ECE 3056: Architecture, Concurrency, and Energy of Computation

Sample Problem Sets: Pipelining
1. Consider the following code sequence used often in block copies. This produces a special form of the load hazard and normally introduces one stall cycle.

```assembly
lw $7, 0($12)
sw $7, 0($13)
```

Show how you can use forwarding to avoid the occurrence of stalls in this situation. Provide your solution by modifying the view of the datapath provided below and precisely state the conditions you would check to implement such forwarding.
2. Consider the following instruction sequence with respect to the pipelined datapath with hazard detection and data forwarding. Assume writes in WB take place in the first half of the cycle and reads take place in the second half of the cycle. Registers $3, $4, and $5 contain 0x11, 0x22 and 0x88 respectively. Branches are assumed not taken with 38% of conditional branches being taken in practice with flushing. Branches are implemented in ID with forwarding.

```
lw $4, 0($3)
slt $5, $4, $2
bne $5, $0, Cont
li $4, 0x0400
addi $4, $4, 0x0004
sw $9, 0($4)
```

**Cont:**

a. If the first instruction starts at location 0x1000000, what are the values of the following variables in the pipeline during cycle 4 assuming the branch is taken?

- IF/ID.PC: _0x10000008___________
- EX/MEM.aluresult: _____0x11___________
- EX/MEM.write.value: _____0x22___________
- EX/MEM.wreg.addr: _____4 _____________

b. For the case of the branch being taken as well as the case of the branch not taken, at what cycle will the last instruction exit the pipeline?

Counting the first cycle as cycle 0.

**Branch Not Taken:** On cycle 11 when not taken since there is a stall after the *lw* and the *slt* instructions and branches are assumed not taken. The *sw* is in IF at cycle 7 and four cycles later exits.

**Branch Taken:** On cycle 11 when the branch is taken since there is a stall after the *lw* and the *slt* instructions. The branch will flush the instruction fetched after the branch and will then start fetching from the *addi* instruction in cycle 6. The *sw* will exit in cycle 11.
3. Assume branch instructions occur 22% of the time and are predicted as not taken, while in practice they are taken 42% of the time with a penalty of 2 cycles. With forwarding, the load delay slot is one cycle and can be filled 55% of the time with useful instructions. 21% of the instructions are loads and 30% of these introduce load delay hazards.

a. What is the increase in CPI due to load delay slots and branch hazards?

load slots + branch delay slots

\[ 0.21 \times 0.3 \times 0.45 + 0.22 \times 0.42 \times 2 = 0.213 \]

b. To improve performance, we would like to pipeline the memory accesses. This will reduce the clock cycle time by 20%, but increase the load delay slots to 2 cycles while not affecting branch delays. Is this worthwhile? Provide a quantitative justification for your answer.

\[ CPI_2 = (0.21 \times 0.3 \times 0.45 \times 2) + (0.22 \times 0.42 \times 2) = 0.2415 \]

\[ CPU\ Time_1 = I \times CPI_1 \times clock\_cycle\_time \]

\[ CPU\ Time_2 = I \times CPI_2 \times (clock\_cycle\_time \times 0.8) \]

The answer reduces to whether

\[ (CPI + 0.213) > (CPI + 0.2415) \times 0.8 \]
4. Consider the following instruction sequence with respect to the pipelined datapath with load hazard detection, data forwarding, and branch prediction (not taken) with flushing and branches are resolved in ID with forwarding. Assume writes in WB take place in the first half of the cycle and reads take place in the second half of the cycle.

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>add $7, $0, $0</td>
</tr>
<tr>
<td>2</td>
<td>addi $5, $0, 72</td>
</tr>
<tr>
<td>3</td>
<td>addi $3, $0, x10028</td>
</tr>
<tr>
<td>4</td>
<td><strong>loop</strong>: lw $4, 0($3)</td>
</tr>
<tr>
<td>5</td>
<td>lw $9, -8($3)</td>
</tr>
<tr>
<td>6</td>
<td>lw $6, -4($3)</td>
</tr>
<tr>
<td>7</td>
<td>add $7, $6, $7</td>
</tr>
<tr>
<td>8</td>
<td>add $7, $9, $7</td>
</tr>
<tr>
<td>9</td>
<td>add $7, $4, $7</td>
</tr>
<tr>
<td>10</td>
<td>addi $3, $3, -12</td>
</tr>
<tr>
<td>11</td>
<td>addi $5, $5, -12</td>
</tr>
<tr>
<td>12</td>
<td>bne $5, $0, loop</td>
</tr>
</tbody>
</table>

a. If the first instruction in the above sequence starts at location 0x400, what are the values of the following variables in the pipeline during cycle 8. The first cycle is cycle 1.

