1 Big versus Little Endian Addressing

Consider the 32-bit hexadecimal number 0xcafe2b3a.

1. What is the binary representation of this number in little endian format? Please clearly mark the bytes and number them from low (0) to high (3).

2. What is the binary representation of this number in big endian format? Please clearly mark the bytes and number them from low (0) to high (3).
2 The MIPS ISA

2.1 Warmup: Computing a Fibonacci Number

The Fibonacci number $F_n$ is recursively defined as

$$F(n) = F(n - 1) + F(n - 2),$$

where $F(1) = 1$ and $F(2) = 1$. So, $F(3) = F(2) + F(1) = 1 + 1 = 2$, and so on. Write the MIPS assembly for the `fib(n)` function, which computes the Fibonacci number $F(n)$:

```c
int fib(int n)
{
    int a = 0;
    int b = 1;
    int c = a + b;
    while (n > 1) {
        c = a + b;
        a = b;
        b = c;
        n--;
    }
    return c;
}
```

Remember to follow MIPS calling convention and its register usage (just for your reference, you may not need to use all of these registers):

- The argument `n` is passed in register $4$.
- The result (i.e., `c`) should be returned in $2$.
- $8$ to $15$ are caller-saved temporary registers.
- $16$ to $23$ are callee-saved temporary registers.
- $29$ is the stack pointer register.
- $31$ stores the return address.

Note: A summary of the MIPS ISA is provided at the end of this handout.
2.2 MIPS Assembly for REP MOVSB

MIPS is a simple ISA. Complex ISAs—such as Intel’s x86—often use one instruction to perform the function of many instructions in a simple ISA. Here you will implement the MIPS equivalent for a single Intel x86 instruction, REP MOVSB, which is specified as follows.

The REP MOVSB instruction uses three fixed x86 registers: ECX (count), ESI (source), and EDI (destination). The “repeat” (REP) prefix on the instruction indicates that it will repeat ECX times. Each iteration, it moves one byte from memory at address ESI to memory at address EDI, and then increments both pointers by one. Thus, the instruction copies ECX bytes from address ESI to address EDI.

(a) Write the corresponding assembly code in MIPS ISA that accomplishes the same function as this instruction. You can use any general purpose register. Indicate which MIPS registers you have chosen to correspond to the x86 registers used by REP MOVSB. Try to minimize code size as much as possible.
(b) What is the size of the MIPS assembly code you wrote in (a), in bytes? How does it compare to REP MOVSB in x86 (note: REP MOVSB occupies 2 bytes)?

(c) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

EAX: 0xccccaaaa
EBP: 0x000002222
ECX: 0xFEE1DEAD
EDX: 0xfeed4444
ESI: 0xdecafffe
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000

Now, consider the MIPS assembly code you wrote in (a). How many total instructions will be executed by your code to accomplish the same function as the single REP MOVSB in x86 accomplishes for the given register state?

(d) Assume the contents of the x86 register file are as follows before the execution of the REP MOVSB:

EAX: 0xccccaaaa
EBP: 0x000002222
ECX: 0x00000000
EDX: 0xfeed4444
ESI: 0xdecafffe
EDI: 0xdeaddeed
EBP: 0xe0000000
ESP: 0xe0000000
Now, answer the same question in (c) for the above register values.
3 Data Flow Programs

Draw the data flow graph for the $\text{fib}(n)$ function from Question 2.1. You may use the following data flow nodes in your graph:

- $+$ (addition)
- $>$ (left operand is greater than right operand)
- Copy (copy the value on the input to both outputs)
- BR (branch, with the semantics discussed in class, label the True and False outputs)

You can use constant inputs (e.g., 1) that feed into the nodes. Clearly label all the nodes, program inputs, and program outputs. Try to use the fewest number of data flow nodes possible.
4 Microarchitecture vs. ISA

a) Briefly explain the difference between the microarchitecture level and the ISA level in the transformation hierarchy. What information does the compiler need to know about the microarchitecture of the machine in order to compile a given program correctly?

