may use multiple instructions for each piece of code.
- You may use registers \texttt{R25} and \texttt{R26} for temporary values (these are guaranteed to not be used anywhere else in the application), but should leave other registers untouched.
- You may use labels, values in memory, and macros, as in normal assembly code.
- You may use other instructions that are affected by the design flaw, but take into account that they will not work correctly.

\textbf{Example:} In this Beta, the \texttt{Z} control signal is the opposite of what it should be, i.e., it is one if and only if the register is nonzero. Write code that performs \texttt{BEQ(R2,target,R28)} correctly.

\textbf{Answer:} \texttt{BNE(R2,target,R28)}
(A) (1 point) In this Beta, the $Z$ control signal is one if and only if the register value is even. Write code that performs $\text{BEQ}(R2, \text{target}, R28)$ correctly.

(B) (2 points) In this Beta, the ALU’s OR unit has been left unconnected, so OR and ORC produce undefined outputs. Write code that performs $\text{OR}(R2, R3, R4)$ correctly.

(C) (2 points) In this Beta, the 16-bit immediate field is zero-extended instead of sign-extended. Write code that performs $\text{ADDC}(R2, \text{literal}, R4)$ correctly.
(D) (2 points) In this Beta, the JMP instruction is unimplemented. Write code that performs `JMP(R1, R2)` correctly, assuming that the program always resides in the top 64KB of memory (so all addresses are 16 bits).

**Hint:** Remember that the Beta is a von Neumann ISA, and instructions are just data. Your code should exploit this by dynamically generating an instruction that is equivalent to JMP, writing it to memory, and executing it.