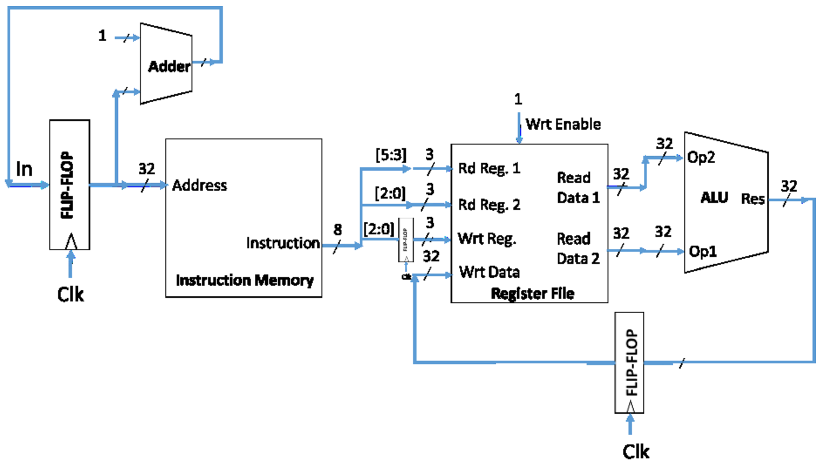
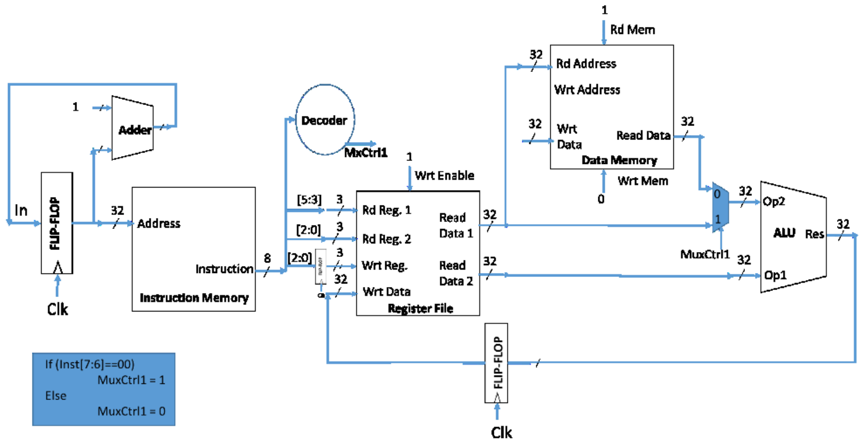
CS-GY-6133 Homework 2

Q1. Single-cycle Processor Design [25 Points]

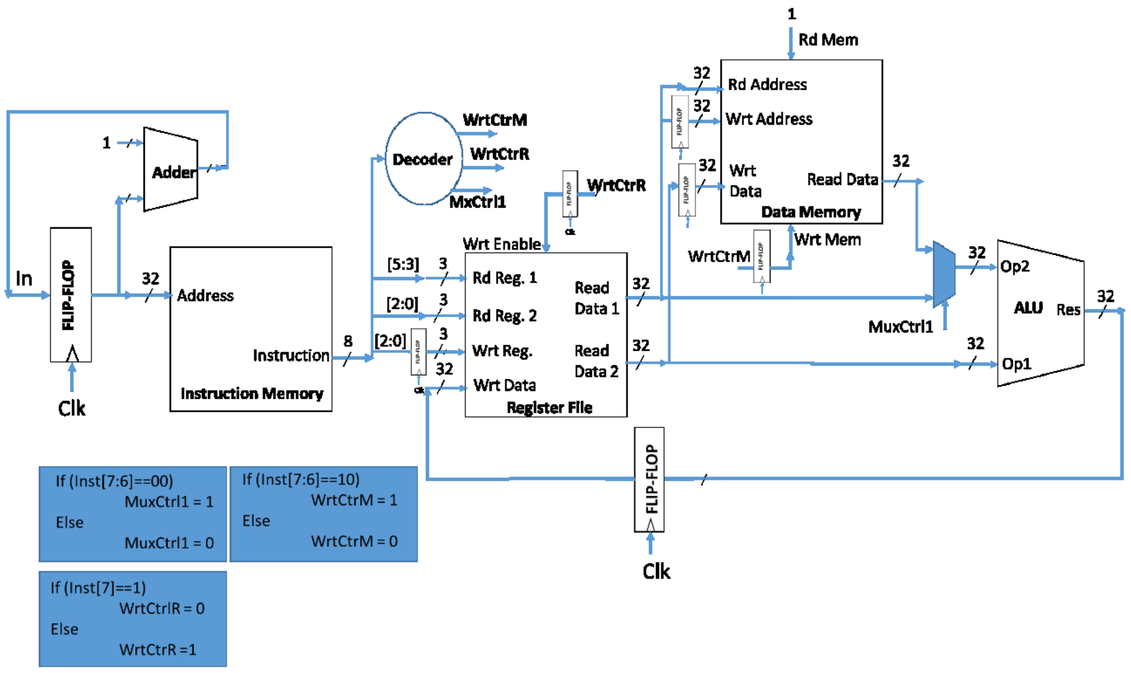
1.1 [5 Points] Draw a micro-architecture diagram of a single-cycle synchronous processor that supports only the *sub* instruction.



