Date:________________

Quiz for Chapter 4 The Processor

Not all questions are of equal difficulty. Please review the entire quiz first and then budget your time carefully.

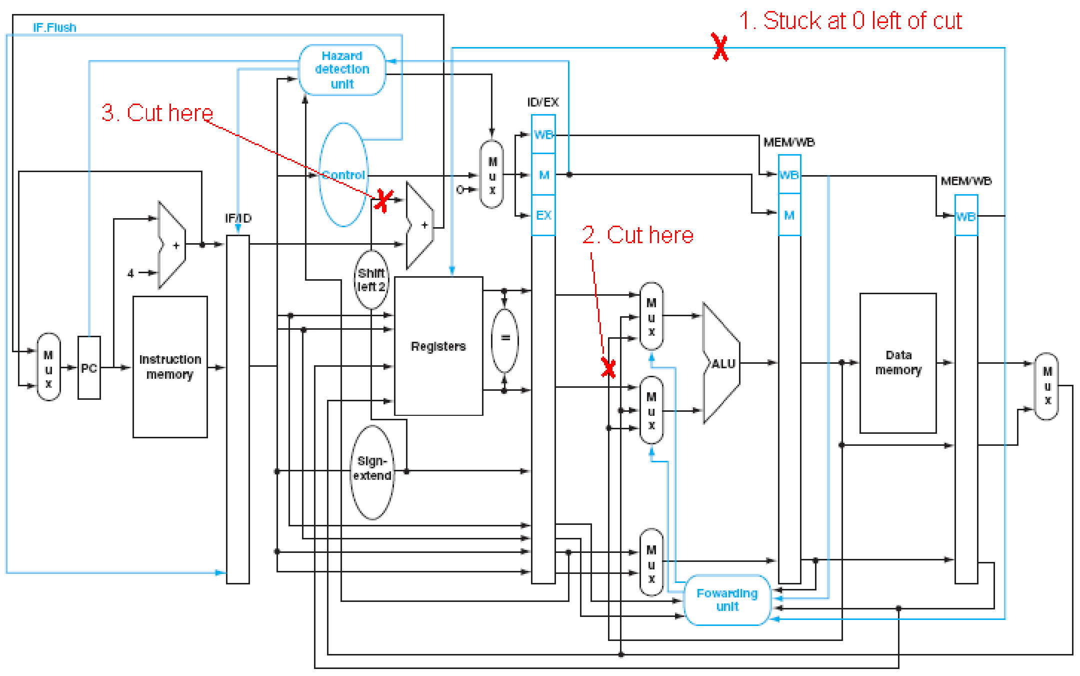
Name:____

Course:____

Solutions in RED

1. [6 points] For the MIPS datapath shown below, several lines are marked with "X". For each one:

- Describe in words the negative consequence of cutting this line relative to the working, unmodified processor.
- Provide a snippet of code that will fail
- Provide a snippet of code that will still work



(1) Cannot write to register file. This means that R-type and any instruction with write back to register file will fail. An example of code snippet that would fail is:

```
add $s1, $s2, $s3
```

An example of a code snippet that will not fail is:

```
sw $s1, 0($s2)
```

(2) Forwarding of the first operand fails. An example of code snippet that would fail is:

```
add $s1, $t0, $t1
add $s1, $s1, $s1
```

An example of code snippet that will not fail is:

```
add $s1, $t0, $t1
add $s1, $t2, $s1 # Here the second operand is forwarded correctly
```

(3) Jumping to a branch target does not work. Example of code that fails:

```
addi $s1, $zero, 2
addi $s2, $zero, 2
beq $s1, $s2, exit
```

Code that will still work:

```
addi $s1, $zero, 10
addi $s2, $zero, 20
beq $s1, $s2, exit
```

2. [3 points] Consider the following assembly language code:

```
I0: ADD R4 = R1 + R0;
I1: SUB R9 = R3 - R4;
I2: ADD R4 = R5 + R6;
I3: LDW R2 = MEM[R3 + 100];
I4: LDW R2 = MEM[R2 + 0];
I5: STW MEM[R4 + 100] = R2;
I6: AND R2 = R2 & R1;
I7: BEQ R9 == R1, Target;
I8: AND R9 = R9 & R1;
```

Consider a pipeline with forwarding, hazard detection, and 1 delay slot for branches. The pipeline is the typical 5-stage **IF, ID, EX, MEM, WB** MIPS design. For the above code, complete the pipeline diagram below (instructions on the left, cycles on top) for the code. Insert the characters **IF, ID, EX, MEM, WB** for each instruction in the boxes. Assume that there two levels of bypassing, that the second half of the decode stage performs a read of source registers, and that the first half of the write-back stage writes to the register file.

Label all data stalls (Draw an **X** in the box). Label all data forwards that the forwarding unit detects (arrow between the stages handing off the data and the stages receiving the data). What is the final execution time of the code?

Name:____

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17
I0																		
I1																		
I2																		
I3																		
I4																		
I5																		
I6																		
I7																		
I8																		

	T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17
I0	IF	ID	EX	MEM	WB													
I1		IF	ID	EX	MEM	WB												
I2			IF	ID	EX	MEM	WB											

Name:______________________________________

I3				IF	ID	EX	MEM	WB										
I4					IF	ID	X	EX	MEM	WB								
I5						IF	X	ID	EX	MEM	WB							
I6							X	IF	ID	EX	MEM	WB						
I7									IF	ID	EX	MEM	WB					
I8										IF	ID	EX	MEM	WB				

The final execution time of the code is 13 cycles.

3. [5 points] Structural, data and control hazards typically require a processor pipeline to stall. Listed below are a series of optimization techniques implemented in a compiler or a processor pipeline designed to reduce or eliminate stalls due to these hazards. For each of the following optimization techniques, state which pipeline hazards it addresses and how it addresses it. Some optimization techniques may address more than one hazard, so be sure to include explanations for all addressed hazards.

(a) Branch Prediction

It addresses control hazards by guessing the outcome of a branch instruction and then speculatively executes the instructions on one side of the branch to keep the pipeline moving. Predictions can be made in hardware or in software by the compiler.

(b) Instruction Scheduling

It addresses structural hazards and data hazards. It addresses data hazards by either moving instructions that are not dependent on an instruction, say A, before some instructions that depend on A and thus avoiding the stall that would have occurred otherwise. It addresses structural hazards by making sure instructions that use functional units that have limited number of instances are be scheduled far apart from each other and there is no unnecessary stall due to this. It can be done in hardware (superscalar processor) or statically by the compiler

(c) delay slots

It addresses control hazards. It helps to avoid a stall that would result due branch target identification during the decode stage by scheduling the execution of some other instruction which anyway has to execute irrespective of the branch condition.

Name:______________________________________

(d) increasing availability of functional units (ALUs, adders etc)

It helps to avoid structural hazards. It is possible to run multiple instructions of the same type at the same time if we have replicated functional units

(e) caches

It addresses data hazards. In particular, caches help to reduce memory latency and hence reduce the load-use latency which in turn reduce the stall duration and improves execution time (by maintaining pipeline steady state).

4. [6 points] Branch Prediction. Consider the following sequence of actual outcomes for a single static branch. T means the branch is taken. N means the branch is not taken. For this question, assume that this is the only branch in the program.

T T T N T N T T T N T N T T T N T N

(a) Assume that we try to predict this sequence with a BHT using one-bit counters. The counters in the BHT are initialized to the N state. Which of the branches in this sequence would be mis-predicted?

