CSEE W4823x
Prof. Steven Nowick

Handout 34
November 20, 2016

Project #2: An Introduction to RTL Design

Designing a Custom Floating-Point Accelerator for Scientific Applications

This homework is due on **Monday, December 12 (last day of classes), at 4pm** (submission details to be announced soon).

In this problem, you will design and optimize an entire subsystem, top down, using an RTL design flow. In particular, you will produce a useful real-world hardware accelerator: a *custom scientific function unit,* which computes three useful operations (*sine, cosine, natural exponential*), with user-specified precision.

The unit operates on IEEE 754 single-precision floating-point (SPFP) numbers. It takes a single data operand as input, a specified operation (sine, cosine, natural exponent) and a desired precision. It then calculates the function, and outputs the result as a single data output. This unit serves a specialized co-processor for scientific applications, to accelerate computation of these functions.

This is a *written problem only.* You are not required to do simulation or VHDL for this problem.

Introduction. This homework is an introduction to top-level digital system design. In this problem, you are given a verbal description of a custom chip (ASIC, i.e. application-specific integrated circuit) or subsystem. You will then write an algorithm for its behavior in pseudo-code, transform and formalize the specification at the register-transfer level (RTL) as a generalized ASM, and then derive a complete microarchitecture, including both the datapath components and a controller (i.e. FSM). You will then work to further optimize your design.

Grading. This project is *worth 18%* of your final grade.

Note: Handout #34(d), to be released soon, includes a short additional assignment, which is an introduction to asynchronous burst-mode controllers using the Minimalist CAD tool. This asynchronous assignment is *worth 2%* of your final grade, and should take you only a few hours to complete.

Working in Groups. You are allowed to do this homework in a group-of-two. You both get the same grade. However, solo homeworks are also allowed.

Overview. In particular, you will follow the first *7 steps* of Handout #30(a)/(b).

1. Write an **algorithm** (in pseudo-code) for the system's behavior;
2. Write a formal specification for the algorithm in RTL style in the form of a **generalized ASM** (*Moore style*);
3. Do **resource allocation** of datapath components, i.e. select datapath blocks needed (including any necessary MUXes, DEMUXes, hardwiring of inputs, and optimizations, closely following the approach presented in Handout #30(a)/(b));
4. Then, **identify status signals** (outputs of datapath, which are inputs to control), and allocate any additional datapath blocks required to generate these status signals;
5. *Draw* the final **micro-architecture** of the system, showing all datapath blocks, a single block for the control, and including all wiring between blocks (external inputs/outputs, control and status signals);
6. *Derive* a **Moore controller ASM specification**, using simple (B/V-style control-only) ASM;

7. Indicate how to transform the Moore controller ASM specification into the corresponding symbolic **Moore state diagram** for the controller specification. *You do not need to draw the resulting Moore state diagram, just precisely indicate all steps needed to make this transformation* (1-2 paragraphs).

That is, you will closely follow Steps #1-7 of Handout #30(a)/(b), but applied to this new problem.

Note: *Modelling the Moore Machine.* Do *not* design a gate-level implementation of the Moore control ASM or state diagram. Just provide the above items.

Ground Rules and Assumptions. Use the following design guidelines. Any further updates will be announced on the class web page or in class.

- **Datapath Blocks.** Handouts #32-33 provide you with the library of components which you are allowed to use in your implementation. Try to use *only* datapath blocks that are given in these handouts. *If you have a very compelling reason to use other blocks, I may permit it; but you must justify to me in advance for permission.*

  You can feel free to generalize any smaller block (e.g. 4- or 8-bit) up to 64 bits, or conversely to take a 32- or 64-bit library block and assume a smaller one. For example, you can allow 32- or 64-bit shift registers, counters, etc. But be sure to add appropriate number of additional control signals, to scale to the larger designs. In general, *you are not allowed to scale any component beyond 64-bits.*

- **ROMs.** For the given problem, you will use 1 small ROM to store coefficients. Details are provided later in this handout.

- **RAMs.** For the given problem, you should not use any RAMs.

- **Barrel shifters/rotators.** These combinational components can be quite useful. If you need them, see description in the combinational handout (#32).

