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.

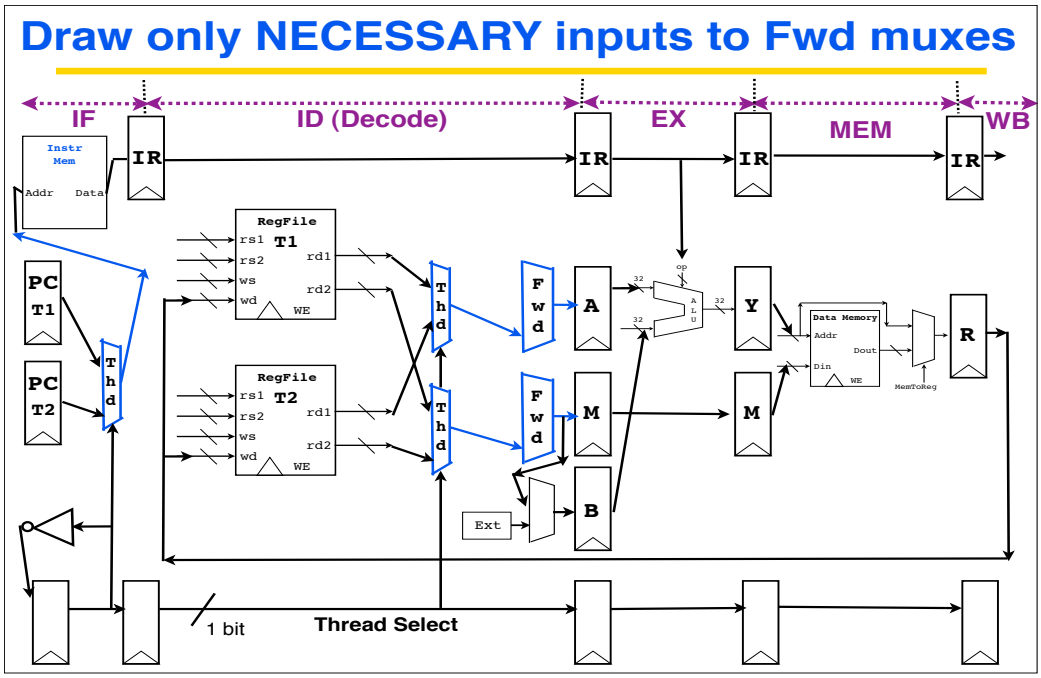
This homework will count for approximately 1.5% of your final grade. The homework is based on the Mid-term II exam from Fall 05. You may discuss the homework problems with fellow students and the TAs, but what you write down must be your own work (no copying the answers from someone else's homework).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

#	Points	
1	12	
2	22	
3	14	
4	18	
5	12	
6	22	
Tot	100	

1 Multithreading (12 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.



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.

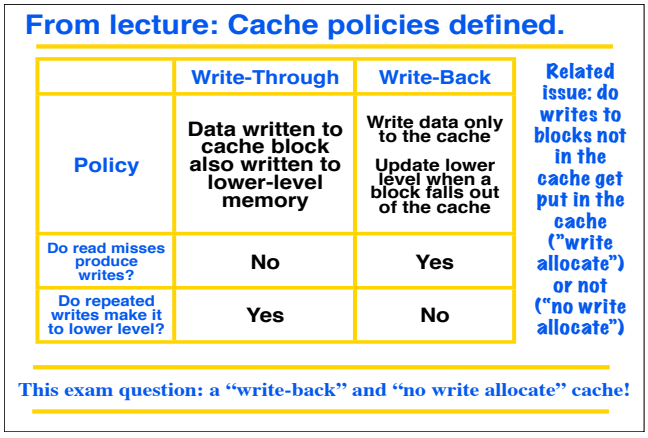
If the cache line does not have unused sets, the cache uses a least-recently used replacement policy, coded by an L bit for each index (see slides on next page for the encoding of the bit).

