Homework Chapter 6 Part 2
Computer Organization Spring 2002

Due date: 2:30 PM, Monday, 3/18/02, TA responsible: Y. Abbas

Problem 1 (40 points)
Recall that forwarding can be done from a load to a store that follows immediately and stores the same data that was loaded. This is MEM to MEM stage forwarding.

Consider the pipeline design that is given in the text that performs the forwarding checks in the EX stage for EX to EX and MEM to EX forwarding, e.g., Figure 6.40.

Describe how you would modify that pipeline to add MEM to MEM forwarding for a lw followed by a dependent sw.

1. Describe (in words not logic equations) the conditions that must be met for forwarding to occur.
2. Identify the source of the data in the pipeline that is forwarded and the destination to which it is forwarded.
3. Describe any additional combinational logic or control lines that must be added to the pipeline and be specific about what stage the logic is in and from where it gets its inputs.
4. Describe changes to existing control signal settings (if any) that are required.
5. Describe any additional information that must be added to the pipeline registers (be specific about which registers and what information).
6. Write the logical conditions in a form similar to those in the text on pages 480 and 481.

Solution:
There are multiple ways of adding forwarding hardware to solve this problem. We describe one that is consistent with our notion of sending data to the stage where it is finally used, i.e., forwarding when the data is consumed.

So in this case, MEM-to-MEM forwarding corresponds to the situation where the store is in the MEM phase and the load is in the WB phase. Since the load read its data in the previous cycle while in the MEM phase the data is available in the MEM/WB register for use in driving the combinational input to the data memory for the store.

The solution requires adding a MUX to the MEM stage whose output is the write data port of the data memory unit. One input is the data line that already exists from the EX/MEM register that contains the data from register rt, i.e., the normal path from which a store would get its data in a nonforwarding situation. The second input to the MUX is the forwarded data from the load in the WB stage, i.e., a path connecting the MEM/WB.writedata field to the MUX.

In order to control this forwarding we must check various conditions:

1. store in MEM stage AND load in WB stage?
2. source register of store and destination register of load the same?
3. destination of load not register 0?

The problem with these conditions is that they must be implemented using the control lines and data contained in the pipeline registers (see the solution to the earlier homework problem where you had to identify the data lines in the registers and see the text for the control lines).

The problematic comparison is the source for the store. Its register number is not propagated to the MEM phase since all that is needed is the address and the data to be stored. We do not however have to add another line. The destination register line which is set to $rt or $rd depending on the instruction type
in the EX stage is not used for the store (since it has no destination register). We can set the control lines for a store so that $rt$ is propagated through that field into the EX/MEM register. Therefore, $rt$ for the store will be available in the field EX/MEM.writerregister.

To detect a load in the WB stage we must also use alternate signals to those used for forwarding in the textbook discussion, i.e., we only have a specific set of signals in the MEM/WB pipeline register.

So assuming the modification to propagate the store source register in EX/MEM.writerregister as discussed above, the code is

```c
if( (EX/MEM.MemWrite = 1) # sw in MEM
    and
    ((MEM/WB.MemToReg = 1) and (MEM/WB.RegWrite = 1)) # lw in WB
    and
    (EX/MEM.writerregister = MEM/WB.writerregister) # register dependence
    and
    (MEM/WB.writerregister not = 0)) # but not reg 0 ?
then
    Mux = 1  # take forwarded lw value
else
    Mux = 0  # take value on normal path
end if
```

We also have to make sure that the load-store dependence that is covered by forwarding does not cause incorrect operation of the stall generation. The current code for hazard detection that concentrates on load-use dependences is:

```c
if( (ID/EX.MemRead = 1)
    and
    ((ID/EX.RegisterRt = IF/ID.RegisterRs)
     or
     (ID/EX.RegisterRt = IF/ID.RegisterRt))
then
    stall pipeline
end if
```

For the code in this problem, this will generate a stall since it detects a load in the EX stage and a destination register of the load identical to either source register of the instruction in the ID stage.

