Digital Systems Design Report 3

Justin Huang
jh4420@ic.ac.uk
Jubo Xu
jx1820@ic.ac.uk

March 20, 2023

The format of the report is not great. Section are ~~in~~ a single column, when the section inside are double column.

0.1 Definitions

FP: Our system uses the IEEE-754 floating-point standard with single precision. For convenience, we use the acronym "FP" to refer to this standard in later discussions.

Relative Percentage Error: The relative error percentage is calculated by comparing with the value calculated from MAT LAB

$$\|\frac{Result_{matlab} - Result_{nios}}{Result_{matlab}}\|$$

FB: Number of fractional bits for fix-point represent

Full Custom: Our fully pipelined final hardware in custom instruction

0.2 Custom Instruction

0.2.1 Adding Floating-Point Units and Custom Instruction

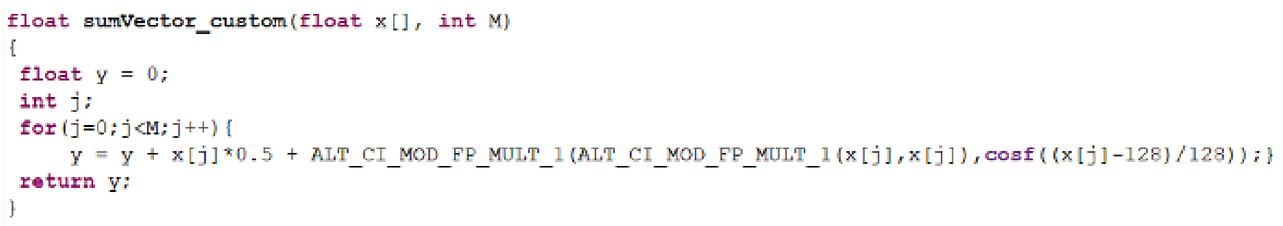
In the previous system, there is no hardware floating-point support, which means that all the floating-point operations are emulated in software at the expense of fixed-point operations. For our performance-oriented function accelerator, this complication significantly increases the application size and the latency of the system. Furthermore, floating-point numbers provide a larger data dynamic range and can perform more accurate computations with smaller rounding errors. Therefore, dedicated floating-point hardware becomes necessary.

generic statement which does not always apply.

0.2.2 Custom Floating-Point Multiplication Instruction

We instantiated an Altera FP multiplier hardware with single precision. The component utilises 55 LUTs and 136 registers on the FPGA. The latency of the operation is 5 clock cycles. The hardware without a pipeline has a throughput of 0.2 per clock cycle. This hardware has the potential to be pipelined, which can have a throughput of 1 per clock cycle. The pipelined version is used in task 8.
A custom FP instruction ALT_CI_MOD_FP_MULT_1 is created which takes two FP operands and returns the FP multiplication result. We adopted the Fixed length Multi-Cycle Custom Instruction Architecture of NiosII. The length of the operation is set to 5 cycles so the custom instruction interface can read the valid result from the FP multiplication IP on the correct clock cycle. The Start and Done signal is not used now for simplicity. These two signals can be added later to enable variable-length multi-cycle operations.

As shown in figure 1, for test case 3, the text size is reduced from 82948 without custom FP instruction to 82940 with custom FP multiplication. One custom FP multiplication instruction replaces 9 (8 + 1) fix-point instructions to complete one multiplication operation. The introduction of FP multiplication hardware slightly reduces the application size which in turn increases the system performance. For test case 3, the latency of the computation is reduced from 94930 ticks without custom FP instruction to 79223 ticks with custom FP multiplication. In addition, the relative percentage error is reduced from 0.0009114% without custom FP instruction to 0.0000038% with custom FP multiplication. This is because FP representations can handle numbers over a larger range without losing precision compared to fix-point representations with the same number of bits.



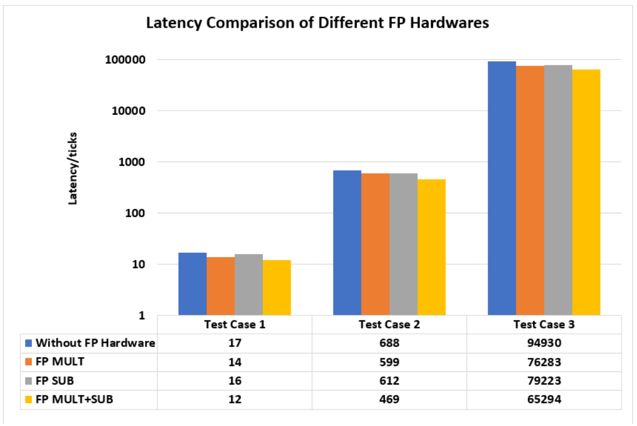
```
float sumVector_custom(float x[], int M)
{
 float y = 0;
 int j;
 for(j=0;j<M;j++){
     y = y + x[j]*0.5 + ALT_CI_MOD_FP_MULT_1(ALT_CI_MOD_FP_MULT_1(x[j],x[j]),cosf((x[j]-128)/128));}
 return y;
}
```

