[[1]](#footnote-1)

16-bit FPU Design and Synthesis (Dec 2021)

Long Nguyen, Uriel Lopez, *Student, IEEE*.

***Abstract*—This paper documents the ideation, design, implementation, and testing of a 16-bit FPU and fast inverse square root circuit in Verilog Hardware Description Language. The theoretical background and motivation of the project will be explored, along with the specifications of implementation. The final section of this paper performs data analysis on results and concludes with relevant engineering concepts applied in the project.**

***Index Terms*—Sequential, FPU, Arithmetic, Floating Point, Verilog , ModelSim, Synopsys.**

# INTRODUCTION AND BACKGROUND

M

OST modern computers run on 32-bit or 64 bit operating systems and support floating point instructions. The most common type is single precision floating point, which utilizes 1 sign bit, 8 exponent bits, and 23 mantissa bits. The other common representation is double representation: 1 sign bit, 11 exponent bits, and 52 mantissa bits. There are also niche representations such as half precision and quadruple precision for floating point. This project focuses on designing a 16-bit FPU via modular approach and utilizing said units to recreate the fast inverse square root algorithm in hardware. A few benefits of using 16 bits representation is that it uses less logic area and power than standard or double representation at the cost of decreased precision.

# GOALS AND SPECIFICATIONS

The goal of this project is to create a floating-point logic unit that computes add, subtract, multiply, divide, and a value comparison for two 16-bit numbers. Note that only the computational unit is implemented in this project, which is only one essential part of a computer architecture. The FPU utilizes truncation as its rounding method. Furthermore, a circuit will utilize the FPU to approximate the inverse square root of a 16-bit number within reasonable accuracy. While the operation may not be as efficient as software, this hardware implementation of the algorithm demonstrates an interesting proof of concept for adaptation to a 16-bit system. This project has three main stages: theory and design, RTL verilog implementation, and simulation with synthesis. For the design, Long devised the floating point arithmetic operations flowchart for the FPU while Uriel designed the comparison and state graphs. Using these, a block diagram of the circuits were created. During the implementation, Long worked on the lower-level arithmetic logic behavior while Uriel worked on the controls and top level. Finally, Long did the simulation and synthesis to verify that the design performs correct computations.