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:

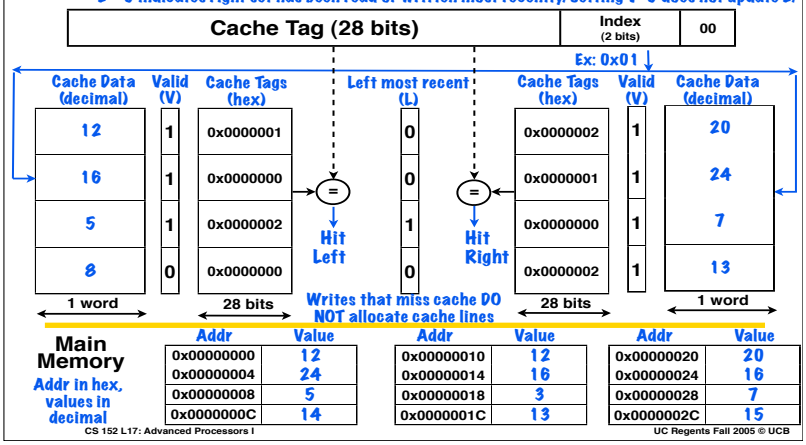
```
SW   R0   16(R0)
LW   R20  20(R0)
LW   R21  24(R0)
LW   R22  12(R0)
SW   R0   0(R0)
```

Executing the program may change cache state and main memory values. **For 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.



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.



Main Memory
Addr in hex, values in decimal

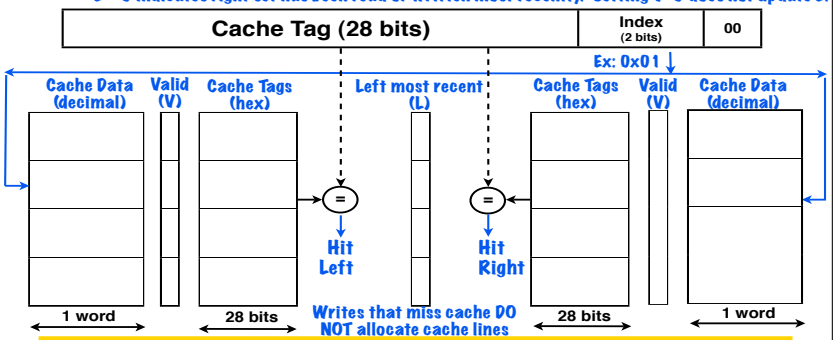
Addr	Value
0x00000000	12
0x00000004	24
0x00000008	5
0x0000000C	14

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.



