School of Electronics and Computer Science

ELEC6234 Embedded Processors

Coursework Report

NISC Processor for Affine Transforms

April 9, 2024

William Fletcher

'wf2g20'

MEng Electronic Engineering

Martin Charlton

Abstract

Abstract

1 Introduction

The aim of this work is to design an RTL implementation of an application specific No-Instruction-Set-Computer (NISC). The processor performs an affine transform on a 2D input point (x_1, y_1) to produce a 2D output point (x_2, y_2). The inputs and outputs are represented as 8-bit signed integers. An affine transform consists of a stretch/rotation and shift (Equation 1) based on the matrix **A** and vector **b** stored in program memory. The coefficients of **A** are represented as signed fixed-point numbers with a 7-bit fractional part (allowing for squeezing). The coefficients of **b** are represented as 8-bit signed integers.

$$\begin{gathered} x_2 = x_1 a_{11} + y_1 a_{12} + b_0 \\ y_2 = x_1 a_{21} + y_1 a_{22} + b_1 \\ x, y, b \in \mathbb{Z}, a \in \mathbb{R} \end{gathered} \tag{1}$$

The final design was synthesised onto an Intel DE1-SoC FPGA, where the input point is entered through a handshaking protocol in accordance with the specification [1]. The design priority was to reduce the cost function (Equation 2). Utilising DSP blocks for embedded multipliers and a simplified NISC architecture reduced ALM usage [2]. Additionally, fixed-point rounding was implemented within the multiplier (Section 2.2) to increase result precision and a switch debouncer (Section 3) was designed so the full-speed 50 MHz FPGA clock could be used.

$$Cost = ALMs + 500max(0, DSP - 2) + 30KbitsRAM \tag{2}$$

Design Details Form

Total Cost:	ALMs:	Memory Bits:	Multipliers:
14.48	14	16	2

Control Word Format (11-bits)

`load` [10]	`branch` [9]	`raddr` [8]	`imm` [7:0]

Affine Transform Program (16 Control Words)

$$\mathbf{A} = \begin{bmatrix} 0.5 & -0.875 \\ -0.875 & 0.75 \end{bmatrix} \quad \mathbf{b} = \begin{bmatrix} 5 \\ 12 \end{bmatrix}$$

#	Hex	Operation
0	`405`	Load b_0 into \$0
1	`50C`	Load b_1 into \$1
2	`200`	Hold PC while SW[8] = 0
3	`040`	\$0 += a_{11}*SW[7:0]
4	`190`	\$1 += a_{21}*SW[7:0]
5	`190`	Allow sync reg write
6	`201`	Hold PC while SW[8] = 1
7	`200`	Hold PC while SW[8] = 0
8	`090`	\$0 += a_{12}*SW[7:0]
9	`160`	\$1 += a_{22}*SW[7:0]
10	`160`	Allow sync reg write
11	`201`	Hold PC while SW[8] = 1
12	`200`	Hold PC while SW[8] = 0, display \$0
13	`301`	Hold PC while SW[8] = 1, display \$1
14	`000`	None
15	`000`	None

Table 1: Final synthesised design cost, control word format and affine transform program.

2 Overall Design Architecture

Using a NISC architecture greatly simplifies the processor design. No instruction set architecture (ISA) must be specified and the processor design can be more closely coupled to the required application, reducing logic requirements. The main logic reduction comes from the removal of an instruction decoder to produce control signals. Moreover, many data-path elements can be simplified (e.g. the ALU) as the program directly applies control signals [2].

Inspecting Equation 1 it is clear the only required operations for an affine transform are multiply-accumulate and sum. The sum operation can be bypassed by loading the appropriate value of **b** into a register as the first calculation step. To support the direct loading of data into a register the processor includes a multiplexer on the register data input to select between the multiply-accumulate result and the immediate value from the control word.

The multiplier always multiplies a coefficient of **A** (signed fixed-point immediate value stored in program memory) by an input entered on the switches (signed integer). Its inputs are hard wired meaning no separate logic is required to handle data input and the required number of registers is reduced. Signed multiplication hardware can be used within a DSP block to reduce ALM usage (Section 2.2). To accumulate the multiplication result with the current value stored in the destination register an adder with fixed inputs is used (Section 2.3).