Figure 1: Custom Floating-Point Multiplication Instruction

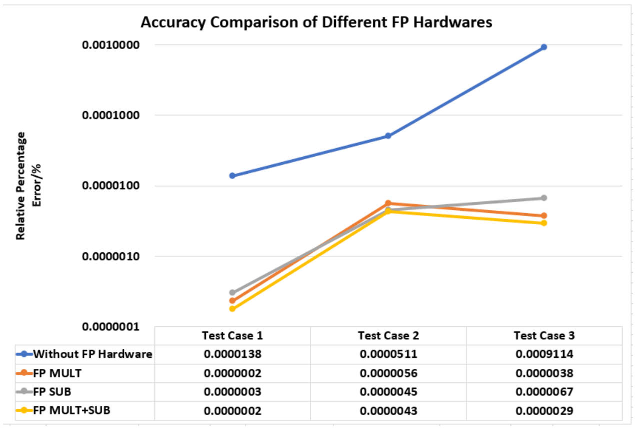
0.2.3 Custom Floating-Point Subtraction Instruction

Is this the minimum latency? Not clear what guided your design decision from the report (not stated), I gave you full marks given your other information

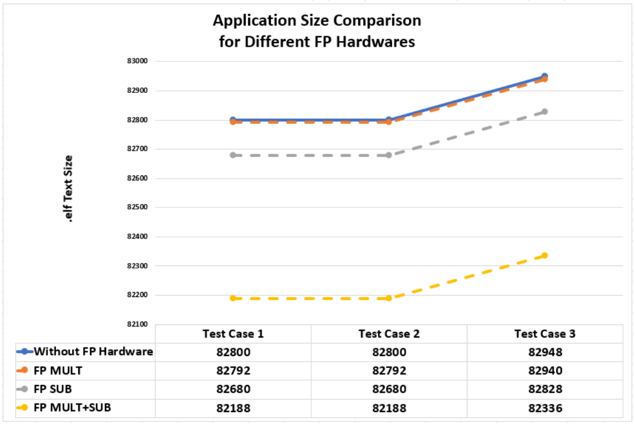
Custom instruction ALT_CI_MOD_FP_SUB_1 is created using the same approach as the previous section.
We instantiated an Altera FP subtraction hardware with single precision and 6 cycles latency. The throughput of the hardware is 0.1667 per clock cycle. The component utilises 1169 LUTs and 902 registers on the FPGA. FP subtraction hardware is more resource-demanding and has longer latency compared to multiplication. FP subtraction involves aligning the binary point of the operands before subtracting them, which requires more circuitry and registers to handle the shifting and alignment of the mantissa and the exponent adjustment. This results in lower parallelism compared to multiplication, which requires more clock cycles to complete the operation. For this reason and with the existence of hardware fix-point multiplication support, it is also reasonable to speculate that more fix-point instructions are required to emulate one custom FP subtraction instruction compared to FP multiplication instruction.
There are more multiplication operations in the function. To make a fair comparison, the following experiment is conducted by replacing one FP multiplication or FP subtraction.
As shown in figure 2, for test case 3, the text size when replacing one FP subtraction is 82828 which is smaller than the text size when replacing one FP multiplication (82940). The latency of the computation when replacing one FP subtraction is 76283 ticks which is smaller than 79223 ticks when replacing one FP multiplication. Even though the FP subtraction hardware introduces more latency (1 cycle) than an FP multiplication, the effect of replacing more fix-point instructions outweighs the slight increase in latency per instruction. Compared to replacing one FP multiplication instruction, the overall performance increase is larger when replacing one FP subtraction instruction.



	Test Case 1	Test Case 2	Test Case 3
🟦 Without FP Hardware	17	688	94930
🟧 FP MULT	14	599	76283
⬜ FP SUB	16	612	79223
🟨 FP MULT+SUB	12	469	65294



	Test Case 1	Test Case 2	Test Case 3
Without FP Hardware	0.0000138	0.0000511	0.0009114
FP MULT	0.0000002	0.0000056	0.0000038
FP SUB	0.0000003	0.0000045	0.0000067
FP MULT+SUB	0.0000002	0.0000043	0.0000029



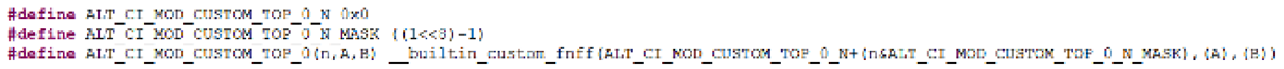
	Test Case 1	Test Case 2	Test Case 3
Without FP Hardware	82800	82800	82948
FP MULT	82792	82792	82940
FP SUB	82680	82680	82828
FP MULT+SUB	82188	82188	82336

Figure 2: Latency, accuracy and application size for different Hardwares Blocks in Task 6

0.2.4 Floating-Point Multiplication and Subtraction

To support two FP instructions, we utilize the Extended Custom Instruction Port Operation in the NiosII custom instruction interface, with N[7:0] field to mux between the result from the FP subtractor and the FP multiplier. N=8'b1 is reserved for subtraction and N=8'b0 for multiplication. The length of the operation is set to 6 cycles because both FP subtractor and multiplier produce valid results on or before cycle 6. As shown in figure 3, for test case 3, the text size is further reduced from 82948 without custom FP instructions to 82336 with two FP instructions. The latency of the computation is further reduced from 94930 ticks without custom FP instructions to 65294 ticks with two custom FP instructions. The relative percentage error is reduced from 0.0009114% without custom FP instruction to 0.0000029% with two FP instructions. ✓

