Allowable Instructions

When writing MIPS assembly, the only instructions that you are allowed to use (so far) are:

- add, addi, sub, addu, addiu
- and, andi, or, ori, xor, xori, nor
- lui
- beq, bne, j
- jal, jr
- slt, slti
- sll, sra, srl
- lw, lh, lb, sw, sh, sb
- la
- syscall

While MIPS has many other useful instructions (and the assembler recognizes many pseudo-instructions), do not use them! We want you to learn the fundamentals of how assembly language works - you can use fancy tricks after this class is over.
1. (10 points) Each phrase below is the definition for some important term relating to pipelining. Give each term.

**Solution: Rubric Note:**
1 correct answer (or less): 0 pts
2 pts for each answer beyond that

“A condition where the input register for a certain instruction is still in the pipeline (and not flushed to registers) when the instruction reads it.”

**Solution:** data hazard

“A condition where the next instruction to fetch is not known, because the current instruction changes the PC (or might do so).”

**Solution:** branch hazard

“The total amount of time it takes for an instruction to run from beginning to end, through the various stages of the pipeline.”

**Solution:** latency

“A feature that allows the EX phase to read its inputs from various pipeline registers - in case the values it read during the ID phase are out of date.”

**Solution:** data forwarding

“Delaying the pipeline for one or more clock cycles, to avoid issues which might cause faulty results.”

**Solution:** stall

“Cancelling instructions which are already in the pipeline, replacing them with NOPs because we’ve realized they never should have run.”

**Solution:** pipeline flush
2. For each question below, give a short answer - a few words or symbols, maybe a sentence or two.

(a) (5 points) Stalls are generally detected in the ID phase. Explain why they cannot be detected any earlier.

**Solution:** Before the ID phase, we don’t even know what instruction it is (because we haven’t decoded it).

(b) (5 points) If we decide to stall an instruction that is currently in the ID phase, what should we write to the ID/EX register?

**Solution:** zeroes

(c) (5 points) Give a two instruction sequence, which would not stall in a processor with data forwarding, but **would** stall in a processor that lacked data forwarding.

**Solution:** Solutions will vary. Generally, look for an instruction that writes to one register, followed by another which reads it.

(d) (5 points) Not all instructions do useful work in all of the pipeline phases. Explain why it is impossible to have an instruction “jump ahead” in the pipeline.

(NOTE: Saying that the CPU simply doesn’t have the wires to get this done is incorrect. You must explain why we could not **add** this feature - at least, not without a **major** change to the CPU.)

**Solution:** There probably is another instruction, just ahead - and and we can’t have two instructions in the same pipeline phase
(e) (5 points) Below, I have modified the standard function epilogue by adding one more instruction. Explain what impact this change would have on your program (assuming, of course, that you called this function).

```assembly
lw $ra, 4($sp)
addiu $ra, $ra,-4 # new!
lw $fp, 0($sp)
addiu $sp, $sp,24
jr $ra
```

Solution: This function would return to the JAL instruction, and thus immediately call back into this function - over and over, forever.

(f) (5 points) Assuming that our processor has the standard pipeline we’ve discussed, how far apart would two instructions (with a dependency between them) need to be so that data forwarding was not required?

- 1 instruction (that is, they are sequential)
- 2 instructions (that is, one instruction in-between)
- 3 instructions (that is, two instructions in-between)
- 4 instructions (that is, three instructions in-between)

Solution: If the two instructions are 3 instructions apart, then the former instruction will be in the WB phase when the latter is in ID; thus, the latter will read the correct value in ID (because WB happens before ID in any given clock cycle).

3. (20 points) Assume that you have a processor with data forwarding. Fill in the following pseudocode function, which figures out the value which should be delivered to the 2nd ALU input.

Your function should use the following control fields as input:

- `idex.rt` - the `rt` field of the instruction currently in EX
- `idex.ALUsrc` - the `ALUsrc` control bit for the instruction currently in EX
- `exmem.regWrite, memwb.regWrite` - for each of the instructions ahead, indicates whether or not the instruction will modify a register
- `exmem.writeReg, memwb.writeReg` - for each of the instructions ahead, indicates which register it will write to
- `memwb.memToReg`

Your function should return the value which should go into the 2nd ALU input, which is one of the following values:

- `idex.rtVal` - the value of the `rt` register, as it was read in the ID phase
- `idex.imm32` - the sign-extended immediate value from an I format instruction
- `exmem.aluResult, memwb.aluResult, memwb.memResult` - various values-in-flight in the pipeline registers ahead

WRITE YOUR ANSWER ON THE NEXT PAGE
Inputs: idex.rt, idex.ALUsrc, exmem.regWrite, memwb.regWrite, exmem.writeReg, memwb.writeReg, memwb.memToReg
Retvals: idex.rtVal, idex.imm32, exmem.aluResult, memwb.aluResult, memwb.memResult

int getALUInput2(...various parameters...)
{
    // Your code here!

    Solution: NOTE: You’re not expected to write comments, but I’ve included them to explain my solution.

