ECE 3056: Architecture, Concurrency and Energy of Computation

Single and Multi-Cycle Datapaths: Practice Problems and Solutions
1. Consider the single cycle SPIM datapath.
   a. Specify the values of the control signals for each of the following instructions. Assume all registers are initialized to 0x44.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUSrc</th>
<th>MemRead</th>
<th>RegDst</th>
<th>RegWrite</th>
<th>ALU Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi $8, $9, -4</td>
<td>1</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>0x40</td>
</tr>
<tr>
<td>lw $t2, 8($at)</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0x4c</td>
</tr>
</tbody>
</table>

b. Suppose that due to device wear out, the RegDst signal becomes permanently stuck at the value 1. What would be the impact of the execution of the addi instruction in part (a).

The destination register address will be taken from bits [15-11] of the addi instruction. These bits are the upper 5 bits of the immediate value, which is 0xfffc. This will generate a register address of 0x1f, which is register $31. The value 0x40 will be written to register $31.
2. For the following questions, consider the single cycle datapath. The value of loop is 0x00400004, the contents of register $t0, $t1, and $t2 are 0x01000033, 0x00004008, and 0x000000c0, respectively, and the contents of register $1 is 0x10010008

a. Fill in the values of control signals in the following table for each instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUSrc</th>
<th>ALUOp</th>
<th>RegDst</th>
<th>MemToReg</th>
<th>Write Data Input to Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>j loop</td>
<td>X</td>
<td>XX</td>
<td>X</td>
<td>X</td>
<td>contents of $16</td>
</tr>
<tr>
<td>sw $t2, 8($at)</td>
<td>1</td>
<td>00</td>
<td>X</td>
<td>X</td>
<td>0x000000c0</td>
</tr>
</tbody>
</table>

The jump instruction does not utilize any portion of the datapath other than the portion that updates the PC with the jump address. However, the register file does utilize bits of the encoded address as register addresses for rs and rt. The input to the write data port of memory is the contents of register rt. Therefore first encode the j instruction to find the value of rt. It will be 16.

b. Errors in fabrication are a major concern in the design of a datapath. Suppose that MemToReg and RegDst multiplexor control signals were stuck at 0, i.e., you cannot assert these signals due to a fabrication defect. Which of the instructions supported by the datapath will still work?

**sw, beq, and j.** This follows from the truth table for the controller. Which rows can still produce valid control signal values?

c. This question deals with the changes to the datapath required to implement the addition of the jal instruction. Full credit requires you to be specific.
- what are the structural changes, if any, that must be made to the datapath

- We need $31 as a hardwired input address to the RegDst Mux. RegDst is now a 2-bit control signal
- We need PC+4 as an input to the MemToReg Mux since it is now a candidate for writing into the register file (into $31). The MemToReg signal is now a 2-bit control signal
- The jal control signal is used to select the jump address from the PC source mux. This mux is expanded to a 2-bit signal.
Show the changes required to the main controller truth table and the PLA implementation of this controller.

<table>
<thead>
<tr>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemToReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Br</th>
<th>AluOp</th>
<th>Jal</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>X</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>XX</td>
<td>1</td>
</tr>
</tbody>
</table>

The PLA implementation will be changed to add another gate to recognize the `jal` instruction. The `RegDst` and `MemToReg` outputs must be modified to produce two output signals. The most significant bit is simply the `jal` while the least significant remains as before. We also introduce a `jal` output.
3. Consider the state of following procedure executing within the single cycle SPIM data path. The first instruction is at address 0x00400440 and registers $4 and $5 contain the values 0x10010440 and 0x00000040 respectively. Assume that all instructions are native instructions.

```plaintext
func:
  add $2, $4, $0
  add $3, $5, $0
  add $3, $3, $2
  add $9, $0, $0
  add $22, $0, $0

loop:
  lw $22, 0($2)
  add $9, $9, $22
  addi $2, $2, 4
  beq $3, $2, end
  j loop

end:
  move $2, $9
  jr $31
```

a. Fill in the values of the signals shown for the following instructions (first time through the loop).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUOutput</th>
<th>Value at the Write Data Input of Data Memory</th>
<th>RegDst</th>
<th>ALUOp</th>
<th>ALUSrc</th>
<th>MemToReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>beq $3, $2, end</td>
<td>0x0000003c</td>
<td>0x100100444</td>
<td>X</td>
<td>01</td>
<td>0</td>
<td>X</td>
</tr>
<tr>
<td>lw $22, 0($2)</td>
<td>0x10010440</td>
<td>0x00000000</td>
<td>0</td>
<td>00</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

b. Suppose in our design the controller was implemented in a separate chip. Due to a fabrication defect the MemToReg control signal has been found to be defective. What modifications can we make to any part of the datapath (excluding the controller) such that the datapath still functions correctly.

Invert the RegDst signal and use it as MemToReg
c. Now suppose the above program was executed on a multi-cycle datapath. Assume that the first cycle for fetching the first instruction is numbered cycle 0. Assume immediate instructions take the same number of cycles as the R-format instructions. On which cycle do the following events take place.

- add $9, $0, $0 completes execution

  15__ (the end of the instruction execution not the EX cycle)

- the branch address for \textbf{beq} $2, $3, \text{end} is computed (for the first time through the loop)

  34__

d. What is the principal advantage of the multi-cycle datapath over the single cycle datapath. Be specific and provide an example to illustrate your point.

Each instruction only takes as long as it needs to. For example, if the clock cycle was 10 ns and the load takes 5 cycles (50 ns) branches now only take 30 ns (3 cycles and R-format instructions only take 40 ns (4 cycles). The net execution time savings over the execution of a program can be substantial.
4. For the following questions, consider the single cycle datapath. The value of `loop` is 0x00400004, the contents of register $t0$, $t1$, and $t2$ are 0x01000033, 0x00004008, and 0x000000c0, respectively, and the contents of register $t$ is 0x10010008

a. Fill in the values of control signals in the following table for each instruction.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUSrc</th>
<th>ALUOp</th>
<th>RegDst</th>
<th>MemToReg</th>
<th>Write Data Input to Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>jal loop</td>
<td>x</td>
<td>x</td>
<td>10</td>
<td>10</td>
<td>contents of $t6$</td>
</tr>
<tr>
<td>bne $t1, $t2$, loop</td>
<td>0</td>
<td>01</td>
<td>X</td>
<td>X</td>
<td>0x000000c0</td>
</tr>
</tbody>
</table>

The `jal` instruction requires some additions to the datapath.
- We need $t31$ as a hardwired input address to the `RegDst` Mux. `RegDst` is now a 2-bit control signal.
- We need PC+4 as an input to the `MemToReg` Mux since it is now a candidate for writing into the register file (into $t31$). The `MemToReg` signal is now a 2-bit control signal.

