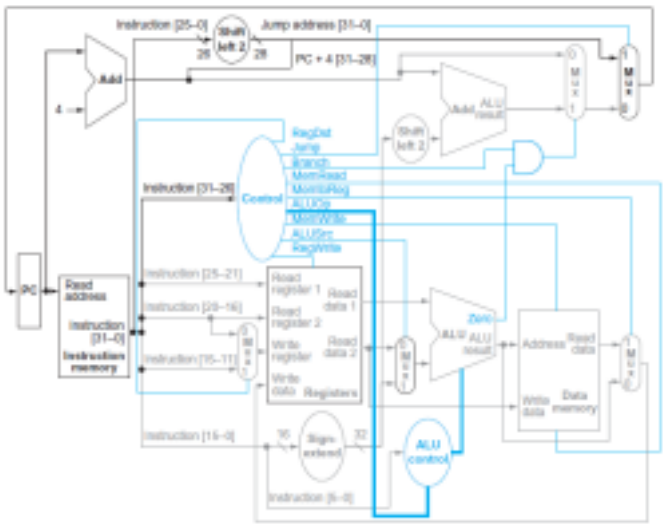
ECE 452: Computer Organization and Design
Spring 2022
Homework 4: The Processor

Assigned: 1 Mar 2022
Due: 8 Mar 2022

Instructions:

- Please submit your assignment solutions via Canvas in a word or pdf file.
- Some questions might not have a clearly correct or wrong answer. In such cases, grading is based on your arguments and reasoning for arriving at a solution.

Q1 (30 points) When silicon chips are fabricated, defects in materials (e.g., silicon) and manufacturing errors can result in defective circuits. A very common defect is for one wire to affect the signal in another. This is called a cross talk fault. A special class of cross-talk faults is when a signal is connected to a wire that has a constant logical value (e.g., a power supply wire). In this case we have a stuck-at-0 or a stuck-at-1 fault, and the affected signal always has a logical value of 0 or 1, respectively. The following problems refer to bit 0 of the Write Register input on the register file in the figure below.



a. **(10 points)** Let us assume that processor testing is done by filling the PC, registers, and data and instruction memories with some values (you can choose which values), letting a single instruction execute, then reading the PC, memories, and registers. These values are then examined to determine if a particular fault is present. Can you design a test (values for PC, memories, and registers) that would determine if there is a stuck-at-1 fault on this signal?

When we are testing for a stuck at one 1 fault for the write register bit 0 we can test this through trying to write to any register where bit 0 == 0. The way that i would test for this fault would be to run the following instruction

Add $t0 , $t1, $t1

This would be able to test if the value of the register was stuck at one because $t0 has a binary value of 01000 thus if we check the value of $t0 after running the instruction we could then see if it correctly wrote

to $t0 or to $t1 because $t1 has the binary value of 01001 we know that if $t1 = 2t1 that we have a stuck at one fault.

b. **(10 points)** Repeat (a) for a stuck-at-0 fault. Can you use a single test for both stuck-at-0 and stuck-at-1? If yes, explain how; if no, explain why not.

I would repeat the same instruction from above but swap $t0 with $t1 and vice versa. Ultimately running

Add $t1, $t0, $t0

For the same reason as before $t1 has a binary value of 01001 while $t0 has a binary value of 01000 thus if we are trying to write to t1 but instead accidentally wrote to t0 we would then know that we have a stuck at 0 fault for bit 0. In the case of my test it would not be possible to run the same test to check for both as if we used the test above to see if we had a stuck at 0 fault it would always exhibit the correct behavior.

c. **(10 points)** If we know that the processor has a stuck-at-0 fault on this signal, is the processor still usable? To be usable, we must be able to convert any program that executes on a normal MIPS processor into a program that works on this processor. You can assume that there is enough free instruction memory and data memory to let you make the program longer and store additional data. Hint: the processor is usable if every instruction "broken" by this fault can be replaced with a sequence of "working" instructions that achieve the same effect.

