CS152 Homework II Spring 2014

Homework II is a self-study guide, to test your knowledge of a few of the topics covered in the lectures from 2/18 to 4/3.

You will not be handing in this homework (and thus, it won't be graded). During the in-class mid-term review session, we will go over the homework questions; during that time frame, we will also post the homework solutions on the class website.

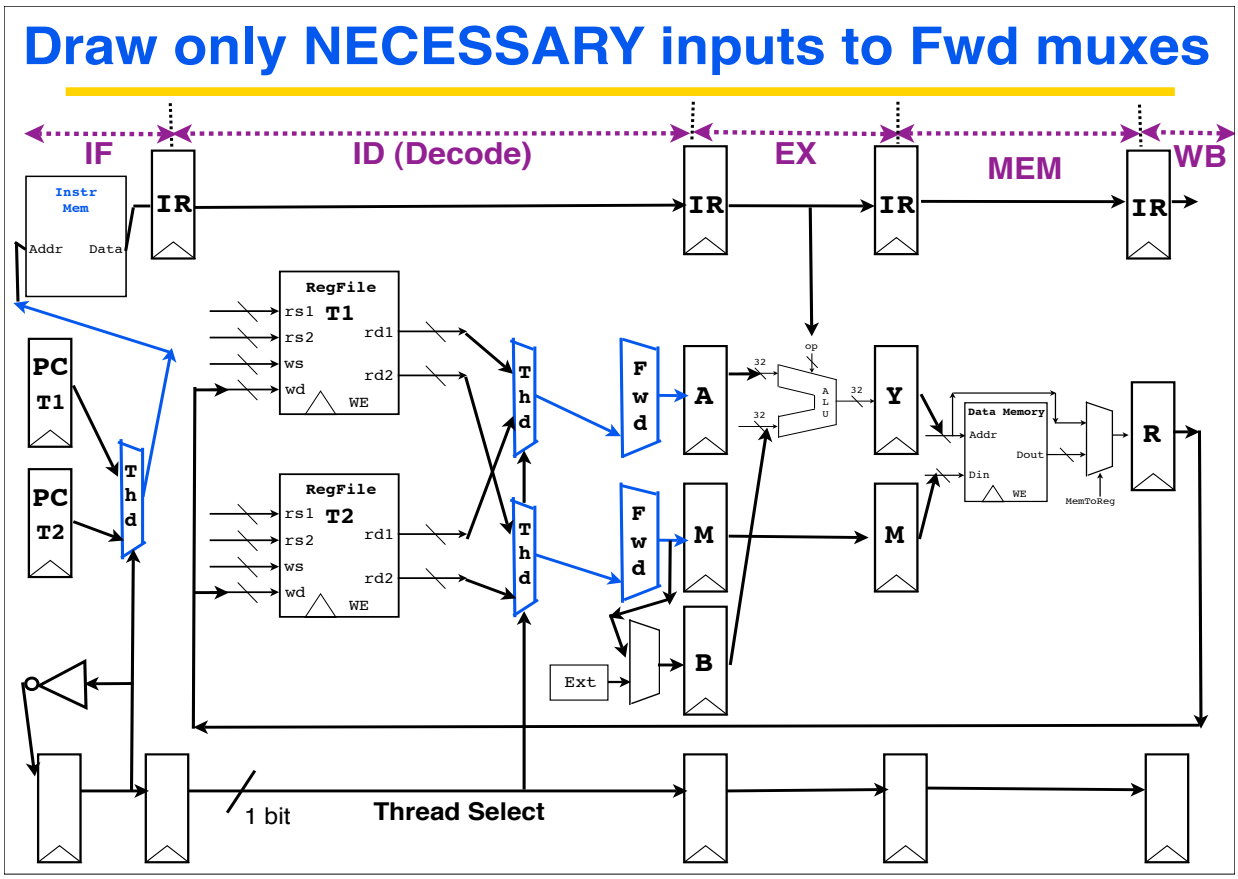
It is based on materials from several exams from previous versions of CS 152 I have taught. Many of the topics I covered in those versions of the class are not being taught this semester (examples: error-correcting codes, disk drive design). As a consequence, this homework is shorter than Homework I.

It is acceptable to ask the sort of "clarifying questions" about homework problems that you might ask during the exam on Piazza, but do not ask (or give) "the answer" online, because that defeats the purpose of the exercise. Written by John Lazzaro.

