Laboratory Exercise 2: Design and Performance Analysis of a 32-bit Array Multiplier

Brandon Diamond Andrew Chin Leo Meyerovich
brandon_diamond@brown.edu achin@brown.edu leo_meyerovich@brown.edu

March 23, 2007

1 Introduction

When designing any practical system, the computer architect faces a great number of important design choices. Of these decisions, the structure and implementation of the ALU is of particular significance; given limited resources, should all effort go toward optimizing the adder, or should some of these resources be spent implementing an in-hardware multiply? Does a blisteringly-fast adder outweigh the cost of a software multiply, or is it better to slow down the adder in order to budget for a multiply operation in hardware? As part of this lab, we have been asked to confront these design decisions, and to explore each option in depth.

To do so, we have explored three different implementations of the multiply operation using the Nios II processor and a DE2 FPGA board. The first implementation exists entirely in software and the second is synthesized in hardware; the final multiplier is implemented using custom hardware described using the Verilog hardware description language (HDL). After completing each exploration, we recorded the number of cycles required to compute a multiplication so as to allow for the comparison of each of these three design decisions.

This laboratory write-up describes the procedures utilized and the results that were observed; relevant analysis is subsequently presented. Lastly, a complete listing of code can be found at the end of this document in appendix A (page 4).

2 Procedure

2.1 Equipment

- Software:
  1. Altera Nios II soft processor
  2. Quartus II CAD system
  3. Altera SOPC Builder and Programmer (accessed through Quartus II)
  4. Verilog hardware description language
- Hardware:
  1. Altera Development and Education (DE2) board
  2. Cyclone II EP2C35F672C6 with EPCS16 16-Mbit serial configuration device

2.2 Setup

- Initial

  1. We began by creating a new Quartus II project utilizing the Cyclone II EP2C35F672C6 FPGA aboard our DE2 development board.
  2. The Nios II soft processor was instantiated via the SOPC (software-on-a-programmable-chip) utility within Quartus II for our FPGA.
  3. Initially, as we expected our algorithm to run at full clock-speed, we configured the clock to run at 50 Mhz.
  4. The appropriate amount of memory (30 Kbytes, 32 bit width) was specified using the On-Chip Memory Configuration wizard.
  5. A USB cable was used to connect the DE2 development board to our Microsoft Windows workstation.
  6. Subsequently, the JTAG UART interface was configured appropriately within Quartus II.
  7. A performance counter was added to our project, and the device was configured to auto-assign base addresses.
  8. Finally, the complete system was generated over the course of several minutes.

- Configuring Nios II

  1. The Verilog code provided was used to configure the Nios II's pins appropriately.
  2. This was achieved by importing the TA provided 'DE2_pin_assignments.csv' file into our project.

- Programming Setup

  1. Upon running the Programmer within the Quartus II environment, the USB connection was configured to utilize USB-Blaster.
  2. Our project configuration file was added to the Programmer so ensure that our configuration was utilized appropriately.
  3. Finally, the FPGA was configured with the above settings in place.
  4. With the device configured, the Nios II EDS integrated development environment was started and configured to use our project's .ptf file.
  5. Throughout the course of development, code was written within this IDE, built for our configuration, and run as Nios II Hardware.

2.3 Approach

Initially, we set out to work through the fairly length configuration of our DE2 board and programming environment. A good deal of time was spent working to build our project appropriately and to set up the integrated development environment so as to allow us to begin writing code. We were very careful to test each step of the setup inclemently by confirming that the project was configured appropriately and, finally, by running the TA provided test code for the built-in multiply operation.

After we were certain that our project had been appropriately configured and that we understood how our system was working, we set out to implement the software multiply unit. As each member of our team is fairly familiar with the C programming language, we were able to proceed in a fairly straightforward manner. Following a brief discussion on how we intended to implement the multiply, it was a simple matter of expressing the operation in C, and executing the code on the soft processor. Fortunately, having spent so much time configuring our project earlier, we were able to get our code running fairly quickly.