You may use this table for your answer:

Predictor state before prediction	Branch outcome	Misprediction?
N	T	
	T	
	T	
	N	
	T	
	N	
	T	
	T	
	T	
	N	
	T	
	N	
	T	
	T	
	T	
	N	
	T	
	N	

Predictor state before prediction	Branch outcome	Misprediction?
N	T	Y
T	T	N
T	T	N
T	N	Y
N	T	Y

T	N	Y
N	T	Y
T	T	N
T	T	N
T	N	Y
N	T	Y
T	N	Y
N	T	Y
T	T	N
T	T	N
T	N	Y
N	T	Y
T	N	Y

(b) Now, assume a two-level branch predictor that uses one bit of branch history—i.e., a one-bit BHR. Since there is only one branch in the program, it does not matter how the BHR is concatenated with the branch PC to index the BHT. Assume that the BHT uses one-bit counters and that, again, all entries are initialized to N. Which of the branches in this sequence would be mis-predicted? Use the table below.

Predictor state before prediction	**BHR**	**Branch outcome**	**Misprediction?**
N	N	T	
		T	
		T	
		N	
		T	
		N	
		T	
		T	
		T	
		N	
		T	
		N	
		T	
		T	
		T	
		N	
		T	
		N	

Predictor state before prediction	**BHR**	**Branch outcome**	**Misprediction?**
N	N	T	Y
T	T	T	Y
T	T	T	N
T	T	N	Y
N	N	T	N
T	T	N	N

N	N	T	N
T	T	T	Y
T	T	T	N
T	T	N	Y
N	N	T	N
T	T	N	N
N	N	T	N
T	T	T	Y
T	T	T	N
T	T	N	Y
N	N	T	N
T	T	N	N

(c) What is a return-address-stack? When is a return address stack updated?

A return address stack is a hardware mechanism used to predict the target of return-from-function-call instructions. The return address stack is updated after either a call instruction or a return instruction is *predicted.*

5. [16 points] The classic 5-stage pipeline seen in Section 4.5 is IF, ID, EX, MEM, WB. This pipeline is designed specifically to execute the MIPS instruction set. MIPS is a load store architecture that performs one memory operation per instruction, hence a single MEM stage in the pipeline suffices. Also, its most common addressing mode is register displacement addressing. The EX stage is placed before the MEM stage to allow it to be used for address calculation. In this question we will consider a variation in the MIPS instruction set and the interactions of this variation with the pipeline structure.

The particular variation we are considering involves swapping the MEM and EX stages, creating a pipeline that looks like this: IF, ID, MEM, EX, WB. This change has two effects on the instruction set. First, it prevents us from using register displacement addressing (there is no longer an EX in front of MEM to accomplish this). However, in return we can use instructions with one memory input operand, i.e., register-memory instructions. For instance: `multf_m f0,f2,(r2)` multiplies the contents of register `f2` and the value at memory location pointed to by `r2`, putting the result in `f0`.

(a) Dropping the register displacement addressing mode is potentially a big loss, since it is the mode most frequently used in MIPS. Why is it so frequent? Give two popular software constructs whose implementation uses register displacement addressing (i.e., uses displacement addressing with non-zero displacements).

Two popular software constructs that use register displacement addressing are:

- Use of arrays and loops stepping through them using an index variable
- Indirect jumps that are used for implementing switch..case constructs in a high level languages may need to access a specific indexed entry in a jump table using register displacement mode.

(b) What is the difference between a dependence and a hazard?

An instruction A is said to be dependent on an instruction B if the A's execution is determined by some condition computed by B or if A uses some data value that is produced by B. A hazard is situation which prevents the pipelined execution of an program and causes a stall. Hazards are usually a consequence of having data dependencies between instructions, but it is possible for hazards to manifest on an architecture even though intrinsically there are no dependences between instructions due to limitations in the number of resources (registers/functional units).

(c) In this question we will work with the SAXPY loop.

```
do I = 0,N
Z[I] = A*X[I] + Y[I]
```

Here is the new assembly code.

```
0: slli r2,r1,#3 // I is in r1
1: addi r3,r2,#X
2: multf_m f2,f0,(r3) // A is in f0
3: addi r4,r2,#Y
4: addf_m f4,f2,(r4)
5: addi r4,r2,#Z
6: sf f4,(r4)
7: addi r1,r1,#1
8: slei r6,r1,r5 // N is in r5
9: bnez r6,#0
```

Using the instruction numbers, label the data and control dependences.

Data Dependencies:

Flow Dependencies: $0 \rightarrow 1$, $1 \rightarrow 2$, $2 \rightarrow 3$, $3 \rightarrow 4$, $0 \rightarrow 3$, $0 \rightarrow 5$, $5 \rightarrow 6$, $7 \rightarrow 8$, $8 \rightarrow 9$

Anti Dependences: $1 \rightarrow 7$, $4 \rightarrow 5$

Output Dependences: $3 \rightarrow 5$

(d) Fill in the pipeline diagram for code for the new SAXPY loop. Label the stalls as d* for data-hazard stalls and s* for structural stalls. What is the latency of a single iteration? (The number of cycles between the completion of two successive #0 instructions). For this question, assume that FP addition takes 2 cycles, FP multiplication takes 3 cycles and that all other operations take a single cycle. The functional units are not pipelined. The FP adder, FP multiplier and integer ALU are all separate functional units, such that there are no structural hazards between them. The register file is written by the WB stage in the first half of a clock cycle and is read by the ID stage in the second half of a clock cycle. In addition, the processor has full forwarding. The processor stalls on branches until the outcome is available which is at the end of the EX stage. The processor has no provisions for maintaining "precise state".

Name:______________________________________

instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0: slli r2,r1,#3	F	D	M	X	W																			
1: addi r3,r2,#X		F	D	M	X	W																		
2: mulf_m f2,f0,(r3)			F	D		M	X	X	X	W														
3: addi r4,r2,#Y				F		D	M	X	W															
4: addf_m f4,f2,(r4)						F	D		M	X	X	W												
5: addi r4,r2,#Z							F		D	M	X		W											
6: sf f4,(r4)									F	D		M	X	W										
7: addi r1,r1,#1										F		D	M	X	W									
8: slei r6,r1,r5												F	D	M	X	W								
9: bnez r6, #0													F	D	M	X	W							
0: slli r2,r1,#3																	F	D	M	X	W			

instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0: slli r2,r1,#3	F	D	M	X	W																			
1: addi r3,r2,#X		F	D	M	X	W																		
2: mulf_m f2,f0,(r3)			F	D	d*	M	X	X	X	W														
3: addi r4,r2,#Y				F	s*	D	M	X	W															
4: addf_m f4,f2,(r4)						F	D	d*	M	X	X	W												
5: addi r4,r2,#Z							F	s*	D	M	X	*s	W											
6: sf f4,(r4)									F	D	d*	M	X	W										
7: addi r1,r1,#1										F	s*	D	M	X	W									
8: slei r6,r1,r5												F	D	M	X	W								
9: bnez r6, #0													F	D	M	X	W							
0: slli r2,r1,#3																	F	D	M	X	W			

(e) In the pipeline for MIPS mentioned in the text, what is the reason for forcing non-memory operations to go through the MEM stage rather than proceeding directly to the WB stage?

To facilitate pipelining by removing structural hazards, every instruction should access a structure at most once and always in the same pipeline stage. The structure here is the register write port.

(f) Aside from the direct loss of register displacement addressing and the subsequent instructions required to explicitly compute addresses, what are two other disadvantages of this sort of pipeline?

The branch penalty is longer since the EX and IF stages are now further away. In addition, load/ALU stall that could not be corrected via forwarding in the original pipeline is now an ALU/load-store stall that cannot be corrected via forwarding. There are also ISA problems with asymmetric operands which violates the regularity and orthogonality principles.