Each control word contains two control signals "`load`, `branch`" the register address "`raddr`" and a 8-bit immediate value "`imm`". Note no write control signal is required for the registers, instead

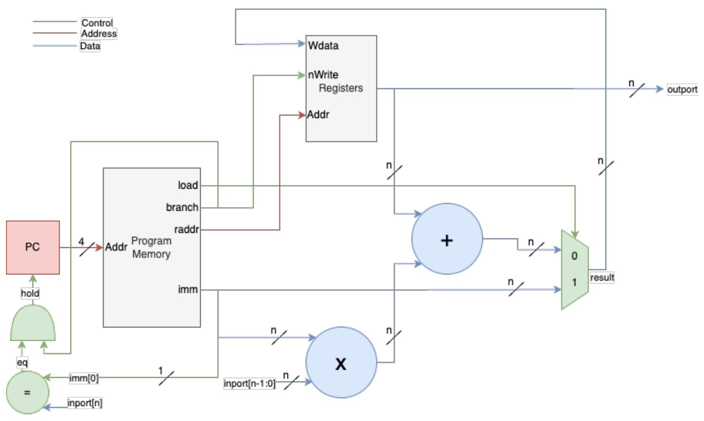

Figure 1: Data-path of the NISC processor. There is no instruction decoder as the control signals are accessed directly from program memory. Instead of an ALU the register data feeds into a multiply-accumulate module where the register write data is either assigned to the result or an immediate value based on the load signal. The program counter always increments its value by one unless the hold input is active, computed by comparing the handshake input with an immediate value. As there are only two registers the register address is a single bit. As the program memory is only 16 control words the program address is only 4-bits.

the inverse of the branch signal is used, i.e. a register is always written on a non-branch operation. Table 2 shows how the control signals can be used to select the mode of operation for the data-path each cycle.

`load`	`branch`	Operation
0	0	\$raddr = \$raddr + imm*inport[7:0]; #Mult-acc
0	1	Hold PC if inport[8] = imm[0], and display \$raddr
1	0	\$raddr = imm; #Load
1	1	Reserved

Table 2: Table showing how all required operations (for an affine transform) are achieved using combinations of the load and branch control signals.

Branches are only required to delay processor operation whilst waiting for the handshake input. Hence, only a single bit branch comparator is required to check if the value of the handshake signal and the LSB of the immediate value are equal (Section 2.4). The immediate value is otherwise unused by branch operations and removes the need for an extra control signal. When the values match and the control word has the branch control set then the program counter holts until the value of the handshake switch changes. Otherwise the program counter increments by one each clock cycle (Section 2.5).

There are only two independent outputs (x_2 and y_2) so only two registers are required within the processor to store the intermediate results. These registers are accessed independently so only one address and data line are required, with the addition of a write enable signal and write data (Section 2.1). Register output data is always displayed on the outport to be continually displayed during branch instructions (while the program counter is halted).

2.1 Registers

There are only two registers required for this design. To utilise memory blocks in synthesis the registers were made synchronous, meaning read and write operations only update on the rising edge of the clock. This reduces logic utilisation from 13 ALMs (for 2 8-bit asynchronous-read registers) down to 16 memory blocks; significantly cheaper in the cost function (Equation 2). Synchronous read and write functionality posed problems with the accumulate functionality as a register value must be read, used in an operation and written back within the same cycle. The solution was to include an additional operation with the branch control signal low (register write high) after each multiply-accumulate operation to allow write back of the previous cycles result.

Figure 2 shows a simulation waveform ..

