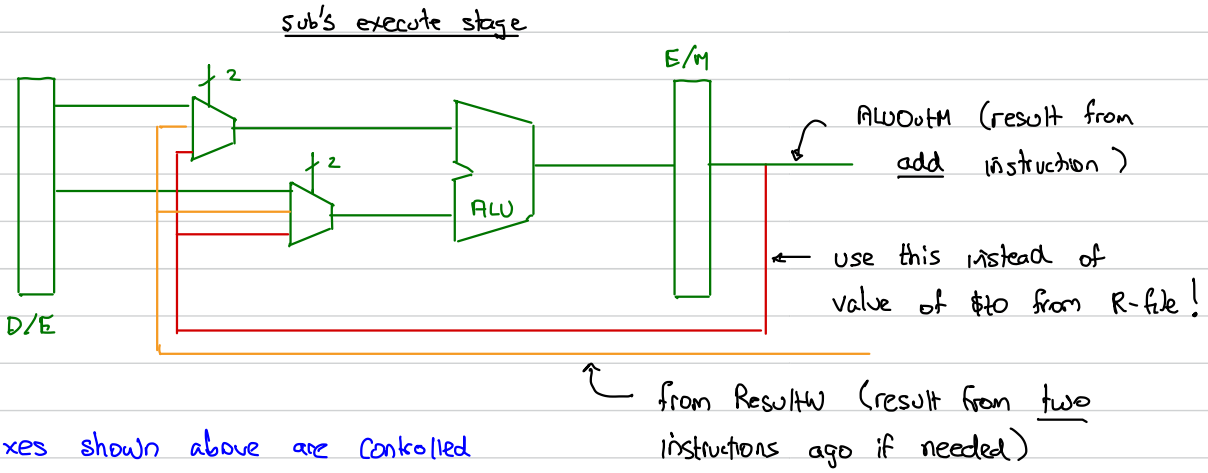
Managing Data Hazards

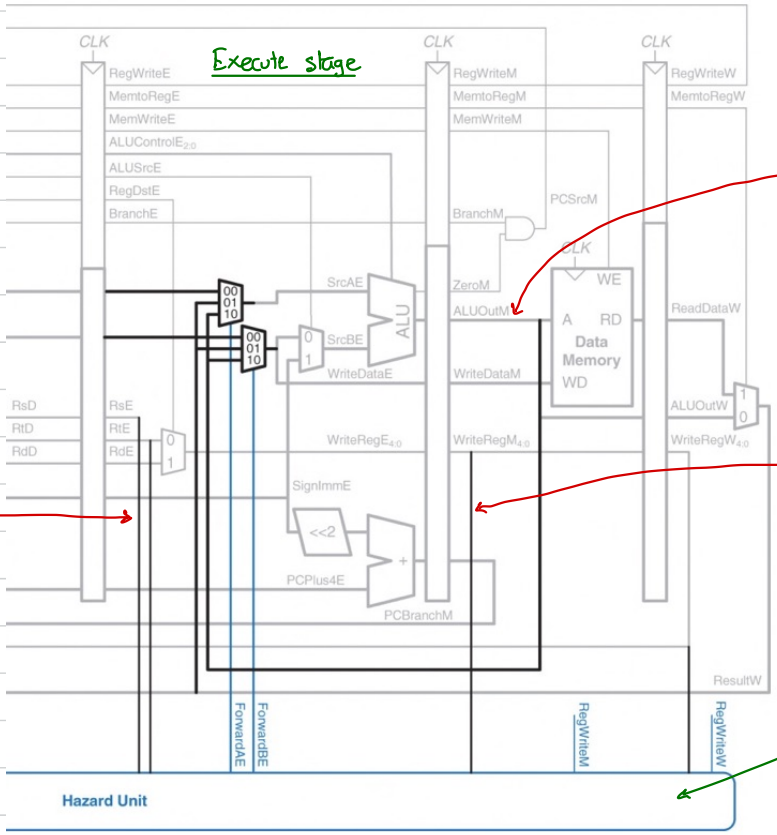
Forwarding was one solution:

```
add  $t0, $t1, $t2
sub  $t3, $t0, $t4
```

F — D — E — M — W
F — D — E — M — W



The new muxes shown above are controlled by the hazard detection circuit (which keeps track of what GPRs are needed and what has most recently been calculated).



add's result is available here!

sub wants to use RsE = $t0 ($01000_2$) Calculated in previous clock cycle (now in the 'M' stage)

add's result is destined for WriteRegM = $t0 ($01000_2$)

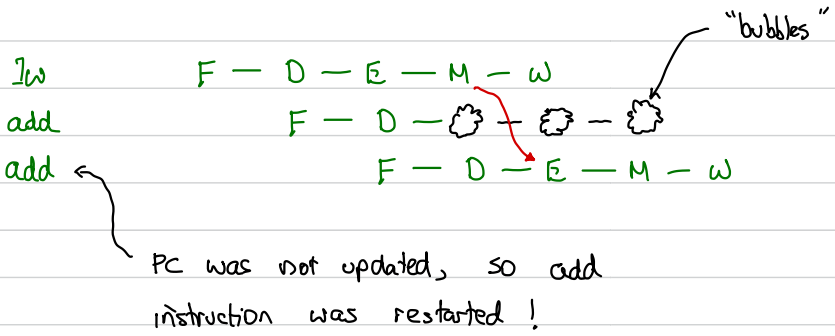
Hazard unit senses RsE = WriteRegM; then controls muxes.

Another solution was stalling and forwarding

```
lw    $8, ($9)
add   $16, $17, $8
```

The hazard unit senses trouble in the Decode stage

- control signals all set to zero; turns into a nop instruction.
- F/D pipeline register is not updated, so the Fetch stage holds onto the add instruction
- PC not updated either



Control hazards — making branches work better

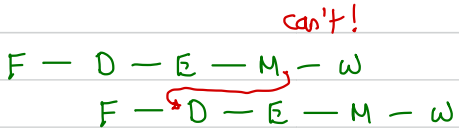
Branch decision is Figures 7.47 and 7.50 made late in the pipeline (Memory stage)

— three instructions entering the pipeline after beq might need to be cancelled.

Better: Move branch decision to Decode stage.

- but still not perfect — hazards still!

```
lw   $17, ($4)
beq  $17, $0, foo
```



— needs a 2-cycle stall and forwarding to fix.

This is the primary motivation for dynamic branch prediction