I Flare CPU Pipeline Stalls With `PipeMemRmw`

I.A Pipeline Stages `EX` and `MEM`

I.A.1 Loads/Stores/Atomics with Memory

- `MEM` should stall when there's a load/store instruction that has a dcache miss. This means that `EX` needs to stall too.
  - Situations:
  - `EX` has a **load**-based RAW pipeline hazard with an instruction that doesn't stall outside of this situation.
    - Example code:

      ```
      ldr r1, [r2]
      add r3, r1
      ```
    - Stalling states:
      - `EX`: stalls until it can forward from `MEM`.
      - `MEM`: stalls until it finishes the **load** instruction.
    - `EX` checks for a **load** instruction and changes FSM state upon `EX.up.isFiring === True`.
    - What happens if after `EX.up.isFiring` that there's an instance of `EX.haltIt()`? Does `MEM` still see the instruction that was previously in `EX`? The answer should be **yes**.
    - If there's a dcache miss, `MEM` should stall by doing `MEM.duplicateIt()` and also deassert `MEM.up(...).modMemWordValid`
      - Also, stop `MEM.up(...).modMemWordValid` upon no longer doing `MEM.duplicateIt()`.
    - If there's a dcache hit, `MEM` should not stall and I think that `EX` doesn't need to either? That's cool! I thought there'd be a hazard even in this case, but if the address loaded from was in dcache, I can already forward, as I already have the result in that case (i.e. I already read it from the BRAM).
  - `EX` has a **load**-based RAW pipeline hazard with an instruction that **does** stall normally in `EX`, i.e. `mul`, `div`, etc.
    - I can't have `MEM.up.isFiring` if I do `EX.haltIt()` immediately following a detected **load** instruction. Thus I think I should do `EX.duplicateIt()` but with pipeline bubbles sent from `EX` to `MEM` until `MEM.up.isFiring`/`EX.down.isFiring`.
      - Pipeline bubbles should be something other than `modMemWordValid` in the actual Flare CPU pipeline, as `modMemWordValid` is only for `PipeMemRmw` itself.
    - Upon `EX.down.isFiring` (and following the detected **load** in `EX.up.isFiring`), `EX` should save a copy of the data being forwarded from `MEM`, then on the following cycle `EX` can send it to the multiplier, divider, etc. For those following cycles until `EX.up.isFiring`, `EX` should do `EX.haltIt()` until the mul, div, etc. completes. On the cycle following the completed instruction, don't do `EX.haltIt()` and maintain the value of `tempUpMod(2)` until `EX.up.isFiring`.
      - I have existing code in the `PipeMemRmw` tester stuff that already does this kind of capture, in the fake `MEM` stage, but `EX` in the Flare CPU needs to do the same thing with its own instruction.
    - `PipeMemRmw` already checks for `up.isValid` when capturing the input pipe payload into `upExt(1)(extIdxSingle)`, so we can at least see the instruction via `upExt(1)(extIdxSingle)` in the `doModInModFrontFunc()` generic. Actually, `doModInModFront()` takes `tempUpMod(1)`, which does still contain the value of `upExt(1)(extIdxSingle)`.
    - When `EX` stalls, *maybe* it should not change the value of `tempUpMod(2)`? Instead *maybe* `EX` should change the value of `tempUpMod(2)` upon no longer stalling via `EX.duplicateIt()`. `tempUpMod(2)` should have its value set to the output of `EX` upon the multiply/divide finishing, at which point we stop doing `EX.duplicateIt()`, after which `tempUpMod(2)`'s value should be held constant until `EX.up.isFiring`.

- Is everything similar for **store** instructions but without the forwarding? I guess just do everything that was needed for a **load** instruction besides the forwarding. Thus deassert a **store** instruction's `modMemWordValid` to prevent its "results" from being forwarded. This approach is guaranteed to work and will use less logic than specially handling **stores**.
- **Is there actually no hazard here? Other than that we can't forward upon a dcache miss?**

I.B