Embedded Systems

02131

The ECG processor!

Bastian Buch, s113432
Jacob Gjerstruo, s113440

Abstract

The main task of this assignment was to develop a custom made processor which needs to be able to run the code for the ECG scanner, which was developed in the previous assignment. By using Gezel, we have created the basic modules for the processor, which would then need to be coupled together to create the actual processor, and furthermore, we have developed the instruction set for the processor, which is written in assembly code. Our conclusion is that we could quite easily connect the modules of the processor and thereafter start working on having this processor start on the instruction set, but unfortunately, we started too late in the project and ended up with a lack of time for doing this.

Indhold

1 Introduction

For this part of the project, we were to continue our work for Medembed in designing and implementing a small embedded processor that can execute the QRS algorithm we developed. However, initially, we would first be required to implement only one filter - the Moving Window Integration filter - to demonstrate the capabilities of the processor - both in terms of size, speed and power consumption.

We were tasked to make this implementation in the hardware description language called Gezel.

2 Requirements

For this assignment, we initially sat down and looked over the requirements, and in total, we found 4 functional requirements and one non-functional requirement. They are as follow:

Functional requirements for the application:

- Creating the basic processor modules
- Creating the instruction set the processor must use
- Implementation of controller.
- An analysis of our implementation, including an analysis of the critical parts, runtime and memory requirements.

Non-functional requirements for the application:

- The processor must be implemented in the Gezel language.

3 Analysis

In designing the actual ECG scanner, there are a few things that we had to consider.

1. What are the basic modules of a processor?

2. How should these modules be implemented?
3. What instruction set should be used?
4. How would the controller be implemented?
5. How do you determine the critical parts?

3.1 Problem 1: The basic modules of a processor

The first question we had to ask ourselves - what are the basic modules of a processor actually? This was a question that we needed to answer firstly, as without these basic modules, implementing the processor itself would be impossible.

3.2 Problem 2: Implementation of the basic modules

The second problem we considered was the question of how to actually implement these basic modules? Most seemed straight forward, but we encountered some issues in regards to the modules:

1. In regards to the Arithmetic Logic Unit (ALU), how can the choices be implemented so the runtime is swift and the size is small?
2. In regards to the memory, how can we read and write data, and furthermore, how should this data look?
3. In regards to the Register, we had to figure out how we ensured that a registry always returned 0 no matter what.

3.3 Problem 3: The instruction set

Designing the instruction set would have to be done manually, and this instruction set would need to be in assembly code, to ensure our processor can understand and interpret the instruction set correctly. Designing this instruction set could be done in multiple ways, and the biggest problem would be how to design this instruction set so it is as compact as possible, while ensuring that it is still compileable.

3.4 Problem 4: Implementation of the controller

3.5 Problem 5: Critical parts

When speaking of critical parts, there are three terms that needs to be discussed - size, speed and power.

The size of the processor is important because the larger it is, the more power-consuming it will be, and secondly, as the ECG scanner in itself cannot be that large, there is a maximum size the processor can have.
The speed is of course also important - the processor needs to be able to make a certain minimum number of calculations per second, in that the scanner itself, which will provide the processor with data, will bring 250 data points to be calculated on per second, and it is important that the processor is able to keep up with this stream of information. On the other hand, it is also important that the processor does not work too fast - having it consume lots of power to calculate, only to have it idle in-between data points would mean that energy is wasted, lowering the lifetime of the ECG scanner.
Finally, power consumption is important in that the more energy the processor consumes, the more often the battery on the scanner will need to be replaced or recharged - neither of which you want the costumer to bother with too often. The way to avoid this is either to reduce power consumption or to increase the battery size - the latter of which can only be done to a certain degree as the ECG scanner has a specific size.

1. How large would the processor be and can this size be reduced?
2. How fast is the processor, and how much idle time is there between data points?
3. What is the energy consumption and what can be done to reduce consumption?

4 Design

When we were designing the basic modules, we very quickly decided to implement them in a testbench setting from the very beginning, as this made testing an easy and very comfortable thing. This setting simply consists of a test datapath as well as a Finite State Machine (FSM) to control the testbench and change the values of the registers, depending on the clock cycle.
Also, when we were designing the instruction set, we had to discuss whether we wanted to use a Div module or if we would make a loop that continually subtracted a set value until the value would be below 0. We decided to go with making a Div module - although this would mean we would need a more complicated module, it also means we would need less calculations, as we would not need a loop.

