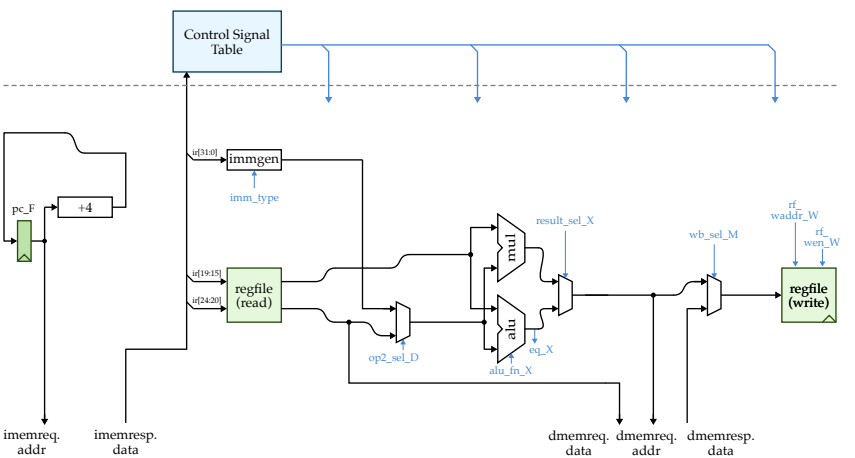
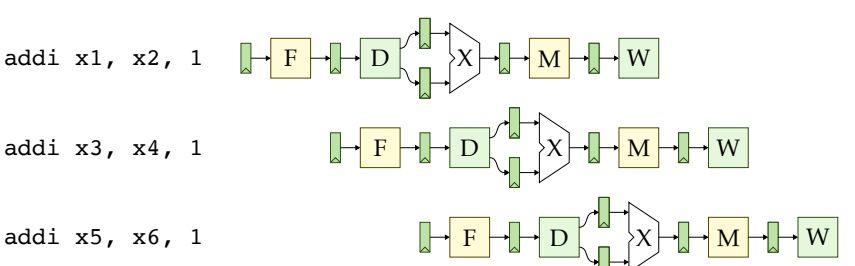
3. Five-Stage Pipelined Processor

- Incrementally develop an unpipelined datapath
- Start with just arithmetic and memory instructions
- Keep data flowing from left to right
- Position control signal table early in the diagram
- Divide datapath/control into stages by inserting pipeline registers
- Keep the pipeline stages roughly balanced
- Forward arrows should avoid "skipping" pipeline registers





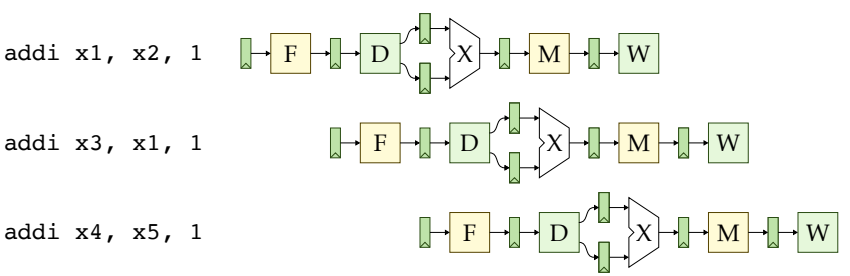
Pipeline diagrams

`addi x1, x2, 1`													
`addi x3, x4, 1`													
`addi x5, x6, 1`													

What would be the total execution time if these three instructions were repeated 10 times?

3.1. RAW Data Hazards Through Registers

RAW data hazards occur when one instruction depends on a data value produced by a preceding instruction still in the pipeline.