However, the register file does utilize bits of the encoded instruction as register addresses for `rs` and `rt`. The input to the write data port of memory is the contents of register `rt`. Therefore encode the `jal` instruction to find the value of `rt`. It will be 16.

b. You are designer and wish to reduce the number of control signals in this datapath. Provide one hardware modification that will eliminate the `MemToReg` control signal.

Use the complement of `RegDst` as the value of this signal. This is not the only solution but one which is readily apparent from the truth table.

c. This question deals with the changes to the datapath required to implement the addition of the `addi` instruction. Full credit requires you to be specific.

- what are the structural changes, if any, that must be made to the datapath

None are required. We have the connections necessary to add an immediate operand to the contents of a register. The issue is simply one of control.
- Show the changes required to the main controller truth table and the PLA implementation of this controller.

<table>
<thead>
<tr>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemToReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Br</th>
<th>AluOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

The values of `AluSrc` and `RegWrite` become modified to include another OR term. Note the change in these two columns on the truth table. Also we have an additional gate on the input to recognize the `addi` instruction.
5. For the following questions, consider the single cycle datapath shown in the text. The value of loop is 0x00400004, the contents of register $t1, $t2, and $t3 are 0x01000033, 0x00004008, and 0x000000c0, respectively, and the contents of register $1 is 0x10010008

   a. Fill in the values of control signals in the following table for each instruction.

   
<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUSrc</th>
<th>ALUOp</th>
<th>RegDst</th>
<th>MemToReg</th>
<th>Write Data Input to Data Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $t3, 4($t2)</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0x000000c0</td>
</tr>
</tbody>
</table>

   b. You are designer and wish to reduce the number of control signals that the controller has to generate. Provide one hardware modification that will eliminate the RegDst control signal (but not the mux).

   Note that the RegDst signal is the complement of the MemToReg signal. This the RegDst mux can be controlled by the MemToReg signal passed through an inverter.
6. Consider the single cycle SPIM datapath. You wish to add a new instruction - dcbr $t0, loop – decrement-and-branch-on-zero. This instruction will first decrement the register $t0 by 1 and then branch to the label loop if the result is 0. Register $t0 is updated with the decremented value.

   a. Clearly show the hardware modifications on the datapath. [To receive any credit your modifications must be legible]
   You can use the implementation of the beq which already performs a subtract operation. Add the value -1 as an input to the ALUSrc mux (now it is a two bit control). Write the result of the ALU operation (which is the decrement) back to the rs register – expand the RegDst mux to include rs as an input while RegDst becomes a 2-bit control signal.

   b. (10 pts) Fill in the truth table below for the single cycle data path including any changes to accommodate the new instruction

<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegDst</th>
<th>AluSrc</th>
<th>MemToReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>AluOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-form</td>
<td>01</td>
<td>00</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>lw</td>
<td>00</td>
<td>01</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>01</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>00</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>DCB</td>
<td>10</td>
<td>10</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>01</td>
</tr>
</tbody>
</table>

   c. (5 pts) Assuming the opcode for the new instruction is 100111, show the modifications required to the controller hardware below for the new instruction. Make and state any assumptions you need.
7. Consider the execution of the following block of SPIM code on a multi-cycle datapath. The text segment starts at 0x00400000 and the data segment starts at 0x10010000. Assume immediate instructions take 4 cycles.

```
.data
start: .word 21, 22, 23, 24
str: .asciiz “CmpE”
.align 4
.word 24, 0x77
.text
.globl main
main:
    li $t3, 4
    lui $t0, 0x1001
    lui $t1, 0x1002
move:
    lw $t5, 0($t0)
    sw $t5, 0($t1)
    addiu $t0, $t0, 4
    addiu $t1, $t1, 4
    addi $t3, $t3, -1
end:
    bne $t3, $zero, move
done:
```

Fill in the following values at the end of the requested cycle. Assume that the first cycle of program execution is cycle number 0!

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALUSrcA</th>
<th>ALUOp</th>
<th>RegWrite</th>
<th>Instruction Register</th>
<th>IorD</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0x8d0d0000</td>
<td>X</td>
</tr>
<tr>
<td>35</td>
<td>1</td>
<td>01</td>
<td>0</td>
<td>0x1560ffff</td>
<td>X</td>
</tr>
</tbody>
</table>
8. Consider the execution of the following block of SPIM code on a multi-cycle datapath. The text segment starts at 0x00400000 and the data segment starts at 0x10010000. Assume immediate instructions take 4 cycles.

```
.data
  start: .word 21, 22, 23, 24
  str: .asciiz “CmpE”
 .align 4
  .word 24, 0x77

.text
.globl main
main:
  li $t3, 4
  lui $t0, 0x1001
  lui $t1, 0x1002
move:
  lw $t5, 0($t0)
  sw $t5, 0($t1)
  addiu $t0, $t0, 4
  addiu $t1, $t1, 4
  add $t3, $t3, -1
end:
  bne $t3, $zero, move
done:
```

Fill in the following values at the end of the requested cycle. Assume that the first cycle of program execution is cycle number 0!

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALUSrcB</th>
<th>ALUOp</th>
<th>RegDst</th>
<th>Instruction Register</th>
<th>PCWriteCond</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>11</td>
<td>00</td>
<td>X</td>
<td>0x3c091002</td>
<td>0</td>
</tr>
<tr>
<td>20</td>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0xad2d0000</td>
<td>0</td>
</tr>
</tbody>
</table>
9. Consider the execution of the following block of SPIM code on a multi-cycle datapath. Assume that the text segment starts at 0x00400000 and that the data segment starts at 0x10010000. Assume that registers $7, $8 and $9 have initial values 16, 0x10010020, and 0x10020020 respectively.

```
.data
  str: .asciiz “Start”
  .align 2
  .word 24, 0x6666
.text
move:
  lw $12, 0($8)
  sw $12, 0($9)
  addiu $8, $8, 4
  addiu $9, $9, 4
  addi $7, $7, -1
end:
  bne $7, $0, move
```

Fill in the following values at the end of the requested cycle. Assume that the first cycle of program execution is cycle number 1.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALUSelA</th>
<th>ALUOp</th>
<th>PCWrite</th>
<th>IR (in HEX!)</th>
<th>ALUOutput</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0xad2c0000</td>
<td>0x10020020</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>00</td>
<td>0</td>
<td>0x20e7ffff</td>
<td>0xf</td>
</tr>
</tbody>
</table>
10. Consider the execution of the following block of SPIM code on a multi-cycle datapath. Assume that the text segment starts at 0x00400000 and that the data segment starts at 0x10010000. Assume that registers $7, $8 and $9 have initial values 16, 0x10010020, and 0x10020020 respectively.

```spim
.data
str: .asciiz “Exams”
.align 4
.word 24, 0x6666
.text
move: lw $12, 0($8)
       sw $12, 0($9)
       addiu $8, $8, 4
       addiu $9, $9, 4
       addi $7, $7, -1
end:  bne $7, $0, move
```