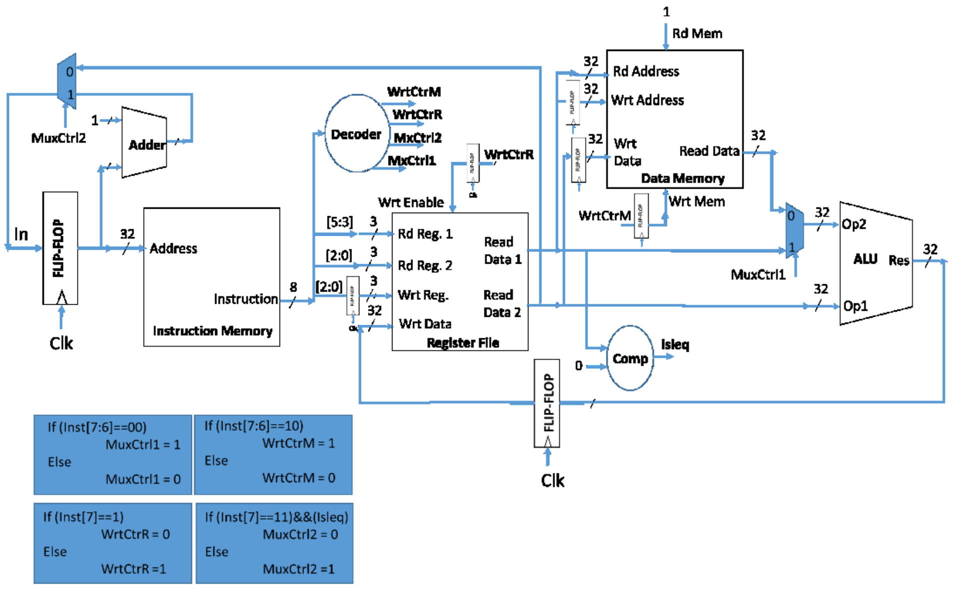
1.2 [5 Points] To the micro-architecture above, add support for the *subm* instruction. Your processor should support *sub* and *subm*.



1.3 [5 Points] Now, add support for the *st* instruction. Your processor should support *sub*, *subm* and *st*.



1.4 [5 Points] Finally, add support for the bleq instruction. This final installation should support the full CSGY6133 ISA.



1.5 [5 Points] Assume that every hardware block in your micro-architecture has a 1 ns latency. This includes the IM, DM, RF and ALU, in addition to hardware modules like decoders, comparators, or adders that you may need. MUXes have a 0.5 ns delay. What is the maximum frequency for your design?

Longest path goes through:
IMEM -> RF -> DMem -> MUX -> ALU (1 + 1 + 1 + 0.5 + 1 = 4.5ns)

