8-Bit Multicycle Processor Design and Implementation in Verilog

---

Benjamin Unger W4f

Supervisor: Samuel Lang
Kantonsschule Enge
December 17, 2018

Benjamin Unger
benjamin.unger@stud.ken.ch
8-Bit Multicycle Processor Design and Implementation in Verilog

Kantonsschule Enge
Steinentischstrasse 10
8002 Zürich

Abstract

This paper illustrates the process of designing a multi-cycle 8-bit computer. It starts with the design of my own assembly language and machine code. Then follows with the design of the microarchitecture and its implementation in Verilog. At the end a simple program is presented that can be run on my computer which calculates the fibonacci numbers.

Acknowledgement

A very obvious thanks goes to my supervisor Samuel Lang for overseeing my project and giving me writing tips on how to write certain paragraphs and with the bibliography which I could nod have done without him.
The ETH master student Simon also deserves a big thank you for giving me access to many lectures slides which helped me find better information about the topic.
Also thanks to my brother Jonathan for always answering questions that I had and helping me with latex which is awfully complicated.
The biggest thanks has to go to my mother because she spent hours digitising my diagrams. Without her I would have had to scan my messy hand-drawn diagrams.

Contents

1 Introduction

The goal of this project is to design my own functional 8-bit processor. This means designing my own assembly language and its machine language. Then designing an microarchitecture which is the implementation of my architecture. The microarchitecture will be implemented in Verilog, a commonly used hardware descriptive language to simulate and design digital circuits. And a simple program will be showcased to check if the architecture and microarchitecture were properly designed and implemented.

In this paper I will use the two terms "processor" and "computer" interchangeably because they actually mean the same thing at this size of a computer. If we think of a computer we think of a monitor, keyboard, mouse and a tower but actually it is just a thing that changes data depending on the data stored in it. The processor is what makes a computer a computer, so a processor with even a small memory may already be called a computer. And this is exactly the goal which I have for this project: To design a simple 8-bit processor.

The computer presented in this paper should not be fast in execution nor should it have many different instructions, it should just work. I had no prior experience in designing digital systems nor computer architectures so I was worried that the project might fail. This is why I kept everything simple so that I would not overburden myself. Still I wanted it to be a multi-cycle computer, which is a bit more complicated to design than a single-cycle computer. I made this decision, because the single-cycle computer has the big disadvantage that it needs to separate instruction and data memory.

Before starting this project I watched the Youtube series “Building an 8-bit computer!”[1] and read the book “Digital Design and Computer Architecture” by Sarah L. Harris and David Money Harris[2]. The latter was the foundation on which I designed my computer. Later when I seeked help with my project I got in contact with Simon Miescher a ETH student, who gave me access to lecture slides on computer engineering[3,4] which helped me getting slightly different information about this topic. It is also crucial to mention that this paper will not be able to go into great detail on how a computer works. This would take too long and is not the goal of

this project. If you want to understand this paper with no prior knowledge, then I advise you to read the book “Digital Design and Computer Architecture”[2].

1.1 Motivation

As a child I always wanted to know how a computer works. It was the only thing that I could not grasp how it worked, because you could hardly see inside. If you did, you did not see much. You only saw some green boards with some black boxes on it. Growing up I tried to find answers in the internet, but never found a solution that would satisfy my demands, because I could not piece together the behaviour of a computer out of fundamental physical laws. I did know about memory and that it is stored in ones and zeros and what a processor does. But understanding why those parts behave the way that they do was a mystery to me. It felt like small elves were controlling the computer because the computer seemed to think. After reading the book "Digital Design and Computer Architecture"[2] and finally understanding how a computer works I thought to myself that designing my own computer would be a great way of proving to myself that I really understand the matter.

2 Material and Methods

2.1 Architecture

2.1.1 Assembly Language

The first step in designing a processor is to define its instruction set. Those are the fundamental commands that make up every program. It should be able to compute everything I could possibly want. This means making it Turing complete. In a nutshell Turing completeness means that the computer will have to be able to manipulate any memory address and conditionally jump to any instruction. This is a very simple definition of Turing completeness. Proving that something is Turing complete formally would mean to show that it can simulate any other Turing machine, which is difficult and outside of the scope of this paper. For my architecture I tock inspiration from other instruction sets like the ARM[5], RISC-V[6] and MIPS[2] instructions sets which are commonly used architectures in processors today. Except for RISC-V which is an open source architecture which can hardly be found in any processor today but shows great promise of becoming popular in the future. From those I picked the most important ones making sure the architecture will be Turing complete. This is the list of simple instructions that I got:

- load word (lw)
- store word (sw)
- add (add)
- subtract (sub)
- move (mov)
- jump (j)
- conditional jump (jc)

At the time I did not know on which condition the conditional jump instruction will jump but I knew that it needs one to make the computer Turing complete. Later, while designing the control unit I decided to jump, if the ALUOut register reads zero and even implemented another instruction which jumps, if an overflow occurred. I made the decision because it makes programming much easier. Because deciding if

the result of a subtraction is negative or if overflow occurres can be evacuated in only one instruction rather than multiple ones.

After finishing and correcting the diagram of the microarchitecture I realised that I forgot to include the move instruction, but quickly found out that the add instruction with the register x0, which is hardwired to 0, is equivalent to a move instruction. Because of this redundancy I decided to omit the move instruction.

After almost completing the whole project I thought back to the Youtube series[1] and thought it would be nice to have a halt instruction which stops the computer. Meaning it would stop loading any new instructions after it. So that a computer could stop after completing a calculation.

Finally the instruction set looked like this:

- load word (lw)
- store word (sw)
- add (add)
- subtract (sub)
- jump (j)
- conditional jump on zero(jcz)
- conditional jump on overflow(jco)
- halt (hlt)

2.1.2 Machine Language

I always knew that my instructions would not be 8 bits long but rather 16 bits. I made this decision because I could not fit enough information into 8 bits (or at least I did not know how). My, till then, seven instructions already need at least three bits, which which would have used almost half the instruction space. So there would not be enough space left to fit in a memory address or up to three register addresses.

