An embedded system, from software to hardware

(EDAN15 VT15 Final Report)

Anton Eliasson, `dat11ael@student.lu.se`
Daniel Lundell, `ada10dlu@student.lu.se`

June 7, 2015

Abstract

This report details the laboratory work where, during five sessions, software and hardware solvers of the Greatest Common Divisor were implemented and compared. A software solution was developed for a uniprocessor system and later parallelized to run on a dual processor system. Parts of it were then implemented in hardware. The solutions were profiled and discussed from three perspectives: performance, hardware utilization and energy consumption. The results demonstrate that significant performance improvements and decreased energy consumption can be gained when algorithm implementations approach hardware. It was shown that power and energy consumption do not necessarily correlate and that an implementation with slightly higher power requirements can have a much lower total energy consumption.

1 Introduction

This is a report on the laboratory work in the *Design of Embedded Systems (EDAN15)* course at LTH. The purpose of the laboratories is to use and implement an embedded system and to observe how different implementations differ in relation to efficiency, power and hardware utilization. The assignment is first solved with a pure software solution which is later parallelized and then transformed into a hardware accelerated solution. In addition the purpose is to explore frameworks and tool chains used in development of embedded systems. The embedded system used in the laboratory is a Xilinx FPGA platform.

The report is organized as follows. Chapter 2 describes the experiments conducted throughout the five laboratory sessions. Chapter 3 presents and discusses the measurements obtained during the experiments. Chapter 4 concludes the report with a summary.

2 Experiments

The experiments were carried out over five lab sessions. The goal was to evaluate software and hardware solutions running on a Xilinix Digilent Nexys-3 FPGA platform. All software was developed in C for the MicroBlaze[5] CPU architecture. The MicroBlaze is very customizable but in this project, the default settings were used. This optimized the CPU for the smallest area. 32 KiB of RAM was assigned to each processor core an AXI bus was used for communication between the CPUs and their peripherals.

The system clock was set to 100 MHz. For timing measurements, an *axi_timer*[3] connected to the AXI bus was used. A simple UART, also connected to the AXI bus, was used for communication with the operator. The hardware was described and synthesized using Xilinx Design Suite 14.2. In every solution the performance (run-time), FPGA area utilization and power was recorded for later comparison.

The first laboratory was to implement and compare two pure software solution of a Greatest Common Divisor (GCD) algorithm for N numbers. Furthermore, the software has to run on a single processor system.

The second laboratory session was to implement and evaluate again a pure software solution of a GCD algorithm for N numbers. This time the architecture should use a multi-processor system. The system uses two MicroBlaze processors, working on the same data set.

The third and fourth laboratory session was to select a part of the GCD algorithm for N numbers and implement it in hardware. The hardware core is first written in VHDL and simulated using Xilinx ISE. It communicates with the processor using a Fast Simplex Link interface[4].

The last laboratory session was to integrate the hardware developed in the previous sessions into a larger system. It has software to communicate with the operator over a serial link, similar to the uniprocessor solution, but uses the hardware core for the actual computations.

2.1 Software algorithms

The algorithm chosen for computation of the greatest common divisor is the Euclidean algorithm. This was chosen because it is both simple to implement, and because it, in its simplified form that is only valid for positive integers, avoids mathematical operations that are difficult to implement in hardware such as division and modulo. An algorithm in pseudo code to calculate the GCD for two integers[2, pp. 318-320] is shown in listing 1. To calculate the GCD for an arbitrary number of integers, the fact that the GCD is an associative function is used, so for every a, b and c, $gcd(a, gcd(b, c)) = gcd(gcd(a, b), c)$ holds.

2.2 Single processor

The single processor system was implemented using a single MicroBlaze core. A simple software program reads data from the serial port, calculate the GCD and outputs it to the user. The GCD used is the one described above.

```
function gcd(a,b)
        while b != 0 {
                if a > b
                        a = a - b
                else
                        b = b - a
        }
        return a
}
```