# FPU DESIGN

1. *Overview of IEEE-754 Half Precision*

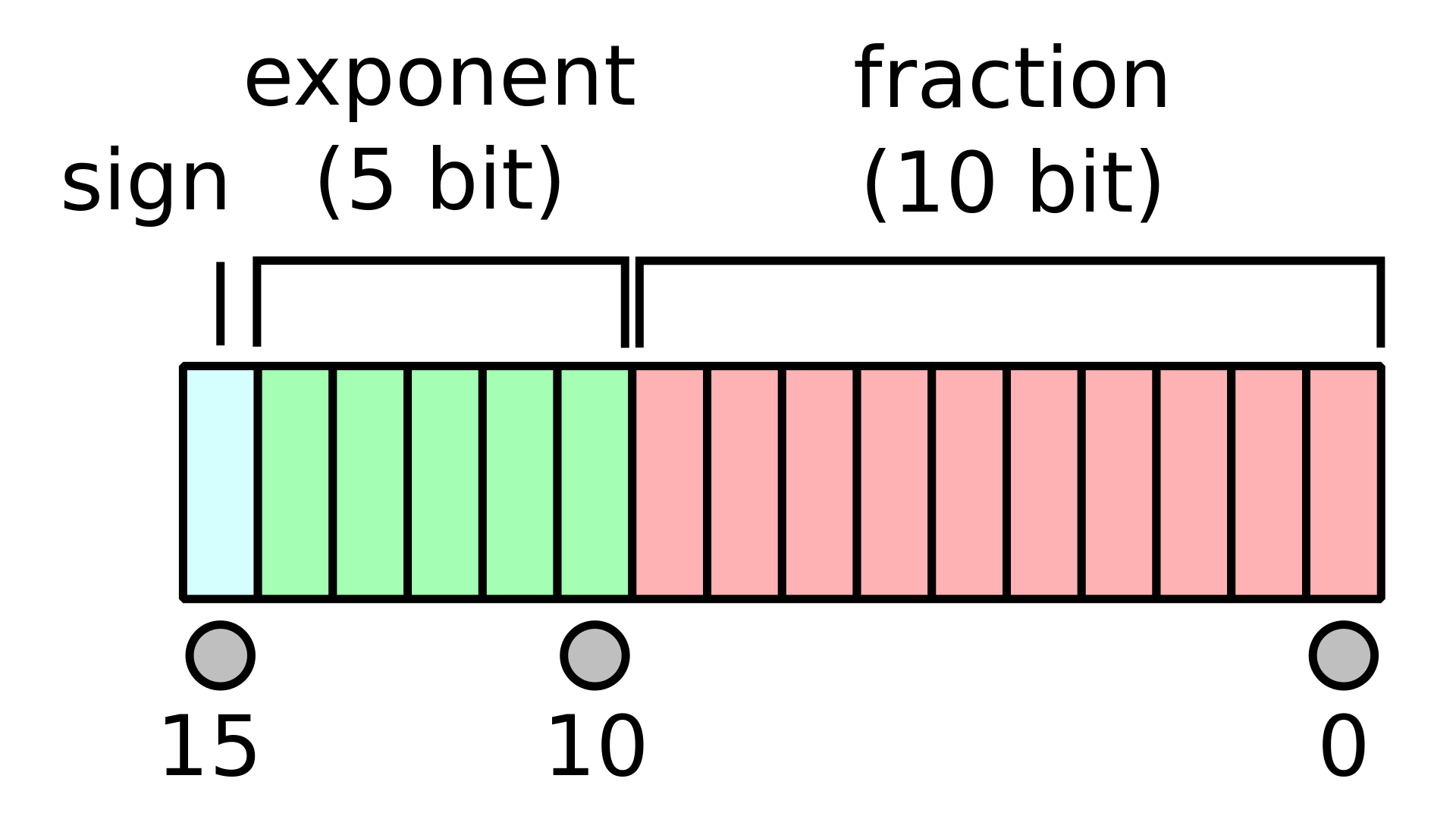


Fig. 1. Half precision floating point binary format.

The value of a half precision number is given by the formula: There are only 5 bits for the exponent and 10 bits for the mantissa. The bias is 15 since there are 5 bits for the exponent. The mantissa also has a hidden bit of 1. If the hidden bit is not 1, then the mantissa must be normalized by shifting the mantissa and incrementing the exponent depending on the number of shifts needed.For this implementation, subnormal numbers will be considered whenever Exp = 0 such that the range will be In which case the formula becomes: [1].

1. *Add-Subtract Circuit*

Diagram

Description automatically generated

Fig. 2. Half precision add/subtract circuit flowchart.

The Add-Subtract circuit performs either operation based on the opcode. Performing both operations is similar thus was made into one circuit. First the circuit checks whether or not any of the inputs are 0. Based on this information it makes the result equal to ‘X’ if ‘Y’ is zero; otherwise the result is ‘Y’ if operation is addition or ‘!Y’ if operation is subtraction. When neither input is 0, and the operation is subtraction, then the sign bit of ‘Y’ is inverted. To add or subtract, the exponents of both are compared to ensure both are equal. In the event the exponents are not equal, the smaller exponent is incremented, and its mantissa is shifted to the right. However, if the mantissa is 0, then the result is the other input. With the exponents being the same, the mantissas are then added or subtracted, and the result is stored into the result with the sign bit being updated. The circuit checks if an overflow occurred at calculation. If it overflows with maximum representable exponent, ‘111111’, then the overflow flag is set ‘OFUF = 2’b10’ and the result is at a high impedance state. Underflow occurs when the hidden bit in the mantissa is 0. When this occurs with the exponent being the minimum represented exponent, ‘5’b0’, then the underflow flag is set ‘UFOF <=2’b01’ and the done signal is returned. Otherwise, the mantissa is normalized and rounded. This circuit outputs the result: updated sign, exponent, and normalized mantissa; then it outputs the done signal.

1. *Multiply-Divide Circuit*

Diagram

Description automatically generated

Fig. 3. Half precision multiply/divide circuit flowchart.

