Pipelining Exercises, Continued

4. Spot all data dependencies (including ones that do not lead to stalls). Draw arrows from the stages where data is made available, directed to where it is needed. Circle the involved registers in the instructions. **Assume no forwarding.** One dependency has been drawn for you.

```
addi $t0 $t1 100   F  D  X  M  W
lw   $t2 4($t0)     F  D  X  M  W
add $t3 $t1 $t2     F  D  X  M  W
sw   $t3 8($t0)     F  D  X  M  W
lw   $t5 0($t6)     F  D  X  M  W
or   $t5 $t0 $t3     F  D  X  M  W
```

5. Redraw the arrows for the above question assuming that **our hardware provides forwarding.**

```
addi $t0 $t1 100   F  D  X  M  W
lw   $t2 4($t0)     F  D  X  M  W
add $t3 $t1 $t2     F  D  X  M  W
sw   $t3 8($t0)     F  D  X  M  W
lw   $t5 0($t6)     F  D  X  M  W
or   $t5 $t0 $t3     F  D  X  M  W
```

6. How many stalls will we have to add to the pipeline to resolve the hazards in Exercise 4? How many stalls to resolve the hazards in Exercise 5?
Instruction Scheduling

Suppose we have an array of structs of this form:

```c
struct point { int x; int y; }
```

We wish to square each member of point and add them to another array:

```c
sum[i] = p[i].x*p[i].x+p[i].y*p[i].y;
```

Suppose the number of points in p is in $a0$, base of p in $a1$, and the base of sum is in $a2$. Then we can perform the operation with this MIPS code:

```mips
compileData:
    beq $a0, $0, exit
    lw $t0, 0($a1)
    lw $t1, 4($a1)
    mul $t0, $t0, $t0
    mul $t1, $t1, $t1
    add $t3, $t0, $t1
    sw $t3, 0($a2)
    addi $a1, $a1, 8
    addi $a2, $a2, 4
    addi $a0, $a0, -1
    j compileData
exit:
```

Exercises:

Assume that you have a dual-issue machine wherein one ALU/branch operation can be scheduled in parallel with a load/store operation. Can you schedule the instructions in the above loop to improve performance?

Unroll the loop by a factor of 2, apply register renaming, and schedule again (you may assume $a0$ is even). How much improvement can be obtained?
Virtual Memory Overview

Virtual address (VA): What your program uses

<table>
<thead>
<tr>
<th>Virtual Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
</table>

Physical address (PA): What actually determines where in memory to go

<table>
<thead>
<tr>
<th>Physical Page Number</th>
<th>Page Offset</th>
</tr>
</thead>
</table>

With 4 KiB pages and byte addresses, $2^{(\text{page offset bits})} = 4096$, so page offset bits = 12.

The Big Picture: Logical Flow

Translate VA to PA using the TLB and Page Table. Then use PA to access memory as the program intended.

Pages

A chunk of memory or disk with a set size. Addresses in the same virtual page get mapped to addresses in the same physical page. The page table determines the mapping.

The Page Table

<table>
<thead>
<tr>
<th>Index = Virtual Page Number (not stored)</th>
<th>Page Valid</th>
<th>Page Dirty</th>
<th>Permission Bits (read, write, ...)</th>
<th>Physical Page Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(Max virtual page number)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each stored row of the page table is called a page table entry (the grayed section is the first page table entry). The page table is stored in memory; the OS sets a register telling the hardware the address of the first entry of the page table. The processor updates the “page dirty” in the page table: “page dirty” bits are used by the OS to know whether updating a page on disk is necessary. Each process gets its own page table.

- **Protection Fault**--The page table entry for a virtual page has permission bits that prohibit the requested operation

- **Page Fault**--The page table entry for a virtual page has its valid bit set to false. The entry is not in memory.
The Translation Lookaside Buffer (TLB)

A cache for the page table. Each block is a single page table entry. If an entry is not in the TLB, it’s a TLB miss. Assuming fully associative:

<table>
<thead>
<tr>
<th>TLB Entry Valid</th>
<th>Tag = Virtual Page Number</th>
<th>Page Table Entry</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Page Dirty</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

The Big Picture Revisited

Exercises

What are three specific benefits of using virtual memory? [there are many]

What should happen to the TLB when a new value is loaded into the page table address register?

x86 has an "accessed" bit in each page table entry, which is like the dirty bit but set whenever a page is used (load or store). Why is this helpful when using memory as a cache for disk?

Fill this table out!

<table>
<thead>
<tr>
<th>Virtual Address Bits</th>
<th>Physical Address Bits</th>
<th>Page Size</th>
<th>VPN Bits</th>
<th>PPN Bits</th>
<th>Bits per row of PT (4 extra bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>32</td>
<td>16KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>26</td>
<td></td>
<td></td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>21</td>
<td>32KB</td>
<td>21</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>25</td>
<td>32KB</td>
<td>25</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>48</td>
<td>32KB</td>
<td>25</td>
<td>28</td>
<td></td>
</tr>
</tbody>
</table>