CS 61C Spring 2024

FSM, Parallelism I

Discussion 9

1 Pre-Check

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 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.

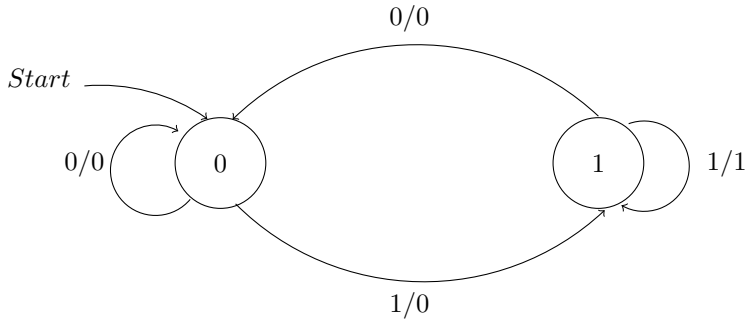
1.2 Intel's SIMD intrinsic instructions invoke large registers available on the architecture in order to perform one operation on multiple values at once.

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)`.

2 Finite State Machines

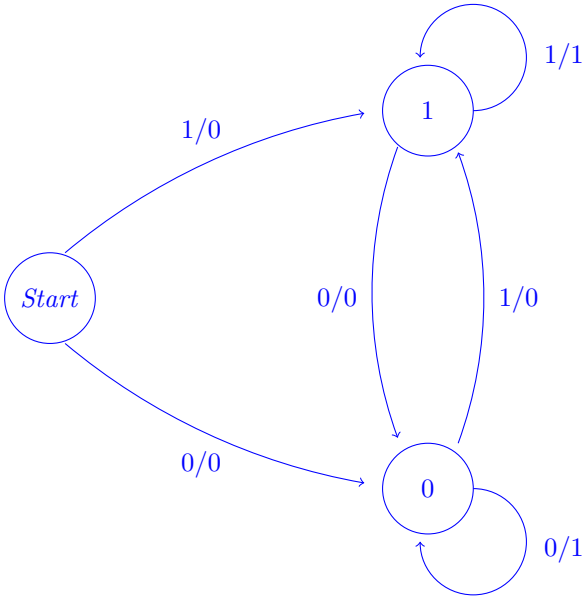
Automatons are machines that receive input and use various states to produce output. A finite state machine is a type of simple automaton where the next state and output depend only on the current state and input. Each state is represented by a circle, and every proper finite state machine has a starting state, signified either with the label "Start" or a single arrow leading into it. Each transition between states is labeled [input]/[output].

2.1 What pattern in a bitstring does the FSM below detect? What would it output for the input bitstring "011001001110"?

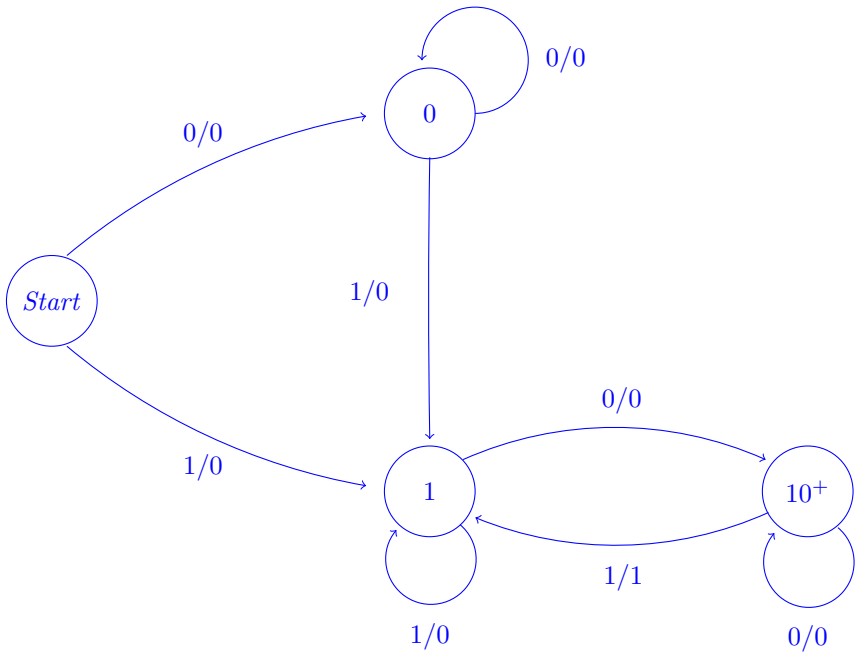


The FSM outputs a 1 if it detects the pattern "11".
The FSM would output "001000000110"