The Multiply-Divide Circuit takes the ‘opcode’ to determine if this circuit will perform the multiplication or division operation. First, if ‘X’ is 0 then the output is also 0. If ‘Y’ is 0 when the operation is multiplication, then the output is also 0. But if the operation is division while “Y’ is 0, then overflow occurs and the OF flag is set. If neither input is zero then the exponents are summed then bias is subtracted if the operation is multiplication. If the operation is division, then the exponents are subtracted and then bias is added. Exponent underflow or overflow can occur if it, ‘zExp’, is less than 0 or above 30; flags are set accordingly. If no overflow or underflow occurs then the mantissas are multiplied or divided accordingly. The mantissa will become a 22 bit result (manTemp[21:0])  after the calculation, if the MSB is 1, then overflow occurs. If the exponent is more than 30, then the overflag is set. To normalize the mantissa if the exponent is within allowed range, the bits [21:13] are taken out to use as the mantissa for the final result and increment the exponent. While no overflow occurs for the 22 bit result after calculations of the mantissa, the hidden bit determines if this number is normalized. If the hidden bit is 1 then the number is normalized and bits [20:11] are extracted for the final mantissa zMantissa. If the tempMantissa is not normalized, then the result is shifted left and the zEXP is decremented. If exponent underflow occurs then the flags are set, otherwise check for normalization again. The final result is then put together with updated sign bit, zExp, and zMantissa.

1. *Comparison Circuit*

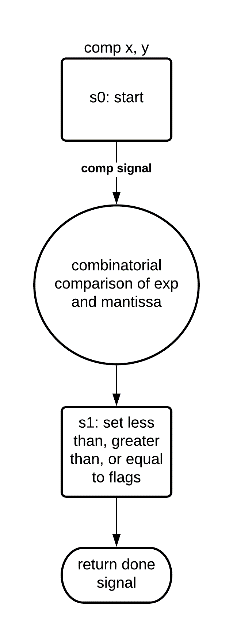


Fig. 4. Half precision comparison circuit flowchart.

The comparison circuit takes inputs ‘X’ and ‘Y’. It first compares their sign bit to determine which is greater. If it’s the same, then the exponent is then compared. If these values are the same, then the mantissa is then compared. If ‘X’ is greater than ‘Y’, then the result is ‘3b’100’. If ‘X’ is less than ‘Y’ then the result is ‘3’b010’. If everything is the same, then the result becomes ‘3b’001’ to show that the inputs are equal.

1. *FPU Controls*

Using the three circuits above, the FPU is achieved. The inputs it takes are X, Y, opcode, clk, and reset. It outputs result, OFUF, done, and compResult. The circuits made above are used via positional association. There is a case statement that uses ‘opcode’ to switch between each case. It sends created wires and new reg addSub and mulDiv to determine the operation to be done. If a circuit is not used due to the opcode, then the corresponding regs, addSub or mulDiv is set to high impedance.

Diagram

Description automatically generated

Fig. 5. FPU state graphs for the submodules.

Diagram, schematic

Description automatically generated

Fig. 6. FPU block diagram.

# FAST INVERSE SQUARE ROOT

* 1. *Applications and Mathematical Theory*

As the performance of computers began to improve, so too did the demands of more computationally intensive software. One notable application is computer graphics, which requires fast vector calculations to render geometry and shading. One particularly important operation is vector normalization, which requires a division by the sum of each squared vector component : . Thus, the fast inverse square root algorithm was devised to approximate this computation with exceptional accuracy and speed by eliminating costly division instructions.

float InvSqrt (float x){

float xhalf = 0.5f\*x;

int i = \*(int\*)&x;

i = 0x5f3759df - (i>>1);

x = \*(float\*)&i;

x = x\*(1.5f - xhalf\*x\*x);

return x;

}

Fig. 7. FIS Algorithm. The second assign statement casts the float variable as an integer without modifying the bit values, allowing the left shift operator to function correction in the following line.[2]

The algorithm above contains many clever tricks to find the solution. The following is one possible mathematical explanation for its behavior.

