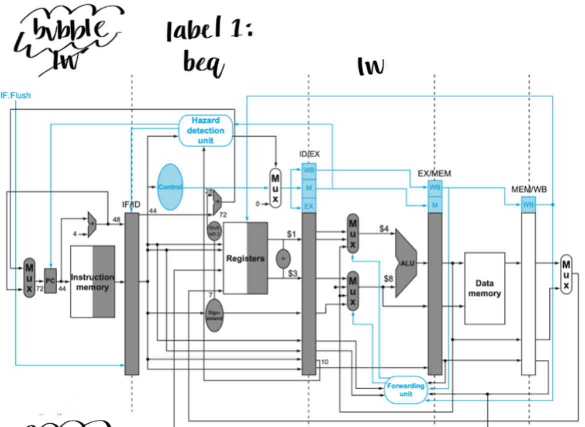
CprE 381 Homework 8

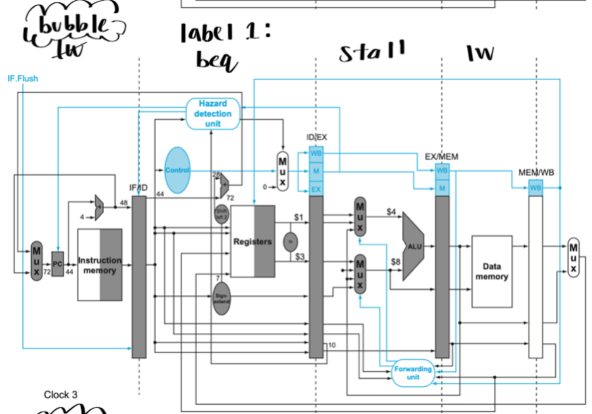
1. Exam 2 Question 7 Rework
    a. For our single-cycle design (shown below), what is the minimum CPI?
    Answer: 1
    b. What is its maximum CPI?
    Answer: 1
    c. For our basic pipeline design (shown below), what is the minimum ideal CPI (i.e., steady-state, no fill/hazards)?
    Answer: 1 (best case scenario would give us a minimum idea CPI of 1 for pipelining if there are no hazards)
    d. Now, assume the pipeline has hazard detection with stalling for data hazards (do not consider control hazards), but an unmodified register file. What is the maximum steady-state CPI (i.e., no fill)?
    Answer: 4
    e. For the 7-stage pipeline design shown below, what is the minimum ideal CPI (i.e., steady-state, no fill/hazards)?
    Answer: 1

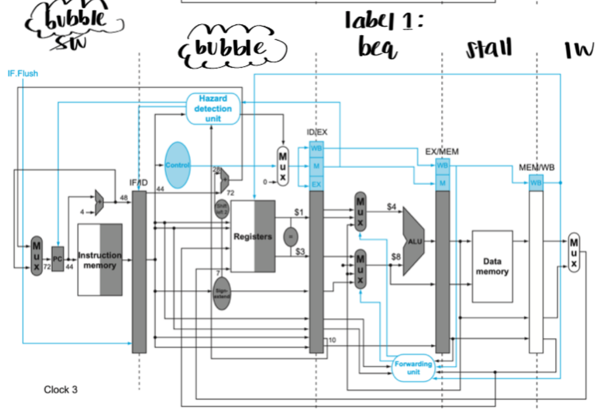
I had the correct understanding in the first two parts of the problem.
However, for part 3, since we started talking about the minimum ideal CPI for a pipeline with no fill/hazards, I thought that would be 1/5 because our pipeline had five stages. I confused the ideal cycle period with the CPI (which Professor Duwe talked about in class).
For part 4, I selected 1/3 because in part 3 I had selected 1/5, so I figured since part 4 has hazard detection with stalling for data hazards, the maximum CPI would only be slightly higher than the previous part. Turns out the previous part was wrong, so my answer for this part sort of followed suit.
For the last part, I put 1/6 as the minimum ideal CPI instead of 1. Also, judging from my previous parts, the error is carried forward. Since in Part 4 I used 1/5, I figured the minimum possible CPI for a 7-stage pipeline would be 1/6, but since parts 3 and 4 were wrong, Part 5 followed the same pattern.

