CMPE 110 Homework #2

John Allard
February 2nd, 2015
jhallard@ucsc.edu

1. **Question #1 - Pipeline Stages**

   (a) **Question 1.A Baseline**
   Fill out the table below for the given pipelined processor. Cycle time is the clock period at which this machine can run. Max clock frequency is the macimum frequency that can be run given the cycle time (lower frequencies are possible). Latency of instruction is the time it takes to get the first output after it starts. Throughput is the rate at which output is produced.

   Explanation - The cycle time for the baseline architecture is 400ps, which is the time associated with the memory access stage. This is the longest stage length, if we shortened the cycle length below 400ps we wouldn't be able to reliably access memory under our current architecture. Max frequency is the reciprocal of the cycle time, and instruction latency is $8 *$ cycle time because there are 8 stages for an instruction to cycle through.

   (b) **Question 1.B Faster Memory**
   Suppose there is an optimization that reduces the memory stage latency by 100 ps. However, the optimization in a memory stage results in a 100ps increase in the write back stage latency. How much would this improve the overall performance of the 8-stage pipeline? Fillout the second row in the table below for this pipeline.

   Explanation - This new configuration would improve performance by 25%, or a 1.25x speed up. I got this number by dividing the latency of the baseline (3200 ps) by the latency f the improved pipeline (2560). The cycle time for this new configuration is 320ps, which is the time associated with the decoding stage. After reducing the memory access stage by 100ps to 300ps the new longest stage was decoding so the cycle time became associated with this stage. The max clock frequency is simply the reciprocal of the cycle time. The instruction latency is $8 *$ cycle time which is 2560ps (2.56ns).

   (c) **Question 1.C Proposed Scheme**
   If you had the option to rearrange the pipeline, how would you arrange it to achieve maximum frequency?

   Explanation - As given originally the total time to get through the pipeline by adding up the different times of each stage was 2240 ps. Dividing this by 8 (for 8 stages in the pipeline) gives us a round number, 280. So if we adjust the boundaries between the stages so as to give us a constant stage latency of 280ps, this would be optimal. Beyond this, decreasing the time for any one stage will necessarily increase the time it takes another stage to run. The max clock frequency is $\frac{1}{\text{Cycle Time}}$. The instructon latency in $8 *$ (cycle time), and since the cycle time is constant, this is just $8 * 280 = 2240$.

Table 1: Q1 Calculations

Processor	Cycle time	Max Clock Freq.	Instr. Latency	Throughput
a.) baseline	400	2.50 Ghz	3200 ps	1 per cycle
b.) faster mem	320	3.125 Ghz	2560 ps	1 per cycle
c.) new scheme	280	3.5714 Ghz	2240 ps	1 per cycle

2. **Question #2 - Extended Tiny ISA**
   Assume we are about to use an extended version of our Tiny ISA in an embedded system (say, a washing machine). Our implementation has 5 pipeline stages, and the ISA, as explained in lecture 2, does not have a multiply instruction. Analyzing the applications required to run on the processor, we learn that the applications needs to perform integer multiply about 1% of time. Now you as the

architect have to decide whether we should add a multiply instruction to the ISA and support it in the hardware, or just use the addition instruction to perform the multiplication through multiple additions.

(a) **Question 2.A** Compare options 1, 2, and 3 given above (in the handout, not rewritten here). Which option do you recommend? Show some sort of justification involving the performance of each option.c

Answer : I will start with a comparison of the three options. The table below summarizes the comparison. To begin, I define a baseline processor that has no multiply instruction or any software support for a integer multiply instruction. Such a processor has CPI = 1, frequency = f, latency = l, and instruction count = IC. All of the options will be compared by their performance relative to this baseline. The table shows the values for how the different options compare in different areas. The formula for performance is $\frac{1}{\text{Time(A, P)}}$ where A is a given machine and P is a given program. Time is calulcated by the following formula : Time = $IC * CPI * Period_{clk}$.

Option #1 is the worst of the 3 options when measuring performance. This method manages to keep the CPI the same as the baseline processor, which is good, but each of those cycles is taking twice as long to complete as the baseline. This effectively reduces the frequency by half, which in turn doubles the latency of the pipeline. The time for this option was calculated as : $1 * IC * 2P_{clk} = \frac{2*IC}{f}$.

