ECE 3056

Exam II - Solutions
March 31st, 2015
3:00 pm – 4:25pm
1. 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>
2. 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)
<nop>          Time for the first instruction to reach WB = 4 cycles
add $t3, $t3, $t4  4 executions of the loop = 4*9 = 36 cycles
addi $t0, $t0, 4    Last 3 nops are not necessary = -3
addi $t2, $t2, -1   Total = 4+36-3 = 37
bne $t2, $0, loop
<nop>                  Total number of instructions = 4*5 = 20
<nop>                  CPI = 37/20
<nop>
```

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.

```
add $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.
3. Consider the following instruction dependency that can be caused by copying data.

\[
\begin{align*}
\text{lw} & \; $8, \; 0($9) \\
\text{sw} & \; $8, \; 4($10)
\end{align*}
\]

Modify the datapath overleaf 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

\[
\begin{align*}
\text{If} \; ((\text{EX}/\text{MEM}.\text{MemWrite} = '1') \; \text{and} \; (\text{MEM}/\text{WB}.\text{MemToReg} = '1') \; \text{and} \; (\text{EX}/\text{MEM}.\text{WriteRegAddr} = \text{MEM}/\text{WB}.\text{WriteRegAddr})) \\
\text{then} \; \text{Forward} = '1'; \; \# \; \text{forward from WB} \\
\text{else} \; \text{Forward} = '0'; \; \# \; \text{do not forward from WB}
\end{align*}
\]

The first two predicates check if there is a load followed by a store, while the last predicate checks if the \(r^+\) 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 X).
4. 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.
5. 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.

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.