CS 61C — Hazards, Data-Level Parallelism

Summer 2024 — Discussion 9

1 Hazards Precheck

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 Without forwarding or double-pumping, data hazards will usually result in 3 stalls.

True. The next instruction must wait for the previous instruction to finish EX, MEM, and WB, before it can begin its EX.

1.2 All data hazards can be resolved with forwarding.

False. Hazards following `lw` cannot be fully resolved with forwarding because the output is not known until after the MEM stage, making a stall necessary.

1.3 Stalling is the only way to resolve control hazards.

False. While one way to resolve control hazards is to stall until the result of the branch instruction is determined, there are other more advanced techniques such as branch prediction, which predicts which path the branch will take and flushes the pipeline if the prediction is wrong.

2 Hazards

One of the costs of pipelining is that it introduces pipeline hazards. Hazards, generally, are issues with something in the CPU's instruction pipeline that could cause the next instruction to execute incorrectly.

The 5-stage pipelined CPU introduces three types: structural hazards (hardware not sufficient), data hazards (using wrong values in computation), and control hazards (executing the wrong instruction).

Structural Hazards

Structural hazards occur when more than one instruction needs to use the same datapath resource at the same time. In the standard 5-stage pipeline, **there aren't structural hazards**, unless there are active changes to the pipeline. The structural hazards that could exist are prevented by RV32I's hardware requirements.

There are two main causes of structural hazards:

- **Register File:** The register file is accessed both during ID, when it is read to decode the instruction, and the corresponding register values; and during WB, when it is written to in the `rd` register. If the RegFile only had one port, then

it wouldn't work since we have one instruction being decoded and another writing back.

  - We resolve this by having separate read and write ports. However, this only works if the read/written registers are different.

- **Main Memory:** Main memory is accessed for both instructions and data. If memory could only support one read/write at a time, then instruction A going through IF and attempting to fetch an instruction from memory cannot happen at the same time as instruction B attempting to read (or write) to data portions of memory.

  - Having a separate instruction memory (abbreviated IMEM) and data memory (abbreviated DMEM) solves this hazard.

Something to remember about structural hazards is that they can always be resolved by adding more hardware.

Data Hazards

Data hazards are caused by data dependencies between instructions. In CS 61C, where we always assume that instructions go through the processor in order, we see data hazards when an instruction **reads** a register before a previous instruction has finished **writing** to that register.

There are three types of data hazards:

- **EX-ID:** this hazard exists because the output from the execute stage is not written back to the RegFile until the writeback stage, yet can be requested by the subsequent instruction in the decode stage.

- **MEM-ID:** this hazard exists because the output from the memory access stage is not written back to the RegFile until the writeback stage, but can be requested from the decode stage, just as in EX-ID.

- **WB-ID** To account for reads and writes to the same register, processors usually write to the register during the first half of the clock cycle, and read from it during in the second half. This is an implementation of the idea of **double pumping**, which is when data is transferred along data buses at double the rate, by utilising both the rising and falling clock edges in a clock cycle.

Solving Data Hazards

For all questions, assume **no branch prediction or double-pumping (i.e. write-then-read in one cycle for RegFile)**.

Forwarding

Most data hazards can be resolved by forwarding, which is when the result of the EX or MEM stage is sent to the EX stage for a following instruction to use.

Side note: how is forwarding (EX to EX or MEM to EX) implemented in hardware? We add 2 wires: one from the beginning of the MEM stage for the output of the

ALU and one from the beginning of the WB stage. Both of these wires will connect to the A/B muxes in the EX stage.

2.1 Look for data hazards in the code below, and figure out how forwarding could be used to solve them.

Instruction	C1	C2	C3	C4	C5	C6	C7
1. `addi t0, a0, -1`	IF	ID	EX	MEM	WB		
2. `and s2, t0, a0`		IF	ID	EX	MEM	WB	
3. `sltiu a0, t0, 5`			IF	ID	EX	MEM	WB

There are two data hazards, between instructions 1 and 2, and between instructions 1 and 3. The first could be resolved by forwarding the ALU output in the MEM stage in C3 to the beginning of the EX stage in C4, and the second could be resolved by forwarding the ALU output in the WB stage in C4 to the beginning of the EX stage in C5.

2.2 Imagine you are a hardware designer working on a CPU's forwarding control logic. How many instructions after the addi instruction could be affected by data hazards created by this addi instruction?