The processor would not be usable if there was a stuck at 0 fault on the bit 0 of the write register signal. This is because it would mean that any of the registers that are identified by an odd binary number would not be accessible. Some of such registers would be the return address and the stack pointer

Q2 (30 points) In this question, we will examine how pipelining affects the clock cycle time of the processor. Problems in this exercise assume that individual stages of the datapath have the following latencies. For each pipelining stage there is also an additional 30ps overhead between each stage.

IF	ID	EX	MEM	WB
400ps	750ps	150ps	200ps	350ps

Also, assume that instructions executed by the processor are broken down as follows:

alu	beq	lw	sw
40%	15%	30%	15%

a. **(5 points)** What is the clock cycle time in a pipelined and non-pipelined processor?

Pipelined: 780ps (period of the longest stage plus overhead)
Non-pipelined: 400ps+750ps+150ps+200ps+350ps = 1850ps (sum of all stage periods)

b. **(5 points)** What is the total latency of an SW instruction in a pipelined and non-pipelined processor?

Pipelined: Total = IF + ID + EX + MEM
= 4(780)ps = 3320ps

Non-pipelined: Total = IF + ID + EX + MEM + WB
= 400ps+750ps+150ps+200ps+350ps = 1850ps

c. **(10 points)** If we can split one stage of the pipelined datapath into two new stages, each with half the latency of the original stage, which stage would you split and what is the new clock cycle time of the processor?

If we could split the ID stage into two different stage both having a latency of 375ps

d. **(5 points)** Assuming there are no stalls or hazards, what is the utilization of the data memory?

Load word and Store word both use memory thus utilization is 30%+15% = 45%

e. **(5 points)** Assuming there are no stalls or hazards, what is the utilization of the write-register port of the "Registers" unit?

Load word and ALU use write register port, thus utilization is 30% +40% = 70%

Q3 (60 points) In this question, we examine how resource hazards, control hazards, and Instruction Set Architecture (ISA) design can affect pipelined execution. Problems in this exercise refer to the following fragment of MIPS code:

```
lw r16, 8(r6)
lw r17, 4(r6)
sub r5, r16, r17
beq r5, r4, label # assume the two registers here are not
equal sub r5, r1, r4
add r5, r5, r6
slt r5, r15,r4
```

Assume that individual pipeline stages have the following latencies:

IF	ID	EX	MEM	WB
350ps	240ps	250ps	220ps	150ps

a. **(10 points)** For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we only have one memory (for both instructions and data), there is a structural hazard every time we need to fetch an instruction in the same cycle in which another instruction accesses data. To guarantee forward progress, this hazard must always be resolved in favor of the instruction that accesses data. What is the total execution time of this instruction sequence in the 5-stage pipeline that only has one memory? We have seen that data hazards can be eliminated by adding nops to the code. Can you do the same with this structural hazard? Why?

This could would take a total of 17 cycles to complete and with each cycle taking 350ps we get a total execution time of 5950ps

We could not simply include NOPS instructions to fix the problem of using the same memory for instructions and data. NOP instructions must still be fetched and decoded before they do nothing, so if we add a nop in order to stall the next instruction from using the memory we will instead access the memory through using NOP instead.

b. **(10 points)** For this problem, assume that all branches are perfectly predicted (this eliminates all control hazards) and that no delay slots are used. If we change load/store instructions to use a register (without an offset) as the address, these instructions no longer need to use the ALU. As a result, MEM and EX stages can be overlapped and the pipeline has only 4 stages. Change this code to accommodate this changed ISA. Assuming this change does not affect clock cycle time, what speedup is achieved in this instruction sequence?

