ELEC 473 ASSIGNMENT 4 - NIOS II CUSTOM INSTRUCTIONS

Ben Hague

MAY 11, 2020

UNIVERSITY OF LIVERPOOL

Ben Hague

Contents

1. Count Leading Zeros Instruction (CLZ)

The methodology behind this instruction is based on binary search. Whilst this approach makes little difference on a 32 bit number compared to a shifting counter, it leads to a much more scalable approach with minimal speed increase when a larger bus is used. This is widely recognised as binary search has a upper bound of O(logn) as opposed to a linear search with a upper bound of O(n).

The ASM for this instruction is shown below

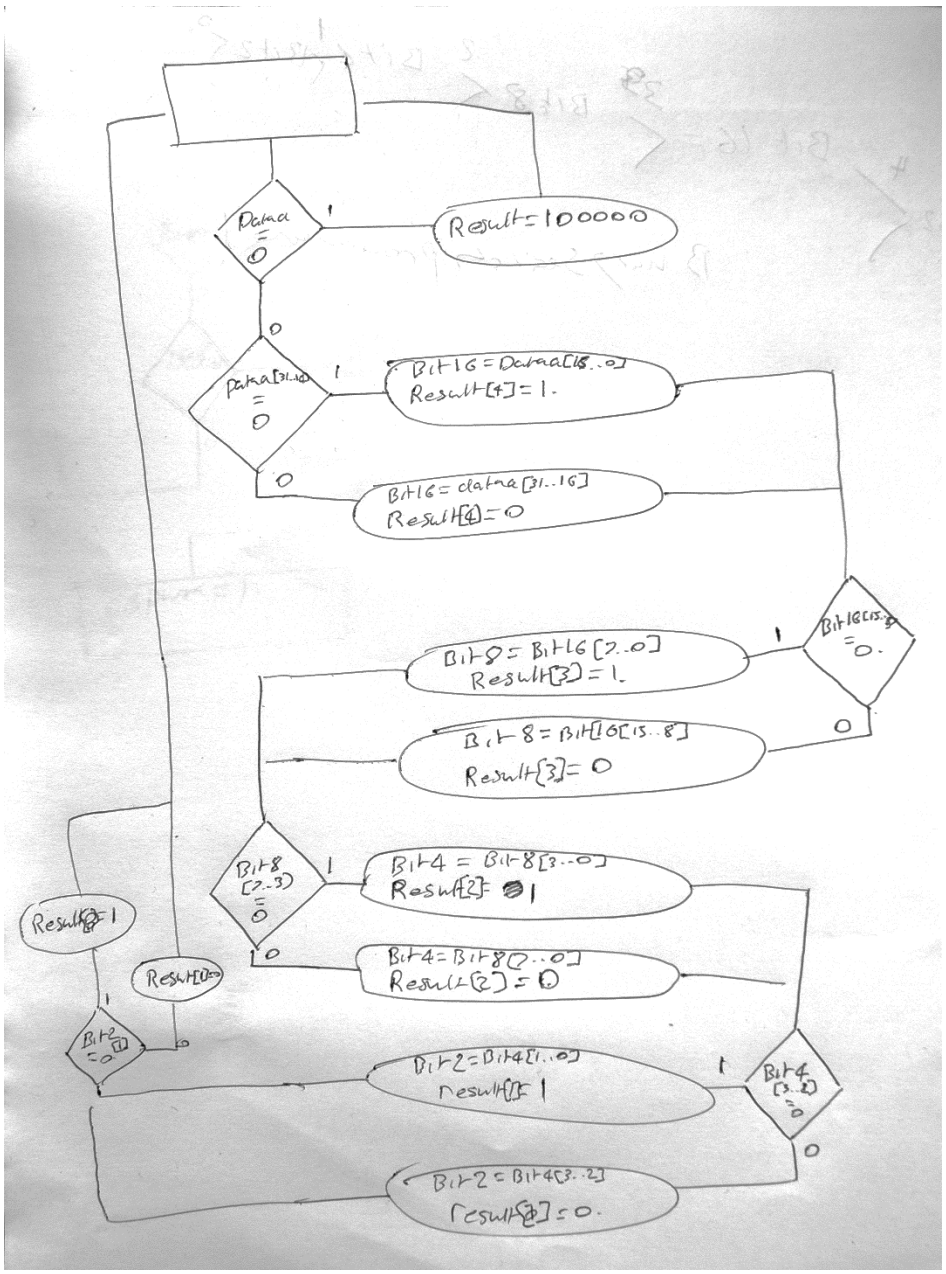

Figure 1, Count Leading Zeros ASM

This has been implemented in VERILOG as shown below

```
// Hardware binary search solution for count leading zeros
// It is easier to imagine this as looking for the last zero
// due to the way the conditional logic works
module ci_clz(
	input [31:0] dataa,
	output [31:0] result
	);
	// Wires for seperating out code, One wire per split, (a multiplexer is used to minimise resources per branch
	wire [31:0] bit32;
	wire [15:0] bit16;
	wire [7:0] bit8;
	wire [3:0] bit4;
	wire [1:0] bit2;

	assign bit32 = dataa;
	// Temparary result for determining 32 zeros when there are no ones to hone in on
	wire [5:0] tmpResult;
	// Set the unused result bits to zero
	assign result[31:6] = 0;
	// decide if there is a one to search for, if not output 32 ie 32 zeros else output the tmpresult
	assign result[5:0] = tmpResult[5] ? 6'b100000 : tmpResult;

	assign tmpResult[5] = bit32==0;

	// are there any ones in the upper 16 bits
	// select the half word with the first one
	assign tmpResult[4] = bit32[31:16] == 0;
	assign bit16 = tmpResult[4] ? bit32[15:0] : bit32[31:16];

	// from the selected half word
	// select the byte with the first one
	assign tmpResult[3] = bit16[15:8] == 0;
	assign bit8 = tmpResult[3] ? bit16[7:0] : bit16[15:8];

	// from the selected byte
