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. (a) (10 points) Name the 5 stages of the standard MIPS pipeline design. Then list one thing which each phase does. (They do many things; just list one for each.)

**Solution: GRADING NOTES:**
- We will accept either the phase name abbreviations, or the full names.
- The actual actions that a student might give will vary; my solutions below list all of the obvious answers I could think of. If a student lists something which is correct but I didn’t itemize below, we’ll give credit for that as well.
- Note that “check if the two registers are equal” is present in two phases; this is because the old BEQ design does this work in EX, but some students may have read ahead, and know that the new design places this in ID.

**IF - read one instruction; advance PC**
**ID - read registers from the register file; break the instruction into fields; set all of the control bits; determine whether or not to stall the instruction; check if the two registers are equal**
**EX - execute the ALU; choose the proper inputs for the ALU; determine if the output is zero; check if the two registers are equal; choose the destination register (that is, choose between rt/rd); do data forwarding into the ALU inputs**
**MEM - read memory; write memory**
**WB - update registers; choose whether to write the Memory or ALU output (that is, execute the MemToReg MUX). Also tolerated: choose the destination register (between rt/rd).**

(b) (5 points) Define bandwidth and latency, in terms of a pipelined processor design.

**Solution:** Bandwidth - how many instructions per second
Latency - How long (start to finish) it takes to complete a single instruction

(c) (5 points) Again, consider the control bits in the pipeline registers. Why is the same control bit saved in many different registers? Why not save it immediately into the proper register?

**Solution:** It has to be saved in pipeline registers for several clock cycles - from the time when it is generated (ID) until it is actually used.

**Instructor’s Note:** This was a difficult question to grade. Note that the question deals with control bits getting copied around - so this isn’t about data forwarding. It’s asking why the same control bit shows up in multiple registers.
You got credit if you communicated one of several ideas: that control bits are used on different clock cycles than they are generated; that “directly writing” to a register would overwrite other data (since each clock cycle generates a new set of control bits); that the control bits “travel with” their respective instructions; or that each instruction in the pipeline has its own set of control bits.

(d) (5 points) What is a pipeline stall?

**Solution:** When we have to hold an instruction back, so that it doesn’t run. OR: when we insert a NOP into the instruction stream, to solve a hazard.

(e) (5 points) Even though some instructions don’t use some pipeline phases, it’s impossible to “skip” a phase - every instruction still goes through each phase. Explain why this is necessary.
Solution: The phases ahead are almost certainly already in use by other instructions!
2. (a) (5 points) What instruction do we know of that will do useful work in every pipeline phase?

**Solution:** LW

(b) (5 points) Not all instructions use all phases of the MIPS pipeline, but every single one uses IF and ID. Explain why this is true.

**Solution:** We don’t even know what type of instruction it is until ID!

(c) (5 points) Give a sequence of instructions which would have to stall, if executed in a processor without data forwarding.

**Solution:** Answers will vary.

One example:

```
add $s0, $s1,$s2
add $s3, $s0,$s4
```

(d) (10 points) Fill in the table below; give the proper CPU control bits for each of the instructions. Refer to the CPU design, which I’ve included on the last page. For “don’t care” values, you may either write X or 0, but 1 will not be accepted.

For the aluOp, you may use either the operation names (such as ADD) or the encodings (such as 2).

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUsrc</th>
<th>aluOp</th>
<th>bInvert</th>
<th>Branch</th>
<th>Jump</th>
<th>MemWrite</th>
<th>MemRead</th>
<th>MemToReg</th>
<th>RegDst</th>
<th>RegWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORI</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SLT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Solution:** GRADING NOTE: Since it’s so hard to tell the difference between random answers and nearly-correct answers, we will only allow a single incorrect column per instruction; if any instruction has more than 1 error, we will mark the entire instruction incorrect.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUsrc</th>
<th>aluOp</th>
<th>bInvert</th>
<th>Branch</th>
<th>Jump</th>
<th>MemWrite</th>
<th>MemRead</th>
<th>MemToReg</th>
<th>RegDst</th>
<th>RegWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORI</td>
<td>1/2</td>
<td>OR (or 1)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>SLT</td>
<td>0</td>
<td>LESS (or 3)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

NOTE: See the Piazza post I made about the error in the ORI question. We will accept either 1 or 2 as the ALUsrc for that question. We will *not* accept 0 or X.
3. (20 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 first or second input.
Solution:
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.

Assume that the processor has no data forwarding!

(a) (10 points)

1. add $s0, $t7,$a0 __5__
2. add $s1, $t3,$s0 _____
3. add $s2, $s1,$s0 _____
4. add $s3, $s1,$s0 _____
5. add $s0, $s1,$s0 _____
6. add $s0, $s1,$s0 _____

Solution:

add $s0, $t7,$a0 __5__
add $s1, $t3,$s0 __8__ // stall: rt depends on prev inst
add $s2, $s1,$s0 _11__ // stall: rs depends on prev; rs depends on 1st (but doesn't require more stalls, since $s0
value is written-back *earlier* than $s1
add $s3, $s1,$s0 _12__ // no stall: rs,rt have dependencies, but the previous instruction waited long enough that this hazard is already solved
add $s0, $s1,$s0 _13__ // same
add $s0, $s1,$s0 _16__ // stall: rt depends on prev

(b) (10 points)

1. lw $s0, 0($s0) __5__
2. add $s0, $s1,$s2 _____
3. $lw \quad $s1, 0($s1)  

4. $lw \quad $s2, 0($s0)  

5. $add \quad $s1, $s1,$s2  

6. $add \quad $s2, $s1,$s2  

Solution:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$lw \quad $s0, 0($s0)</td>
<td>5</td>
</tr>
<tr>
<td>$add \quad $s0, $s1,$s2</td>
<td>6</td>
</tr>
<tr>
<td>$lw \quad $s1, 0($s1)</td>
<td>7</td>
</tr>
<tr>
<td>$lw \quad $s2, 0($s0)</td>
<td>9</td>
</tr>
<tr>
<td>$add \quad $s1, $s1,$s2</td>
<td>12</td>
</tr>
<tr>
<td>$add \quad $s2, $s1,$s2</td>
<td>15</td>
</tr>
</tbody>
</table>
5. (5 points) **DO THIS QUESTION LAST.** It’s not worth a lot of points.