$\boxed{A}$ Something that you can improve is the investigation on the cache.



```
#define ALT_CI_MOD_CUSTOM_TOP_0_N 0x0
#define ALT_CI_MOD_CUSTOM_TOP_0_N_MASK ((1<<3)-1)
#define ALT_CI_MOD_CUSTOM_TOP_0(n,A,B) __builtin_custom_fnff(ALT_CI_MOD_CUSTOM_TOP_0_N+(n&ALT_CI_MOD_CUSTOM_TOP_0_N_MASK),(A),(B))
```

Figure 3: Floating-Point Multiplication and Subtraction

0.3 Cordic related Hardware Design

0.3.1 Floating to Fix

this is not clear. It is not clear what the objective is.

In our case, the total word length we choose for our fixed point representation is 24 bits with 1 sign bit, 2 integer bits, and 21 fractional bits. The logic of our floating to fix transferring block is based on a huge decoder, since our integer part has 2 bits, the maximum exponential we care about is 128, then the first case is that if the 8 bits exponential part, which is float_in[30:23], is equal to 128, then the most significant integer bit of fixed out should be 1, and the least significant integer bit should be the first bit of mantissa of floating in, then the fractional part should be float_in[21:1]. The second special case is when the exponential part is equal to 127, which means the mantissa has no shift, then the most significant integer bit should be 0 and the least significant integer bit should be 1, the fractional part is just the most significant 21 bits of mantissa. If the exponential part is smaller than 127, which means the mantissa would be right shifted, in this case, the integer bits would be 0, and the most significant (POINT- EXP + 2) bits of fractional part should be 0, where POINT is the number of bits for our fixed point fractional part, and EXP is the absolute difference between the value of exponential of floating in and 127. The following bits of fixed out fractional part should be equal to the combination of 1 and floating_in[22:(22 − POINT + EXP + 1)]. Since we only have 21 fractional bit, if the exponential part of floating_in is smaller than 106, then we would not have enough bits to represent it, so the result would always be 0, so for our decoder we only consider about the range from 106 to 128.

0.3.2 Inside_CORDIC

To calculate the $\frac{x-128}{128}$, we avoid using the floating point unit subtractor. Since the input into the CORDIC is in the range of [-1, 1], which means that $\frac{x-128}{128} = \frac{x}{128} - 1$ is in the range of [-1, 1], so $\frac{x}{128}$ is in the range of [0, 2], to calculate $\frac{x}{128}$, we could simply subtract 7 from the exponential part of x, and we could first transfer $\frac{x}{128}$ into fixed point representation, and that's also why we have two integer bits. Since if $\frac{x}{128}$ is larger than 1, then minus 1 would not loss any accuracy in fixed point representation, if $\frac{x}{128}$ is smaller than 1, then minus 1 in fixed point representation may provide a little bit more inaccuracy, and that's why we have 21 bits for redun-~~dancy cause usually 18 bits would be enough for satisfying our accuracy requirement~~. By calculating $\frac{x-128}{128}$ ~~in this way, we could get~~ the result in a very simple combinational logic in one cycle, which can speed up the calculation.

This is vague "usually" "enough"? You need to be precise + back up claims with data.

0.3.3 CORDIC

The basic structure of our CORDIC module is that we have a basic module that would do the operation for each iteration, including the plus or minus the constant stored for the angle based on the sign of the previous calculated angle, as well as the shift and addition or subtraction of X and Y, all the constants we need are precalculated and stored in the registers, and these stored constants would be directly used in each iteration, and this should be the fastest way to get the constants. Our basic pipelined version is to do each iteration in each cycle, and because we have 21 bits fractional part, then we need 21 iterations in total, therefore we have 21 pipelined stages for our basic structure. Each cycle when the new input comes in, it would get into the block to do the iteration at that cycle. This kind of structure should have the highest throughput in our case, but requires more resources because we need to have the actual hardware block for each iteration. The second structure requires the lowest resources, because we only need one hardware block to do the actual computation, but we need to iterate 21 cycles, and we have to put the new value in after 21 cycles when the original input finish its calculation, therefore such kind of structure has the lowest throughput. The third structure we used is mainly to combine with our other hardware blocks to increase the speed and decrease the latency. The structure is similar to the highest throughput version, but instead of only doing one iteration in one cycle, we do multiple iterations in one cycle, which could reduce the total pipelined stages as well as the delay.

Perfomance analysis of CORDIC

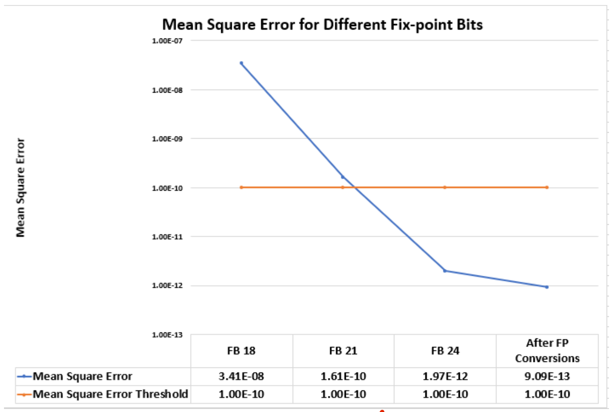
*So far, the selection of the wordlength for CORDIC is not justifiable. You need to check the conversation where I am asking you to do Monte Carlo

Also, the two architecture are not optimal. You need to combine multiple iterate to a single cycle to improve performance.