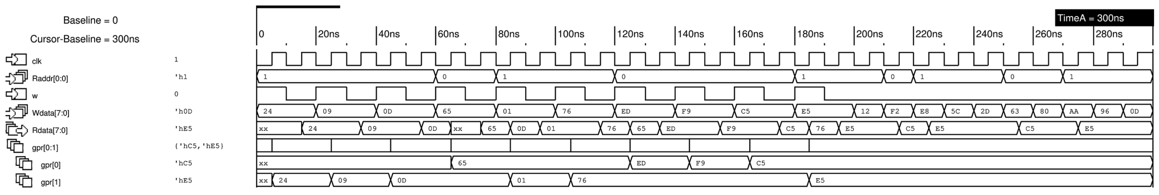

Figure 2: Simulation waveform for the register module with two registers. Each register write and read is synchronised to the clock, hence a write only propagates to the read output on the following clock cycle.

2.2 Multiplier

The multiplier module computes the signed product of a signed 8-bit fixed-point number (7-bit fraction) and a signed 8-bit integer. Standard signed multiplication hardware can be used to produce a signed 16-bit result where the bottom 7-bits are the fractional part. Therefore, extracting bits [14:7] gives the 8-bit signed integer result as required. The truncation of the fractional portion of the result leads to a round down for positive numbers and a round up for negative numbers. Hence, the expected error from multiplication is ± 1.

To improve the multiplier design rounding was added as listed below. If the most significant fractional bit of the full un-truncated result `mult` (weighted 0.5) is set, then one is added to the truncated result for correct rounding. This approach works for both positive and negative values due to 2's compliment bit weightings. Rounding reduces multiplication error to ± 0.5 and total error to ± 1 when two multiplications are used for each result of the affine transform.

```
if (mult[f-1] == 1'b1)
    result = tmp_result + 1'b1;
else
    result = tmp_result;
```

Figure 3 shows how the multiplier result aligns with the expected golden floating-point result for a random set of inputs. Note how in cases where a multiplier without rounding would simply truncate the value and always round down, the improved multiplier is able to round values up when the fractional part exceeds 0.5.

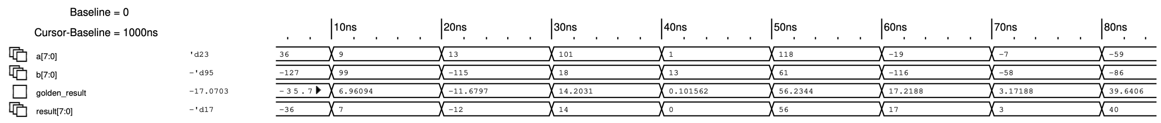

Figure 3: Simulation waveform showing how the multiplier module produces the same result as a golden reference within a ± 0.5 rounding tolerance. Note how rounding enables a lower error.

2.3 Adder

The adder module sums two 8-bit integers without any carry or overflow flag. As an optimisation to save ALM usage the adder uses a 16-bit multiplier within a DSP block to add by multiplication.

This uses 1 DSP block instead of 5 ALMs saving on total cost as 2 DSP blocks are given for free. To add by multiplication Equation 3 can be used where operands are concatenated into a single input and multiplied by a constant value. The output is then truncated to the 8 most significant bits. Note for synthesis the multiplier must be contained in a separate SV module to use a DSP block.

$$A + B = \{A, B\} * \{\{(n-1)\{1'b0\}\}, 1'b1, \{(n-1)\{1'b0\}\}, 1'b1\} \tag{3}$$

Figure 4 shows a simulation waveform of the adder using a multiplier. It is clear the adder result matches the golden result for an adder for a random set of inputs.

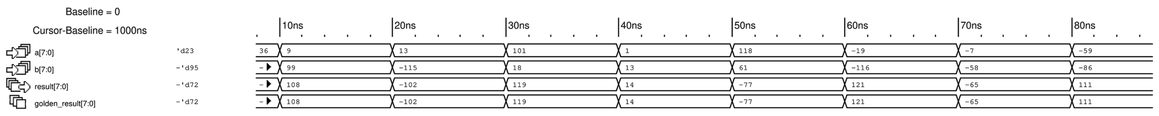
Baseline = 0
Cursor-Baseline = 1000ns
10ns
20ns
30ns
40ns
50ns
60ns
70ns
80ns
a[7:0] 'd23 36 9 13 101 1 118 -19 -7 -59
b[7:0] -'d95 - 99 -115 18 13 61 -116 -58 -86
result[7:0] -'d72 - 108 -102 119 14 -77 121 -65 111
golden_result[7:0] -'d72 - 108 -102 119 14 -77 121 -65 111

Figure 4: Simulation waveform showing how the adder module using a 16-bit multiplier produces the same result as a golden reference for random inputs (-64-63 to prevent overflow).