- **Register Files vs. Separate Registers.** In general, since you are designing a custom ASIC, it is convenient to use a set of independent individual registers. This allows you the option to write or read arbitrary numbers of them in the same clock cycle, hence support more parallelism and complex and varying operations. However, an alternative more structured use of registers is also allowed: using a register file (see Handout #33).

  Register files allow indexed access, which may simplify some designs, but limit your access to them. In particular, you may only use a register file with single write port and dual read ports, i.e. which can perform at most 1 write and 2 reads per cycle. You can also use a combination of both free registers and a register file. However, you may not need any register files, and are not required to use them.

- **Combinational Integer Multipliers.** You are allowed to use combinational integer multipliers, as presented in the class. I.e. you can assume either the unoptimized or optimized array multipliers of Handout #17. You may assume any size multiplier and multiplicand up to 64 bits. You may assume that the multiplication always completes in less than 1 clock cycle.

- **Input and Output Bus Operations: Timing Assumptions.**

  Bus Reads/Writes. When reading a value from an input bus, follow the same strategy as in Handout #30(a)/(b): do *not* do any operations directly using an operand on the input bus, it must be written to a register or other storage unit first. The reading from the input bus and storing result is assumed to take an entire clock cycle; *no other operations can be performed on the input during that cycle.* Likewise, when writing a value to an output bus, assume that this write operation takes an entire clock cycle; *no other operations can be performed on the output during that cycle.* That is, do not concatenate or combine a combinational operation (e.g. to generate a result) which is followed by placing its result on the output bus, all in the same clock cycle.

ROM Reads. Similar to buses, assume the ROM takes an entire cycle to complete a read operation. Your design cannot read a ROM result and use it as an operand in a function unit in the same cycle. (See details in the project description below.)

Input Bus Validity. How long should data be assumed to be valid on the input bus? In Handout #30(a)/(b), we assume it is valid in the same cycle as "Start" is asserted, and remains valid for 1 extra cycle (enough time for it to be stored by the datapath). Use the same assumption for your solution; if you have compelling reasons to change this assumption, you must get permission of the instructor.

- **Number of Operations Per Clock Cycle.** You must use reasonable assumptions as to the clock speed. For example, you can assume that every basic combinational function block (Handout #32) completes in less than 1 clock cycle, such as adders, combinational multipliers, barrel shifters, comparators, etc.

  In most cases, *do not concatenate two large combinational blocks within a given clock cycle.* For example, you cannot have a multiplication followed by an addition all within the same clock cycle; the multiplication should be in one cycle and the addition in the next clock cycle. Otherwise, you would be requiring a slow clock rate.

  However, there are *useful exceptions to this rule.* Small components like MUXes, DEMUXes, etc. should be considered as having very little delay, so you *can* have a MUX concatenated with a larger block, and assume that together they always compute within the same clock cycle.

  Also, you *can* have a *barrel shifter or other combinational shifter,* attached to inputs or outputs of another combinational block (adder, multiplier, ALU), where the combined shift + operation fits into one clock cycle. This is a common combined structure used in many processors' "EX" stages, for example. Likewise, a pair of fast combinational operations such as "decode" followed by "add" (or a logical bitwise operation) are also allowed in the same clock cycle. Use good judgment, and follow the basic types of decisions you see in Handout #30(a)/(b).

  The use of two consecutive or serial combinational operations in the same cycle is called **chaining.** The resulting harware is a combined structure with one combinational block feeding as input to the next block, *together forming a single custom unit.* You must explicitly designate such a combined unit in your "datapath allocation" step, if you use chaining.

  Ask the instructor for permission on cases of chaining that you want to use. In general, you should largely *avoid chaining* except for rare simple cases.

  Notating legal chaining operations in your generalized RTL specification: For cases where chaining is allowed, such as the above example of add-followed-by-shift, you *must notate* the combined operation in a single RTL statement in your specification, so it can be translated explicitly to the RTL implementation. For example, to note "add R2 + R3, then shift-right-by-1, then write to R5" in your RTL, you would use the notation:

  $$R5 := [(R2 + R3) >> 1],$$

  which captures the sequencing of two *combinational operations,* i.e. $R2+R3$, followed by a combinational (not sequential) shift-right-by-1 on the intermediate result of the addition, before writing the chained result to register $R5$.