Having completed the first two parts of the assignment, we set out to design our custom multiply units and adders. We proceeded carefully, first devising an algorithm, and then discovering how to express our algorithm in Verilog. While this approach did not enable us to acquire a good understanding of the Verilog syntax and semantics until several iterations into development, it did allow us to jump right in and work fairly quickly. Fortunately, we were able to implement the full adder and half adder after one or two tries; we utilized the Verilog In One Day tutorial[1] rather thoroughly in order to do this.

After incrementally testing our one-bit adders and ensuring that these worked as expected in the simulator, we proceeded to devise our array multiplier. The approach taken here was also incremental; we began by whiteboarding until we were able to understand the array adder depicted in the lecture slides. We then worked to extrapolate a few of the details that were glazed over in lecture. Finally, we devised a set of special-cases, an inductive definition, and a collection of base cases for our array multiplier algorithm.

Subsequently, we spent some time perusing the Verilog tutorial in order to learn how to express our algorithm most efficiently. Following this research, we set out to implement each special case, followed by the rows and then columns of the multiplier using our inductive definition. Subsequently, we tied this into our code using a small wrapper module, and proceeded to test. We used the debug view thoroughly to first ensure that our special cases were correct, then each of our columns, and finally the tricky bottom row. Any impedance was resolved, and it wasn't long until our multiplier was working appropriately. For more details on the algorithm and method used, please see below.

After testing a variety of simple multiplies by 0, 1, and so forth, we proceeded to test that our array multiplier handled overflow appropriately. By multiplying $2^{32} - 1$ by powers of two, we were able to ensure that the bits were shifting to the left appropriately, and therefore, that overflow was being handled sanely. Each of our datastructures were examined by hand to confirm that our gates were arranged appropriately.

Once we were fairly confident in our circuit within the simulator, we followed the aforementioned procedure to run the code on the chip. We again tried our test cases in the C framework using the Nios II EDS IDE and obtained the same results. Finally, we ran the TA test code and confirmed that the same final value was obtained as our two prior test runs, confirming that our multiplier was fully functional.

3 Experimental Results

For each part of the lab, we compiled and ran test code (see appendix A.1 on page 4), using the three difference multiplier units. The test code simply performed some predetermined multiplications while counting the number of cycles the test code took to run. The results for each part:

In part 1 (software multiply), we found that the answer was 61556835, computed in 1923904024 cycles.

In part 2 (hardware multiply), we found that the answer was 61556835, computed in 5213 cycles.

In part 3 (our custom hardware multiply), we found that the answer was 61556835, computed in 13836 cycles.

These results line up very well with our expectations. The slowest part was clearly part 1, where the multiply operation was synthisized in software by doing a series of additions. We expected to find that this method was the slowest because of number of instructions involved in a software implementation; not only are there a bunch of addition instructions, but branching and testing is also involved. The fastest part was part 2, where we ran the test code on hardware with a built-in multiply instruction. Again, we expected this to be much faster than part 1, and were not surprised to find that it was faster than our own hardware implementation. The hardware implementation for the multiplication instruction was presumably designed by professional engineers, so it would be reasonable to assume that a good deal of time had been spent optimizing the hardware multiplier (thus being faster than our non-optimized hardware). It's important to note how big the time

[1] http://www.asic-world.com/verilog/veritut.html

difference between software multiply in hardware multiply is. In the case of these test results, the speed to compute the multiply in part 2 is nearly 370000 times faster than the software multiply.

Our own hardware implementation was about twice as slow as the built-in hardware multiplier, but also significantly faster than the software multiplier (by about a factor of 140000). Again, this is inline with our expectations. The important thing to note here is that even a simple hardware multiplier (such as ours) is much faster than doing a multiplication in software.

As we were unaware of exactly how the performance counter kept cycle counts, it may be that our test code ran a bit slower than if we had run it without the performance counter. However, given that we are looking at the relative speeds of the three multiplier versions, we can safely ignore any potential slowdowns due to the performance counter.

Again, it is clear that a multiply unit implemented in hardware is drastically faster than a multiply unit implemented in software.

4 Conclusion

From our experiments with the three multiply operations — and having implemented an array multiplier at the gate level — we were able to gain a great deal of insight into the various tradeoffs involved when considering a software multiply unit, a synthesized multiply unit, and a custom multiply unit for a given processor architecture. While this choice is largely application dependent, there are a number of observations that can be made with respect to our expectations as engineers as well as our analysis of each device's performance.