The inspiration for my instruction encodings came from the MIPS architecture[4]. For the order of bits I went with the little-endian form because it is the more intuitive one. These are the three instruction types that I came up with:

- W-type for lw and sw
- S-type for add and sub
- J-type for j, jcz, jco (and hlt)

For all three types the first four bits are reserved for the opcode. The four bits left enough room to encode all instructions and more if I wanted to define more in the future. The rest of the instruction is parcelled into registers and memory addresses where to read, write or jump to. Because I wanted my memory to have an 8-bit data output the processor will have to read each instruction in two steps, so I tried that no field starts in one half and ends in the other. I think reading instructions like this is necessary to keep the computer Turing complete. Even if this is not the case at least it gives interesting possibilities of programming the computer which will be shown in the chapter 2.4 programming.

The W-type instruction needs to contain a source (rs) or destination register (rd) and an address where to read or write the data to or from. The encoding of the W-Type instruction can be seen in figure 2.1.

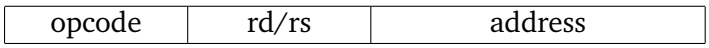

Fig. 2.1: W-type encoding.

The S-type instruction needs to be able to add or subtract the values of two registers and store the result to another. The instruction starts with the opcode like any other. Then it follows with the destination register where the result of the operation will be stored. This position was chosen because we already used this position for the destination register in the W-type instruction. This makes designing the decoder hardware easier. Finally it follows with the two source registers from which the values will be added or subtracted. But because subtraction does not have the commutative property the second register will be subtracted from the first.

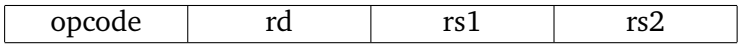

Fig. 2.2: S-type encoding.

the J-type instruction contains an opcode and only needs the memory address where to jump to, so four bits will be unused. Because the W-type already contains the memory address in the second half of the instruction that position will be used too. This means the four bits next to the opcode are don't-cares which are denoted as "xxxx" in figure 2.3.

Fig. 2.3: J-type encoding.

When I designed the control unit I gave every instruction its opcode. The first two bits are determined by the instruction type. The W-type got the bits 00, S-type the bits 01 and J-type 10. The second two are consecutively numbered by their appearance in table 2.1. This made designing the control unit much easier because

for each instruction type I only had to write one state and conditionally turn on some control lines depending on the second two bits. The only exception is the halt instruction for which the control unit will just check if the opcode equals 1011. The halt instruction also does not need an address so the address field can be filled with don't-cares too.

instruction	opcode
lw	0000
sw	0001
add	0100
sub	0101
j	1000
jcz	1001
jco	1010
hlt	1011

Tab. 2.1: Opcodes of all instructions.

2.2 Microarchitecture

The Microarchitecture is the implementation of an instruction set architecture in a given processor. In the following diagrams the blue lines indicate which connections are new, and black if they can be seen in previous diagrams.

2.2.1 State Elements

All state elements can be seen in figure 2.4 where the ports of the memory, register file and ALU can be clearly seen. Because of the multi-cycle design processor instruction memory and data memory are not separated. This means starting with a single memory block. It takes in an 8-bit address (A) and has an 8-bit data output (RD). To write to it there is an 8-bit data input. When the write enable port (WE) is 1 the value on the write data port will be stored at the memory address supplied to the address input at the rising edge of the clock.

Like many others, this processor has a register file which has three 4-bit address inputs, an 8-bit data input and two 8-bit data outputs. The first two address inputs (A1, A2) are to access data from registers and the third (A3) where to store the input data (WD3) when write enable (WE3) turns on. The register file contains 16 registers where register x0 is hardwired to the constant 0.

The ALU (Arithmetic Logic Unit) is the last fundamental element. It performs various mathematical and logical operations depending on the control signal (top). It takes in two operands (left) and computes the wanted result on the output line (right).

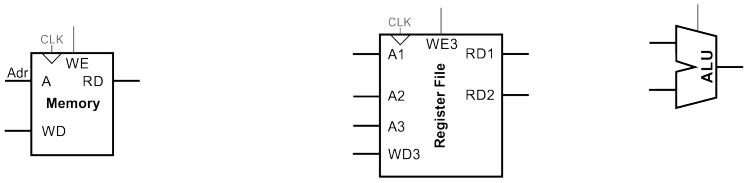

Fig. 2.4: State elements of the processor.

2.2.2 Datapath

Now we will connect the state elements introduced in the prior chapter and connect them, so that they can execute every instruction introduced in the chapter 2.1.1 Assembly Language.

The first step in executing an instruction is to load it from memory. So we will connect the program counter (PC) to the address input (A) of the memory. The program counter is a register that stores the address of the instruction that is to be executed. When the address of the program counter is supplied to the memory, it will read what is stored at that address. Because the two halves of one instruction are stored in two consecutive address locations, we will have to increment the program counter after reading the first half and then read the second half. The two halves are stored in two registers instead of one, because if we only had one we would lose the first halve while loading the second into the instruction register. The first four bits of the instruction register contain the opcode which is needed by the control unit. The control unit will be subject of the following chapter so the bits containing the opcode are routed to the top and are not connected to anything yet.

For the load word instruction the address specified in the instruction has to be read from memory. To read it, the second instruction register containing the address is routed to the address input of the memory. As the program counter is already hooked up to that port, we will have to put a multiplexer in between to control between program counter and RWAdr. After the new address is requested from memory the data is on the data line (RD). It can be routed to the write data input of the register file (WD3) to store it there. In the future we will also have to be able to write results of calculations to the register file so we will again have to use a

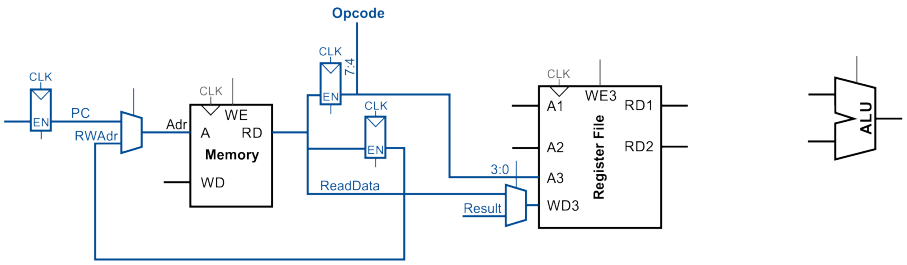

Fig. 2.5: Connections to load instructions and execute load word instructions.

multiplexer to choose between the two connections. All connections made until now can be seen in figure 2.5. We have successfully made all connections to execute the load word instruction.