Addi t0, r6, 4
Lw r17, t0
Addi t0, t0, 4
Lw r16, t0
Sub r5, r16, r17
Beq r5, r4 label
Sub r5, r1, r4
Add r5, r5, r6
Slt r5, r15, r4

This new ISA would take 12 cycles, with each cycle taking 350ps, resulting in a total execution time of 4200ps
Speed up = Time old / Time new = 11(350ps)/12(350ps) = 0.916 total speed up

c. (10 points) Assuming stall-on-branch and no delay slots, what speedup is achieved on this code if branch outcomes are determined in the ID stage, relative to the execution where branch outcomes are determined in the EX-stage?

old

lw	if	id	ex	mem	wb								
lw		if	id	ex	mem	wb							
sub			if	id	ex	mem	wb						
beq				stall	stall	if	id	ex	mem	wb			
sub							if	id	ex	mem	wb		
add								if	id	ex	mem	wb	
slt									if	id	ex	mem	wb

new

lw	if	id	ex	mem	wb								
lw		if	id	ex	mem	wb							
sub			if	id	ex	mem	wb						
beq				stall	if	id	ex	mem	wb				
sub						if	id	ex	mem	wb			
add							if	id	ex	mem	wb		
slt								if	id	ex	mem	wb	

The old ISA in which branch outcomes are determined at the EX stage would take a total of 13 cycles. While The new design where branch outcomes are determined at the ID stage we then would only need to take a total of 12 cycles total. Meaning that the overall speed up would be 13/12 or a 1.08 speed up

d. **(10 points)** Given these pipeline stage latencies, repeat the speedup calculation from (b), but take into account the (possible) change in clock cycle time. When EX and MEM are done in a single stage, most of their work can be done in parallel. As a result, the resulting EX/MEM stage has a latency that is the larger of the original two, plus 20 ps needed for the work that could not be done in parallel.

Old cycle time = 350 ps
New cycle time = 350 ps + 20ps overhead

Speed up = 11(350)/12(370) = 0.867

e. **(10 points)** Given these pipeline stage latencies, repeat the speedup calculation from (c), taking into account the (possible) change in clock cycle time. Assume that the latency of the ID stage increases by 70% and the latency of the EX-stage decreases by 15% when branch outcome resolution is moved from EX to ID.

New time table

IF	ID	EX	MEM	WB
350ps	408ps	212.5ps	220ps	150ps

Old cycle time = 350ps
New cycle time = 408ps
Speed up = 13(350)/12(408) = 4550/4896 = 0.919

f. (10 points) Assuming stall-on-branch and no delay slots, what is the new clock cycle time and execution time of this instruction sequence if beq address computation is moved to the MEM stage? What is the speedup from this change? Assume that the latency of the EX-stage is reduced by 40 ps and the latency of the MEM stage is unchanged when branch outcome resolution is moved from EX to MEM.

OLD

lw	if	id	ex	mem	wb								
lw		if	id	ex	mem	wb							
sub			if	id	ex	mem	wb						
beq				stall	stall	if	id	ex	mem	wb			
sub							if	id	ex	mem	wb		
add								if	id	ex	mem	wb	
slt									if	id	ex	mem	wb

NEW

lw	if	id	ex	mem	wb								
lw		if	id	ex	mem	wb							
sub			if	id	ex	mem	wb						
beq				stall	stall	stall	if	id	ex	mem	wb		
sub								if	id	ex	mem	wb	
add									if	id	ex	mem	wb
slt										if	id	ex	mem	wb

New time table

IF	ID	EX	MEM	WB
350ps	240ps	40ps	220ps	150ps

Old Cycle count = 13
New Cycle count = 14

Speed up = 13(350ps)/14(350ps) = 13/14 = 0.928

Q4 (60 points) The importance of having a good branch predictor depends on how often conditional branches are executed. Together with branch predictor accuracy, this will determine how much time is spent stalling due to mispredicted branches. In this exercise, assume that the breakdown of dynamic instructions into various instruction categories is as follows:

R-Type	BEQ	JMP	LW	SW
30%	20%	10%	30%	10%

Also, assume the following branch predictor accuracies:

Always taken	Always not taken	2-Bit
60%	45%	80%

a. **(10 points)** Stall cycles due to mispredicted branches increase the CPI. What is the extra CPI due to mispredicted branches with the always-taken predictor? Assume that branch outcomes are determined in the EX-stage, that there are no data hazards, and that no delay slots are used

A stalled branch will add two cycles for each mispredicted branch. The probability that we fail when using the always taken approach is (1-0.6) = 0.4
Additional CPI = BEQ * Probability * cycles per stall = 0.2*0.4*2 = 0.16

.

b. **(10 points)** Repeat (a) for the "always-not-taken" predictor.