Our results clearly show that a multiplier implemented in hardware is, by several orders of magnitude, faster than a software multiplication unit. Because of this, it becomes obvious why a lot of hardware has their own multiply instruction, instead of leaving it up to the software.

Certainly, despite our results, there are a number of complex factors that influence which design an architect ultimately selects over another. For example, a cheap embedded processor might be better suited to a simple, minimal architecture, leaving complex operations such as square root and multiplication to be implemented in code, whereas an expensive DSP might benefit most from lighting-fast multiplication and vector operations that are justified by the device's target audience and application. In fact, given the performance boost we saw from using a hardware multiplier, it would seem that one must be used in a DSP in order to do real-time signal processing.

Thus, the architect would skimp on the hardware for the embedded processor, while he might pack tons and tons of functionality into the complex digital signal processor.

A Code

A.1 Test code

A.1.1 Code of part 1 and part 2

```

//This is the application for part 1 and part 2.
//Rename this to main.cc
#include <stdio.h>
#include "system.h"
#include "altera_avalon_performance_counter.h"


#define H_LENGTH 226
#define X_LENGTH 226
#define LENGTH 226


int h[] = {638,451,45,787,365,138,657,497,451,846,262,473,851,426,
1010,938,236,192,629,753,935,200,906,525,499,237,744,279,861,688,
238,66,453,351,619,388,914,946,112,869,331,708,219,7,1008,1008,
```

```
658,840,761,240,246,159,987,452,175,387,408,425,647,687,731,553,
430,900,634,997,37,688,882,784,125,998,801,230,666,947,342,553,
99,579,582,854,119,283,160,713,179,738,507,867,283,543,477,784,
903,138,520,641,87,716,1021,576,750,425,504,578,893,755,760,659,
946,781,330,531,244,477,355,357,126,254,126,363,945,73,881,260,
561,885,278,179,538,243,737,417,125,491,440,658,925,217,205,125,
168,516,794,179,720,918,459,163,133,103,77,576,81,910,456,474,
877,633,160,518,534,257,884,662,171,29,436,148,542,40,236,4,370,
572,343,229,461,527,515,148,571,923,713,652,383,735,803,539,94,
706,128,1019,167,440,527,594,627,562,1019,224,270,464,302,475,
762,754,239,739,926,391,642,316,797,416,63,96,111,702,794,575,
976,10,16,752};

int x[] = {876,727,355,891,403,270,946,551,151,600,680,575,369,864,
680,705,30,420,1006,647,817,603,982,752,884,172,605,991,293,658,
463,571,729,499,1014,635,679,702,564,457,693,505,334,889,716,674,
699,302,551,175,1021,385,446,49,910,158,611,366,787,217,252,656,
261,540,572,902,838,936,165,449,1002,172,705,837,985,615,232,737,
585,404,389,199,2,555,858,932,759,415,200,985,470,629,726,298,
536,835,881,1013,4,154,270,937,867,708,318,773,656,856,398,976,
144,883,961,453,240,479,257,610,500,943,173,615,161,587,30,472,
422,525,147,389,568,496,385,332,759,392,235,994,776,584,626,526,
758,1008,31,486,8,907,683,773,702,44,824,805,570,492,251,224,50,
184,834,911,774,363,418,994,231,486,672,118,408,662,431,528,748,
311,386,790,644,791,850,382,199,167,287,123,816,943,32,21,342,
678,703,625,82,945,799,183,130,847,523,14,570,471,245,985,568,
256,256,39,312,49,718,459,875,71,319,908,359,888,109,493,804,
445,324,70};


int testFunction12();

int main_parts12()
{


PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);


int result = 0;
result = testFunction();

PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);

alt_u64 time = 0;
time = perf_get_section_time((void*)PERFORMANCE_COUNTER_0_BASE, 1);
printf("That was fun, the answer is %d, it took %d cycles\n", result, time);



  /* Event loop never exits. */
  while (1);

  return 0;
}


int testFunction(){

    int i;
    int j;
    int y = 0;

    PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE,1);
    for ( i=0; i < LENGTH; i++) {

```

```
        y = y + h[i]*x[i];
    }

    PERF_END(PERFORMANCE_COUNTER_0_BASE,1);

    return y;
}
```

