Floating Point Implementation
UKY EE 480

Grant Cox
University of Kentucky
grant.cox@uky.edu

Josh Carroll
University of Kentucky
joshua.carroll@uky.edu

Tyler Williams
University of Kentucky
phillip.williams1@uky.edu

***Abstract*—The final component of the PinKY architecture was to be a "mutant" 16-bit floating point module (FPU). The FPU performs conversions, multiplications, reciprocals, additions, and subtractions on floats with 1 sign bit, 8 exponent bits, and 7 mantissa bits. This is a multicycle design using a state machine in Verilog. It was simulated and tested in Icarus Verilog with GtkWave.**

I. Approach

A. FTOI

The floating point to integer (FTOI) operation was the first instruction that we chose to implement. The goal of this operation is to convert a floating point value to a relatively close integer value. There are many ways to do FTOI, and most of them differ in the way that they choose to round the floating point value (i.e. round down/up, round to zero, truncate). In our case we chose to merely truncate the decimal components of the floating point value being converted. The only special case for this instruction is when the original floating point value, passed to the floating point unit as 16 0's in *Rn*. Handling this case is fairly easy, as the floating point and integer representations of 0 are identical. All other cases of floating point to integer conversion can be handled in the following manner. The first step of floating point to integer conversion is to determine whether or not the exponent (bits [14:7]) is greater than or less than/equal to 134. The value 134 is significant because it is the value of the bias subtracted from the exponent in the general formula for converting binary floating point to a decimal floating point value. See equation 1.

$$(-1)^S \cdot 2^{E-bias-7} \cdot \{1'b1, T\} \tag{1}$$

If the exponent in the float representation is greater than 134, then the integer result is the mantissa of the floating point value (bits [6:0]) left shifted by (134 - EXP). This is equivalent the multiplying the mantissa by a power of 2 equal to (134 - EXP). If the exponent in the float representation is less than (or equal to) 134, then the integer result is the mantissa of the floating point value (bits [6:0]) right shifted by (134 - EXP). This is the equivalent of dividing the mantissa by a power of 2 equal to (134 - EXP). The next step is to check whether or not the sign of the floating point value was positive or negative. This can be done by checking the top bit of the float. If the top bit is 0 (meaning that the float is positive) then the instruction is complete and the done flag is set. If the top bit is 1 (meaning that the float is negative), then the instruction moves to the next stage (which occurs in a new clock cycle). If the original float value was found to be negative, then the integer result found earlier is converted using two's complement to its negative equivalent and the done flag is set.

B. ITOF

There is no point to having a floating point to integer conversion instruction if you cannot convert back from an integer representation to floating point. This issue elicits the need for the integer to float instruction (ITOF). The goal of this operation is to convert an integer value to a floating point representation that is equivalent to the original integer. This instruction also has the special case of having to convert 0 to a float. Luckily, in our representation the 16-bit floating point representation of 0 is *16'h0000*, which is the same as the integer representation. All other integers can be handled in the following manner. The first step in converting a 16-bit integer to a float is to determine the sign, which can be done by looking at the top bit of the integer. The sign bit of the floating point result is set equal to the top bit of the integer representation. If the top bit is 1 (meaning the integer is negative), then the 16-bit register containing the integer is converted to positive by using two's complement. The next stage of this instruction (which occurs in a second clock cycle) is to determine the exponent and fraction values for the floating point result. The first stage of this instruction, discussed above stores the positive integer into a 16-bit register called *int*. This register is the input to a module that counts the leading 0's of any 16-bit value. The output of this module is stored in a register called *d* that holds the number of leading 0's found in the input binary number. The amount of leading 0's in the integer representation contributes to the resulting exponent of the floating point representation. The smallest positive integer that can be represented in 16-bits is 1, which would cause the leading 0's module to count 15 0's. In order to correctly represent the integer 1 as a float the exponent is calculated by:

$$exp <= 127 + (15 - d) \tag{2}$$

This formula gives way to the fact that the largest exponent value is 142 (*8'b10001110*), and the smallest exponent value is 127 (*8'b01111111*). In addition to determining the exponent value, the number of leading 0's also determines the fraction component of the floating point representation. The 16-bit

integer value is shifted left by $1 + (\#ofleading0's)$, which essentially shifts off the implicit 1 that is the assumed 8th bit of the fraction component. The third stage of this instruction concatenates the sign (1 bit), exponent (8 bits), and the top 7 bits of the shifted integer value together to create the 16-bit floating point result (and the done flag is set).