Listing 1: Euclidean subtracion algorithm

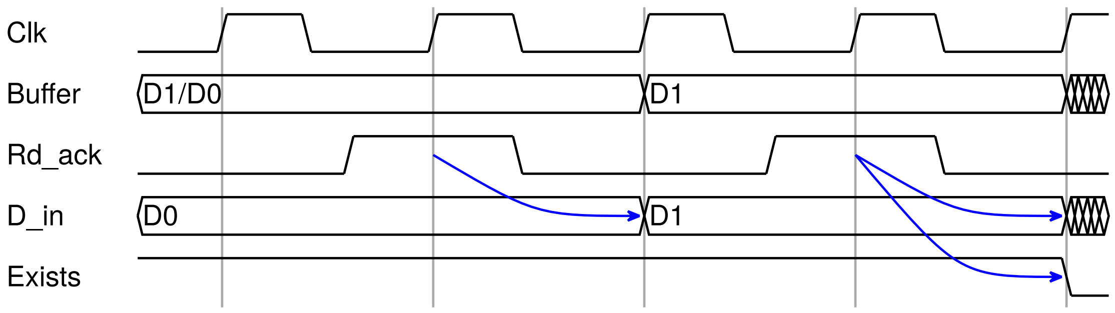

Figure 1: Timing diagram for an FSL read operation

2.3 Dual processor

The dual processor system was implemented using two identical MicroBlaze cores with separate memory. The two cores communicates using two unidirectional Fast Simplex Links (FSL), creating a bidirectional interface. A master-slave model of communication between the cores was used. The master core receives data from the serial link, splits the data set into two halves and sends one half over FSL to the slave core. The cores calculate the GCD for their own halves separately and in the end the slave sends its result back over FSL. The master computes the GCD for the final two values and sends the result back over the serial link.

2.4 Hardware accelerated

The part chosen for hardware acceleration was the GCD algorithm for two numbers. The hardware runs on a custom IP core on the same board as the MicroBlaze core running the software. The software reads the input over serial into a buffer during program startup. It then sends a pair of the input elements over FSL to the hardware core and waits for the result. The GCD is computed in hardware and sent back to the MicroBlaze core over FSL. As long as buffer in the program is not empty, the previous result is sent back to the hardware core along with the next element.

Figure 2 shows the timing diagram for an FSL read operation. Pulsing the *Rd_ack* high for one clock cycle causes the next word in the buffer to be moved to *D_in* on the

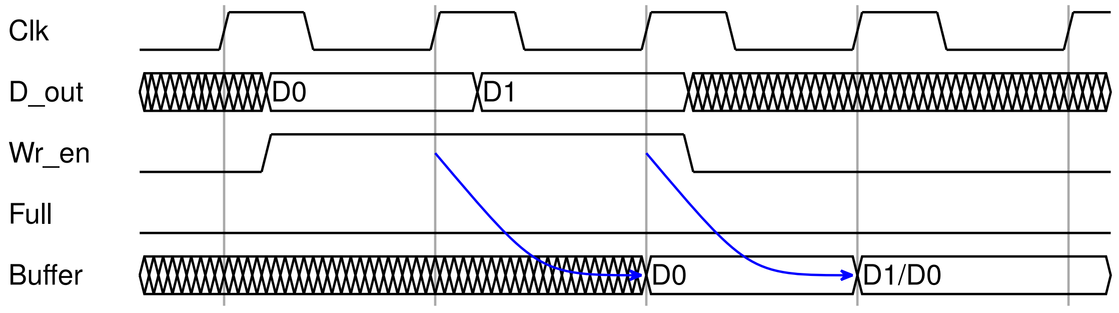

Figure 2: Timing diagram for an FSL write operation

next low-high clock transition. *Exists* tells the recipient that more data is available in the buffer.

Figure 2 shows the timing diagram for an FSL write operation. There is no acknowledgement signal here. The sender tells the FSL bus that it is writing a word to *D_out* on each clock cycle as long as it is holding *Wr_en* high. It is allowed to do so as long as the recipients buffer is not full, as indicated by the *Full* signal.

A Mealy machine state diagram for the custom IP core is shown in figure 3. Inputs are shown in black and outputs in green. Some transitions don't have transition conditions, and some lacks outputs. This is because the state diagram shows a clocked circuit and during some transitions signals like *Rd_ack* must be held high for exactly one clock cycle and then pulled low again.

3 Measurements and Discussion

This section describes the results obtained from the experiments. To get consistent results, the *axi_timer* that was used was configured to measure only the actual calculation and not the communication with the operator. In the hardware accelerated solution the timer was started just before the operands were sent to the hardware IP core and stopped just after the final result could be read back.