5 Implementation

5.1 The basic modules of a processor

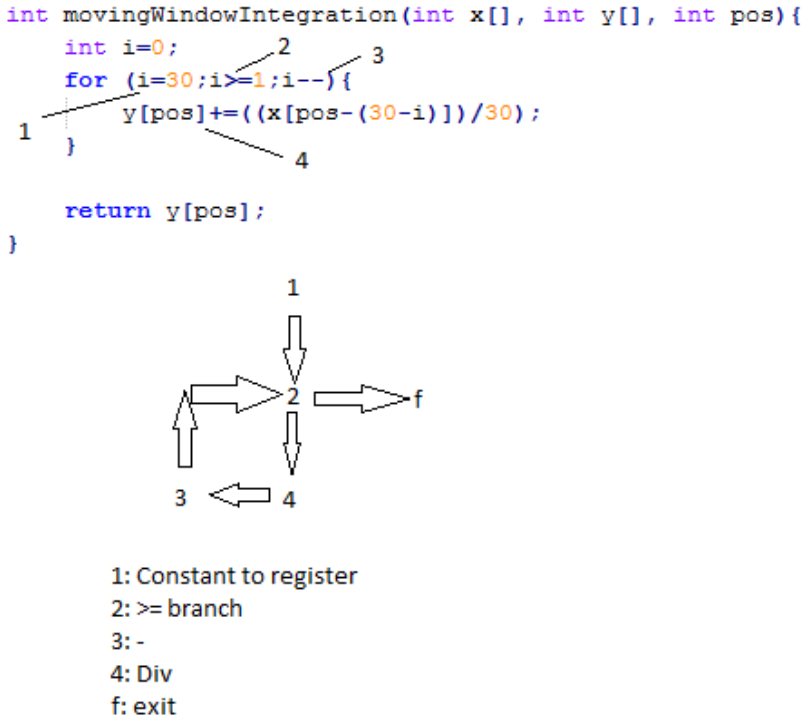
The basic modules of the processor we would implement were decided to be a program counter, an adder, a multiplexer, an Arithmetic Logic Unit (shortened to ALU from now on), a register file and instruction memory. Furthermore, a controller would need to be implemented as an advanced module.

5.2 Implementation of the modules

Implementations of the first modules - the counter, the adder and the multiplexer, were all fairly straightforward, and we encountered no major issues with these. However, the next module, the ALU, were more complicated in that it was a combination of the multiplexer and the adder, and the combination of these turned out to possess some interesting challenges.
First of all, the choice in itself could either be done through a series of nested if/else functions, it could be done with a controller that defines when the ALU would do what, and the controller in itself could also be done in multiple ways - it could be a finite state machine, or it could just be a hardwired controller.
Ultimately, we decided with the simplest approach, which was simply 3 if/else statements. This, we concluded, would take up the least amount of space, and would likely also be the most efficient, as it results in less comparisons and calculations.
The next issue we encountered were in the ram-block, and the question was how to read and write data, and furthermore, what format this data should take. We accomplished this by using a predefined library module, the Ram module, which includes both a read and a write command, as well as parameters for the file used to read from.
In regards to how the data should look, it was defined that the data values would take on the form of two columns, where the first column would be the address and the second column would be a hexadecimal format with a word length of 32 bits.
Finally, the last issue in the standard modules we encountered were in the register module where we had to ensure that a specific registry always returned 0 no matter what. This was accomplished by noticing that Gezel always initializes a register as 0, and therefore we just did not include a function to edit the register.

5.3 The instruction set

When we were determining the instruction set, the first thing we did was creating a diagram over our C code and what commands we would need for translating the C code to assembly code.



Figur 1: A diagram depicting the translation from C to assembly code.