- **Moore vs. Mealy Generalized ASM.** Use a Moore-style generalized ASM, as in Handout #30(a)/(b) (not Mealy).

- **Initialization, Reset.** Your design should follow the same basic assumptions as in the one presented in Handout #30(a)/(b). In particular, (a) assume a *Start* external input which activates the algorithm; (b) when done, unless stated otherwise, generate a *Done* external output for 1 clock cycle, as well as place the result on an *Output* data bus for that same one cycle and for one extra cycle.

Project Problem: a Custom FP Scientific Function Unit with Variable Precision

In this project, you will design a custom floating-point unit for mathematical functions, which serves as a *co-processor* or *accelerator* for scientific applications. The unit provides the capability of computing three widely-used functions: two trigonometric (sine, cosine) and one additional (natural exponential function, e^x).

An interesting feature of the unit is that it supports *variable precision.* In particular, the user can specify a desired precision level for the result, within a given range. This capability allows the unit to be used in applications where less or more precision is needed. The former provides higher performance (i.e. lower latency, or fewer # clock cycles per operation, or lower energy), while the latter provides greater accuracy.

Inputs and Outputs. Inputs to the unit include: (i) a 32-bit data operand, $Data_{IN}$ (in single-precision IEEE 754 floating-point format); (ii) a 3-bit control signal indicating *mode* (i.e. what operation to perform: sine, cosine, exponent); and (iii) a 4-bit control signal field indicating precision, called *res* (i.e. resolution; details are below). The operand, $Data_{IN}$ may be any legal single-precision floating-point number.

For simplicity, assume that *mode* has 3 wires ($mode_s$, $mode_c$, $mode_e$) and uses a 1-hot encoding: 100 selects sine, 010 selects cosine, and 001 selects natural exponential.

Outputs to the unit include: (i) a 32-bit data result, $Data_{OUT}$ (in single-precision IEEE 754 floating-point format).

In addition, similar to Handout #30(a)/(b), the unit has a control input *Start* to activate its operation, and a control output *Done* to indicate a valid result.

Basic Data and Timing Assumptions. The assumptions below are fairly similar to those in Handout #30(a)/(b). In particular, assume the input operand arrives on a 32-bit input bus ($Data_{IN}$), and the output result is placed on a a 32-bit input bus ($Data_{OUT}$).

Assume that inputs (i)-(iii) above arrive in the same clock cycle as the control input *Start=1* is asserted high. Input (i) (i.e. the data operand) remains valid for 2 clock cycles (allowing it time to be stored by the unit). In contrast, assume the control inputs (ii) and (iii) remain valid for only 1 clock cycle (this is sufficient time for them to be observed and used), and *Start* is asserted high for only 1 clock cycle.

Similarly, assume that output (i) (i.e. the data result) is placed on the output bus in the same cycle as the control output *Done* is asserted high. Output (i) (i.e. the data result) stays valid for 2 clock cycles (allowing it time to be stored by the external requesting processor). In contrast, assume that *Done* is asserted high for only 1 clock cycle.

Approximating Scientific Functions: Taylor and Maclaurin Series. The unit computes the sine, cosine and exponential functions based on *Taylor series expansions.* These series are widely used to approximate many useful functions: e^x, $\sqrt{1+x}$, $log(1-x)$, and all common trigonometric functions. Special-case versions of these expansions, which you will use for this project, are called *Maclaurin series.*

Use the following Maclaurin series expansions:

$$e^x = 1 + x + \tfrac{1}{2!}x^2 + \tfrac{1}{3!}x^3 + \ldots \qquad \text{(for all x)}$$

$$sin(x) = x - \tfrac{1}{3!}x^3 + \tfrac{1}{5!}x^5 - \tfrac{1}{7!}x^7 + \ldots \qquad \text{(for all x)}$$

$$cos(x) = 1 - \tfrac{1}{2!}x^2 + \tfrac{1}{4!}x^4 - \tfrac{1}{6!}x^6 + \ldots \qquad \text{(for all x)}$$

Supporting Variable Precision. The unit will be capable of supporting user-specified precision of results. In particular, the unit will be able to calculate *a range of terms* for each of the above Maclaurin series. The user will supply a precision or resolution, as a 4-bit control input *res* (see above). *Legal precision values for "res" are between 5 and 15,* encoded in binary. The supplied "res" value indicates the **number of terms in the series to**

calculate. Hence, the unit will be able to calculate between 5 to 15 terms of each Maclaurin series.

