The University of Alabama in Huntsville
Electrical & Computer Engineering Department
CPE 431 01
Test 2 Solution
Fall 2013

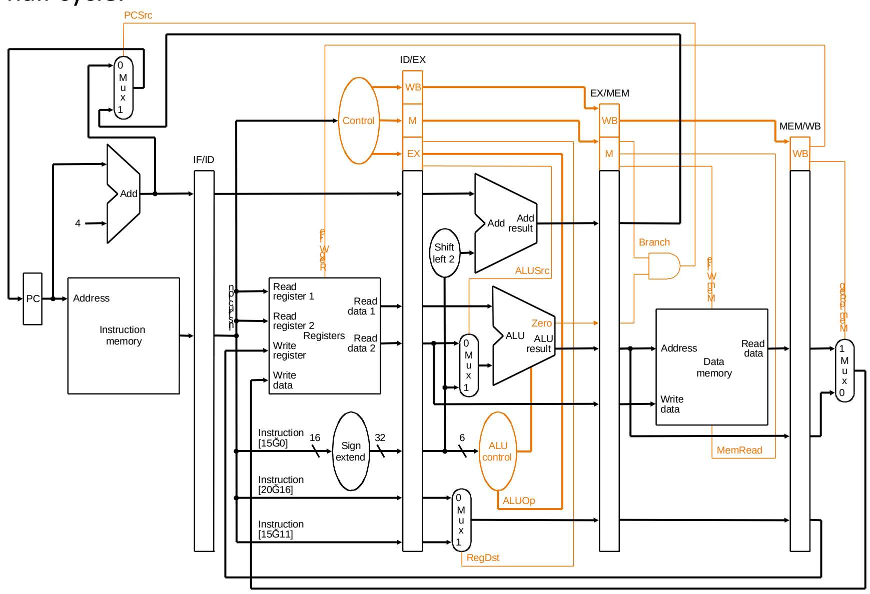
Show all work. You will not receive full credit for a problem if you do not show your work!

1. (1 point) _**Temporal**_ locality is the principle stating that if a data location is references then it will tend to be referenced again soon.
2. (1 point) A memory _**hierarchy**_ is a structure that uses multiple levels of memories.
3. (1 point) _**Write through**_ is a scheme in which writes always update both the cache and the next lower level of the memory hierarchy.
4. (1 point) _**Address translation**_ is the process by which a virtual address is mapped to an address used to access memory.
5. (1 point) _**Swap space**_ is the space on the disk reserved for the full virtual memory space of a process.
6. (10 points) A cache designer wants to increase the size of a 4 KB virtually indexed, physically tagged cache. Given a page size of 16 KB, is it possible to make a 16 KB direct-mapped cache, assuming 2 words per block? If not, how would the designer increase the size of the cache? If so, is 16 KB the largest direct-mapped cache size possible?

$$16KB \times \frac{1word}{4bytes} \times \frac{1block}{2words} \times \frac{1set}{1block} = 2Ksets$$

There is a byte offset of 2 bits, a block offset of 1 bit and an index of 11 bits. For this to be possible, all of these need to fit inside the page offset. For a page size of 16 KB, the page offset is 14, so yes, it is possible. 11 + 2 + 1 $\leq$ 14. Since the number of bits is already the max (14), we cannot make a bigger direct mapped virtually indexed, physically tagged cache with 2 words per block. We could make a bigger cache by having multiple blocks per set.

7. (15 points) Consider executing the following code on a pipelined datapath like the one shown except that 1) it supports j instructions that complete in the ID stage, and 2) it has MEM/WB forwarding only. The register file does support writing in the first half cycle and reading in the second half cycle.




```
sort:    addi  $sp, $sp, -20
         sw    $ra, 16($sp)
         sw    $s3, 12($sp)
         sw    $s2, 8($sp)
         sw    $s1, 4($sp)
         sw    $s0, 0($sp)
         add   $s2, $a0, $zero
         add   $s3, $a1, $zero
⇒        add   $s0, $zero, $zero
for1tst: slt   $t0, $s0, $s3
         beq   $t0, $zero, exit1
         addi  $s1, $s0, -1
for2tst: slt   $t0, $s1, $zero
         bne   $t0, $zero, exit2
         add   $t1, $s1, $s1
         add   $t1, $t1, $t1
         add   $t2, $s2, $t1
         lw    $t3, 0($t2)
         lw    $t4, 4($t2)
         slt   $t0, $t4, $t3
         beq   $t0, $zero, exit2
         add   $a0, $s2, $zero
         add   $a1, $s1, $zero
         jal   swap
         addi  $s1, $s1, -1
         j     for2tst
exit2:   addi  $s0, $s0, 1
         j     for1tst
exit1:   lw    $s0, 0($sp)
         lw    $s1, 4($sp)
         lw    $s2, 8($sp)
         lw    $s3, 12($sp)
         lw    $ra, 16($sp)
         addi  $sp, $sp, 20
         jr    $ra
```