Convert the hexadecimal value below to a 32-bit number (I suggest that you write it as eight 4-bit nibbles, to make it easy to read.)

Then decode the instruction and give the MIPS assembly language statement that it represents.

Make sure to refer to the tables you have been provided to help you.

8e b7 03 31

---

**Solution:**

Binary: 1000 1110 1011 0111 0000 0011 0011 0001

Opcode: 1000 11 = 0x23 = 35<sub>10</sub> = LW

**Fields:**

<table>
<thead>
<tr>
<th>Field</th>
<th>Binary</th>
<th>Opcode</th>
<th>Rs</th>
<th>Rt</th>
<th>Imm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1000 1110 1011 0111 0000 0011 0011 0001</td>
<td>1000 11</td>
<td>10 101</td>
<td>1 0111</td>
<td>0000 0011 0011 0001</td>
</tr>
</tbody>
</table>

rs: 10 101 = 0x15 = 21 = s5

rt: 1 0111 = 0x1f = 23 = s7

imm: 0x0331

The instruction is thus: `lw $s7, (0x0331)$s5`
<table>
<thead>
<tr>
<th>Register name</th>
<th>Number</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>$zero</td>
<td>0</td>
<td>constant 0</td>
</tr>
<tr>
<td>$at</td>
<td>1</td>
<td>reserved for assembler</td>
</tr>
<tr>
<td>$v0</td>
<td>2</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$v1</td>
<td>3</td>
<td>expression evaluation and results of a function</td>
</tr>
<tr>
<td>$a0</td>
<td>4</td>
<td>argument 1</td>
</tr>
<tr>
<td>$a1</td>
<td>5</td>
<td>argument 2</td>
</tr>
<tr>
<td>$a2</td>
<td>6</td>
<td>argument 3</td>
</tr>
<tr>
<td>$a3</td>
<td>7</td>
<td>argument 4</td>
</tr>
<tr>
<td>$t0</td>
<td>8</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t1</td>
<td>9</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t2</td>
<td>10</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t3</td>
<td>11</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t4</td>
<td>12</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t5</td>
<td>13</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t6</td>
<td>14</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t7</td>
<td>15</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$s0</td>
<td>16</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s1</td>
<td>17</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s2</td>
<td>18</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s3</td>
<td>19</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s4</td>
<td>20</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s5</td>
<td>21</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s6</td>
<td>22</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$s7</td>
<td>23</td>
<td>saved temporary (preserved across call)</td>
</tr>
<tr>
<td>$t8</td>
<td>24</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$t9</td>
<td>25</td>
<td>temporary (not preserved across call)</td>
</tr>
<tr>
<td>$k0</td>
<td>26</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$k1</td>
<td>27</td>
<td>reserved for OS kernel</td>
</tr>
<tr>
<td>$gp</td>
<td>28</td>
<td>pointer to global area</td>
</tr>
<tr>
<td>$sp</td>
<td>29</td>
<td>stack pointer</td>
</tr>
<tr>
<td>$fp</td>
<td>30</td>
<td>frame pointer</td>
</tr>
<tr>
<td>$ra</td>
<td>31</td>
<td>return address (used by function call)</td>
</tr>
</tbody>
</table>

**FIGURE A.6.1** MIPS registers and usage convention.

The stack pointer. The executing procedure uses the frame pointer to quickly access values in its stack frame. For example, an argument in the stack frame can be loaded into register $v0 with the instruction

\[
\text{lw } v0, 0(fsp)
\]
FIGURE A.10.2 MIPS opcode map. The values of each field are shown to its left. The first column shows the values in base 10 and the second column in base 16. The first field defines the operation type, and the second field is the register destination.

- **op(31:26)**: This field specifies the operation type.
- **rs(25:21)**: This field specifies the source register.
- **rt(20:16)**: This field specifies the destination register.
- **func(5:0)**: This field specifies the function of the operation.

Some operations are specified based on the value of the third field (bits 4 to 0): if \( z = 0 \), the operation type is specified in the fourth field; if \( z = 1 \) or \( z = 2 \), the operation type is specified in the fifth field; if \( z = 3 \), the operation type is specified in the sixth field.

The operations are specified as follows:

- **add**: Addition
- **sub**: Subtraction
- **and**: Logical AND
- **or**: Logical OR
- **xor**: Exclusive OR
- **nor**: Logical NOR
- **neg**: Logical NOT
- **s**: Sign extension
- **trunc**: Truncation
- **c.ule**: Carry less than or equal to
- **c.ole**: Carry or less than or equal to
- **c.olt**: Carry or less than
- **c.un**: Carry or not equal
- **funct(5:0)**: Function code

- **move**: Move
- **moveu**: Move unsigned
- **movew**: Move word
- **moveh**: Move half word

- **load**: Load
- **store**: Store

- **shift**: Shift
- **rotate**: Rotate

- **branch**: Branch
- **jump**: Jump

- **other**: Other operations (e.g., syscall, break, etc.)