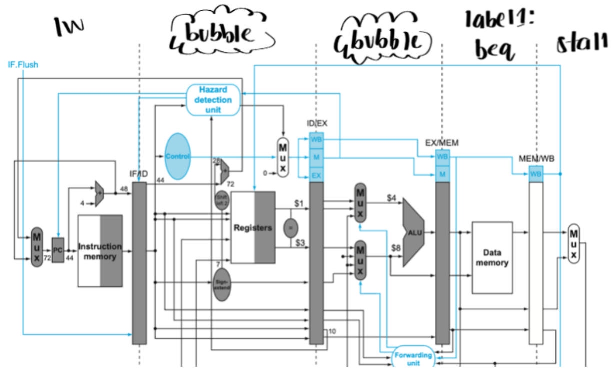
2.

a.

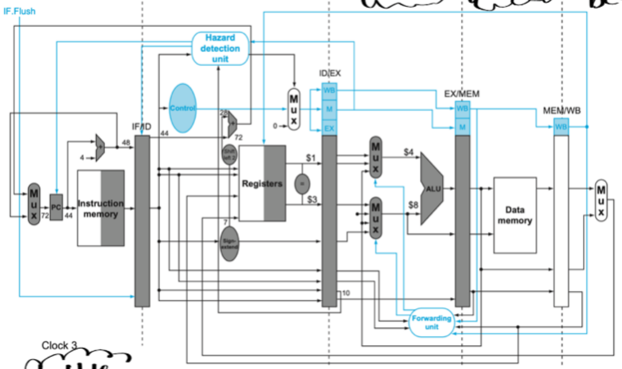

bubble
~~lw~~
label 1:
beq
lw
IF.Flush
Hazard detection unit
Control
IF/ID
ID/EX
EX/MEM
MEM/WB
Registers
Instruction memory
Data memory
ALU
Forwarding unit


