6.823 Computer System Architecture
Module #1 Problem Set #2 Solutions

Last Updated: 3/2/2017

http:/csg.csail.mit.edu/6.823/

Problem M4.1: Fully-Bypassed Simple 5-Stage Pipeline

Problem M4.1.A — Stall

We still need the logic for stalls, because we cannot prevent load-use hazard. If a load instruction is followed by an instruction which takes the loaded value as a source operand, we cannot avoid stalling for a cycle. The following instruction sequence illustrates this hazard.

```
LW  R1, 0(R2)     # R1 <- M[R2]
ADD R3, R5, R1    # R1 is a source operand of ADD (data dependency)
                  # The correct value of R1 is not available when
                  # ADD is in ID stage.  So it has to stall for a cycle.
```

Problem M4.1.B — Bypass Signal

Here are the bypass conditions.

Bypass $_{EX->ID}$ ASrc = $(rs_D=ws_E).we\text{-}bypass_E.re1_D$

Bypass $_{MEM->ID}$ = $(rs_D=ws_M).we_M.re1_D$

Bypass $_{WB->ID}$ = $(rs_D=ws_W).we_W.re1_D$

Priority: Bypass $_{EX->ID}$ > Bypass $_{MEM->ID}$ > Bypass $_{WB->ID}$
(In order to execute a given program correctly, the value from the latest producer must be taken if multiple bypass paths are active.)

Problem M4.1.C — Partial Bypassing

It is an open question and there is no single correct answer. Here are a couple of issues to consider as a guideline.

First, you may consider the penalty for not having all the bypass paths. If we don't have the bypass path EX→ID, we have to stall for three cycles for the hazard to be resolved. Likewise, not having MEM→ID results in a stall of two cycles, and not having WB→ID, in one. Therefore, you can conclude that the bypass path between EX→ID is the most beneficial.

Secondly, the best bypass path depends on the access patterns of data. The EX→ID bypass path is effective if a producer instruction is followed by a consumer, except load-use cases (See solution for M4.1.A). On the other hand, the MEM→ID bypass path works best if there are many load-use cases or many (producer, consumer) pairs have an independent instruction between them. Likewise, the WB→ID bypass path helps when many (producer, consumer) pairs are separated by exactly two independent instructions.

Problem M4.2: Basic Pipelining

Problem M4.2.A **Mux Control Signals (1)**

PCEn = (S==Execute)

IREn = (S==I-Fetch)

AddrSrc = Case S

I-Fetch => PC

Execute => ALU

Problem M4.2.B **Modified pipeline**

A stall can occur in 2 different cases.

1. A structural hazard in the shared memory.
   LD R1, 16(R2)
   Any instruction following this LD instruction should be stalled.

2. The other is caused by a control hazard, because we don't have a delay slot.
   J 200
   Any instruction following this J instruction should be flushed.

Problem M4.2.C **Mux Control Signals (2)**

PCEnable = not ((opcode == LW) or (opcode == SW))

AddrSrc = Case opcode

not (LW or SW) => PC

(LW or SW) => ALU

IRSrc = Case opcode

LW or SW or Jump or Br_{taken} => nop

Else => Mem

Problem M4.2.D

Time	PC	"IR"	PCenable	PCSrc1	AddrSrc	IRSrc
t_0	I_1:100	-	1	pc+4	PC	Mem
t_1	I_2:104	I_1	1	Pc+4	PC	Mem
t_2	**I_3:108**	**I_2**	**0**	*****	**ALU**	**Nop**
t_3	**I_3:108**	**-**	**1**	**pc+4**	**PC**	**Mem**
t_4	**I_4:112**	**I_3**	**1**	**jabs**	**PC**	**Nop**
t_5	**I_7:312**	**-**	**1**	**pc+4**	**PC**	**Mem**
t_6	**I_8:316**	**I_7**	**1**	**pc+4**	**PC**	**Mem**

Problem M4.2.E **Self-Modifying Code**

The answer is no. The hazard is resolved by the datapath itself because (1) memory accesses are serialized by the stall logic at the shared memory and (2) memory write takes only one cycle.

Problem M4.2.F

Due to this rerouting we will now have to stall even if it is an ALU instruction.

Problem M4.2.G **Architecture Comparison**

The Princeton architecture is cheaper than the Harvard architecture, but the Harvard architecture is faster than the Princeton architecture.

Problem M4.3: Processor Design (Short Yes/No Questions)

Problem M4.3.A **Interlock vs. Bypassing**

No. Data dependencies are preserved with either interlocks or bypassing, so the processors always generate the same results. Bypassing improves performance by eliminating stalls.