3.1 Performance

Tables 1, 2 and 3 show the time takes, as measured in CPU clock cycles, to calculate the GCD for the sample data[1] using the different solutions implemented. The performance figures from the first lab, comparing a naive algorithm for computing the GCD with an implementation of the Euclidean algorithm, are shown in table 1. It shows that the Euclidean algorithm is almost 100 times faster for all datasets but one (N=30). The divergent result is believed to provoke the worst-case complexity of the Euclidean algorithm since, as will be demonstrated later in this section, it takes significantly more time to compute for all implementations used. There appears to be a negative correlation between the clock cycles required and the final GCD. This could be explained by

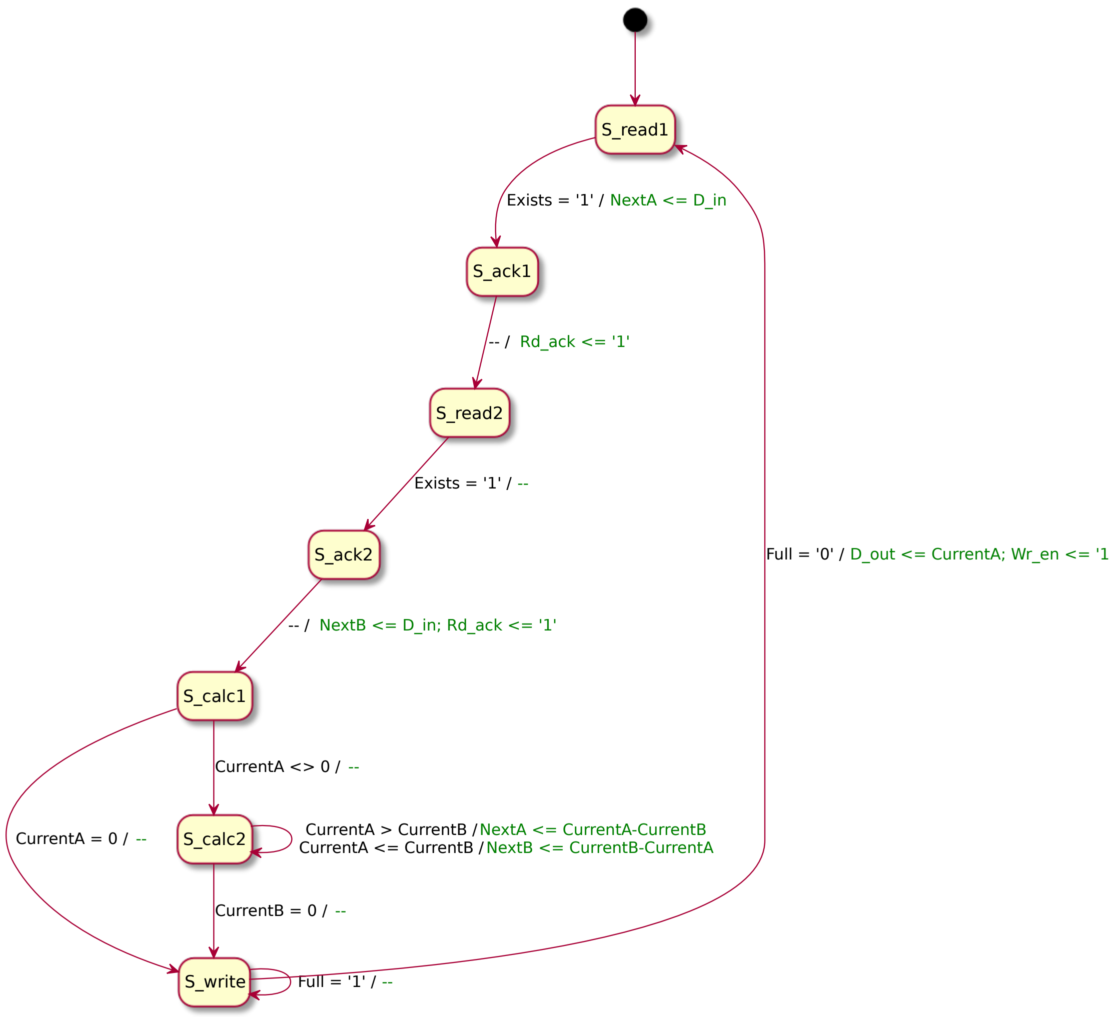
S_read1
Exists = '1' / NextA <= D_in
S_ack1
-- / Rd_ack <= '1'
S_read2
Exists = '1' / --
S_ack2
-- / NextB <= D_in; Rd_ack <= '1'
S_calc1
CurrentA <> 0 / --
CurrentA = 0 / --
S_calc2
CurrentA > CurrentB / NextA <= CurrentA-CurrentB
CurrentA <= CurrentB / NextB <= CurrentB-CurrentA
CurrentB = 0 / --
S_write
Full = '1' / --
Full = '0' / D_out <= CurrentA; Wr_en <= '1