C. MULF

The relatively easiest mathematical operation instruction for floating point values is multiplication. The first step of floating point multiply is to check whether or not either float value is 0. If either operand is 0 then the result of the multiply operation is 0 and the done flag is set. If both operands are non-zero, the next step is to determine the sign of the floating point result. The sign of the result is determined by an *XOR* operation of the sign bits of both floating point operands. The 8-bit result exponent (stored in *exp*) is determined by adding the exponent values of the two operands and subtracting the bias of 127. A second result exponent (stored in *exp_p1*) is calculated in the first stage by adding the operands' exponent values and subtracting 126 (the bias 127 - 1). The result fraction component is determined by multiplying the two 8-bit mantissa components (the implicit leading 1 concatenated to the 7-bit fractions) of each operand. This product is stored into a 16-bit register (the maximum number of bits for an 8-bit integer multiply). The second stage of the multiply instruction determines the final floating point result. If the leading (16th) bit of the mantissa product is 1 (meaning that the product overflowed) then the exponent is *exp_p1* and the fraction bits are taken from bits [14:8] of the 8-bit mantissa product. Otherwise, the exponent is *exp* and the fraction bits are taken from bits [13:8] of the 8-bit mantissa product (with a zero concatenated on the end as the 7th bit). Once the exponent and fraction values are determined, the sign is concatenated to the beginning to create the 16-bit floating point product of the two operands.

D. RECF

Instead of implementing a floating point division instruction (which is very complicated and expensive), we can merely take the reciprocal of the divisor and multiply it by the dividend. Having already implemented the multiply instruction, in order to have dividing functionality we needed to implement a reciprocal float instruction (RECF). There are many ways to calculate the reciprocal of a floating point number at the bit level. Because we are using a 16-bit floating point representation we are able to use a lookup table to convert the original floating point value to its reciprocal. More specifically, a lookup table (provided by Dr. Dietz and placed in a Verilog memory file) is used to convert the fraction component of the original floating point number to the fraction component of its reciprocal. Therefore, we loaded the memory file (128 7-bit values) into a 128 member array of 8-bit registers. Conveniently enough the value of the 7-bit fraction component corresponded to the array index of the 7-bit fraction component of the reciprocal. The exponent of the reciprocal is determined by subtracting the exponent of the original float from 253 (bias 127*2 - 1). The sign bit of the reciprocal is equivalent to the sign bit of the original float. The second stage of this instruction concatenates the sign bit, 8-bit exponent, and 7-bit fraction components to create the 16-bit floating point reciprocal.

E. ADDF

The addition operation is the most complex. We implemented addition of two positive or two negative numbers, but were unable to implement the addition of a positive and a negative number. However, the implementation of addition was staged as follows:

1) Check the signs of the inputs. If they are the same sign, set the output sign and set the next stage to stage 3; else, set the next state stage two and make the output sign the sign of the larger number. In addition, determine which of the inputs is the larger and smaller by the exponent. With this, shift the smaller mantissa by the difference to have the same decimal place as the larger. Add the difference to the exponent of the smaller.
2) If it is a "- + +" or "+ + -" operation, 2's complement the shifted numbers from stage 1. Be careful to shift *in* the implicit 1 in the mantissa if right shifting. Go to stage 4.
3) Add the mantissas. The result is stored in a temporary register which has an extra bit on top for checking for mantissa overflow in stage 4.
4) If there is mantissa overflow, store the top 7 bits of the result into the fraction and add 1 to the exponent. If there is not overflow, store the bottom 7 bits from the mantissa addition into the fraction. If there was a mantissa subtraction and there are leading zeros, normalize the mantissa appropriately and subtract the difference from the exponent.
5) Store the result in the output.

F. SUBF

As stated above, ADDF works for positive plus positive and negative plus negative addition. This was extended for SUBF for the case of positive minus a negative and negative minus positive, because these can be translated to addition. When these cases are encountered, the instruction points to the approprate state of the ADDF instruction and continues operation in that state machine.

II. Issues

FTOI, ITOF, MULF, and RECF work as expected for all positive and negative numbers. ADDF works with two positive or two negative numbers. Our issues arose when working on addition with numbers of opposite signs. There is a problem with with the twos complement operation of the mantissa. It was unclear how an overflow in the mantissa addition effected the result. Should the overflow be ignored, or should some normalization process occur after the addition? This was unclear for the *general* case. We were able to compute the

result for some, but not all cases, particularly when there was mantissa overflow.

Our group ran out of time to fully implement the SUBF operation. Again, this was hindered by a lack of understanding of how the implicit 1 and potential mantissa addition overflow affected twos complement result.

Our group also failed to resolve timing issues when running the floating point module with the PinKY processor. There was difficulty understanding how to monitor the ”done” signal from the FPU. The processor’s interlock trigger would be driven with x’s under all implementations that we tried. Even though the FPU would finish the operation, the CPU was stuck in a loop.