b) Classify the following attributes of a machine as either a property of its microarchitecture or ISA:

<table>
<thead>
<tr>
<th>Microarchitecture?</th>
<th>ISA?</th>
<th>Attribute</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>The machine does not have a subtract instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The ALU of the machine does not have a subtract unit</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The machine does not have condition codes</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A 5-bit immediate can be specified in an ADD instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>It takes n cycles to execute an ADD instruction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>There are 8 general purpose registers</td>
</tr>
<tr>
<td></td>
<td></td>
<td>A 2-to-1 mux feeds one of the inputs to ALU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The register file has one input port and two output ports</td>
</tr>
</tbody>
</table>
5 Performance Metrics

• If a given program runs on a processor with a higher frequency, does it imply that the processor always executes more instructions per second (compared to a processor with a lower frequency)? (Use less than 10 words.)

• If a processor executes more of a given program’s instructions per second, does it imply that the processor always finishes the program faster (compared to a processor that executes fewer instructions per second)? (Use less than 10 words.)
6 Performance Evaluation

Your job is to evaluate the potential performance of two processors, each implementing a different ISA. The evaluation is based on its performance on a particular benchmark. On the processor implementing ISA A, the best compiled code for this benchmark performs at the rate of 10 IPC. That processor has a 500 MHz clock. On the processor implementing ISA B, the best compiled code for this benchmark performs at the rate of 2 IPC. That processor has a 600 MHz clock.

• What is the performance in Millions of Instructions per Second (MIPS) of the processor implementing ISA A?

• What is the performance in MIPS of the processor implementing ISA B?

• Which is the higher performance processor: A B Don’t know

Briefly explain your answer.
7 Single-Cycle Processor Datapath

In this problem, you will modify the single-cycle datapath we built up in Lecture 11 to support the JAL instruction. The datapath that we will start with is provided below. Your job is to implement the necessary data and control signals to support the JAL instruction, which we define to have the following semantics:

\[
\text{JAL} : \quad R31 \leftarrow PC + 4 \\
PC \leftarrow PC_{31...28} \ || \ \text{Immediate} \ || \ 0^2
\]

Add to the datapath on the next page the necessary data and control signals to implement the JAL instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).
8 LC-3b Microprogramming

As we learned in class, the LC-3b is a microprogrammed design, which means that control signals throughout the LC-3b datapath (Patt+Patel, Appendix C, Figure 3) determine what the processor does during any given cycle. The control signals are stored in a memory called the control store, and they must be carefully chosen to correctly implement the LC-3b state machine (Patt+Patel, Appendix C, Figure 2). The contents of the control store can be easily described using a 2D spreadsheet with rows representing different LC-3b states and columns specifying all 35 control signal bits required for each state.

Unfortunately, a clumsy TA has accidentally erased the contents of the control store! Your task is to recover this data manually (i.e., determine every bit in the control store) using your knowledge of the LC-3b architecture.

Fill out the microcode in the microcode.csv file handed out with this homework. Enter a 1 or a 0 or an X as appropriate for the microinstructions corresponding to each state. We have filled out state 18 as an example. Hint: all of the information can be inferred using the information and figures provided in Patt+Patel Appendix C.
Let’s say you are the lead architect of the next flagship processor at Advanced Number Devices (AND). You have decided that you want to use the LC-3b ISA for your next product, but your customers want a smaller semantic gap and marketing is on your case about it. So, you have decided to implement your favorite x86 instruction, REP MOVSB, in LC-3b.

Specifically, you want to implement the following definition for REP MOVSB (in LC-3b parlance): REP-MOVSB SR1, SR2, DR which is encoded in LC-3b machine code as:

```
15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
1010  DR  SR1 0 0 0  SR2
```

REPMOVSB uses three registers: SR1 (count), SR2 (source), and DR (destination). It moves a byte from memory at address SR2 to memory at address DR, and then increments both pointers by one. This is repeated SR1 times. Thus, the instruction copies SR1 bytes from address SR2 to address DR. Assume that the value in SR1 is greater than or equal to zero.