Three instructions. For example, with the addi instruction, any instruction that uses t0 that has its ID stage in C3, C4, or C5 will not have the result of addi's writeback in C5. If, however, we are allowed to assume double-pumping (write-then-read to registers), then it would only affect two instructions since the ID stage of instruction 4 would be allowed to line up with the WB stage of intruction 1.

Stalls

2.3 Look for data hazards in the code below. One of them cannot be solved with forwarding—why? What can we do to solve this hazard?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8
1. `addi s0, s0, 1`	IF	ID	EX	MEM	WB			
2. `addi t0, t0, 4`		IF	ID	EX	MEM	WB		
3. `lw t1, 0(t0)`			IF	ID	EX	MEM	WB	
4. `add t2, t1, x0`				IF	ID	EX	MEM	WB

There are two data hazards in the code. The first hazard is between instructions 2 and 3, from t0, and the second is between instructions 3 and 4, from t1. The hazard between instructions 2 and 3 can be resolved with forwarding, but the hazard between instructions 3 and 4 cannot be resolved with forwarding. This is because even with forwarding, instruction 4 needs the result of instruction 3 at the beginning of C6, and it won't be ready until the end of C6.

We can fix this by stalling: insert a nop (no-operation) between instructions 3 and 4.

2.4 Say you are the compiler and can re-order instructions to minimize data hazards while guaranteeing the same output. How can you fix the code above?

Reorder the instructions 2-3-1-4, because instruction 1 has no dependencies.

Detecting Data Hazards

Say we have the $rs1$, $rs2$, $RegWEn$, and rd signals for two instructions (instruction n and instruction $n+1$) and we wish to determine if a data hazard exists across the instructions. We can simply check to see if the rd for instruction n matches either $rs1$ or $rs2$ of instruction $n+1$, indicating that such a hazard exists (why does this make sense?).

We could then use our hazard detection to determine which forwarding paths/number of stalls (if any) are necessary to take to ensure proper instruction execution. In pseudo-code, part of this could look something like the following:

```
if (rs1(n + 1) == rd(n) && RegWen(n) == 1) {
    set ASel for (n + 1) to forward ALU output from n
}
if (rs2(n + 1) == rd(n) && RegWen(n) == 1) {
    set BSel for (n + 1) to forward ALU output from n
}
```

Control Hazards

Control hazards are caused by **jump and branch instructions**, because for all jumps and some branches, the next PC is not PC + 4, but the result of the ALU available after the EX stage. We could stall the pipeline for control hazards, but this decreases performance.

2.5 Besides stalling, what can we do to resolve control hazards?

We can try to predict which way branches will go, and if this prediction is incorrect, flush the pipeline and continue with the correct instruction. (The most naive prediction method is to simply predict that branches are always not taken, which is effectively the same as not having any branch prediction at all.)

Extra for Experience

2.6 Given the RISC-V code above and a pipelined CPU with no forwarding, how many hazards would there be? What types are each hazard? Consider all possible hazards between all instructions.

How many stalls would there need to be in order to fix the data hazard(s), if the RegFile supports double-pumping (i.e. write-then-read)? What about the control hazard(s), if we use branch prediction?

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9
1. `sub t1, s0, s1`	IF	ID	EX	MEM	WB				
2. `or s0, t0, t1`		IF	ID	EX	MEM	WB			
3. `sw s1, 100(s0)`			IF	ID	EX	MEM	WB		
4. `bgeu s0, s2, loop`				IF	ID	EX	MEM	WB	
5. `add t2, x0, x0`					IF	ID	EX	MEM	WB

There are four hazards: between instructions 1 and 2 (data hazard from t1), instructions 2 and 3 (data hazard from s0), instructions 2 and 4 (from s0), and instructions 4 and 5 (a control hazard).

Assuming that we can read and write to the RegFile on the same cycle, two stalls are needed between instructions 1 and 2 (WB→ID), and two stalls are needed between instructions 2 and 3 (WB→ID). For the control hazard, if we predicted correctly, then no stalls are needed, but if we predicted incorrectly, then we need 3 stalls while flushing the pipeline (MEM→1 cycle before IF). We don't need to stall for the hazard between 2 and 4 because stalling for instruction 3 already handles that.