2.4 Branch Comparator

The branch comparator is only required for halting the program counter based on the handshake input (inport[8]). So the program counter can be halted on both high and low handshake inputs the handshake signal is compared with the LSB of the immediate value in the control word. Hence the branch comparator has fixed inputs and can be synthesised into a single XOR gate with a not gate on its output. The branch comparator produces the `eq` signal which only halts the program counter if the `branch` signal from the control word is also set. As the module is so simple no verification was needed.

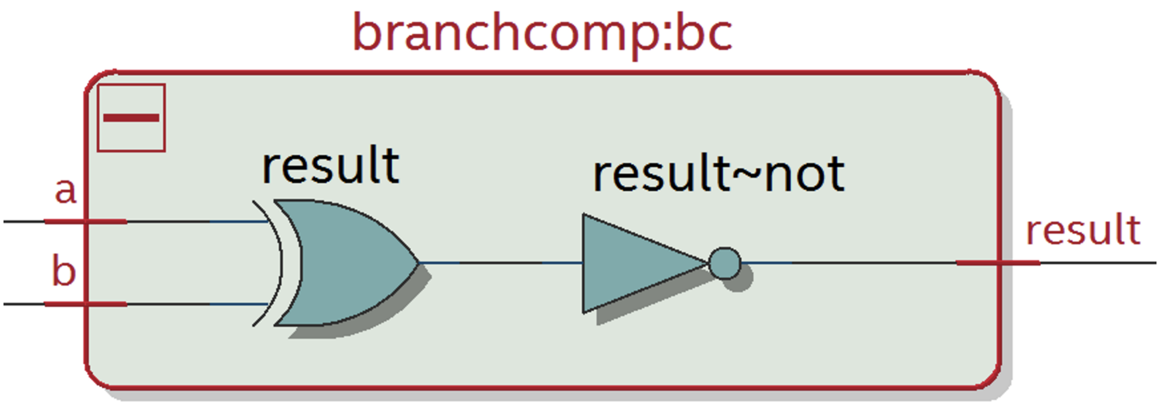

Figure 5: Synthesised RTL block diagram fro the branch comparator, from Quatus.

2.5 Program Counter

Using the control word format to create an example hex program for an affine transform defines the program length as less than 13 control words. Therefore, the full program can be addressed with 4-bits. The program counter always increments its value by one each cycle to access the next control word. Since no true branch operations are implemented the program counter requires no branch address. Instead for the program to loop null operations are inserted to make a 16 word long program where the program counter will wrap around to zero upon reaching the end. As mentioned in Section 2.4 the program counter has a hold input which will disable its incrementing when active.

Figure 6 shows a simulation waveform for the program counter where all addresses are incremented through (one each cycle). MOreover the hold input functions correctly; stalling the program counter when active.

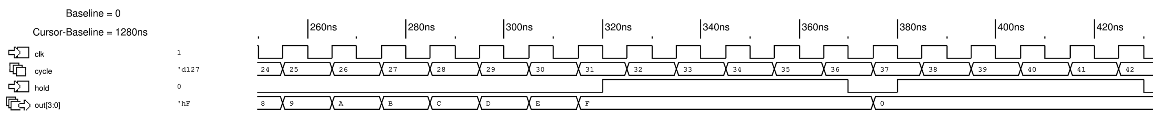

Figure 6: Simulation waveform for the program counter module, showing how the counter wraps-around and stops incrementing every cycle when the hold signal is high.

2.6 Program Memory

The program memory is ROM to hold 16 control words (the maximum length of a program). To be synthesised in FPGA memory blocks the program memory must be made synchronous. However this is not possible with the branch implementation. When a branch operation is loaded the program counter will have already incremented on the previous cycle before the the hold signal is active, meaning an erroneous operation will be executed. This could be avoided by filling operations after a branch will a null operation, but this would require increasing program size (program address to 5-bits).HOW MUCH COULD BE SAVEED.