Option #2 is the second best by my calculations. This method also manages to keep the CPI at 1, which is good. The frequency and hence the latency is unchanged, which also is a good thing. The increase in instruction count isn't very significant, but it's not enough to edge out option #2 over option #3. This makes sense to me intuitively, forcing integer multiplication to be implemented in the software shouldn't be as performance effective as an intelligent implementation in the hardware. It is interesting that this option is better than the first, it shows that some naive hardware implementations will be slower than basic software implementations. The time calculated for this option was $1.17IC * P_{clk} = \frac{1.17IC}{f}$.

Option #3 is the best option by my calculations. This approach does increase CPI by 20%, which is not insignificant. This unwanted increase in CPI is more than beaten out by the halving of the cycle time that increasing the number of stages from 5 to 8 affords us. This halving of the cycle time means the frequency doubles, so even if we are using 20% more cycles per instruction, each cycle is happening twice as fast and thus we end up better than where we started. Interestingly, the latency of the pipeline also decreases, but this time it decreases to 80% of the value that the baseline processor had. This is because although the frequency (and thus the delay at each stage) was halved, the number of stages was increased from 5 to 8. The time for this option was calculated as : $1.2 * IC * \frac{1P_{clk}}{2} = \frac{1.2IC}{2f} = \frac{0.6IC}{f}$

Since Performance = $\frac{1}{\text{Time}}$, and option 3 had the lowest time, option 3 will have the highest performance, and thus that is the option that I recommend.

Table 2: Q 2.A : Performance Calculations

Processor	CPI	Freq.	Latency	# Instructions
Baseline	1	f	l	IC
Option #1	1	$\frac{1}{2}f$	$2l$	IC
Option #2	1	f	l	$1.17IC$
Option #3	1.2	$2f$	$\frac{4}{5}l$	IC

(b) **Question 2.B** Lets assume we decided to go with option #3. The software team comes to you and propose a compiler optimization which helps limit the increase in CPI, so it only increases by 5% instead of 20%. However, the software team points out that this optimization could possibly increase the instruction count. What is the maximum increase in the instruction count that you can accept?

Answer : The maximum percentage increase for the inctruction count that would still be acceptable is 14.29%. At this percentage, all of the gains in performance that have been made by

reducing the CPI from 1.2 to 1.05 will be erased by the increase in instruction count. Any higher of a percentage and we actually lose performance, even with the decrease in CPI. I determined this by setting $\text{Time}_{old} = 1.2 * CPI * IC * P_{clk}$ and $\text{Time}_{new} = 1.05 * CPI * (x * IC) * P_{clk}$, where x is the factor by which the instruction count would increase. Setting these two equations equal to eachother and solving for x yield $x = 1.14285$, thus the maximum increase is roughly 14.3%.

3. **Question #3 : CPI Changes** The table below (in the handout) shows the distribution of each instruction type, on a processor running at a 1 GHz clock frequency.

   (a) **Question 3.A : Average CPI** What is the average CPI of the processor?
   Answer : The average CPI is simply the weighted average of the CPIs of the individual instruction, with the weights being the % of times that instruction shows up in a representative code sample. Taking a weighted average over the first two columns (the dot product of C_1 and C_2^T) gives a final CPI value of 2.08.

   (b) **Question 3.B : Optimization** With some optimization, and the support of some extra logic, the microarchitecture can decrease the load and store CPI to 2 and branch to 1 (option #1). Alternatively, the extra logic that can be spent can be used to optimize the mult/div logic to support CPI = 1 (option #2). Which optimization makes better sense to improve performance?

   Answer : Given the data in the table describing the two different optimizations, optimization #1 comes out as the clean winner, at least as far as CPI is concerned. This has to do with the principle of simplifying the operations which occur most frequently. Optimization #1 makes modest decreases in the CPI of very common operations, which ends up reducing its CPI from 2.08 to 1.53, a 26.5% decrease in CPI. For optimization #2, they got a very good optimization on a single instruction that rarely occurs, and thus there CPI only decreased from 2.08 to 1.9, a respectable but non-optimal 8.7% decrease in CPI. Given these facts, optimization #1 is the clear winner.