Once we had the block diagram, we proceeded with writing the actual assembly code itself. This code can be seen on figure 2 - seen below.

```
//Created by Bastian Buch, s113432
//Final instruction set; 8 opcodes;
ADD
ADDI
SUB
FIDIV
BNEG
B
LOAD
OUT

//Assembly code; instruction count = 17
ADDI $1, $0, 30         //Initialization, BLOCK 1 begins here
ADDI $2, $0, 1          //Initialization
ADD $5, $0, 32          //This is the position of X[0] in the RAM
ADD $6, $0, 65          //This is the position of Y[0] in the RAM
LOAD $7, $98            //This is Pos
SUB $7, $7, $2          //Pos is reduced by 1, see explanation
ADD $3,$6,$7            //This is Y[0+(pos-1)], and therefore Y[Pos]
LOAD $3, $6             //This loads the value of Y[Pos] in register 3
ADD $4,$5,$7            //This is X[0+(pos)], and therefore X[Pos+1]. This makes the loop easier.
ADD $7, $7, $2          //Pos is increased by 1
SUB $1, $1, $2          //Substraction to check loop, BLOCK 2 begins here
BNEG 17                 //Goes to line 17
SUB $4,$4,$2            //Substracts 1 from X[pos+1]
LOAD $5, $4             //Loads the value of X[pos]
FIDIV $5,$5,30          // Divdes the value by 30
ADD $3, $3, $5          // Adds the value of X[pos] to the value of Y[pos]
B 12                    //Returns to line 11
OUT $3                  // Sets the new value as output
```

Figur 2: The assembly code.

We have implemented the assembler code with focus on minimizing the amount of instructions inside the "loop", as well as attempting to include as few operator codes as possible, generally attempting to use as few hardware resources as possible. This leads to some curious coding - substracting one from a register and adding one later, for example.

An explanation of our operator codes follow:

1. ADD, addition of two registers. First variable is the register saved to, second and third are the added registers.
2. ADDI, addition of register and integer. Same as ADD, with third register replaced by an int.
3. SUB, substraction of two registers. Syntax same as ADD, except substraction instead of addition.
4. FIDIV, division of a register with an integer. Same as ADDI, except the second register is divided with the integer.
5. BNEG, jump if negative. If the result of the previous calculation is negative, the assembler jumps to the line specified.
6. B, jump to. The assembler returns to the line specified with an integer.
7. LOAD, get external data. Data returns data to the first register, taken from the line of our RAM specified with the second register/integer
8. OUT, output external data. Returns the value of a register as output.

5.4 Implementation of the controller

When implementing the controller, we would start out with building the datapath - that is, connecting the various components so they can actually calculate correctly, by initially creating the components that would be a part of the ADDI and the SUB/ADD assembly instructions. Once these datapaths were completed and connected, we would then start building the controller itself, whose job it would be to control the signals of the datapaths by defining the signals depending on the instruction set. The controller itself would be implemented as a separate datapath.

5.5 Critical parts

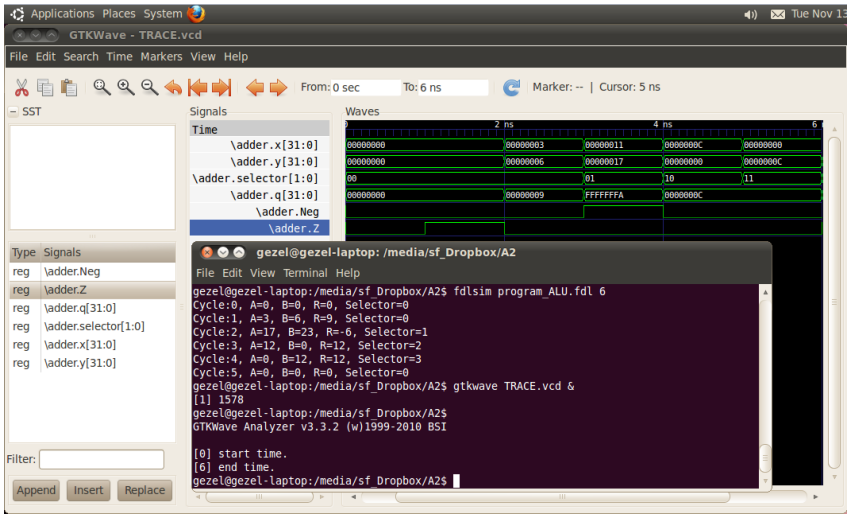
As we never got to implement the controller, we unfortunately could not evaluate the critical parts. However, had we managed to implement the controller, we would have evaluated the speed by the amount of clock cycles it would take to go through one data point on average. We would then look into what part of the program that requires the most time by loading in data from an external memory, and then use said data to evaluate the time - varying it throughout the testing to find out what takes the longest.
In regards to size, the simplest way to do so is to simply count the number of registries, adders, and multiplexers - along with their bit-width. When we were to