(g) Reduce the stalls by pipeline scheduling a single loop iteration. Show the resulting code and fill in the pipeline diagram. You do not need to show the optimal schedule for a correct response.

Here is an example. The key is that multf_m (which is a 3 cycle operation) is moved 3 instructions ahead of addf_m. addf_m (which is a 2 cycle operation) remains 2 instructions ahead of sf. Notice, the

Name:______________________________________

branch is still at the end, and no RAW dependences are violated, although one had to be resolved via renaming.

```
0: slli r2,r1,#3 // I is in r1
1: addi r3,r2,#X
2: addi r4,r2,#Y
3: multf_m f2,f0,(r3) // A is in f0
4: addi r8,r2,#Z
5: addi r1,r1,#1
6: addf_m f4,f2,(r8)
7: slei r6,r1,r5 // N is in r5
8: sf f4,(r4)
9: bnez r6,#0
```

instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0: slli r2,r1,#3	F	D	M	X	W																			
1: addi r3,r2,#X		F	D	M	X	W																		
2: addi r4,r2,#Y			F	D	M	X	W																	
3: mulf_m f2,f0,(r3)				F	D	M	X	X	X	W														
4: addi r8, r2, #Z					F	D	M	X	W															
5: addi r1,r1,#1						F	D	M	X	s*	W													
6: addf_m f4,f2,(r4)							F	D	M	X	X	W												
7: slei r6,r1,r5								F	D	M	X	s*	W											
8: sf f4,(r4)									F	D	M	X	W											
9: bnez r6, #0										F	D	M	X	W										
0: slli r2,r1,#3														F	D	M	X	W						

6. [8 points] A two-part question.

(Part A) Dependence detection

This question covers your understanding of dependences between instructions. Using the code below, list all of the dependence types (RAW, WAR, WAW). List the dependences in the respective table (example INST-X to INST-Y) by writing in the instruction numbers involved with the dependence.

```
I0: A = B + C;
I1: C = A - B;
I2: D = A + C;
I3: A = B * C * D;
I4: C = F / D;
I5: F = A ^ G;
I6: G = F + D;
```

RAW Dependence		WAR Dependence		WAW Dependence	
From Instr	To Instr	From Instr	To Instr	From Instr	To Instr

Name:____

RAW Dependence		WAR Dependence		WAW Dependence	
From Instr	To Instr	From Instr	To Instr	From Instr	To Instr
I0	I1	I0	I1	I0	I3
I0	I2	I1	I3	I1	I4
I1	I2	I2	I4		
I3	I5	I3	I4		
I2	I3	I4	I5		
I1	I3	I5	I6		
I2	I4				
I3	I5				
I5	I6				

(Part B) Dependence analysis

Given four instructions, how many unique comparisons (between register sources and destinations) are necessary to find all of the RAW, WAR, and WAW dependences. Answer for the case of four instructions, and then derive a general equation for N instructions. Assume that all instructions have one register destination and two register sources.

For four instructions, the number of unique comparisons:

$(2(3) + 2(2) + 2(1)) + (2(3) + 2(2) + 2(1)) + (3 + 2 + 1) = 30$

The first summand is for RAW comparisons, the second summand is for WAR comparisons and the last summand is for WAW comparisons.

The general equation for N instructions = $(5*(n-1)*n)/2$

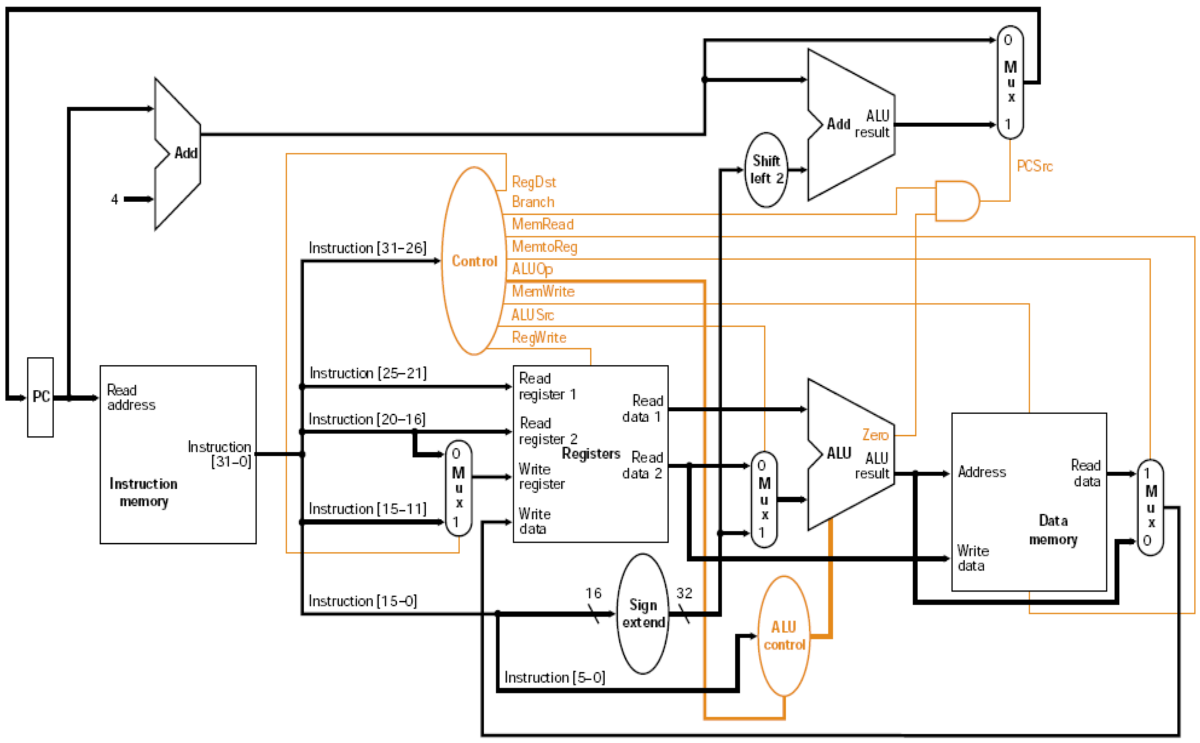
7. [10 points] This is a three-part question about critical path calculation. Consider a simple single-cycle implementation of MIPS ISA. The operation times for the major functional components for this machine are as follows:

Component	Latency
ALU	10 ns
Adder	8 ns
ALU Control Unit	2 ns
Shifter	3 ns
Control Unit/ROM	4 ns
Sign/zero extender	3 ns
2-1 MUX	2 ns
Memory (read/write) (instruction or data)	15 ns
PC Register (read action)	1 ns
PC Register (write action)	1 ns
Register file (read action)	7 ns
Register file (write action)	5 ns
Logic (1 or more levels of gates)	1 ns

Name:______________________________________

Below is a copy of the MIPS single-cycle datapath design. In this implementation the clock cycle is determined by the longest possible path in the machine. The critical paths for the different instruction types that need to be considered are: R-format, Load-word, and store-word. All instructions have the same instruction fetch and decode steps. The basic register transfer of the instructions are:

```
Fetch/Decode: Instruction <- IMEM[PC];
R-type: R[rd] <- R[rs] op R[rt]; PC <- PC + 4;
load: R[rt] <- DMEM[ R[rs] + signext(offset)]; PC <- PC +4;
store: DMEM[ R[rs] + signext(offset)] <- R[Rt]; PC <- PC +4;
```



(Part A)

In the table below, indicate the components that determine the critical path for the respective instruction, in the order that the critical path occurs. If a component is used, but not part of the critical path of the instruction (ie happens in parallel with another component), it should not be in the table. The register file is used for reading and for writing; it will appear twice for some instructions. All instruction begin by reading the PC register with a latency of 2ns.