Finding the roots of f(x)=0 yields the following:

Newton's Method states that given an initial guess and input , the approximation of the root is defined as:

Solving this equation yields: . The real mystery is why 0x5f3759df - (i >> 1) yields an excellent initial guess. While the complete mathematical derivation of this constant is beyond the scope of this paper, it appears that this constant is approximately equal to , where *k* is a very small constant relative to the total sum [2]. The most interesting value here is 127, which is exactly equal to the exponent offset of a single precision floating number. Thus, by applying the same logic to 16-bit offset of 15, the magic constant becomes , for a fined tuned *k* = 2.38 found through trial and error. This modified constant miraculously works for the half precision implementation, demonstrating the merit of the theory through empirical results.

half invSqrt(half x) {

const half threeHalfs = 1.50;

half xHalf = x / 2;

half y = x;

int bitHack = \*(half\*)&y;

bitHack = 0x59BB - (bitHack >> 1);

y = \*(half\*)&bitHack;

y = y \* (threehalfs-(xHalf\*y\*y));

return y;

}

Fig. 8. Modified FIS algorithm for 16 bit floating point.

* 1. *Implementation*Diagram

     Description automatically generated

Fig. 9. Fast inverse square root circuit flowchart.

Diagram

Description automatically generated

Fig. 10. Pipelined fast inverse square root circuit flowchart.

With theory out of the way, the next step was to implement this in hardware. The FPU designed in the previous part could handle all the required arithmetic operations, thus the only thing left to do was to implement a state machine that faithfully performs the software algorithm line by line. Using a similar approach as the previous circuits, a Moore state machine was devised. The top module consists of a single FPU circuit, temporary registers to store the inputs and outputs, and controls logic module. In hardware, the value of bitHack = 0x59BB - (bitHack >> 1) would be computed with relative ease without worrying about casting the number from half to int and vice versa. The first state is 0, which is the asynchronous reset stage. From there, the circuit transitions to the next states and adjusts the FPU inputs and opcode accordingly. Whenever the FPU returns an overflow or underflow signal, the controls will change the next state to an invalid error state and stop computation. However, cases of underflow or overflow is rare for the fast inverse square root circuit. The instruction y = y\*(threehalfs-(xHalf\*y\*y)) requires 4 instructions to perform with only 1 FPU. However it could also be written as y = (y\*1.50) - (y\*y)(y\*xHalf), which opens up the opportunity for pipelining at this step using 2 FPUs in 3 instructions. Thus, at the cost of more area can the execution time be improved. This pipelined version was also implemented and compared with the original circuit.

Diagram

Description automatically generated

Fig. 11. State graphs for fast inverse square root circuit and its pipelined version.

Diagram

Description automatically generated

Fig. 12 Fast inverse square root block diagram.

Diagram, schematic

Description automatically generated

Fig. 13 Pipelined fast inverse square root block diagram.

# SIMULATION WAVEFORMS AND ANALYSIS

The Verilog code was compiled and simulated using ModelSim. Because there was no support for half precision waveform output formatting, the inputs and outputs were converted to their 32-bit equivalents for convenience on the simulation waveform. Furthermore, the test bench includes a counter that starts at 0 and increments every clock cycle until the done signal is set to HIGH to count the number of cycles were needed for that specific computation.

Diagram

Description automatically generated

Fig. 14. FPU addition operation results for 0.000427246 + 0.000228882 = 0.000656128 in 5 clock cycles.

A picture containing diagram

Description automatically generated

Fig. 15. FPU addition operation results for 2.77539 – (-0.219727) = 2.99414 in 8 clock cycles.

Graphical user interface

Description automatically generated

Fig. 16. FPU addition operation results for -177.625 \* 28.125 = 4992.00 in 4 clock cycles.

A picture containing graphical user interface

Description automatically generated

Fig. 17. FPU addition operation results for 2.77539 / -0.219727 = -12.625 in 5 clock cycles.

Overall, the results were unexpectedly accurate within less than 0.1% error of the correct answer. However, 16 bits feature much less precision than 32-bit or 64-bit floating point numbers. Next, the test bench for the fast inverse square root circuit is like the FPU. However, it compares both the pipelined and non-pipelined version of the circuit for the same input.

A picture containing diagram

Description automatically generated

Fig. 18. Fast inverse square root circuit for in 28 cycles standard and 23 cycles pipelined.