1. Complete the state diagram shown below, using the notation of the LC-3b state diagram. Describe inside each bubble what happens in each state and assign each state an appropriate state number. Add additional states not present in the original LC-3b design as you see fit.
2. Add to the LC-3b datapath any additional structures and any additional control signals needed to implement REPMOVSB. Clearly label your additional control signals with descriptive names. Describe what value each control signal would take to control the datapath in a particular way.
3. Describe any changes you need to make to the LC-3b microsequencer. Add any additional logic and control signals you need. Clearly describe the purpose and function of each signal and the values it would take to control the microsequencer in a particular way.
10 Mystery LC-3b Instruction (I)

A pesky engineer has implemented a mystery instruction on the LC-3b! It is your job to determine what the instruction does. The mystery instruction is encoded as:

```
  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
  1010   DR   SR1  0  0  0  0  0  0  0  0  0  0  0  0  0
```

The modifications we make to the LC-3b datapath and the microsequencer are highlighted in the attached figures (see the next two pages). We also provide the original LC-3b state diagram, in case you need it. (As a reminder, the selection logic for SR2MUX is determined internally based on the instruction.)

The additional control signals are:

- **GateTEMP1/1**: NO, YES
- **GateTEMP2/1**: NO, YES
- **LD.TEMP1/1**: NO, LOAD
- **LD.TEMP2/1**: NO, LOAD

**ALUK/3**: OR1 (A\(\mid 0x1\)), LSHF1 (A\(<1\)), PASSA, PASS0 (Pass value 0), PASS16 (Pass value 16)

**COND/4**:
- \(\text{COND}_{0000}\): Unconditional
- \(\text{COND}_{0001}\): Memory Ready
- \(\text{COND}_{0010}\): Branch
- \(\text{COND}_{0011}\): Addressing mode
- \(\text{COND}_{0100}\): Mystery 1
- \(\text{COND}_{1000}\): Mystery 2

The microcode for the instruction is given in the table below.

<table>
<thead>
<tr>
<th>State</th>
<th>Cond</th>
<th>J</th>
<th>Asserted Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>001010  (10)</td>
<td>COND_{0000}</td>
<td>001011</td>
<td>ALUK = PASS0, GateALU, LD.REG, DRMUX = DR (IR[11:9])</td>
</tr>
<tr>
<td>001011  (11)</td>
<td>COND_{0000}</td>
<td>101000</td>
<td>ALUK = PASSA, GateALU, LD.TEMP1, SR1MUX = SR1 (IR[8:6])</td>
</tr>
<tr>
<td>101000  (40)</td>
<td>COND_{0000}</td>
<td>110010</td>
<td>ALUK = PASS16, GateALU, LD.TEMP2</td>
</tr>
<tr>
<td>110010  (50)</td>
<td>COND_{1000}</td>
<td>101101</td>
<td>ALUK = LSHF1, GateALU, LD.REG, SR1MUX = DR, DRMUX = DR (IR[11:9])</td>
</tr>
<tr>
<td>111101  (61)</td>
<td>COND_{0000}</td>
<td>101101</td>
<td>ALUK = OR1, GateALU, LD.REG, SR1MUX = DR, DRMUX = DR (IR[11:9])</td>
</tr>
<tr>
<td>101101  (45)</td>
<td>COND_{0000}</td>
<td>111111</td>
<td>GateTEMP1, LD.TEMP1</td>
</tr>
<tr>
<td>111111  (63)</td>
<td>COND_{0100}</td>
<td>010010</td>
<td>GateTEMP2, LD.TEMP2</td>
</tr>
</tbody>
</table>

Describe what this instruction does.
Code:

State 10: DR ← 0
State 11: TEMP1 ← value(SR1)
State 40: TEMP2 ← 16
State 50: DR = DR << 1
   if (TEMP1[0] == 0)
      goto State 45
   else
      goto State 61
State 61: DR = DR | 0x1
State 45: TEMP1 = TEMP1 >> 1
State 63: DEC TEMP2
   if (TEMP2 == 0)
      goto State 18
   else
      goto State 50
LC-3b to operate correctly with a memory that takes multiple clock cycles to read or store a value. Suppose it takes memory five cycles to read a value. That is, once MAR contains the address to be read and the microinstruction asserts READ, it will take five cycles before the contents of the specified location in memory are available to be loaded into MDR. (Note that the microinstruction asserts READ by means of three control signals: MIO.EN/YES, R.W/RD, and DATA.SIZE/WORD; see Figure C.3.)