We will continue with the store word instruction. For it we have to read a register from the register file and store it to memory. We connect the source register field of the instruction (located in the first instruction register) to A1 on the register file to read the requested register. Then we connect the output from RD1 on the register file to write data on the memory (WD) so that it can be stored there. The connection for the address where the data should be stored to is already made because the address field of both the store and load word instruction are in the same place. The added datapath is shown in figure 2.6.

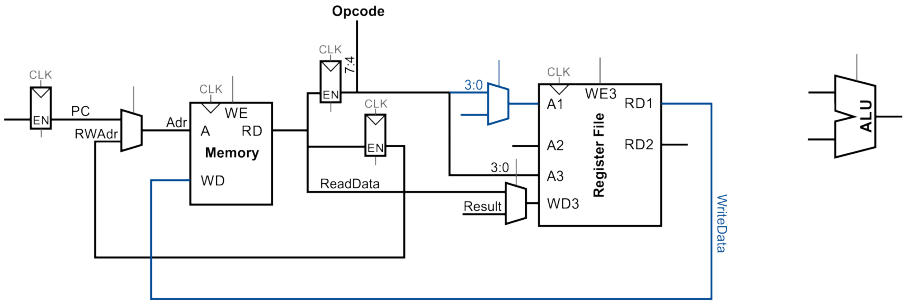

Fig. 2.6: Connections for the store word instruction.

If we proceed to do the same for all instructions we get a datapath that is depicted in figure 2.7.

Now the datapath is able to execute every instruction. But it also needs to be able to increment the program counter. We will reuse the ALU but because it is already connected to the register file to execute the add and sub instructions we have to put two multiplexers in front of the two inputs. Now we can connect the program counter and a hardwired 1 to the two inputs to increment the program counter. The

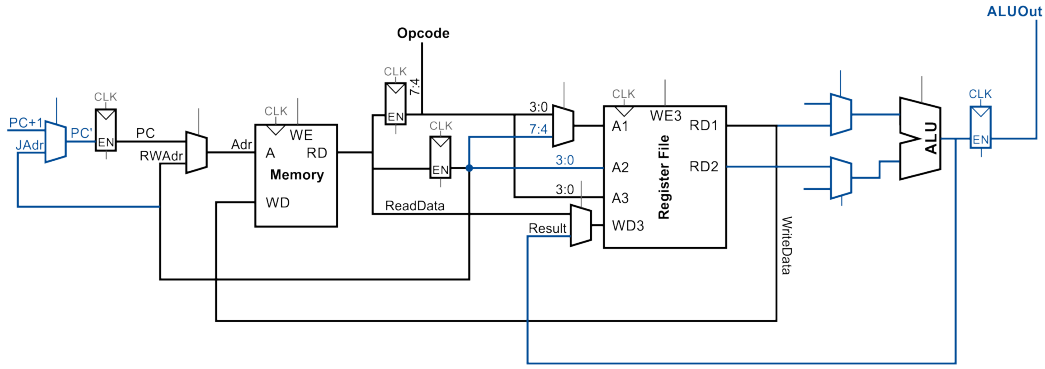

Fig. 2.7: All connections to execute all instructions.

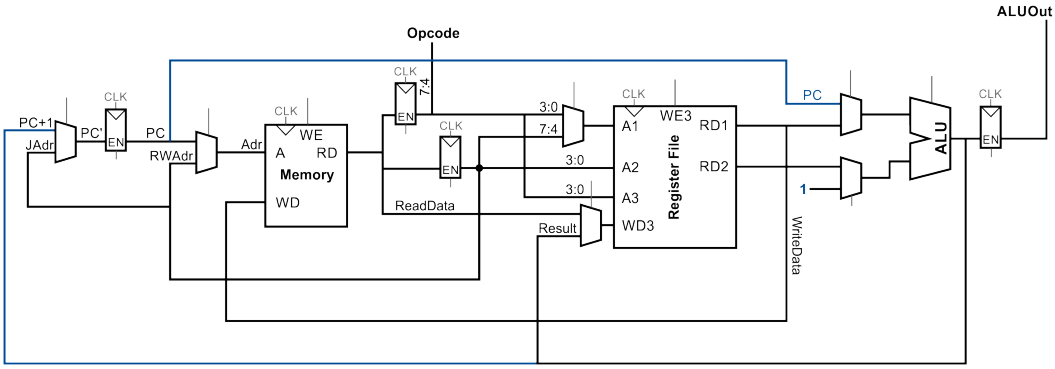

Fig. 2.8: Finished datapath.

result of the calculation has to be stored back to the program counter so the output of the ALU is connected to the PC+1 port on the multiplexer in front of the program counter. The finished datapath is depicted in figure 2.8.

2.2.3 Control Unit

The control unit, like the name suggests controls the flow of data through the datapath constructed in the previous chapter. It needs to turn on and off all control lines (grey) shown in figure 2.9. The control unit connects to all control lines and takes in the opcode and ALUOut register to set them. Designing the control unit was by far the most complicated part of this project. Reading the slides of the ETH lecture "Multi-cycle MIPS Processor"[3] helped me to understand how to construct the state diagram seen in figure 2.10. In it the two load cycles for loading each half of the instruction and decoding can clearly be seen. For each instruction type (and the halt instruction) there is only one state. This is possible because the control unit is a mealy machine where its outputs also depend on its input and not only on the current state. In the jump state for example the PCEnable only turns on, if the second two bits of the opcode are 00, meaning that the instruction is an unconditional jump or the second two bits encode a conditional jump instruction and the corresponding flag is 1.