<table>
<thead>
<tr>
<th>Both Forwarding Unit Outputs</th>
<th>00, 00</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output of the ALU</td>
<td>10024</td>
</tr>
<tr>
<td>IF/ID.PC</td>
<td>41c</td>
</tr>
<tr>
<td>EX.MEM. RegWrite</td>
<td>1</td>
</tr>
<tr>
<td>ID/EX.Register.Rs</td>
<td>10028</td>
</tr>
</tbody>
</table>

The instruction in IF is **add $7, $9, $7**

b. From the code, you can determine how many times this loop is executed. Furthermore, remember branches are predicted as not taken (with flushing), and full forwarding and load hazard detection. What is the CPI value that you would obtain if you computed this value only over this block of code? Ignore the pipeline setup time and assume there is forwarding to the branch logic in ID.

74/57 (last time through the loop the branch is not taken)

Note that there are three stall cycles – between instructions 6-7, 11-12, and after the branch. The loop executes 6 times, each iteration taking 12 cycles (9 instructions + 3 stall cycles). The first three instructions take 3 cycles (assuming we start counting when the first instruction is in WB thereby ignoring pipeline startup time). During the last iteration, there is no stall cycle after the **bne** instruction. Therefore the loop takes (6x12) - 1 cycles = 71. Including the first three instructions we have 71+3 = 74 cycles. The total number of instructions are 3 + 6x9 = 57.
c. Suppose a page fault occurred on the data reference for instruction 6. State i) the instructions in the pipeline at the time, ii) those which are allowed to complete execution, and iii) at which instruction does execution resume after the page fault has been serviced, i.e., which is the first instruction to be fetched.

IF - instr 8
ID - instr 7
EX -- bubble
MEM - instr 6
WB -=- instr 5

The above shows the state when the fault occurs. Instruction 5 is allowed to complete and execution will resume with the fetching of instruction 6.
5. Assume branch instructions occur 15% of the time and are predicted as not taken, while in practice they are taken 40% of the time with a penalty of 3 cycles. With forwarding, the load delay slot is one cycle and can be filled 60% of the time with useful instructions. 20% of the instructions are loads and 30% of these introduce load delay hazards.

   a. What is the increase in CPI due to load delay slots and branch hazards?
   
   \[
   \text{CPI} = 1 + \text{increase due to branches} + \text{increase due to loads} \\
   = 1 + 0.15 \times 0.4 \times 3 + 0.2 \times 0.3 \times 0.4 \times 1 \\
   = 1.204
   \]

   b. To improve performance, we would like to pipeline the ALU. This will reduce the clock cycle time by 15%, but increase the load delay slots to 2 cycles, and the branch delays to 4 cycles. Provide a quantitative basis for determining if this is a good tradeoff.

   \[
   \text{CPI}_{\text{new}} = 1 + 0.15 \times 0.4 \times 4 + 0.2 \times 0.3 \times 0.4 \times 2 = 1.288 \\
   \\
   \text{Ex}_{\text{new}} = I \times \text{CPI}_{\text{new}} \times 0.85 \times \text{clock}\_cycle = I \times 1.0948 \times \text{clock}\_cycle \\
   \text{Ex}_{\text{old}} = I \times \text{CPI}_{\text{old}} \times \text{clock}\_cycle = I \times 1.204 \times \text{clock}\_cycle \\
   \\
   \text{This is a good trade-off.}
   \]
6. Consider a new high speed SPIM processor design where we have an 8-stage pipeline. Relative to the design in the text, the memory stage has been partitioned into 2 stages and the ALU stage has been partitioned in to three stages (hence the new 8 stage pipeline). As a result, the clock cycle time has been reduced to 6 ns.

   a. What is the latency experienced by an instruction and the theoretical maximum throughput of this processor in instructions/second? Ignore hazards.