Figure 3: State diagram for the custom IP core

N	GCD	Naive single	Euclid uniprocessor	Euclid/naive ratio
10	71	2,433,868	24,978	1.0 %
30	3	1,992,704	1,846,630	92.7 %
50	6211	13,142,315	17,097	1.3 %
70	128	1,479,966	108,319	7.3 %
100	587	3,152,887	37,876	1.2 %

Table 1: Performance numbers for the different implementations

N	GCD	Euclid single	Euclid dual	Dual/single ratio
10	71	24,978	12,726	50.9 %
30	3	1,846,630	928,490	50.3 %
50	6211	17,097	9,207	53.8 %
70	128	108,319	58,745	54.2 %
100	587	37,876	21,217	56.0 %

Table 2: Performance numbers for the different implementations

properties of the euclidean algorithm. The time required is proportional to the number of divisions required, or in its simplified version, the number of subtractions.

The performance figures comparing the single and dual processor software implementations are shown in table 2. The results show that the dual processor solution takes just over half the time compared to the single processor solution in all datasets. This was not surprising to us since each core can calculate the GCD for half the dataset independently, without having to wait for or synchronize with the other core until the end. The FSL transfers are included in the clock cycle counts, but the dual processor solution is still almost twice as fast. The fraction-of-a-percent divergence from 50 % for the small datasets are explained by the constant time required for the FSL transfers.

It was found that the while the FSL buffers can be quite large, the default size is only 16[4]. This means that for the larger datasets, the slave CPU's buffer would be filled up, causing `putfsl` to block. It might explain why, for N=50, 70 and 100, the clock cycle ratio between the dual and single processor implementation appears to increase beyond 50 %. A larger FSL queue depth may yield better performance here. Another performance improvement could be to use shared memory instead, as in that case no data would have to be exchanged at all.

The hardware accelerated solution is an order of magnitude faster than the single processor implementation of the Euclidean algorithm, as shown by the performance numbers in table 3. The GCD calculation is moved completely into hardware. The software is only responsible for communication with the operator and communicating the operands and result with the hardware IP core.

Even better performance would likely be obtained by hardware accelerating more parts, like the entire GCD for N numbers algorithm. The program would first have

N	GCD	Euclid single	Hardware accelerated	Hardware/single ratio
10	71	24,978	1,595	6.4 %
30	3	1,846,630	100,623	5.4 %
50	6211	17,097	2,485	14.5 %
70	128	108,319	7,688	7.1 %
100	587	37,876	5,129	13.5 %

Table 3: Performance numbers for the different implementations

O	Single	Dual	Hardware	Hardware/dual ratio	Slave size	Hardware master size
0	1,993 k	928 k	100.6 k	11 %	9,994	9,866
1		356 k	100.1 k	28 %	5,986	9,246
2		371 k	100.1 k	27 %	5,994	9,254
3		464 k	100.1 k	22 %	6,086	9,390

Table 4: Performance and code size for different optimization levels

to send the total number of elements, N, to the hardware core so that it knows how many numbers to expect. The hardware core would then proceed to read two operands, calculate their GCD and then for each result, calculate the GCD between the result and the next operand in the buffer, as long as less than N numbers have been read. Alternatively, the result could be initialized with 1 and the hardware core would then always use the last result as one of the operands. Some kind of reset signal would have to be implemented to reset the result back to 1. This would likely be faster than the current solution because the hardware would spend less time waiting for the CPU to send it more work to do.

3.2 Compiler optimization

Table 4 shows some clock cycle measurements as well as compiled code sizes for different optimization levels. Again $N = 30$ in all samples. It is clear that some optimization can vastly improve the performance of software implemented algorithms, while also reducing the code size. Further optimization appears to worsen the performance while also increasing the code size slightly. One possible explanation for this is that high levels of optimization tends to for instance unroll loops which increases the code size. Larger code causes more instruction cache misses which in turn causes more slow memory access operations. Unfortunately, all but the $O = 0$ performance measurement of the uniprocessor implementation were lost.

The program used in the hardware accelerated solution basically only handles communication between the operator and the custom IP core and does not do any actual calculations. For this reason, optimizing it does not do much difference to the run-time of the entire program.