If the `add $s0` instruction one instructions before the `for1tst` label begins executing in cycle 1 and the `beq $t0, $zero, exit1` is taken, what instructions are found in each of the five stages of the pipeline in the 9th cycle? Show the instructions being executed in each stage of the pipeline during each cycle. What value is stored in the ALUResult of the EX/MEM pipeline register in the 9th cycle? Assume that before the instructions are executed, the state of the machine was as follows:

The PC has the value 200_{10}, the address of the add \$s0 instruction
Every register has the initial value 20_{10} plus the register number.
Every memory word accessed as data has the initial value 10000_{10} plus the byte address of the word.

Cycle	IF	ID	EX	MEM	WB
1	add $s0				
2	slt $t0	add $s0			
3	beq $t0	slt $t0	add $s0		
4	beq $t0	slt $t0	bubble	add $s0	
5	addi $s1	beq $t0	slt $t0	bubble	add $s0
6	addi $s1	beq $t0	bubble	slt $t0	bubble
7	slt $t0	addi $s1	beq $t0	bubble	slt $t0
8	bne $t0	slt $t0	addi $s1	beq $t0	bubble
9	lw $s0	bubble	bubble	bubble	beq $t0

The instruction of interest is the one in the MEM stage during the 9th cycle, it is the addi $s1, $s0, -1 instruction that has been turned into a bubble by zeroing out the control lines.

EX/MEM.ALUResult = $s0 -1 = 36 -1 = 35

8. (10 points) Consider a pipeline for a register-memory architecture. The architecture has two instruction formats: a register-register format and a register-memory format. There is a single memory addressing mode (offset + base register). There is a set of ALU operations as follows:

ALUop← Rdest,Rsrc1,Rsrc2,offset
Rdest ← Rsrc1 ALUop Rsrc2
or Rdest ← Rsrc1 ALUop MEM[Rsrc2 + offset]
or
Rdest ← MEM[Rsrc2 + offset]
or
MEM[Rsrc2 + offset] ← Rsrc1

where the ALUop is one of the following:

Add, Subtract, And, Or(with or without offset)
Load(Rsrc1 omitted)
Store(Rdest omitted)

Rdest, Rsrc1 and Rsrc2 are registers.

Branches use a full compare of two registers and are PC-relative. Assume that this machine is pipelined so that a new instruction is started every clock cycle. The pipeline structure is

IF	RF	ALU1	MEM	ALU2	WB					
	IF	RF	ALU1	MEM	ALU2	WB				
		IF	RF	ALU1	MEM	ALU2	WB			
			IF	RF	ALU1	MEM	ALU2	WB		
				IF	RF	ALU1	MEM	ALU2	WB	
					IF	RF	ALU1	MEM	ALU2	WB

The first ALU stage is used for effective address calculation for memory references and branches. The second ALU stage is used for operations and branch comparisons. RF is both a decode and register-fetch stage. Assume that when a register read and a register write of the same register occur in the same clock cycle, the write data is forwarded.