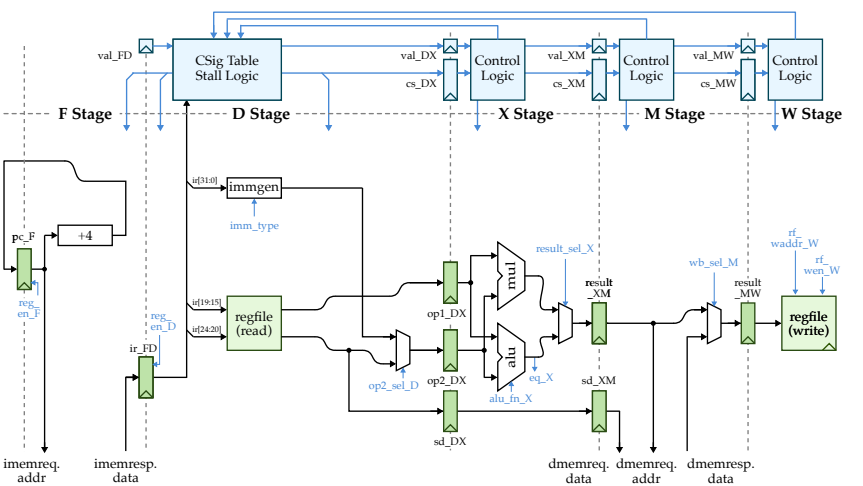
`addi x1, x2, 1`													
`addi x3, x1, 1`													
`addi x4, x5, 1`													

3.2. RAW Data Hazards → Hardware Stalling

- Hardware includes control logic that freezes later instructions (in front of pipeline) until earlier instruction (in back of pipeline) has finished producing data value.

`addi x1, x2, 1`												
`addi x3, x1, 1`												
`addi x4, x5, 1`												

Modifications to datapath/control to support hardware stalling



Deriving the stall signal

```
stall_waddr_X_rs1_D =
  val_D && rs1_en_D && val_X && rf_wen_X
        && (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)

stall_waddr_M_rs1_D =
  val_D && rs1_en_D && val_M && rf_wen_M
        && (inst_rs1_D == rf_waddr_M) && (rf_waddr_M != 0)

stall_waddr_W_rs1_D =
  val_D && rs1_en_D && val_W && rf_wen_W
        && (inst_rs1_D == rf_waddr_W) && (rf_waddr_W != 0)

... similar for stall signals for rs2 source register ...

stall_D = val_D
        && (    stall_waddr_X_rs1_D || stall_waddr_X_rs2_D
             || stall_waddr_M_rs1_D || stall_waddr_M_rs2_D
             || stall_waddr_W_rs1_D || stall_waddr_W_rs2_D )

stall_F = stall_D
```

Draw the pipeline diagram assuming data hazards are resolved with hardware stalling

`addi x1, x0, 100`													
`addi x2, x0, 4`													
`add  x3, x1, x2`													
`addi x0, x0, 0`													
`addi x4, x3, 3`													

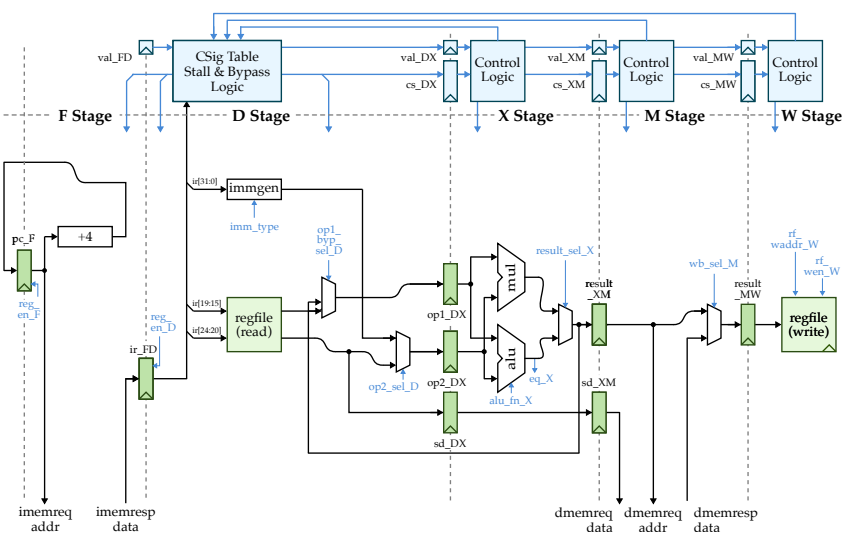
3.3. RAW Data Hazards → Hardware Bypassing

Hardware allows values to be sent from an earlier instruction (in back of pipeline) to a later instruction (in front of pipeline) before the earlier instruction has left the pipeline. Sometimes called "forwarding".