Problem M4.3.B **Delay Slot**

Yes. The instruction following a taken branch is executed on processor A, but killed on processor B so the processors can generate different results.

Problem M4.3.C **Structural Hazard**

No. Both processors retrieve the same data values. There is only a performance difference because processor A must stall an instruction fetch to allow a load instruction to access memory.

Problem M5.1: Pipelined Cache Access

Problem M5.1.A

Ben's initial datapath design is shown below:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check	Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-back

Alyssa suggests combining the third and fourth stages, which would result in the following design (used in the MIPS R4000 processor discussed in Appendix A of the textbook):

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-Back

This scheme allows an instruction to be read from the register file before it is known whether the instruction is valid. However, reading values from the register file does not affect processor state and thus does not affect the correctness of the program execution. If the tag check fails—meaning that the fetched instruction is invalid—the incorrect instruction can be replaced with a NOP in the Execute stage, and the processor can wait for the correct instruction to be brought into the I-cache.

That raises the question of whether Ben can similarly combine the data cache tag check stage with the write-back stage. Theoretically, the answer is yes, although the issues involved with combining these two stages make it highly impractical. Thus, both answers are acceptable—the important thing to consider is the reasoning used. Combining the last two stages would result in the following pipeline:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check & Write-Back

The obvious problem with this scheme is that a load instruction that misses in the data cache will write an incorrect value into the register file—therefore merging the stages does not work. This

is correct. However, one can also argue that the scheme can be made to work by modifying the pipeline. This argument is based on the fact that even if a load instruction places incorrect data into a register, the load can re-execute and place the correct data into the register, overwriting the wrong value. As a side note, it should be pointed out that allowing processor state to be incorrectly updated in a machine which implements precise interrupts would not work without substantial hardware modifications. However, ignoring the issue of interrupts (which had not been covered in lecture at the time of the problem set), there is a more fundamental issue with this approach. Ben's pipeline currently has no means of correctly re-executing the load instruction. Simply flushing the pipeline on a data cache miss and restarting execution with the load instruction does not work because of the following type of instruction:

```
LW R1, 0(R1)
```

If the load results in a D-cache miss, it will have overwritten the value in R1 before it re-executes, meaning that the incorrect address will be calculated the second time around. Another alternative is to store the address once it has been calculated in the Execute stage. This requires special address registers in each pipeline stage starting with D-Cache Address Decode. But another problem is the fact that cache access is pipelined, so a load in the write-back stage that has caused a D-cache miss has to be sent backwards in the pipeline (along with the correct address) in order to access the cache once the correct data has been fetched. This requires additional bypass paths in the processor. In general, speculatively updating processor state requires rollback mechanisms to be implemented. Backing up the pipeline is the approach used in the MIPS R4000 in the event of a data cache miss, but the tag check and write-back stages are separate.

Problem M5.1.B

Ben's current design does not work for data writes because the tag needs to be checked before the cache is updated. One solution is to add a fourth stage which handles the actual write in the event of a cache hit. However, unless the cache can handle two simultaneous accesses, this scheme does not allow a store to be in this fourth stage at the same time that another memory operation is in the D-Cache Array Access stage. A better solution is to use a delayed write buffer (also see Problem M5.2). The store data is written into the write buffer, and if a hit occurs in the D-Cache Tag Check stage, the data will be written into the cache at a later time (for example, when the next store instruction is processed)—the processor can continue execution as normal. This requires load instructions to check the write buffer as well as the cache to ensure that the correct value is read. With this scheme, a three-stage pipeline can be maintained for the data cache.

Problem M5.1.C

Ben's final 8-stage pipeline is shown below:

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	D-Cache Address Decode	D-Cache Array Access	D-Cache Tag Check	Write-Back

This pipeline uses direct-mapped instruction and data caches. Replacing these direct-mapped caches with set-associative caches could potentially reduce the miss rate, at a possible cost in hit time. However, a close examination of the pipeline and the diagram for a set-associative cache (seen in Problem M2.1.B) shows that the I-cache must be direct-mapped. For a set-associative cache, when a word is being read, the result of the tag check is used as an enable signal for the value being read. However, in the above pipeline, the instruction is needed at the beginning of the I-Cache Tag Check stage so that it can be decoded in parallel with the tag check. Thus, the I-cache must be direct-mapped.

For the data cache, the tag check occurs in its own stage. This makes it possible to use a set-associative cache, since the data for a load instruction isn't needed until the beginning of the Write-Back stage. However, in practice this would probably be a bad idea, since the extra delay required to wait for the tag check before driving out the data might lengthen the clock period.