Therefore max frequency is 1/ (4.5 ns) = 222 MHz

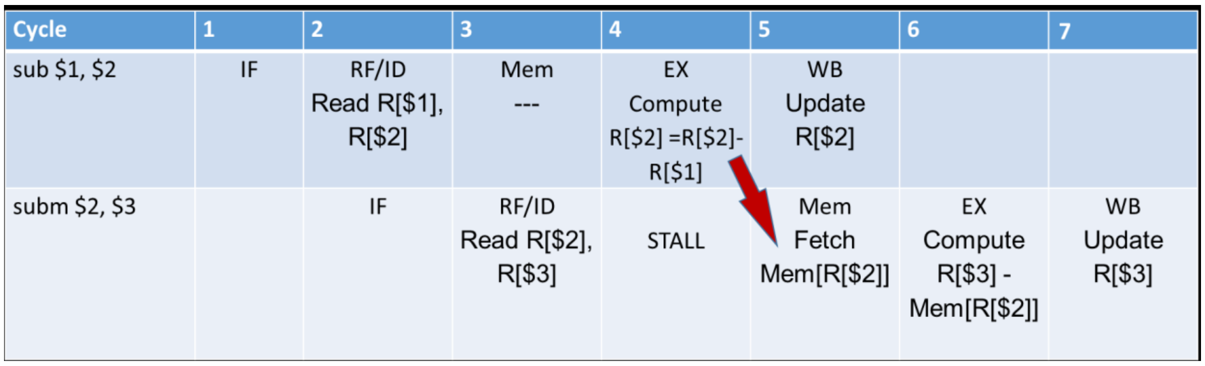
Q2. Pipelined CSGY6133 [20 Points]

Now consider a five stage pipelined implementation of the CSGY6133 processor. The five stages are described below:

- **IF**: Read the instruction from IM using the current PC. Update the PC to PC+1 for non-branch instructions. (Branch instructions are resolved in the EX stage; if the branch is taken, the PC is updated to the branch address instead of PC+1).
- **RF/ID**: Read operands for the current instruction from the RF. Decode the instruction and generate control signals.
- **Mem**: Read from or write to DM for the *subm* and *st* instructions, respectively.
- **EX**: Compute the result of *sub* and *subm* instructions by subtracting one operand from the other. (Compare the two operands for the *bleq* instruction and determine if the branch condition is true).
- **WB**: Write-back the result to the RF for the *sub* and *subm* instructions.

Two consecutive sub instructions in the CSGY6133 pipeline have a RAW dependency if the result of one is an operand of the next. For example the following two instructions have:

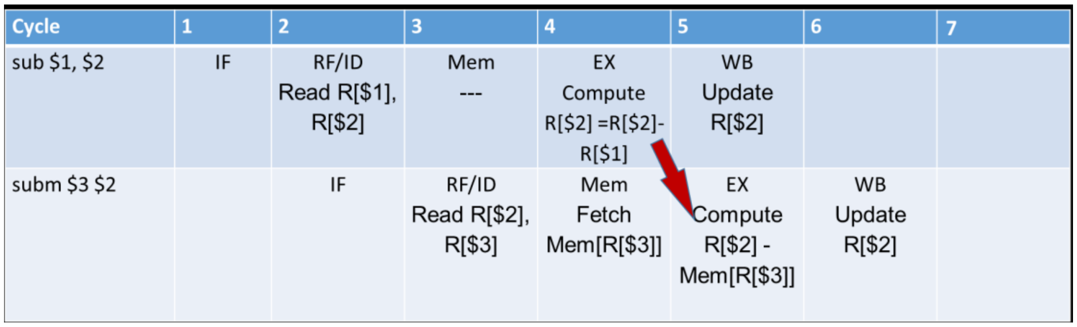
a)



Cycle	1	2	3	4	5	6	7
sub $1, $2	IF	RF/ID Read R[$1], R[$2]	Mem ---	EX Compute R[$2] =R[$2]-R[$1]	WB Update R[$2]		
subm $2, $3		IF	RF/ID Read R[$2], R[$3]	STALL	Mem Fetch Mem[R[$2]]	EX Compute R[$3] - Mem[R[$2]]	WB Update R[$3]

RAW dependency between the EX stage of sub and MEM stage of subm.
Resolved using stall + EX-MEM forwarding.

b)