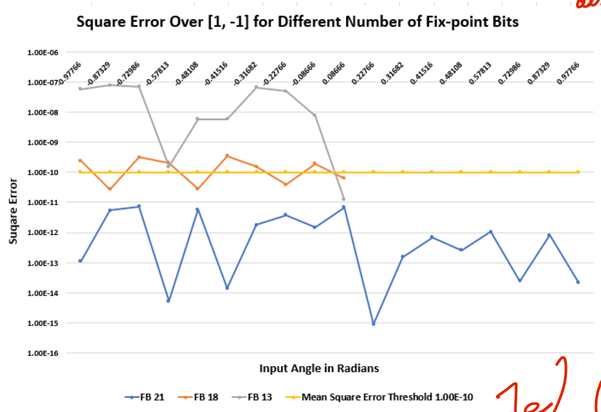
Mean square error: It can be seen from the figure 4 that the Mean Square Error Threshold is the requirement for accuracy in our case, and we choose the number of fractional bits for fixed point representation based on this value. When the number of fractional bits is 18, the general mean square error is larger than the threshold, which doesn't satisfy the requirement completely. It can also be seen when the number of fractional bits is 21, the mean square error is the closet to the threshold, and that's why we finally choose this value. Figure 4 shows our experiment result that shows the square error for each test case in Monte Carlo experiment, when the number of fractional bit is 21, it can always satisfy the condition in our case. The reason why we don't have the data for FB18 and FB13 for large angles is because they've already doesn't satisfy the requirement in the previous tests, so we just don't test them any more.

OK.

But the # of bits also depends on the # iteration.



	FB 18	FB 21	FB 24	After FP Conversions
Mean Square Error	3.41E-08	1.61E-10	1.97E-12	9.09E-13
Mean Square Error Threshold	1.00E-10	1.00E-10	1.00E-10	1.00E-10



7e) $\boxed{B}$

7b) $\boxed{A-}$

Figure 4: Mean square error and Square error over[1,-1] for different number of fix-point bits

(p ~~day~~ why you did not use the 95% confidence? (specification requirement).

0.3.4 Fix to floating

We designed two ways to achieve fixed point to floating point transferring. The first one is a huge combinational circuit. The basic logic is to determine the structure of the floating out based on where the first 1 appears in the fractional part of the fixed-point representation. If the integer parts are 0 and fixed_in[20] is 1, then the exponential part of the floating out would be 126, and the mantissa would be fixed_in[19:0] followed by 3 zeros. Otherwise, if fixed_in[19] is 1, then the exponential part of the floating out would be 125, and the mantissa would be fixed_in[18:0] followed by 4 zeros, and so on. The same logic could be used to set the structure of floating out based on where the first 1 appears. This method is very simple and easy to be implemented, but has the problem of large fan-in and fan-out. Our second method could solve this problem to some extent. This is a kind of hybrid structure, and the main point is to find where the first one appears in the fractional part if integer part is 0. Since we have 21 fractional bits, we first using the same logic as method 1 for the most significant 5 bits, which would be a small decoder with only 5 inputs. To find the first 1 appearing of the remaining 16 bits, we use a kind of divide and conquer structure. There's a 4 bit register R to store the value of position of first 1 appearing. The basic structure is like this: when the 16 bits come in, check whether the leftmost 8 bits equal to 0, if yes, then set all bits of R to 0, if not, then set the most significant bit of R to 1, there is also a MUX to select the 8 bits output based on the same condition, if yes, then the output would be rightmost 8 bits, if not, then the output is leftmost 8 bits. In the second level, the logic is the same. First determine the leftmost 4 bits of the 8 bits input, if it equals to 0, then the R would be no changed, and the output is the rightmost 4 bits, otherwise, set the third bit of R to 1, and the output is the leftmost 4 bits. In the third level, if the leftmost 2 bits equal to 0, then R has no change, and the output is the rightmost 2 bits, otherwise, set the second bit of R to 1, and output the leftmost 2 bits. In the last level, if the left bit is 0, R has no change, otherwise, set the least significant bit of R to 1. Now the value of R is equal to the index of the first 1 appears. Then after getting the value of R, there's a 4 bits input decoder to choose the corresponding output structure. This method can solve the problem of large fan-in, and such kind of architecture is also very suitable for pipelining. As shown in figure 5. ✓