	// select the nibble with the first one
	assign tmpResult[2] = bit8[7:4] == 0;
	assign bit4 = tmpResult[2] ? bit8[3:0] : bit8[7:4];

	// from the selected nibble
	// select the pair with the first one
	assign tmpResult[1] = bit4[3:2] == 0;
	assign bit2 = tmpResult[1] ? bit4[1:0] : bit4[3:2];

	// You already know there is a one and you havnt found it
	// Therefore you only need to check the upper bit
	assign tmpResult[0] = bit2[1] == 0;
endmodule
```

Code Section 1, Count leading Zeros Verilog module

Testing

The module has then been tested to ensure it compiles and functions as expected. Please note the lower radix is in denary notation.

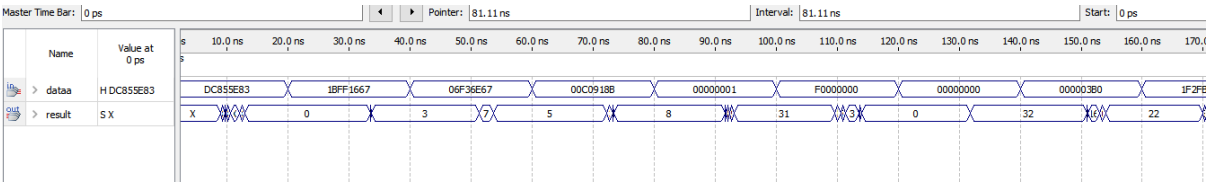

Figure 2, Testing Count leading zeros module

We can observe at 80-140ns that the outcome is as expected in fringe cases as well as in general operation.

2. Testing in C++

The Instruction has then further been integrated into the NIOSII system and tested with the following C program

```
#include "sys/alt_stdio.h"
#include "system.h"

int main()
{
	unsigned A = 0x00;
	unsigned Z = 0x00;
	Z = ALT_CI_CI_CLZ_0(A);
	alt_printf("Count Leading zeros of %x = %x \n(Lowest Boundary test)\n\n", A, Z);

	A = 0x00000001;
	Z = ALT_CI_CI_CLZ_0(A);
	alt_printf("Count Leading zeros of %x = %x \n(Lowest Boundary test 2)\n\n", A, Z);

	A = 0x80000000;
	Z = ALT_CI_CI_CLZ_0(A);
	alt_printf("Count Leading zeros of %x = %x \n(Upper Boundary Test)\n\n", A, Z);

	A = 0xFFFFFFFF;
	Z = ALT_CI_CI_CLZ_0(A);
	alt_printf("Count Leading zeros of %x = %x \n(Max test)\n\n", A, Z);


	A = 0x00012581;
	Z = ALT_CI_CI_CLZ_0(A);
	alt_printf("Count Leading zeros of %x = %x \n(Random chosen number)\n\n", A, Z);



	/* Event loop never exits. */
```

```
        while (1);

        return 0;
}
```

Code Section 2, C Testing code for Custom Instruction

3. Testing Results

The result of the simulation is shown below. The bounds of the instruction are tested and the result is as expected.

