CS152 Homework II, Fall 2006

Name: ____________________
SSID: ____________________

Homework II is due in class on Thursday November 30th at 11:10 AM. This class is the Mid-term II review session.

Late homeworks are NOT accepted. Thus, if you will not be attending the review session, you MUST make arrangements to hand off the homework to the instructor before class time.

Homework will be graded on effort (did you make an honest attempt to solve each problem?), not correctness. We will distribute the correct answers for the homework in the review session, but we will probably not return the homework you hand in until after the exam. So, you may wish to make a copy for reference before you hand it in.

This homework will count for approximately 1.5% of your final grade. The homework is based on the Mid-term II exam from Fall 05.
You may discuss the homework problems with fellow students and the TAs, but what you write down must be your own work (no copying the answers from someone else’s homework). Good luck! John Lazzaro
1 Multithreading (10 points)

In class, we showed a 4-way static multithreading architecture. Below we show a variant of this architecture, that supports 2 threads instead of 4 (note the thread select line is only 1 bit wide). The architecture uses load delay and branch delay slots; branch comparison is done in the ID stage, so that control hazards do not occur.

To prevent RAW data hazards in this datapath, it is necessary to add forwarding paths. Thus, we have added the two muxes labelled Fwd. Draw in all NECESSARY forwarding paths to the FWD muxes to handle data hazards. ONLY draw in the necessary forwarding paths; DO NOT draw in forwarding paths that are not needed to prevent RAW data hazards. Points will be taken off for each unnecessary forwarding path drawn.

[Diagram showing necessary forwarding paths marked with Fwd]
2 Write-Back Caches (22 points)

The top slide on the next page shows a 2-way set-associative cache. The cache is a write-back cache (see slide below for a reminder). Each line of each set of the cache holds one word of data (32 bits).

If a read misses the cache, and the cache line for the address has unused sets \((V = 0)\), read data is placed in an unused set of the cache line. Its \(V\) bit is set to 1, and its L bit is updated.

If the cache line does not have unused sets, the cache uses a least-recently used replacement policy, coded by an L bit for each index (see slides on next page for the encoding of the bit).

A read or a write that uses a set updates the L bit, but an invalidate (setting \(V = 0\)) of a set does not update the L bit. Writes that miss the cache do not allocate a line in the cache (i.e. a “no write allocate” cache).

The top slide on the next page shows the initial values of the state elements in the cache. The top slide also shows the initial values of 12 words in main memory. After the state snapshot shown in this slide, the following MIPS memory commands are executed:

\[
\begin{align*}
\text{SW} & \quad \text{R0} & \quad 16(\text{R0}) \\
\text{LW} & \quad \text{R20} & \quad 20(\text{R0}) \\
\text{LW} & \quad \text{R21} & \quad 24(\text{R0}) \\
\text{LW} & \quad \text{R22} & \quad 12(\text{R0}) \\
\text{SW} & \quad \text{R0} & \quad 0(\text{R0})
\end{align*}
\]

Executing the program may change cache state and main memory values. **For 22 points**, fill in all changed values (cache data, tag fields, cache V and L bits, and main memory) after execution of all commands, on the bottom slide on the next page.

---

### From lecture: Cache policies defined.

<table>
<thead>
<tr>
<th>Policy</th>
<th>Write-Through</th>
<th>Write-Back</th>
<th>Related issue do writes to blocks not in the cache get put in the cache (“write allocate”) or not (“no write allocate”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do read misses produce writes?</td>
<td>No</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Do repeated writes make it to lower level?</td>
<td>Yes</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

**This exam question:** a “write-back” and “no write allocate” cache!
2-way set associative write-back data cache

L: LRU bit.  L = 1 indicates left set (as drawn on page) has been read or written most recently.
L = 0 indicates right set has been read or written most recently. Setting V=0 does not update L.

Cache Tag (28 bits)

Index

00

Cache Data (decimal)

Cache Tags (hex)

Valid (V)

Left most recent (L)

Right

Valid (V)

Cache Data (decimal)

Ex: 0x01

Hit

Left

Hit

Right

1 word

28 bits

Fill in CHANGED cache and main memory fields

L: LRU bit.  L = 1 indicates left set (as drawn on page) has been read or written most recently.
L = 0 indicates right set has been read or written most recently. Setting V=0 does not update L.