then optimize it, we would look into which registries we might not need, and which we could reduce in width without compromising the integrity of the processor.
Finally, in regards to power consumption, there are generally three important terms to consider - Switching, which is power consumption because of active components; Short-circuit, which is when nMOS and pMOS transistors are on at the same time; and Static, which is power consumption because of inactive components. When we were to optimize, we would choose only to look at switching, as the amount of power needed goes up proportional with the activity - meaning that if the activity goes up a lot, the power consumption goes even higher.
One measures the amount of switching by looking at the amount of toggles there is per clock cycle - that is, the amount of bits switching from 0 to 1 or the other way around. Finding out just how many bits switch is quite easy in Gezel - we would just need to toggle on operation profiling, which is two commands. This would then show us the amount of evaluations and toggles, and through this tool, we would be able to see where we would need to optimize, and where we would be fine.

6 Results

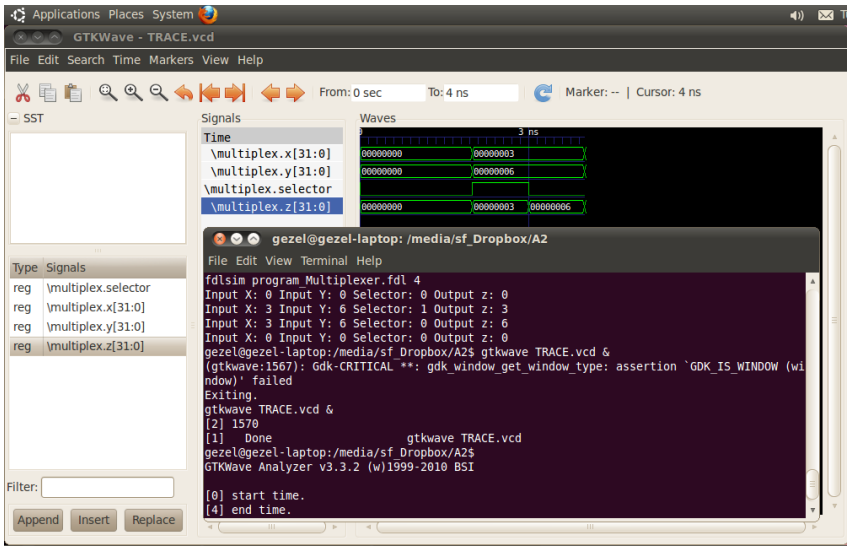
The results and traces of our tests were intended to fully cover all possible errors in functionality. For the ALU, this would mean testing all functions and checking their results to see if they were correct, for example, whereas the Multiplexer simply needs to demonstrate that the function picks the correct result.

Figure 3, 4, 5 and 6 are screenshots of the tests we have run on our basic modules.