   Latency = 48 ns
   Maximum throughput is 167 MIPS

   b. Now we are examining the next generation design, where we will have two such pipelines operating concurrently, i.e., a superscalar design. An instruction fetched from memory may be executed in either pipeline. Memory is fast enough to supply two instructions in each fetch cycle. Repeat (a) for this pipeline.

   Latency is unchanged - each instruction must still go through the entire pipeline = 48 ns

   The maximum throughput has been doubled since two instructions can complete in each clock cycle = 334 MIPS
7. Consider the state of the following program executing within the SPIM pipeline during cycle 6 (start at cycle 1). Assume full hardware support for forwarding, branches predicted as not taken, and flushing when branches are taken and the branch condition is resolved normally in EX. The first instruction is at address 0x400. Assume that the mov instruction is a native instruction.

```
mov $6, $0
addi $4, $0, 32
loop :
 lw $10, 1000($4)
 blt $10, $6, next -- check if $10 < $6
 mov $10 $6 -- move $6 to $10
next:
 addi $4, $4, -4
 bne $4, $0, loop
```

a. What instructions are in each stage of the pipeline.

IF - mov $10, $6
ID - blt $10, $6, next
EX - bubble
MEM - lw $10, 0($4)
WB - addi $4, $0, 32

b. What are the values of the following fields?

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MEM/WB.ALUData</td>
<td>0x20</td>
</tr>
<tr>
<td>EX/MEM.MemToReg</td>
<td>1</td>
</tr>
<tr>
<td>IF/ID.PC</td>
<td>0x410</td>
</tr>
<tr>
<td>ID/EX.ALUOp</td>
<td>00</td>
</tr>
</tbody>
</table>

(c. Identify all the data and control hazards assuming no forwarding support.

<table>
<thead>
<tr>
<th>Data Hazards</th>
<th>Control Hazards</th>
</tr>
</thead>
<tbody>
<tr>
<td>addi ---&gt; lw</td>
<td>blt</td>
</tr>
<tr>
<td>lw ---&gt; blt</td>
<td>bne</td>
</tr>
<tr>
<td>addi ---&gt; bne</td>
<td></td>
</tr>
</tbody>
</table>
8. Consider the state of following procedure executing within the SPIM pipeline during cycle 6. Assume that the first instruction is fetched in cycle 0! Assume full hardware support for forwarding, branches predicted as not taken, and flushing when branches are taken. The penalties for hazards are as dictated by the datapath in the figure. Also assume concurrent read/write to the same register in the same cycle. The first instruction is at address 0x00400000. Assume that all instructions are native instructions.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>lw $t0, 0($a0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
<td>add, $a0, $a0, $a1</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>addi $sp, $sp, -16</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>sw $fp, 4($sp)</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>addi $fp, $sp, 16</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>sw $ra, 0($fp)</td>
</tr>
<tr>
<td></td>
<td>7</td>
<td>sw $t0, -4($fp)</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>addi $a1, $t0, $a0</td>
</tr>
</tbody>
</table>

a. What instructions are in each stage of the pipeline.

<table>
<thead>
<tr>
<th></th>
<th>IF</th>
<th>sw $t0, -4($fp)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ID</td>
<td>sw$fra, 0($fp)</td>
</tr>
<tr>
<td></td>
<td>EX</td>
<td>addi $fp, $sp, 16</td>
</tr>
<tr>
<td></td>
<td>MEM</td>
<td>sw $fp, 4($sp)</td>
</tr>
<tr>
<td></td>
<td>WB</td>
<td>addi $sp, $sp, -16</td>
</tr>
</tbody>
</table>

b. What are the values of the following fields? ALL multiplexor inputs are numbered from the top to bottom starting at 0. MuxA is the first one from the top in EX.

