CS61C Spring 2025

Hardware Performance, DLP

Discussion 10

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.

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

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

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. addi t0, a0, -4`	IF	ID	EX	MEM	WB					

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

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 CPI ≈ 2.

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. addi t0, a0, -4`	IF	ID	EX	MEM	WB					

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.?

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**
`__m128i`	Datatype for a 128-bit vector.
`__m128i _mm_set1_epi32(int i)`	Creates a vector with four signed 32-bit integers where every element is equal to `i`.
`__m128i _mm_loadu_si128(__m128i *p)`	Load 4 consecutive integers at memory address `p` into a 128-bit vector.
`void _mm_storeu_si128(__m128i *p, __m128i a)`	Stores vector `a` into memory address `p`
`__m128i _mm_add_epi32(__m128i a, __m128i b)`	Returns a vector = $(a_0 + b_0, a_1 + b_1, a_2 + b_2, a_3 + b_3)$
`__m128i _mm_mullo_epi32(__m128i a, __m128i b)`	Returns a vector = $(a_0 \times b_0, a_1 \times b_1, a_2 \times b_2, a_3 \times b_3)$.
`__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.

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 = ________________________________________;

    // Vectorized Loop
    for (int i = 0; i < _______________; i += _________________) {

        prod_v = ____________________________________________________;
    }

    _mm_storeu_si128(_________________________, ______________________);

    // Handle tail case
    for (int i = ____________________; i < _____________; i++) {

        result[0] *= ________________________;
    }

    return ____________________________________________________;
}
```

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

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

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.

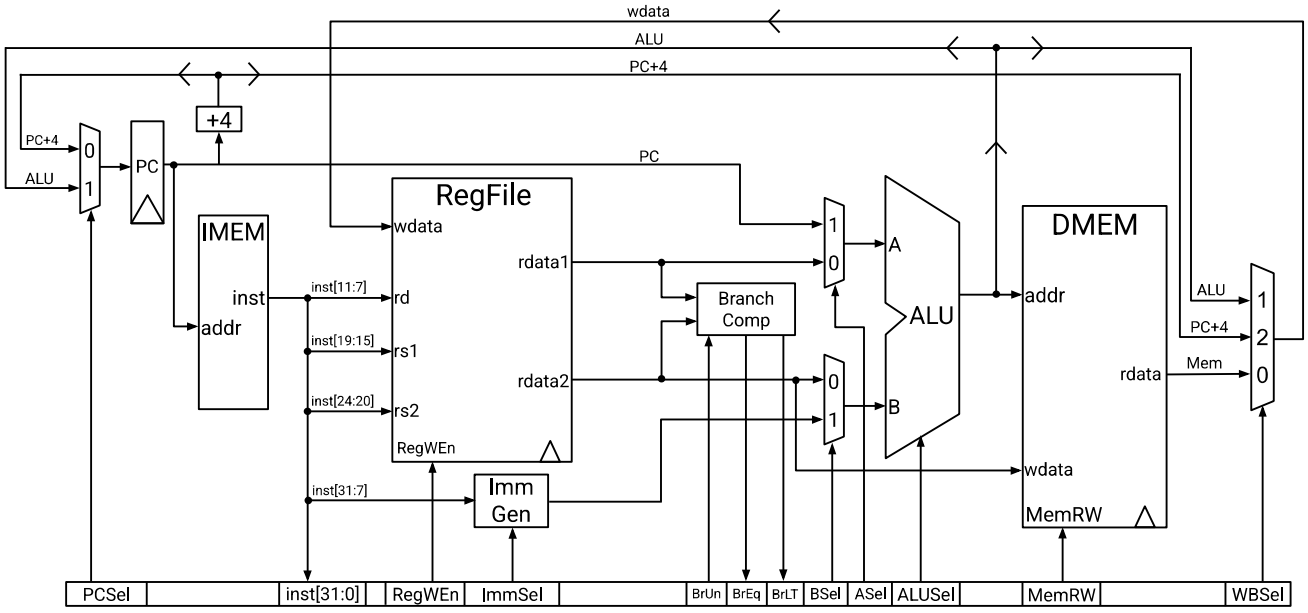
2.3 Now we want to write a similar function that will only *add* elements given a certain condition. For example:

```
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 */
}
```

Single-Cycle Datapath Diagram



5-Stage Datapath Diagram