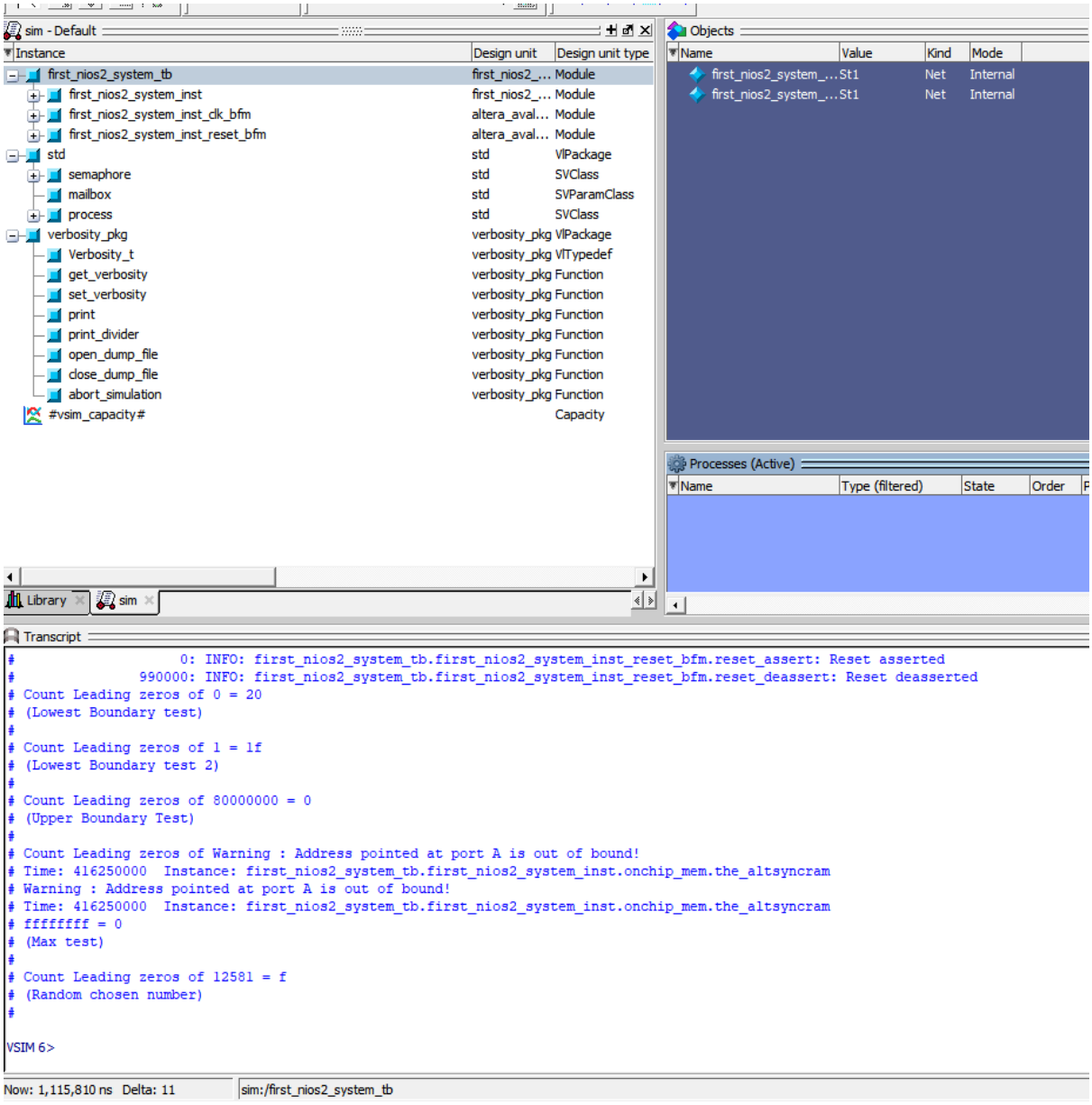

Figure 3, Testing Count leading zeros as custom NIOSII instruction

4. Speed Comparison

To test the speed of the execution we must ensure that we fully evaluate the time taken for multiple instructions with identical overhead. A software implementation of CLZ has been written on the same methodology:

```
int software_CLZ(int x)
{
	int result = 0;
	if (x==0)
		return 32;
	else{
		if (!(x & 0xffff0000)) { // Check Upper 16 Bits, if 0 than shift
			x <<= 16;
			result += 16;
		}
		if (!(x & 0xff000000)) { // Check Check upper 8 Bits (Post Shift) then shift
			x <<= 8;
			result += 8;
		}
		if (!(x & 0xf0000000)) { // Check Check upper 4 Bits (Post Shift) then shift
			x <<= 4;
			result += 4;
		}
		if (!(x & 0xc0000000)) { // Check Check upper 2 Bits (Post Shift) then shift
			x <<= 2;
			result += 2;
		}
		if (!(x & 0x80000000)) { // Check Check upper Bit (Post Shift)
			x <<= 1;
			result += 1;
		}
	}
	return result;
}
```

To test the timing of each implementation an LED has been turned on for the duration of the instruction.

This code is shown below, where the intended test is uncommented before operation