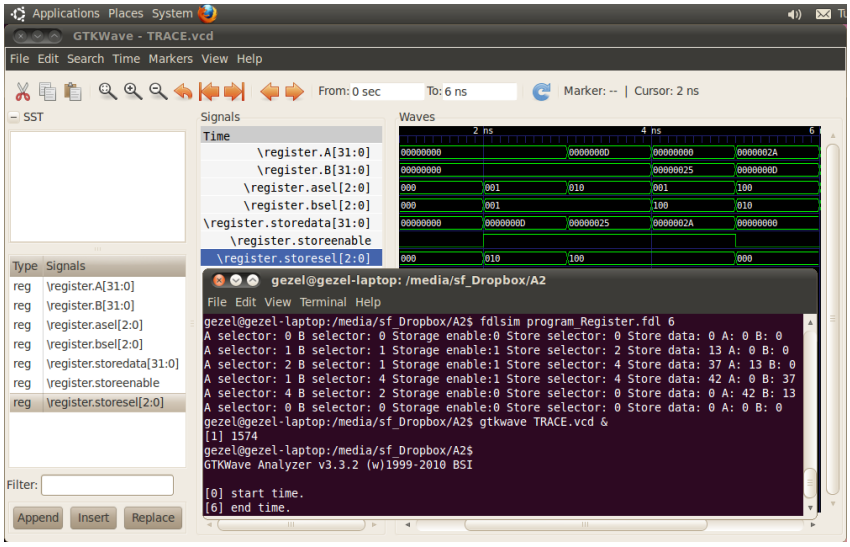


Figur 3: The results of the ALU testbench.

The screenshots consists of the gtkTrace result of all variables and their bit size minus one, as well as the test results from the terminal. If nothing is noted in the



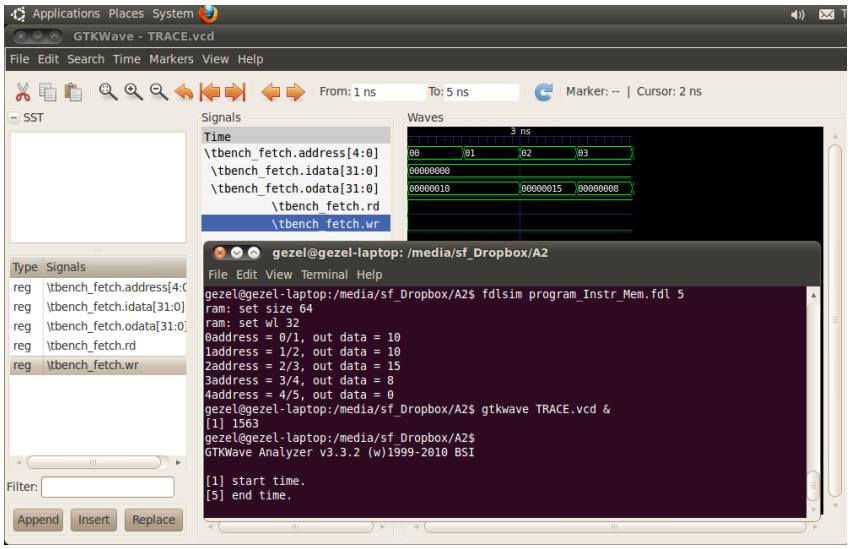
Figur 4: The results of the Multiplexer testbench.



Figur 5: The results of the Register testbench

GTK next to the variable name, the variable is a single bit (1 or 0).
The ALU tests its functions after a zero cycle:

1. Addition
2. Substraction
3. Return X
4. Return Y



Figur 6: The results of the RAM testbench

The multiplexer tests its functions after a zero cycle:

1. Return X
2. Return Y

The Register tests after a zero cycle:

1. Store in a register (register 2)
2. Read register 2 with A and store in register 4
3. Read register 4 as B and store again in register 4
4. Read register 4 with A and register 2 with B
5. Zero cycle (to counteract trace drop)

The RAM block simply reads the lines from program.txt.

As is quickly ascertained, the trace output the correct results from the variables. It should be noted that Trace outputs the numbers in hexadecimal, and that the trace consistently does not recieve the results from the last clock cycle in the test. This has been compensated for by running an extra cycle, which is usually a test where all variables are zero.

7 Conclusion

We did unfortunately not manage to implement the controller and thus, we could not analyse the performance of it either. However, we did create all the basic components of a processor, as well as the instruction set that the processor has to use for using the Moving Window Integration filter.

Litteratur

[1] Michael Reibel Boesen, Linas Kaminskas, Karsten Juul Frederiksen and Dusan Vuckovich
Assignment 2: The ECG processor
1st Edition
2012.

[2] Gezel Basic Syntaks pdf

[3] `http://rijndael.ece.vt.edu/gezel2/manual.html`
Date of use: 30/10-2012

[4] `http://users.dickinson.edu/~braught/talksnpapers/ccscne01/KandS/instructions.html`
Date of use: 10/11-2012

Appendix

A Who wrote what

Jacob Gjerstrup, s113440 wrote: Introduction, Requirements, Analysis, Design, Implementation (50 percent), Abstract, Conclusion
Bastian Buch, s113432 wrote: Results, Instruction set (50 percent)

For programming, a comment at each document has been made that shows whom programmed this particular part.