As an example, if the user supplies *res = 0111,* then the first 7 terms of the appropriate series will be calculated, and the result placed on the output bus.

Assumption: You can assume that the user only supplies values on "res" in the above range.

FP Operations: Which to Support. Your unit will simply calculate a number of terms in each of the above Maclaurin series. For a given calculation, the user will supply the data operand, the mode (indicating which operation to perform: sine, cosine, or natural exponential), and the precision level. As indicated above, each Maclaurin series consists *only* of FP addition/subtraction operations, and FP multiplication operations. You should just implement these two classes of operations. *Do not implement FP division!*

Maclaurin Series: Storing and Accessing Factorial Coefficients. Note that each of the two Maclaurin series above uses factorial coefficients. For the natural exponential, e^x, these *multiplicative* coefficients are:

$$\frac{1}{0!}, \frac{1}{1!}, \frac{1}{2!}, \frac{1}{3!}, \frac{1}{4!}, \ldots$$

For sine, these *multiplicative* coefficients are:

$$\frac{1}{1!}, \frac{1}{3!}, \frac{1}{5!}, \frac{1}{7!}, \ldots$$

For cosine, these *multiplicative* coefficients are:

$$\frac{1}{0!}, \frac{1}{2!}, \frac{1}{4!}, \frac{1}{6!}, \ldots$$

Note that all of the above constant multiplicative coefficients are positive numbers. Assume the division has already been performed, and each such coefficient is a *single multiplier term.*

As part of your design, assume that all of these multiplicative coefficients have been pre-computed, and are stored in a small ROM. In particular, assume a ROM, called *Coeff*, which has 32 addressable entries, from i = 0 to 31. All ROM entries are 32-bit positive values, and stored in IEEE 754 single-precision floating-point format. The i^{th} entry is:

$$Coeff[i] = \frac{1}{i!}$$

Note: As indicated earlier, assume a ROM read takes an entire clock cycle. At the end of the cycle you can write the read coefficient into a register, for further use. However, you *cannot do any data operations* using the ROM value *during* the clock cycle in which it is being read – it first must complete the entire read cycle.

Handling Special Values: Zero and Infinity. You should allow the special floating point values of "signed zero" (both positive and negative). These values should be correctly used in operations. However, you do *not* need to handle "signed infinity."

Rounding Modes. You do not need to perform advanced rounding, simply support *truncation.*

Note: For accurate results, *you must perform truncation after each floating-point operation.* You cannot do multiple operations and truncate once after all of them are complete.

Overflow and Underflow. For simplicity, you do not need to handle overflow and underflow: assume these do not happen.

Design Requirements: Maximum Number of Function Blocks To Allocate. The primary optimization you should focus on is to improve overall performance. Increased performance often comes at the expense of increased area, and you should use judgment if the increase is reasonable. *Increasing your design by a factor of 10 area is not reasonable!* A good rule of thumb, that we will generally enforce, is that *you may use of up to 4 function blocks of the same type* if you can support them in a correct algorithm with clear pseudo-code, and so that they can handle all input scenarios.

That is, your design is only allowed to use up to 4 ALU's, 4 integer multipliers, and so on. In fact, you are encouraged to use *fewer* than this number – you should not be needing so many even to meet the latency targets

listed below. Of course, the above does not restrict your use of individual registers, MUXes, DEMUXes, etc.

More Restrictions on Components and Operations. Floating point operations are *not* simple RTL operations: *you cannot assume pre-existing FP units,* since they do not appear in Handouts #32 and #33, and you cannot write RTL code with FP multiplication operations. Instead, part of the purpose of this problem is that you break down the desired complex operations into simple RTL code, using simple RTL operators, that can support the flow of FP multiplication and addition, along with other required operations.

Optimizing Your Design: Guidelines and Suggestions. *Most of your optimization should be captured in your generalized ASM specification,* i.e., optimizing the underlying algorithm specification to allow for greater parallelism, etc. This generalized ASM should still translate fairly directly to your microarchitecture. That is, you should not make a lot of hand optimizations on the micro-architecture, beyond the small types done in Handout #30(a)/(b); overall, your generalized ASM specification should translate cleanly and directly into the optimized RTL implementation.