Recall our discussion in Section C.2 of the function of state 33, which accesses an instruction from memory during the fetch phase of each instruction cycle. For the LC-3b to operate correctly, state 33 must execute five times before moving on to state 35. That is, until MDR contains valid data from the memory locations specified by the contents of MAR, we want state 33 to continue to re-execute. After five clock cycles, the contents of MDR will be available.

Figure C.6: Additional logic required to provide control signals
Figure C.2: A state machine for the LC-3b.
11 Mystery LC-3b Instruction (II)

An engineer implemented the mystery instruction described below on the LC-3b. Unfortunately, we do not know what this engineer was thinking, and we can’t figure out what the instruction does. It is your job to determine this. The mystery instruction is encoded as:

<table>
<thead>
<tr>
<th></th>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1010</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The instruction is only defined if the value of SR1 is not equal to zero.

The modifications we make to the LC-3b datapath and the microsequencer are highlighted in the attached figures (see the next three pages).

We also provide the original LC-3b state diagram, in case you need it.

The additional control signals are:

**GateLSHF/1**: NO, YES

**LD.TEMP/1**: NO, LOAD

**ALUK/3**: AND, ADD, XOR, PASSA, PASSB, DECA (Decrement A)

**COND/3**:
- **COND0**: Unconditional
- **COND1**: Memory Ready
- **COND2**: Branch
- **COND3**: Addressing mode
- **COND4**: Mystery

The microcode for the instruction is given in the table below.

<table>
<thead>
<tr>
<th>State</th>
<th>Cond</th>
<th>J</th>
<th>Asserted Signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>001010  (10)</td>
<td>COND0</td>
<td>101000</td>
<td>ALUK = PASSB, GateALU, LD.REG, DRMUX = DR (IR[11:9])</td>
</tr>
<tr>
<td>101000  (40)</td>
<td>COND4</td>
<td>010010</td>
<td>ALUK = PASSA, GateALU, LD.TEMP, SR1MUX = SR1 (IR[8:6])</td>
</tr>
<tr>
<td>110010  (50)</td>
<td>COND0</td>
<td>110011</td>
<td>ALUK = DECA, GateALU, LD.REG, SR1MUX = DR, DRMUX = DR (IR[11:9])</td>
</tr>
<tr>
<td>110011  (51)</td>
<td>COND4</td>
<td>010010</td>
<td>GateLSHF, LD.TEMP</td>
</tr>
</tbody>
</table>

Describe what this instruction does.
Code:

State 10: DR = 15
State 40: TEMP = SR1
    if (SR1 is negative)
        Go to State 18/19 (Fetch)
    else
        Go to State 50
State 50: DR = DR - 1
State 51: Left Shift TEMP
    if (TEMP is negative)
        Go to State 18/19 (Fetch)
    else
        Go to State 50
## MIPS Instruction Summary