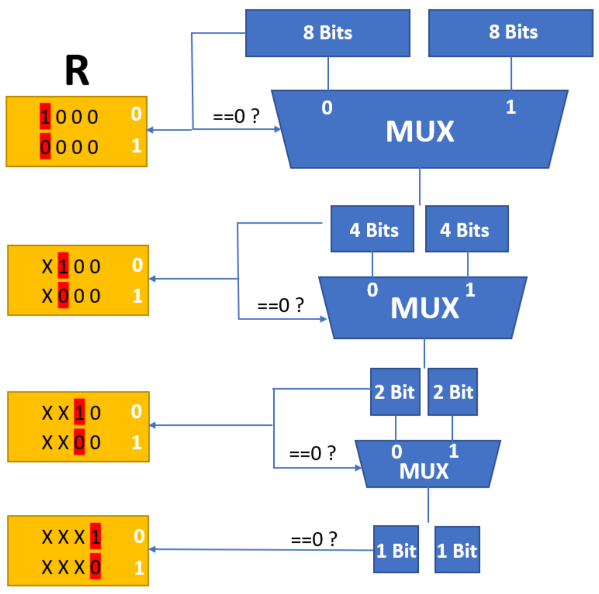


Figure 5: Fix to floating second method

0.4 The final structure

As mentioned in earlier sections, all the FP processing units have pipeline capability which can be exploited to further increase the throughput and performance. Previously, for the simplicity of the NiosII custom instruction interface, we did not implement pipelines for FP computations. The interface waits for a fixed number of clock cycles before reading the valid result from the FP processing unit. Most of the stages remain idle during the operation and their computation capacity is therefore wasted. In order to utilize the full capability of the pipeline, we redesigned the custom instruction interface and introduced control logic for the pipeline. In addition, we include additional shift registers and accumulators to align and sum the result with various delays in the pipeline. These optimisations are explained in the following section. ✓

0.4.1 Custom Instruction Interface

✓ ✓ Good

We utilized the Extended Variable Cycle Custom Instruction. The instruction takes in two floating-point operands. Each time the custom instruction is called, two x array elements are pushed into the pipeline. The Extended N field of the custom instruction is used to signal the controller that the last element(s) has entered the pipeline (N = 8'b1 indicating the last element). Ideally, to exploit the full capacity of the pipeline, new operands should enter the pipeline every clock cycle. To achieve this, the custom instruction should be called every clock cycle. However, this is not achievable in the given NiosII custom instruction interface. The combinational custom instruction does not have a clock port which makes it impossible to clock the pipeline. Therefore, we use the variable length custom instruction with the start and done signal. During full pipeline computation, the done signal is asserted every two clock cycles so the current instruction is terminated and the upcoming instruction can bring new operands to the pipeline. Variable length is used because computations in the pipeline have different latencies. The last element in the function needs to propagate through the whole pipeline before the valid result is generated. From the perspective of the custom instruction interface, the length of the instruction during full pipeline computation is two clock cycles. And the length of the instruction is prolonged when calculating the last element. ✓

0.4.2 Control Logic for the Pipeline

As shown in figure 6, the module which is responsible for pipeline control is CUSTOM_INSTRUCTION. Its implementation is a finite state machine with three stages: IDLE, PROCESS, and FINISH. The IDLE state waits for the first clk_en and triggers the internal clock enable clk_en_use. The clk_en_use is asserted throughout the computation until the result is valid. This signal ensures that the pipeline remains operational even if the custom instruction is terminated (clk_en de-asserted). The PROCESS state is where full pipeline computation takes place. New dataa and datab is transferred to the Processing Element (PE) every two clock cycles when Start is asserted (ie. When new elements enter the pipeline via custom instruction). When the last element is not entered to the pipeline (N != 1), the state machine remains at PROCESS stage. During the verification stage of our design, we discovered that the ALFP_MULT IP output undetermined states between cycle 4 to cycle 7 after startup. These undetermined states propagate in the pipeline and corrupt the computation result. To resolve this issue, we introduce the counter COUNT_INIT. This asserts the signal tick (in module PE) during cycle 4 to cycle 7. The tick is used to mux between 32'b0 during cycle 4 to cycle 7 and valid multiplication result after startup. The FINISH state waits for the computation on the last element to complete. Counter COUNT is triggered at the beginning and keeps counting to CYCLE_DELAY. At this point, a valid result is stablised in the adder tree and a final done signal is asserted so the custom instruction interface can read the result from the hardware. ✓

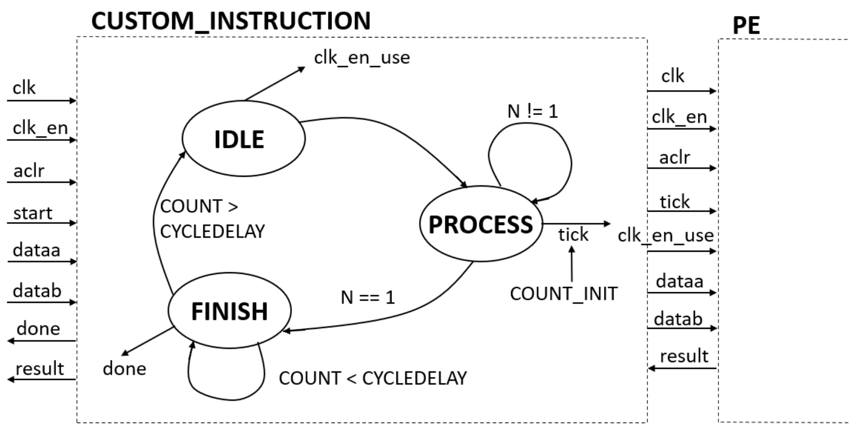


Figure 6: Hardware Calculation Architecture

0.4.3 Hardware calculation part

As shown in figure 7, our architecture is mainly divided into three parts, the first part is DOT, the second part is basic accumulator, and the last part is a tree structured adding system.