    // if ALUsrc is 0, then we want to read the immediate value,
    // and data forwarding is meaningless.
    if (idex.ALUsrc == 0)
        return idex.imm32;

    // if both instructions ahead of us happen to write to the same
    // register, the one currently in MEM is later in the program, and
    // thus the current value for this instruction.
    if (exmem.regWrite == 1 && exmem.writeReg == idex.rt)
        return exmem.aluResult;

    if (memwb.regWrite == 1 && memwb.writeReg == idex.rt)
    {
        // the value might be something we read from memory, or a
        // value calculated by the ALU
        if (memwb.memToReg == 1)
            return memwb.memResult;
        else
            return memwb.aluResult;
    }

    // base case: we’re not reading the immediate field, and we don’t
    // need forwarding from anywhere.
    return idex.rtVal;
4. In each part below, you will be given a numbered sequence of instructions. Assume that instruction 1 will be in IF on clock cycle 1. Indicate which clock cycle each instruction will be in the WB phase.

For each part, give the answer both for a pipelined processor that has data forwarding, and also for one that does not.

(a) (10 points)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Forwarding</th>
<th>No Forwarding</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $s0, $s1,$s2</td>
<td><strong>5</strong></td>
<td><strong>5</strong></td>
</tr>
<tr>
<td>add $s3, $s0,$s0</td>
<td>_____</td>
<td>_____</td>
</tr>
<tr>
<td>add $s4, $s0,$s0</td>
<td>_____</td>
<td>_____</td>
</tr>
<tr>
<td>add $s4, $s7,$s6</td>
<td>_____</td>
<td>_____</td>
</tr>
<tr>
<td>add $s5, $s5,$s5</td>
<td>_____</td>
<td>_____</td>
</tr>
<tr>
<td>add $s6, $s0,$s0</td>
<td>_____</td>
<td>_____</td>
</tr>
</tbody>
</table>

**Solution:**

**Forwarding:** Since there are no LW instructions, no instructions will need to stall.

**No Forwarding:** Require at least 2 instructions *between* the write and read

(b) (10 points)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Forwarding</th>
<th>No Forwarding</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $s0, 0($s0)</td>
<td><strong>5</strong></td>
<td><strong>5</strong></td>
</tr>
<tr>
<td>add $s0, $s0,$s0</td>
<td>_____</td>
<td>_____</td>
</tr>
<tr>
<td>lw $s1, 0($s0)</td>
<td>_____</td>
<td>_____</td>
</tr>
</tbody>
</table>
**Solution:** 

**Forwarding:** Insert 1 instruction (or stall) between an LW and an instruction which uses that value.

**No Forwarding:** Require at least 2 instructions *between* the write and read.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Forwarding</th>
<th>No Forwarding</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw $s0, 0($s0)</td>
<td><em><strong>5</strong></em></td>
<td><em><strong>5</strong></em></td>
</tr>
<tr>
<td>add $s0, $s0,$s0</td>
<td><em><strong>7</strong></em></td>
<td><em><strong>8</strong></em></td>
</tr>
<tr>
<td>lw $s1, 0($s0)</td>
<td><em><strong>8</strong></em></td>
<td><em><strong>9</strong></em></td>
</tr>
<tr>
<td>lw $s2, 0($s0)</td>
<td><em><strong>9</strong></em></td>
<td><em><strong>10</strong></em></td>
</tr>
<tr>
<td>add $s1, $s0,$s0</td>
<td><em><strong>10</strong></em></td>
<td><em><strong>12</strong></em></td>
</tr>
<tr>
<td>add $s2, $s0,$s0</td>
<td><em><strong>11</strong></em></td>
<td><em><strong>13</strong></em></td>
</tr>
</tbody>
</table>

// stall twice
// stall twice
// no add’l stalls
5. (15 points) Draw arrows to show all data forwarding that happens in this sequence of instructions. Draw your arrows so that it is obvious if you are forwarding into the rs or rt input.
Solution:
6. (5 points) **DO THIS QUESTION LAST.** It’s not worth a lot of points.

Convert the MIPS assembly below into an equivalent C function. Use high-level C code (loops, function calls, etc.); **avoid** simply converting the assembly literally into C.

Use only two named variables in C (although you may have parameters as well); I’ve included their names in the comments below.

```assembly
.text
.globl callAndTotal
callAndTotal:
    addiu $sp, $sp,-24
    sw $ra, 4($sp)
    sw $fp, 0($sp)
    addiu $fp, $sp,20
    sw $a1, -8($fp)
    addiu $sp, $sp,-8
    sw $s0, 0($sp)
    sw $s1, 4($sp)
    addi $s0, $zero,0  # s0 - sum
    add $s1, $a0,$zero  # s1 - i
LOOP:
    lw $a1, -8($fp)
    slt $t2, $s1,$a1
    beq $t2,ZERO, LOOP_END
    add $a0, $s1,$zero
    add $a1, $s0,$zero
    jal each
    add $s0, $s0,$v0
add $s1, $s1,1
j LOOP
LOOP_END:
    add $v0, $s0,$zero
    lw $s0, 0($sp)
    lw $s1, 4($sp)
    addiu $sp, $sp,8
    lw $fp, 0($sp)
    lw $ra, 4($sp)
    addiu $sp, $sp,-24
    jr $ra
```

**Solution:**
```c
int callAndTotal(int start, int end)
{
    int sum = 0;
    for (int i=start; i < end; i++)
        sum += each(i, sum);
    return sum;
}
```