A.1.2 Code for part 3

```

//This is the application that uses your multiplier.
//Make sure you rename this to main.cc.
#include <stdio.h>
#include "system.h"
#include "altera_avalon_performance_counter.h"
//#include "conv.h"

#define H_LENGTH 226
#define X_LENGTH 226
#define LENGTH 226

int h[] = {638,451,45,787,365,138,657,497,451,846,262,473,851,426,
1010,938,236,192,629,753,935,200,906,525,499,237,744,279,861,688,
238,66,453,351,619,388,914,946,112,869,331,708,219,7,1008,1008,
658,840,761,240,246,159,987,452,175,387,408,425,647,687,731,553,
430,900,634,997,37,688,882,784,125,998,801,230,666,947,342,553,
99,579,582,854,119,283,160,713,179,738,507,867,283,543,477,784,
903,138,520,641,87,716,1021,576,750,425,504,578,893,755,760,659,
946,781,330,531,244,477,355,357,126,254,126,363,945,73,881,260,
561,885,278,179,538,243,737,417,125,491,440,658,925,217,205,125,
168,516,794,179,720,918,459,163,133,103,77,576,81,910,456,474,
877,633,160,518,534,257,884,662,171,29,436,148,542,40,236,4,370,
572,343,229,461,527,515,148,571,923,713,652,383,735,803,539,94,
706,128,1019,167,440,527,594,627,562,1019,224,270,464,302,475,
762,754,239,739,926,391,642,316,797,416,63,96,111,702,794,575,
976,10,16,752};

int x[] = {876,727,355,891,403,270,946,551,151,600,680,575,369,864,
680,705,30,420,1006,647,817,603,982,752,884,172,605,991,293,658,
463,571,729,499,1014,635,679,702,564,457,693,505,334,889,716,674,
699,302,551,175,1021,385,446,49,910,158,611,366,787,217,252,656,
261,540,572,902,838,936,165,449,1002,172,705,837,985,615,232,737,
585,404,389,199,2,555,858,932,759,415,200,985,470,629,726,298,
536,835,881,1013,4,154,270,937,867,708,318,773,656,856,398,976,
144,883,961,453,240,479,257,610,500,943,173,615,161,587,30,472,
422,525,147,389,568,496,385,332,759,392,235,994,776,584,626,526,
758,1008,31,486,8,907,683,773,702,44,824,805,570,492,251,224,50,
184,834,911,774,363,418,994,231,486,672,118,408,662,431,528,748,
311,386,790,644,791,850,382,199,167,287,123,816,943,32,21,342,
678,703,625,82,945,799,183,130,847,523,14,570,471,245,985,568,
256,256,39,312,49,718,459,875,71,319,908,359,888,109,493,804,
445,324,70};

int testFunction();

int main()
{


PERF_RESET(PERFORMANCE_COUNTER_0_BASE);
PERF_START_MEASURING(PERFORMANCE_COUNTER_0_BASE);


int result = 0;
result = testFunction();

```

```
PERF_STOP_MEASURING(PERFORMANCE_COUNTER_0_BASE);

alt_u64 time = 0;
time = perf_get_section_time((void*)PERFORMANCE_COUNTER_0_BASE, 1);
printf("That was fun, the answer is %d, it took %d cycles\n", result, time);



  /* Event loop never exits. */
  while (1);

  return 0;
}


int testFunction(){

    int i;
    int j;
    int y = 0;

    PERF_BEGIN(PERFORMANCE_COUNTER_0_BASE,1);
    for ( i=0; i < LENGTH; i++) {
        y = y + ALT_CI_MY_MULT(h[i],x[i]);          //name of the mult instruction
    }
    PERF_END(PERFORMANCE_COUNTER_0_BASE,1);

    return y;
}
```

