

1 Pipelining Performance

In this question, we will be calculating the performance of the following RISC-V code:

1. `addi t0, a0, -4`
2. `sw t0, 0(a0)`
3. `slli a1, t1, 4`
4. `slli a2, t2, 4`

- 1.1 Calculate the number of cycles-per-instruction (CPI) given we run the code on a single-cycle datapath.

For single-cycle datapath, CPI = 1 cycle per instruction because each instruction finishes in a single cycle.

- 1.2 Use the Iron Law to calculate the runtime of our program for our single-cycle datapath given $t_{\text{clk-period}} = 400 \text{ ns}$.

Hint: $\frac{\text{time}}{\text{program}} = \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}}$

$$\begin{aligned} \frac{\text{time}}{\text{program}} &= \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}} \\ &= \frac{4 \text{ instructions}}{\text{program}} \times \frac{1 \text{ cycle}}{\text{instruction}} \times \frac{400 \text{ ns}}{\text{cycle}} \\ &= 1600 \text{ ns} \end{aligned}$$

- 1.3 Assume we have a 5-stage pipeline with no data forwarding, but our register file supports same-cycle write-then-read. Fill out the pipeline diagram below, inserting NOPs where appropriate:

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
1. <code>addi t0, a0, -4</code>	IF	ID	EX	MEM	WB					
<code>nop</code>		IF	X	X	X	X				
<code>nop</code>			IF	X	X	X	X			
2. <code>sw t0, 0(a0)</code>				IF	ID	EX	MEM	WB		
3. <code>slli a1, t1, 4</code>					IF	ID	EX	MEM	WB	
4. <code>slli a2, t2, 4</code>						IF	ID	EX	MEM	WB

There is a data hazard between instructions 1 and 2. Because we do not have data forwarding, we have to stall after the first instruction until `t0` gets updated in the register file in C5. There exist no other hazards.

- 1.4 What is the CPI for our 5-stage pipeline calculated from the diagram above?

We measure CPI starting from the cycle that first instruction finishes execution up until when the last instruction finishes. Our range of cycles are then C5-C10 so we have a cycle count of 6. This gives us a CPI of $\frac{\text{cycles}}{\text{instructions}} = \frac{6}{4} \approx 1.5$.

- 1.5 Use the Iron Law to calculate the runtime of our program for our 5-stage pipelined datapath given $t_{\text{clk-period}} = 200$ ns and approximating $\text{CPI} \approx 2$.

$$\begin{aligned}
 \frac{\text{time}}{\text{program}} &= \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}} \\
 &= \frac{4 \text{ instructions}}{\text{program}} \times 1.67 \frac{\text{cycles}}{\text{instruction}} \times \frac{200 \text{ ns}}{\text{cycle}} \\
 &\approx 4 \times 2 \times 200 \text{ ns} \\
 &= 1600 \text{ ns}
 \end{aligned}$$

- 1.6 If we modify our code to be:

1. `addi t0, a0, -4`
2. `slli a1, t1, 4`
3. `slli a2, t2, 4`
4. `sw t0, 0(a0)`

Fill out the pipeline diagram assuming we run on the same 5-stage pipeline as above. Calculate the CPI and the execution time of our program ($t_{\text{clk-period}} = 200$ ns).

Instruction	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
1. <code>addi t0, a0, -4</code>	IF	ID	EX	MEM	WB					
2. <code>slli a1, t1, 4</code>		IF	ID	EX	MEM	WB				
3. <code>slli a2, t2, 4</code>			IF	ID	EX	MEM	WB			
4. <code>sw t0, 0(a0)</code>				IF	ID	EX	MEM	WB		

We measure CPI starting from the end of instruction 1 which are cycles C5-C8 = 4 cycles. Thus, $\text{CPI} = \frac{4}{4} = 1$.

$$\begin{aligned}
 \frac{\text{time}}{\text{program}} &= \frac{\text{instructions}}{\text{program}} \times \frac{\text{cycles}}{\text{instruction}} \times \frac{\text{time}}{\text{cycle}} \\
 &= \frac{4 \text{ instructions}}{\text{program}} \times 1 \frac{\text{cycles}}{\text{instruction}} \times \frac{200 \text{ ns}}{\text{cycle}} \\
 &\approx 4 \times 1 \times 200 \text{ ns} \\
 &= 800 \text{ ns}
 \end{aligned}$$

- 1.7 Why did reordering the instructions increase the performance of our code? And more generally, what motivates us to study data forwarding, branch prediction, etc.?

