THINKING ABOUT PIPELINING

- The textbook solution is only one solution to the pipeline.
- Dr. Meier presented an alternative solution in his lecture sections.
- Any solution that implements the ISA is a correct microarchitecture (μA).
- The pipeline uses its circuits efficiently because each stage of instruction execution has some instruction within it during every clock period. Thus, we say "five instructions are in flight."
- One challenge with pipelining is overcoming data dependencies during decode. The decoding instruction will decode incorrect data if it depends on the write-back register of some instruction that is ahead of it in the pipeline. Thus, either pipeline stalls or forwarding is needed. Forwarding is the preferred approach because it keeps the pipeline moving.
- A second challenge with pipelining is keeping correct instructions fetching on each clock period. Branch hazards require flushing of an incorrect instruction in IFID when the decode circuit decides the branch is taken.
- A final challenge with pipelining is keeping the pipeline moving in the presence of data loads from memory. On a cache memory hit, the pipeline must stall until the LDR instruction reaches WB. On a cache memory miss, the pipeline will experience a longer stall because the cache controller will hold the pipeline in stall until the data moves up from lower levels of memory.

WHAT IS THE PURPOSE OF PIPELINE REGISTERS?

- Pipeline registers divide the processor circuitry into five stages: fetch, decode, execute, memory access, and write-back.
- The circuit is "divided" because clock edges are needed to sample voltage from one stage into the next.
- The pipeline registers have classic names: IFID, IDEX, EXMEM, MEMWB.
- Note that both PC and REGFILE are also technically pipeline registers because they sample and hold numbers that form part of the state of the machine. The PC must stall when IFID stalls, for example.
- Some pipeline registers must flush – synchronously create a NOP because of pipeline hazards. These pipeline registers have a flush control signal.
- Some pipeline registers must stall – hold data values into the next clock period rather than sampling new ones. These pipeline registers have a stall control signal. Stall is really a "load" control signal – a signal most students are already comfortable with.
- One approach in university classes is to make a general pipeline register that has both stall and flush. This approach provides a uniform component used in schematic layout and simulation. Note, however, that two control signals increase complexity and thus likely results in both larger and slightly slower registers. In practice, architects will only build stall and flush into registers that need them.

WHAT PIPELINES?

Inter-stage pipeline registers must sample any signal created by the stage on the left for use by any stage on the right. For example, the IFID register samples any signal created in fetch that is used by decode or later stages. Here is an initial set – you may find that there are more as you review and study!

TABLES MAY BE INCOMPLETE! STUDY YOUR CIRCUITS.

DATA

IFID	IDEX	EXMEM	MEMWB
IBUS	RD1		
PC+4	RD2	RD2	
PC+8	RDEST (A3)	RDEST (A3)	RDEST (A3)
	IMM32		
		ALUF	ALUF
			MEMQ

CONTROL SIGNALS

IFID	IDEX	EXMEM	MEMWB
	ALUS		
	MEMWR	MEMWR	
	REGSRC	REGSRC	REGSRC
	REGWR	REGWR	REGWR

HOW DO I HANDLE BRANCH HAZARDS?

Branch control can be placed in decode or in execute. If a separate branch controller is placed in execute, the flag register outputs are tapped and used to make a PCSRC decision. A taken branch results in a flush of both IFID and IDEX because both the fetching (PC+8) and decoding (PC+4) instruction are now wrong. This introduces two bubbles into the pipeline.

But branch hazards ***can*** be detected in the decode stage to reduce the number of bubbles to one. In ARMv4, a previous instruction has set the flag bits in the flag register. The nearest dependency could be one instruction ahead, for example:

```
CMP R9, R10
BEQ POWER2
```

Thus, flag register output bits are wired back to the decode stage. The flag register is modified to update on the opposite clock edge as the pipeline registers when placing the branch controller in the decode stage. This allows you to avoid a stall.

Assuming that rising-edges advance the pipeline:

pipeline advanced, ALU calculates, bits ready	flag register samples CVNZ, decode makes branch decision

This approach requires a clock period that allows enough time after the flag register sample for the decode circuitry to interpret CVNZ and change voltage on PCSRC if needed.