A stalled branch will add two cycles for each mispredicted branch. The probability that we fail when using the always not taken approach is (1-0.45) = 0.55
Additional CPI = BEQ * Probability * cycles per stall = 0.2*0.55*2 = 0.22

c. **(10 points)** Repeat (a) for the 2-bit predictor.

A stalled branch will add two cycles for each mispredicted branch. The probability that we fail when using the two bit approach is (1-0.8) = 0.2
Additional CPI = BEQ * Probability * cycles per stall = 0.2*0.2*2 = 0.08

d. **(10 points)** With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaces a branch instruction with an ALU instruction? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

New BEQ = 10%
Probability fails = (1-0.8) = 0.2
Cycles per stall = 2

Additional CPI = 0.1 * 0.2 * 2 = 0.04

e. **(10 points)** With the 2-bit predictor, what speedup would be achieved if we could convert half of the branch instructions in a way that replaced each branch instruction with two ALU instructions? Assume that correctly and incorrectly predicted instructions have the same chance of being replaced.

New BEQ = 10%
Probability fails = (1-0.8) = 0.2
Cycles per stall = 2

Additional CPI = 0.1 * 0.2 * 2 = 0.04

f. **(10 points)** Some branch instructions are much more predictable than others. If we know that 80% of all executed branch instructions are easy-to-predict loop-back branches that are always predicted correctly, what is the accuracy of the 2-bit predictor on the remaining 30% of the branch instructions?

.8/1

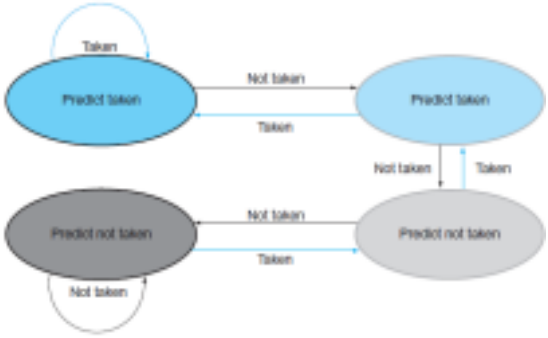
Q5 (30 points) This question examines the accuracy of various branch predictors for the following repeating pattern (e.g., in a loop) of branch outcomes: NT, T, NT, T, NT

a. **(10 points)** What is the accuracy of always-taken and always-not-taken predictors for this sequence of branch outcomes?

Sequence = NT, T , NT ,T ,NT
Always taken = T , T ,T ,T ,T
Never taken = N ,N ,N ,N ,N

Accuracy always taken = ⅖ = 0.4
Accuracy never taken = ⅗ = 0.6

b. **(10 points)** What is the accuracy of the two-bit predictor for the first 4 branches in this pattern, if the predictor starts off in the bottom left state (predict not taken) in the figure below?



Sequence = NT, T , NT ,T ,NT
Prediction= NT, NT, NT, NT, NT

Accuracy = ⅗ = 0.6

c. **(10 points)** What is the accuracy of the two-bit predictor if this pattern is repeated forever? Hint: The first few recurrences of this pattern do not have the same accuracy as the later ones because the predictor is still warming up. To determine the accuracy in the "steady state", you must work through the branch predictions until the predictor values start repeating.

Sequence = NT, T , NT ,T ,NT, NT, T , NT ,T ,NT, NT, T , NT ,T ,NT
Prediction= NT, NT, NT, NT, NT, NT, NT, NT, NT, NT, NT, NT, NT, NT, NT

Steady state accuracy = ⅗ = 0.6

Q6 (40 points) Compare the performance of 1-issue and 2-issue processors, taking into account program transformations that can be made to optimize for 2-issue execution for the code below.

```
for (i=0; i!=j; i+=2)
    b[i] = a[i+1] + a[i+2]
```

When writing MIPS code, assume that variables are kept in registers as follows and that all registers except those indicated as Free are used to keep various variables, so they cannot be used for anything else.