If the $rs$ for the store is the same as the $rt$ for the load then the address computation of the store done in the EX stage needs the load data and a stall is required.

However, if the address does not depend on the data loaded and the instruction in the ID stage is a store then $rt$ for the store can be the same as $rt$ for the load because the new forwarding unit above will remove the need for a stall.
The new code that adds a condition to suppress the stall is as follows:

```c
if(
    (ID/EX.MemRead = 1)
    and
    (ID/EX.RegisterRt = IF/ID.RegisterRs)
    or
    (ID/EX.RegisterRt = IF/ID.RegisterRt)
) and
not(
    (IF/ID.MemWrite = 1)
    and
    (ID/EX.RegisterRt not = IF/ID.RegisterRs)
)
then
    stall pipeline
end if
```

Note that we have used a new signal IF/ID.MemWrite. Since control signals are not available until after the ID phase, this signal must be generated from the op code in the IF/ID register by detecting the op code of a store.

**Problem 2 (40 points)**

The forwarding logic that is described in the text and the notes for EX to EX and MEM to EX forwarding is such that the logic is placed in the EX stage of the instruction that consumes the forwarded data.

It is also possible to put the logic in the ID stage. This logic would produce the control signals needed to cause forwarding when the consuming instruction reached the EX stage.

Consider the pipeline design that is given in the text that performs the forwarding checks in the EX stage for EX to EX and MEM to EX forwarding, e.g., Figure 6.40.

Describe how you would modify that pipeline to make decisions for forwarding to the EX stage of a consuming instruction during the ID stage of the consuming instruction. **Note that the forwarding still occurs during the EX stage of the consuming instruction.** The signals needed to make it happen are produced when the consuming instruction is in its ID stage.

1. Describe (in words not logic equations) the conditions that must be met for forwarding to occur.
2. Identify the sources of the data in the pipeline that is forwarded and the destinations to which it is forwarded.
3. Describe any additional combinational logic or control lines that must be added to the pipeline and be specific about what the logic does and from where it gets its inputs.
4. Describe changes to existing control signal settings (if any) that are required.
5. Describe any additional information that must be added to the pipeline registers (be specific about which registers and what information).
6. Write the logical conditions in a form similar to those in the text on pages 480 and 481.

**Hint:**

The pipeline diagram shows the relationship between the consuming instruction and the three preceding instructions in the code that may have as their destination one of the source registers of the consumer, i.e., they are producers.
Consider this table and decide which of the instructions must be considered when generating the logic for ID stage forwarding decisions and which, if any, do not. Also note the stages in which the producers are when the consumer is in the ID stage and the EX stage. This should help decide where data is located for the decision making and for the actual forwarding.

**Solution:**

The forwarding decision must be made when the consumer instruction is in the ID stage. The signals ForwardA and ForwardB will be evaluated and placed in the ID/EX register for use on the next cycle when the consumer is in its EX stage and the data must actually be forwarded (so we must add two fields to the ID/EX register).

The timing diagram shows the relationship between the consumer and the three preceding instructions in the code that may have as their destination one of the source registers of the consumer.

| producer 3 | IF | ID | EX | ME | WB |
| producer 2 | IF | ID | EX | ME | WB |
| producer 1 | IF | ID | EX | ME | WB |
| consumer   | IF | ID | EX | ME | WB |

We first consider the location in the pipeline of the producers when the consumer is in the EX stage. This identifies which instructions may have to forward and where the data that must be forwarded will be. When the consumer is in the EX stage, the data from producer 3 can be read in the second half of the cycle since producer 3 is in its WB stage. No forwarding is needed and we need not consider forwarding from producer 3 or any other instruction earlier in the instruction stream.

Data from producer 2 will be in the MEM/WB register when the consumer is in the EX stage. So ForwardA and/or ForwardB should be set to 01 as appropriate to forward data to the ALU from the MEM/WB register. However, these lines must be set during the ID stage of the consumer and therefore we must use the information on producer 2 that is, at that time, contained in the EX/MEM register.