Reordering the instructions eliminated the hazards and stalls in the code. Techniques such as data forwarding and branch prediction also seek to eliminate stalls in our pipeline which can decrease CPI and yield increased performance benefits.

Note that compilers can reorder instructions in software, and some hardware architectures can reorder instructions on-the-fly (called “out-of-order execution”)! Take CS152 to learn more.

2 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.

Below is a small selection of the available Intel intrinsic instructions. All of them perform operations using 128-bit registers. When we use an instruction with “`epi32`”, we treat the register as a pack of 4 32-bit integers.

Function	Description
<code>__m128i</code>	Datatype for a 128-bit vector.

Function	Description
<code>__m128i _mm_set1_epi32(int i)</code>	Creates a vector with four signed 32-bit integers where every element is equal to <i>i</i> .
<code>__m128i _mm_loadu_si128(__m128i *p)</code>	Load 4 consecutive integers at memory address <i>p</i> into a 128-bit vector.
<code>void _mm_storeu_si128(__m128i *p, __m128i a)</code>	Stores vector <i>a</i> into memory address <i>p</i>
<code>__m128i _mm_add_epi32(__m128i a, __m128i b)</code>	Returns a vector = $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
<code>__m128i _mm_mullo_epi32(__m128i a, __m128i b)</code>	Returns a vector = $(a_0 \times b_0, a_1 \times b_1, a_2 \times b_2, a_3 \times b_3)$.
<code>__m128i _mm_and_si128(__m128i a, __m128i b)</code>	Perform a bitwise AND of 128 bits in <i>a</i> and <i>b</i> , and return the result.
<code>__m128i _mm_cmpeq_epi32(__m128i a, __m128i b)</code>	The <i>i</i> th element of the return vector will be set to 0xFFFFFFFF if the <i>i</i> th elements of <i>a</i> and <i>b</i> are equal, otherwise it'll be set to 0.

A longer list of Intel intrinsics can be found in the precheck worksheet!

2.1 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);

    // Vectorized Loop
    for (int i = 0; i < n/4 * 4; i += 4) {
        prod_v = _mm_mullo_epi32(
            prod_v,
            _mm_loadu_si128((__m128i *) (a + i))
        );
    }

    _mm_storeu_si128((__m128i *) result, prod_v);

    // Handle tail case
    for (int i = n/4 * 4; i < n; i++) {
        result[0] *= a[i];
    }

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

```

2.2 Recall that Amdahl's Law can be used to measure the maximum speedup that can be obtained through parallelization:

$$\text{Speedup} = \frac{1}{(1 - \text{frac}_{\text{optimized}}) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}}$$

Assume that we measure `product_vectorized` to be 4x faster than its scalar version. We measure that 20% of our overall program is run serially while 80% is run in parallel. Calculate the performance increase gained from parallelizing our code.

$$\begin{aligned}
 \text{Speedup} &= \frac{1}{(1 - \text{frac}_{\text{optimized}}) + \frac{\text{frac}_{\text{optimized}}}{\text{factor}_{\text{improvement}}}} \\
 &= \frac{1}{(1 - 0.80) + \frac{0.80}{4}} \\
 &= \frac{1}{0.20 + 0.20} \\
 &= \frac{1}{0.4} \\
 &= 2.5x \text{ performance increase}
 \end{aligned}$$

- 2.3 Now we want to write a similar function that will only *add* elements given a certain condition. For example:

```
static int add20_naive(int n, int *a) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        if (a[i] == 20) {
            sum += a[i];
        }
    }
    return sum;
}
```

Fill in the function to use a vector mask to add elements only if they are equal to 20:

```
static int add20_vectorized(int n, int *a) {
    int result[4];

    // Fill sum_v with zeros
    __m128i sum_v = _____;

    int32_t twenty[4] = {20, 20, 20, 20};
    __m128i vec_twenty = _____;

    // Vectorized Loop
    for (int i = 0; i < _____; i += _____) {
        // Load array into vector
        __m128i vec_arr = _____;

        // Create vector mask
        __m128i vec_mask = _____;

        sum_v = _____;
    }

    _mm_storeu_si128(_____);

    // Tail case...
    /* Omitted */
}
```

```

static int add20_vectorized(int n, int *a) {
    int result[4];
    __m128i sum_v = _mm_set1_epi32(0);

    int32_t twenty[4] = {20, 20, 20, 20};
    __m128i vec_twenty = _mm_loadu_si128((__m128i *) twenty);

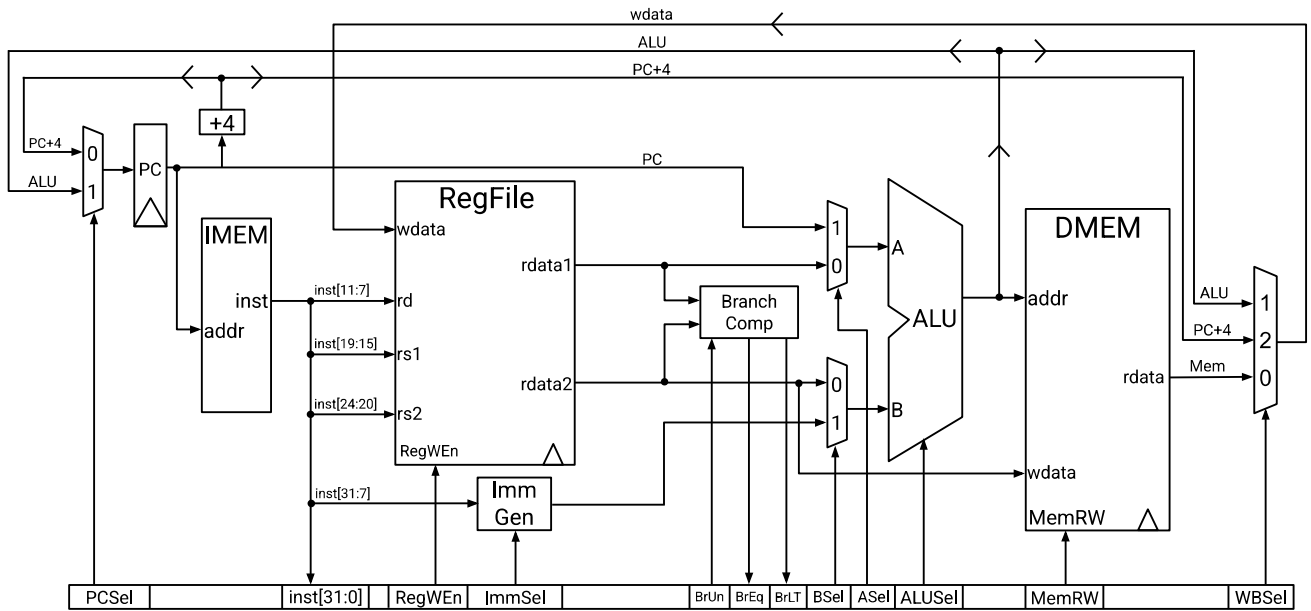
    // Vectorized Loop
    for (int i = 0; i < n/4 * 4; i += 4) {
        __m128i vec_arr = _mm_loadu_si128((__m128i *) (a + i));
        __m128i vec_mask = _mm_cmpeq_epi32(vec_arr, vec_twenty);
        sum_v = _mm_add_epi32(
            sum_v,
            _mm_and_si128(vec_arr, vec_mask)
        );
    }

    _mm_storeu_si128((__m128i *) result, sum_v);

    // Tail case...
    /* Omitted */
}

```

Single-Cycle Datapath Diagram



5-Stage Datapath Diagram