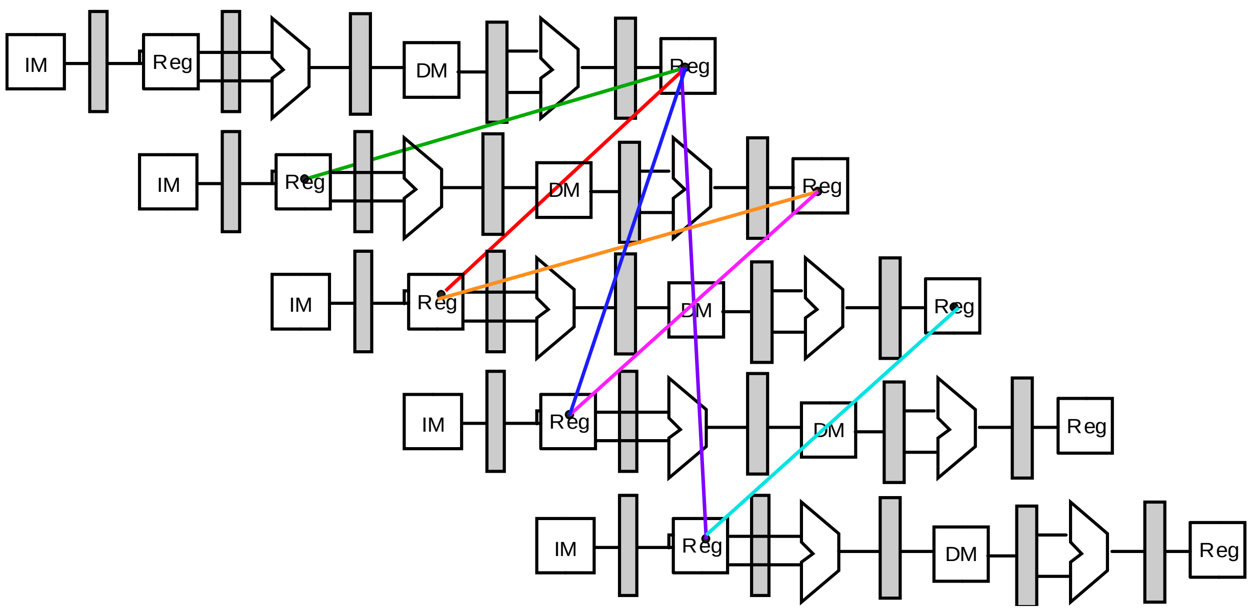
For the following code fragment:

a add $1, $1, 0($15)
b add $2, $1, 4($15)
c add $3 ,$2, 0($1)
d add $2, $2, 4($1)
e store $3, 8($1)

identify the data dependencies and draw a figure (using the multiple clock style) showing them as lines.

Dependencies

(1) a-b, (2) a-c, (3) a-d, (4) a-e, (5) b-c, (6) b-d, (7) c-e



9. (25 points) a) (5 points) Consider the following loop executing on a MIPS pipeline with full forwarding. Calculate the number of cycles it takes to execute this loop, neglecting pipeline fill cycles. b) (15 points) Unroll the loop so that 3 iterations of the loop are executed at once and schedule the unrolled code on a 2-issue pipeline in which any instruction can be issued in any slot. c) (5 points) Calculate the speedup from the original loop to the unrolled loop scheduled on a 2-issue pipeline.

```
(a)     add   $s1, $zero, $zero
        addi  $t1, $s0, -240
Loop:   add   $t4, $t2, $t1
        lw    $t3, 0($t4)
1 stall
        lw    $t3, 0($t3)
1 stall
        add   $s1, $s1, $t3
        addi  $t1, $t1, 4
        bne   $t1, $s0, Loop
1 stall
```

Total execution time = 59 iteration x (6 instructions + 2 data stalls + 1 control stall) + 1 iteration x (6 instructions + 2 data stalls) = 59*9 + 8 = 531 + 8 = 539 cycles.

(b) Unrolling three iterations:

```
add  $t4, $t2, $t1
lw   $t3, 0($t4)
lw   $t3, 0($t3)
add  $s1, $s1, $t3
lw   $t3, 4($t4)
lw   $t3, 0($t3)
add  $s1, $s1, $t3
lw   $t3, 8($t4)
lw   $t3, 0($t3)
add  $s1, $s1, $t3
addi $t1, $t1, 12
bne  $t1, $s0, Loop
```

Cycle	Issue Slot 1	Issue Slot 2
1	`add $t4, $t2, $t1`	
2	`lw $t3, 0($t4)`	`lw $t5, 4($t4)`
3	`lw $t6, 8($t4)`	`addi $t1, $t1, 12`
4	`lw $t3, 0($t3)`	`lw $t5, 0($t5)`
5	`lw $t6, 0($t6)`	
6	`add $s1, $s1, $t3`	
7	`add $s1, $s1, $t5`	
8	`add $s1, $s1, $t6`	`bne $t1, $s0, Loop`
9		
10		

(c) New execution time = 19 iterations x (8 cycles + 1 control stall) + 1 iteration x 8 cycles = 171 + 8 = 179 cycles
Speedup = 539 cycles/179 cycles = 3.01