1 Multithreading

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. Do not add unnecessary forwarding paths.



2 Write-Back Caches

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:

```
SW   R0   16(R0)
LW   R20  20(R0)
LW   R21  24(R0)
LW   R22  12(R0)
SW   R0   0(R0)
```

Executing the program may change cache state and main memory values. For this problem, 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.

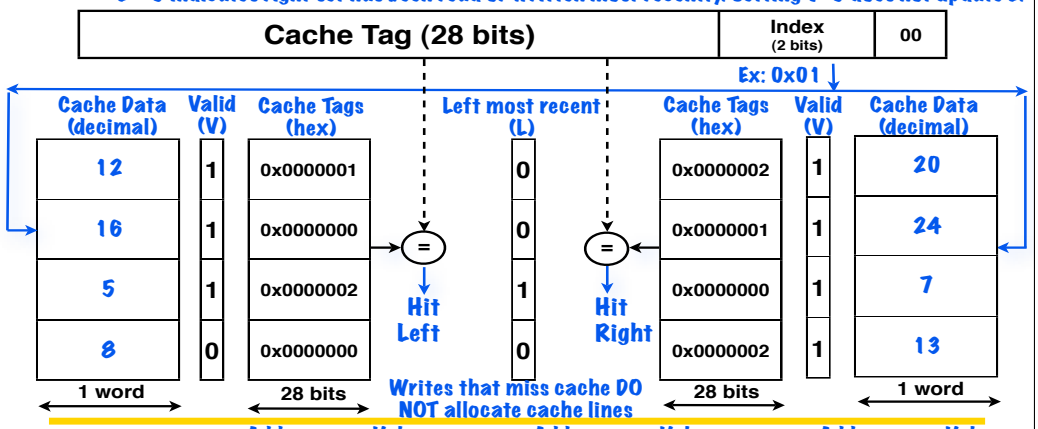
	Write-Through	Write-Back
Policy	Data written to cache block also written to lower-level memory	Write data only to the cache Update lower level when a block falls out of the cache
Do read misses produce writes?	No	Yes
Do repeated writes make it to lower level?	Yes	No

Related issue: do writes to blocks not in the cache get put in the cache ("write allocate") or not ("no write allocate")

This exam question: a "write-back" and "no write allocate" cache!

Initial values of cache and main memory

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.



Writes that miss cache DO NOT allocate cache lines

Main Memory

Addr in hex, values in decimal

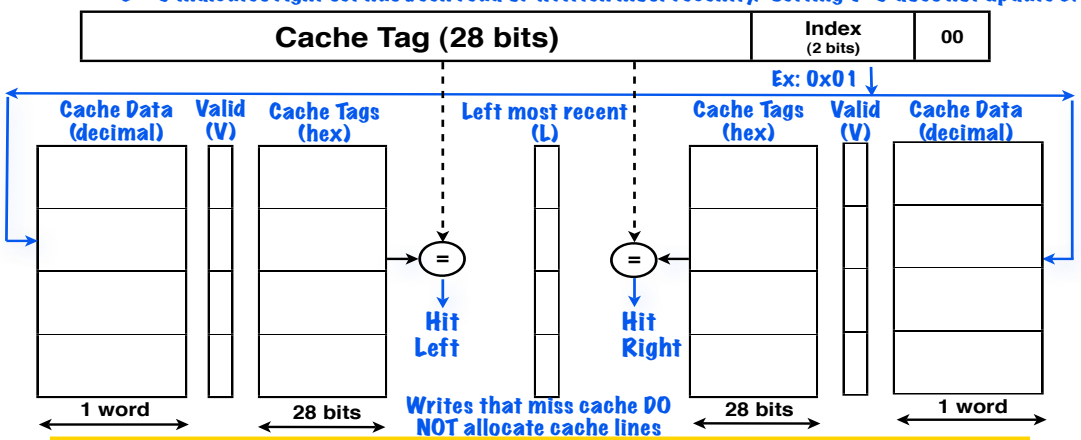
Addr	Value
0x00000000	12
0x00000004	24
0x00000008	5
0x0000000C	14

Addr	Value
0x00000010	12
0x00000014	16
0x00000018	3
0x0000001C	13

Addr	Value
0x00000020	20
0x00000024	16
0x00000028	7
0x0000002C	15

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.