Fill in the following values at the end of the requested cycle. Assume that the first cycle of program execution is cycle number 1.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALUSelB</th>
<th>ALUOp</th>
<th>RegDst</th>
<th>IR (in HEX!)</th>
<th>MemToReg</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>11</td>
<td>00</td>
<td>X</td>
<td>0xad2c0000</td>
<td>X</td>
</tr>
<tr>
<td>17</td>
<td>XX</td>
<td>XX</td>
<td>0</td>
<td>0x25290004</td>
<td>0</td>
</tr>
</tbody>
</table>
11. Consider the execution of the following block of SPIM code on a multi-cycle datapath. Assume that the fetch cycle for the first instruction is cycle number 0. Assume that \texttt{addiu} and \texttt{addi} require the same number of states as an add instruction.

\begin{verbatim}
add $t4, $0, $0
addi $t3, $0, 4
loop:  lw $t0, 0($t1)
    add $t4, $t4, $t0
    addiu $t1, $t1, 4
    addi $t3, $t3, -1
    bne $t3, $0, loop
\end{verbatim}

\textbf{a.} How many cycles does this code sequence take to execute to completion?

88 (the loop takes 20 cycles and executes 4 times).

\textbf{b.} On what cycle is the add instruction in the body of the loop executed for the second time (remember the first cycle is cycle number 0)?

Cycle no. 35 for the EX cycle of the add instruction.

\textbf{c.} Fill in the following values during the requested cycle (remember the first cycle is cycle number 0!).

\begin{tabular}{|c|c|c|c|c|c|}
\hline
Cycle & ALUSrcB & ALUOp & RegDst & PCWrite & MemToReg & PCSource \\
\hline
13 & 01 & 00 & 0 & 1 & 0 & 00 \\
\hline
27 & 00 & 01 & 0 & 0 & 0 & 01 \\
\hline
\end{tabular}

The first row corresponds to the fetch cycle of the add instruction. The second row corresponds to third cycle of the branch instruction.
12. Consider the execution of the following block of SPIM code on a multi-cycle datapath. Assume that the fetch cycle for the first instruction is cycle number 0. Assume that \texttt{li} and \texttt{addi} require the same number of states as an \texttt{add} instruction, that \texttt{jal} and \texttt{slt} require the same number of states as a \texttt{bne} instruction.

\begin{verbatim}
.main:    li $t0, 2
        add $t2, $zero, $zero
        li $a0, 1
.mode:
.loop:   jal solo
        addi $t0, $t0, -1
        addi $a0, $a0, 1
        add $t2, $t2, $v0
        slt $t1, $t0, $zero
        bne $t1, $zero, loop
\end{verbatim}

a. How many cycles does this code sequence take to execute to completion?

33

b. On what cycle is the \texttt{slt} instruction in the body of the loop fetched for the first time (remember the first cycle is cycle number 0)?

27

c. Fill in the following values during the requested cycle (remember the first cycle is cycle number 0!).

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALUSrcB</th>
<th>IorD</th>
<th>PCWrite</th>
<th>RegDst</th>
<th>MemToReg</th>
<th>PCSource</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>26</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

The first row corresponds to a decode cycle. The second row corresponds to a write back cycle.
13. Consider a base, multi-cycle SPIM machine operating at 200 Mhz, with the following program execution characteristics.

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Type + Immediate</td>
<td>4</td>
<td>45%</td>
</tr>
<tr>
<td>lw</td>
<td>5</td>
<td>25%</td>
</tr>
<tr>
<td>sw</td>
<td>4</td>
<td>20%</td>
</tr>
<tr>
<td>beq</td>
<td>3</td>
<td>7%</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>3%</td>
</tr>
</tbody>
</table>

Each of the following parts are independent of each other.

a. The designers decided to add a new addressing mode to help with array addressing where we often add a byte offset to a base address stored in a register. Therefore a sequence of instructions such as code block 1 below can be replaced by the instruction shown in code block 2. We have 40% of the lw instructions affected in this manner. Assuming that the cycle time does not change and the new instruction also takes 5 cycles, what is the speedup produced by the use of this new addressing mode?

```
add $t2, $t1, $t0
lw $t3, 0($t2)
becomes
lwx $t2, $t1+$t0
lw $t3, 0($t2)
```

code block 1 code block 2

\[
\text{CPI}_{\text{old}} = 0.45 \times 4 + 0.25 \times 5 + 0.20 \times 4 + 0.07 \times 3 + 0.03 \times 3 = 4.15 \\
\text{Ex}_{\text{old}} = \#\text{Instr} \times 4.15 \times \text{clock\_cycle\_time}
\]

With the optimization the number of instructions has been changed. If 40% of the lw instructions have been affected that means 0.4 \times 0.25 = 0.1 or 10% of the total number of instructions have been affected. For each of these instructions we can eliminate one instruction by combining two instructions. The new total number of instructions is therefore (0.9 \times \#\text{Instr}). The eliminated instructions are all R-format instructions. Therefore the percentage of such instructions are reduced by 10%.

\[
\text{CPI}_{\text{new}} = 0.35 \times 4 + 0.25 \times 5 + 0.2 \times 4 + 0.07 \times 3 + 0.03 \times 3)/0.9 = 4.17 \\
\text{Ex}_{\text{new}} = (0.9 \times \#\text{Instr}) \times 4.17 \times \text{clock\_cycle\_time}
\]

Comparing the execution times we can see that this is a good trade-off.
b. Is it possible to halve the execution time of programs by simply speeding up R-format and immediate instructions, i.e., realize a speedup of 2? Justify your answer.

From Amdahl’s Law we have Speedup = $1/(s + ((1-s)/n))$ where $s$ is the fraction that is unaffected, and $n$ is the factor by which this component can be sped up. Let us assume that R-format and immediate instructions are infinitely sped up, i.e., $n = \infty$. In this case we have

$$\text{Speedup} = 1/(0.55) < 2.$$  
Therefore the answer is no.

c. What is the maximum MIPS rating of the multi-cycle datapath?

The fastest instruction takes 3 cycles, i.e., has a CPI of 3. This is the theoretically lowest CPI possible for any program (although it is unrealistic to have a program that only branches opr jumps!)