4. **Question #4 : CPI Computation for Different Pipelines**

   Assume the instruction breakdown as defined in the table (in the handout). Branch and Jump instructions are resolved in the end of Execute stage and there is no branch delay slot. Assume no data or structural hazards. Branches are taken 80% of the time. Answer each of the following questions. Each sub-question is independent.

   (a) **Question 4.A : CPI** Calculate the CPI of the system given in the handout. Assume no data dependencies.
   Answer : The CPI of the system is calculated almost as normal, where we take the weighted average of the individual CPIs for each instruction multiplied by their individual probabilities of occuring. The CPI for the instructions is normally 1, but the branch and jump instructions can introduce an increased CPI into the mix. The CPI is 5 (1+ 4 stalls) when a branch occurs because all of the instructions below it are already into their 4th stage by the end of the branch instructions execute cycle, at which time the branch instruction is resolved. This means that those 4 cycles are wasted, increasing the CPI to 5 for taken branch instructions. Jump instructions occur infrequenctly (only 5% of the time), but when they do occur the CPI is 4, because any instructions below are already in their 3rd stage by the time the jump instruction is resolved at the end of its execute stage. Thus the total CPI is

   $$.15 * .8 * 5 + .15 * .2 * 1 + .05 * 4 + 0.3 * 1 + 0.3 * 1 + 0.2 * 1 = 1.63$$

   The first two terms account for the CPI of the branch instruction (2 terms for when it taken 80% of the time and not taken 20% of the time). The 3rd term accounts for the jump instruction's CPI of 3, the next two terms account for load and store, and the final term accounts for all other instructions.

(b) **Question 4.B : Taken Branches** What is the CPI if all branches are assumed to be taken? Assume no data dependencies
Answer : If the branch is taken 100% of the time, then the CPI increases slightly. Adjusting the last equation slightly to account for this change yields ...

$$.15 * 1 * 5 + .05 * 4 + 0.3 * 1 + 0.3 * 1 + 0.2 * 1 = 1.75$$

This result is predictable, if we are always taking the branch then we are always taking an instruction with a higher CPI. This is because branch instruction has a CPI of 5 when a branch is taken and 1 when a branch is not taken.

(c) **Qestion 4.C : Branch Delay Slot** Assume the CPU has one branch delay slot and it is taken 50% of the time. What is the CPI, ignoring start-up?
Answer : In this case, when the branch is taken and the delay slot is filled, we save ourselves a wasted cycle and thus the CPI for that particular instance of the branch isntruction drops from 4 to 3 (because we save a single cycle). So now we not only have to account for how often branches are taken, but also how often the branch delay slot is used given that a branch is taken. This leaves us with the following calculation for overall CPI of the CPU.

$$.15 * (.8 * 0.5 * 4 + 0.8 * 0.5 * 5) + 0.15 * 0.2 * 1 + 0.05 * 4 + 0.3 + 0.3 + 0.2 = 1.57$$

The first large term account for the 15% of the time a branch instruction occurs, then the 80% of the time that that branch is taken, times the 50% probabilities that the delay slot will be used or unused. The second term account for the 20% of the time a branch instruction shows up but isn't taken, the rest of the terms are the same as in the above problems. This leaves us with the lowest CPI yet of 1.57, thanks to the branch delay slot reducing the number of hazards that occur.

(d) **Question 4.D : CPI** Assume we have data hazards. Fifty percent of the instructions following Load and Other use the result from that previous instruction. Example: the following sequence has an ADD instruction following a LW that uses the LW result. This occurs in 50% of the instructions. Notice that the instruction following the ADD also has a 50% chance of using the result of the ADD. **Answer :**

(e) **Question 4.E : Data Hazards II** Consider that 25% of the time the second instruction after the Load and Other instruction can use the data from the instruction. Example: the following sequence has an ADD instruction following the LW with no data hazard (50% chance from the previous part), but the following SUB instruction has a 25% chance of using the data generated by the LW. This 25% is independent of the 50% chance from the first dependenceCalculate the CPI of the system given in the handout. Assume no data dependencies.

```
lw r1, 0(r2)
add r5, r2, r3
sub r4, r1, r3
```