The basic operation of the program memory uses the "\$readmemh()" function to load data from a hex file into ALM LUTs. This data can be asynchronously read by providing the correct address.

2.7 Processor

The top level processor module connects all constituent modules and includes the result multiplexer. For verification a golden reference implements Equations 1 using floating-point values. The stimulus toggled the `readyIn` input and provided/extracted the correct data corresponding to the specification. (x_1, y_1) point was randomly generated using `63-$urandom_range(127)` to avoid overflow in the adder [3] and tested for two sets of transform coefficients. The design output was checked against the golden reference to identify any errors.

Simulation across a range of input sample values was successful proving the correct functionality. It was clear that rounding errors were causing deviations up to ±1 (explained in Section 2.2). These errors are intrinsic to the design, so a function to give ±1 tolerance was used in future simulations. Figure 7 shows an example waveform for a single transform.

3 FPGA Implementation

1-2 pages

Pin bindings were added to the RTL based on the DE1-SoC data-sheet [4] so the switches (SW[9:0]) act as input and the LEDs (LED[7:0]) act as output. The FPGS's internal 50MHz clock was used for the synchronous design. RTL was synthesised using Quatus Prime and download onto the DE1-SoC FPGA. TODO.

3.1 Timing Analysis

[5].

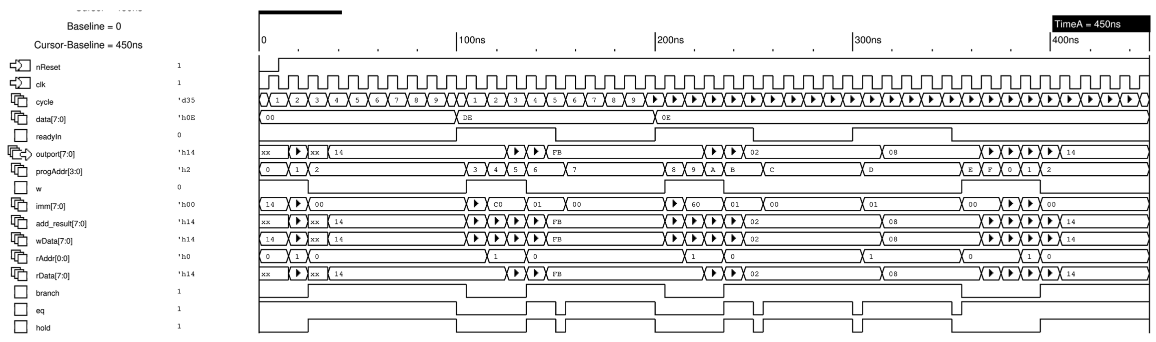

Figure 7: Simulation waveforms for a single affine transform sequence. $x_1 = -34(DE), y_1 = 14(0E)$ giving the output $x_2 = 2, y_2 = 8$ based on the value of outport at the corresponding readyIn signal values. The transform output aligns with expected values from A, b coefficients.

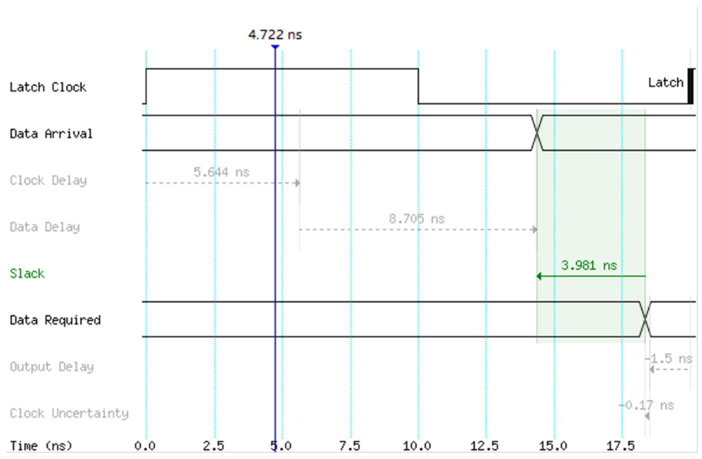

Figure 8: TimeQuest result showing the worst case clock slack for all signals at 50 MHz is 3.98 ns, meaning design meets all timing constraints. Maximum operating frequency is 64 MHz.

