Designing a Custom Trigonometric Floating-Point Function Unit

Kavita Jain-Cocks & Amrita Mazumdar
CSEE 4823 - Advanced Logic Design

December 10, 2012

1 Pseudocode Algorithms

1.1 General Algorithm

1.1.1 Variables

Inputs:

- data_in (32-bit FP)
- mode (sine = 1, cosine = 0) (1-bit)
- res (4-bit)

Outputs:

- data_out (32-bit FP)

1.1.2 General Algorithm Pseudocode

```
count_res = 0, ans = 0, count_exp = 0;
temp = 1, shift1 = 0, shift2 = 0
while (count_res < res){
        temp = 1;
        count_exp = 0;
        while (count_exp < res){
```

```
                temp = temp*data_in; // ** fp multiply**
                count_exp ++;
        }
        if (sin){
                temp = temp*coeff[2*count_res + 1]
        }
        else {
                temp = temp*coeff[2*count_res]
        }

        if (count_res_LSB == 0){
                ans = ans + temp;
        }
        else {
                ans = ans - temp;
        }
}
data_out = ans;
```

1.1.3 Concept

- Variable *temp* stores the temporary answer to the problem
- First, *temp* is multiplied by the input *data_in* until it represents *data_in* raised to an exponent
- Then, the coefficient of the appropriate index is acquired from the ROM *coeff* and multiplied by the *temp*, which now holds the value of one number in the Maclaurin series
- The value *temp* is either added or subtracted from the final value *and* depending on whether it is even or odd.
- Note: in this conceptualization, the floating point add/subtract/multiply have been abstracted out.

1.1.4 Discussion & Optimization of General Algorithm

The implementation and optimization of our algorithm came in stages through the design process. Our first design was a straightforward implementation

of the above algorithm, which came in at 15 clock cycles per loop, and 2-3 operations per clock cycle. After a few subtle organizational optimizations (calculating values earlier and using more conditional branching to reduce states), we reduced our loop to 10 clock cycles per loop, and 3-4 operations per cycle. One reason for our incremental progress was that we did not integrate our floating point calculations into the finalized ASM yet, so we could not yet fully optimize. Our final ASM, which included all floating point calculations, has 7 clock cycles per loop, and up to seven operations within a single clock cycle.

1.2 Floating Point Algorithms

1.2.1 Addition

- check if exponents are equal
- if not, shift manitssa left and decrement larger exponent until equal
- carry-add mantissa, pass exponent down
- normalize the result

1.2.2 Subtraction

- check if exponents are equal
- if not, shift manitssa left and decrement larger exponent until equal
- carry-subtract mantissa, pass exponent down
- normalize the result

1.2.3 Multiplication

- XOR the signs
- final exponent is exp1 + exp2 - bias (bias = 127)
- combinational-multiply the mantissa
- normalize the result

1.2.4 Discussion & Optimization of FP Operations

Our first rudimentary algorithm for addition/subtraction did not accommodate the many edge-cases of floating-point addition that would result from opposing signs, differing magnitudes, etc. In order to reduce the number of clock cycles per loop as well as deal with edge cases our final implementation utilizes more complex decision trees. This allows each state to address only the specific situation that leads to it. Our final implementation involves first checking whether the numbers are of the same order of magnitude by checking the exponents, normalizing if necessary, then evaluating what type of operation to do based on the signs of the first and second term and the magnitudes. For example, if the operation were (7 + (-4)), the sign of the result would be different than if it were (4 + (-7)), and it is easiest to calculate in both cases by subtracting and taking the appropriate sign, even though it is addition.
For multiplication, not much changed from our initial algorithm to final implementation, although we struggled with understanding the appropriate normalization procedure following a carry-multiply, and eventually decided on shifting the mantissa 23 bits to result in an appropriately truncated mantissa.

2 Identifying Status Signals

- sin
- $count\_res = 0$
- $count\_res_{LSB}$
- $count\_exp = 0$
- $count\_exp < index$
- $count\_resres$
- $ans\_sign > temp\_sign$
- $ans\_sign = temp\_sign$
- $ans\_exp > temp\_exp$

- $temp\_exp > ans\_exp$
- $ans\_exp = temp\_exp$
- $ans\_mant > temp\_mant$
- ans_sign
- $carry\_out$

3 Transforming the Moore ASM to Moore FSM

In order to convert the Moore ASM into the FSM, the first step taken is to convert decision blocks into the inputs that determine arcs in the FSM. Any signal not stated explicitly in the decision block is essentially a dont care. All values displayed in the state box are outputs and any values not explicitly stated should be allowed to float. Specifically, the first step would be to recreate the states in the ASM, since there will be the same number of states in the FSM as in the ASM Then, the arcs should be drawn so that they are connected in the same way as in the ASM. Each decision block results in 2 arcs, one where the decision evaluates to 1, and one where it evaluates to 0; each goes into its respective state block. If there are two or more decision blocks linked together in a decision tree then each additional block adds in another explicitly stated variable on the arc, and an additional arc since the variable needs to be accounted for if it evaluates to 0 or 1. There should be as many arcs as there are paths from one state to another on the Moore ASM.