Answer :

5. **Question #5 : Multipliers** In this problem, we consider several different implementations of an unsigned two-inpuit integer multiplier capable of multiplying a 32-bit operand by a 4-bit operand to produce atruncated 32-bit result. Any of these implementations may be used to allow for the Tiny ISA to support the multiply instruction mentioned in the previous problem The diagram below illustrates the datapaths for five multiplier microarchitectures: (a) a single-cycle microarchitecture, (b) a four-cycle iterative microarchitecture, (c) a two-cycle pipelined microarchitecture, (d) a four-cycle pipelined microarchitecture, and (e) a variablecycle pipelined microarchitecture. For all parts in this problem we will assume that our multiplier is processing the following sequence of four independent instructions:

```
mul r0, r0, 0xf
mul r1, r1, 0x7
mul r2, r2, 0x3
mul r3, r3, 0x1
```

In each part, we will study one of these microarchitectures, and our goal is to gradually fill in the table below. The table already includes the results for the single-cycle multiplier microarchitecture.

Table 3: Q 5 : Performance Calculations

	Microarchitecture	Num Instr. #	Cycle Time τ	B2B Instr. Latency *cyc* τ	Throughput CPI	Total Exec. Time τ
a	1-Cycle	4	63	1 63	1	252
b	Iterative	4	22	4 88	4	352
c	2-Cycle	4	33	2 66	1	165
d	4-Cycle	4	18	4 72	1	126
e	Var-Cycled	4	24	5 120	1.75	184

(a) **Question 5.A : Iterative Microarchitecure**
Consider the iterative multiplier microarchitecture shown in (b). This microarchitecture computes a multiplication iteratively, across four cycles. It does this by sending back intermediatevalues building up to the result which is ready by the fourth cycle. In other words, we can complete each instruction in exactly four cycles, and we are ready to start the next instruction after exactly four cycles. Fill in the instruction vs. time diagram on the next page, illustrating the execution of the four multiplication instructions on this microarchitecture. Use the symbol X0 to indicate on which cycle each instruction is using the iterative multiplier. Use your instruction vs. time diagram to fill in the row in the first table for the iterative multiplier.

Table 4: Q 5.A Iterative Pipeline Instructions vs Cycles

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
I0	X0	X0	X0	X0												
I1					X0	X0	X0	X0								
I2									X0	X0	X0	X0				
I3													X0	X0	X0	X0

Q5.A Explanation - This is a very inefficient pipeline. Each multiply instruction takes 4 cycles, where each cycle feeds back a result to the next cycle, and the last cycle resutls in the final designated multiplication value. Each of the 4 cycles for a single multply take 21τ seconds, 4 to progogate to the first register, and 17τ seconds to progate through the multiply black-box and back to the first control unit. This means to complete the 4 cycles, it takes 84τ seconds (4 times the cycle time). For each of the 4 multiplications to take place, it the requires $4 * 84\tau = 332\tau$ seconds to complete the given sequence of instructions.

(b) **Question 5.B : 2-Cycle Pipelined Architecture**
Consider the two-cycle pipelined multiplier microarchitecture shown in diagram (c). In this microarchitecture, we use a similar approach as in (a) but we break it up the operation intom two cycles. In this microarchitecture, two different instructions can use the multiplier at the same time (i.e., one in the X0 stage and one in the X1 stage) .Fill in the instruction vs. time diagram below, illustrating the execution of the four multiplication instructions on this microarchitecture. Use the symbols X0 and X1 to indicate on which cycle each instruction is using that part of the multiplier. Use your instruction vs. time diagram to fill in the row in the first table for the 2-Cycle pipelined multiplier.

