1. Convert the following machine code instruction to assembly:

```
7 < 2   #8  x1  x2
L DUR
```

```
LDUR  x2  [x1, #8]
```
2. We wish to add the following instruction to our single-cycle CPU:

CBGT Rn, Rd, CondAddr19: A conditional branch. Branches if Rn > Rd. Note: ignore the fact that Rn and CondAddr19 use the same bits in the instruction – we’ll let the programmers sort that out...

This requires no change to our datapath, though your control logic will have access to all of the ALU signals ZERO, NEGATIVE, OVERFLOW, CARRYOUT. Show the control signals needed to implement this instruction.

Reg2Loc: O
ALUSrc: O
MemToReg: X
RegWrite: O
MemWrite: O
BrTaken: Zero ∨ (Negative ∨ Overflow)
UncondBr: O
ALUOp: Subtract 1011
3. Short answer questions on pipelining. These each have very short, simple, direct answers.

(a) In class we put the forwarding muxes towards the end of REG/DECODE, while the textbook puts them towards the beginning of EXECUTE. Why do we use class version instead of the textbook version in lab #4?

forwarding to accelerated branches.

(b) In lab #4 we do branching in the REG/DECODE stage, which forces us to have a branch delay slot. Why don’t we just do the branching logic in the instruction fetch stage?

slow clock cycle: Instruction Memory to branch adder.
4. We have a 2-way set associative cache, 4-byte blocks, 16 bytes total capacity, LRU replacement. For the following sequence of byte address accesses, indicate whether the given access is a HIT or a MISS.

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>HIT or MISS?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Miss</td>
</tr>
<tr>
<td>4</td>
<td>Miss</td>
</tr>
<tr>
<td>8</td>
<td>Miss</td>
</tr>
<tr>
<td>10</td>
<td>Hit</td>
</tr>
<tr>
<td>15</td>
<td>Miss</td>
</tr>
<tr>
<td>16</td>
<td>Miss</td>
</tr>
<tr>
<td>12</td>
<td>Hit</td>
</tr>
<tr>
<td>8</td>
<td>Hit</td>
</tr>
<tr>
<td>1</td>
<td>Miss</td>
</tr>
</tbody>
</table>

\[ a_{0:3} = 5 \]

\[
\begin{array}{c|c|c|c}
4-7 & 8-11 & 12-15 & 45 \\
\end{array}
\]

\[ 3M \times 1H \times 2M \times 2H \times 1/4 \]
**WARNING: read this problem CAREFULLY!**

5. For the following two programs, we are thinking of changing our cache structure. For each of the listed changes, indicate what the change in outcome would be:

- **BETTER**: would have significantly better cache performance.
- **WORSE**: would have significantly worse cache performance.
- **SAME**: would have roughly the same cache performance.

Random

```c
char data[1200];
while (1) {
    int i = random(0..1199);
    data[i]++;
}
```

Sequential

```c
char data[1200];
while (1) {
    for(int i=0; i<1200; i++) {
        data[i]++;
    }
}
```

Starting cache: Direct Mapped cache, 1-byte blocks, 1024 total capacity, 1 cycle access time.

<table>
<thead>
<tr>
<th>Change</th>
<th>Random: Better/Worse/Same</th>
<th>Sequential: Better/Worse/Same</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct Mapped cache</td>
<td>Same</td>
<td>Better</td>
</tr>
<tr>
<td>8 byte blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024 total capacity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 cycle access time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Direct Mapped cache</td>
<td>Better</td>
<td>Better</td>
</tr>
<tr>
<td>1 byte blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2048 total capacity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 cycle access time</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Fully Associative cache</strong></td>
<td>Same</td>
<td>Worse</td>
</tr>
<tr>
<td>1 byte blocks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1024 total capacity</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 cycle access time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6. The following code was written for your lab #3 single-cycle CPU. Please schedule it onto a 2-way VLIW. Only the left issue slot can do loads, only the right issue slot can do branches; both can do ALU operations. There are no delay slots. Instructions issued in the same VLIW word as a branch ALWAYS execute regardless of whether the branch is taken or not.

Please draw the constraint graph (you do NOT have to label the edges), and then schedule the code. Your code should be as fast as possible.

1: LDUR X3, [X4, #8]
2: LDUR X2, [X4, #0]
3: SUBS X3, X5, X2
4: B.LT NO_SWAP
5: STUR X2, [X4, #8]
6: ADDI X4, X4, #8
7: SUBS X3, X1, X0
8: B.LT INNER_LOOP
9: SUBS X0, X0, X5
10: CBZ X0, DONE OUTER

**Constraint Graph:**

1. LDUR 2. LDUR
   3. SUBS
      4. B.LT
         5. STUR
         6. ADDI
         7. SUBS
         8. B.LT
         9. SUBS
      10. CBZ

**Schedule:**

<table>
<thead>
<tr>
<th>ALU/Load/Store</th>
<th>ALU/Branch</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>Loop</td>
</tr>
<tr>
<td>Loop</td>
<td>10</td>
</tr>
</tbody>
</table>