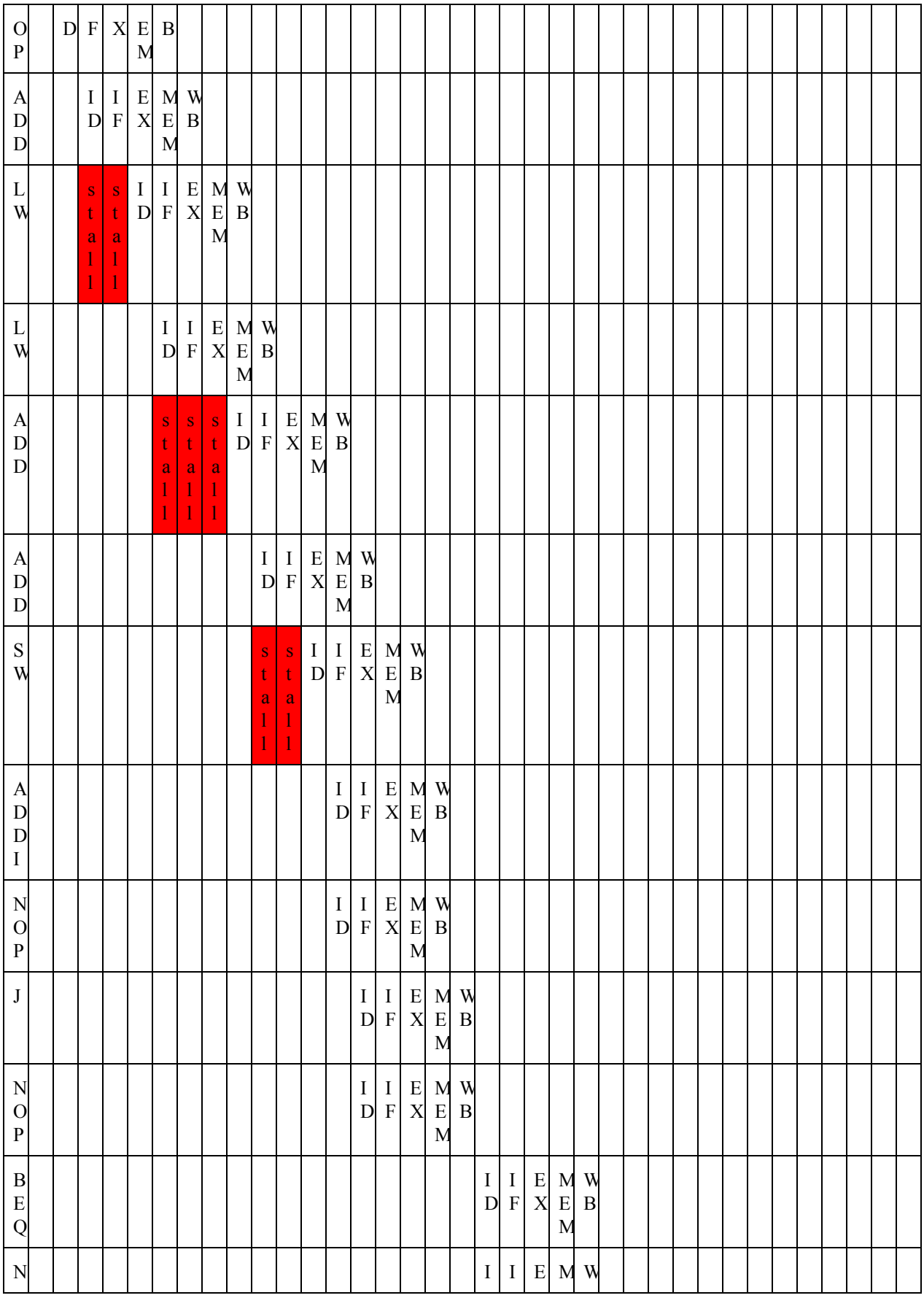
i	j	a	b	c	Free
R5	R6	R1	R2	R3	R10, R11, R12