Table 5: Q 5.B 2-Cycle Pipeline Instructions vs Cycles

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
I0	X0	X1														
I1		X0	X1													
I2			X0	X1												
I3				X0	X1											

Q5.B Explanation - This architecutre is much more efficient than the last one. Each multiplication instuction is broken up into two relatively long stages (33τ), but two stages for subsequent multiplication instructions can overlap, which allows us to achieve an ideal CPI of 1. The 33τ cycle time is calculated as 33τ is the latency of the longest stage in the pipeline. The latency for each instruction is 66τ, since each instruction takes two stages of 33τ each. Each multiply instruction takes 66τ seconds to complete, but because of the pipelined structure where subsequent instructions overlap we see a significant total time reduction over the naive $4 * 66\tau = 264\tau$ runtime if the instructions couldn't overlap at all. The total execution time for this pipeline is $5 * 33\tau = 165\tau$. This includes the startup time, which is why it takes 5 total cycles to finish.

(c) **Question 5.C : 4-Cycle Pipelined Architecture** - Consider the four-cycle pipelined multiplier microarchitecture shown in diagram (d). In this microarchitecture, we use the same basic approach as the two-cycle multiplier, but we go even further and break up the operation into four stages, where there can be four different instructions using the multiplier at the same time (i.e., different instructions in X0, X1, X2, and X3). Fill in the instruction vs. time diagram below, illustrating the execution of the four multiplication instructions on this microarchitecture. Use the symbols X0, X1, X2, and X3 to indicate on which cycle each instruction is using that part of the multiplier. Use your instruction vs. time diagram to fill in the row in the top table for the 4-Cycle pipelined multiplier.

Table 6: Q 5.C 4-Cycle Pipeline Instructions vs Cycles

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
I0	X0	X1	X2	X3												
I1		X0	X1	X2	X3											
I2			X0	X1	X2	X3										
I3				X0	X1	X2	X3									

(d) **Question 5.C Explanation** - This pipeline is like the 2 stage pipeline, except it has 4 stages each of shorter length then the stages from the previous problem. Because each multiply instruction takes 4 cycles to complete, the B2B latency is 4 cycles. Since each cycle is 18τ time units, the latency is $4*18\tau = 72\tau$. This latency is slightly longer than for some of the previous examples, but the total execution time is shorter than for pipelines (a), (b), and (c). The total number of cycles it takes to complete all 4 multiplication instructions is 7, including the start-up time. Because each cycle takes 18τ units of time, the total execution time is 126τ units of time, a significant improvement over the predecessors. The CPI for ths pipeline is still 1, because there are no branches, jumps, or data hazards (or even data dependencies) to be found in this pipeline over the four given instructions. This pipeline is a good demonstration of the performance increases that can be afforded by having a deeper pipeline. Although the latency of a single instruction is longer than for a more shallow pipeline, the ability of this pipeline to work on more instructions at once end up affording it a performance edge over more shallow pipelines.

(e) **Question 5.D : Variable-Cycle Pipelined Architecture**
Consider the variable-cycle pipelined multiplier shown in (e). 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.

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.

Table 7: Q 5.C 4-Cycle Pipeline Instructions vs Cycles

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
I0	Y	X0	X1	X2	$X3_M$											
I1		Y	Y	X0	X1	$X2_M$										
I2			Y	Y	Y	X0	$X1_M$									
I3				Y	Y	Y	Y	$X0_M$								

Question 5.D Explanation - This was an interesting problem to solve, the pipeline is set up so that it doesn't waste time on multiplication with bits that are zero, it has a mechanism set-up to forward data down the data-path when it detects that there is no more significant computation (aka on 1 bits) to perform. This can be very useful, for instance if it multiplies something by 1 it can compute this in 2 cycles (one for Y, one for X0) with some MUX overhead added on. This is much better than taking 4 cycles no matter what the multiplicaton operands are. However, this forwarding scheme introduces possible structural hazards into the mix. These hazards arise when two multiplication results are trying to be sent to the final MUX at the same time. Since we can only return one value at a time, this introduces a hazard. To prevent this, the Y stage needs to analyze the constant input operand and figure out how many stages it takes to complete, and thus whether or not it will have create a hazard with any of the previous instructions. What makes this problem so interesting is that the combination of instructions virtually nullifies the speed increase gained by the forwarding mechanism. Every instruction is succeeded by one that takes one fewer stage, so each subsequent instruction introduces another structural hazard. This means that even though the 4th instruction is multiplication by 1, which could only take 2 cycles, ends up delayed and taking 5 cycles. The CPI for this pipeline over these specific instructions is $\frac{7}{4} = 1.75$. I got this by looking at the above table, and dividing the number of cycles it takes for a single from the first time a stage is accessed till the last time it is accessed, divided by the number of instructions. This CPI is higher than one because of the multiple delays introduced into the system by the forwarding mechanism. The cycle time is the time of the longest stage plus the time it would take to forward the data through the mux and to the output register. This means the cycle time is $18\tau + 6\tau = 24\tau$, which the 18 being from the multiplication stage and the extra 6 occuring if the data needs to forwarded to the final mux and output register. Since the cycle time is 24τ and there are 8 cycles, the total running time for this pipeline is 184τ units of time. This time is slower than the last 4 stage pipeline, once again because of the data hazards. The instruction latency depends on whether a data hazard occurs or not, since hazards occur diring all of the instructions, all of the instructions end up taking 5 cycles to complete.