The ID stage logic

```bash
if(
  (EX/MEM.RegWrite = 1)
  and
  (IF/ID.RegisterRs = EX/MEM.RegisterRd)
)
then
  ID/EX.ForwardA = 01
end if

if(
  (EX/MEM.RegWrite = 1)
  and
  (IF/ID.RegisterRt = EX/MEM.RegisterRd)
)
then
  ID/EX.ForwardB = 01
end if
```

(The complication of the load to register 0 is ignored here.) The data on these lines will be latched into the ID/EX pipeline register at the next clock trigger and will be used to drive the combination logic that chooses the data from the MEM/WB pipeline register.
Data from producer 1 will be in the EX/MEM register when the consumer is in the EX stage. So ForwardA and/or ForwardB should be set to 10 as appropriate to forward data to the ALU from the EX/MEM register. The code that sets these lines during the ID stage of the consumer must use the information on producer 1 that is, at that time, contained in the ID/EX register.

The logic is

```plaintext
if(
  (ID/EX.RegWrite = 1)
  and
  (IF/ID.RegisterRs = ID/EX.RegisterRd)
)
then
  ID/EX.ForwardA = 10
end if

if(
  (ID/EX.RegWrite = 1)
  and
  (IF/ID.RegisterRt = ID/EX.RegisterRd)
)
then
  ID/EX.ForwardB = 10
end if
```

As noted above, the lines ID/EX.ForwardA and ID/EX.ForwardB must be added to the pipeline register. The updates, of course, to the ID/EX lines would be clocked in at the next leading clock edge and do not overwrite the values that are controlling forwarding in the EX stage during the same cycle the forwarding decision for the consumer is being made in the ID stage, i.e., any forwarding to the instruction that we have labelled producer 1 above.

Of course, additional logic must be added, as in the text on page 483 to handle the situation when producers 1 and 2 both update the same source register for the consumer. We leave that as a further exercise.

**Problem 3 (20 points)**

Consider the following code:

```plaintext
and  $15,$13,$11
addu $4,$2,$7
ori $2,$5,160
lw $2,0($2)
lb $5,0($2)
subu $3,$5,$4
addiu $9,$10,1
```

1. Identify any dependences in the code. Which, if any, dependences do not cause stalls due to forwarding? Which, if any, dependences cause stalls even with forwarding?

2. Fill in a pipeline diagram like the one given in the hint in problem 2.

**Solution:**

```plaintext
and  $15,$13,$11
addu $4,$2,$7
```
ori: $2, $5,160
lw: $2, 0($2)
lb: $5, 0($2)
subu: $3, $5, $4
addiu: $9, $10, 1

The *addu* must consume $2 before the *ori* produces it. Since the read of the *addu* occurs before any possible update of $2 by the *ori* this dependence is not a problem.

The dependence via $2 from the *ori* to the *lw* address computation does not cause a stall if EX-to-EX forwarding is used.

The dependence via $2 from the *lw* to the *lb* address computation does a stall even with MEM-to-EX forwarding. The dependence via $5 from the *lb* to the *subu* does a stall even with MEM-to-EX forwarding. (These two are essentially load-use stalls.)

There are no other dependences. The pipeline diagram assuming stalling in the ID stage is below. Cycles that are either the source or destination of a forwarding operation as identified above are in lower case.

<table>
<thead>
<tr>
<th>cycle</th>
<th>and</th>
<th>addu</th>
<th>ori</th>
<th>lw</th>
<th>lb</th>
<th>subu</th>
<th>addiu</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>ID</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>EX</td>
<td>ID</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>ME</td>
<td>EX</td>
<td>ID</td>
<td>IF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>WB</td>
<td>ME</td>
<td>ex</td>
<td>ID</td>
<td>IF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>WB</td>
<td>ME</td>
<td>ex</td>
<td>ID</td>
<td>IF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>