- Branch not taken: do nothing
- Branch taken: flush instruction behind you, route the branch address
- Result: no stall, one bubble inserted

Remember that each bubble contributes to decreased performance. Bubbles are units of time where no operation completes (NOP). These are undesirable units of time. While advancing the branch decision to decode removes one bubble, it does not fully solve the problem. Computer architects are heavily invested in **branch prediction** so that the fetch circuit rarely inserts incorrect instructions in the stream. A wide variety of branch prediction techniques have been implemented including both compile-time techniques and run-time (dynamic) techniques. Today, most processors implement a statistical dynamic run-time table that updates as branches are executed. The statistical model for each branch gets better as it executes repeatedly. These processors achieve greater than 90% and some closer to 98% success. **Accurate prediction** results in constant pipeline flow and no bubbles are inserted. So, to achieve 98% success implies that architects have achieved techniques that keep the pipeline full most of the time. Branch prediction is a topic in the Computer Architecture 2 elective course.

HOW DO I HANDLE DATA HAZARDS

There are two types of data hazards:

- register-use, and
- load-use

Register-use hazards exist because a data source address, Rn or Rm in the case of ARMv4, is the same as the destination bit field of instructions that are farther down the pipeline but not yet finished. Load-use hazards exist because your data sources are the destination register of an LDR instruction.

There are three basic hardware approaches to handling data hazards.

1. Stall the pipeline until the instruction you depend on completes.
2. Forward the result from the function ahead of you back to the ALU for use.
3. Reordering the assembly language instructions so that instructions that don't change the final calculated program results are moved between the instructions experiencing the hazard. This "out-of-order" instruction sequencing can be done by the compiler at compile-time or identified by a hardware out-of-order dispatch circuit at run-time. These approaches are advanced topics examined in MSOE elective courses in computer architecture.

HANDLING REGISTER-USE HAZARDS WITH FORWARDING

- A forwarding controller must be placed in either decode or execute.
- Placing the forwarding controller in the decode circuit results in multiplexers to select RD1 and RD2 to the left of IDEX. These multiplexers receive the REGFILE RD1 and RD2 as well as the ALUF result from EX, MEM, and the WB result from WB. Because the opcode bits, command bits, and other machine code bits are available in decode they can be directly connected to the forwarding controller and do not need to pipeline.
- Placing the forwarding controller in the execute circuit requires no additional multiplexers in decode but does expand multiplexers in execute. The execute multiplexers now must include the ALUF result from MEM and the WB result from WB. Because this controller resides in execute, the opcode bits, command bits, and other machine code bits needed to complete the comparison checks must be piped from decode to execute through the IDEX register.

HANDLING LOAD-USE HAZARDS WITH FORWARDING

- Similar to register-use, the forwarding controller handles this type of hazard. Thus, it can be handled in either decode or execute depending on controller placement.
- LDR requires the pipeline to stall until it reaches WB. This is because LDR is accessing memory during the memory phase and the clock period has been set to be just greater than this access delay. Thus, there is no time for any use of the data before the clock edge – the controller must stall one clock period while this access occurs and the result reaches WB. The result can then be forwarded from WB. In class, the possibility of forwarding MEMQ was explored in the forwarding multiplexers. In practice, the stall is most often inserted because of the tight timing constraints and MEMQ is not routed and multiplexed in decode.

SPECIAL CASES OF FORWARDING

- ARMv4 does have one special case of forwarding: STR.
- The STR instruction format is: STR Rd, [Rn+imm].
- Forwarding for Rn is no different than any other Rn forwarding condition and thus the forwarding controller will already be handling this potential dependency.
- But, STR does not use Rm and thus any hazard on Rd will not be handled correctly.
- Here is an example:

```
ADD R5, R6, R7
STR R5, [R9]
```

Thus, the forwarding controller must also compare the STR Rd field against the destination addresses ahead of it in the pipeline!