B Sourcecode

B.1 Basic Modules

B.1.1 Program Counter

```
// Created by Jacob Gjerstrup, s113440
$option "vcd"

dp PC(out pc_out : ns(5)){
        reg a : ns(3);
        $trace (pc_out, "outputs.txt");
        // traces output to the Outputs textfile
        always{
                pc_out = a;
                $display($dec, "Cycle: ", $cycle); // just for debug
                $display($dec, "pc_out:",pc_out); // just for debug
                a=a+1;

        }
}

system PCsystem{
        PC(pc_out);
}
```

B.1.2 Adder

```
// Created by Jacob Gjerstrup, s113440
$option "vcd"

dp adder(in x, y : tc(32); out z : tc(32)) {
        $trace (z, "outputs.txt");
        always{ z=x+y; }
}
dp tbench_Adder(out A, B:tc(32); in R:tc(32)) {
        always{$display($dec, "Cycle:", $cycle, ", A=", A ,
                ", B=", B , ", R=", R);}
        sfg test_0 { A=3; B=6;}
        sfg test_1 { A=23; B=17;}
        sfg test_2 { A=12; B=0;}
}
// state machine to control testbench
fsm f_testbench(tbench_Adder){
        initial s0; // begin with state s0
        state s1, s2; // other states are: s1, s2
        @s0 (test_0) -> s1; // run test_0 and go to s1
        @s1 (test_1) -> s2; // run test_1 and go to s2
        @s2 (test_2) -> s0; // run test_2 and go to s0
}
system myFirstSystem {
        adder(A, B, R);
        tbench_Adder(A, B, R);
}
```

B.1.3 ALU with flags

We have decided only to show the sourcecode of the extended ALU with flags, as it is identical to the standard ALU, except it holds a few more registers and calculations, which is used for calculating the flags.

```
// Created by Jacob Gjerstrup, s113440
$option "vcd"

dp ALU(in x, y : tc(32); in selector: ns(2);
        out q : tc(32); out Neg, Z : ns(1)) {
        $trace (x, "traceofALUX.txt");
        $trace (y, "traceofALUY.txt");
        $trace (q, "traceofALUQ.txt");
        $trace (selector, "traceofALUSelector.txt");
        $trace (Neg, "traceofALUNeg.txt");
        $trace (Z, "traceofALUZ.txt");
        always{
                q = (selector==0b00) ? x+y :
                    (selector==0b01) ? x-y :
                    (selector==0b10) ? x : y;
                Z = (q!=0) ? 0 : 1;
                Neg = (q>=0) ? 0 : 1;
        }
}


dp tbench_ALU(out A, B:tc(32); out selector: ns(2);
                in R:tc(32); in Neg, Z : ns(1)) {
        always{$display($dec, "Cycle:", $cycle, ", A=", A ,
                ", B=", B , ", R=", R, ", Selector=", selector ,
                ", Neg =", Neg, ", Z=", Z);}

        sfg test_0 { A=3; B=6; selector=0b00;}
        sfg test_1 { A=23; B=17; selector=0b01;}
        sfg test_2 { A=12; B=0; selector=0b10;}
        sfg test_3 { A=5; B=3; selector=0b11;}
        sfg test_4 { A=5; B=5; selector=0b01;}
        sfg test_5 { A=5; B=10; selector=0b01;}
}
// state machine to control testbench
fsm f_testbench(tbench_ALU){
        initial s0; // begin with state s0
        state s1, s2, s3, s4, s5;
```

```
        // other states are: s1, s2, s3, s4, s5

        @s0 (test_0) -> s1;             // run test_0 and go to s1
        @s1 (test_1) -> s2;             // run test_1 and go to s2
        @s2 (test_2) -> s3;             // run test_2 and go to s3
        @s3 (test_3) -> s4;             // run test_3 and go to s4
        @s4 (test_4) -> s5;             // run test_4 and go to s5
        @s5 (test_5) -> s0;             // run test_5 and go to s0
}
system ALU {
        ALU(A, B, selector, R, Neg, Z);
        tbench_ALU(A, B, selector, R, Neg, Z);
}
```

