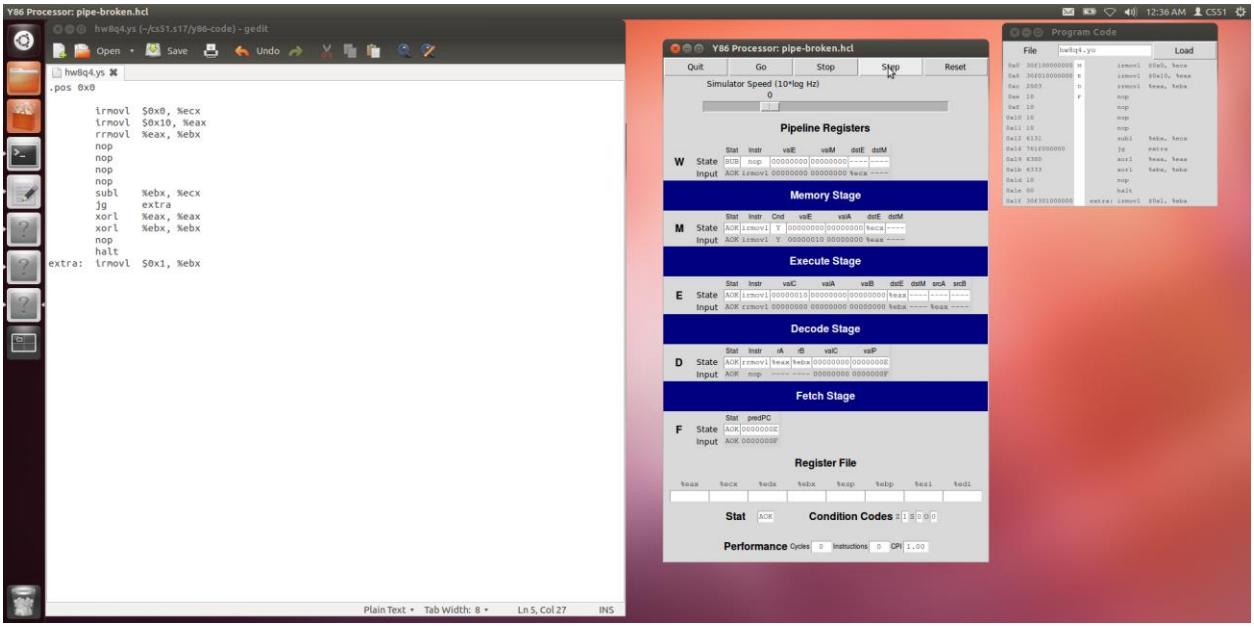
Graham Hazlett

HW8q4:

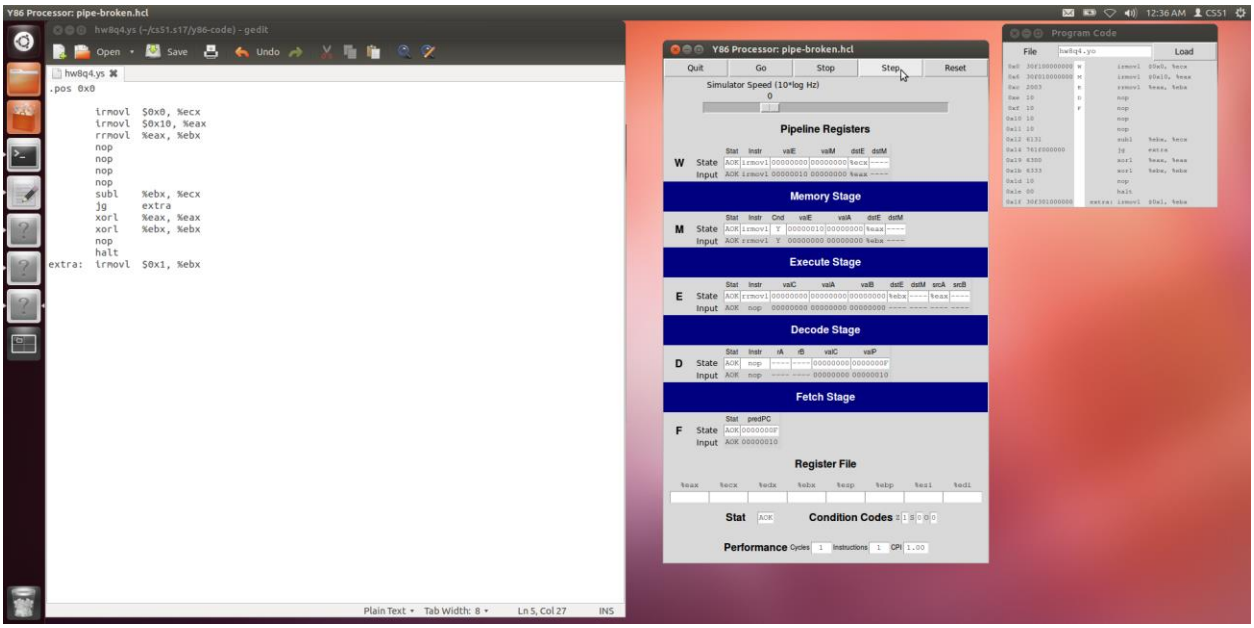
This code demonstrates a naïvely pipelined y86 executing a data and control hazard.