<table>
<thead>
<tr>
<th></th>
<th>MEM/WB.MemToReg</th>
<th>1 (note the convention for labeling)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ForwardMuxA, ForwardMuxB</td>
<td>01, 00</td>
</tr>
<tr>
<td></td>
<td>IF/ID.PC</td>
<td>0x00400018</td>
</tr>
<tr>
<td></td>
<td>ID/EX.RegWrite</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>EX/MEM.RegWrite</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>EX/MEM.WriteRegisterAddr</td>
<td>XX</td>
</tr>
</tbody>
</table>

c. Extensive analysis of the instruction stream of code targeted for the pipelined MIPS with forwarding has produced the following statistics. To support larger caches we have accepted a two cycle access time and therefore the IF and MEM stages have been partitioned into two stages. The alternative is to keep the single cycle hazards but with a clock that is 10% slower. The overall pipeline is now 7 stages. Penalties for load hazards are 2 cycles and those for control hazards are also 2 cycles. We note that 20% of load instructions produce a hazard and 30% of these load delay slots can be filled via
instruction scheduling. Similarly, we can use scheduling can fill 70% of the branch delay slots. Compute the new CPI. If the CPI without additional pipe stages was 1.5, is this design a good idea? Justify your answer.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>22%</td>
</tr>
<tr>
<td>Stores</td>
<td>13%</td>
</tr>
<tr>
<td>ALU Operations</td>
<td>42%</td>
</tr>
<tr>
<td>Branches</td>
<td>18%</td>
</tr>
</tbody>
</table>

New CPI = 1.5 + 0.22*0.2*0.7*1 + 0.18*0.3*1 = 1.5 + 0.0848 = 1.5848 (note the old CPI already accounts for 1 cycle).

Ex_old = I * 1.5 * 1.1 * clock_cycle
Ex_new = I * 1.5848 * clock_cycle

Not a good idea.
9. Extensive analysis of the instruction stream has produced the following statistics.
Data hazards incur a 2 cycle penalty while control hazards incur a 3 cycle penalty. You can leave you answers in expression form.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loads</td>
<td>22%</td>
</tr>
<tr>
<td>Stores</td>
<td>13%</td>
</tr>
<tr>
<td>ALU Operations</td>
<td>42%</td>
</tr>
<tr>
<td>Branches</td>
<td>18%</td>
</tr>
</tbody>
</table>

a. Assuming an ALU or Load/Store instruction has probability of 0.2 of experiencing a data hazard, what is the maximum increase in CPI due to data hazards and control hazards? There is no instruction scheduling and no forwarding.

Max Increase in CPI due to Data Hazards = (0.42+0.22+0.13) * 2 * 0.2
Max increase due to control hazards = (0.18 * 3).

b. Now assume that you have full hardware support for forwarding, and branches predicted as not taken supported by instruction flushing when branches are taken. Branches are assumed to be not taken and instructions are fetched accordingly. Branches are actually taken 44% of the time. 34% of the load instructions produce a single load delay slot which can be filled 60% of the time. What is the improvement in pipeline CPI from (a)?

The new CPI increase is
Due to Hazards = (0.22 * 0.34 * 0.4) * 1
Due to control hazards = (0.18 * 0.44 * 3)
c. Rather than using forwarding and hardware support for flushing on branches, suppose that instruction scheduling could successfully fill 70% of the delay slots produced by data and control hazards. Furthermore, the clock cycle time could be reduced by 10% since the multiplexors used for forwarding were on the critical path. Is this option a better one than the use of forwarding? If expressions are not evaluated, show precisely how you can answer this question.

New CPI increase due to data hazards is \((0.42+0.22+0.13) \times 2 \times 0.2 \times 0.3\)

New CPI increase due to control hazards is \(0.18 \times 3 \times 0.3\)

Execution time is \(I \times CPI_{\text{new}} \times (0.9 \times \text{cycle\_time})\)
Old Execution time is \(I \times CPI_{\text{old}} \times \text{cycle\_time}\).
Compare the two.
10. Consider the state of following procedure executing within the SPIM pipeline during cycle 8. Assume that the first instruction is fetched in cycle 0! Assume full hardware support for forwarding, branches predicted as not taken, and flushing when branches are taken. The penalties for hazards are as dictated by the datapath in the figure. The first instruction is at address 0x00400000. Assume that all instructions are native instructions.

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

loop:
  lw $22, 0($2)  -- WB
  hazard cycle -- MEM
  add $9, $9, $22  -- EX
  addi $2, $2, 4  -- ID
  bne $2, $3, loop  -- IF
  hazard cycle
  move $2, $9
  jr $31
```

a. What instructions are in each stage of the pipeline.