MIPS Rating = (Clock Rate in MHZ)/CPI = 200/3 = 66.7 MIPS
14. We wish to make our datapath implement instructions such as those that are found in the Power PC. Specifically we are interested in adding an instruction that will increment a counter, and branch to a label if the counter value is 0: \texttt{icb $t0, loop}. This instruction operates just as a branch instruction would if the branch is taken. Assume that this instruction uses the same format as a branch instruction (with the \texttt{rt} field set to 0). Assume that the opcode for this new instruction is 101000. Explicitly state any assumptions you need to answer the following questions.

a. State any additions/modifications required to the hardware data path or describe how the instruction can be implemented with no physical modifications.

The execution of \texttt{icb} requires the contents of the register to be decremented. We can provide this value of 1 as an input to the \texttt{AluSrcB} mux which makes the number of inputs 5, and therefore the \texttt{AluSrcB} mux requires a three bit control signal. The \texttt{RegDst} field must be expanded to a 2 bit field since the \texttt{rs} register must now be written using \texttt{rs} as a destination address. Thus this instruction has a WB state.

b. Show the modified state machine that will handle this instruction. Provide the values of relevant control signals in each additional state. Any control signals not specified will assumed to be X.

```
from decode

AluSrcA = 1
AluSrcB = 100
AluOp = 00
PCWriteCond
PCSource = 10

RegWrite
RegDst = 10
MemToReg = 0
to fetch
```
c. Show the additions to the microcode to implement this instruction. The existing micro-coded implementation of the state machine before addition of this instruction is shown below. You need only show additions/modifications. Note that there is more than one approach to solving this problem. Some solutions may require additions to the microinstruction format in your text. If so state them explicitly.

### Dispatch ROM 1

<table>
<thead>
<tr>
<th>Op</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>R-format</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>jmp</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>beq</td>
<td>1000</td>
</tr>
<tr>
<td>100011</td>
<td>lw</td>
<td>0010</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0010</td>
</tr>
<tr>
<td>101000</td>
<td>icb</td>
<td>1010</td>
</tr>
</tbody>
</table>

### Dispatch ROM 2

<table>
<thead>
<tr>
<th>Op</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>lw</td>
<td>0011</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0101</td>
</tr>
</tbody>
</table>

### ALU control

<table>
<thead>
<tr>
<th>label</th>
<th>ALU control</th>
<th>src1</th>
<th>src2</th>
<th>Register Control</th>
<th>Memory</th>
<th>PCwrite Control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>add</td>
<td>PC</td>
<td>4</td>
<td>read PC</td>
<td>ALU</td>
<td>seq</td>
<td></td>
</tr>
<tr>
<td>Mem1</td>
<td>add</td>
<td>A</td>
<td>extend</td>
<td>read ALU</td>
<td>dispatch-1</td>
<td></td>
<td>dispatch-2</td>
</tr>
<tr>
<td>LW2</td>
<td></td>
<td></td>
<td></td>
<td>write MDR</td>
<td>seq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW2</td>
<td></td>
<td></td>
<td></td>
<td>write ALU</td>
<td>fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rformat1</td>
<td>Func</td>
<td>A</td>
<td>B</td>
<td>write ALU</td>
<td>seq</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>ALUOut-Cond</td>
<td>fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>jmp addr</td>
<td>fetch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICB</td>
<td>add</td>
<td>A</td>
<td>+1</td>
<td>ALUOut-cond</td>
<td>seq</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write ALU</td>
<td>fetch</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
15. Consider the manner in which exceptions are accommodated in the SPIM datapath, and the manner in which the data path controller is modified to support exceptions.

a. In order to support \( n \) exceptions, how many additional states should be added to the state machine? How many additional microinstructions should added?

\[ n \]

b. Consider an illegal memory address exception. Show the changes to the state machine for this exception. Make any assumptions that you feel may be necessary. For each additional state clearly show the values of relevant control signals. Any signals left unspecified will be assumed to be X (don’t care).

The illegal memory address exception is detected in any state that performs a memory access.
c. What is the minimum number of cycles from the occurrence of an exception to the time the first instruction of the exception handler is fetched?

One. Exceptions are handled in one state in this datapath (refer to figures in the text). The next state is the fetch state.
16. Consider a base, multi-cycle SPIM machine operating at 200 Mhz, with the following program execution characteristics.

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Type + Immediate</td>
<td>4</td>
<td>45%</td>
</tr>
<tr>
<td>lw</td>
<td>5</td>
<td>25%</td>
</tr>
<tr>
<td>sw</td>
<td>4</td>
<td>20%</td>
</tr>
<tr>
<td>beq</td>
<td>3</td>
<td>7%</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>3%</td>
</tr>
</tbody>
</table>

The following parts are independent of each other.

a. The designers decided to add a new addressing mode wherein registers can be automatically incremented by 4. Therefore a sequence such as code block 1 below can be replaced by the instruction shown in code block 2. We have 30% of the lw instructions affected in this manner. Assuming that the cycle time does not change and the new instruction also takes 5 cycles, what is the speedup produced by the use of this new addressing mode?

```plaintext
lw $t0, 0($t1)  becomes  lw $t0, ($t1)+
addi $t1, $t1, 4
```

CPI_old = 0.45*4 + 0.25*5 + 0.20*4 + 0.07*3 + 0.03*3 = 4.15
Ex_old = #Instr * 4.15 * clock_cycle_time

With the optimization the number of instructions has been changed. If 30% of the lw instructions have been affected that means 0.3 * 0.25 = 0.075 or 7.5% of the total number of instructions have been affected. For each of these instructions we can eliminate one instruction by combining two instructions. The new total number of instructions is therefore (0.925 * #Instr). The eliminated instructions are all immediate instructions. Therefore the percentage of such instructions are reduced by 7.5%.

CPI_new = 0.375*4 + 0.25*5 + 0.2*4 + 0.07*3 + 0.03*3)/0.925 = 4.17
Ex_new = (0.925 * #Instr) * 4.17 * clock_cycle_time

Comparing the execution times we can see that this is a good trade-off.
b. New ALU implementation technology has made it possible to double the speed of the R-format and immediate instructions. What is overall the speedup that can be realized for programs with the instructions statistics shown in the table?

From Amdahl’s Law we have Speedup = 1/(s + ((1-s)/n)) where s is the fraction that is unaffected, and n is the factor by which this component can be sped up. Plugging in the values from the table we have

Speedup = 1/(0.55 + (0.45/2)) = 1.29 or 29% speedup.

c. What is the maximum MIPS rating of the multi-cycle datapath?

The fastest instruction takes 3 cycles, i.e., has a CPI of 3. This is the theoretically lowest CPI possible for any program (although it is unrealistic to have a program that only branches opr jumps!)