First, the code executes a data hazard, when 0x10 is written to %eax, but the machine tries to execute rrmovl %eax, %ebx before 0x10 is written to %eax, causing the wrong value to be written to ebx.

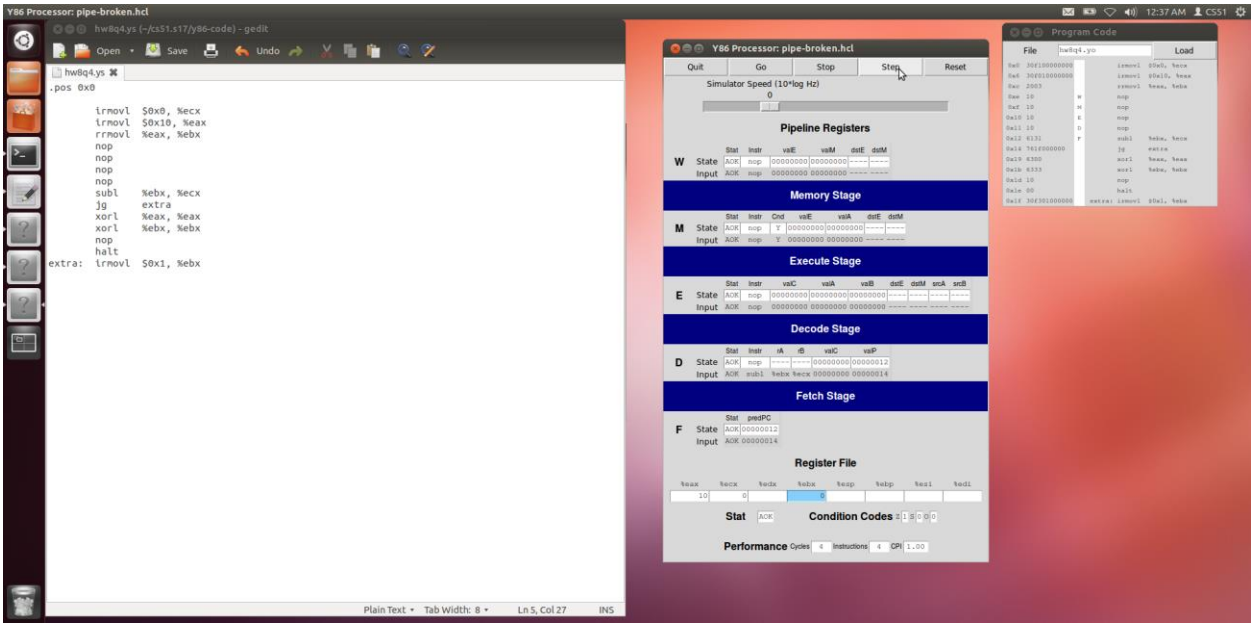
First, it can be seen that as the rrmovl instruction is decoded, the irmovl just above has not yet entered the write phase, and 0 is still in %eax:



This means that during the execute phase, it is soon ready to move 0x0 to dstE = %ebx:



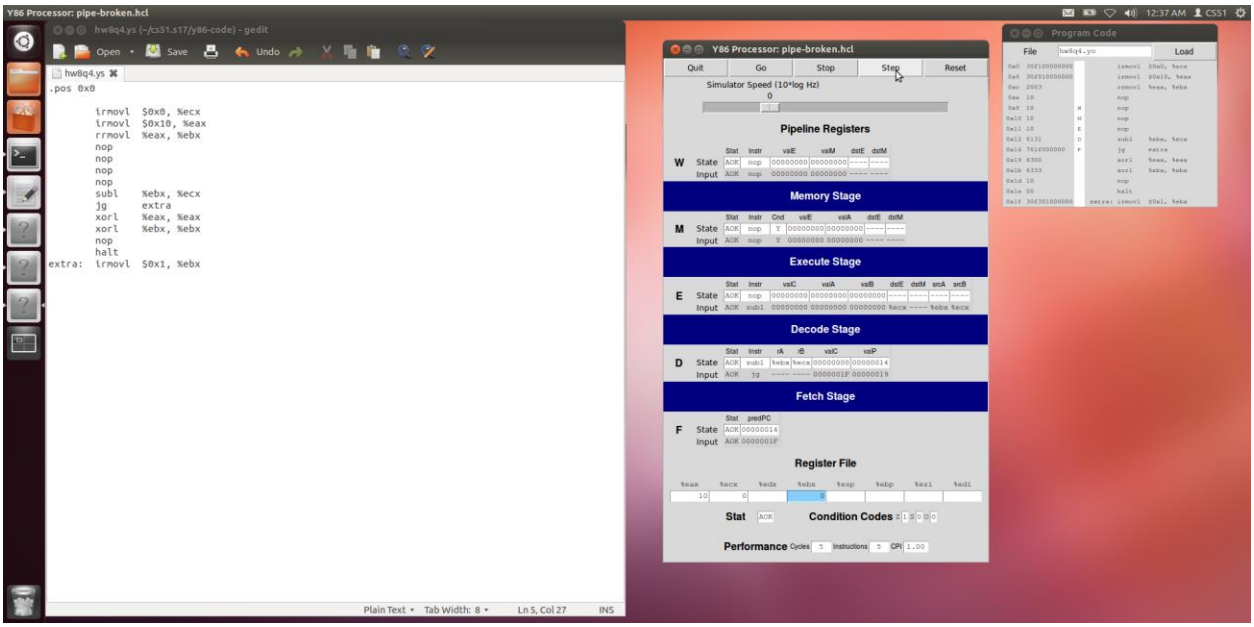
And as proven later, after the write back phase is executed, although a 0x10 was written to %eax eventually, it was not picked up in the pipelining quick enough for the rrmovl, and by now the code has chosen to write 0x0 to %ebx:



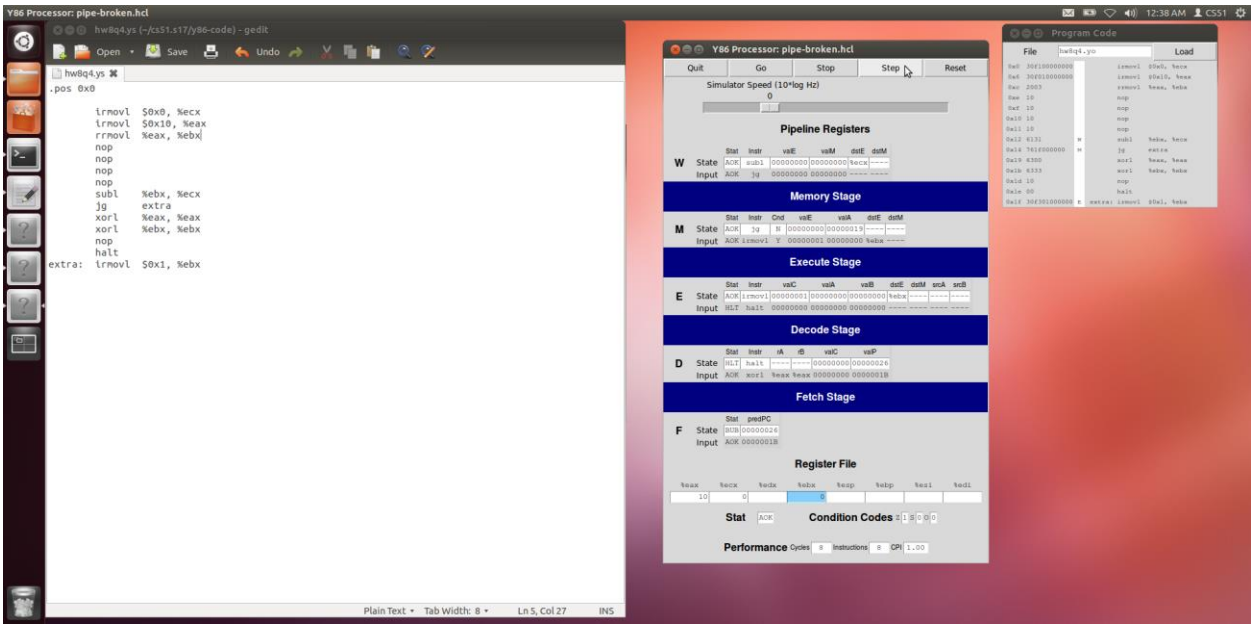
To fix this broken pipelining, the microarchitecture could do one of two things; first, it could insert a few nop statements (4 should be sufficient), known as bubbles, to slow down the pipeline, so that the proper value from the irmovl is put into %eax before the rrmovl instruction tries to read %eax. The second option would be to add extra pipes in the y86 uarch, so that if a rrmovl is coming down the pipeline, but a irmovl lies too close in front of it for it to properly read the value, it should read the value directly from the phase right in front of it, instead of from the memory.

Next, this code demonstrates a control hazard, when the code encounters a jg command. Here, the y86 does not know whether or not to jump to the desired address or not, because it has not yet executed the operation necessary to determine whether it should jump or not; here, it defaults to jumping, and executes an irmovl command at 0x1f that it should never have executed.

First, before the jg is encountered, the code is executing properly, but when the jg is fetched, the next fetch defaults to the address of the jg, 0x1f:

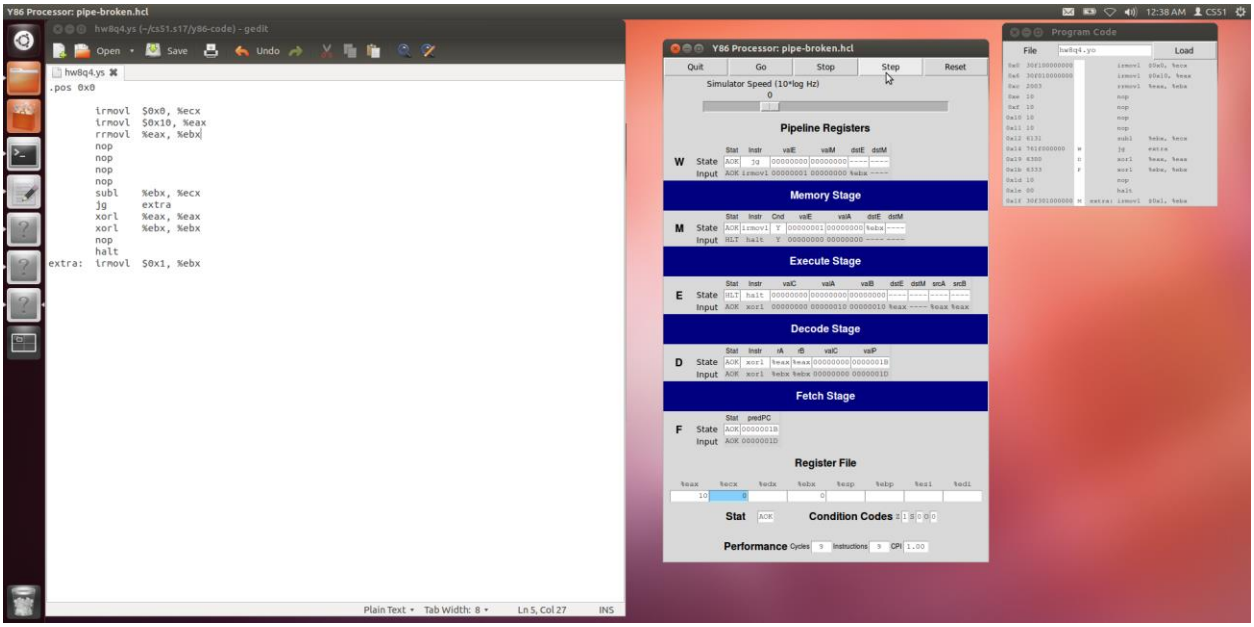


After this, it starts fetching instructions from 0x1f, even though clearly at this point %ebx is not greater than %ecx:

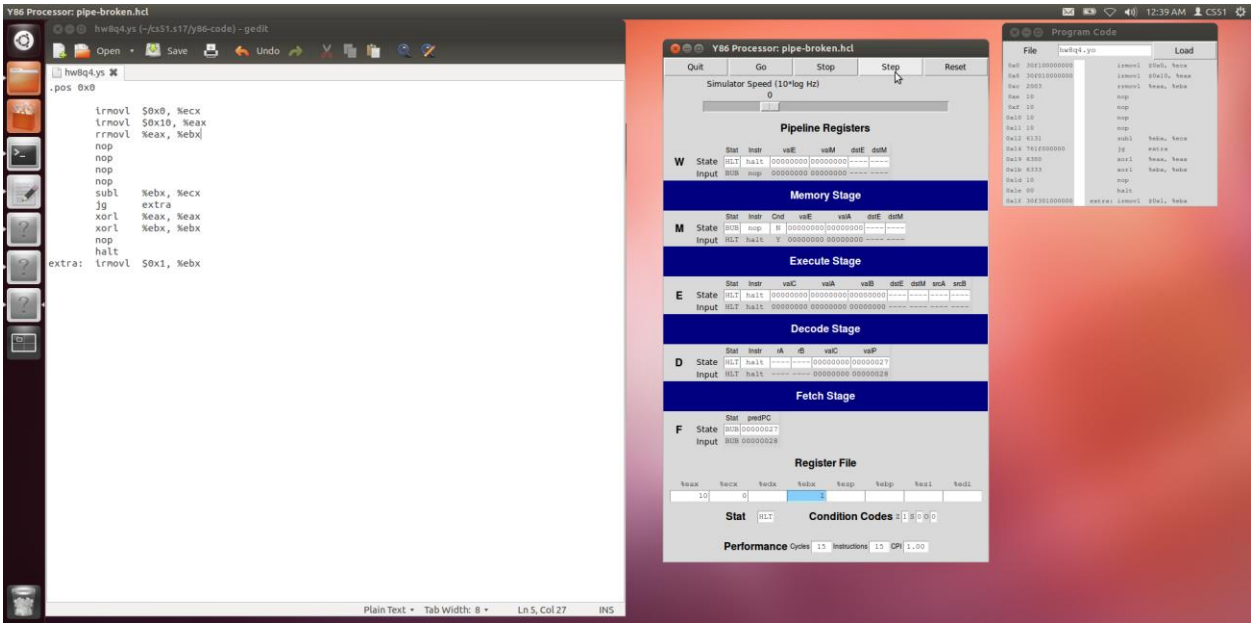


As a result of this, it ends up executing the code at 0x1f, and by the time of this picture, in its execute stage it is ready to insert 0x01 into %ebx; additionally, it has kept fetching after 1f, and has inserted halts into the pipeline, further compounding things.

Just after the state shown above, the jg command with the correct next address reaches the end of the pipeline, and the code begins trying to execute after the jg command:



It then continues to try to execute after the jg command, but unfortunately due to the halts inserted into the pipeline it does not even properly execute the xorl commands after the jg due to this control hazard; it does manage to also incorrectly execute 0x1f, moving 0x1 into %ebx by the time the program has fully halted:



This control hazard could be mitigated by the microarchitecture in two ways: first, it could have inserted a few nops (bubbles) after the jg command, so that by the time the jg command has been executed and the proper next instruction address has been determined do we feed it into the PC. The other option would be to select one decision to make when the jump is encountered, (to jump or not to jump) and if this jump was correct, keep going, if not, then backtrack and somehow reverse all of the incorrectly executed instructions, and then update the PC to the correct value.