```
IF              bne $2, $3, loop
ID              addi $2, $2, 4
EX              add $9, $9, $22
MEM             nop
WB              lw $22, 0($2)
```

b. What are the values of the following fields? Mux inputs are numbered from the top to bottom starting at 0. MuxA is the first one from the top in EX.

```
MEM/WB.MemToReg 1 (according to text, 0 according to my convention stated above)
ForwardMuxA, ForwardMuxB 00 01
IF/ID.PC         0040001c
ID/EX.RegWrite   1
EX/MEM.RegWrite  0
EX/MEM.WriteRegister 0x16
```

While the nop has de-asserted control signals RegDst is 0 and the rest of the pipeline register has been updated and contains the contents of the add instruction.
c. If there was no forwarding, what is the earliest cycle in which the following instructions could execute for the first time in the stage shown. Remember we start counting cycles at cycle 0!

```
add $9, $9, $22 : In EX 11
bne $2, $3, loop: In MEM 16
```
11. Consider the following code sequence which computes a running sum of an array of integers stored starting at location Start: through memory address End:

```
add $7, $0, $0
la $4, End
la $3, Start

loop: lw $6, 0($3)
lw $5, 4($3)
add $7, $6, $7
add $7, $5, $7
addi $3, $3, 8
bne $3, $4, loop
```

a. Rewrite the code sequence assuming that the pipelined SPIM provides no support for forwarding or branch hazards (branches resolved in ID) and the compiler must ensure correctness by the proper insertion of nops. Assume register file write operations take place in the first half of the cycle and read operations in the second half of the cycle.

```
add $7, $0, $0
la $4, End
la $3, Start
nop
nop

loop: lw $6, 0($3)
lw $5, 4($3)
nop
add $7, $6, $7
nop
nop
add $7, $5, $7
addi $3, $3, 8
nop
nop
bne $3, $4, loop
nop
```
b. Now schedule the code in 2(a) to remove as many nops as possible.

```
la $3, Start
la $4, End
add $7, $0, $0

loop: lw $6, 0($3)
lw $5, 4($3)
addi $3, $3, 8
add $7, $6, $7
nop
bne $3, $4, loop
add $7, $5, $7
```
12. Consider execution of a MIPS code sequence on a pipelined data path shown on the next page. Consider a 2 GHz MIPS processor with a canonical 5-stage pipeline and 32 general-purpose registers. The branch condition and jump addresses are computed in decode. Branches are assumed not taken. The data path implements forwarding.

a. Show the modifications required to the pipelined datapath overleaf to implement support for the jal instruction.

There is more than one solution. The solution below has three functional additions.

1. Compute the jump address: Take the least significant 26 bits of the instruction, multiply by 4 (convert word address to byte address), concatenate the most significant 4 bits of the PC to generate a 32 bit address. This address is now a new input to the PCSrc mux. The controller generates the proper PCSrc mux control bits.
2. Add a third input to the RegDst mux hardwired to 31. Mux control is now 2 bits.
3. Make PC+4 available through WB and as a new input to to the MemToReg mux. Expand the number of bits that controls this mux.
4. Controller is updated to include a flush of IF/ID and generate a jal signal.
5. Since register 31 is not written until jal reaches WB, the procedure must have at least one instruction before executing a jr instruction.

b. What would be the values of all of the control signals to realize this implementation.

Note that some of these control signal values can be don’t care (e.g., RegDst)

<table>
<thead>
<tr>
<th>Instr.</th>
<th>RegDest</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>jal</td>
<td>10</td>
<td>0</td>
<td>10</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>00</td>
</tr>
</tbody>
</table>
c. Now consider the execution of the following code sequence – specifically the execution of the function `func`: Why will execution be incorrect in the datapath described on page 2 and what can you do to ensure correct execution? Assume support for the `jr` instruction is similar to that of the `jal` instruction and `addi` and `move` are supported similar to the `add` instruction.