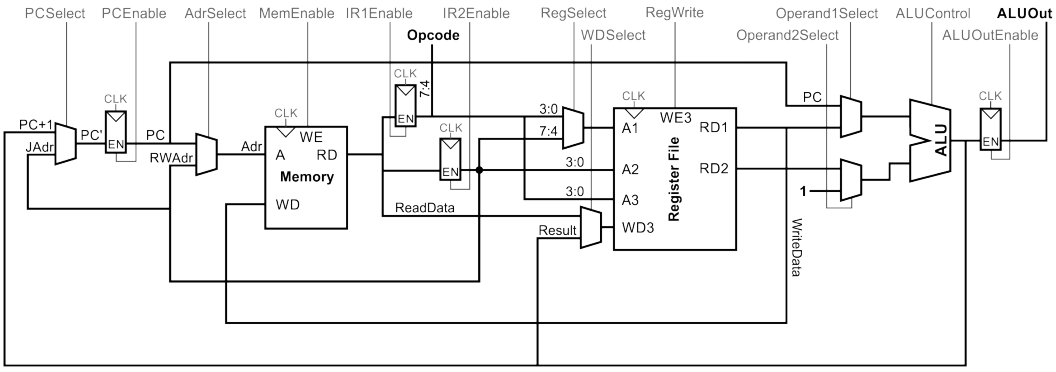

Fig. 2.9: Computer with all control lines drawn in grey.

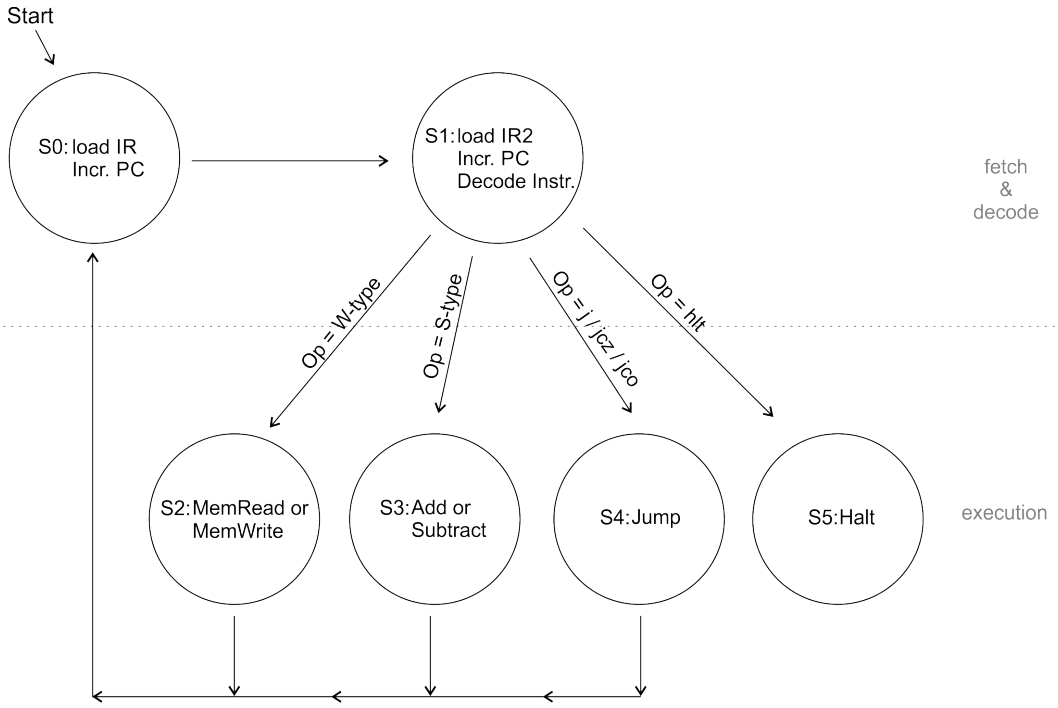

Fig. 2.10: State diagram of the control unit.

Now we will take a closer look at the state diagram of the control unit with an example. The first step (S0) in executing any instruction is to fetch the instruction from memory at the address held in the program counter. At the same time program counter is incremented. Doing this simultaneously saves time and is possible because no two state elements are used at the same time. In the second state (S1) the second instruction register is loaded with the the incremented address stored in the program counter and the program counter is incremented again so that later the next instruction can be read. During the transition to the execution states the instruction gets decoded. Decoding the instruction means to determine the next state depending on the opcode. Let's say for example that the just loaded instruction is an add instruction. So the control unit transitions to S2. In this state the control unit reads the two source register from the register file and sets the multiplexers so that the two values get fed to the ALU. The control unit looks at the second two bits of the opcode which are 01 in the case of an add instruction, and sets the ALUControl so that the ALU performs addition. It also sets the WDSelect signal so that the result is fed to the WD3 port and turns on the RegWrite signal so that it is stored back to the register file. After successfully executing the instruction the state is set to S0 and the whole process starts again.

2.3 Implementation in Verilog

During the implementation of the design in Verilog there were no serious issues except some minor bugs. Implementing the datapath was very straightforward but the control unit was quite difficult to implement so I read the paper "Multicycle Processor Design in Verilog"[7] to get inspiration. While implementing I did find two flaws in my design:

The first was that I did not split my instruction register in two. This would have been a problem because if the control unit tried to load the second half of the instruction the first half would have been overwritten.

The second mistake that I noticed was that the ALUOut register was wired to the program counter. There are actually two things wrong with that: The first is that each instruction would have taken one clock tick longer to execute and the other is that even though the program counter could have incremented correctly, the ALUOut register loses its content and a jump instruction processes incorrectly.

To make my code easier I did not implement the ALU like depicted in the diagrams. I included the ALUOut register in the ALU itself and made it also evaluate the zero flag and overflow flag, which turns on if the last computed equals to zero or overflow

occurred. So the only difference between the diagram and Verilog code is that the flags which tell the control unit wether to jump or not are not evaluated in the control unit but rather in the ALU itself. All files can be found in the appendix.