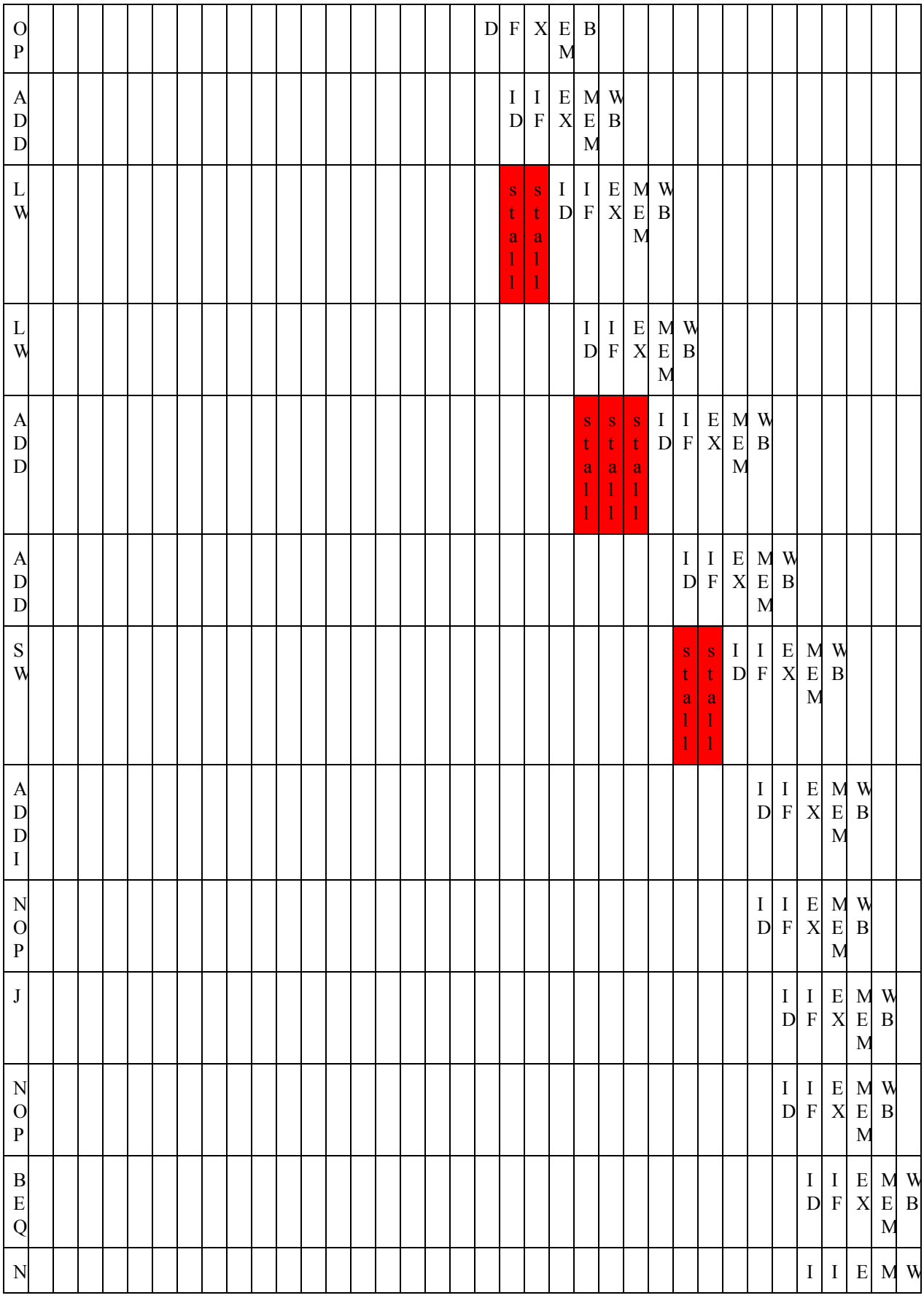
a. **(10 points)** Translate this C code to MIPS instructions. Your translation should be direct, without rearranging instructions to achieve better performance.

```
        Add r5, r5, 0
loop:
        Beq r5, r6 stop         # check if i = j if they are stop
        Add r10, r1, r5         # increment address of r1 by r5 to make r10 contain a[i]
        Lw r11, 4(r10)          # load the value of a[i+1] into r11
        Lw r12, 8(r10)          #load the value of a[i+2] into r12
        Add r11, r11, r12       # add a[i+1] and a[i+2] into r11
        Add r10, r2, r5         # increment the address of r2 by r5 to make r10 contain b[i]
        Sw r11, 0(r10)          # store the value of r11 into b[i]
        Addi r5, r5, 2          # increment the value of i by two before checking again
        J loop
Stop:
        exit
```