(f) **Question 5.E : Composition of Designs**

Which microarchitecture has the highest performance? Explain some of the trade-offs between-these microarchitectures. Would we ever want to consider a multiplier with many more stages (e.g., a 20-cycle pipelined multiplier)? Discuss when we would want to choose this variable-latency pipelined multiplier over a fixed-latency design and vice-versa. What are the trade-offs between fixed-latency and variable-latency?

Answer : - Over the specific instructions given to us for this problem, the 4-Cycle pipeline finished the fastest. This outcome is very instruction dependent though, if the instructions

weren't ordered in such a way as to always cause a structural hazard with the next instruction, then the variable-latency pipeline would have done the best. This is because the variable-latency pipeline has the ability to perform a multiplication in a single, short cycle if the conditons are right (no hazards). As far as the fixed latency pipelines go, the 4-cycle one is the best when there are no data dependencies, this is because it maintains a throughput of 1 while reducing the cycle time significantly over both 2-Cycle and iterative pipelines. If we did have 4 consectuvie multiply instructions that were data dependent on eachother, then there is a good chance that the 2-cycle pipeline would do the best, simply becaause it isn't as deep and thus wouldn't have to perform as many stalls as the 4-cycle to resolve data dependencies. The iterative approach is interesting, there is no chance for a data or structural hazard, this is because there is no parallelism and thus by the time one instruction is ready to be executed all previous instructions are already done writing and reading their data. The iterative approach, while avoiding hazards, is still a very bad pipeline design. The CPI is always 4, and there is no way to increase it because of the pipelines fixed style.

So, to summarize, for these specific 4 instructions the 4-Cycle pipeline was the best at completing the task the fastest, but for the majority of the cases (where they aren't ordered in such a way as to always cause structural hazards) the variable cycle pipeline would have the highest performance.

(g) **Question 5.F : Integrating with Processor**

Suppose that we like the variable-latency multiplier design for the multiplier and decide to integrate it with our 5-stage pipelined processor. This block would exist alongside the ALU in the Execute stage so when an instruction reaches the Execute stage, it either goes through the ALU or through the multiplier. If it goes through the ALU, we denote this using Execute 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 Decode stage in the processor. That means that the Decode 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):

```
mul r0, r3, 0xf
mul r1, r0, 0x7
mul r2, r1, 0x3
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.

Table 8: Q 5.C 4-Cycle Pipeline Instructions vs Cycles

	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
I0	F	D	X0	X1	X2	X3	M	WB								
I1		F	D	D	D	D	X0	X1	X2	M	WB					
I2			F	F	F	F	D	D	D	X0	X1	M	WB			
I3							F	F	F	D	D	X0	M	WB		

Question 5.F Explanation - This example is even less efficient than the last one, this is because now we not only have structural hazards but now we've also introduced multipled data hazards.

There are data hazards whenever an instruction tried to operate on data that is being written to by a previous instruction, which occurs in every one of the 3 instructions after the first 1. These extra hazards increase the CPI from $\frac{7}{4} = 1.75$ in example 5.D to a CPI of $\frac{9}{4} = 2.25$ in this example. I got this CPI value by counting how many cycles it takes from the first time a fetch instruction is called till the last time the fetch stage is used, which is 9 cycles. Since there are 4 instructions, we get a CPI of $\frac{9}{4}$. The data forwarding happens between the first two rows from the X3 stage of instruction #1 to the X0 stage of instruction #2. This is because instr. #2 depends on the result of the multiply in instr. #1 to perform its calculation. The value from instr. #2 gets forwarded to the 3rd instruction after the X2 stage, since the second instruction does not need to use X3. The structural hazards arise because each instruction is trying to use less stages then the previous, which causes stalls to accumulate.

6. **Question #6 : Hazards**
Consider the following MIPS assembly code where each instruction is 32-bit (4 bytes). We want to run this code on a variable length pipeline. The pipeline is a typical 5-stage, except that the Multiply instruction (mul), which uses dedicated hardware that has 3 stages. Hence, mul spends 3 cycles in the Execute stage. The figure below shows the pipeline diagram. At each cycle, 1 instruction is allowed into the Execute stage if the requested unit is not occupied. Assume that branch and jump instructions are resolved in the Decode stage.