2.4 Programming

This was the most fun part of the whole project. During programming I eliminated the last bugs and made sure that the computer operates correctly. To test the computer thoroughly I would have needed to test all kinds of possible programs. As it is extensive to write machine code, I just wrote some typical programs like the program that calculates the fibonacci numbers which is being presented here. Writing machine code is like writing the programming lannguage Brainfuck. It is a great challenge but causes headache after 20 minutes because trying to combine simple instructions to perform more complex tasks is almost impossible.

As a short reminder the fibonacci numbers are defined like this:

$$F_0 = 0,\ F_1 = 1 \quad \text{and} \quad F_{n+1} = F_n + F_{n-1}$$

The program that I came up with looks like this:

```
 0 0000_0010 //lw 2, 200
 1 1100_1000
 2 0100_0011 //add 3, 1, 2
 3 0001_0010
 4 0100_0001 //add 1, 2, 0
 5 0010_0000
 6 0100_0010 //add 2, 3, 0
 7 0011_0000
 8 1000_0000 //j 2
 9 0000_0010
10
11 @c8 //address 200
12 0000_0001
```

You can clearly see the two halves of any instructions being stored in two memory location right after another. In line 0 for example we can read that the opcode is 0000 which means that the instruction is a load word instruction. The second four bits are 0010 which translates to a 2, meaning that the loaded memory is getting written to register x2. And in line 1 we can read the address 1100_1000 which equals to 200. In the comments you can see each translation in assembly language so it can easily be read by a human.

This program starts by loading the value from memory address 200 where the number 1 is stored and stores it in register x2. After the first instruction the contents of register x1 is 0. Then in row 2 of the program, register x1 and x2 are being added to each other and the result is stored in register x3. So now there is a 1 stored in register x3. One might see the fibonacci numbers already start appearing. After that we want to move the contents of register x2 to register x1 and from x3 to x2. To do that we use the add instruction in row 4 and 6 of the program. After the shift of values, register 1 contains a 1 like expected of the fibonacci numbers. Then we want to jump unconditionally back to address 2 to repeat the process of addition and shifting of values to calculate all the fibonacci numbers up to 255. This is because the registers are only 8-bits wide and can not store bigger values.

In the appendix there is another program which calculates the nth prime number which will not be showcased. I will only explain how to use it because its user interface is literally changing memory address 200 to the wanted n. The program is quite long and it uses some interesting ways of programming which I mentioned before and will briefly explain. For example in line 0 of the program the first instruction loads the first half of itself. But why does that make sense? Because this address not only encodes an opcode and a register address but can also simply be read as a 1 which is the value that the program needs in register x1. Another interesting instruction is the sub instruction on line 22. It calculates some difference and stores it to register x0. This seems to make no sense because it would lose the result but if we look at the next instruction we can figure out why the subtraction is not pointless. The sub instruction does not store back the result to the register file but still stores it in the ALUOut register which is simply read by the control unit to evaluate if it should jump or not. This trick is not necessary but might be useful if the program uses all 16 registers and the subtraction would otherwise overwrite some other data.

3 Results and Future Direction

Running the Verilog files creates a .vcd (variable change dump) file, which shows the values of each register and wire at any moment in time. To view the .vcd files I utilised a free software tool called “Scansion”. In Figure 3.1 and 3.2 you can see how opening such a file in Scansion looks. To check if the processor is functioning correctly we have to manually check if all registers and wires contain the wanted value at every moment in time. First will check if a single instruction executes correctly by looking at figure 3.1.

Before the first rise of the clock the control unit is in state 0 and the program counter is reading 0. The control unit sets Operand2Select and Operand1Select to 0 so that the ALU takes in the value of the program counter and the hardwired 1. ALUControl is set to 1 so it adds the two and calculates the next value for the program counter. PCSelect chooses the computed 1 and the correct result can be seen at the top at the list: nextPC = 1. PCEnable is high, like it should, so that it loads the 1 into the program counter. At the same time the value held in the program counter is routed to memory and memoryAddress = 0 which reads the first halve of the instruction: dataOut = 00000010. IR1Enable is high so that the value is loaded into the first instruction register. After the first rise of the clock the program counter took on the value 1 given to it and instructionReg1 contains the first half of the instruction.

Now the control unit increments the program counter the same way again and sets IR1Enable low and IR2Enable high so that the second half of the instruction gets loaded into the second instruction register. Meanwhile it looks at the opcode which is 0000 which means the next state will be S2.

At the next clock tick we can see the state actually being S2. Because of that AdrSelect is set to 1 so that the memory address 200 stored in the instructionReg2 is fed to memory and can be read. This can be seen at the time = 4ps, where memoryAddress = 200 which returns the value dataOut = 1. WDSelect turns on so that the output of memory gets to the WD port of the register file. The address of the destination register is 2 according to the instruction and we can actually see RFA3 being 2 so that the 1 is stored there. Because the second two bits of the opcode read 00 the control unit knows to enable RegWrite so the data is stored in the wanted register. After the third rising edge of the clock we can finally see the 1 stored in register 2.