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Example Assembly</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>add $1, $2, $3</td>
<td>$1 = $2 + $3</td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>sub $1, $2, $3</td>
<td>$1 = $2 - $3</td>
</tr>
<tr>
<td><strong>add immediate</strong></td>
<td>addi $1, $2, 100</td>
<td>$1 = $2 + 100</td>
</tr>
<tr>
<td><strong>add unsigned</strong></td>
<td>addu $1, $2, $3</td>
<td>$1 = $2 + $3</td>
</tr>
<tr>
<td><strong>subtract unsigned</strong></td>
<td>subu $1, $2, $3</td>
<td>$1 = $2 - $3</td>
</tr>
<tr>
<td><strong>add immediate unsigned</strong></td>
<td>addiu $1, $2, 100</td>
<td>$1 = $2 + 100</td>
</tr>
<tr>
<td><strong>multiply</strong></td>
<td>mult $2, $3</td>
<td>hi, lo = $2 * $3</td>
</tr>
<tr>
<td><strong>multiply unsigned</strong></td>
<td>multu $2, $3</td>
<td>hi, lo = $2 * $3</td>
</tr>
<tr>
<td><strong>divide</strong></td>
<td>div $2, $3</td>
<td>lo = $2/$3, hi = $2 mod $3</td>
</tr>
<tr>
<td><strong>divide unsigned</strong></td>
<td>divu $2, $3</td>
<td>lo = $2/$3, hi = $2 mod $3</td>
</tr>
<tr>
<td><strong>move from hi</strong></td>
<td>mfhi $1</td>
<td>$1 = hi</td>
</tr>
<tr>
<td><strong>move from low</strong></td>
<td>mflo $1</td>
<td>$1 = lo</td>
</tr>
<tr>
<td><strong>and</strong></td>
<td>and $1, $2, $3</td>
<td>$1 = $2 &amp; $3</td>
</tr>
<tr>
<td><strong>or</strong></td>
<td>or $1, $2, $3</td>
<td>$1 = $2</td>
</tr>
<tr>
<td><strong>and immediate</strong></td>
<td>andi $1, $2, 100</td>
<td>$1 = $2 &amp; 100</td>
</tr>
<tr>
<td><strong>or immediate</strong></td>
<td>ori $1, $2, 100</td>
<td>$1 = $2</td>
</tr>
<tr>
<td><strong>shift left logical</strong></td>
<td>sll $1, $2, 10</td>
<td>$1 = $2 &lt;&lt; 10</td>
</tr>
<tr>
<td><strong>shift right logical</strong></td>
<td>srl $1, $2, 10</td>
<td>$1 = $2 &gt;&gt; 10</td>
</tr>
<tr>
<td><strong>load word</strong></td>
<td>lw $1, 100($2)</td>
<td>$1 = memory[$2 + 100]</td>
</tr>
<tr>
<td><strong>store word</strong></td>
<td>sw $1, 100($2)</td>
<td>memory[$2 + 100] = $1</td>
</tr>
<tr>
<td><strong>load upper immediate</strong></td>
<td>lui $1, 100</td>
<td>$1 = 100 &lt;&lt; 16</td>
</tr>
<tr>
<td><strong>branch on equal</strong></td>
<td>beq $1, $2, label</td>
<td>if ($1 == $2) goto label</td>
</tr>
<tr>
<td><strong>branch on not equal</strong></td>
<td>bne $1, $2, label</td>
<td>if ($1 != $2) goto label</td>
</tr>
<tr>
<td><strong>set on less than</strong></td>
<td>slt $1, $2, $3</td>
<td>if ($2 &lt; $3) $1 = 1 else $1 = 0</td>
</tr>
<tr>
<td><strong>set on less than immediate</strong></td>
<td>slti $1, $2, 100</td>
<td>if ($2 &lt; 100) $1 = 1 else $1 = 0</td>
</tr>
<tr>
<td><strong>set on less than unsigned</strong></td>
<td>sltu $1, $2, $3</td>
<td>if ($2 &lt; $3) $1 = 1 else $1 = 0</td>
</tr>
<tr>
<td><strong>set on less than immediate</strong></td>
<td>sltu $1, $2, 100</td>
<td>if ($2 &lt; 100) $1 = 1 else $1 = 0</td>
</tr>
<tr>
<td><strong>jump</strong></td>
<td>j label</td>
<td>goto label</td>
</tr>
<tr>
<td><strong>jump register</strong></td>
<td>jr $31</td>
<td>goto $31</td>
</tr>
<tr>
<td><strong>jump and link</strong></td>
<td>jal label</td>
<td>$31 = PC + 4; goto label</td>
</tr>
</tbody>
</table>