b. **(10 points)** If the loop exits after executing only two iterations, draw the pipeline diagram for your MIPS code executed on a 2-issue processor. Assume the processor has perfect branch prediction and can fetch any two instructions in the same cycle.

ADD	ID	IF	EX	MEM	WB																											
NOP	ID	IF	EX	MEM	WB																											
BEQ		ID	IF	EX	MEM	WB																										
N		I	I	E	M	W																										


OP D F X EM B
ADD ID IF EX MEM WB
LW ID IF EX MEM WB
stall stall
LW ID IF EX MEM WB
ADD ID IF EX MEM WB
stall stall stall
ADD ID IF EX MEM WB
SW ID IF EX MEM WB
stall stall
ADDI ID IF EX MEM WB
NOP ID IF EX MEM WB
J ID IF EX MEM WB
NOP ID IF EX MEM WB
BEQ ID IF EX MEM WB
N I I E M W


OP
ADD
LW
LW
ADD
ADD
SW
ADDI
NOP
J
NOP
BEQ
N
D
F
X
EM
B
ID
IF
EX
MEM
WB
stall
stall
stall
stall
stall
stall

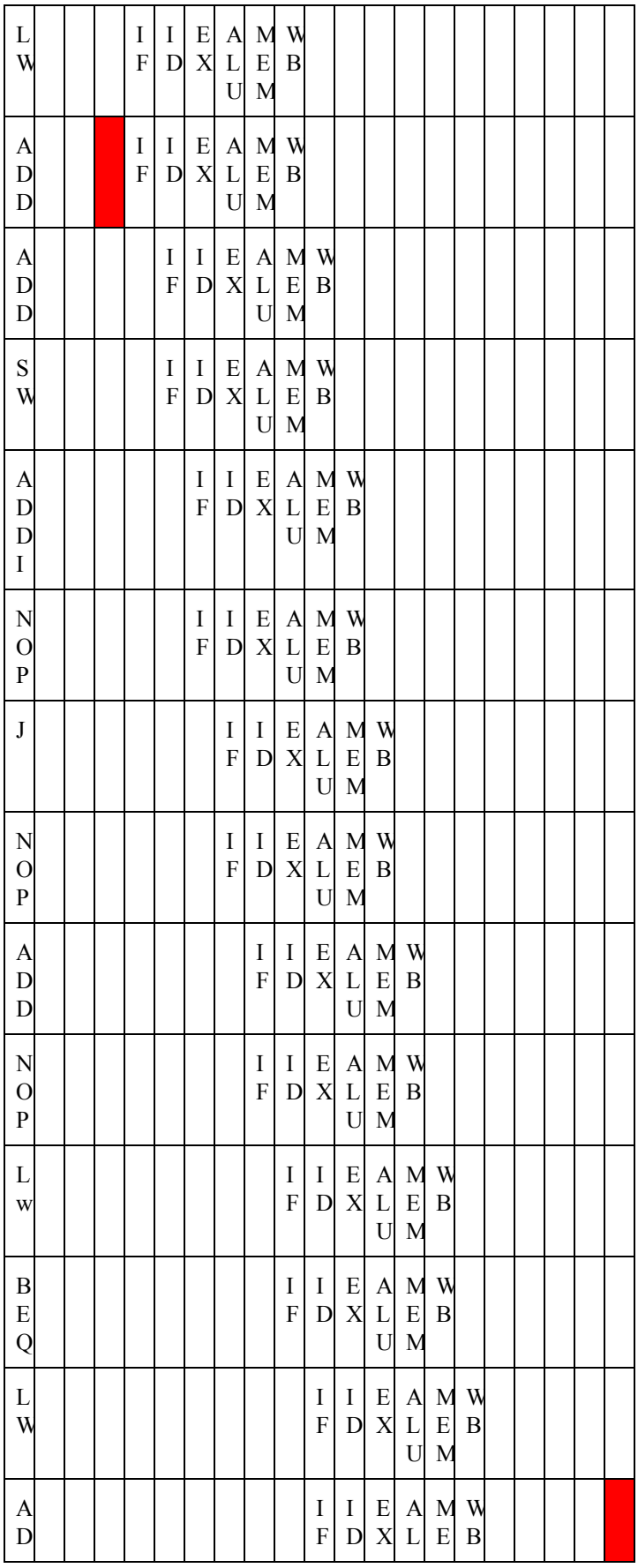
OP																													D	F	X	EM	B