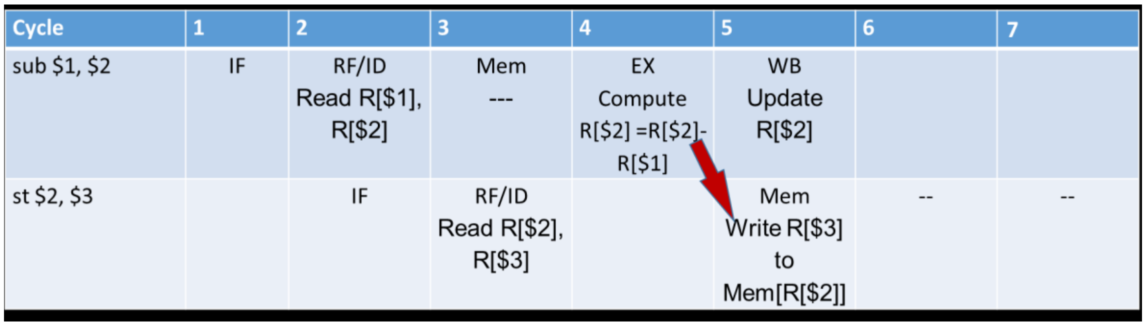


Cycle	1	2	3	4	5	6	7
sub $1, $2	IF	RF/ID Read R[$1], R[$2]	Mem ---	EX Compute R[$2] =R[$2]-R[$1]	WB Update R[$2]		
subm $3 $2		IF	RF/ID Read R[$2], R[$3]	Mem Fetch Mem[R[$3]]	EX Compute R[$2] - Mem[R[$3]]	WB Update R[$2]	

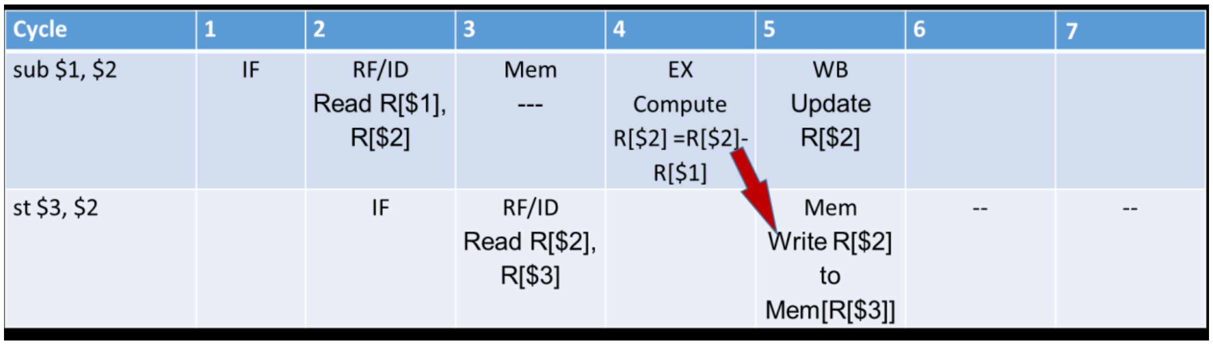
RAW dependency between the EX stage of sub and EX stage of subm.
Resolved using EX-EX forwarding.

(NOTE: there are two types of dependencies between the sub and subm instructions, depending on whether the dependent register is rs or rd. These are resolved in different ways.)

c)



Cycle	1	2	3	4	5	6	7
sub $1, $2	IF	RF/ID Read R[$1], R[$2]	Mem ---	EX Compute R[$2] =R[$2]-R[$1]	WB Update R[$2]		
st $2, $3		IF	RF/ID Read R[$2], R[$3]		Mem Write R[$3] to Mem[R[$2]]	--	--



Cycle	1	2	3	4	5	6	7
sub $1, $2	IF	RF/ID Read R[$1], R[$2]	Mem ---	EX Compute R[$2] =R[$2]-R[$1]	WB Update R[$2]		
st $3, $2		IF	RF/ID Read R[$2], R[$3]		Mem Write R[$2] to Mem[R[$3]]	--	--

RAW dependencies between the EX stage of sub and MEM stage of st.
Resolved using Stall + EX-MEM forwarding.