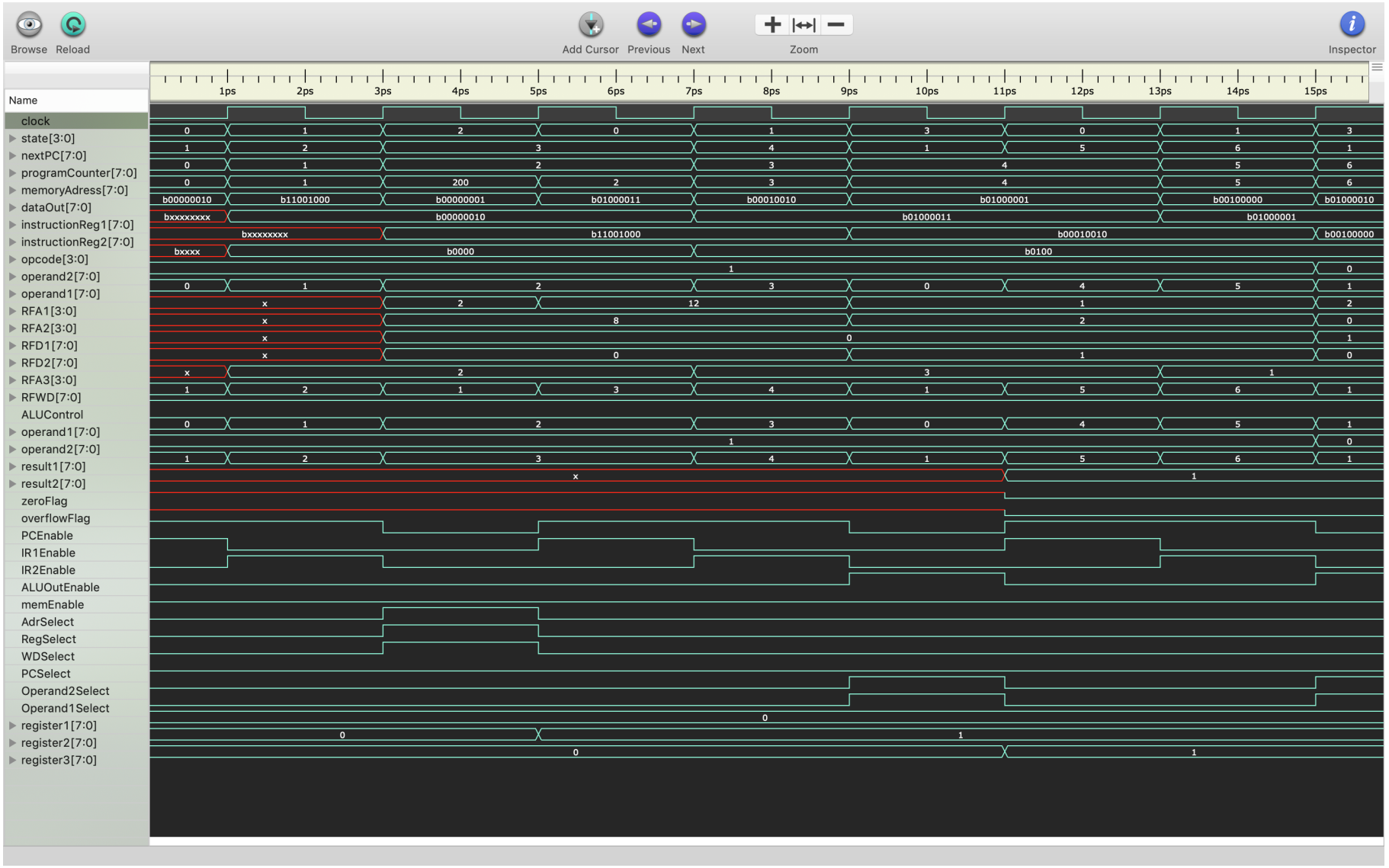

Fig. 3.1: View of three instructions and its control signals.

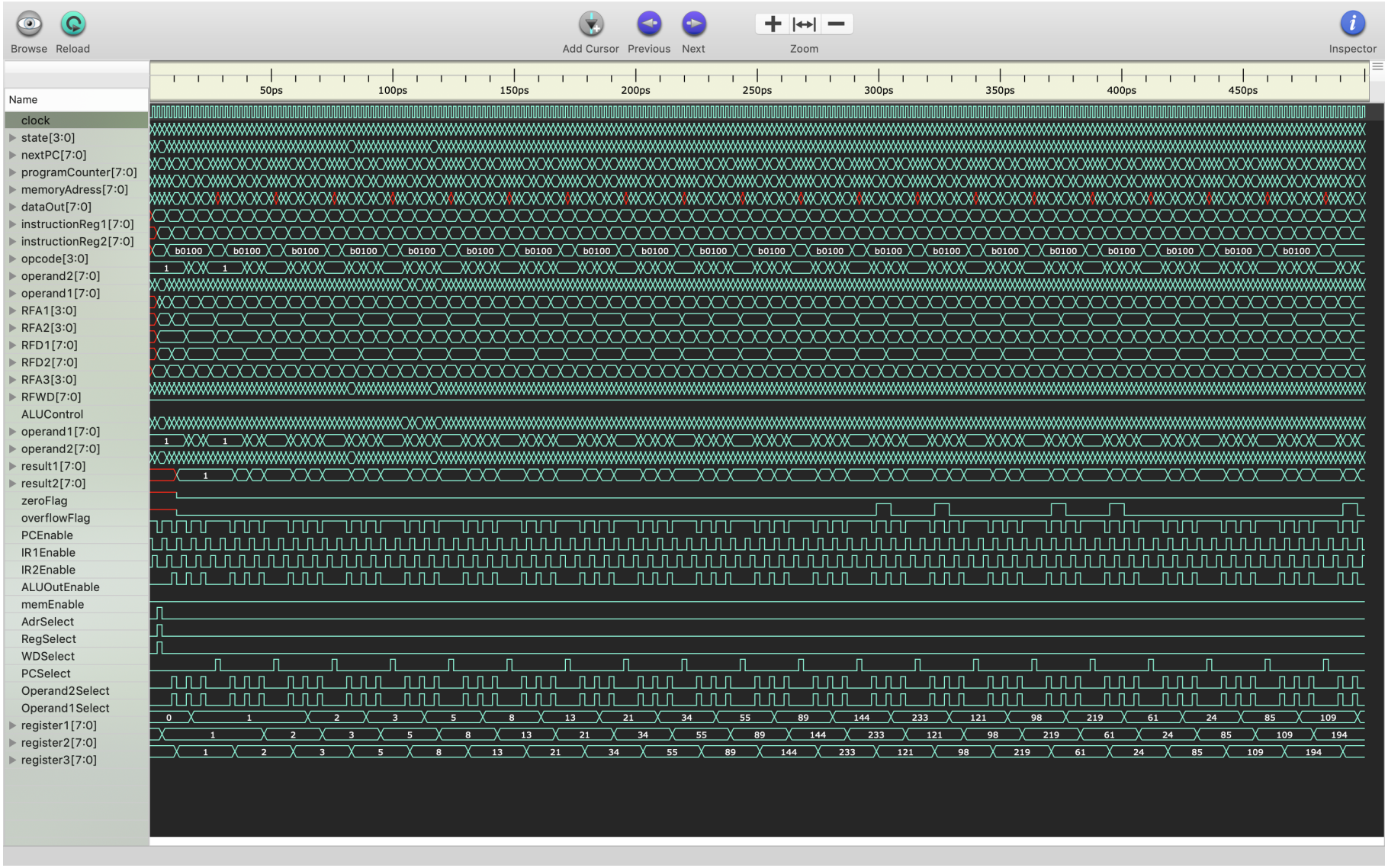

Fig. 3.2: View over many instructions.