```
.data
start:  .word 0x32, 33, 0x34, 35

.text
la $a0, start
addi $a1, $a0, 0x10
jal func
li $v0, 10
syscall

func:  add $t7, $zero, $zero

loop:  lw $t6, 0($a0)
       add $t7, $t7, $t6
       addi $a0, $a0, 4
       bne $a1, $a0, loop
       move $v0, $t7
       jr $31
```

Since branches are resolved in decode `bne` will not receive the correct forwarded value. Therefore we must either insert 2 stall cycles or 2 `nops` between `addi $a0, $a0, 4` and `bne $a1, $a0, loop`.

There is also a load-to-use hazard on the `lw` instruction that must either be supported in hardware or utilize a `nop`. 
14. We wish to implement an atomic swap instruction – \texttt{swap r1, offset(r2)}. This instruction atomically swaps the contents of a register with a location in memory.

   a. What does it mean for the execution of this instruction to be atomic?

   The execution of the instruction is atomic, if all of its actions are completed in their entirety without any intervening actions (read or write) to the state that is being accessed. For example, the increment of a memory location is atomic if when once begun, there is no intervening access to the memory location until the increment operation has completed.

   b. Using a few instructions, provide a MIPS assembly language implementation of this instruction using the following MIPS instructions.

   

   load linked: \texttt{ll r1, offset(r2)}

   store conditional: \texttt{sc r1, offset(r2)}

   The following code sequence atomically swaps the contents of register $s4$ and the memory location whose address is in register $s1$.

   

   loop: \texttt{add $t0,$zero,$s4}

   \texttt{ll $t1,0($s1) ;load linked}

   \texttt{sc $t0,0($s1) ;store conditional}

   \texttt{beq $t0,$zero,loop ;branch store fails}

   \texttt{add $s4,$zero,$t1 ;put load value in $s4}
15. Consider the following block of SPIM code. The text segment starts at 0x00400000 and the data segment starts at 0x10010000. Register $t1 is initialized to 0x10010000. The remaining registers are initialized to 0. The first three words of memory starting at 0x10010000 contain the values 0x4, 0x8, 0xC. Consider the state of the pipeline with forwarding and hazard detection on cycle 7 (the first cycle is 0!).

```
lw $t3, 8($t1)
lw $t0, 0($t1)
lw $t2, 4($t1)
add $t5, $t5, $t2
add $t5, $t5, $t0
add $t5, $t5, $t3
sw $t5, 12($t1)

li $v0, 10
syscall
```

List the instruction in each stage of the pipeline and fill in the table below.

<table>
<thead>
<tr>
<th>IF/ID.PC4</th>
<th>0x00400018</th>
</tr>
</thead>
<tbody>
<tr>
<td>ID/EX.ForwardA (top mux below)</td>
<td>01</td>
</tr>
<tr>
<td>ID/EX.RegisterRt (contents see below)</td>
<td>0x00000004</td>
</tr>
<tr>
<td>EX/MEM.WriteData</td>
<td>0x00000008</td>
</tr>
<tr>
<td>MEM/WB.ALUResult</td>
<td>0x100100004 (or 0x00000000 for nop)</td>
</tr>
</tbody>
</table>
16. Consider the following segment of code. Assume full forwarding and branches resolved in EX and support for the `addi` instruction.

```
  loop:    lw $t4, 0($t0)   #fetch array element
         add $t3, $t3, $t4   #update sum
         addi $t0, $t0, 4   #point to next word
         addi $t2, $t2, -1  #decrement count
         bne $t2, $0, loop
```

a. What is the CPI of the above code sequence assuming that the loop executes exactly 4 times? Count all cycles starting with the fetch of the first instruction and until the last instruction exits the pipeline. It should be clear as to exactly how you computed the CPI.

```
lw $t4, 0($t0)          # Time for the first instruction to reach WB = 4 cycles
<nop>
add $t3, $t3, $t4       4 executions of the loop = 4*9 = 36 cycles
addi $t0, $t0, 4
addi $t2, $t2, -1
bne $t2, $0, loop
<nop>
<nop>
<nop>  # Last 3 nops are no necessary = -3
<nop>
<nop>  # Total = 4+36-3 = 37
```

b. Now assuming no hardware support for forwarding or hazard detection, reschedule the code using `nops`. Minimize the number of `nops`. Writes take place in the first half of the cycle and reads in the second half.