A picture containing graphical user interface

Description automatically generated

Fig.19. Fast inverse square root circuit for in 28 cycles standard and 21 cycles pipelined.

Graphical user interface

Description automatically generated with medium confidence

Fig. 20. Fast inverse square root circuit for in 28 cycles standard and 21 cycles pipelined.

The pipelined version reduces the total computation time of the circuit by about 5 clock cycles. However, it could reduce it even more such as the example from figure xxxx. This leads to about a 20% performance boost of the pipelined version over the standard version. Since it uses the same FPU hardware for computations, the results are usually within 0.1% error of the correct result, which is a limitation of half precision. However, that is adequate precision for any non-critical calculations.

# SYNOPSYS SYNTHESIS REPORTS

FPU Synthesis

* Timing:
  + Clock Period: 1.5 ns (0.32 slack)
* Power:
  + Cell Internal Power = 1.3119 mW
  + Net Switching Power = 655.5172 uW
  + Total Dynamic Power = 1.9675 mW
  + Cell Leakage Power = 217.2053 uW
* Area:
  + Combinational area: 2337.874010
  + Buf/Inv area: 204.022002
  + Noncombinational area: 937.117998
  + Total cell area: 3274.992009

Fig. 21. FPU ASIC Synthesis on Synopsys for 45nm.

FastInvSqrtPipelined Synthesis

* Timing:
  + Clock Period: 1.5 ns (0.32 slack)
* Power:
  + Cell Internal Power = 1.4501 mW
  + Net Switching Power = 27.9948 uW
  + Total Dynamic Power = 1.4781 mW
  + Cell Leakage Power = 326.7899 uW
* Area:
  + Combinational area: 3386.978015
  + Buf/Inv area: 281.162003
  + Noncombinational area: 1645.209982
  + Total cell area: 5032.187998

Fig. 22. Pipelined fast inverse square root circuit ASIC Synthesis on Synopsys for 45nm.

While closely analyzing the synthesis reports are beyond the scope of this course, some general statements can be made. First, both circuits had a clock period of 1.5 ns, which presumably is constrained by the mulDivCircuit performing hardware multiplication and division. Next, the total cell area of the pipelined inverse square root circuit is roughly less than half of two FPUs. This could mean that the synthesis tool optimized away some of the hardware. Finally, the total power for the single FPU exceeded the inverse square root circuit that utilizes 2 FPUs. This is because Synopsys simulates power by switching the inputs on and off, thus showing that the FPU is more sensitive to changes in input than the fast inverse square root circuit.

# CONCLUSION

At first, the project idea and proposal appeared to be quite complex for the scope of an ECE 176 project. There was much difficulty each step of floating-point arithmetic from the beginning o the process to the end normalization or overflow/underflow error report. Furthermore, 16-bit floating point is a rather niche topic that has not been thoroughly documented, thus there was less material available for theoretical research. We had to adapt most of the theory from singe precision procedures to half precision representation. One feature of the FPU that was not planned was the usage of guard, round, and sticky bits to improve precision, but could be implanted in future iterations of this FPU. The other surprising aspect was that we totally did not the fast inverse square root algorithm to work for this project. It originates from software, which sometimes is quite difficult to efficiently implement as hardware (especially those that require loops). However, we believe that this is the first time the fast inverse square root has been adapted to 16 bits and implemented in hardware. This proves that applying sound theory and logic to new concepts can sometimes lead to new discoveries. Thus, while this implementation of an FPU and pipelined fast inverse square root circuit may be too inefficient to utilize in modern computing, the success of both project components bears engineering merit of its own.

# REFERENCES

|  |  |
| --- | --- |
| [1] | “Half Precision Floating-Point, Visualized” Available: <https://observablehq.com/@rreusser/half-precision-floating-point-visualized> |
| [2] | R. Munafo, “Notable properties of specific numbers” Notable Properties of Specific Numbers (page 16) at MROB. [Online]. Available: <https://mrob.com/pub/math/numbers-16.html#le009_16>. |

1. This manuscript was submitted on December 13 2021 as part of the ECE 176 Final Design Project for Dr. Stillmaker at CSU Fresno. All design files, block diagram schematics, synthesis, and verilog files were submitted. [↑](#footnote-ref-1)