And the first instruction executed correctly. This is how to check each instruction wether the computer works correctly or not.

Now we also have to check if the program I wrote calculates the fibonnaci numbers correctly. Here we can not do much more than zoom out and look at the values that are stored in register 1. In figure 3.2 we can clearly see the fibonacci numbers appearing in register 1 one after the other. Right at t = 300ps the sum is greater than 255 so the sum will just wrap around. As you can see the overflow flag is 1 until the next calculation is performed.

The computer is working perfectly but still for the Future I have three ideas how I could improve my computer:

The first and easiest is to implement more instructions. Especially S-Type ones which would make programming easier. For example a shift instruction which would open many possibilities like division and easy multiplication by powers of two.

The second is to add a stack that would allow the usage of functions. functions are very useful for many applications. For example: Sorting, recursion and reusability of routine operations.

Thirdly I could change the design from a multi-cycle to a pipelined processor which would enable faster clock cycles but would also make the control unit much more complex. This is not easy and means creating a whole new datapath.

But all in all I am satisfied with the computer. It works great and is really fun to program. In the future I might load my verilog files onto an FPGA to not only simulate it but realise my computer in hardware.

List of Figures

List of Tables

Bibliography

[1] B. Eater, Building an 8-bit breadboard computer! https://www.youtube.com/playlist?list=PLowKtXNTBypGqImE405J2565dvjafglHU last accessed 16 December 2018

[2] S. L. Harris and D. M. Harris, Digital Design and Computer Architecture. MA: Elsevier, 2016.

[3] S. Capkun and F. K. Gürkaynak, Multi-cycle MIPS Processor. http://www.syssec.ethz.ch/content/dam/ethz/special-interest/infk/inst-infsec/system-security-group-dam/education/Digitaltechnik_14/21_Architecture_MultiCycle.pdf last accessed 16 December 2018

[4] L. Thiele, Technische Informatik 1, 4 – Prozessor Einzeltaktimplementierung. https://lectures.tik.ee.ethz.ch/ti1/slides/4.pdf accessed 16 December 2018

[5] Arm THUMB Instruction Set. https://ece.uwaterloo.ca/~ece222/ARM/ARM7-TDMI-manual-pt3.pdf accessed 16 December 2018

[6] The RISC-V Instruction Set Manual Volume I: User-Level ISA, SiFive Inc, 2017. https://riscv.org/specifications/# accessed 16 December 2018

[7] J. Chargo and M. Falat, Multicycle Processor Design in Verilog. http://read.pudn.com/downloads164/doc/748258/MulticycleProcessor.pdf accessed 16 December 2018

APPENDIX

Computron.v

```
1  `timescale 1ps/1ps
2
3  `include "RegisterFile.v"
4  `include "ALU.v"
5  `include "Memory.v"
6  `include "ControlUnit.v"
7
8  module Computron;
9
10 reg [7:0] programCounter;
11 reg [7:0] instructionReg1;
12 reg [7:0] instructionReg2;
13 reg clock;
14
15 wire [7:0] memoryAdress;
16 wire [7:0] dataOut;
17 wire [3:0] RFA1;
18 wire [3:0] RFA2 = instructionReg2[3:0];
19 wire [3:0] RFA3 = instructionReg1[3:0];
20 wire [7:0] RFD1;
21 wire [7:0] RFD2;
22 wire [7:0] operand1;
23 wire [7:0] operand2;
24 wire [7:0] result1;
25 wire [7:0] result2;
26
27 wire [7:0] RFWD;
28 wire [7:0] nextPC;
29
30 wire PCEnable;
31 wire IR1Enable;
32 wire IR2Enable;
33 wire ALUOutEnable;
34 wire memEnable;
35 wire regEnable;
36
37 wire zeroFlag;
38 wire overflowFlag;
```

```
39
40 wire WDSelect;
41 wire Operand1Select;
42 wire Operand2Select;
43 wire RegSelect;
44 wire AdrSelect;
45 wire PCSelect;
46
47
48 Memory mem(
49    .data(dataOut),
50    .clock(clock),
51    .writeEnable(memEnable),
52    .address(memoryAdress),
53    .writeData(RFD1)
54 );
55
56 RegisterFile rf(
57    .data1(RFD1),
58    .data2(RFD2),
59    .clock(clock),
60    .writeEnable(regEnable),
61    .address1(RFA1),
62    .address2(RFA2),
63    .address3(RFA3),
64    .writeData(RFWD)
65 );
66
67 ALU alu(
68    .result1(result1),
69    .result2(result2),
70    .zeroFlag(zeroFlag),
71    .overflowFlag(overflowFlag),
72    .clock(clock),
73    .ALUControl(ALUControl),
74    .ALUEnable(ALUOutEnable),
75    .operand1(operand1),
76    .operand2(operand2)
77 );
78
79 ControlUnit cu(
80    .PCEnable(PCEnable),
81    .IR1Enable(IR1Enable),
82    .IR2Enable(IR2Enable),
83    .ALUOutEnable(ALUOutEnable),
84    .PCSelect(PCSelect),
85    .AdrSelect(AdrSelect),
86    .RegSelect(RegSelect),
87    .WDSelect(WDSelect),
88    .Operand1Select(Operand1Select),
89    .Operand2Select(Operand2Select),
```

```
90    .ALUControl(ALUControl),
91    .memEnable(memEnable),
92    .regEnable(regEnable),
93    .clock(clock),
94    .opcode(instructionReg1[7:4]),
95    .zeroFlag(zeroFlag),
96    .overflowFlag(overflowFlag)
97 );
98
99
100 always @ (posedge clock) begin
101   if (PCEnable) programCounter <= nextPC;
102   if (IR1Enable) instructionReg1 <= dataOut;
103   if (IR2Enable) instructionReg2 <= dataOut;
104 end
105
106
107 assign nextPC = PCSelect ? instructionReg2 : result1;
108 assign RFWD = WDSelect ? dataOut : result1;
109 assign operand1 = Operand1Select ? RFD1 : programCounter;
110 assign operand2 = Operand2Select ? RFD2 : 8'b0000_0001;
111 assign RFA1 = RegSelect ? instructionReg1[3:0] : instructionReg2[7:4];
112 assign memoryAdress = AdrSelect ? instructionReg2 : programCounter;
113
114
115
116
117 initial  begin
118   $dumpfile ("Computron.vcd");
119   $dumpvars;
120   programCounter = 0;
121   clock = 0;
122   #500 $finish;
123 end
124
125 always begin
126    #1 clock = ~clock;
127 end
128
129 endmodule
```