```
addi $t2, $t2, -1
lw $t4, 0($t0)
addi $t0, $t0, 4
bne $t2, $0, loop
add $t3, $t3, $t4
<nop>
<nop>
```

There are other solutions that also use only 2 nops.
17. Consider the following instruction dependency that can be caused by copying data.

\[
\text{lw } $8, 0($9) \\
\text{sw } $8, 4($10)
\]

Modify the pipelined datapath to add forwarding so that this sequence can execute correctly without stall cycles. Describe (functionally) how this should work including values of any new control signals you may add.

Add a multiplexor at the input to MemData of the memory module. The two inputs to the mux are the WriteData input that was transmitted from the EX stage and the output of the MemToReg multiplexor from the WB stage (equivalently you can take the top input to the MemToReg mux). Now the mux will select according to the following forwarding condition

\[
\text{If } ((\text{EX/MEM.MemWrite} = '1') \text{ and } (\text{MEM/WB.MemToReg} = '1') \text{ and } \\
(\text{EX/MEM.WriteRegAddr} = \text{MEM/WB.WriteRegAddr}))
\]

\[\text{then Forward} = '1'; \# \text{ forward from WB} \]
\[\text{else Forward} = '0'; \# \text{ do not forward from WB} \]

The first two predicates check if there is a load followed by a store, while the last predicate checks if the \text{rt} registers are the same. The assumption here is that the \text{RegDst} signal is 0 for the store word command (the original truth table denotes \text{X}).
18. With forwarding and hazard detection, assume branch instructions occur 10% of the time and are predicted as not taken, while in practice they are taken 40% of the time with a penalty of 3 cycles. With forwarding, the load delay slot is one cycle and can be filled 40% of the time with useful instructions. 25% of the instructions are loads and 30% of these introduce load delay hazards. You may leave your answers in expression form.

a. What is the increase in CPI due to load delay slots and branch hazards?

\[ \text{CPI}_n = \text{CPI}_\text{base} + (0.1 \times 0.4 \times 3) + (0.25 \times 0.3 \times 0.6 \times 1) \]

Branches incur a 3 cycle penalty while load-to-use stalls incur 1 cycle penalty

b. Since memory is much slower than the core, to improve performance, we would like to pipeline all stages that access memories by making them take 3 cycles instead of 1. This will reduce the clock cycle time by 15%. Provide a quantitative basis for determining if this is a good tradeoff.

\[ \text{CPI}_\text{new} = (0.1 \times 0.4 \times 5) + (0.25 \times 0.3 \times 0.6 \times 3) \]

Both IF and MEM stages have added two pipeline stages. Hence the branch penalty is now 5 cycles and the load-to-use penalty is 2 cycles.

\[
\text{Exec}_\text{old} = I \times \text{CPI}_n \times \text{clock}_\text{cycle}_\text{time} \\
\text{Exec}_\text{new} = I \times \text{CPI}_\text{new} \times 0.85 \times \text{clock}_\text{cycle}_\text{time}
\]

Compare execution times to determine if this is a good tradeoff.
19. Consider two threads spawned to execute a function on different data sets and sharing the pipeline as shown below – only a segment of the function code is shown. The first instruction of thread #1 is fetched in the first cycle (cycle 0) and instructions from each thread are interleaved on an instruction-by-instruction basis thereafter. All registers are initialized to 0 except for $t0 which is initialized to 0x1001000 for thread #1 and 0x10010024 for thread #2 (initialization not shown). The first instruction (the load) is stored at 0x40040048.

![Diagram of two threads with instruction flow](image)

**Thread #1**
- `lw $t3, 0($t0)`
- `add $t2, $t2, $t3`
- `addi $t0, $t0, 4`
- `addi $t1, $t1, -1`
- `bne $t1, $zero, loop`

**Thread #2**
- `lw $t3, 0($t0)`
- `add $t2, $t2, $t3`
- `addi $t0, $t0, 4`
- `addi $t1, $t1, -1`
- `bne $t1, $zero, loop`

a. What are the contents of the following during cycle 9 (the first cycle is 0!)?

Thread #2, register $t0: 0x10010024 (note that the write happens at the end of the cycle)

ID/EX.PC (include thread ID): ____0x40040058____

b. Distinguish between fine grain and coarse grain multithreading.

In fine grained multithreading, threads share the core pipeline at a fine granularity, e.g., one instruction at a time as in the example above or a few instructions at a time.

In coarse grain multithreading, threads occupy the core pipeline for a much larger number of instructions. Thread swapping often happens on high latency events such as a cache miss or I/O operations. Coarse grain multithreading is better suited for software threads while fine grain multithreading is applied to hardware threads.