MIPS Rating = (Clock Rate in MHZ)/CPI = 200/3 = 66.7 MIPS
17. We wish to make our datapath implement instructions such as those that are found in the Power PC. Specifically we are interested in adding the following instruction: \texttt{lwr} \texttt{f} \texttt{r} \texttt{f} \texttt{s}t0, \texttt{$t1+$t2}. As you may recall, this instruction uses the sum of the contents of registers \texttt{$t1} and \texttt{$t2} as addresses. Assume that this instruction uses R-format encodings where the \texttt{shamt} and \texttt{func} fields are ignored and an opcode 101000. Explicitly state any assumptions you need to answer the following questions.

a. State any additions/modifications required to the hardware data path or describe how the instruction can be implemented with no physical modifications.

No changes are required. Simply add \texttt{rs} and \texttt{rt} to obtain the memory address.

b. Show the modified state machine that will handle this instruction. Provide the values of relevant control signals in each additional state. Any control signals not specified will assumed to be \texttt{X}.

\begin{center}
\begin{tikzpicture}
  \node (decode) at (0,0) {\texttt{ALUSrcA = 1}};
  \node (alu) [below of=decode] {\texttt{ALUSrcB = 00 \ ALUOp = 00}};
  \node (memread) [below of=alu] {\texttt{MemRead IorD = 1}};
  \node (regwrite) [right of=memread] {\texttt{RegDst = 1 \ RegWrite MemToReg = 1}};
  \node (tofetch) [above of=regwrite] {\to fetch};

  \draw[->] (decode) -- (alu);
  \draw[->] (alu) -- (memread);
  \draw[->] (memread) -- (regwrite);
  \draw[->] (regwrite) -- (tofetch);
\end{tikzpicture}
\end{center}
c. Show the additions to the microcode to implement this instruction. The existing micro-coded implementation of the state machine before addition of this instruction is shown below. You need only show additions/modifications. Note that there is more than one approach to solving this problem. If any changes are necessary to the microinstruction format, state them explicitly.

<table>
<thead>
<tr>
<th>Dispatch ROM 1</th>
<th>Dispatch ROM 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Op</strong></td>
<td><strong>Name</strong></td>
</tr>
<tr>
<td>000000</td>
<td>R-format</td>
</tr>
<tr>
<td>000010</td>
<td>jmp</td>
</tr>
<tr>
<td>000100</td>
<td>beq</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>label</th>
<th>ALU control</th>
<th>src1</th>
<th>src2</th>
<th>Register Control</th>
<th>Memory</th>
<th>PCwrite Control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>add</td>
<td>PC</td>
<td>4</td>
<td>read PC</td>
<td>ALU</td>
<td>seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>PC</td>
<td>extshft</td>
<td></td>
<td></td>
<td>dispatch-1</td>
<td></td>
</tr>
<tr>
<td>Mem1</td>
<td>add</td>
<td>A</td>
<td>extend</td>
<td></td>
<td></td>
<td>dispatch-2</td>
<td></td>
</tr>
<tr>
<td>LW2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rformat1</td>
<td>Func</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td>seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEQ1</td>
<td>Subt</td>
<td>A</td>
<td>B</td>
<td>ALUOut-Cond</td>
<td></td>
<td>fetch</td>
<td></td>
</tr>
<tr>
<td>JUMP1</td>
<td></td>
<td></td>
<td></td>
<td>jmp addr</td>
<td></td>
<td>fetch</td>
<td></td>
</tr>
<tr>
<td>LWRF</td>
<td>add</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td>seq</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


18. Consider the manner in which exceptions are accommodated in the SPIM datapath, and the manner in which the data path controller is modified to support exceptions.

a. How are exceptions different from procedure calls?
   - Procedure calls perform state saving operations and make use of the stack. Exceptions need not be concerned with state saving operations.
   - Exceptions must encode the cause of the exception or use a vectored implementation to decode the exception.
   - Exception logic stores the address of the offending instruction rather than the next instruction.
   - Procedure calls are normal changes in the flow of control rather than unexpected changes.
   - Exceptions can lead to unexpected program termination.

b. Consider an illegal instruction exception. Show the changes to the state machine for this exception. Make any assumptions that you feel may be necessary. For each additional state clearly show the values of relevant control signals. Any signals left unspecified will be assumed to be X (don’t care).

The illegal instruction exception can be detected in the decode state. A transition is made to the exception state where the code is recorded and the state machine moves to the fetch state to start fetching the first instruction of the exception handler.
c. What is the minimum number of cycles from the occurrence of an exception to the time the first instruction of the exception handler is fetched?

One. Exceptions are handled in one state in this datapath (refer to figures in the text). The next state is the fetch state.
19. Consider the execution of the following block of SPIM code on a multi-cycle datapath. Assume that the fetch cycle for the first instruction is cycle 0.

```
add $t4, $0, $0
addi $t3, $0, 8
```

```
loop: lw $t0, 0($t1)
add $t4, $t4, $t0
addiu $t1, $t1, 4
addi $t3, $t3, -1
bne $t3, $0, loop
```

a. Fill in the following values at the end of the requested cycle.

The first row corresponds to the WB cycle for the load instruction. The second row corresponds to the decode cycle.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALUSrcB</th>
<th>ALUOp</th>
<th>RegDst</th>
<th>PCWrite</th>
<th>MemToReg</th>
<th>PCSource</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>00</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>00</td>
</tr>
<tr>
<td>18</td>
<td>11</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>

b. On what cycle is the branch instruction fetched for the last time?

165. The loop executes 8 times. Each immediate instructions takes 4 cycles.
20. Consider a base, multi-cycle SPIM machine operating at 50 Mhz, with the following program execution characteristics.

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Type</td>
<td>4</td>
<td>40%</td>
</tr>
<tr>
<td>lw</td>
<td>5</td>
<td>25%</td>
</tr>
<tr>
<td>sw</td>
<td>4</td>
<td>25%</td>
</tr>
<tr>
<td>beq</td>
<td>3</td>
<td>7%</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>3%</td>
</tr>
</tbody>
</table>

We can increase reduce the clock cycle time by 15% at the expense adding an additional cycle for data memory accesses.

a. Is this a good tradeoff. Provide a clear and quantitative justification for your answer.

\[
CPI_{\text{old}} = 0.4*4 + 0.25*5 + 0.25*4 + 0.07*3 + 0.03*3 = 4.15
\]

\[
\text{Execution Time}_{\text{old}} = I * 4.15 * \text{clock\_cycle}
\]

\[
CPI_{\text{new}} = 0.4*4 + 0.25*6 + 0.25*5 + 0.07*3 + 0.03*3 = 4.65
\]

\[
\text{Execution Time}_{\text{new}} = I * 4.65 * 0.85 * \text{clock\_cycle}
\]

\[
\text{Execution Time}_{\text{new}} < \text{Execution Time}_{\text{old}}
\]

b. What is the minimum improvement in clock cycle time that is necessary for this approach to be profitable?

\[
x* 4.65 < 4.15
\]

\[
x < 4.15/4.65 = 0.892.
\]
21. Consider the multi-cycle datapath.
   a. What is the minimum number of cycles for the execution of any instruction?
      3
   b. For a datapath with a $N$ state controller, what is the size of the microcode store (in number of microinstructions)?
      $2 \log N$
   c. What is it that determines the duration of the clock cycle time?
      The state (or cycle) in the datapath that has the longest execution path or delay.
22. Consider a base, multi-cycle SPIM machine operating at 50 Mhz, with the following program execution characteristics.