B.1.4 Multiplexer

```
// Created by Bastian Buch, s113432
$option "vcd"

dp multiplex(in x, y : tc(32); in selector : ns(1);
                out z : tc(32)){
        $trace (x, "traceofMultiplexerX.txt");
        $trace (y, "traceofMultiplexerY.txt");
        $trace (selector, "traceofMultiplexerSelector.txt");
        $trace (z, "traceofMultiplexerZ.txt");
        always{ z=selector ? x : y; }
}
dp tbench_Multiplex(out x, y : tc(32); out selector : ns(1);
                        in z : tc(32)){
        always {$display($dec,"Input X: ",x," Input Y: ",y,
                " Selector: ",selector , " Output z: ", z);}
        sfg test_0 { x=3; y=6; selector=1;}
        sfg test_1 { x=3; y=6; selector=0;}
}

fsm f_testbench(tbench_Multiplex){
        initial s0; // begin with state s0
        state s1; // other states are: s1
        @s0 (test_0) -> s1; // run test_0 and go to s1
        @s1 (test_1) -> s0; // run test_1 and go to s0
}

system multiplexsystem {
        multiplex(x,y,selector,z);
```

```
        tbench_Multiplex(x,y,selector,z);
}
```

B.1.5 Register

```
// Created by Bastian Buch, s113432
$option "vcd"

dp register(in asel, bsel, storesel : ns(3);
                in storeenable : ns(1);
                in storedata : tc(32);
                out A, B : tc(32)){
        reg rega: tc(32);
        reg regb: tc(32);
        reg regc: tc(32);
        reg regd: tc(32);
        reg rege: tc(32);
        reg regf: tc(32);
        reg regg: tc(32);
        reg regh: tc(32);
        $trace (asel, "traceofRegisterAsel.txt");
        $trace (bsel, "traceofRegisterBsel.txt");
        $trace (storesel, "traceofRegisterStoresel.txt");
        $trace (storeenable, "traceofRegisterStoreenable.txt");
        $trace (storedata, "traceofRegisterStoredata.txt");
        $trace (A, "traceofRegisterA.txt");
        $trace (B, "traceofRegisterB.txt");
        always {

        regb = (storesel==1) ? ((storeenable==1) ? storedata :
                regb): regb;
        regc = (storesel==2) ? ((storeenable==1) ? storedata :
                regc): regc;
        regd = (storesel==3) ? ((storeenable==1) ? storedata :
                regd): regd;
        rege = (storesel==4) ? ((storeenable==1) ? storedata :
                rege): rege;
        regf = (storesel==5) ? ((storeenable==1) ? storedata :
                regf): regf;
        regg = (storesel==6) ? ((storeenable==1) ? storedata :
                regg): regg;
        regh = (storesel==7) ? ((storeenable==1) ? storedata :
                regh): regh;

        A = (asel==0b000) ? rega :
```

```
                (asel==0b001) ? regb :
                (asel==0b010) ? regc :
                (asel==0b011) ? regd :
                (asel==0b100) ? rege :
                (asel==0b101) ? regf :
                (asel==0b110) ? regg :
                (asel==0b111) ? regh :
                0;

        B = (bsel==0b000) ? rega :
                (bsel==0b001) ? regb :
                (bsel==0b010) ? regc :
                (bsel==0b011) ? regd :
                (bsel==0b100) ? rege :
                (bsel==0b101) ? regf :
                (bsel==0b110) ? regg :
                (bsel==0b111) ? regh :
                0;
        }
}

dp tbench_Register(out asel, bsel, storesel : ns(3);
                out storeenable : ns(1);
                out storedata : tc(32);
                in A,B : tc(32)){
        always {$display($dec,"A selector: ",asel," B selector: ",bsel,
                " Storage enable:",storeenable," Store selector: ",store
                " Store data: ", storedata," A: ",A," B: ",B);}
        sfg test_1 { asel=1; bsel=2; storesel=1;
                storeenable=1; storedata=2;}
        sfg test_2 { asel=1; bsel=1; storesel=2;
                storeenable=1; storedata=13;}
        sfg test_3 { asel=2; bsel=1; storesel=4;
                storeenable=1; storedata=37;}
        sfg test_4 { asel=1; bsel=4; storesel=4;
                storeenable=1; storedata=42;}
        sfg test_5 { asel=4; bsel=2; storesel=0;
                storeenable=0; storedata=0;}
}

fsm f_testbench(tbench_Register){
        initial s0;
        state s1,s2,s3,s4;
```

```
        @s0 (test_1) -> s1;
        @s1 (test_2) -> s2;
        @s2 (test_3) -> s3;
        @s3 (test_4) -> s4;
        @s4 (test_5) -> s0;
}

system registersystem {
        register(asel,bsel,storesel,storeenable,storedata,A,B);
        tbench_Register(asel,bsel,storesel,storeenable,storedata,A,B);
}
```

