
ID#

HARDWARE COMPS

March 29, 2004

This part of the comps is made up of five questions for a total of 100 points.

Question	Points
1	/25
2	/15
3	/15
4	/20
5	/25
Total	/100

1 (25 points)

It is possible to use a Karnaugh map in order to find a three level implementation of a boolean function. Thus, rather than constraining ourselves to the simple two level implementation of the Sum-of-Products or the Product-of-Sums form, we may derive extra levels, thus generating factored representations. As an example, look at the following Karnaugh map.

xy \ zw	00	01	11	10
00	0	0	1	0
01	1	1	1	1
11	1	1	1	1
10	0	1	1	1

=

Figure 1: Function $f = y + zw + xw + xz$

xy \ zw	00	01	11	10
00	0	0	1	0
01	1	1	1	1
11	1	1	1	1
10	0	*	1	*

+

Figure 2: $y + zw$

xy \ zw	00	01	11	10
00	0	0	*	0
01	*	*	*	*
11	*	*	*	*
10	0	1	*	1

Figure 3: $(x+y)(z+w)$

Thus, we can circle the implicates and arrive at the factored solution:
 $F = y + zw + (x+y)(z+w)$

We are giving you the following Karnaugh map. The same process as described in the previous example can be used in the dual sense to find the prime **implicates** of the *first* Karnaugh map and then the **implicants** of the *residual* Karnaugh map. Please use this process in order to derive a factored form that matches the logic template given in the next picture and complete the schematic by marking the correct inputs. (Assume inverted inputs exist for all variables.)

ab \ cd	00	01	11	10
00	0	0	0	0
01	1	0	0	0
11	1	1	0	1
10	1	1	0	0

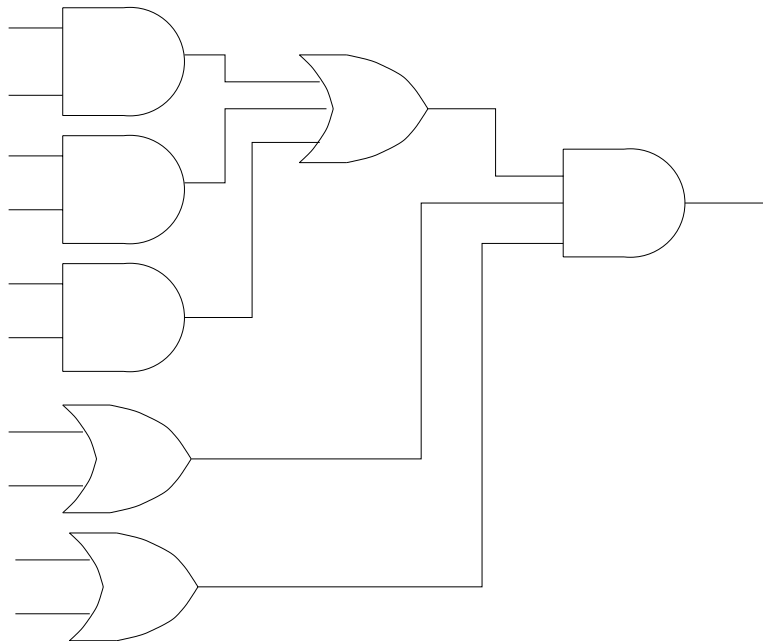
=

ab \ cd	00	01	11	10
00				
01				
11				
10				

*

ab \ cd	00	01	11	10
00				
01				
11				
10				

=



2 (15 points)

Consider the following instruction set, where $op = (add, sub, or, and, xor, logical\ or, logical\ and)$, has 64 registers, and uses a fixed location and size for each instruction opcode encoding.

```
Op   Rx, Ry, Rz

Load Rd, Rs(Rt)
Store Rs(Rt), Rd
BEQZ Rx
```

(Part A) How many bits are required for each instruction (if they are all the same size)?

(Part B) Write the smallest program that sums the values in registers 16 thru 32.

(Part C) Write the smallest program that sums the values in memory locations 1000_{16} to 2000_{16} .

3 (15 points)

A computer with 64 bit virtual address has a 48KB cache with cache blocks of 16 Bytes. Assume the machine is byte addressable. (Write all calculations down to receive full credit)

(Part A) How many bits are needed for the block offset, for the set index, and the tag bits if the cache is direct mapped, for a virtual addressed cache? How many blocks and sets are there?

(Part B) How many bits are needed for the block offset, set index, and tag bits if the cache is **3-way** set associative, for a virtual addressed cache? How many blocks and sets are there?

4 (20 points)

Assume you have a 128 byte cache, with 32 bytes per block. The memory is byte addressable. All cache blocks start out as invalid. Answer the following questions filling in the table below using the following address stream:

Addresses in the order they are accessed:

90, 160, 280, 50, 190, 90, 250, 120, 200, 60, 300, 30, 130, 192, 96, 30, 220, 280

Fill in for each address:

- a The set each address indexes into (the sets start at number 0).
- b The full block address range (starting address and ending address of the block) stored there after performing the access.
- c Indicate if the address access is a hit or miss in the cache.
- d If miss, what block (its full block address range) was evicted from the cache using LRU?

Fill in the table below assuming that the cache is 2-way associative, and use **LRU** for replacement.

Address	Set Index	Block Addresses	Hit or Miss	If Miss, Block Evicted
90				
160				
280				
50				
190				
90				
250				
120				
200				
60				
300				
30				
130				
192				
96				
30				
220				
280				

5 (25 points)

Consider the classical 5-stage MIPS/DLX pipeline with delayed branches. The pipeline stages are *Fetch (F)*, *Instruction Decode (ID)*, *Execute (EX)*, *Memory Access (M)*, and *Write Back (WB)*.

Assume that the branch condition is computed at the end of the ID stage, while the branch target address is available at the end of the EX stage. The following microarchitectural modification has been effected as well: If the branch is resolved as *not taken* at the end of the ID stage, the PC is incremented **once more**.

(Part A) Explain how the compiler should schedule instructions into the branch delay slots. How many branch delay slots are there? If the branch is known to be *taken* most of the time, where should the compiler get the branch delay slot instructions? What if the branch is known to be *not taken* most of the time? (Do not assume canceling instructions. Assume that there exist no instructions that can be utilized for filling in the branch delay slots after the branch join point.)

(Part B) Schedule the branch delay slots for the following example, according to the policy you suggested. Assume that in the example code in the left, *Case 1*, the branch is anticipated to be more frequently not taken, while in *Case 2*, it is expected to be taken most of the time.

Case 1		Case 2	
add	R1, R2, R3	add	R1, R2, R3
bnez	R1, L1	bnez	R1, L1
sub	R1, R1, R2	sub	R1, R1, R2
xor	R5, R6, R6	xor	R5, R6, R6
jmp	L2	jmp	L2
L1:	and R5, R1, R2	L1:	and R5, R1, R2
	add R1, R1, R2		add R4, R1, R2
L2:		L2:	xor R4, R4, R1

Case 1	Case 2