Cache Tag (28 bits)

Index

00

Cache Data (decimal)

Cache Tags (hex)

Valid (V)

Left most recent (L)

Main Memory

Addr in hex, values in decimal

Addr

Value

Addr

Value

Addr

Value

Addr

Value

0x00000000

0x00000004

0x00000008

0x0000000C

0x00000010

0x00000014

0x00000018

0x0000001C

0x00000020

0x00000024

0x00000028

0x0000002C

0x00000000

0x00000004

0x00000008

0x0000000C

0x00000010

0x00000014

0x00000018

0x0000001C

0x00000020

0x00000024

0x00000028

0x0000002C
3 Hamming Codes (ECC) (14 points)

In the slide below, we show a design for a single-error correction Hamming code for 11 data bits, protected by 4 parity bits. At the bottom of the slide, we show the equation for computing the parity bit $P_o$.

In the table on the slide, fill in the logic equations for computing the parity bits $P_1$, $P_2$, and $P_3$. Each equation is worth 4.66 points. Only the equations written in the table boxes will be graded. Please write subscripts clearly.

Finish specifying this Hamming Code

$P_3P_2P_1P_0$

Fill in the equations for $P_1$, $P_2$, $P_3$

$P_0 = D_{10} \ xor \ D_8 \ xor \ ... \ xor \ D_0$

$P_3 = D_{10} \ xor \ D_9 \ xor \ D_8 \ xor \ D_7 \ xor \ D_6 \ xor \ D_5 \ xor \ D_4$

$P_2 = D_{10} \ xor \ D_9 \ xor \ D_8 \ xor \ D_7 \ xor \ D_3 \ xor \ D_2 \ xor \ D_1$

$P_1 = D_{10} \ xor \ D_9 \ xor \ D_8 \ xor \ D_7 \ xor \ D_5 \ xor \ D_0$

$P_0 = D_{10} \ xor \ D_8 \ xor \ D_6 \ xor \ D_4 \ xor \ D_3 \ xor \ D_1 \ xor \ D_0$

Use this word bit arrangement

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>D_{15}</td>
<td>D_{14}</td>
<td>D_{13}</td>
<td>D_{12}</td>
<td>D_{11}</td>
<td>D_{10}</td>
<td>D_9</td>
<td>D_8</td>
<td>D_7</td>
<td>D_6</td>
<td>D_5</td>
<td>D_4</td>
<td>P_3</td>
<td>P_2</td>
<td>P_1</td>
</tr>
</tbody>
</table>

$C_3C_2C_1C_0$ signals the flipped bit position.
4 Branch Prediction (18 points)

Below are slides that summarize the simple branch predictor we described in class.

![Simple branch predictor block diagram](image)

**Simple branch predictor block diagram**

Address of BNEZ instruction: 0b0110[...01001000]

BNEZ R1 Loop

Branch Target Buffer (BTB):
- 28-bit address tag: 0b0110[...0100]
- 2 bits: 0b01
- Target address: PC + 4 + Loop

Branch History Table (BHT):
- N: Prediction for Next branch: 1 = take, 0 = not take
- L: Was Last prediction correct?: 1 = yes, 0 = no

Branch History State:

<table>
<thead>
<tr>
<th>old N</th>
<th>old L</th>
<th>branch</th>
<th>new N</th>
<th>new L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>not taken</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>taken</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>not taken</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>taken</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>not taken</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>taken</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>not taken</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>taken</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

When replacing the tag value for a line, initialize branch history state to (N = 1, L = 1) for taken branches or to (N = 0, L = 1) for "not taken" branches.

Hit

On a miss, replace BTB for the line with the new branch tag & target. Next slide defines initial BHT N and L.
Assume a program is running on the machine. At some point, we stop program execution, and take a snapshot of the contents of the branch prediction hardware (top part of the bottom slide). The program then resumes. The next 7 branches executed by the program are shown in the top slide below. After these branches complete, we stop the program again.