Instruction Type	**Hardware Elements Used By Instruction**											
R-Format												
Load												
Store												

Name:____

Instruction Type	Hardware Elements Used By Instruction											
R-Format	PC	Instruction Memory	Control	Read regs	ALU control	ALU	Control	Reg write				
Load	PC	Instruction Memory	Control	Read Reg	ALU control	ALU	Data Memory	Control	Reg Write			
Store	PC	Instruction Memory	Control	Read Regs	ALU control	ALU	Read Reg	Control	Data Memory			

(Part B)

Place the latencies of the components that you have decided for the critical path of each instruction in the table below. Compute the sum of each of the component latencies for each instruction.

Instruction Type	Hardware Latencies For Respective Elements											Total
R-Format	2 ns											
Load	2 ns											
Store	2 ns											

Instruction Type	Hardware Latencies For Respective Elements											Total
R-Format	2 ns	15 ns	4 ns	7 ns	2 ns	10 ns	4 ns	5 ns				
Load	2 ns	15 ns	4 ns	7 ns	2 ns	10 ns	15 ns	4 ns	5 ns			
Store	2 ns	15 ns	4 ns	7 ns	2 ns	10 ns	7 ns	4 ns	15 ns			

(Part C)

Use the total latency column to derive the following critical path information:

- Given the data path latencies above, which instruction determines the overall machine critical path (latency)?
- What will be the resultant clock cycle time of the machine based on the critical path instruction?
- What frequency will the machine run?

The store instruction determines the overall machine critical path latency since it has the maximum latency.

The clock cycle time should be long enough to permit the critical path instruction to complete. So it should be at least 66ns

The frequency of the machine would be (1/66) ns = 15.1 MHz

8. [18 points] This problem covers your knowledge of branch prediction.

The figure below illustrates three possible predictors.

- Last taken predicts taken when 1
- Up-Down (saturating counter) predicts taken when 11 and 10

Name:____

- Automata A3 predicts taken when 11 and 10

Fill out the tables below and on the next page for each branch predictor. The execution pattern for the branch is NTNNTTTN.

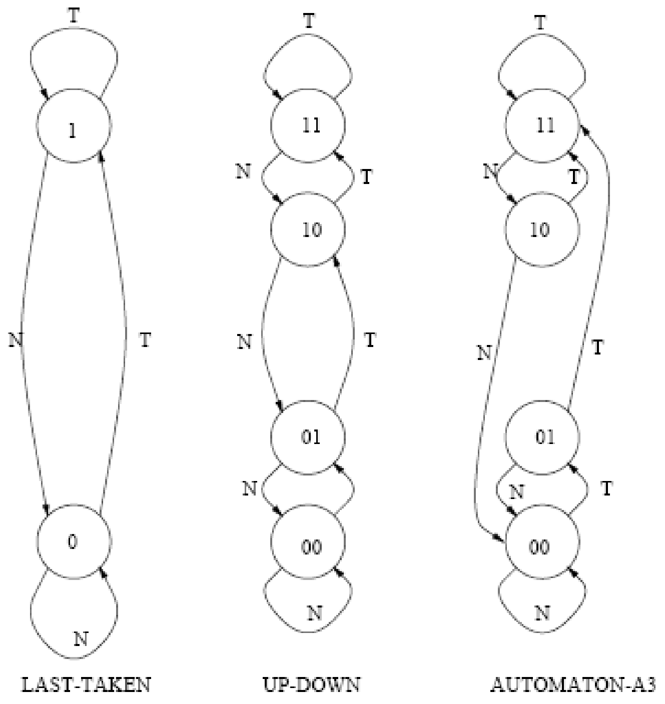


Table 1: Table for last-taken branch predictor

Execution Time	Branch Execution	State Before	Prediction	Correct or Incorrect	State After
0	N	0			
1	T				
2	N				
3	N				
4	T				
5	T				
6	T				
7	N				

Execution Time	Branch Execution	State Before	Prediction	Correct or Incorrect	State After
0	N	0	NT	Correct	0
1	T	0	NT	Incorrect	1
2	N	1	T	Incorrect	0
3	N	0	NT	Correct	0
4	T	0	NT	Incorrect	1
5	T	1	T	Correct	1
6	T	1	T	Correct	1
7	N	1	T	Incorrect	0

Name:_______________________________________

Table 2: Table for saturating counter (up-down) branch predictor.

Execution Time	Branch Execution	State Before	Prediction	Correct or Incorrect	State After
0	N	01			
1	T				
2	N				
3	N				
4	T				
5	T				
6	T				
7	N				

Execution Time	Branch Execution	State Before	Prediction	Correct or Incorrect	State After
0	N	01	NT	Correct	00
1	T	00	NT	Incorrect	01
2	N	01	NT	Correct	00
3	N	00	NT	Correct	00
4	T	00	NT	Incorrect	01
5	T	01	NT	Incorrect	10
6	T	10	T	Correct	11
7	N	11	T	Incorrect	10

Table 3: Table for Automata-A3 branch predictor.

Execution Time	Branch Execution	State Before	Prediction	Correct or Incorrect	State After
0	N	01			
1	T				
2	N				
3	N				
4	T				
5	T				
6	T				
7	N				

Execution Time	Branch Execution	State Before	Prediction	Correct or Incorrect	State After
0	N	01	NT	Correct	00
1	T	00	NT	Incorrect	01
2	N	01	NT	Correct	00
3	N	00	NT	Correct	00
4	T	00	NT	Incorrect	01
5	T	01	NT	Incorrect	11
6	T	11	T	Correct	11
7	N	11	T	Incorrect	10

Calculate the prediction rates of the three branch predictors:

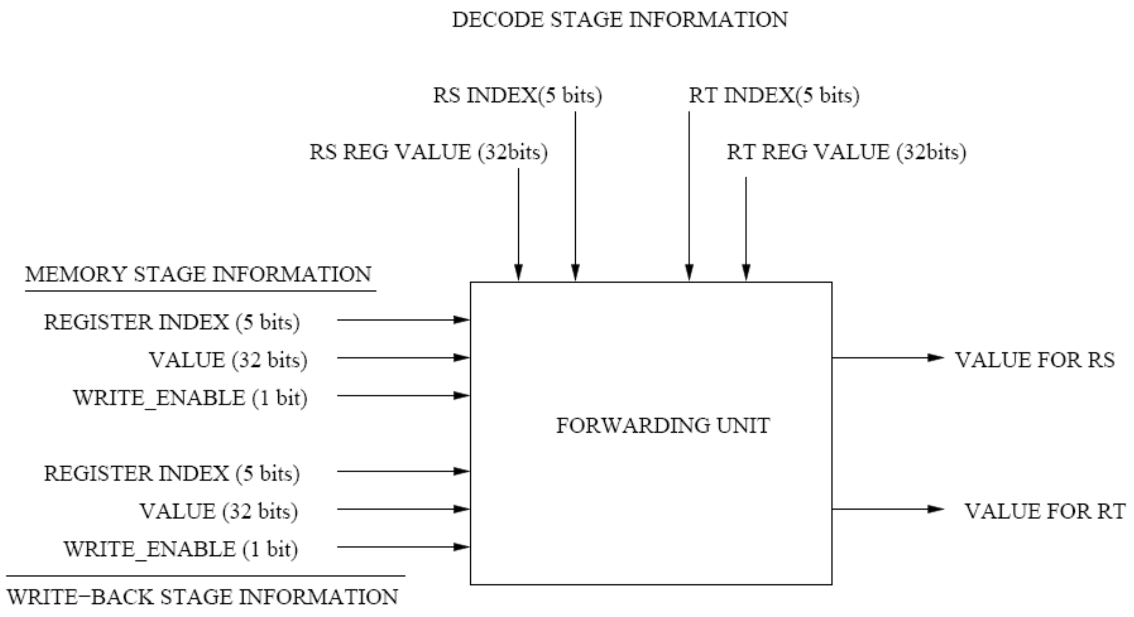
Predictor	Prediction Accuracy
Last-taken	
Up-Down	
Automata-A3	