A.2 Custom multiplier, in VERILOG

```
// our half_adder.
// Inputs: x,y
// Outputs: sum,carry
module half_adder(x,y,sum,carry);
        input x,y;
        output sum,carry;

        and U_carry (carry,x,y);
        xor U_sum (sum,x,y);

endmodule

// ourfull adder
// inputs: x,y,carry_in
// outputs: sum,carry_out
module full_adder(x,y,carry_in,sum,carry_out);     //TODO fixed in/out order
        input x,y,carry_in;
        output sum,carry_out;
        wire and1,and2,and3,sum1;

        and U_and1 (and1,x,y),
            U_and2 (and2,x,carry_in),
            U_and3 (and3,y,carry_in);
        or  U_or   (carry_out,and1,and2,and3);
        xor U_sum (sum,x,y,carry_in);

endmodule


// our custom multiply instruction, were we
// implement a carry-save array multiplier
// inputs: dataa,datab (both 32bit)
// output: myZ (64bit)
module mult(
dataa, // operand A <always required>
```

```
datab, // operand B <optional>
myZ // result <always required>
);

input [31:0] dataa;
wire [31:0] dataa;

input [31:0] datab;
wire [31:0] datab;

//output [31:0] result;
output [63:0] myZ;
wire [63:0] myZ;

// a wirearray to connect all of our half- and full-adders
wire tmps [31:0][31:0][0:1] ; // [row][column][0 = sum, 1 = carry]
// a specially named wire array to take the results from the last row of adders and connect them to the outputs
wire bottomrow [30:0][0:1] ; // [column][0 = sum, 1 = carry]

// some counters for our loops
genvar r ;
genvar c ;

generate
//first row of the adder
for ( c = 0; c < 32; c = c + 1) begin:loopa

        //half_adder ha1( dataa[c] && datab[0] , 1'b0 , tmps[0][c][0] , tmps[0][c][1] );
        assign tmps[0][c][1] = 1'b0;  // a constant
        assign tmps[0][c][0] = dataa[c] && datab[0];
end

// the main loop. here we connect the bulk of our adders together
for ( r = 1; r < 32; r = r + 1) begin:loopb
        // c of 31 (left column: only take in carry)
        // this column of adders needs special attention, because it only takes in the previous row's carryout
        half_adder ha2(dataa[31] && datab[r], tmps[r-1][31][1], tmps[r][31][0], tmps[r][31][1]);
        //c of 0:30
        for (c = 0; c < 31; c = c + 1) begin:loopc
                // everything else that's not in the left most column.
                full_adder ha3( dataa[c] && datab[r] , tmps[r - 1][c][1], tmps[r-1][c + 1][0],
                tmps[r][c][0], tmps[r][c][1]);
        end
end

//right corner: c, s, no acc
half_adder ha4(tmps[31][0][1], tmps[31][1][0], bottomrow[0][0], bottomrow[0][1]);
assign myZ[32] = bottomrow[0][0];  // here we actually connect a result to an output wire

for ( r = 0; r < 32; r = r + 1) begin:loope
        //right column
        assign myZ[r] = tmps[r][0][0];   // connect the result to the output wires
end

for ( c = 1; c < 31; c = c + 1) begin:loopf
        //bottom row
        full_adder ha5(tmps[31][c][1], tmps[31][c+1][0], bottomrow[c - 1][1], bottomrow[c][0], bottomrow[c][1]);
        assign myZ[c + 32] = bottomrow[c][0];     // connect the result to the output wires
end
```

```

// finally, connect the last of the outputs from our array of adders to the output
    wires
assign myZ[63] = bottomrow[30][1];

endgenerate


endmodule



module my_mult(
dataa, // operand A <always required>
datab, // operand B <optional>
result // result <always required>
);


input [31:0] dataa;
input [31:0] datab;
output [31:0] result;
wire [63:0] myZ;

mult test_mult(dataa, datab, myZ);

assign result = myZ[31:0]; // only connect the lower 32bits




endmodule


// neat test case.
module test();

reg [31:0] dataa;
reg [31:0] datab;
wire [31:0] result;

wire res_s;
wire res_c;

initial begin
        // somem debugging stuff to help use monitor the output of our custom adder
        $monitor ("num1:%d num2:%d result:%d",dataa,datab,result);
        dataa = 4294967295;
        datab = 4;


end

//full_adder argh(1'b1,1'b1,1'b1,res_s,res_c);
my_mult argh(dataa, datab, result);

initial begin
                //$monitor("res_s: %d\nres_c: %d", res_s, res_c);

        $monitor ("res num1:%d num2:%d result:%d",dataa,datab,result);
end

endmodule
```