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Type</td>
<td>4</td>
<td>45%</td>
</tr>
<tr>
<td>lw</td>
<td>5</td>
<td>25%</td>
</tr>
<tr>
<td>sw</td>
<td>4</td>
<td>20%</td>
</tr>
<tr>
<td>beq</td>
<td>3</td>
<td>7%</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>3%</td>
</tr>
</tbody>
</table>

a. With clever algorithms for register allocation we can reduce the number of *lw* and *sw* instructions by 10%. What is the improvement in execution time?

\[
\begin{align*}
\text{CPI}_{\text{old}} &= 4*0.45 + 5*0.25 + 4*0.2 + 3*0.07 + 3*0.03 = 4.15 \\
\text{CPI}_{\text{new}} &= (0.45*4 + 5*0.25*0.9 + 4*0.2*0.9 + 3*0.07 + 3*0.03)/I'' = 3.945/I'' \\
I'' &= 0.45 + 0.25*0.9 + 0.2*0.9 + 0.07 + 0.03 = 0.955 \\
\text{CPI}_{\text{new}} &= 3.945/0.955 = 4.13 \\
\text{Execution Time Old} &= I * 4.15 * \text{clock_cycle_time} \\
\text{Execution Time New} &= 0.955* I * 4.13 * \text{clock_cycle_time} \\
\end{align*}
\]

Compare!

b. Would a 10% reduction in clock cycle time justify a single cycle increase in the number of cycles for memory accesses?

\[
\begin{align*}
\text{CPI}_{\text{new}} &= (0.45*4 + 6*0.25 + 5*0.2 + 3*0.07 + 3*0.03) = 4.6 \\
\text{Execution Time New} &= I * 4.6 * 0.9 * \text{clock_cycle_time} \\
&= I * 4.14 * \text{clock_cycle_time} \\
\end{align*}
\]

Barely better
23. Consider a base, multi-cycle SPIM machine operating at 100 Mhz, with the following program execution characteristics. Compiler optimizations reduce the number of instructions in each class to the amount shown.

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
<th>Frequency</th>
<th>Compiler Opt</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Type</td>
<td>4</td>
<td>45%</td>
<td>95%</td>
</tr>
<tr>
<td>lw</td>
<td>5</td>
<td>30%</td>
<td>90%</td>
</tr>
<tr>
<td>sw</td>
<td>4</td>
<td>20%</td>
<td>95%</td>
</tr>
<tr>
<td>beq</td>
<td>3</td>
<td>5%</td>
<td>75%</td>
</tr>
</tbody>
</table>

a. What is the base CPI with and without compiler optimizations?

Without Opt = 0.45*4 + 0.3*5 + 0.2*4 + 0.05*3 = 4.25

With Optimization = Cycles/#instr

\[ = \frac{(0.45*4*0.95+0.3*5*0.9+0.2*4*0.95+0.05*3*0.75)}{(0.45*0.95+0.3*0.9+0.2*0.95+0.05*0.75)} = \text{CPI}_{\text{new}} \]

b. What is the speedup due to compiler optimizations?

\[ \text{Speedup} = \frac{T_{\text{old}}}{T_{\text{new}}} = \frac{(I \times 4.25 \times \text{cycle}_\text{time})}{(I'' \times \text{CPI}_{\text{new}} \times \text{cycle}_\text{time})} \]

where

\[ I'' = (0.45*0.95+0.3*0.9+0.2*0.95+0.05*0.75) \] (from above: this is the new number of instructions)
24. Consider the annotated multi-cycle datapath shown in your text. Fill in the fill in the following table for the sequence of instructions shown below. Each entry should contain the value of the requested signal at the end of the requested cycle. The initial contents of registers $1$, $2$ and $3$ are 0x0000088, 0x0000022 and 0x00000033 respectively. The address of Loop is 0x00400004.

<table>
<thead>
<tr>
<th>PC</th>
<th>Instruction</th>
<th>Cycle</th>
<th>ALUOutput</th>
<th>Value at the Output of the ALU</th>
<th>Rt</th>
<th>ALUOp</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0040000c</td>
<td>beq $1, $2, Loop</td>
<td>3</td>
<td>0x00400004</td>
<td>0x00000066</td>
<td>0x0000022</td>
<td>01</td>
</tr>
<tr>
<td>0x00400010</td>
<td>sw $1, 4($2)</td>
<td>4</td>
<td>0x0000026</td>
<td>0x0000026</td>
<td>0x0000088</td>
<td>00</td>
</tr>
</tbody>
</table>
25. Consider the execution of the following block of SPIM code on a multi-cycle datapath. Assume that the fetch cycle for the first instruction is cycle number 0. Assume that la, li and addi require the same number of states as an add instruction.

```
.data
L1:  .word 2,3,4,5
L2:  .space 16
.text
la $t8, L1
la $t9, L2
li $t7, 4
move: lw $t2, 0($t8)
      sw $t2, 0($t9)
      addi $t8, $t8, 4
      addi $t9, $t9, 4
      addi $t7, $t7, -1
end:  bne $t7, $0, move
```

a. What would be the CPI and MIPS rating for a 266 MHz machine for the program shown above?

The loop executes four times. Each time through the loop the instructions take a total of 24 cycles. The first three instructions take 12 cycles. The total number of cycles is 108. The number of instructions in the loop are 6. Therefore the total number of instructions executed is $6 \times 4 + 3 = 27$. The CPI = $108 / 27 = 4.0$.

MIPS rating = $266 / CPI$

b. Fill in the following values during the requested cycle (remember the first cycle is cycle number 0!).