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9
1. `sub t1, s0, s1`	IF	ID	EX	MEM	WB				
2. `or -> nop`		IF	ID	EX	MEM	WB			
2. `or s0, t0, t1`		IF	ID	EX	MEM	WB			
3. `sw s1, 100(s0)`			IF	ID	EX	MEM	WB		
4. `bgeu s0, s2, loop`				IF	ID	EX	MEM	WB	
5. `add t2, x0, x0`					IF	ID	EX	MEM	WB

3 DLP Precheck

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

3.1 SIMD is ideal for flow-control heavy tasks (i.e. tasks with many branches/if statements).

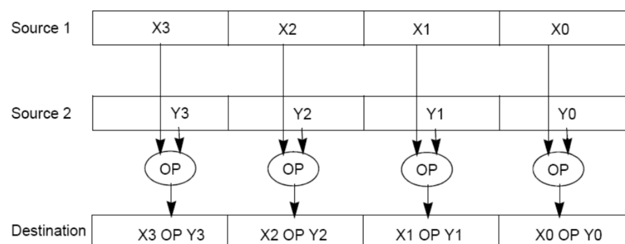
False. Data-level parallelism really shines through when we need to repeatedly perform the same operation on a large amount of data. Flow control statements disrupt the continuous flow of computation, which makes programs with them hard to take advantage of SIMD.

3.2 Intel's SIMD intrinsic instructions invoke large registers available on the architecture in order to perform one operation on multiple values at once.

True. For example, we can pack four 32-bit integers in a single 128-bit register and perform the same arithmetic operation on all four integers in one go, using an instruction such as `__m128i _mm_add_epi32(__m128i a, __m128i b)`.

4 Data-Level Parallelism

The idea central to data level parallelism is vectorized calculation: applying operations to multiple items (which are part of a single vector) at the same time.



Some machines with x86 architectures have special, wider registers, that can hold 128, 256, or even 512 bits. Intel intrinsics (Intel proprietary technology) allow us to

use these wider registers to harness the power of DLP in C code.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. The type __m128i is used when these registers hold 4 ints, 8 shorts or 16 chars; __m128d is used for 2 double precision floats, and __m128 is used for 4 single precision floats. Where you see "epiXX", epi stands for **e**xtended **p**acked **i**nteger, and XX is the number of bits in the integer. "epi32" for example indicates that we are treating the 128-bit register as a pack of 4 32-bit integers.

- `__m128i _mm_set1_epi32(int i)`:
  Set the four signed 32-bit integers within the vector to `i`.
- `__m128i _mm_loadu_si128( __m128i *p)`:
  Load the 4 successive ints pointed to by `p` into a 128-bit vector.
- `__m128i _mm_mullo_epi32(__m128i a, __m128i b)`:
  Return vector $(a_0 \cdot b_0, a_1 \cdot b_1, a_2 \cdot b_2, a_3 \cdot b_3)$.
- `__m128i _mm_add_epi32(__m128i a, __m128i b)`:
  Return vector $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
- `void _mm_storeu_si128( __m128i *p, __m128i a)`:
  Store 128-bit vector `a` at pointer `p`.
- `__m128i _mm_and_si128(__m128i a, __m128i b)`:
  Perform a bitwise AND of 128 bits in a and b, and return the result.
- `__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)`:
  The ith element of the return vector will be set to 0xFFFFFFFF if the ith elements of a and b are equal, otherwise it'll be set to 0.

4.1 SIMD-ize the following function, which returns the product of all of the elements in an array.

```
static int product_naive(int n, int *a) {
    int product = 1;
    for (int i = 0; i < n; i++) {
        product *= a[i];
    }
    return product;
}
```

Things to think about: When iterating through a loop and grabbing elements 4 at a time, how should we update our index for the next iteration? What if our array has a length that isn't a multiple of 4? What can we do to handle this tail case?

```
static int product_vectorized(int n, int *a) {
    int result[4];
    __m128i prod_v = __mm_set1_epi32(1);
    for (int i = 0; i < n/4 * 4; i += 4) { // Vectorized loop
        prod_v = __mm_mullo_epi32(prod_v, __mm_loadu_si128((__m128i *) (a + i)));
    }
    _mm_storeu_si128((__m128i *) result, prod_v);
    for (int i = n/4 * 4; i < n; i++) { // Handle tail case
        result[0] *= a[i];
```

```
    }
    return result[0] * result[1] * result[2] * result[3];
}
```