Fill in the updated state of the branch prediction hardware in the bottom part of the bottom slide.

### Trace of branch instructions

<table>
<thead>
<tr>
<th>Execution order</th>
<th>Address of branch</th>
<th>Branch opcode</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x 0000 0000</td>
<td>BEQ R1 R2 Lab1</td>
<td>Taken</td>
</tr>
<tr>
<td>2</td>
<td>0x 0000 0034</td>
<td>BEQ R7 R8 Lab4</td>
<td>Not Taken</td>
</tr>
<tr>
<td>3</td>
<td>0x 0000 006C</td>
<td>BEQ R13 R14 Lab7</td>
<td>Not Taken</td>
</tr>
<tr>
<td>4</td>
<td>0x 0000 0058</td>
<td>BEQ R11 R12 Lab6</td>
<td>Taken</td>
</tr>
<tr>
<td>5</td>
<td>0x 0000 0020</td>
<td>BNE R5 R6 Lab3</td>
<td>Taken</td>
</tr>
<tr>
<td>6</td>
<td>0x 0000 0034</td>
<td>BEQ R7 R8 Lab4</td>
<td>Taken</td>
</tr>
<tr>
<td>7</td>
<td>0x 0000 006C</td>
<td>BEQ R13 R14 Lab7</td>
<td>Not Taken</td>
</tr>
</tbody>
</table>

### Branch predictor state before first inst. in trace executes

<table>
<thead>
<tr>
<th>28-bit address tag</th>
<th>target address</th>
<th>N</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 0000 000</td>
<td>PC + 4 + Lab1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x 0000 003</td>
<td>PC + 4 + Lab4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0x 0000 005</td>
<td>PC + 4 + Lab6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0x 0000 007</td>
<td>PC + 4 + Lab8</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

### Fill in ALL underlines ("_"), and ALL N and L boxes.

<table>
<thead>
<tr>
<th>28-bit address tag</th>
<th>target address</th>
<th>N</th>
<th>L</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x 0000 002</td>
<td>PC + 4 + Lab3</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0x 0000 003</td>
<td>PC + 4 + Lab4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x 0000 005</td>
<td>PC + 4 + Lab6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0x 0000 006</td>
<td>PC + 4 + Lab7</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
5 Router Switch Arbitration (12 points)

In class, we showed the switching fabric of an Internet router. The slide below, and the two slides on the next page, are from the lecture, to remind you how the switching fabric works. The actual question follows these slides ...

A pipelined arbitration system decides how to connect up the switch. The connections for the transfer at epoch N are computer in epochs N-3, N-2 and N-1, using dedicated switch allocation wires.

What if two inputs want the same output?

- **Inputs**
  - A
  - B
  - C
  - D

- **Outputs**
  - A
  - B
  - C
  - D

- **Line**
- **Engine**
A complete switch transfer (4 epochs)

Epoch 1: All input ports ready to send data request an output port.

Epoch 2: Allocation algorithm decides which inputs get to write.

Epoch 3: Allocation system informs the winning inputs and outputs.

Epoch 4: Actual data transfer takes place.

Allocation is pipelined: a data transfer happens on every cycle, as does the three allocation stages, for different sets of requests.

Allocator examines top array ....

<table>
<thead>
<tr>
<th>Input Ports (A, B, C, D)</th>
<th>Output Ports (A, B, C, D)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 0 0 1 0</td>
<td>A 0 0 1 0</td>
</tr>
<tr>
<td>B 1 0 0 1</td>
<td>B 1 0 0 0</td>
</tr>
<tr>
<td>C 0 1 0 0</td>
<td>C 0 1 0 0</td>
</tr>
<tr>
<td>D 1 0 1 0</td>
<td>D 0 0 0 0</td>
</tr>
</tbody>
</table>

A 1 codes that an input has a packet ready to send to an output. Note an input may have several packets ready.

Allocator returns a matrix with at most one 1 in each row and column to set switches. Algorithm should be "fair"; so no port always loses ... should also "scale" to run large matrices fast.
Question. The slide below shows an allocator input matrix for a switch with 5 inputs and 5 outputs (A/B/C/D/E). However, unlike the class example, an input can specify each packet transfer as “high-priority” or “low-priority”, by using the number “2” or “1” in the array. The top of the slide shows an example input array to the allocator.