10. (5 points) The following code is written in MATLAB, where elements within the same column are stored contiguously. Does `C(1, I)` exhibit spatial locality? temporal locality? Explain your answers. A, B, and C are all arrays of integers 8000 by 8000.

```
for I=1:8000
  for J=1:8
      A(I,J) = B(J,1) + A(J, I) + C(1, I);
```

C(1, I) does not exhibit spatial locality, C(1, 1) C(1, 2) are not close together
C(1, I) does exhibit temporal locality as it is accessed 8 times in a row

11. (15 points) Here is a series of address references given as byte addresses: 118, 483, 2069, 321, 368, 1077, 1505, 812, 2832, 373, 1411, 511, 1463, 690, 4820, 1714, 1508. Assuming a two-way set associative-mapped cache with four-word blocks and a total size of 32 words that is initially empty and uses LRU, (a) label each reference in the list as a hit or a miss and (b) show the entire history of the cache, including tag and data.

$$32 words \times \frac{1 block}{4 words} \times \frac{1 set}{2 blocks} = 4 sets$$

	Index Block offset byte offset	
118	0000 0000 01 11 01 10	miss
483	0000 0001 11 10 00 11	miss
2069	0000 1000 00 01 01 01	miss
321	0000 0001 01 00 00 01	miss
368	0000 0001 01 11 00 00	miss
1077	0000 0100 00 11 01 01	miss
1505	0000 0101 11 10 00 01	miss
812	0000 0011 00 10 11 10	miss
2832	0000 1011 00 01 00 00	miss
373	0000 0001 01 11 01 01	hit
1411	0000 0101 10 00 00 11	miss
511	0000 0001 11 11 11 11	miss
1463	0000 0101 10 11 01 11	miss
690	0000 0010 10 11 00 10	miss
4820	0001 0010 11 01 01 00	miss
1714	0000 0110 10 11 00 10	miss
1508	0000 0101 11 10 01 00	hit

	Tag	Data	Tag	Data
0	**5**	**MEM[320:335]**	**22**	**M[1408:1423]**
1	**~~32~~, 75**	**~~MEM[2064:2079]~~, M[4816:4831]**	**44**	**M[2832:2847]**
2	**~~7~~, 12**	**~~MEM[480:495]~~, M[800:815]**	**23**	**M[1504:1519]**
3	**~~1~~, ~~16~~, ~~7~~, 10**	**~~MEM[112:127]~~, ~~M[1072:1087]~~, ~~M[496:511]~~, M[688:703]**	**~~5~~, ~~22~~, 26**	**~~MEM[368:383]~~, ~~M[1456:1471]~~, M[1712:1727]**

12. (15 points) Virtual memory uses a page table to track the mapping of virtual addresses to physical addresses. Consider this stream of virtual addresses as seen on a system: 9452, 30964, 19136, 46502, 38110, 16653, 48480. Assume 8KB pages, a four-entry fully associative TLB, and true LRU replacement. If pages must be brought in from disk, increment the next largest page number. Given the address stream, and the initial TLB and page table states shown above, show the final state of the system. Also, list for each reference if it is a hit in the TLB, a hit in the page table, or a page fault.

TLB

Valid	Tag	Physical Page Number
1	11	12
1	7	4
1	3	6
0	4	9

Page table

Valid	Physical page or in disk
1	5
0	Disk
0	Disk
1	6
1	9
1	11
0	Disk
1	4
0	Disk
0	Disk
1	3
1	12

These are byte addresses. The page offset is $\log_2 8K = 13$. The virtual page number can be obtained by dividing the byte address by 8KB (8192) and taking the integer part of the result.

	VPN	TLB?	PT?	PF?
9452	1	N	N	Y
30964	3	Y	Y	N
19136	2	N	N	Y
46502	5	N	Y	N
38110	4	N	Y	N
16653	2	Y		
48480	5	Y		

TLB

Valid	Tag	Physical Page Number
1	~~11~~, 2	~~12~~, 14
1	~~7~~, 5	~~4~~, 11
1	3	6
0, 1	~~4, 1~~, 4	~~9~~, ~~13~~, 9

Page table

	Valid	Physical page or in disk
0	1	5
1	0, 1	~~Disk~~, 13k
2	0, 1	~~Disk~~, 14
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	0	Disk
9	0	Disk
10	1	3
11	1	12