Pipeline diagram showing hardware bypassing for RAW data hazards

`addi x1, x2, 1`													
`addi x3, x1, 1`													
`addi x4, x5, 1`													

Adding single bypass path to support limited hardware bypassing

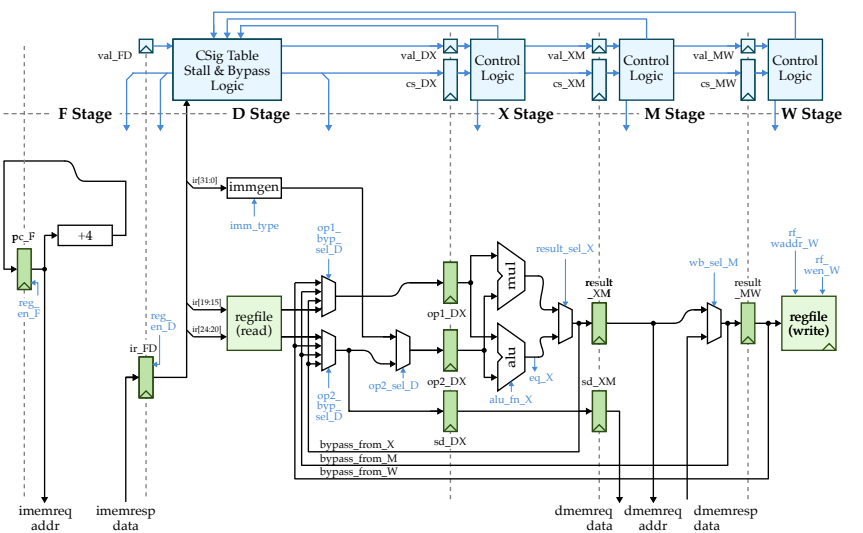


Deriving the bypass and stall signals

```
stall_waddr_X_rs1_D  = 0
bypass_waddr_X_rs1_D =
  val_D && rs1_en_D && val_X && rf_wen_X
        && (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)
```

Pipeline diagram showing multiple hardware bypass paths

`addi x2, x10, 1`													
`addi x2, x11, 1`													
`addi x1, x2, 1`													
`addi x3, x4, 1`													
`addi x5, x3, 1`													
`add  x6, x1, x3`													



Handling load-use RAW dependencies

ALU-use latency is only one cycle, but load-use latency is two cycles.

`lw x1, 0(x2)`													
`addi x3, x1, 1`													

`lw x1, 0(x2)`													
`addi x3, x1, 1`													

```
stall_load_use_X_rs1_D =
  val_D && rs1_en_D && val_X && rf_wen_X
        && (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)
        && (op_X == lw)

stall_load_use_X_rs2_D =
  val_D && rs2_en_D && val_X && rf_wen_X
        && (inst_rs2_D == rf_waddr_X) && (rf_waddr_X != 0)
        && (op_X == lw)

stall_D =
  val_D && ( stall_load_use_X_rs1_D || stall_load_use_X_rs2_D )

bypass_waddr_X_rs1_D =
  val_D && rs1_en_D && val_X && rf_wen_X
        && (inst_rs1_D == rf_waddr_X) && (rf_waddr_X != 0)
        && (op_X != lw)

bypass_waddr_X_rs2_D =
  val_D && rs2_en_D && val_X && rf_wen_X
        && (inst_rs2_D == rf_waddr_X) && (rf_waddr_X != 0)
        && (op_X != lw)
```

Pipeline diagram for simple assembly sequence

Draw a pipeline diagram illustrating how the following assembly sequence would execute on a fully bypassed pipelined TinyRV1 processor. Include microarchitectural dependency arrows to illustrate how data is transferred along various bypass paths.

`lw x1, 0(x2)`													
`addi x2, x1, 4`													
`lw x3, 0(x2)`													
`lw x4, 0(x3)`													
`addi x4, x4, 1`													
`addi x4, x4, 1`													

3.4. RAW Data Hazards Through Memory

So far we have only studied RAW data hazards through registers, but we must also carefully consider RAW data hazards through memory.

```
sw x1, 0(x2)
lw x3, 0(x4) # RAW dependency occurs if R[x2] == R[x4]
```

`sw x1, 0(x2)`													
`lw x3, 0(x4)`													