Predictor	Prediction Accuracy
Last-taken	0.5
Up-Down	0.5
Automata-A3	0.5

9. [3 points] Pipelining is used because it improves instruction throughput. Increasing the level of pipelining cuts the amount of work performed at each pipeline stage, allowing more instructions to exist in the processor at the same time and individual instructions to complete at a more rapid rate. However, throughput will not improve as pipelining is increased indefinitely. Give two reasons for this.

Pipelining has a fixed (or relatively fixed) absolute overhead per stage which results from latch overhead and clock/data skew. This means that the latency of a pipeline stage cannot be driven to zero. Second, increasing the pipeline depth lengthens hazard penalties, increasing the CPI. For instance, increasing the depth of the pipeline between the fetch and execute stage increases the branch mis-prediction penalty.

10. [3 points] Forwarding logic design. For this problem you are to design a forwarding unit for a 5-stage pipeline processor. The forwarding unit returns the value to be forwarded to the current instruction. There are three places that the values for register RS and register RT can come from: decode stage (register file), memory stage, and write-back stage.



The write-back and memory stage information consists of: _INDEX- explaining which inflight register index is to be written _VALUE- the value that is to be written _ENABLE- whether or not the instruction in the stage is writing.

Name:____

The decode stage simply states the register index (for RS and RT) and the corresponding register value from the register file.

Generally three values could exist, one of which the forwarding unit should choose for each of the RS and RT register value requests. The memory stage has value MEM, the write-back stage has value WB, and the register file has value RS-REG or RT-REG.

Using the table below which contains information about all of the instruction stages, indicate which value should be forwarded to the current instruction: MEM, WB, RS-REG, or RT-REG. Each line represents a Forwarding unit evaluation; there is no connection between evaluation lines in the table. You do not need to worry about hazard detection, only value bypassing.

	Mem Stage		**Write-Back Stage**		**Register Stage**		**RS Value**	**RT Value**
Evaluation	Index	Write	Index	Write	RS-Index	RT-Index		
0	5	1	23	0	6	7		
1	7	0	16	1	16	8		
2	10	1	10	1	11	10		
3	17	0	12	1	12	12		
4	19	0	19	0	19	25		

	Mem Stage		**Write-Back Stage**		**Register Stage**		**RS Value**	**RT Value**
Evaluation	Index	Write	Index	Write	RS-Index	RT-Index		
0	5	1	23	0	6	7	RS-REG	RT-REG
1	7	0	16	1	16	8	WB	RT-REG
2	10	1	10	1	11	10	RS-REG	MEM
3	17	0	12	1	12	12	WB	WB
4	19	0	19	0	19	25	MEM	RT-REG

11. [12 points] Consider a MIPS machine with a 5-stage pipeline with a cycle time of 10ns. Assume that you are executing a program where a fraction, f, of all instructions immediately follow a load upon which they are dependent.

(a) With forwarding enabled what is the total execution time for N instructions, in terms of f ?

When pipeline is filled,

(1 – f)*N instructions take 1 cycle

f*N instructions take 2 cycles (including 1 cycle for load-use stall)

Total cycles = (1-f)*N + 2*f*N + 4 (then number of cycles to fill the pipeline)

Total time = 10 *(N*(1+f) + 4)

(b) Consider a scenario where the MEM stage, along with its pipeline registers, needs 12 ns. There are now two options: add another MEM stage so that there are MEM1 and MEM2 stages or increase the cycle time to 12ns so that the MEM stage fits within the new cycle time and the number of pipeline stages remain unaffected. For a program mix with the above characteristics, when is the first option better than the second. Your answer should be based on the value of f.

Let us ignore the fill cycles to simplify the calculations.

Option 1 : Time = ((1-f)*N + 3*f*N) * 10 = (1 + 2*f)*10*N

(additional load-use latency contributes to 1*f*N)

Option 2 : Time = ((1-f)*N + 2*f*N)*12 = (1 + f)* 12*N

Option 1 is better when (1+2*f)*10x < (1+f)*12*x
⇨ f < ¼

(c) Embedded processors have two different memory regions – a faster scratchpad memory and a slower normal memory. Assume that in the 6 stage machine (with MEM1 and MEM2 stages), there is a region of memory that is faster and for which the correct value is obtained at the end of the MEM1 stage itself while the rest of the memory needs both MEM1 and MEM2 stages. For the sake of simplicity assume that there are two load instructions load.fast and load.slow that indicate which memory region is accessed. If 40% of the fraction f mentioned above get their value from the fast memory, how does the answer to the previous question change ?

Option 1 : Time = ((1-f)*N + 0.4*2*f*N + 0.6*3*f*N)*10
= (1 + 1.6*f)*10*N

Option 1 is better when
(1 + 1.6*f)*10*N < (1 + f)*12*N
⇨ f < ½

12. [10 points] You are given a 4-stage pipelined processor as described below.

- IF: Instruction Fetch
- IDE: Instruction Decode, Register Fetch, ALU evaluation, branch instructions change PC, address calculation for memory access.
- MEM: memory access for load and store instructions.
- WB: Write the execution result back to the register file. The writeback occurs at the 2nd half of the cycle.

Assume the delayed branching method discussed in Section 4.3. For the following program, assume that the loop will iterate 15 times. Assume that the pipeline finishes one instruction every cycle except when a branch is taken or when an interlock takes place. An interlock prevents instructions from being executed in the wrong sequence to preserve original data dependencies. Assume register bypass from both the IDE output and the MEM output. Also assume that r2 will not be needed after the execution returns.

A		ADD	$r3 \leftarrow 0$
B		LOAD	$r1 \leftarrow M(0200)$
C	Loop:	LOAD	$r2 \leftarrow M(0208+r1)$
D		ADD	$r3 \leftarrow r2 + r3$
E		SUB	$r1 \leftarrow r1 - 4$
F		BRANCH! =	$PC \leftarrow$ Loop if $r1 != 0$
G		NOP	a delay slot to be filled
H		STORE	$M(0204) \leftarrow r3$
I		RETURN	return from function
J		NOP	a delay slot to be filled

Name:______________________________________

(a) Is there any interlock cycle in the program? If so, perform code reordering on the program and show the new program without interlock cycle.

There is an interlock cycle in the program resulting out of a dependence between instruction C and D, for which there is a stall of 1 cycle. By reordering the code in the following way:

```
A               ADD             r3 <- 0
B               LOAD            r1 <- M(0200)
C Loop:         LOAD            r2 <- M(0208 + r1)
E               SUB             r1 <- r1 - 4
D               ADD             r3 <- r2 + r3
F               BRANCH !=       PC <- Loop if r1 != 0
G               NOP             a delay slot
H               STORE  M(0204) <- r3
I               RETURN return from function
J               NOP             a delay slot
```

the interlock cycle can be eliminated

(b) Derive the total number of cycles required to execute all instructions before and after you eliminated the interlock cycle.

