Overview of the pipelined simulator

Authors: Haoyi Shi & Haoyuan Du

1. How does it work?

This is a simulator using C code to implement the behavior of a pipelined datapath.

It has five stages and five pipeline buffers as the graph below.

IF stage	IFID buffer	ID stage	IDEX buffer	EX stage	EXMEM buffer	MEM stage	MEMWB buffer	WB stage	WBEND buffer

Next, I will introduce each of stages and give a comprehensive explanation.

IF stage: All instructions (machine codes) has been preloaded on the Instruction Memory. For each cycle, one instruction will be fetched at the PC (program counter) address. This fetched instruction as well as PC+1 will be sent to the IFID buffer.

ID stage: All instructions will be decoded at this stage. Firstly, the instruction is read from IFID buffer and decoded into several fields depended on R-type, I-type or O-type. Secondly, some fields should read values from Register File and carry those values. Meanwhile, the offset field will be sign extended. Lastly, All fields and PC+1 will be sent to the IDEX buffer. Notice that IDEX only reads values from Register File, any “Bypassing” should not affect this stage.

EX stage: Firstly, EX stage reads all values such as RegA, RegB, offset, PC+1 etc. from IDEX buffer. Secondly, EX stage will check for data hazard and reslove hazard

with “Bypassing” (using temporary variables which saved EXMEM & MEMWB & WBEND buffers’ values from the last state) or a “Load Stall”. “Bypassing” is needed when the previous instructions get newer values (this value is newer than the value is about to execute), but haven’t written back to the Register File yet. Thus, we need the up to date information to execute by passing the newer value to it. When a “Bypassing” happened, EXMEM’s information takes the highest priority, after that is MEMWB, then WBEND. A special type of “Bypassing” is “Load Stall”, there is an instance such as: lw 2 1 0 || add 4 2 0. Notice that “lw” is going to load a new value and in the same cycle, “add” needs that value to do addition. However, lw can only load that new value until it reaches MEMWB buffer while add has been executed with the wrong, old value. To avoid that, we stall everything before the EXE stage and let the right side keep going, replacing the EXMEM buffer with a “NOOP” instruction, going to the next cycle, passing the new value back to the EXE stage. Thirdly, after checking data hazard, the ALU will do the appropriate calculation and another ALU will add PC+1 and offset together as the branch target. Finally, writing results (aluresult, branchtarget etc.) to EXMEM buffer.

MEM stage: Firstly, reading information from EXMEM buffer. Secondly, checking if the instruction is “beq” and the aluresult equals zero. If so, update PC by branchtarget, write results to the MEMWB buffer and “flush” all stages and buffers on the left side including EXMEM buffer with “NOOP” instructions. If not, then executing the certain instruction via or not via Data Memory as usual (for example: load value from Data Mem if it’s a lw, save value to Data Mem if it’s a sw, etc.).

Finally, write the results to MEMWB buffer.

WB stage: All results such as aluresult or new values from Data Memory are reading from MEMWB buffer and writing back to the Register File at this stage. Last but not least, back up results to the WBEND buffer (for Bypassing cases).

Notice: in the real datapath, each stage is being executed synchronously. Unfortunately, in C codes, it's asynchronously (it's not a JavaScript). But in each one cycle, each stage will be executed in a correct order, from left to right.

2. ~~Any~~ Tons of difficulties

1. Come up with a comprehensive "Bypassing" checking is really difficult. Since we may meet a case such as:

WBEND	lw	4	0	a
MEMWB	lw	4	0	b
EXMEM	add	1	6	7
IDEX	sw	2	4	4
IFID	...	...	...	...

The instance above is a state screenshot. First of all, there is no need for a Load Stall. Because no lw followed by an instruction that uses the register being loaded. However, we indeed need to check the "Bypassing" seriously. In IDEX, we have a sw that need to use R[4] to calculate the address, but R4 has been loaded twice in WBEND and MEMWB buffer. Thus, in our code, we have to implement a tons of "if" statement to accomplish each possibility of permutations, aiming to pass the up to date value. For example, we pass R[4] from MEMWB buffer instead of WBEND buffer in the case

above.

2. Another difficuly is about the test suite. In the 15th test case, we used the example from Project 1 which is a for loop case. In this case, the for loop executed five times which is hard to go through each cycle and get the answer by human hands (there are countless “flush”, “Load Stall + Bypassing” and “Bypassing” going on). All three of us (Haoyi, Haoyuan and our simualtor) got different answers with each other. So it takes time to figure out who is correct and whether we need to modify our codes. This progress was difficult and frustrating, but once we have completed it, we have never felt so clear and confident about the pipelined simulator.