(NOTE: there are two types of dependencies between the sub and st instructions, depending on whether the dependent register is rs or rd, but both are resolved using the same forwarding path. Ideally, we would prefer that you identify both but solutions that identify only one of the two will get full credit.)

d)

Cycle	1	2	3	4	5	6	7
subm $2, $3	IF	RF/ID Read R[$2], R[$3]	Mem Fetch Mem[R[$2]]	EX Compute R[$3]=R[$3] - Mem[R[$2]]	WB Update R[$3]		
sub $3, $1		IF	RF/ID Read R[$3], R[$1]	Mem ---	EX Compute R[$1] =R[$1]-R[$3]	WB Update R[$1]	

RAW dependency between the EX stage and EX stage resolved using EX-EX forwarding.

e)

Cycle	1	2	3	4	5	6	7
subm $2, $3	IF	RF/ID Read R[$2], R[$3]	Mem Fetch Mem[R[$2]]	EX Compute R[$3]=R[$3] - Mem[R[$2]]	WB Update R[$3]		
subm $3, $1		IF	RF/ID Read R[$3], R[$1]	STALL	Mem Fetch Mem[R[$3]]	EX Compute R[$1] =R[$1]-Mem[R[$3]]	WB Update R[$1]

RAW dependency between the EX stage and MEM stage resolved using STALL and EX-MEM forwarding.

f)

Cycle	1	2	3	4	5	6	7
subm $2, $3	IF	RF/ID Read R[$2], R[$3]	Mem Fetch Mem[R[$2]]	EX Compute R[$3]=R[$3] - Mem[R[$2]]	WB Update R[$3]		
subm $1, $3		IF	RF/ID Read R[$3], R[$1]	Mem Fetch Mem[R[$1]]	EX Compute R[$3] =R[$3]- Mem[R[$1]]	WB Update R[$3]	

RAW dependency between the EX stage and EX stage resolved using EX-EX forwarding.

g)

Cycle	1	2	3	4	5	6	7
subm $2, $3	IF	RF/ID Read R[$2], R[$3]	Mem Fetch Mem[R[$2]]	EX Compute R[$3]=R[$3] - Mem[R[$2]]	WB Update R[$3]		
st $3, $1		IF	RF/ID Read R[$3], R[$1]	STALL	Mem Write R[$1] to Mem[R[$3]]	--	--

Cycle	1	2	3	4	5	6	7
subm $2, $3	IF	RF/ID Read R[$2], R[$3]	Mem Fetch Mem[R[$2]]	EX Compute R[$3]=R[$3] - Mem[R[$2]]	WB Update R[$3]		
st $1, $3		IF	RF/ID Read R[$1], R[$3]	STALL	Mem Write R[$3] to Mem[R[$1]]	--	--

RAW dependency between the EX stage and MEM stage Resolved using Stall + EX-MEM forwarding.

Q3. Pipelined MIPS [35 Points]

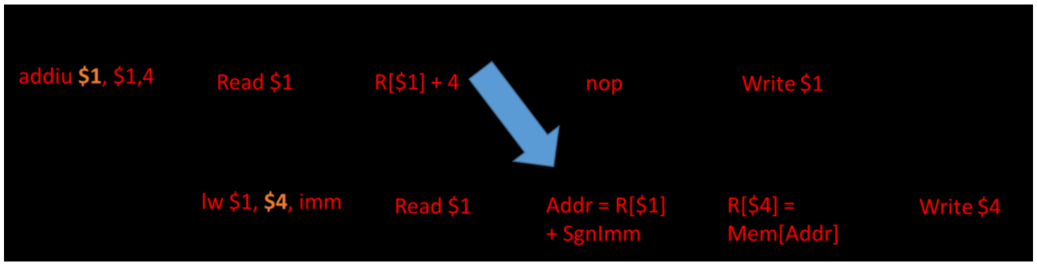
The following MIPS assembly code executes on a 5-stage pipelined MIPS processor. The goal is to compute the sum of two arrays, A and B, and to copy the contents of the array B to array A.

//baseA and baseB are the base addresses of arrays A and B
//Each array has N 32-bit elements

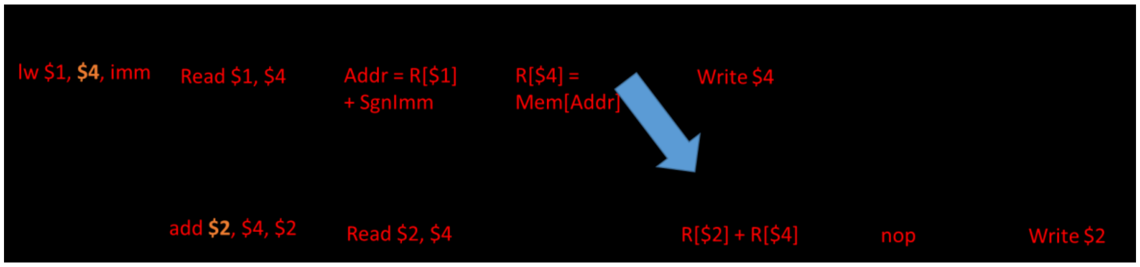
//Reg. $1 holds loop counter i, initialized to i=-4
//Reg. $2 holds sum1, initialized to sum1=0
//Reg. $3 holds sum2, initialized to sum2=0
//Reg. $4 holds tmp1, initialized to tmp1=0
//Reg. $5 holds tmp2, initialized to tmp2=0
//Reg. $6 holds count=(N-1)*4
//Note: The operands in mnemonics below appear in the same order as in the instruction format

