Problem 1: New DLX instruction jalr.safe is like a jalr instruction except that it automatically returns after a certain number of instructions in the called routine have been executed. Suppose r1 contains 0x1000 and r2 contains 23 when jalr.safe r1, r2 is executed. Execution would jump to address 0x1000 and PC+4 would be saved in r31. Nothing special happens if a jr r31 (a return) is executed within 23 instructions. If after 23 instructions a return is not executed control will automatically return to the instruction following jalr.safe. When jalr.safe is used the called procedure cannot call another procedure.

(a) 

(5 pts) Show how jalr.safe might best be coded.

(b) Modify the pipeline on the next page so that it executes the jalr.safe instruction.

- The output of the Live box is 1 if the corresponding stage contains a non-squashed instruction that will advance to the next stage in the next cycle. (The instruction is neither squashed nor stalled.)
- \[ \text{Ret} \] can be used for detecting return instructions, \[ \text{jalr.safe} \] for jalr.safe instructions, etc.

Continued on next page.
Problem 1, continued:

(7 pts) Show the hardware needed to properly save the automatic return address and count. (For the automatic return address do not use the contents of $r31$, instead add a register for this address.)

(2 pts) Base the return on the number of instructions that will complete, squashed instructions should not be counted.

(2 pts) Make sure `jalr.safe` can be squashed before it sets a return address.

(2 pts) Generate a signal named `Auto Return`, it shall be true when there is to be an automatic return and false other times.

(2 pts) Explain what the controller must do when `Auto Return` is asserted.
Problem 2: The programs below run on statically and dynamically scheduled systems. All systems are single issue (not superscalar), have perfect branch prediction, have an unlimited number of functional units, and use non-blocking caches. The programs run for a large number of iterations, and the first \texttt{lw} in every iteration will miss the cache. On a cache miss data arrives 10 cycles after \texttt{MEM} or \texttt{L2} is entered. The line size is 1024 characters.

\begin{verbatim}
! Program 1
LOOP:
  lw   r1, 0(r2)
  addi r2, r2, #1024
  add   r3, r3, r1
  bneq r1, LOOP

! Program 2
LOOP:
  lw   r1, 0(r2)
  lw   r2, 4(r2)
  add   r3, r3, r1
  bneq r1, LOOP
\end{verbatim}

(a) Suppose the programs were run on a statically scheduled machine and loop unrolling was being considered. \textit{Note: The following important point was not included in the 2000 final exam.} The statically scheduled machine treats load misses like floating-point operations: it allows them to complete out of order if there are no name or data dependencies with following instructions.

\begin{itemize}
  \item [(a)] For a statically scheduled system applying loop unrolling to Program 1 would improve performance: (a) \square by a large amount; (b) \square by a small amount; (c) \square not at all; (d) \square none of the above.
  \item [(b)] For a statically scheduled system applying loop unrolling to Program 2 would improve performance: (a) \square by a large amount; (b) \square by a small amount; (c) \square not at all; (d) \square I do not wish to reveal my intent.
\end{itemize}

\begin{itemize}
  \item [(b)] Suppose the programs were to be run as is (not unrolled).

  \begin{itemize}
    \item [(a)] Compared to a statically scheduled machine, a dynamically scheduled machine would run Program 1: (a) \square much faster; (b) \square slightly faster; (c) \square about the same speed; (d) \square dimple.
    \item [(b)] Compared to a statically scheduled machine, a dynamically scheduled machine would run Program 2: (a) \square much faster; (b) \square slightly faster; (c) \square about the same speed; (d) \square I assume that if my answer below is correct points will not be deducted for this choice.
  \end{itemize}

\begin{itemize}
  \item [(b)] Explain the two answers above. In particular explain, if appropriate, why loop unrolling is more effective on one program than the other.
\end{itemize}
Problem 3: The code below executes on a dynamically scheduled system with branch prediction but without branch target prediction. The branch is predicted taken but it ultimately is not taken. The processor is single-issue (not superscalar) but, conveniently, has an unlimited number of functional units and can handle any number of write-backs per cycle. At most one instruction per cycle can be committed.

(a) The code below executes on such a machine in which the register map is **not** backed up when branches are decoded. Registers are intentionally omitted from the last three instructions, assume those instructions are not data-dependent on the loads.

---

(6 pts) Complete the pipeline execution diagram for this system until all instructions complete.