Main Memory
Addr in hex, values in decimal

Addr	Value
0x00000000	
0x00000004	
0x00000008	
0x0000000C	

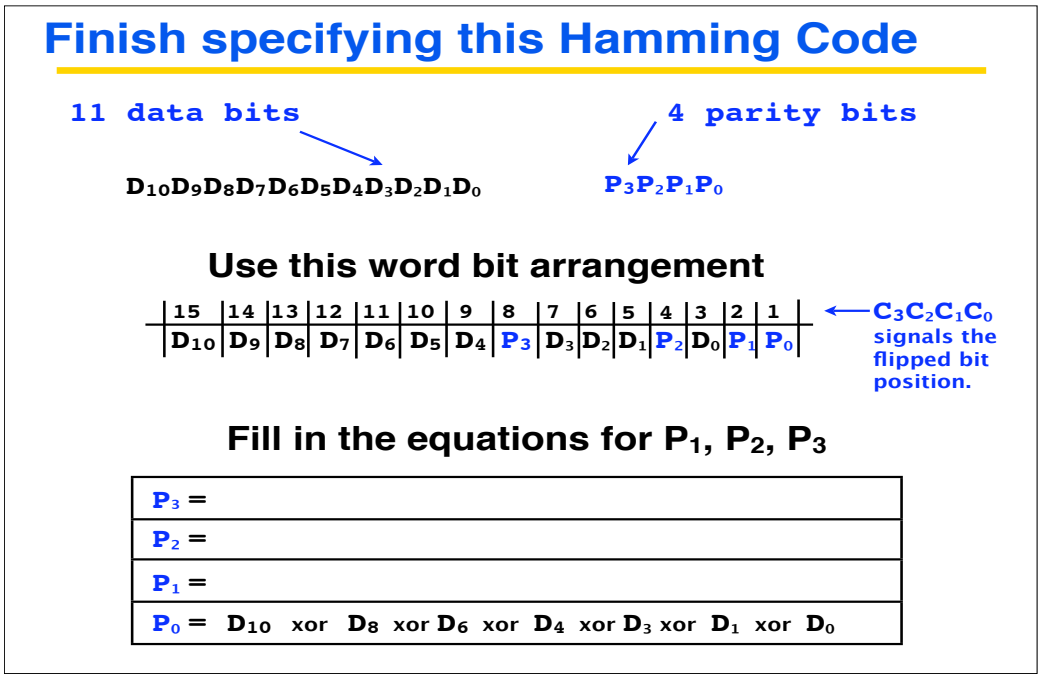
Addr	Value
0x00000010	
0x00000014	
0x00000018	
0x0000001C	

Addr	Value
0x00000020	
0x00000024	
0x00000028	
0x0000002C	

3 Hamming Codes (ECC) (14 points)

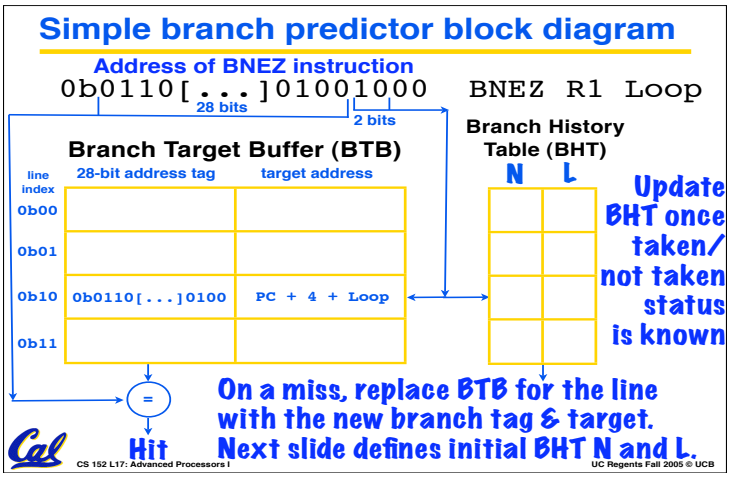
In the slide below, we show a design for a single-error correction Hamming code for 11 data bits, protected by 4 parity bits. At the bottom of the slide, we show the equation for computing the parity bit P_o.

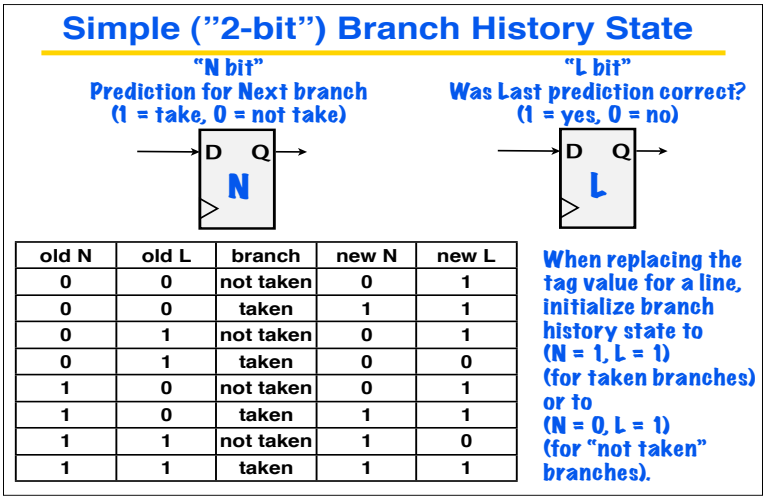
In the table on the slide, fill in the logic equations for computing the parity bits P_1, P_2, and P_3. Each equation is worth 4.66 points. **Only the equations written in the table boxes will be graded. Please write subscripts clearly.**



