CMPE 110 Computer Architecture, Fall 2014
Homework #3

Computer Engineering
UC Santa Cruz

October 31, 2014

Name: _____

Email: _____

Submission Guidelines:

- This homework is due on Thursday 10/30/14.
- Bring the homework to class before 8pm
  - Anything later is a late submission
- **Please write your name and your UCSC email address**
- **The homework should be "readable" without too much effort**
  - If your handwriting is like mine, type it or risk not being graded
- Provide details on how to reach a solution. An answer without explanation has no credit. Clearly state all assumptions.
- Points: 40 = 10 + 14 + 16

Question	Part A	Part B	Part C	Part D	Total
1			-	-	
2					
3					
Total					

Question 1. Stack CPU (10 points)

In this problem we will study a processor that implements a Stack ISA similar to the one from Homework 1. A summary of Stack ISA instructions are given in the first two columns of the table below.

This Stack CPU is pipelined and uses the same stages as Tiny ISA (Fetch, Decode, Execute, Memory, and Writeback). The primary difference of the Stack CPU is that it uses a Stack instead of a Register File for local storage. Instead of reading from a Register File in the ID stage, the Stack CPU pops from the Stack in the ID stage. Additionally, instead of writing to a Register File in the WB stage, the Stack CPU pushes to the Stack in the WB stage.

Suppose that in this Stack CPU, only a single value may be popped from the Stack per cycle. Additionally, only a single value may be pushed to the Stack per cycle. This is analogous to a Register File with one read port and one write port.

Instruction	**Operation**	**Instruction Latency**
`add`	pop x; pop y; push $x + y$	6 cycles
`bgtz label`	pop x; if $x > 0$, jump to address specified by `label`	
`dup`	pop x; push x; push x	
`goto label`	jump to address specified by `label`	
`pushi imm`	push `imm`	
`pushm`	pop $addr$; push M[$addr$]	
`popm`	pop x; pop $addr$; M[$addr$] $\leftarrow x$	
`sub`	pop x; pop y; push $y - x$	
`swap`	pop x; pop y; push x; push y	

Question 1.A Instruction Latency (4 points)

Fill out the last column in the table above. Assume no stalling for hazards. The first one has been done for you and it accounts for the fact that the add instruction requires two pops from the stack in order to have its operands for the EX stage.

Question 1.B Pipeline Behavior (6 points)

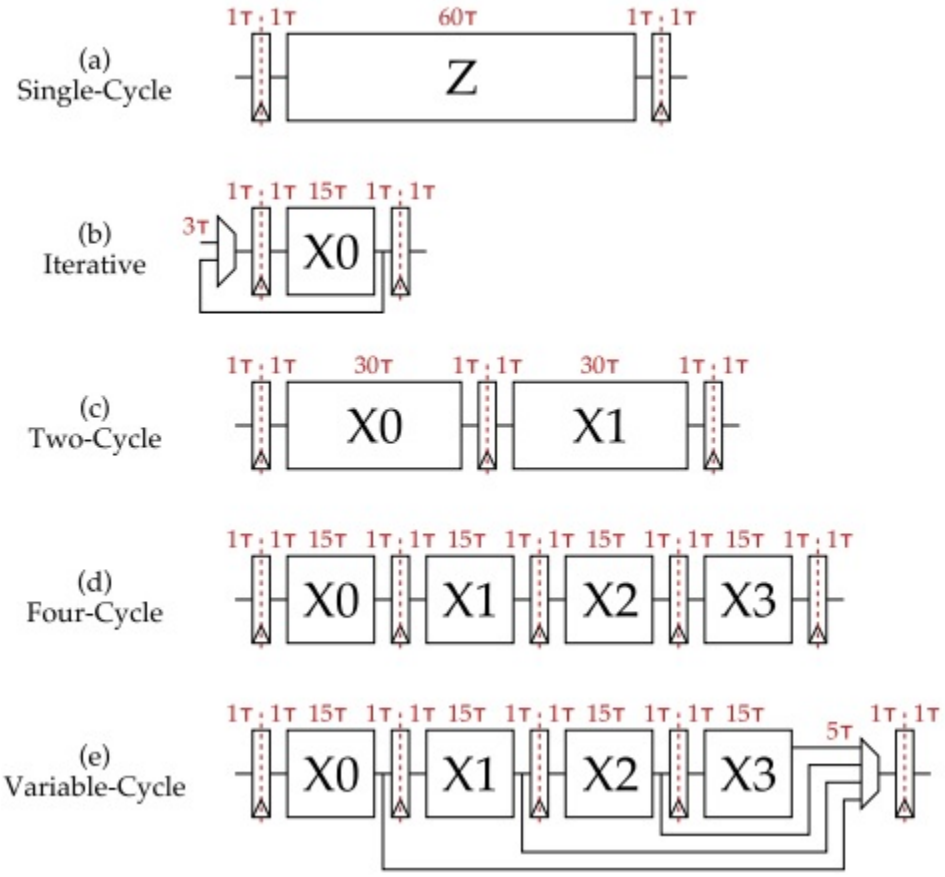
Fill out the following pipeline diagram for the following code segment of Stack ISA instructions. **Assume that the pipeline stalls to resolve structural hazards and uses bypassing to resolve data hazards.** Circle pipeline stages in which bypassing is necessary and draw arrows indicating data dependencies between stages.