34 cycles total per two iterations + 2 cycles for overhead

c. **(10 points)** Rearrange the code to achieve better performance on a 2-issue statically scheduled processor.

```
        Add r5, r5, 0
        nop
Loop:
        Add r10, r1, r5          # increment address of r1 by r5 to make r10 contain a[i]
        nop
        Lw r11, 4(r10)           # load the value of a[i+1] into r11
        Beq r5, r6 stop          # check if i = j if they are stop
        Lw r12, 8(r10)           #load the value of a[i+2] into r12
        Add r10, r2, r5          # increment the address of r2 by r5 to make r10 contain b[i]
        Add r11, r11, r12        # add a[i+1] and a[i+2] into r11
        Sw r11, 0(r10)           # store the value of r11 into b[i]
        Addi r5, r5, 2           # increment the value of i by two before checking again
        Nop
        J loop
        nop
Stop:
        exit
```

ADD	IF	ID	EX	ALU	MEM	WB													
NOP	IF	ID	EX	ALU	MEM	WB													
ADD		IF	ID	EX	ALU	MEM	WB												
NOP		IF	ID	EX	ALU	MEM	WB												
Lw			IF	ID	EX	ALU	MEM	WB											
BEQ			IF	ID	EX	ALU	MEM	WB											


LW
IF ID EX ALU MEM WB
ADD
IF ID EX ALU MEM WB
ADD
IF ID EX ALU MEM WB
SW
IF ID EX ALU MEM WB
ADDI
IF ID EX ALU MEM WB
NOP
IF ID EX ALU MEM WB
J
IF ID EX ALU MEM WB
NOP
IF ID EX ALU MEM WB
ADD
IF ID EX ALU MEM WB
NOP
IF ID EX ALU MEM WB
Lw
IF ID EX ALU MEM WB
BEQ
IF ID EX ALU MEM WB
LW
IF ID EX ALU MEM WB
AD
IF ID EX AL ME WB

D												U	M						
ADD										IF	ID	EX	ALU	MEM	WB				
SW										IF	ID	EX	ALU	MEM	WB				
ADDI											IF	ID	EX	ALU	MEM	WB			
NOP											IF	ID	EX	ALU	MEM	WB			
J												IF	ID	EX	ALU	MEM	WB		
NOP												IF	ID	EX	ALU	MEM	WB		
ADD													IF	ID	EX	ALU	MEM	WB	
NOP													IF	ID	EX	ALU	MEM	WB	
Lw														IF	ID	EX	ALU	MEM	WB
BEQ														IF	ID	EX	ALU	MEM	WB

17 cycles per 2 iterations + 1 overhead for initial add + 2 overhead for checking if finished

d. **(10 points)** What is the speedup going from the 1-issue processor to a 2-issue processor? Assume that 1,000,000 iterations of the loop are executed. Also assume perfect branch predictions on both 1-issue and 2-issue processors.

Cycles old = 34*500,000 + 2 = 17,000,002
Cycles new = 17 *500,000 + 3 = 8,500,003

Speed up = 17,000,002/ 8,500,003 = 1.99999953