Problem M5.1.D

Pipelining the caches has a harmful effect on branches. If conditional branch instructions resolve in the Execute stage, then the processor's branch delay is 3 cycles, as shown by the following example in which there are no delay-slot instructions and the datapath is fully-bypassed:

```
    ADDI R1, R0, #1
    BEQ  R1, R0, L1
    SUB  R2, R3, R4
L1: AND  R5, R6, R7
```

	t1	t2	t3	t4	t5
IAD	BEQ				SUB
IAA	ADDI	BEQ			
ITC/ID		ADDI	BEQ		
EX			ADDI	BEQ	
DAD				ADDI	BEQ
DAA					ADDI
DTC					
WB					

Problem M5.1.E

Since a data cache access takes 3 cycles, it will take more cycles (as compared to the five-stage pipeline) to obtain the result of a load instruction. If an instruction depends on the load, a simple scheme is to wait until after the D-Cache Tag Check stage before bypassing the load value. This will ensure that the dependent instruction does not execute with incorrect data. An interlock can be used to implement this solution. If an instruction in the Instruction Decode stage needs to read the result of a load instruction that is either in the Execute, D-Cache Address Decode, D-Cache Array Access, or D-Cache Tag Check stages, then that dependent instruction will be stalled until the load reaches the Write-Back stage (at which point the load value will be bypassed to the Execute stage). This is illustrated by the below example.

```
LW R1, 0(R2)
ADD R3, R1, R2
```

	t1	t2	t3	t4	t5	t6	t7
IAD	ADD						
IAA	LW	ADD					
ITC/ID		LW	ADD	ADD	ADD	ADD	
EX			LW				ADD
DAD				LW			
DAA					LW		
DTC						LW	
WB							LW

As shown by the above resource usage diagram, the load delay for this scheme is 3 cycles.

Problem M5.1.F

Another alternative to waiting until after the D-Cache Tag Check stage before bypassing the load value is to bypass the value at the end of the D-Cache Array Access stage. If there is a tag mismatch, the processor will wait for the correct data to be brought into the cache; then it will re-execute the load and all of the instructions behind it in the pipeline. In order to implement this scheme, only the program counter of the load instruction needs to be saved in the event of a tag mismatch. The load instruction will be nullified (as well as instructions behind it in the pipeline). When the **DataReady** signal is asserted (indicating that the load data is now available in the cache), the processor can restart the load instruction and continue as normal. The benefit of this scheme is that the load delay is now reduced to 2 cycles.

Problem M5.1.G

Even with the scheme in Problem M5.1.F, the load delay is 2 cycles, while it was only 1 cycle in the original 5-stage pipeline (although to be fair, the cycle time should be shorter in the 8-stage pipeline). One solution to this problem is the addition of a fast-path cache that can be accessed in one cycle. The resulting pipeline is shown below.

I-Cache Address Decode	I-Cache Array Access	I-Cache Tag Check, Instruction Decode & Register Fetch	Execute	Fast-Path D-Cache Access and Tag Check & Slow Path D-Cache Address Decode	Slow-Path D-Cache Array Access	Slow-Path D-Cache Tag Check	Write-Back

The benefit of this approach is that a load instruction that hits in the fast-path cache will now have its value available at the end of the Slow-Path D-Cache Address Decode stage, whereas before it wasn't available until the end of the Slow-Path D-Cache Array Access stage. We can re-examine the instruction sequence from the solution to Problem M5.1.E:

```
LW R1, 0(R2)
ADD R3, R1, R2
```

If the fast-path cache always hits, the load delay will only be 1 cycle, which saves 1 cycle over the scheme from Problem M5.1.F and 2 cycles over the scheme from Problem M5.1.E. This scheme differs from having a single D-cache in the original 5-stage pipeline because the fast-path cache will be very small in order to avoid lengthening the cycle time. The idea is to keep the low miss rate of a large primary cache, the shorter cycle time available with a pipelined cache, and the single-cycle load delay associated with an unpipelined cache.

Problem M5.2: Write Buffer for Data Cache

Problem M5.2.A

Little's law: T = 1 / (20*2) = 1 / 40
L = 100
Therefore, N = T*L = 2.5 (entries on average)

Problem M5.2.B

Stall = (Popcount(Wbuf) >= (N – 2)) . (IR == Store)

If you assume that you can figure out the number of store instructions in flight by decoding the IR in each stage, you will be able to eliminate (-2) in the answer above.

Problem M5.2.C

Stall = (Popcount(WBuf) + Popcount(Pipeline) >N)

If you assume in the previous question that you can figure out the number of store instructions in flight by decoding the IR in each stage, you may conclude the optimization does not make any change.