The number of cycles to execute before the transformation = 4 + 2 + 6*15 + 3 = 99 cycles. The number of cycles after the transformation = 4 + 2 + 5*15 + 3 = 84 cycles

(c) Fill the delay slots. Describe the code reordering and/or duplication performed. Show the same program after delay slot filling. Recall that RETURN is also a branch instruction. Use a ' to mark the new copy of a duplicated instruction. For example, if you duplicated D, name the new copy D'.

```
A               ADD             r3 <- 0
B               LOAD    r1 <- M(0200)
C Loop:         LOAD            r2 <- M(0208 + r1)
E               SUB             r1 <- r1 - 4
F               BRANCH !=       PC <- Loop if r1 != 0
D               ADD             r3 <- r2 + r3
I               RETURN return from function
H               STORE  M(0204) <- r3
```

(d) Derive the total number of cycles required to execute all instructions after you filled the delay slots.

The total number of cycles after filling in delay slots = 4 + 2 + 4*15 + 2 = 68 cycles

13. [3 points] Using any ILP optimization, double the performance of the following loop, or explain why it is not possible. The machine can only do one branch per cycle, but has infinite resources otherwise.

```
r1 = ... ; r1 is head pointer to a linked list
r3 = 0
LOOP:
r2 = M[r1 + 8]
r3 = r3 + r2
r1 = M[r1]
branch r1 != 0, LOOP
... = r3 ; r3 is used when loop complete
```

By performing the following code reordering we can avoid a 1-cycle stall due to a dependence between the first and second instructions of the loop:

```
r1 = ... ; r1 is head pointer to a linked list
r3 = 0
LOOP:
r3 = r3 + r2
r2 = M[r1 + 8]
r1 = M[r1]
branch r1 != 0, LOOP
... = r3 ; r3 is used when loop complete
```

14. [10 points] Consider the datapath below. This machine does not support code with branch delay slots. (It predicts not-taken with a 1-cycle penalty on taken branches.) For each control signal listed in the table below, determine its value at cycles 3 through 9, inclusive. Also, show the instruction occupying each stage of the pipeline in all cycles. (Assume the IF/ID write-enable line is set to the inverse of the Stall signal.)

The initial state of the machine is:

```
PC = 0
```

All pipeline registers contain 0s
All registers in the register file contain 0s.
The data memory contains 0s in all locations
The instruction memory contains:

```
00: addiu $3, $zero, 4
04: lw $4, 100($3)
08: addu $2, $4, $3
0C: beq $4, $zero, 0x14
10: addiu $3, $3, 1
14: addu $2, $2, $3
```

all other locations contain 0

Use data forwarding whenever possible. All mux inputs are numbered vertically from “top” to “bottom” starting at 0 as you look at the datapath in the proper landscape orientation. Also, the values for ALUOp are:

Value	Desired ALU Action
00	Add
01	Subtract
10	Determine by decoding funct field

Instruction formats:

Name:______________________________________

	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
R:	op	rs	rt	rd	shamt	funct
I:	op	rs	rt	address / immediate		
J:	op	target address				

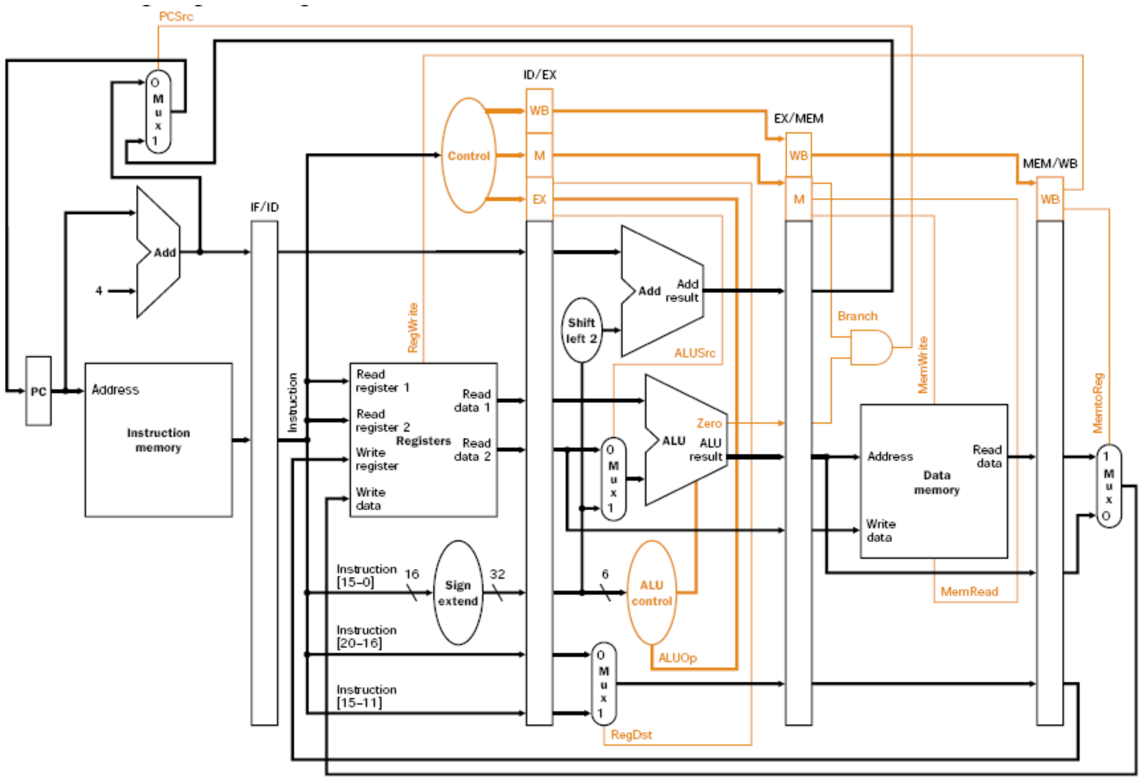
Time	**PCWrite**	**IF.Flush**	**Branch**	**Stall**	**ALUSrc**	**ALUOp**	**RegDst**	**ForwardA**	**ForwardB**	**MemRead**	**MemWrite**	**MemtoReg**	**RegWrite**
	IF		ID		EX					M		WB	
0	00: addiu		–		–					–		–	
	1	0	0	0	0	00	0	00	00	0	0	0	0
1	04: lw		00: addiu		–					–		–	
	1	0	0	0	0	00	0	00	00	0	0	0	0
2	08: addu		04: lw		00: addiu					–		–	
	1	0	0	0	1	00	0	00	00	0	0	0	0
3													
4													
5													
6													
7													
8													
9													

Time	**PCWrite**	**IF.Flush**	**Branch**	**Stall**	**ALUSrc**	**ALUOp**	**RegDst**	**ForwardA**	**ForwardB**	**MemRead**	**MemWrite**	**MemtoReg**	**RegWrite**
	IF		ID		EX					M		WB	
0	00: addiu		–		–					–		–	
	1	0	0	0	0	00	0	00	00	0	0	0	0
1	04: lw		00: addiu		–					–		–	
	1	0	0	0	0	00	0	00	00	0	0	0	0
2	08: addu		04: lw		00: addiu					–		–	
	1	0	0	0	1	00	0	00	00	0	0	0	0

Name:______________________________________

3	0C: beq		08: addu		04: lw					00: addiu		–	
	0	0	0	1	1	00	0	10	0	X	0	X	0
4	0C: beq		08:addu		LOAD-USE STALL (NOP)					04:lw		00:addiu	
	1	0	0	0	X	X	X	X	X	1	0	1	1
5	10: addiu		0C: beq		08: addu					NOP		04: lw	
	1	1	1	0	1	10	1	01	00	X	0	0	1
6	14: addu		TAKEN BR (NOP)		0C : beq					08: addu			
	1	0	0	0	X	X	X	X	X	X	0	X	0
7	18:nop		14: addu		NOP					0C : beq		08: addu	
	1	0	0	0	X	XX	X	X	X	X	0	1	1
8	1c:nop		18:nop		14: addu					NOP		0C : beq	
	1	0	0	0	1	10	1	00	00	X	0	X	0
9	20:nop		1c:nop		18:nop					14: addu		NOP	
	1	0	0	0	X	XX	X	X	X	X	0	X	0

15. [6 points] Consider the following data path diagram:



(a) Discuss the functionality of the RegDst and the ALUSrc control wires.