bubble
~~lw~~
label 1:
beq
stall
lw
Clock 3

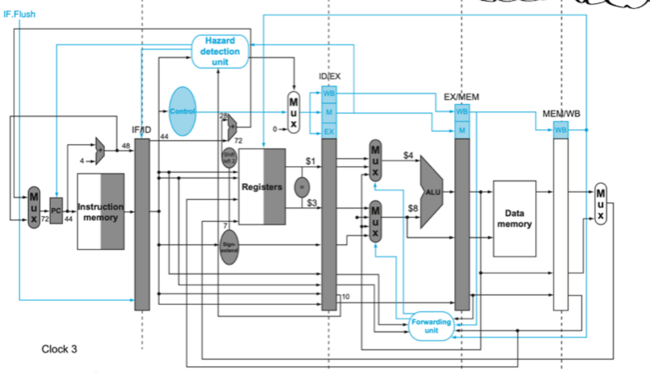

bubble
~~sw~~
bubble
label 1:
beq
stall
lw
Clock 3


lw
bubble
bubble
label1:
beq
stall

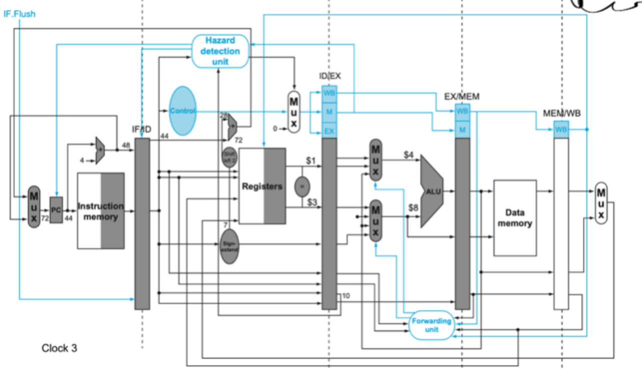
beq $3 lw bubble bubble label 1: beq



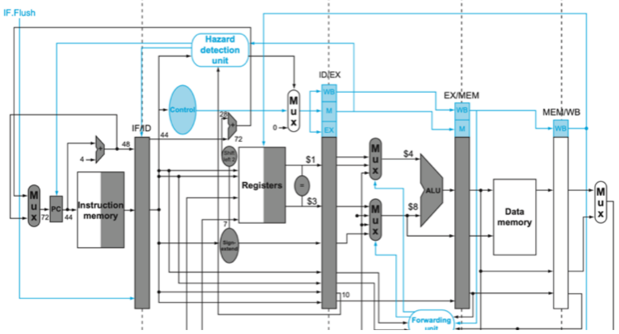
bubble add beq $3 lw bubble bubble

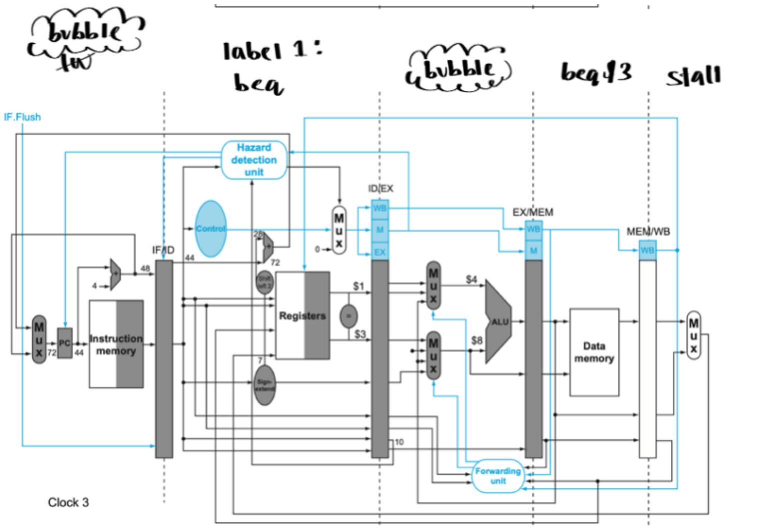


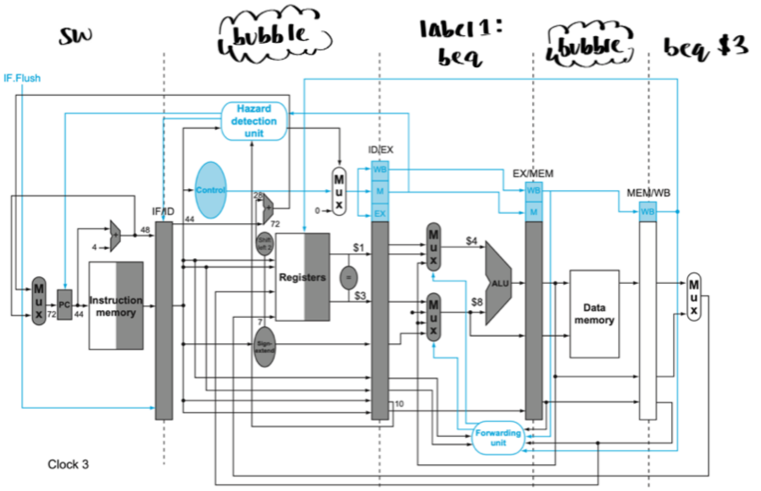
bubble add beq $3 stall lw bubble

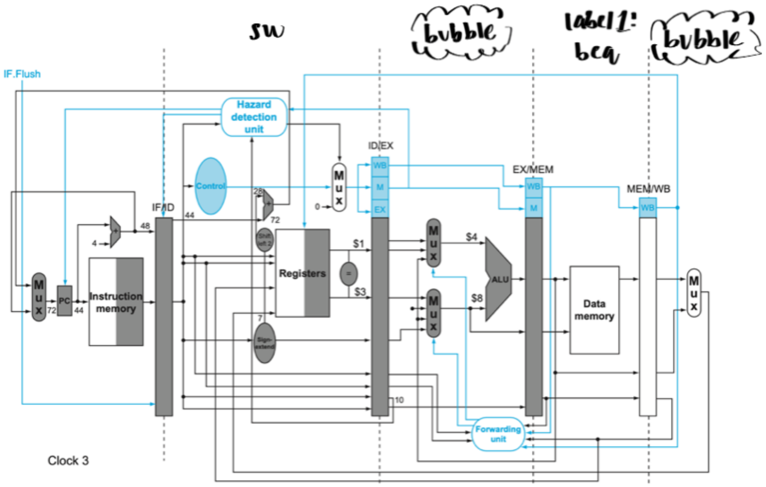


label 1: beq bubble beq $3 stall lw









and then sw propogates all the way.

b.