(2 pts) Show instruction commitment and squashing.

- Don’t forget to check for dependencies!

Branch predicted taken, but branch is **NOT** taken.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw r3, 0(r4)</td>
<td>IF</td>
<td>ID</td>
<td>L1</td>
<td>L2</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>RS</td>
<td>L2</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lw r1, 0(r2)</td>
<td>IF</td>
<td>ID</td>
<td>L1</td>
<td>L2</td>
<td>RS</td>
<td>L2</td>
<td>WB</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

bneq r1, TARGET

xor r5, r6, r7

sgt r8, r9, r10

...  

TARGET:

add r11, r12, r13

sub r14, r15, r16

and r17, r18, r19

...
Problem 3, continued: (b) The code below executes on a version of the machine in which the register map is backed up (checkpointed) when branches are decoded.

\[ \text{(6 pts) Complete the pipeline execution diagram for this system until all instructions complete.} \]

- Show instruction commitment and squashing.

\[ \text{Branch predicted taken, but branch is NOT taken.} \]
\[ \text{Cycle 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16} \]
\[ \text{l} \text{w r}3, 0(r4) \quad \text{IF ID L1 L2 RS RS RS RS L2 WB} \]
\[ \text{l} \text{w r}1, 0(r2) \quad \text{IF ID L1 L2 RS L2 WB} \]
\[ \text{b} \text{neq r}1, \text{TARGET} \]
\[ \text{xor r}5, \text{r}6, \text{r7} \]
\[ \text{sgt r}8, \text{r}9, \text{r10} \]
\[ \ldots \]
\[ \text{TARGET:} \]
\[ \text{add} \]
\[ \text{sub} \]
\[ \text{and} \]

\[ \text{(c) Suppose the processor from the previous part has the following defect: it does not back up or restore the register map, but it executes instructions as though it did. Suppose execution up until the code fragment above is okay.} \]

\[ \text{(6 pts) Add registers to the last three instructions in the previous part so that the xor instruction executes correctly but the sgt (set greater than) executes incorrectly.} \]
Problem 4: A system has a 1-megabyte \(2^{20}\) byte) two-way set-associative cache with 256-character blocks and a 50-bit address space addressing 16-bit characters.

- (7 pts) Fill in the rounded boxes in the diagram below so that it describes this cache.

- (5 pts) Show the smallest set of addresses that cannot all be in this cache at the same time.

- (5 pts) What would be the associativity of a fully associative cache with the same capacity and block size as this one?
Problem 5: Answer each question below.

(a) The DLX program below runs on a system using a one-level branch predictor with a $2^{16}$-entry BHT, each BHT entry is a two-bit saturating counter. The loop iterates many times.

Please do not confuse `andi` with `addi`.

```
LOOP:
  addi r1, r1, #1
  andi r2, r1, #1
  bneq r2, SKIP
  nop
  nop
  nop
  SKIP:
  sub  r3, r1, r4
  bneq r3, LOOP
```

(4 pts) What are the best-case and worst-case prediction accuracies for the first branch. Briefly explain.

(3 pts) What is the smallest BHT size for which there will not be a collision between the two branches?

(b) Consider a dynamically scheduled four-way superscalar processor with a common data bus (CDB) that can handle two write-backs per cycle.

(3 pts) Compare its speed to that of an ordinary dynamically scheduled two-way superscalar processor. Justify your answer.

(3 pts) Compare its speed to that of an ordinary dynamically scheduled four-way superscalar processor. Justify your answer.
(c) (4 pts) Why is branch target prediction potentially more useful for DLX jalr instructions than it is for bneq and beqz instructions?

(d) (4 pts) What is a predicated instruction? Show how predicated instructions can be used in the code below. (Exact syntax is not important.)

```
beqz r1, SKIP
add r2, r2, r3
SKIP:
or r4, r4, r5
```
(e) Consider a statically scheduled DLX implementation in which the floating-point add functional unit is two stages and the floating-point multiply functional unit is four stages, and both are fully pipelined. An exception can occur in any stage of the FP functional units.

(4 pts) Show how the code below would execute to ensure precise exceptions.

```
muld f0, f2, f4
addd f2, f8, f10
or r1, r2, r3
```

(4 pts) Suppose that floating-point instructions did not have precise exceptions. Show how a test instruction could be used to ensure that an exception in `muld` was precise. Illustrate with a pipeline execution diagram.