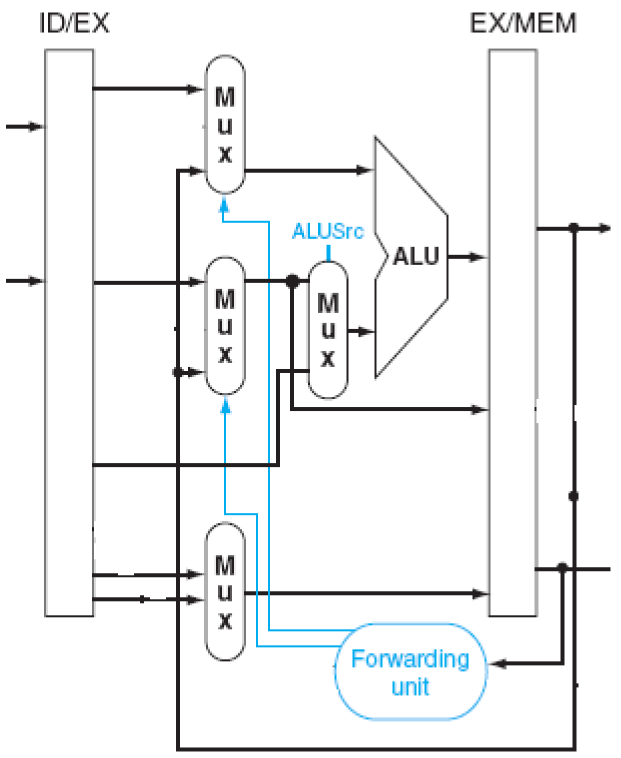
The RegDst control signal is used to specify the register destination number for instructions that have a register as their destination operand (rd).

The ALUSrc control signal is used to choose between either the second register operand from the instruction or the sign-extended lower 16 bits of the instruction.

Name:______________________________________

(b) Next, modify the diagram to indicate the datapath changes (and any additional multiplexing) needed to provide bypassing from EX to EX for all possible RAW hazards on arithmetic instructions. How does ALUSrc change when bypassing is added?

The ID/EX and EX/MEM latches will have a new forwarding unit between them directing the outputs of the EX stage into the MUX's that feed the inputs into the ALU. The ALUSrc control signal will not have to be modified since it is already the output from one of the MUX's. The modification (only the relevant part of the diagram is shown below) is as follows:



16. [3 points] Imagine an instruction whose function is to read four adjacent 32-bit words from memory and places them into four specified 32-bit architectural registers. Assuming the 5-stage pipeline is filled with these instructions and these instructions ONLY, what is the minimum number of register file read and write ports that would be required?

The minimum number of write ports is four and the minimum number of read ports is two (in this case an additional read port is to read the register that holds the memory address)

17. [12 points] Pipelining and Bypass. In this question we will explore how bypassing affects program execution performance. To begin consider the standard MIPS 5 stage pipeline. For your reference, refer to the figure below. For this question, we will use the following code to evaluate the pipeline's performance:

```
1 add $t2, $s1, $sp
2 lw $t1, $t1, 0
3 addi $t2, $t1, 7
4 add $t1, $s2, $sp
```

```
5 lw $t1, $t1, 0
6 addi $t1, $t1, 9
7 sub $t1, $t1, $t2
```

(a) What is the load-use latency for the standard MIPS 5-stage pipeline?

The load use latency for the standard MIPS 5-stage pipeline is 1 cycle.

(b) Once again, using the standard MIPS pipeline, identify whether the value for each register operand is coming from the bypass or from the register file. For clarity, please write REG or BYPASS in each box.

Instruction	Src Operand 1	Src Operand 2
1		
2		N/A
3		N/A
4		
5		N/A
6		N/A
7		

Instruction	Src Operand 1	Src Operand 2
1	Register File	Register File
2	Register File	N/A
3	Bypass	N/A
4	Register File	Register File
5	Bypass	N/A
6	Bypass	N/A
7	Bypass	Register File

(c) How many cycles will the program take to execute on the standard MIPS pipeline?

It takes 13 cycles to execute on the standard MIPS pipeline

(d) Assume, due to circuit constraints, that the bypass wire from the memory stage back to the execute stage is omitted from the pipeline. What is the load-use latency for this modified pipeline?

The load use latency for this pipeline is 2 cycles

(e) Identify whether the value for each register operand is coming from the bypass or from the register file for the modified pipeline. For clarity, please write REG or BYPASS in each box.

Instruction	Src Operand 1	Src Operand 2
1		
2		N/A
3		N/A
4		
5		N/A
6		N/A
7		

Name:____

Instruction	Src Operand 1	Src Operand 2
1	Register File	Register File
2	Register File	N/A
3	Register File	N/A
4	Register File	Register File
5	Bypass	N/A
6	Register File	N/A
7	Bypass	Register File

(f) How long does the program take to execute on the modified pipeline?

It takes 15 cycles to execute on the modified pipeline.

18. [3 points] Scheduling. “A schedule defends from chaos and whim.”—Annie Dillard

Consider this pseudo-assembly snippet:

```
1: mov r1 = 0 ;; iteration count i
2: mov r2 = 0 ;; initialize x
loop:
3: sll r3 = r1, 2 ;; r3 = r1*4
4: lw r4 = A(r3) ;; load A[i]
5: add r2 = r2, r4 ;; x += A[i]
6: add r1 = r1, 1 ;; increment counter
7: bne r1, 1024, loop ;; backwards branch
```

Assume you have a machine with infinite resources (i.e. infinite width, infinite instruction window, infinite number of functional units, infinite rename resources, infinite number of outstanding/overlapping loads, etc.), and that it is fully pipelined so that all normal instructions take 1 cycle. Loads which hit in the cache also take 1 cycle, but loads that miss take 10 cycles. Also assume perfect branch prediction.

(a) Show the dynamic schedule of the instructions of the first two iterations of the loop in the table below assuming that the load misses in the cache in even iterations (r1=0,2,. . .) and hits in the cache in odd iterations (r1=1,3,. . .). Be sure to indicate the loop iteration each instruction is associated with. For example, if instruction 5 of iteration 7 can run in cycle 3, then you would write “5.7” in one of the boxes in cycle 3’s row. Note that the first cycle has been filled in for you and that the cells in the table are more than you need.