After optimizing for improved performance, secondarily, improvements to area will be considered. There are a number of creative ways to increase parallelism for this problem, and thereby to optimize the critical path. Other optimizations include reducing unnecessary hardware. Points are also given for elegance of solution. Above all, even if your design is heavily optimized, the top-level generalized-ASM specification should be clean and understandable.

Optimizing the Performance of Your Design: Grading.
Introduction. The focus of performance optimization is on "steady-state" performance: how many clock cycles per inner loop iteration. It is assumed that you will have (i) a few states before the inner loop of your generalized ASM, to handle initialization, setup, and calculating a couple of initial terms of a series, and (ii) a few states after the inner loop, to place results on the output bus. These will typically be ignored in evaluating your steady-state performance (unless they are excessive!).

The design target is *steady-state performance,* which is evaluated by determining how many clock cycles to compute the remainder of the series, and determining the resulting *average # of cycles to compute each term*. For example, if the user specifies a Maclaurin series of 12 terms, you may handle the first 1 or 2 terms before the inner loop, and the rest of the terms in the inner loop. The steady-state performance would be the total remaining number of clock cycles (ignoring (i) and (ii)) to compute the series, divided by the number of terms computed (again ignoring (i) and (ii)).

Performance Targets. For this project, a basic slow but correct solution, or one with moderate parallelism, with steady-state performance of 10-16 clock cycles per term, will obtain up to *90% of perfect grade*. An improved design with greater parallelism, with steady-state performance of 6-9 clock cycles per term, will obtain up to *100% of perfect grade*.

Other Grading Guidelines: Reduced Assignment. You may hand in a reduced assignment, where you only implement one function: natural exponential (e^x). If you do, and obtain steady-state performance of 10-16 clock cycles per term, you will obtain up to *65% of perfect grade*; if you obtain steady-state performance of 6-9 clock cycles per term, you will obtain up to *75% of perfect grade*.

Hints: Suggested Strategy. The project problem can be solved with many different generalized ASM specifications and microarchitectural implementations.

First, (i) understand FP add/sub and mult operations well, and (ii) sketch basic fragments of pseudo-code and generalized ASM, to handle each of these FP operations. These operations are the foundation of supporting a unit to calculate the required three Maclaurin series. You may not use the latter exactly in the final generalized ASM, but it solidifies your understanding of how these FP operations work, and how they can be supported. Remember that – for the RTL specification – each FP operation will be broken down into several *simple steps* over several

clock cycles (i.e. generalized ASM states), and that each step is a *simple RTL operation*.

Next, you should work out a basic solution to the entire project problem. In general, as a design strategy, first focus on a correct and simple design. Do not make it either very sub-optimal or over-optimized/complex. Instead, work out a "clean" fairly efficient initial solution which works correctly and has an easy-to-follow flow. It should not have much parallelism, but provide a simple sequential set of steps to implement each series.

A good approach is to first focus on one function – natural exponential (e^x) – and complete a basic specification, without any user-specified precision. Next, add the capability for user-specified precision. Finally, incorporate the specification of the other two functions: sine and cosine.

Then, for your final version, focus on optimizing your specification (generalized ASM) and its resulting micro-architecture. You should explore simple and clear techniques to have operations occur in parallel, as well as to share function units. You should entirely avoid any manual hacks on the hardware.

Background and Related Reading. The basic IEEE standard floating point representation is covered in B/V ch. 5.7.2 (3rd edition), as already assigned and covered in lecture.

A very useful reference, which covers FP addition and multiplication, is Patterson/Hennessy "Computer Organization and Design", in section 4.8, in an older edition (2nd edition); *several copies are on reserve for the class in the Engineering Library.*

You should *ignore* the hardware implementations provided in this book – they are *not* what you will be producing, and can only confuse understanding by leading to bottom-up design. In contrast, you are to start with top-level pseudo-code and a generalized ASM, and work downward.

Other resources include the Wiki page on 'floating point', http://en.wikipedia.org/wiki/Floating_point, and in other books and online documents. Taylor and Maclaurin series are covered in the Wiki page:
http://en.wikipedia.org/wiki/Taylor_series.