3.1 [10 Points] Assuming a standard 5-stage MIPS pipeline, identify each pair of instructions that results in a RAW hazard (including those that are non-consecutive). For each pair, draw a timing diagram illustrating how the MIPS processor resolves the dependency (using either forwarding, stalling or both). You can assume that the processor has EX-EX, MEM-EX and MEM-MEM forwarding.

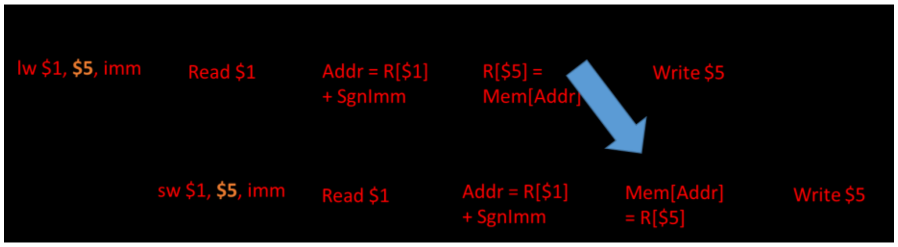
a)



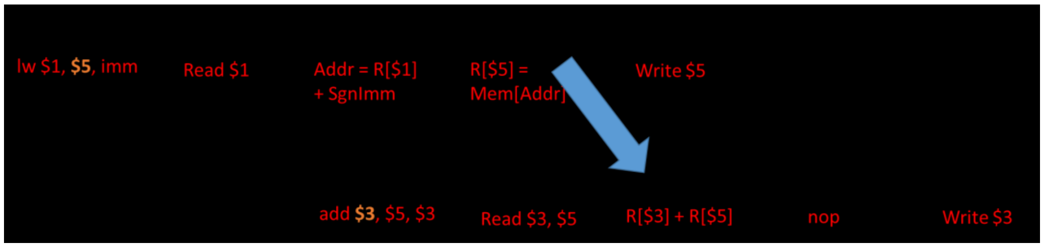
b)



c)