Cycle	Instructions					
1	1	2				
2						
3						
4						
5						
6						
7						
8						
9						
10						
11						

12						
13						
14						
15						
16						
17						
18						

Cycle	Instructions					
1	1	2				
2	3.1					
3	4.1					
4	4.1					
5	4.1					
6	4.1					
7	4.1					
8	4.1					
9	4.1					
10	4.1					
11	4.1					
12	4.1					
13	5.1	6.1				
14	7.1					
15	3.2					
16	4.2					
17	5.2	6.2				
18	7.2					

19. [3 points] Prediction and Predication. “When a decision has to be made, make it.” –General George Patton

(a) Consider the following piece of code:

```
for(i=0; i<1000000; i++){
a = random(100);
if(a >= 50){
  ...
 }
}
```

Assume that random(N) returns a random number uniformly distributed between 0 and N-1 inclusive. Consider the branch instruction associated with the if statement. Is there a type of branch predictor that predicts this branch well? Explain your answer.

A predictor which constantly predicts Taken (or Not Taken) is likely to be right half of the times here. Since a is uniformly distributed between 0 and 99, the probability that it is greater than 50 is 0.5. So one would expect that in a large number of trials, about half the number of times the branch would be taken. Any other predictor is likely to be of no use since the possibility of whether a particular instance of branch is taken is independent of all other instances of the branch.

(b) Predication can eliminate all forward conditional branches in a program. The backward branches in a program are, typically, associated with loops and hence are mostly taken. So it is possible to eliminate all forward branches and statically predict the backward branches as Taken and this would get rid of the complex branch predictors in a machine. But the Itanium processor which has hardware support for predication still retains a complex two level branch predictor. Explain why this is the case.

Firstly eliminating all forward branches can result in a lot of dynamic instructions to be wastefully executed due to misspeculated execution of predicated instructions. Secondly in case of nested loops, having a two level branch predictor can help in accumulating histories specific to each branch and be able to learn each backward branch's periodicity accurately by not letting the patterns of the two branch instructions to interfere with each other.

20. [10 points] Compilers are useful (occasionally). A common loop in scientific programs is the so-called SAXPY loop:

```
for (i=0;i<N;i++)
{
Y[i] = A*X[i] + Y[i]
}
```

(a) For A=3 and N=100, convert the C code given above into MIPS assembly. You may assume that X and Y are 32-bit integer arrays. Try to avoid using pseudo-ops as they will hinder the next parts of this question. Also, avoid introducing false dependences. Please use $s0 to hold the value of i. You may also assume that $s2 and $s3 initially hold the address of the start of X and Y, respectively. Hint: Since A=3=2+1, you can easily perform the multiplication without using the mult instruction.

One possible MIPS assembly sequence:

```
       li $s0, 100
loop:
       lw $t0, 0($s2)
       add $t1, $t0, $t0
       add $t1, $t1, $t0
       lw $t2, 0($s3)
       add $t4, $t1, $t2
       sw $t4, 0($s3)
       addi $s2, $s2, 4
       addi $s3, $s3, 4
       subi $s0, $s0, 1
       bne $s0, $zero, loop
```

(b) How many static instructions (total instructions in memory) does the snippet have? How many dynamic instructions (total instructions executed) does the snippet have?

The number of static instructions is 11. The number of dynamic instructions is 1001.

(c) Suppose we wanted to do more than one SAXPY loop:

```
for (i=0;i<N;i++) {
Y[i] = A*X[i] + Y[i]
}
for (i=0;i<N;i++) {
Z[i] = A*X[i] + Z[i]
}
```

Using your computations from parts (a) and (b), how many dynamic and how many static instructions will this snippet have?

The number of static instructions will be 22. The number of dynamic instructions is 2002.

(d) There are two optimizations that can be performed to this code. The first is loop fusion, which combines two similar loops into one loop:

```
for (i=0;i<N;i++) {
Y[i] = A*X[i] + Y[i]
Z[i] = A*X[i] + Z[i]
}
```

Another optimization we can do is common sub-expression elimination. This means that we can factor out repeated computation:

```
for (i=0;i<N;i++) {
T = A*X[i];
Y[i] = T + Y[i]
Z[i] = T + Z[i]
}
```

Convert each of these snippets into MIPS assembly. $s4 will contain the address of the start of Z. Use $s1 to store the value of T. How many dynamic and how many static instructions does each snippet have? How much do you save with loop-fusion (over unoptimized code). How much do you save with constant sub-expression elimination (over loop fusion)?

Using loop-fusion:

```
        li $s0, 100
loop:
        lw $t0, 0($s2)
        add $t1, $t0, $t0
        add $t1, $t1, $t0
        lw $t2, 0($s3)
        add $t4, $t1, $t2
        sw $t4, 0($s3)

        lw $t0, 0($s2)
        add $t1, $t0, $t0
        add $t1, $t1, $t0
        lw $t2, 0($s4)
        add $t4, $t1, $t2
        sw $t4, 0($s4)

        addi $s2, $s2, 4
        addi $s3, $s3, 4
        addi $s4, $s4, 4
        subi $s0, $s0, 1
        bne $s0, $zero, loop
```

The number of static instructions is 18 (saved 4 instructions) and the number of dynamic instructions is 1701 (saved 301 instructions).

Using common sub-expression elimination:

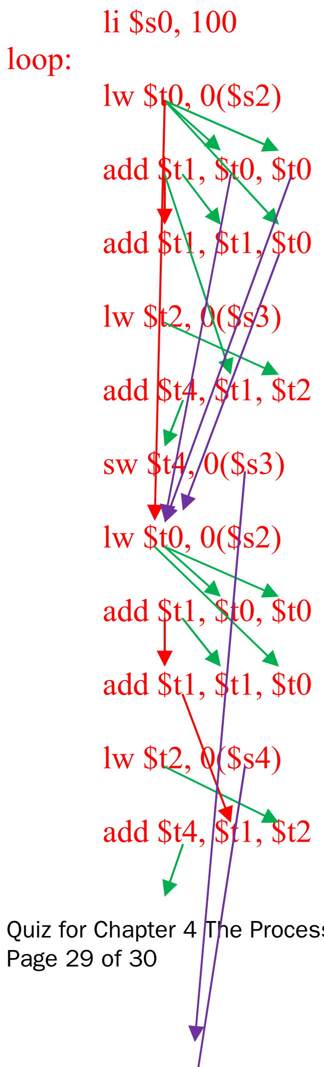
Name:____

```
      li $s0, 100
loop:
      lw $t0, 0($s2)
      add $t1, $t0, $t0
      add $t1, $t1, $t0

      lw $t2, 0($s3)
      add $t4, $t1, $t2
      sw $t4, 0($s3)

      lw $t2, 0($s4)
      add $t4, $t1, $t2
      sw $t4, 0($s4)

      addi $s2, $s2, 4
      addi $s3, $s3, 4
      addi $s4, $s4, 4
      subi $s0, $s0, 1
      bne $s0, $zero, loop
```

The number of static instructions is 15 (saved 7 instructions). The number of dynamic instructions is 1401 (saved 601 instructions)

(e) Annotate assembly code for the loop fused version from part (d) with data dependence arcs (flow, anti, and output - not control).

For loop-fused code:



```
      li $s0, 100
loop:
      lw $t0, 0($s2)

      add $t1, $t0, $t0

      add $t1, $t1, $t0

      lw $t2, 0($s3)

      add $t4, $t1, $t2

      sw $t4, 0($s3)

      lw $t0, 0($s2)

      add $t1, $t0, $t0

      add $t1, $t1, $t0

      lw $t2, 0($s4)

      add $t4, $t1, $t2
```

Name:____

```
sw $t4, 0($s4)

addi $s2, $s2, 4

addi $s3, $s3, 4

addi $s4, $s4, 4
subi $s0, $s0, 1

bne $s0, $zero, loop
```

Here the green arrows indicate flow dependency, red arrows indicate anti dependency and the purple arrows indicate output dependency