(S is for Stall)

lw $2, 0($1)	IF	ID	Ex	MEM	WB									
beq $2, $0, label2		IF	ID	S	Ex	MEM	WB							
lw $3, 0($2)			IF	ID	S	Ex	MEM	WB						
sw $1, 0($2) [bubble]														
beq $3, $0, label 1						IF	ID	Ex	MEM	WB				
add $1, $3, $1							IF	ID	Ex	MEM	WB			
beq $2, $0, label2								IF	ID	Ex	MEM	WB		
lw $3, 0($2)									IF	ID	Ex	MEM	WB	
sw $1, 0($2)										IF	ID	Ex	MEM	WB

c. Hi

d. Logic needed:
if (iD/EX.MemRead && ((ID/EX.RegisterRt == IF/ID.RegisterRs) || (ID/EX.RegisterRt == IF/ID.RegisterRs)))
This indicates there would be two stall cycles needed, and the hazard that this new logic is supposed to detect data hazards.

e.

By migrating the branch resolution from the EX stage to the ID stage of the pipeline, the processor can resolve and hence recognize all control hazards earlier in the pipeline, reducing stall cycles for branch instructions. In a traditional 5-stage MIPS pipeline, **a branch instruction resolved in the EX stage incurs a two-cycle penalty** because the processor still fetches instructions even though it is likely not on the proper path. **If the branch is resolved in the ID stage, the decision is made one cycle earlier**; hence, the cycle penalty is only one cycle. Using the instruction stream presented in the previous example, the overall cycles dropped from 10 to 9 when one branch instruction (beq $2, $0, label2) is encountered when resolved in the ID stage. Assuming the added comparison logic in the ID stage did not increase the clock cycle, this is an approximate overall speedup of 1.11x. While the gain is small for this short code segment, the performance impact for programs with repeatedly branching code could be more substantial, making early branch resolution a very important optimization.

f.

To enable support for branch execution in the ID stage and illustrate the first branch instruction (**beq $2, $0, label2**), we need to add forwarding logic that ensures that the source registers involved in comparing $2 against $0 contain the latest value. For the standard MIPS pipeline, forwarding paths are defaulted to being available only from the EX stage. In this case, the branch evaluation will occur in the ID stage; therefore, the register values needed for the comparison may still not be available. In this case, **if the instruction immediately before the branch evaluates one of the registers involved, it would only be staged in the EX stage**. This means we need to insert a new forwarding path to forward values from the EX/MEM or MEM/WB pipeline registers to the ID stage comparator. This will change the data path design and have control logic in place that can recognize and forward values back through the ID stage to the comparator for branch comparisons.

Overall, while the existing forwarding logic only required extending outside the normal following pipeline stage for use in the EX stage (see COD Figure 4.62), **adding this branch support adds more complexity** because forwarding the data path is now earlier in the pipeline and there may be more checks for hazards, as well as more mux configurations required. While this enhances performance by allowing earlier branch resolution, it does so at the cost of increased hardware complexity.

3. The article that I selected is titled “Downfall bug affects years of Intel CPUs, could leak encryption keys and more” from Ars Technica, which was published on August 9th 2023. The author presents the “Downfall" vulnerability, which is a speculative execution vulnerability in Intel CPUs. In short, the Downfall vulnerability takes advantage of speculative execution, which is a method of optimization during performance improvements, to leak confidential data, such as keys for encryption. Mitigations include software patches and microcode updates, which reduce performance by 30 to 50% for certain workloads. In addition to a performance drop, the mitigations increased power consumption, as CPUs will be executing extra unnecessary instructions redundantly to close the different security loopholes. The researchers mention the need to continue to keep both security and efficiency in mind while considering modern computer architecture.

   This article is specific to speculative execution, which is a pipeline optimization where CPUs predictively execute instructions to minimize idle time. The problem with the Downfall vulnerability occurs when each aspect of the predictions could potentially leak information unknowingly. The mitigations ultimately increase CPI (cycles per instruction), as there would be extra checks added to the processor, which would increase the average # of cycles that would be needed per instruction. Extra CPI means a longer execution time, which would ultimately slow things down in something like data encryption. It also increases the power draw, as the programmers had to add more operations per task, which is extra energy being consumed.

   Sources used:
   https://arstechnica.com/information-technology/2023/08/data-leaking-downfall-bug-affects-six-generations-of-intel-pc-and-server-cpus/