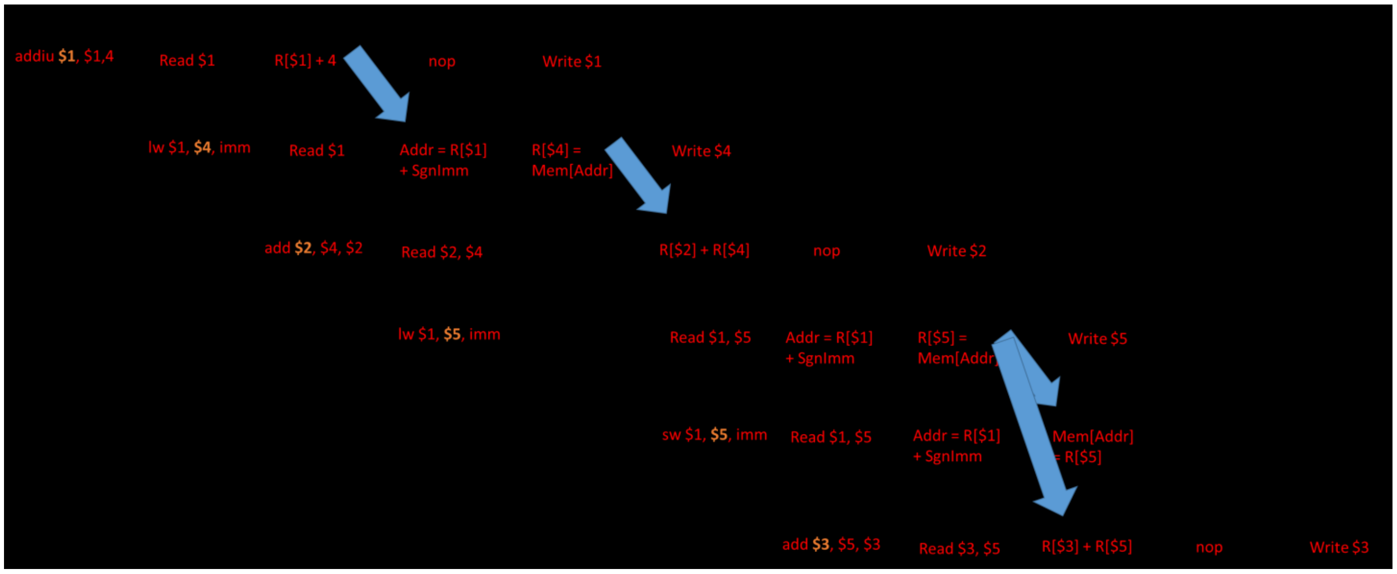


d)



Note that the instruction between the two instructions above is not shown.

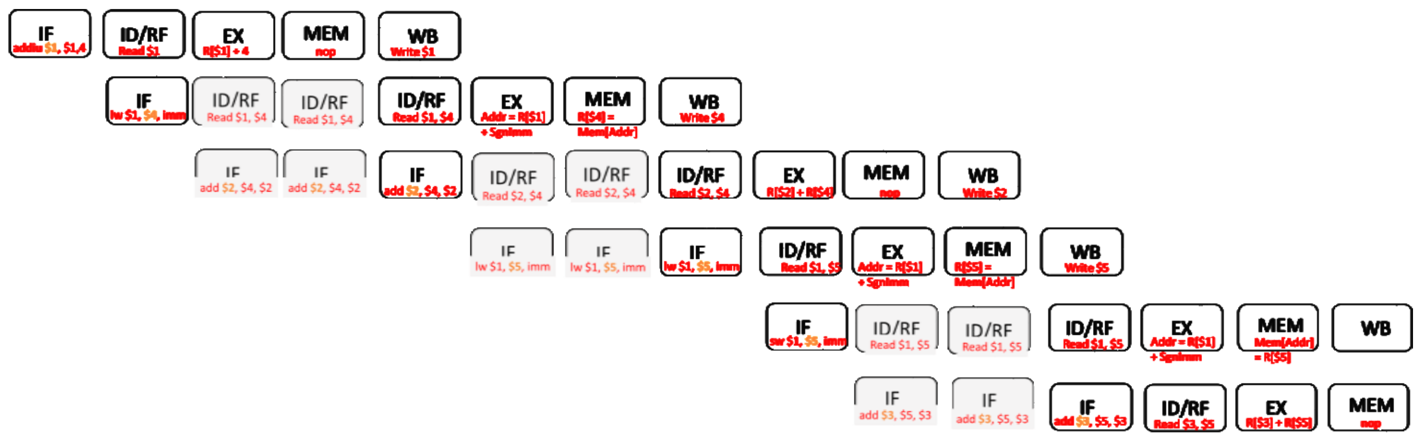
3.2 [5 Points] Draw a timing diagram that illustrates the execution of the first 6 instructions of the code above (i.e., up to but not including the bneq instruction). You can assume that there are no instructions in the pipeline before the loop executes and the registers/memory are (magically) initialized.



3.3 [5 Points] Estimate the number of clock cycles the loop above would take to completely execute if N=10. Branch instructions in the MIPS pipeline are resolved in the EX stage. To address control hazards, the compiler inserts two NOPS after each branch instruction (for example, after the bne instruction in the code above).

3.4 [5 Points] Repeat problem 3.2 assuming that the MIPS pipeline does not have any forwarding. Instead, it addresses hazards by detecting RAW dependencies and stalling the pipeline till the dependency is resolved.

Note that since the pipeline has no forwarding, younger dependent instructions in the ID/RF stage wait till the older instruction on which they depend reaches the WB stage. At this point, the RAW dependency is resolved and the pipeline moves forward. In the diagram below, the grey boxes represent stalls.