4 Design Experience Discussion

Our design process involved incremental growth and multiple subtle optimizations. We began by designing a basic ASM for each floating-point operation. We then designed a high-level algorithm for the general function unit, and worked to optimize the basic unit first. This allowed us to eliminate extraneous steps early on in the design process. After achieving a strong level of optimization, we integrated the floating point unit into our ASM. Here,

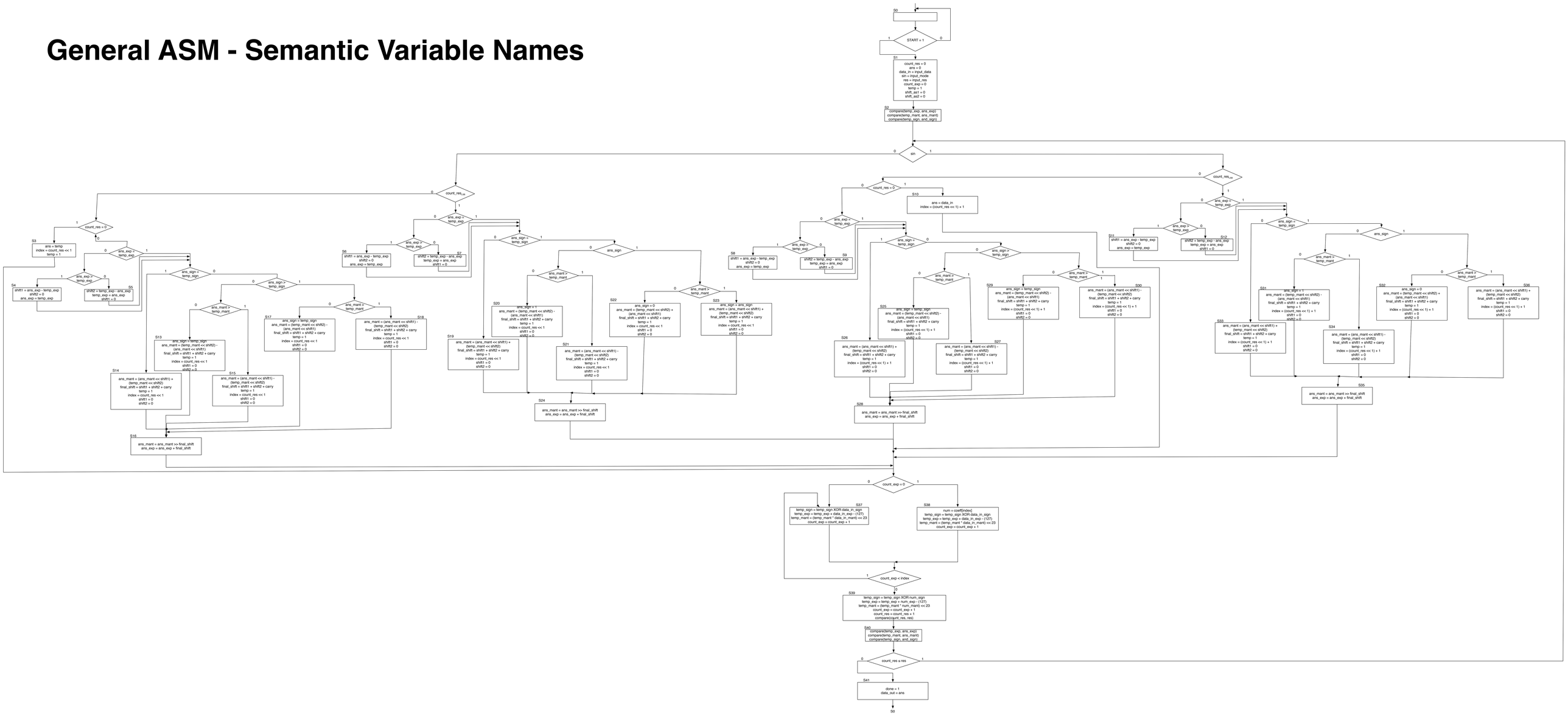
we found we had to restructure parts of our algorithm to accommodate for the additional loads. The floating point operations also required additional decision blocks, since it had its own set of nuanced optimizations in our abstraction.

One major obstacle we faced was integrating floating point into our ASM. We originally designed our algorithm with a flags to indicate floating-point operations, treating it as an operation executing in one state and not accommodating the appropriate states needed. Once we finalized our floating-point addition, subtraction, and multiplication, we discovered it would significantly increase the number of clock cycles used in each inner-loop, and had to find innovative optimizations to bring our ASM into the right threshold again. We were able to solve this problem by using complex decision trees to isolate corner cases and handle numerous different operations with few explicit states.

Another challenge was maintaining a low number of ALU's in our micro architecture. Part of this could have been a consequence of our initial abstraction of floating-point operations; had we considered the unit as part of our microarchitecture from the start, we may have found it easier to keep the number of ALU's small. In our current microarchitecture we reuse the same ALU units in different portions of our ASM and so even though they are redrawn for clarity we were able to keep the number used down to no more than 4 of each type.

Our final algorithm has 7 clock cycles per inner loop, with as few as 2 and as many as 7 steady-state operations per clock cycle. This level of performance does not account for a single loop inside the main loop, where we raise the input data *data_in* to an exponent power.

General ASM - Semantic Variable Names



Moore-style ASM - Microarchitecture Variable Names