```
#include "sys/alt_stdio.h"
#include "system.h"
#include "sys/alt_irq.h"
#include "altera_avalon_pio_regs.h"
#include "altera_avalon_timer_regs.h"


int main()
{
	//Register ISR for timer event
```

```
        unsigned A = 0x0;
        unsigned Z = 0x00;
        unsigned Count = 0;
        while (1){
                A = Count%32;
                IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, 0x01);
                //Z = ALT_CI_CI_CLZ_0(1<<A);
                //Z = software_CLZ(1<<A);
                IOWR_ALTERA_AVALON_PIO_DATA(LED_PIO_BASE, 0x00);
                Count++;
                alt_printf("Execution Number %x, Input = %x, Result = %x \n", Count,A, Z);

        };

        return 0;
}
```

For the hardware implementation we can observe a constant calculation time of 40,000ns. As shown between the two cursors below.

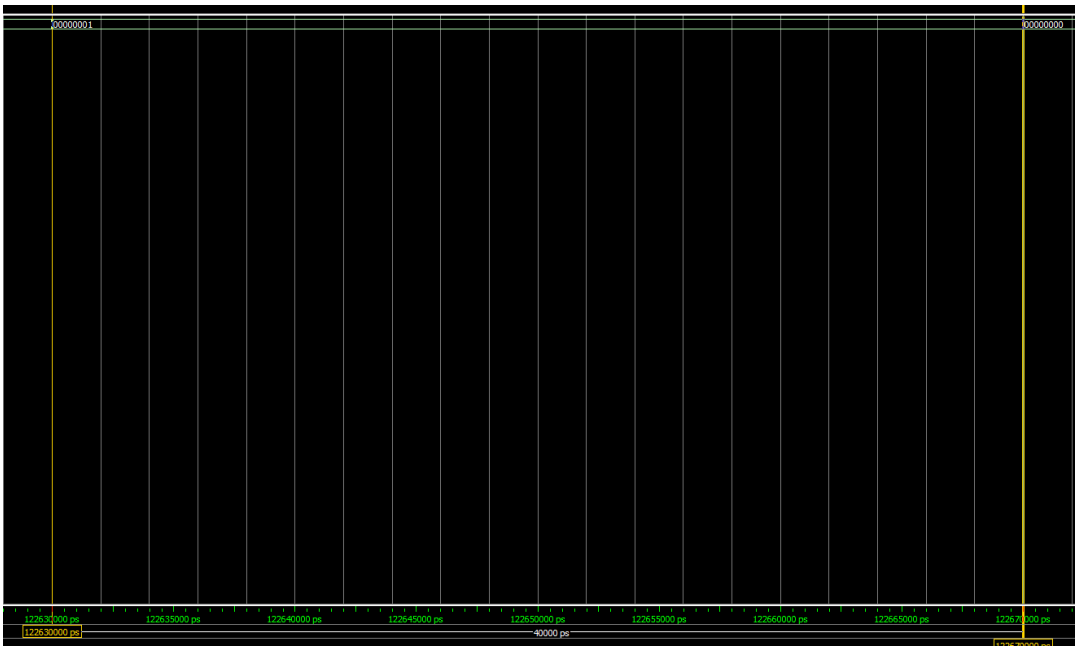

Figure 4, Hardware CLZ implementation

This differs from the software implementation where the time taken for execution is related to the number of leading zeros. This can vary from 20,000ps to 1240,000 depending on the size of the input signal. This is shown below:

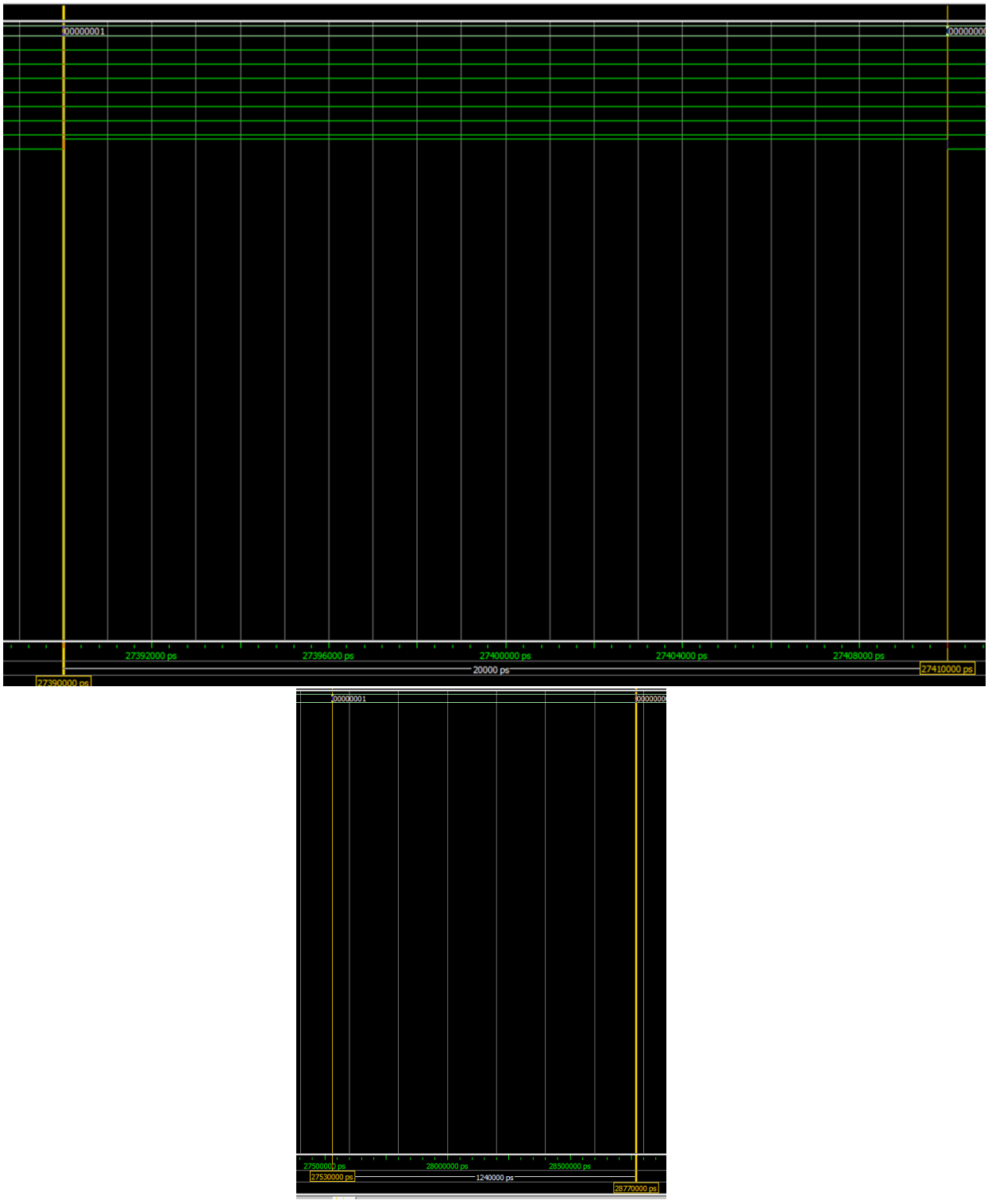

Figure 5, Software CLZ implementation (upper 0x01, Lower 0xFFFFFFFF)

5. Evaluation

This is a massive variance affected mostly by how many of the if statements are needed to be executed. We can see how this time would be less depending on the number of bits of data the system used. In a 32 Bit system it would be faster to maintain a hardware CLZ implementation. In an lower bit width CPU such as an 8 bit it would make little difference to speed of the CLZ instruction.

This variance leaves a more unpredictable nature to the time taken to complete an instruction and therefore damages the repeatability of the instruction result.

Both options provide Adequate implementations for most use cases however the low component cost (<50 logic elements) provides a clear advantage for most general purpose applications to include instructions such as CLZ