Use the pipeline diagram to calculate the average CPI of this Stack CPU.

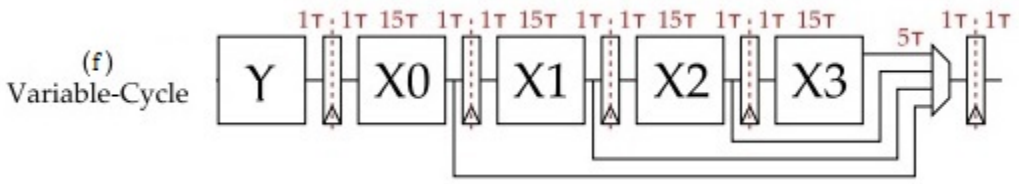
Instruction	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
`pushm`																
`pushi #1`																
`add`																
`swap`																
`dup`																
`popm`																

Question 2. Multiplier (14 points)

In the previous homework, we considered several different implementations of an unsigned two-input integer multiplier capable of multiplying a 32-bit operand by a 4-bit operand to produce a truncated 32-bit result. We will now look at a new design for a multiplier, shown in (e) in the figure below. The datapaths for the multipliers in the previous homework are shown in (a) through (d) for reference.



The datapath in (e) is for a variable-cycle pipelined multiplier. This microarchitecture exploits the fact that when some of the bits in the four-bit operand are zero, we dont actually have to do any work. We add a new stage at the beginning of the pipeline (denoted with the Y symbol) that is responsible for determining the bit position of the most significant one in the four-bit operand. This control information then goes down the pipeline and is used to write the result to the final output register as soon as we are sure the rest of the stages will do no work. Note that this requires an extra mux before the output register, and we will need to carefully handle the structural hazard caused by multiple stages writing the same register. **Assume that the multiplier stalls in the Y stage whenever it detects that letting the current transaction go down the pipeline would cause a structural hazard.**



Question 2.A Standalone Performance (4 points)

Like in the previous homework, for this part we will be running the following four multiplication instructions.

```
1  mul $r0, $r0, 0xf
2  mul $r1, $r1, 0x7
3  mul $r2, $r2, 0x3
4  mul $r3, $r3, 0x1
```

Fill in the instruction vs. time diagram below illustrating the execution of these four multiplication instructions on the pipelined variable-cycle microarchitecture. Use the symbols Y, X0, X1, X2, and X3 to indicate on which cycle each transaction is using that part of the multiplier. Look at the four-bit operand in each of the four multiplication transactions to determine how many stages of computation are actually required.