<table>
<thead>
<tr>
<th>Cycle #</th>
<th>ALUSrcB</th>
<th>IorD</th>
<th>PCWrite</th>
<th>RegDst</th>
<th>MemToReg</th>
<th>PCSource</th>
</tr>
</thead>
<tbody>
<tr>
<td>17</td>
<td>01</td>
<td>0</td>
<td>1</td>
<td>X</td>
<td>X</td>
<td>00</td>
</tr>
<tr>
<td>40</td>
<td>XX</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>XX</td>
</tr>
</tbody>
</table>

The first row corresponds to the fetch cycle for the `sw` instruction. The second row corresponds to the write back cycle for the `lw` instruction executed for the second time through the loop.
26. On the figure of the multi-cycle datapath, show any modifications that would be required to add the \texttt{jr} instruction. Below show the modifications to the state machine how any additional states would fit in the state machine. For each additional state that you have added, show the values of the relevant control signals. Any signals that you omit from the description of a state are assumed to be de-asserted.

According to the format definition for the \texttt{jr} instruction, the \texttt{rs} field identifies the register. This value will be available in the \texttt{A} register at the end of the decode cycle. Add another signal from the \texttt{A} register to the \texttt{PCSource} mux. The value of \texttt{PCSource} of \texttt{11} will now select the contents of the \texttt{A} register to be written to the PC. This writeback can be handled in one extra state after decode with \texttt{PCWrite} = 1 and \texttt{PCSource} = 11.
27. Consider the execution of the following block of SPIM code on a multi-cycle datapath. The text segment starts at 0x00400000 and that the data segment starts at 0x10010000. Assume immediate instructions take 4 cycles.

```
.data
start: .word 21, 22, 0x44, 0x12
str: .asciiz "Done 
.align 4
.byte 0x88, 0x32
.text
.globl main
main: li $t3, 2
    lui $t0, 0x1001
    lui $t1, 0x1001
    addi $t1, $t1, 8
move: lw $t5, 0($t0)
    lw $t6, 0($t1)
    add $t7, $t5, $t6
    addi $t0, $t0, 4
    addi $t1, $t1, 4
    addi $t3, $t3, -1
end:  bne $t3, $zero, move
done:
```

a. Fill in the following values at the end of the requested cycle. Assume that the first cycle of program execution is cycle number 0!

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALUSrcB</th>
<th>MemToReg</th>
<th>RegWrite</th>
<th>Branch Address</th>
<th>RegDst</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>01</td>
<td>X</td>
<td>0</td>
<td>0x00400030</td>
<td>X</td>
</tr>
<tr>
<td>28</td>
<td>00</td>
<td>X</td>
<td>0</td>
<td>0x0041e05c</td>
<td>X</td>
</tr>
</tbody>
</table>

b. Show the addresses and corresponding contents of the locations in the data segment that are loaded by the above SPIM code. Clearly associate addresses and corresponding contents.

<table>
<thead>
<tr>
<th>Data Segment Addresses</th>
<th>Data Segment Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x10010000</td>
<td>0x00000015</td>
</tr>
<tr>
<td>0x10010004</td>
<td>0x00000016</td>
</tr>
<tr>
<td>0x10010008</td>
<td>0x00000044</td>
</tr>
<tr>
<td>0x1001000c</td>
<td>0x00000012</td>
</tr>
<tr>
<td>0x10010010</td>
<td>0x656e6f44</td>
</tr>
<tr>
<td>0x10010014</td>
<td>0x00000020</td>
</tr>
<tr>
<td>0x10010018</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x1001001c</td>
<td>0x00000000</td>
</tr>
<tr>
<td>0x10010020</td>
<td>0x00003288</td>
</tr>
<tr>
<td>0x10010024</td>
<td></td>
</tr>
</tbody>
</table>
28. Consider a base, multi-cycle SPIM machine operating at 300 Mhz, with the following program execution characteristics.

<table>
<thead>
<tr>
<th>Instruction class</th>
<th>CPI</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-Type or Immediate</td>
<td>4</td>
<td>40%</td>
</tr>
<tr>
<td>lw</td>
<td>5</td>
<td>25%</td>
</tr>
<tr>
<td>sw</td>
<td>4</td>
<td>20%</td>
</tr>
<tr>
<td>bne</td>
<td>3</td>
<td>10%</td>
</tr>
<tr>
<td>j</td>
<td>3</td>
<td>5%</td>
</tr>
</tbody>
</table>

Questions 26(a), 26(b), and 26(c) are independent of each other.

a. The designers propose adding a new branch instruction, decrement-and-branch-if-zero: \texttt{dcb $t0, loop}. This instruction decrements the register $t0$. If the resulting value is equal to 0 then the program branches to the instruction at label loop. Therefore a sequence of instructions such as code block 1 below can be replaced by the instruction shown in code block 2. We have 35% of the branch instructions affected in this manner. Assuming that the cycle time does not change and the new instruction takes 4 cycles, is this a good idea?

\begin{verbatim}
addi $t2, $t1, -1
bne $t2, $0, loop
\end{verbatim}

\texttt{code block 1}

\texttt{dcb $t2, loop}

\texttt{code block 2}

\begin{align*}
\text{CPI}_\text{old} &= 0.4*4 + 0.25*5 + 0.20*4 + 0.10*3 + 0.05*3 = 4.10 \\
\text{Ex}_\text{old} &= \#\text{Instr} \times 4.1 \times \text{clock_cycle_time}
\end{align*}

With the optimization the number of instructions has been changed. If 35% of the branch instructions have been affected that means $0.35 \times 0.1 = 0.035$ or 3.5% of the total number of instructions have been affected. For each of these instructions we can eliminate one instruction by combining two instructions. The new total number of instructions is therefore $(0.965 \times \#\text{Instr})$. The eliminated instructions are all immediate instructions. Therefore the percentage of such instructions are reduced by 3.5%. However they are replaced by instructions taking 4 cycles. Therefore the net effect is the reduction of the number of branch instructions by 35% (or to 65% of the original).

\begin{align*}
\text{CPI}_\text{new} &= 0.365*4 + 0.25*5 + 0.2*4 + 0.1*3*0.65 + 0.05*3 + 4*0.035)/0.965 = 4.1 \\
\text{Ex}_\text{new} &= (0.965 \times #\text{Instr}) \times 4.1 \times \text{clock_cycle_time}
\end{align*}

Comparing the execution times we can see that this is a good trade-off.
b. If improved register allocation techniques in the compiler can reduce the number of \texttt{lw} and \texttt{sw} instructions by 10\% what is the improvement in the MIPS rating?

Current MIPS rating is 300/4.1 = 73.17 MIPS
With the reduced number of instructions the new CPI is given by \( (0.4 \times 4 + 0.225 \times 5 + 0.18 \times 4 + 0.1 \times 3 + 0.05 \times 3)/I' \)
where \( I' \) is given by \( 0.4 + 0.9 \times 0.25 + 0.9 \times 0.2 + 0.1 + 0.05 \)
The new MIPS rating is \( 300/CPI_{\text{new}} \)
The improvement is given by their differences.

c. Suppose in a typical program executable, 20\% of the instructions are from system libraries and implementing operating system related functionality. What is the maximum speedup that can be obtained by speeding up the application code sequences?