Name	Single processor	Dual processor	Hardware accelerated	Available
CLB Flip-Flops	1,684	2,499	1,812	18,224
LUTs used as logic	2,092	3,222	2,352	9,112
LUTs used as memory	152	301	187	2,176
Occupied Slices	962	1,411	992	2,278
16K block RAMs	16	32	16	32

Table 5: Hardware utilization for the different implementations

Column 6 in table 4 lists the code sizes of the slave program in the dual processor software solution. Column 7 lists the only (master) program in the hardware accelerated solution. As the master program does much more than the slave program (for example handles the UART communication), their sizes cannot be directly compared in a meaningful way.

3.3 Device Utilization

The Xilinx tools are capable of exporting hardware utilization numbers in great detail. A selection of these that were found interesting are presented in table 5. The table shows that hardware parts that are used exclusively in the MicroBlaze cores, like memories, were basically doubled when a second core is added. However, parts like Flip-Flops that are common in many digital circuits were not doubled. The conclusion was drawn that this is because the entire system also contains parts that are shared between the processor cores, like buses, the clock generator and the reset circuit. There are also hardware interfaces to the UART, which is only used by the master processor, and the completely unused LEDs and DIP-switches.

Table 5 also shows that with two processors using 32 KiB RAM each, the FPGA's dedicated block RAMs are completely occupied. This puts an upper limit on the performance of the current software implementation, as no more CPU cores can be added. The custom IP core on the other hand uses very little hardware resources. This was expected, as the IP core is very application specific compared to a general purpose CPU. If greater performance is needed, many more can be added to the system to work in parallel before the FPGA is fully occupied.

3.4 Power and Energy

The power and energy usage numbers are presented in table 6. The power has been estimated with the Xilinx XPower Analyser. The energy consumption has been calculated using equation (1) where the frequency is 100 MHz. The result is measured in Watt-seconds, or Joules. For the energy consumption comparisons, the performance numbers for $N = 30$ was used, as this had the longest run-time across all three implementations.

$$E = P\Delta t = P \cdot \frac{clock\ cycles}{frequency} \tag{1}$$

	Single processor	Dual processor	Hardware accelerated
Power (W)	0.163	0.172	0.164
Energy (mJ)	3.25	1.60	0.165

Table 6: Power and energy consumption for the different implementations calculated from the GCD for $N = 30$ results

The power consumption does not differ much between the implementations. The single processor implementation has the lowest power consumption. This was expected, since it has the least amount of hardware. The increase in power consumption when a second MicroBlaze core was added is surprisingly small. As with the hardware utilization, this is probably explained by the fact that a lot of the hardware is shared by the the processors. The power consumption in the hardware accelerated solution is only slightly higher than the uniprocessor solution. This was not surprising, since the hardware IP core as mentioned in the previous section is very small and simple.

There was however a significant difference in the energy consumption between the implementations. After reading the power consumption number, this was not unexpected since the energy consumption is proportional to the clock cycles used, as can be seen in equation (1). As the power is fairly constant between the implementations, the energy used by the hardware solution is an order of magnitude lower than the software implementations.

4 Summary

The experiments that were detailed and analysed in this report show that it can be very beneficial in both performance and energy usage to implement time-consuming parts of an algorithm in hardware. Massive gains were observed when a hardware IP core was developed to compute the GCD of two integers in a program that computes the GCD of an arbitrary number of integers. For a very small additional cost in FPGA fabric area and power consumption, the run-time of the program decreased by an order of magnitude, causing the energy usage to decrease by almost as much.

Simply parallelizing the algorithm to run on two processor showed similar improvements, albeit not to as large extent. The fairly low increase in power consumption when adding a second core was initially surprising. It did demonstrate the relatively high constant cost in terms of support circuits required of implementing a computer with a single processor core. As the number of cores increase, this constant cost becomes less relevant.

The experiments gave a valuable insight in hardware development using the hardware description language VHDL. Splitting the algorithm into a clocked and a combinatorial part was initially challenging as it requires a different mindset than is used in software programming.

Writing the report was a valuable exercise in reading and writing timing and state diagrams. When working with projects like this, they can be very useful.

References

[1] *EDAN15 – Assignments*. URL: `http://cs.lth.se/edan15/labs/assignments` (visited on 05/29/2015).

[2] Donald Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms (3rd edition)*. Addison-Wesley, 1997.

[3] Xilinx. *LogiCORE IP AXI Timer v2.0 Product Guide (AXI)*. Apr. 2, 2014.

[4] Xilinx. *LogiCORE IP Fast Simplex Link (FSL) Bus (v2.11f) Data Sheet*. Dec. 18, 2012.

[5] Xilinx. *MicroBlaze Processor Reference Guide*. EDK v6.2. June 2004.