DOT

The first level of DOT has three operations, the first one is $0.5x$, the second one is x^2, and the third one is $\cos\frac{x-128}{128}$. $0.5x$ can be calculated simply by subtracting 1 from exponential part of x, which is a combinational circuit, and the x^2 is calculated based on the ALFT_MULT Ip, we choose 6 cycles to complete the calculation, and our basic pipelined CORDIC has 22 cycles, and since our $\frac{x-128}{128}$ is also a simple combinational circuit, the total $\cos\frac{x-128}{128}$ also needs 22 cycles. ✓ Therefore we need to make sure the result of three operations should be synchronized before getting into the second stage, so we need to add 16 delays for multiplication and 22 delays for $0.5x$ in this stage. In the second level, the only operation is the multiplication between x^2 and $\cos\frac{x-128}{128}$, which is also 6 cycles, so another 6 cycles delay should be added for $0.5x$ in this stage. In the third level, the operation is the addition between $0.5x$ and $x^2\cdot\cos\frac{x-128}{128}$, which requires 8 cycles in our case, so the total pipeline stages in DOT is 36. This can be reduced if we change the structure of CORDIC, the most basic version is that only one iteration will be applied in one cycle, we could combine several iterations into one combinational logic and do it in one cycle, which would decrease the latency. In this case, we could adjust the number of iterations in each stage to make CORDIC has the same cycle as the multiplier, which is 6 in our case. In this way, we could further reduce the total cycles and increase the speed if we are using the same clock frequency in all cases.

Accumulator

In this part, since we can simultaneously put two inputs into the custom instruction, we have two DOTs run in parallel in hardware, and the first adder is to add the result of two DOTs, the second adder is a kind of recursive adder, since the operation cycle of the floating point adder we used is 8 cycles, such kind of structure could add the values with integral of 8 cycles, for example, if the input gets into our hardware every cycle, which means the hardware is completely pipelined, considering about the moment that the last value just comes in, then the output of this recursive adder would be $\sum_{i=0} x_{8i}$, $\sum_{i=0} x_{8i+1}$, $\sum_{i=0} x_{8i+2}$,