3.5 [10 Points] Re-order the instructions in the code above such that you achieve the same functionality, but minimize the number of cycles taken to execute the first 6 instructions. As in 3.5, assume that the MIPS processor only implements stalling.

```
LOOP:  addiu $1, $1, 4          //i = i+4
lw $1, $5, baseB        //tmp2 = B[i]
lw $1, $4, baseA        //tmp1 = A[i]
add $2, $4, $2          //sum1 = tmp1 + sum1
add $3, $5, $3          //sum2 = tmp2 + sum2
sw $1, $5, baseA        // A[i] = tmp2
bne $1, $6, LOOP        //if i != count = (N-1)*4 then goto LOOP
```

The code above has only two RAW hazards, namely one between addiu and lw (which also existed before), and one between lw and add (this existed before too). As a consequence, there will be fewer stalls, thus reducing execution time.

Q4. Caches [20 Points]

4.1 [5 Points] Determine the number of offset bits, index bits and tag bits for a 1GB byte-addressable memory system has a 8 KB cache with the following parameters (separate answers for each case):

Total length of an address: 30 bits

(a) Direct-mapped, with 1 Byte blocks

Block Offset: 0 bits
Index bits: log2(2^13 Bytes/1 Byte) = 13 bits
Tag bits: 30-13 = 17 bits

(b) Direct mapped, with 4 Byte blocks
Block Offset = 2 bits
Index = log2(2^13/2^2) = 11
Tag = 17 bits

(c) 2-way set-associative, with 8 Byte blocks

Block offset: 3 bits
Index: log2(2^13/(2^3 * 2)) = 9 bits
Tag: 30-9-3 = 18 bits

(d) 4-way set-associative, with 16 Byte blocks

Block offset: 4 bits
Index: log2(2^13/(2^4*2^2)) = 7 bits
Tag: 19 bits

(e) Fully-associative, with 8 Byte blocks

Block offset: 3 bits
Index: 0 bits
Tag: 27 bits

4.2 [10 Points] Consider a small direct mapped 8-Byte cache with 2 Byte blocks. The cache is accessed with the following sequence of addresses. (The cache is Byte addressable so each access returns a Byte of data.)

0,1,2,3,4,5,6,7,8,9, 0,1,2,3,4,5,6,7,8,9 …

Determine if each access is a Hit or a Miss. If Miss, determine if the miss is a compulsory, conflict or capacity miss. You can assume the cache is initially cold/empty.

The cache has 4 sets with two ways each.

State of the cache after Accesses 0,1,2,3,4,5,6,7

0 (Miss) Compulsary	1 (Hit)
2 (Miss) Compulsary	3 (Hit)
4 (Miss) Compulsary	5 (Hit)
6 (Miss) Compulsary	7 (Hit)

After Accesses 8,9

8 (Miss) Compulsary	9 (Hit)
2	3
4	5

6	7

After Accesses 0,1,2,3,4,5,6,7

0 (Miss) Capacity	1 (Hit)
2 (Hit)	3 (Hit)
4 (Hit)	5 (Hit)
6 (Hit)	7 (Hit)

After Accesses 8,9

8 (Miss) Capacity	9 (Hit)
2 (Hit)	3 (Hit)
4 (Hit)	5 (Hit)
6 (Hit)	7 (Hit)

4.3 [5 Points] Now assume that we add a small 2-entry victim cache (each entry hold an entire cache block) to the direct mapped cache above. For the same pattern of accesses, determine which ones hit in the cache and which ones miss. On a hit, indicate if the hit was in the main cache or the victim cache.

State of the cache

After Accesses 0,1,2,3,4,5,6,7

0 (Miss) Compulsary	1 (Hit in Main)
2 (Miss) Compulsary	3 (Hit in Main)
4 (Miss) Compulsary	5 (Hit in Main)
6 (Miss) Compulsary	7 (Hit in Main)

Victim cache

Empty	Empty
Empty	Empty

After Accesses 8,9

8 (Miss) Compulsary	9 (Hit)
2	3
4	5
6	7

Victim cache

0	1
Empty	Empty

After Accesses 0,1,2,3,4,5,6,7

0 (Hit in Victim)	1 (Hit in Main)
2 (Hit in Main)	3 (Hit in Main)
4 (Hit in Main)	5 (Hit in Main)
6 (Hit in Main)	7 (Hit in Main)

Victim cache

8	9
Empty	Empty

After Accesses 8,9

8 (Hit in Victim)	9 (Hit in Main)
2	3
4	5
6	7