4 Branch Prediction (18 points)

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





old N	old L	branch	new N	new L
0	0	not taken	0	1
0	0	taken	1	1
0	1	not taken	0	1
0	1	taken	0	0
1	0	not taken	0	1
1	0	taken	1	1
1	1	not taken	1	0
1	1	taken	1	1

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).

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 ...

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

Branch predictor state before first inst. in trace executes

28-bit address tag	target address	N	L	line index
0x 0000 000	PC + 4 + Lab1	0	0	0b00
0x 0000 003	PC + 4 + Lab4	1	0	0b01
0x 0000 005	PC + 4 + Lab6	0	1	0b10
0x 0000 007	PC + 4 + Lab8	1	1	0b11

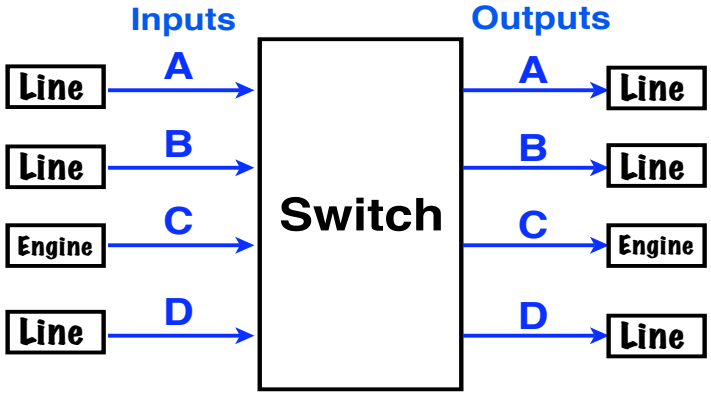
Fill in ALL underlines ("_"), and ALL N and L boxes.

28-bit address tag	target address	N	L	line index
0x 0000 _ _ _	PC + 4 + Lab _			0b00
0x 0000 _ _ _	PC + 4 + Lab _			0b01
0x 0000 _ _ _	PC + 4 + Lab _			0b10
0x 0000 _ _ _	PC + 4 + Lab _			0b11

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 ...

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.