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
I0																
I1																
I2																
I3																

Use your diagram to fill in the table below, which is similar to the table found on the previous homework.

	Microarchitecture	Num Instr (#)	Cycle Time (τ)	Instruction Latency (cyc)	Instruction Latency (τ)	Throughput CPI	Total Execution Time (τ)
(e)	Var-Cycle Pipelined	4					

Question 2.B Variable Latency vs Fixed-Latency (4 points)

Discuss when we would want to choose this variable-latency pipelined multiplier over a fixed-latency design like from the previous homework, and vice-versa. What are the trade-offs?

Question 2.C Integrating with Processor (6 points)

Suppose that we like this design for the multiplier and decide to integrate it with our 5-stage pipelined processor. This block would exist alongside the ALU in the EX stage so when an instruction reaches the EX stage, it either goes through the ALU or through the multiplier. If it goes through the ALU, we denote this using EX stage as normal. If it goes through the multiplier, we denote this using the X0, X1, X2, and X3 stages (similar to part A). Note that we no longer need the Y stage because the Y stage is replaced by the ID stage in the processor. That means that the ID stage will stall if it detects that letting the current instruction go down the pipeline would cause a structural hazard.

Assume that the pipeline stalls to avoid structural hazards and uses bypassing to avoid data hazards.

The instructions we want to run through the processor are as follows (note that they are different from the instructions in part A):

```
1  mul $r0, $r3, 0xf
2  mul $r1, $r0, 0x7
3  mul $r2, $r1, 0x3
4  mul $r3, $r2, 0x1
```

Fill out the pipeline diagram on the next page, illustrating the execution of these four multiplication instructions on the updated pipelined processor. Use this diagram to calculate the average CPI of this newly integrated processor.

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17
I0																		
I1																		
I2																		
I3																		

Question 3. Resolving Data Hazards (16 points)

In this problem we look at a 5-stage pipelined processor that uses the tiny ISA. We will look at the following code segment that includes data dependencies.

```
1  add $r0, $r1, $r2
2  sub $r1, $r0, $r3
3  ld  $r4, [$r7]
4  st  [$r8], $r9
5  add $r5, $r4, $r0
6  bne $r4, $r5, #10
```

Question 3.A Stalling (1 point)

The pipeline diagram below shows these instructions going through the processor. In this baseline design, the processor stalls to resolve data hazards.

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14
I0	IF	ID	EX	ME	WB										
I1		IF	ID	ID	ID	EX	ME	WB							
I2			IF	IF	IF	ID	EX	ME	WB						
I3						IF	ID	EX	ME	WB					
I4							IF	ID	ID	EX	ME	WB			
I5								IF	IF	ID	ID	ID	EX	ME	WB

Use the above pipeline diagram to calculate the average CPI of the baseline processor.

Question 3.B Scheduling (6 points)

One way to deal with data hazards is to use *scheduling*. Scheduling is usually done on the software side by the compiler and involves re-ordering the instructions in the assembly. This is allowed as long as it does not change the overall functionality of the code. How would you re-order the code segment to improve the CPI of the processor? Write the newly re-scheduled code and fill in the pipeline diagram on the next page.

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17
I0																		
I1																		
I2																		
I3																		
I4																		
I5																		

Question 3.C Bypassing (6 points)

Another way to deal with data hazards is to use *bypassing*. Bypassing involves modifying the hardware to add muxes to allow for data to be forwarded between stages so that there is no need to wait for the WB stage of the previous instruction necessarily. In this case, other stages in the pipeline can read new values before they are written back.

Fill in the pipeline diagram on the next page showing how instructions go through the pipeline using bypassing. Circle stages and draw arrows on the diagram between them showing how data is forwarded. Use this diagram to calculate average CPI of the processor with bypassing.

Question 3.D Trade-Offs (3 points)

Compare the different ways to deal with data hazards studied in parts A, B, and C and discuss their trade-offs.

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17
I0																		
I1																		
I2																		
I3																		
I4																		
I5																		