3.2 Test Vectors

To test the FPGA implementation test vectors consisting of input values for x_1, y_1 and expected output values for x_2, y_2 given transform coefficients were used. These test vectors were produced from simulation where the expected values correspond to the values produced by the verified design. Values from calculated equations cannot be used as a reference because some error is inherent within the design (see Section 2.7). The vectors aim to test positive and negative values.

To input a test vector on the FPGA the specified pseudo-code was followed. This consists of entering data in order x_1, y_1 on each rising edge of the handshake signal (SW8). Subsequent handshake cycles display the results.

All test vectors passed showing the expected functionality. Input data is displayed on the LEDs after the handshake and the sequence may be repeated once the final result is displayed.

4 Conclusions

The RTL was verified in Xcelium against a golden reference and met the full specification. Within simulation ± 2 error was observed in the output data due to rounding errors from using fixed-point numbers. This was improved to ± 1 using fixed-point rounding in the multiplication module. After downloading onto an FPGA specific test vectors prove functionality. Further to the specification a

Program		Test			
A	**b**	x_1	y_1	x_2	y_2
$\begin{bmatrix} 0.75 & 0.5 \\ -0.5 & 0.75 \end{bmatrix}$	$\begin{bmatrix} 20 \\ -20 \end{bmatrix}$	49: 00110001	-34: 11011110	40: 00101000	-69: 10111011
		29: 00011101	32: 00100000	58: 00111010	-10: 11110110
		-37: 11011011	-28: 11100100	-22: 11101010	-22: 11101010
		-50: 11001110	14: 00001110	-10: 11110110	16: 00010000
$\begin{bmatrix} 0.5 & -0.875 \\ -0.875 & 0.75 \end{bmatrix}$	$\begin{bmatrix} 5 \\ 12 \end{bmatrix}$	49: 00110001	-34: 11011110	60: 00111100	-56: 11001000
		29: 00011101	32: 00100000	-8: 11111000	11: 00001011
		-37: 11011011	-28: 11100100	12: 00001100	23: 00010111
		-50: 11001110	14: 00001110	-32: 11100000	67: 01000011

Table 3: FPGA test vectors showing inputs and corresponding expected outputs (from simulation) in binary format.

switch debounce module was implemented, allowing the full 50 MHz clock to be used.

The design has intrinsic issues with handling ALU overflow. The output data could be given as a 9-bit value (with 9-bit bus width) to prevent overflow. Furthermore, inputting values using the FPGA switches is slow and laborious. Instead, values could be inputted from RAM or a UART connection to allow for faster calculation times. Expanding the FFT butterfly could involve increasing precision or offering the ability to change the FFT length. HOW GOOD IS THE SIZE, I GOT IT PRETY SMALL BY CONSTRAINING THE DESIGN AND FOCUSINGG ON OPTIMISATIONS!!!

References

[1] T. J. Kazmierski, *Elec6234 coursework instructions*, Feb. 2024.

[2] B. Gorjiara, M. Reshadi, and D. Gajski, "Designing a custom architecture for dct using nisc technology," Feb. 2006, p. 2, ISBN: 0-7803-9451-8. DOI: `10.1109/ASPDAC.2006.1594664`.

[3] Apr. 2020. [Online]. Available: `https://verificationguide.com/systemverilog/randomize-variable-in-systemverilog/#urandom_range`.

[4] T. Technologies Inc., *De1-soc board datasheet*, Nov. 2014. [Online]. Available: `https://secure.ecs.soton.ac.uk/notes/ellabs/2/m4/files/DE1-SoC.pdf`.

[5] A. Corp, *Using TimeQuest Timing Analyzer*. 2014, p. 16.