Writes that miss cache DO NOT allocate cache lines

Main Memory

Addr in hex, values in decimal

Addr	Value
0x00000000	
0x00000004	
0x00000008	
0x0000000C	

Addr	Value
0x00000010	
0x00000014	
0x00000018	
0x0000001C	

Addr	Value
0x00000020	
0x00000024	
0x00000028	
0x0000002C	

3 Set-associative Caches

For an N-way set-associative cache, with K cache lines, and B bytes in a cache block (i.e. in a 4-way set-associative cache, each line has 4 blocks for a total of 4B cache data bytes), derive a general-purpose equation for the fraction of total cache RAM bits (defined as tag bits + block bits + valid bits) that is devoted to tag RAM bits.

The equation should take the form $\frac{1}{1+f(K,B)}$. Draw a box around your final equation.

4 Cache Debugging

We fabricate a processor with separate instruction and data caches, and with no address translation hardware (thus, like your class CPU, programs run in physical address space). The data cache is write-thru, direct-mapped, and is not allocate-on-write. The data cache has 28-bit cache tags and 4 cache lines. Thus, each line stores a single word.

After the chip goes out to fab, we realize we made an error in the design, and the data cache valid bits are permanently stuck at "1". The processor has no special cache-control instructions, just normal LW and SW instructions. We must assume that on power-up, the cache tag and cache block RAM for each line will hold arbitrary values. However, the RAMs works correctly – whatever value it holds remains until the cache state machine changes it. (questions begin on next page).

Question 2a The processor starts reading instructions from location 0x000000, in which we can put a program to execute at power up. Write a program to place at address location 0x00000000 that is guaranteed to fill the data cache with data that matches the data held in (fully populated) main memory. Your program may only use LW instructions. A known solution uses 8 LW instructions.

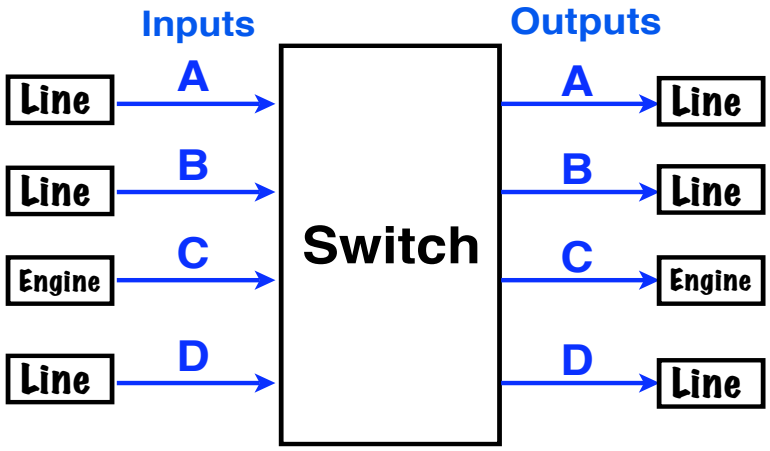
Question 2b Assume this bug (stuck V bits) happened in your direct-mapped instruction cache. Like the data cache, the instruction cache has 28-bit cache tags and 4 cache lines. The processor starts reading instructions from location 0x000000, in which we can put a program to execute at powerup. However, all reads (including the first) go through the instruction cache (as the designers assumed the V bits would be cleared at startup, not set!).

If you are very unlucky, when your machine powers up, the instruction cache tag and cache data RAMs will hold values that will make it impossible for you to get control of the machine. Describe example values of the instruction cache RAMs that would produce this problem (there are many: just one example will suffice).

5 Router Switch Arbitration

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 ...

What if two inputs want the same output?



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.

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 ....

Output Ports (A, B, C, D)

Input Ports (A, B, C, D)

