1. The PC is incremented in the IF stage.

2. Branches, if not correctly predicted, incur 3 stalls. Branches are resolved at the end of the EX stage, and then propogated to the PC at the beginning of the MEM stage, which means the PC will be updated with the new value when the branch enters the WB stage.

3. We can easily reduce the number of stalls by moving branch resolution to the EX stage (by not having the branch target address computation adder drive `PCMux` directly, rather than first being latched by the EX/MEM register). We can be more aggressive by adding a comparator on the output of the register file (as shown in the second picture, we can potentially resolve branches in the ID stage (1 stall).

4. That is a problem. We'd need to add forwarding datapath that forwards to the end of the ID stage, and we'd need to stall one cycle when a branch follows an arithmetic instruction and two cycles when it follows a load.

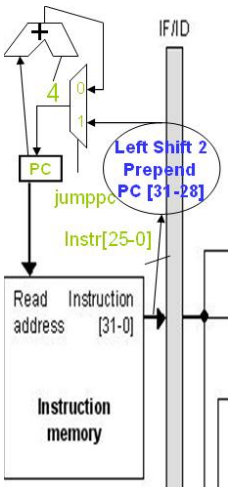
Problems

1. The pipeline will always start fetching the wrong instruction for a `j` instruction. (except in the case when the `target` is immediately next to the `j`, which is very unlikely). Therefore, the next instruction must be flushed and the correct instruction fetched from the jump target address once we figure out that the instruction we're executing is a `j` which is in the ID phase.

Table 1. Pipeline Diagram with stall caused by `j target`

Instruction	1	2	3	4	5	6	7
`j target`	IF	ID	EX	MEM	WB		
		IF×	IF	ID	EX	MEM	WB

2. (a) The changes to the datapath and the necessary control signals are shown:



(b) One control signal (`jumppc`) needs to be added, to decide what is the next PC value (either jump target for a jump instruction or PC + 4 otherwise[1]. Required logic: `jumppc = (opcode == j)`.

[1]The path for branch instructions is not shown; branches can be handled by placing the `PCMux` between the `jumppc` mux and the PC register.

3. (a) Forwarding can be performed from either the beginning of the `lw`'s BR stage to the `beq`'s MEM stage or the beginning of the `lw`'s WB stage to the `beq`'s BR stage.

   (b) Not predicting is problematic because the instruction after the branch will not be fetched until the cycle after the branch reaches the BR stage (4 stall cycles). If the target address is computed in the BR stage, then **predict taken** has the same problem. In contrast, **predict not-taken** will avoid all stalls on correct predictions.

   (c) If the instruction immediately following a branch is a store, and the branch is **incorrectly** predicted not-taken (*i.e.*, the branch should be taken), then the store might have completed before the branch resolves (overwriting some data). We can avoid this problem by stalling all pipeline stages before MEM when a store follows a branch; we inject a nop into the MEM stage. *Note: we have to do this even for branches that end up being predicted correctly, because we won't know in time whether the prediction is correct.*

```
Stall_at_EX = (EX/MEM.opcode == Branch) && (ID/EX.opcode == Store);
```

Instruction	1	2	3	4	5	6	7	8
`branch`	IF	ID	EX	MEM	BR	WB		
`store`		IF	ID	EX	**EX**	MEM	BR	WB
..			IF	ID	**ID**	EX	MEM	BR
..				IF	**IF**	ID	EX	MEM
..					–	IF	ID	EX

   (d) If branch targets are computed in the ID stage then a predict **taken** policy would have 1 stall cycle on a correct prediction and 4 stall cycles on an incorrect prediction. In contrast, a predict **not-taken** policy would only have a stall cycle when the successor was a store (only about 15% of instructions are stores on average) on a correct prediction and 4 stall cycles on an incorrect prediction. Naively, it would seem that **predict not-taken** is better, but it really depends on the relative frequency of taken and not-taken branches in the code we are running. *Note: we'll assume the likelihood of the post-branch instruction being a store is 15%.*

   The break even point of the two policies is at x% taken branches where:

$$predict\_taken\_penalty(x) = predict\_nottaken\_penalty(x) \tag{1}$$

$$1(x) + 4(1 - x) = .15(1 - x) + 4(x) \tag{2}$$

$$x = 3.85/6.85 = \ 56\% \tag{3}$$

   If more than 56 percent of branches are taken then predicting **taken** will be a better policy; otherwise, predicting **not-taken** is a better policy.