B.1.6 RAM

```
// Created by Jacob Gjerstrup, s113440
$option "vcd"

ipblock instmem(in address : ns(5); in wr,rd : ns(1); in idata : ns(32);
        out odata : ns(32)){

        iptype "ram";
        ipparm "size=64";
        ipparm "wl=32";
        ipparm "file=program.txt";
}

dp tbench_fetch(out adress : ns(5);
                out wr, rd : ns(1);
                out idata : ns(32);
                in odata : ns(32)){

        reg ad : ns(5);
        reg i_data, o_data : ns(32);
        $trace (address, "traceofInstrMemAddress.txt");
        $trace (wr, "traceofInstrMemWr.txt");
        $trace (rd, "traceofInstrMemRd.txt");
        $trace (idata, "traceofInstrMemIdata.txt");
        $trace (odata, "traceofInstrMemOdata.txt");
        sfg write{
                wr = 1;
                rd = 0;
                idata = i_data;
                o_data = odata;
                adress = ad;
                $display($cycle, "adress = ", ad, ",
```

```
                        in data = ", idata);
        }

        sfg read{
                wr = 0;
                rd = 1;
                idata = i_data;
                o_data = odata;
                adress = ad;
                $display($cycle, "adress = ", ad, ",
                        out data = ", odata);
        }

        sfg increase_ad  { ad = ad + 1;}
}

fsm t_fetch(tbench_fetch) {
   initial s0;
   @s0 (read, increase_ad) -> s0;

}

dp ipBlock {
        sig adress : ns(5);
        sig wr, rd : ns(1);
        sig idata, odata : ns(32);
        use instmem(adress, wr, rd, idata, odata);
        use tbench_fetch (adress, wr, rd, idata, odata);
}


system PC {
                ipBlock;
}
```

B.1.7 Assembly Code

```
//Created by Bastian Buch, s113432
//Final instruction set; 8 opcodes;
ADD
ADDI
SUB
FIDIV
BNEG
B
LOAD
OUT

//Assembly code; instruction count = 17
ADDI $1, $0, 30         //Initialization, BLOCK 1 begins here
ADDI $2, $0, 1          //Initialization
ADD $5, $0, 32          //This is the position of X[0] in the RAM
ADD $6, $0, 65          //This is the position of Y[0] in the RAM
LOAD $7, $98            //This is Pos
SUB $7, $7, $2          //Pos is reduced by 1, see explanation
ADD $3,$6,$7            //This is Y[0+(pos-1)], and therefore Y[Pos]
LOAD $3, $6             //This loads the value of Y[Pos] in register 3
ADD $4,$5,$7            //This is X[0+(pos)], and therefore X[Pos+1]. This makes the loop easier.
ADD $7, $7, $2          //Pos is increased by 1
SUB $1, $1, $2          //Substraction to check loop, BLOCK 2 begins here
BNEG 17                 //Goes to line 17
SUB $4,$4,$2            //Substracts 1 from X[pos+1]
LOAD $5, $4             //Loads the value of X[pos]
FIDIV $5,$5,30          // Divdes the value by 30
ADD $3, $3, $5          // Adds the value of X[pos] to the value of Y[pos]
B 12                    //Returns to line 11
OUT $3                  // Sets the new value as output
```