2.2 Fill in the following FSM for outputting a 1 whenever we have two repeating bits as the most recent bits, and a 0 otherwise. You may not need all states.

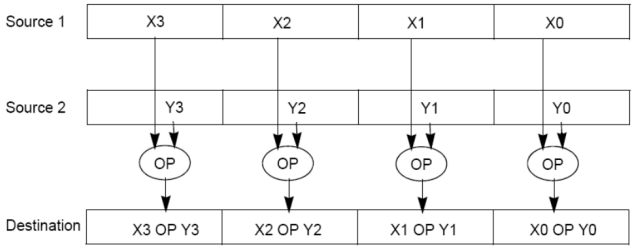


2.3 Draw an FSM that will output a 1 if it recognizes the regex pattern {10+1}. (That is, if the input forms a pattern of a 1, followed by one or more 0s, followed by a 1.)



3 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.

3.1 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];
}
```

4 Amdahl's Law

In attempting to parallelize a program, the overall performance speedup will always be limited by the fraction of the program that cannot be sped up. This overall speedup can be formulated by Amdahl's Law, which states that

$$\textbf{Speedup} = \frac{1}{(1-F) + \frac{F}{S}}$$

Speedup refers to the theoretical speedup of the program compared to its naive implementation. Note that **Speedup** > 1 since we're making our program faster than the original.

F refers to the fraction of the program that can be optimized;

S is the speedup factor for how much that portion of the program can be optimized by, where (**S** > 1)

4.1 Derive Amdahl's Law using the ratio: **Speedup** $= t_{\text{naive}}/t_{\text{optimized}}$

First, we can split the overall time a program takes into the time it takes for the part of the program that can be optimized and the rest of it. Letting F represent the fraction that can be sped up, we have:

$t_{\text{naive}} = F(t_{\text{naive}}) + (1-F)t_{\text{naive}}$

Then, we can implement the optimization, known as the speedup factor S into our equation by dividing the optimizable portion to get:

$t_{\text{optimized}} = \frac{F(t_{\text{naive}})}{S} + (1-F)t_{\text{naive}}$

Solving for the ratio **Speedup** $= t_{\text{naive}}/t_{\text{optimized}}$ leads to Amdahl's Law.

4.2 Assuming we have infinite threads and resources, what would our overall speedup be for a program with some fraction of our code that can be parallelized F?

With infinite scaling factor S, our total speedup will approach $\frac{1}{1-F}$. However, in reality there would be some non-zero overhead that is required to properly split up work.

4.3 You write code that will search for the phrases "Hello Sean", "Hello Jon", "Hello Dan", "Hello Man", "Bora is the Best!" in text files. With some analysis, you determine you can speed up 40% of the execution by a factor of 2 when parallelizing your code. What is the true speedup?

Using Amdahl's Law with F=0.4, S=2:

$$\frac{1}{0.6 + \frac{0.4}{2}} = \frac{1}{0.8} = 1.25$$

4.4 You run a profiling program on a different program to find out what percent of time within the program each function takes. You get the following results:

Function	% Time
`f`	30%
`g`	10%
`h`	60%

(a) Assuming that each of these functions can be parallelized by the same speedup factor, which one, if parallelized, would cause the most speedup for the entire program?

`h`

(b) What speedup would you get if you parallelized just this function with 8 threads? Assume that work is distributed evenly across threads and there is no overhead for parallelization.

$1/(0.4 + 0.6/8) \approx 2.1$