```
1          xor r3, r3, r3
2          addiu r2, r3, 1
3          j L1
4   loop   mul r3, r3, r3
5          add r4, r1, r4
6          lw r5 0(r4)
7          sub r5, r5, 1
8          sw r5, 0(r6)
9          addiu r3, r3, 1
10  L1     bne r2, r3, -28
11  END
```

(a) **Question 6.A : Data Dependencies** - Identify all data dependencies (Read After Write)

Answer : - Below is a table that completely describes all RAW depencies found in this program. The first column shows the instruction that is dependent on the instruction in the second column. The 3rd column gives a short explanation of why (not sure if necessary, but can't hurt).

Table 9: Q 6.A Data Dependncy List and Descriptions

This depends	on this	reasons :
line 2	line 1	input operand r3 is being written to by instr #1
line 10	line 2	comparison operand r2 is being written to by instr #2
line 6	line 5	address in r4 is being written to by prev. instruction
line 7	line 6	perform arithmetic on a value (r5) thats still being written to
line 8	line 7	we need to save the r5 value to memory but it's still being written by line 7
line 10	line 9	performing comparison on value being written to after arithmetic operation.

(b) **Question 6.B : Stalling to Remove Hazards**

Populate the pipeline diagram on the next page to show the first 20 cycles of the execution. Assume the pipeline does NOT support data forwarding (bypassing). Assume the pipeline keeps fetching next instructions after jump and branch before they are resolved. The pipeline has to stall in case of a hazard, which is detected in decode stage, and does not have a branch delay slot.

Table 10: Question 6.B Answers

Instruction Fetched	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16	C17	C18	C19
`xor r3, r3, r3`	F	D	EX	M	WB															
`addiu r2, r3, 1`		F	D	D	D	EX	M	WB												
`j L1`			F	F	F	D	EX	M	WB											
`mul r3, r3, r3`						F	x	x	x	x	x	x								
`bne r2, r3, -28`							F	D	EX	M	WB									
`mul r3, r3, r3`								F	D	X1	X2	X3	M	WB						
`add r4, r1, r4`									F	D	D	D	EX	M	WB					
`lw r5, 0(r4)`										F	F	F	D	D	D	EX	M	WB		
`sub r5, r5, 1`													F	F	F	D	D	D	EX	M

Question 6.B Explanation - This questions shows how innefficient a pipeline can be without data forwarding, where it is forced to introduce stalls for every type of hazard that occurs. To fill out the first 20 cycles, we only get through 9 instructions, which is pretty poor compared to more advanced architectures. The CPI of this system is $\frac{15}{9} = 1.6667$. This will be improved significantly in the upcoming problems by using data forwarding. One thing I was confused about was what is loaded into memory after the `bne` instruction is fetched and before it is decoded, because the next instruction is a PSUEDO-OP indicating that it's the end of the code. I made the assumption that `bne` would simply load the branch location instruction right after being fetched.

(c) **Question 6.C : Bypassing to Resolve Hazards -**

(d) **Question 6.D : Branch Delay Slot -**

(e) **Question 6.E : Structural Hazards -**