This is governed by Amdahl’s Law. We see the maximum speedup as \( 1/0.2 = 5 \).
29. We wish to have our multi-cycle datapath implement the \texttt{slti $t0, $t1, 0x4000} instruction. Explicitly state any assumptions you need to answer the following questions.

a. Either state any additions/modifications required to the hardware data path or describe how the instruction can be implemented with no physical modifications.

We can simply have the ALU perform a slt than operation on the inputs and select the ALU inputs accordingly. The key problem is getting the ALU controller to issue a \texttt{slti} opcode to the ALU (111). This is the main hardware modification that is required. Suppose we use the unused ALUOp value of 11 to signify this. Note that the truth table for the ALU controller must be re-implemented since we do not have don’t cares in one of the ALUOp bit fields any more.

b. Show the modified state machine that will handle this instruction. Provide the values of relevant control signals in each additional state. Any control signals not specified will assumed to be X.
c. Show the additions to the microcode to implement this instruction. The existing micro-coded implementation of the state machine before addition of this instruction is shown below. You need only show additions/modifications. Note that there is more than one approach to solving this problem. Some solutions may require additions to the microinstruction format in the text. If so state them explicitly.

Dispatch ROM 1

<table>
<thead>
<tr>
<th>Op</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>R-format</td>
<td>0110</td>
</tr>
<tr>
<td>000010</td>
<td>jmp</td>
<td>1001</td>
</tr>
<tr>
<td>000100</td>
<td>beq</td>
<td>1000</td>
</tr>
<tr>
<td>100011</td>
<td>lw</td>
<td>0010</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0010</td>
</tr>
<tr>
<td>001010</td>
<td>slti</td>
<td>1010</td>
</tr>
</tbody>
</table>

Dispatch ROM 2

<table>
<thead>
<tr>
<th>Op</th>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>100011</td>
<td>lw</td>
<td>0011</td>
</tr>
<tr>
<td>101011</td>
<td>sw</td>
<td>0101</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>label</th>
<th>ALU control</th>
<th>src1</th>
<th>src2</th>
<th>Register Control</th>
<th>Memory</th>
<th>PCwrite Control</th>
<th>Sequencing</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>add</td>
<td>PC</td>
<td>4</td>
<td>read PC</td>
<td>ALU</td>
<td>seq</td>
<td>dispatch-1</td>
</tr>
<tr>
<td></td>
<td>add</td>
<td>PC</td>
<td>extshft</td>
<td>read ALU</td>
<td></td>
<td>seq</td>
<td>SW2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write MDR</td>
<td></td>
<td>fetch</td>
<td></td>
</tr>
<tr>
<td>Mem1</td>
<td>add</td>
<td>A</td>
<td>extend</td>
<td></td>
<td></td>
<td>dispatch-2</td>
<td></td>
</tr>
<tr>
<td>LW2</td>
<td></td>
<td></td>
<td></td>
<td>write ALU</td>
<td></td>
<td>fetch</td>
<td>Rformat1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>seq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SW2</td>
<td></td>
<td></td>
<td></td>
<td>write ALU</td>
<td></td>
<td></td>
<td>BEQ1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>seq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rformat1</td>
<td>Func</td>
<td>A</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td>JUMP1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>write ALU</td>
<td></td>
<td></td>
<td>SLTI</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>seq</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Write ALU-rt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>fetch</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>seq</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
30. Consider the manner in which exceptions are accommodated in the SPIM datapath, and the manner in which the data path controller is modified to support exceptions. You certainly do not want your programs to jump to addresses in the data segment, or for that matter let us say to any address greater than 0x10010000. If j instructions generate such an illegal address you want to generate an exception. This questions addresses the implementation of such an exception in the SPIM datapath.

a. How can you test for the occurrence of this exception?

The occurrence of the exception is detected by using an ALU to compare the jump address with 0x10010000 and generating a single bit exception single to denote the result of this comparison.

b. Describe the hardware modifications required to control to support such an exception.

From state 1001 we now make a transition to the exception state. To make this transition we must modify the next state logic of the state machine implementation. Currently the next state after state 1001 is state 0000. This must now be modified to move to either state 0000 (if no exception) or state 1010 (exception state) of an exception occurred. You can use a table with these two alternatives just as we did for the next state logic out of state 0010. The input to the table is a single bit signal denoting whether the exception occurred or not (from 5(a)).

31. Consider the execution of the following block of SPIM code on a multi-cycle datapath. Assume that the fetch cycle for the first instruction is cycle number 0. Assume that i) all immediate instructions require the same number of states as an add instruction, ii) all branch instructions require the same number of states as a beq instruction. Text starts at 0x00400000 and the data segment starts at 0x10010000.

```assembly
# a simple counting for loop and a conditional if-then-else statement
.data
L1:   .word 0x44,22,33,55
.text
.globl main
main:  lui $t0, 0x1001
      li $t1, 4
      add $t2, $t2, $zero
loop:  lw $t3, 0($t0)
      add $t2, $t2, $t3
      addi $t0, $t0, 4
      addi $t1, $t1, -1
      bne $t1, $zero, loop
      bgt $t2, $0, then
      move $s0, $t2
      j exit
then:  move $s1, $t2
exit:   li $v0, 10
        syscall
```

c. What is the contents of ALUOut after cycle number 14 completes? Why?

0x1001000. Cycle 14 is the address calculation cycle of the lw instruction. The address calculation is $t0 + 0. The contents of $t0 are 0x10010000.

d. Fill in the following values during the requested cycle (remember the first cycle is cycle number 0!).

<table>
<thead>
<tr>
<th>Cycle</th>
<th>ALUOp</th>
<th>ALUSrcB</th>
<th>IRWrite</th>
<th>RegWrite</th>
<th>MemToReg</th>
<th>PCSource</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>00</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>XX</td>
</tr>
<tr>
<td>18</td>
<td>00</td>
<td>11</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>XX</td>
</tr>
</tbody>
</table>

Cycle number 10 corresponds to the addition operation for the add $t2, $t2, $zero instruction. Cycle number 18 corresponds to the decode state for the add $t2, $t2, $t3 instruction.

Loop1: lw $t3, 0($t0) 
addi $t0, $t0, 4 
bne $t1, $zero, Loop1

Loop2: lw $t3, 0($t0) 
addi $t0, $t0, 4 
add $t2, $t2, $t3