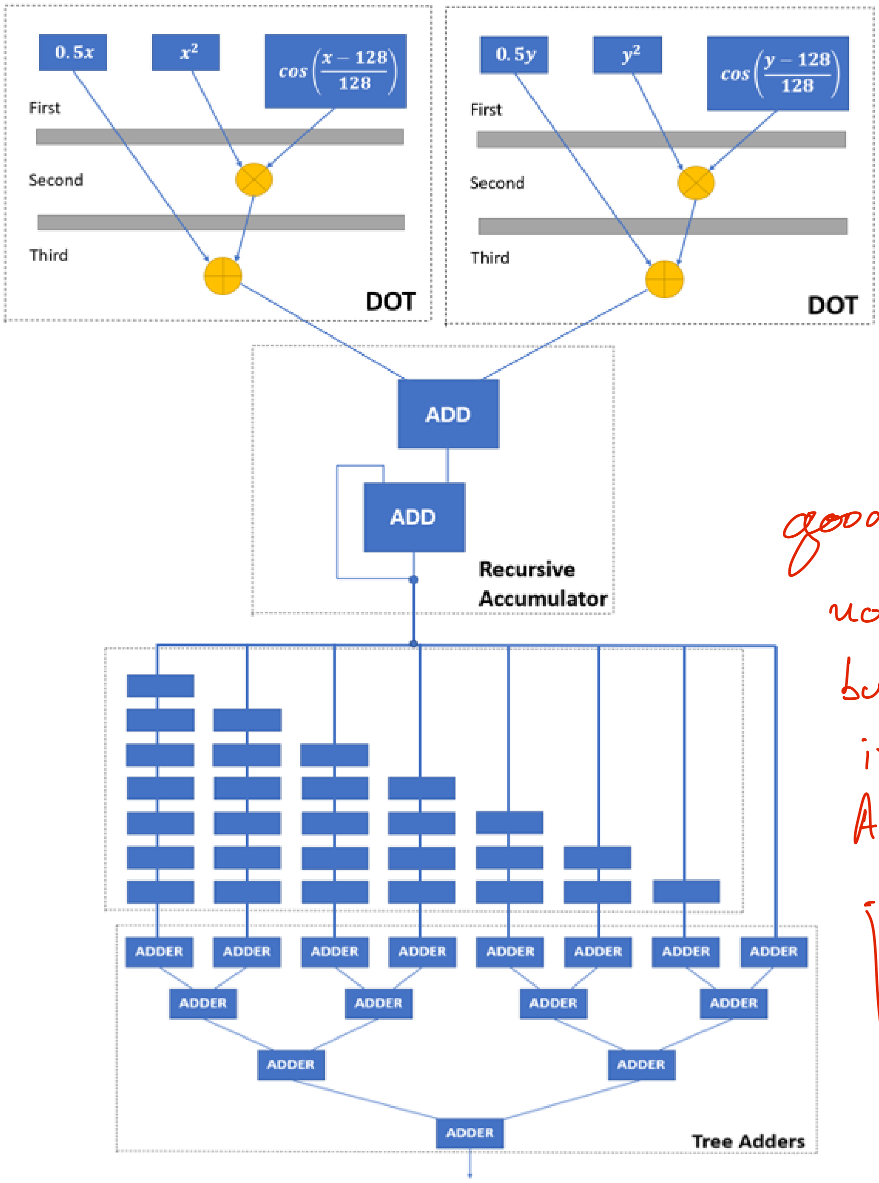


Figure 7: Hardware Calculation Architecture

$\sum_{i=0} x_{8i+3} \dots \sum_{i=0} x_{8i+7}$ at each cycle after that moment. Therefore, we also need a structure to add these 8 partial sum values to get the final result, and that's what part3 does.

Tree Structured adders

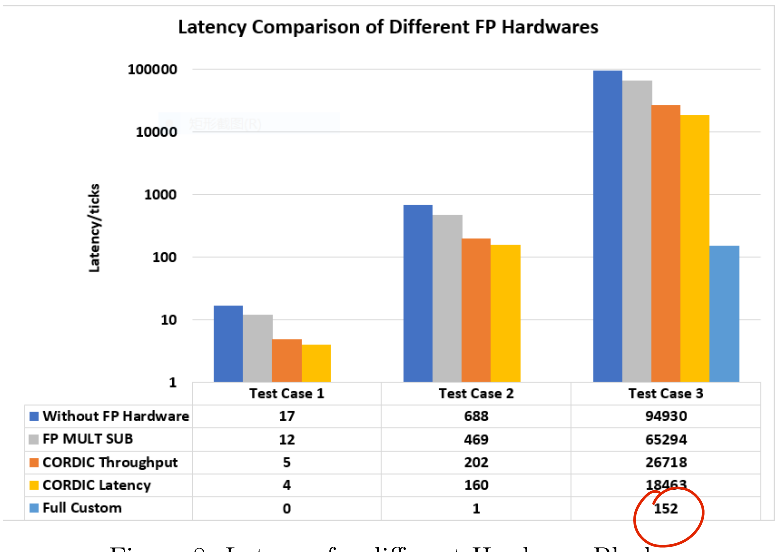
This part is mainly used to calculate the total sum. Since after the last value gets into the last recursive adder, we need to wait 8 cycles such that all the partial sums are calculated by this adder, and we need to add these 8 values together, so now this adder can be considered as a kind of FIFO, each cycle, one value would pomp out, and we need to store the first out values for the final sum. That's why the output of this adder is sent to 7 sets of 'shift registers', these sets will be delayed by 7 cycles, 6 cycles, 5 cycles, 4 cycles, 3 cycles, 2 cycles, 1 cycle, respectively. Seven outputs of this shift structure together with the direct connection of output of that recursive adder will be sent to the tree structured adders, such kind of structure should be the fastest but requires more resources. So, for the pipelined version, the result would be valid by 76 cycles after the last valid input comes in.

good that you noticed that, but the structure is very expensive. A better one would be when you use the same adder but you feed it with the last 7 cycles. 49 cycle on top it is a minimum penalty of pass.

0.5 Perfomance comparison

0.5.1 Latency

As shown in figure 8, when there's no FP hardware, the latency is quite high for all three test cases, and increases dramatically as the array size increases. The more hardware we added, the smaller the latency would be. When the CORDIC module is added, both throughput optimized version and latency optimized version, the speed increases a lot. This is mainly because to calculate cosine in software, usually we need to use taylor expansion or other recursive method, and requires lots of calculations. But to calculate it in hardware, only around 20 clock cycles is needed, which is much faster. It could also be seen that when we put all the computing elements into hardware and using the fully pipelined way to calculate, the speed increases significantly, especially for large arrays, for example, the latency of the test case three decreases from 94930 to only about 150 ticks.

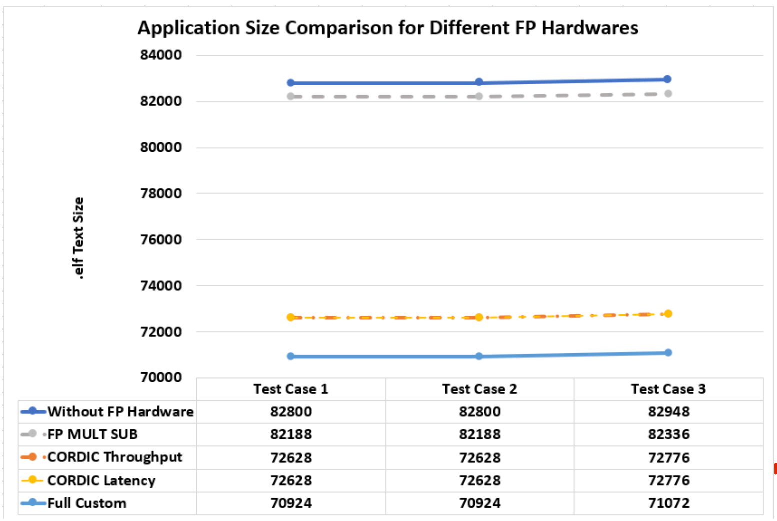


	Test Case 1	Test Case 2	Test Case 3
🟦 Without FP Hardware	17	688	94930
⬜ FP MULT SUB	12	469	65294
🟧 CORDIC Throughput	5	202	26718
🟨 CORDIC Latency	4	160	18463
🟦 Full Custom	0	1	152