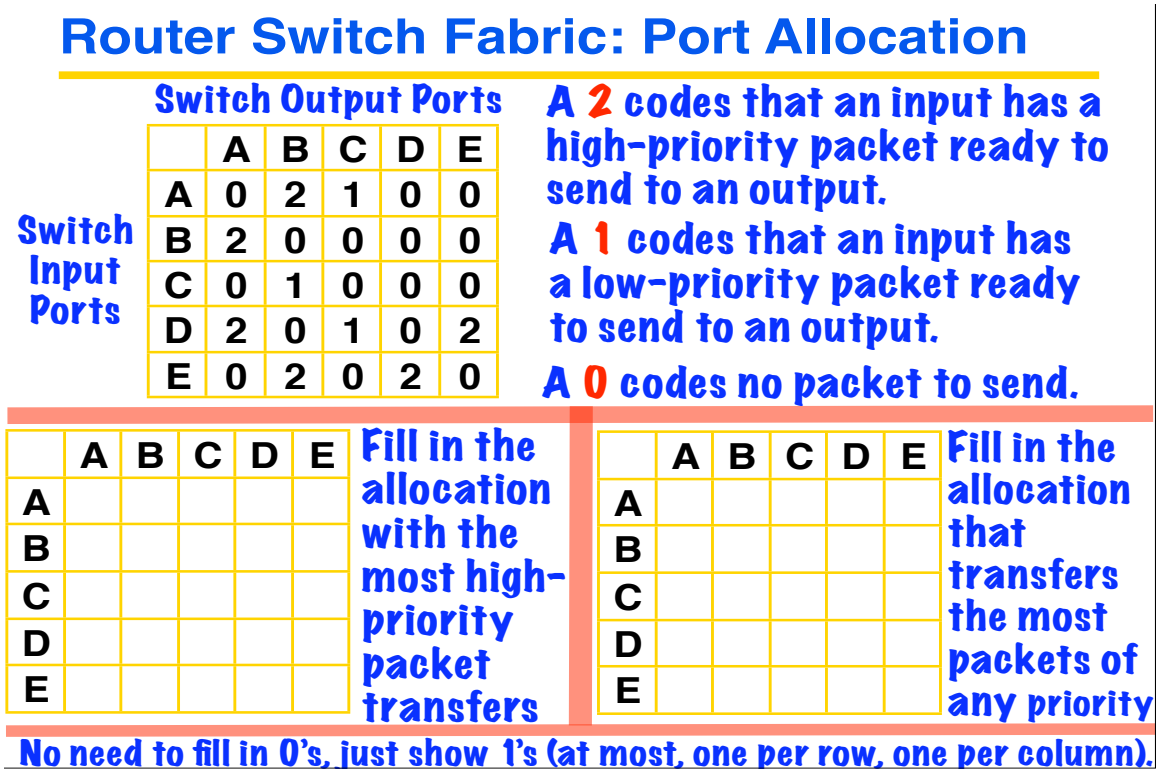
Allocator returns a matrix with at most one 1 in each row and column to set switches. Algorithm should be "fair", so no port always loses ... should also "scale" to run large matrices fast.

	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

	A	B	C	D	E
A					
B					
C					
D					
E					

	A	B	C	D	E
A					
B					
C					
D					
E					

6 Reorder Buffer Operation (22 points)

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 (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 =            R2 =
```

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

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										

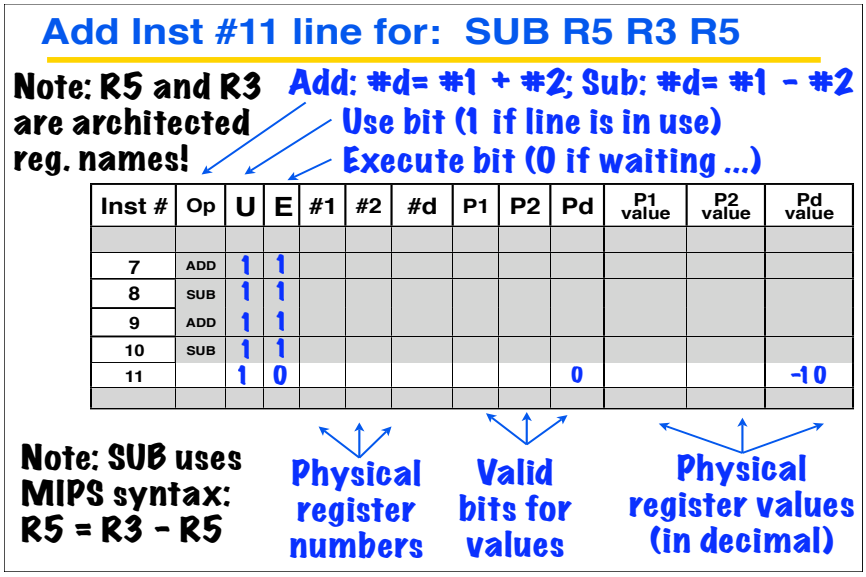
Add next inst, in program order.

Physical register numbers

Valid bits for values

Physical register values (in decimal)

Question 6c (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.



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