	A	B	C	D
A	0	0	1	0
B	1	0	0	1
C	0	1	0	0
D	1	0	1	0

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

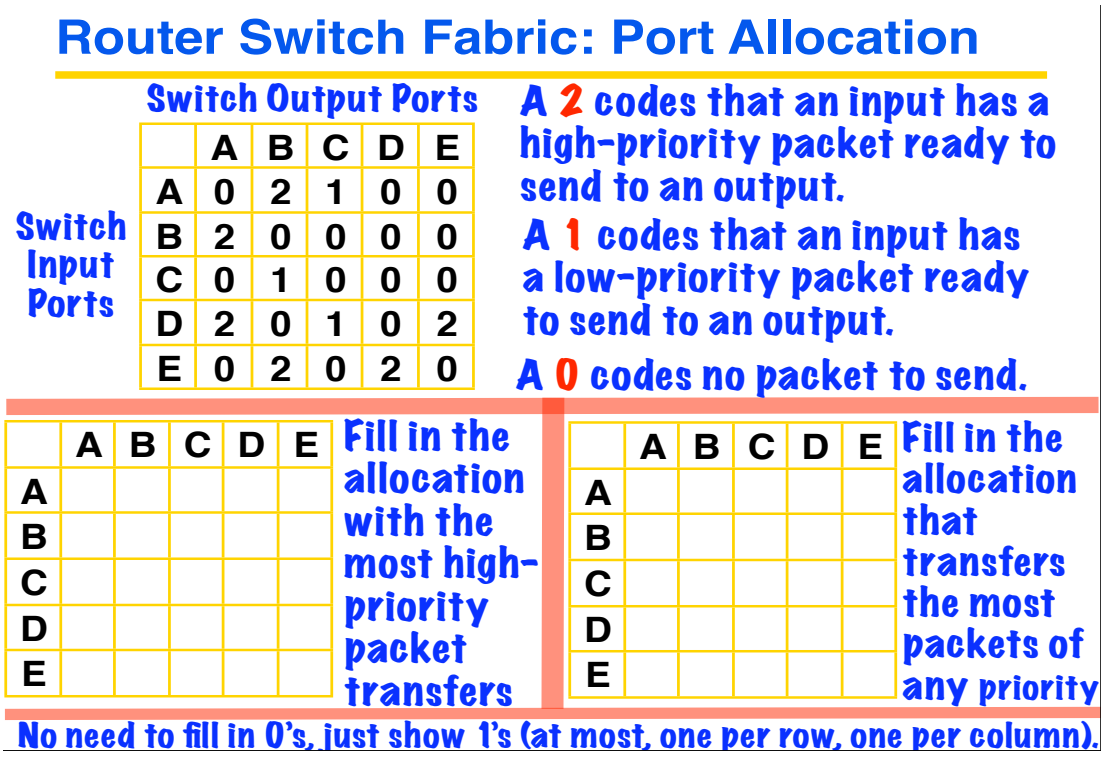
Allocator returns a matrix with at most one 1 in each row and column to set switches. Algorithm should be "fair", so no port always loses ... should also "scale" to run large matrices fast.

	A	B	C	D
A	0	0	1	0
B	1	0	0	0
C	0	1	0	0
D	0	0	0	0

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).



	A	B	C	D	E
A	0	2	1	0	0
B	2	0	0	0	0
C	0	1	0	0	0
D	2	0	1	0	2
E	0	2	0	2	0

6 Reorder Buffer Operation

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

```
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. By examining the issue logic setup, fill in the values of the architected registers below, at the moment BEFORE instruction 7 executes:

```
R1 =            R2 =
```

Question 6b. 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

Use bit (1 if line is in use)

Execute bit (0 if waiting ...)

Tail of List

Head of List

Inst #	Op	U	E	#1	#2	#d	P1	P2	Pd	P1 value	P2 value	Pd value
		0										
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
		0										
		0										

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

Use bit (1 if line is in use)

Execute bit (0 if waiting ...)

Tail of List

Head of List

Inst #	Op	U	E	#1	#2	#d	P1	P2	Pd	P1 value	P2 value	Pd value
		0										
7	ADD	1		01	02	03						
8	SUB	1		03	01	13						
9	ADD	1		02	13	04						
10	SUB	1		13	04	05						
		0										
		0										

Add next inst, in program order.

Physical register numbers

Valid bits for values

Physical register values (in decimal)

Question 6c. 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.

Add Inst #11 line for: SUB R5 R3 R5

Note: R5 and R3 are architected reg. names!

Add: #d= #1 + #2; Sub: #d= #1 - #2

Use bit (1 if line is in use)

Execute bit (0 if waiting ...)

Inst #	Op	U	E	#1	#2	#d	P1	P2	Pd	P1 value	P2 value	Pd value
7	ADD	1	1									
8	SUB	1	1									
9	ADD	1	1									
10	SUB	1	1									
11		1	0						0			-10

Note: SUB uses MIPS syntax: R5 = R3 - R5

Physical register numbers

Valid bits for values

Physical register values (in decimal)