Figure 8: Latency for different Hardware Blocks

0.5.2 Application size

As shown in figure 9, in terms of application size, if there's no hardware, then the application size should be the highest because everything needs to be calculated in software way, then there would be lots of instructions to do, which increase the general application size. The more hardware blocks we have, the smaller the application size should be, especially for our full custom version because the only instruction in software is the for loop and vector generation code. However, it can be seen that the difference is not that significant, this is perhaps because that although we are using the hardware to calculate, there should be corresponding firmware or drivers for those hardware blocks, which might also needs further code or instructions.

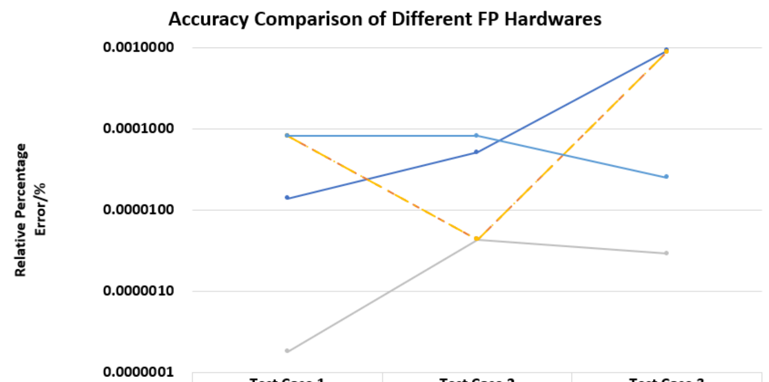


	Test Case 1	Test Case 2	Test Case 3
● Without FP Hardware	82800	82800	82948
● FP MULT SUB	82188	82188	82336
● CORDIC Throughput	72628	72628	72776
● CORDIC Latency	72628	72628	72776
● Full Custom	70924	70924	71072

Figure 9: Application size for different Hardware Blocks

0.5.3 Accuracy

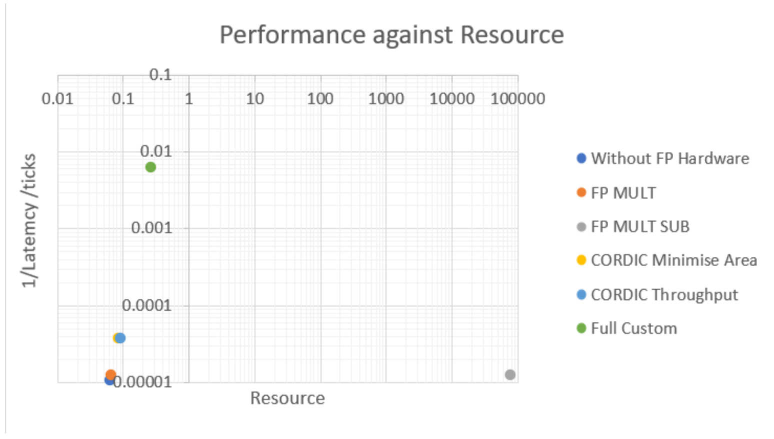
As shown in figure 10, when comparing with the result calculated by MATLAB, the overall accuracy of our full custom version is good, especially when the array size is small, this is because as array size increases, the summation may exceeds our representation range because there is a fixed-point block in CORDIC, which would reduce the accuracy.



	Test Case 1	Test Case 2	Test Case 3
Without FP Hardware	0.0000138	0.0000511	0.0009114
FP MULT SUB	0.0000002	0.0000043	0.0000029
CORDIC Throughput	0.0000817	0.0000043	0.0008781
CORDIC Latency	0.0000817	0.0000043	0.0008781
Full Custom	0.0000817	0.0000818	0.0000251

The fact that do not match raises question about the correctness of your design. However as you have changed the order that you have accumulate the data, such difference can be manifested! It would be nice to investigate this 4

Figure 10: Accuracy for different Hardware Blocks



$\boxed{A+}$

Figure 11: Performance versus Resource

0.6 Conclusion

In these tasks, we successfully designed a fully pipelined hardware structure to calculate the function, and increase the speed of the calculation significantly. However, there are still lots of limitations, for example, because the custom instruction could only transmit two 32 bits simultaneously, if we use this structure, the highest parallel processing element is 2. The structure of our design could be extended easily into highly parallel processing structure if we could find some way to extract multiple data at the same time, we could just add the same structure in parallel and adding more levels in tree adder. Now we are mainly using trick to treat the system that we are done and sending new data in, we could use DMA to pomp data in continuously for fully pipelining, and the CPU could also do other tasks at the same time. Besides, the custom instruction also has the external interface that can connected to other hardware blocks, and we could also use Avalon MM interface to directly connect to SDRAM controller to get the data. Furthermore, we could also use Dual Core to calculate the function for more data in parallel, but we need to add more logic to manipulate the communication. In general, the structure we achieved is a parallel programming structure that is pipelined completely, but we could improve it to be a fully SIMD structure in the future. ✓