On the bottom left, fill in an allocation answer that maximizes the number of high-priority packet transfers. There is no need to draw in the 0s, just the 1s are OK. Remember that, at most, each row and each column may have one “1” in it (if a port output is unused, it may have no “1” in its column). Thus, each input port may send a packet to at most one output port, and each output port may receive a packet from at most one input port.

On the bottom right, fill in an allocation answer that maximizes the total number of packet transfers (irregardless of packet priority).
6 Reorder Buffer Operation (22 points)

The following MIPS machine language program is to be run out of order:

7: ADD R3 R1 R2
8: SUB R3 R3 R1
9: ADD R4 R2 R3
10: SUB R5 R3 R4
11: SUB R5 R3 R5

The top slide of the next page shows the initial state of the reorder buffer structure shown in class, after issue logic has set up the buffer to execute instructions 7, 8, 9, and 10.

Question 6a (3 points). By examining the issue logic setup, fill in the values of the architected registers below, at the moment BEFORE instruction 7 executes:

R1 = 10  R2 = 20

Question 6b (16 points). Assume the execution engine executes instructions 7-10. Fill in all columns for lines 7, 8, 9, and 10, showing the final state in the reorder buffer after all instructions have executed. Your answer should assume that completion hardware has NOT removed any of the instructions from the buffer. You only need to fill in state values that have been changed by the execution engine.
Reorder Buffer: Initial Values ...

First instr to "commit", (complete).
Add: \( *d = *1 + *2 \); Sub: \( *d = *1 - *2 \)

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{Inst \#} & \text{Op} & \text{U} & \text{E} & \#1 & \#2 & \#d & \text{P1} & \text{P2} & \text{Pd} & \text{P1 value} & \text{P2 value} & \text{Pd value} \\
\hline
7 & ADD & 1 & 0 & 01 & 02 & 03 & 1 & 1 & 0 & 10 & 20 & 31 \\
8 & SUB & 1 & 0 & 03 & 01 & 13 & 0 & 1 & 0 & 32 & 10 & 40 \\
9 & ADD & 1 & 0 & 02 & 13 & 04 & 1 & 0 & 0 & 20 & -33 & 12 \\
10 & SUB & 1 & 0 & 13 & 04 & 05 & 0 & 0 & 0 & 32 & 32 & -16 \\
\hline
\end{array}
\]

Next inst in program goes here. 
Physical register numbers
Valid bits for values
Physical register values (in decimal)

Fill in row 7-10 column changed values

Next instr to "commit", (complete).
Add: \( *d = *1 + *2 \); Sub: \( *d = *1 - *2 \)

\[
\begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{Inst \#} & \text{Op} & \text{U} & \text{E} & \#1 & \#2 & \#d & \text{P1} & \text{P2} & \text{Pd} & \text{P1 value} & \text{P2 value} & \text{Pd value} \\
\hline
7 & ADD & 1 & 1 & 01 & 02 & 03 & 1 & 1 & 1 & 30 \\
8 & SUB & 1 & 1 & 03 & 01 & 13 & 1 & 1 & 1 & 30 & 20 \\
9 & ADD & 1 & 1 & 02 & 13 & 04 & 1 & 1 & 1 & 20 & 20 & 40 \\
10 & SUB & 1 & 1 & 13 & 04 & 05 & 1 & 1 & 1 & 20 & 40 & -20 \\
\hline
\end{array}
\]

Add next inst, in program order. 
Physical register numbers
Valid bits for values
Physical register values (in decimal)
Question 6c (3 points). After instructions 7-10 have executed, the issue logic places instruction 11 (shown on the first page of this question) in the buffer. Fill in line 11 in the slide below to show how the issue logic fills the line. Use the naming convention that we used in Question 7a (i.e. architected register R7 uses the series of physical registers PR07, PR17, PR27, etc) and assume the issue logic knows the current values of all physical registers you computed in the previous part of the question. Your answer should show the state values after the issue logic has filled the line, but before execution begins.