Memory.v

```
1 module Memory(
2   output [7:0] data,
3   input clock,
4   input writeEnable,
5   input [7:0] address,
6   input [7:0] writeData
7 );
```

```
 8 reg [7:0] memory [255:0];
 9
10 initial $readmemb("FibbonaciNrs", memory);
11
12 always @ (posedge clock) begin
13   if (writeEnable) memory[address] <= writeData;
14 end
15
16 assign data = memory[address];
17
18 endmodule
```

RegisterFile.v

```
 1 module RegisterFile(
 2   output [7:0] register0,
 3   output [7:0] register1,
 4   output [7:0] register2,
 5   output [7:0] register3,
 6   output [7:0] register4,
 7   output [7:0] register5,
 8   output [7:0] register6,
 9   output [7:0] register7,
10   output [7:0] register8,
11   output [7:0] register9,
12   output [7:0] register10,
13   output [7:0] register11,
14   output [7:0] register12,
15   output [7:0] register13,
16   output [7:0] register14,
17   output [7:0] register15,
18   output [7:0] data1,
19   output [7:0] data2,
20   input clock,
21   input writeEnable,
22   input [3:0] address1,
23   input [3:0] address2,
24   input [3:0] address3,
25   input [7:0] writeData
26 );
27
28 reg [7:0] registerFile [15:0];
29 reg [4:0] i;
30
31 initial begin
32   for (i = 0; i < 16; i++)
33     registerFile[i] = 8'b0000_0000;
34 end
35
36 always @ (posedge clock) begin
```

```
37   if (address3)
38     if (writeEnable) registerFile[address3] <= writeData;
39 end
40
41 assign data1 = registerFile[address1];
42 assign data2 = registerFile[address2];
43
44 assign register0 = registerFile[0];
45 assign register1 = registerFile[1];
46 assign register2 = registerFile[2];
47 assign register3 = registerFile[3];
48 assign register4 = registerFile[4];
49 assign register5 = registerFile[5];
50 assign register6 = registerFile[6];
51 assign register7 = registerFile[7];
52 assign register8 = registerFile[8];
53 assign register9 = registerFile[9];
54 assign register10 = registerFile[10];
55 assign register11 = registerFile[11];
56 assign register12 = registerFile[12];
57 assign register13 = registerFile[13];
58 assign register14 = registerFile[14];
59 assign register15 = registerFile[15];
60
61 endmodule
```

ALU.v

```
1  module ALU(
2    output [7:0] result1,
3    output [7:0] result2,
4    output zeroFlag,
5    output overflowFlag,
6    input clock,
7    input ALUControl,
8    input ALUEnable,
9    input [7:0] operand1,
10   input [7:0] operand2
11 );
12
13 reg [8:0] internalResult;
14 reg [8:0] ALUResult;
15
16 always @ (*) begin
17   internalResult = ALUControl ? operand1 + operand2 : operand1 - operand2
       ;
18 end
19
20 always @ (posedge clock) begin
21   if(ALUEnable)
```

```
22    ALUResult = internalResult;
23 end
24
25 assign zeroFlag = !ALUResult;
26 assign overflowFlag = ALUResult[8];
27
28 assign result1 = internalResult[7:0];
29 assign result2 = ALUResult[7:0];
30
31 endmodule
```

ControlUnit.v

```
 1 module ControlUnit(
 2    output PCEnable,
 3    output IR1Enable,
 4    output IR2Enable,
 5    output ALUOutEnable,
 6    output PCSelect,
 7    output AdrSelect,
 8    output RegSelect,
 9    output WDSelect,
10    output Operand1Select,
11    output Operand2Select,
12    output ALUControl,
13    output memEnable,
14    output regEnable,
15    input clock,
16    input [3:0] opcode,
17    input zeroFlag,
18    input overflowFlag);
19
20 reg PCEnable, IR1Enable, IR2Enable, ALUOutEnable, PCSelect,AdrSelect,
       RegSelect, WDSelect, Operand1Select, Operand2Select, ALUControl,
       memEnable, regEnable;
21
22 parameter loadInstruction = 0;
23 parameter decodeInstruction = 1;
24 parameter loadStore = 2;
25 parameter compute = 3;
26 parameter jump = 4;
27 parameter halt = 5;
28
29 reg [3:0] state = loadInstruction;
30 reg [3:0] nextstate;
31
32 always @ (posedge clock) state = nextstate;
33
34 always @ (state, opcode) begin
35    case(state)
```

```
36      loadInstruction: begin //Load Instruction
37        PCEnable = 1;
38        IR1Enable = 1;
39        IR2Enable = 0;
40        ALUOutEnable = 0;
41
42        PCSelect = 0;
43        AdrSelect = 0;
44        RegSelect = 0;
45        WDSelect = 0;
46        Operand1Select = 0;
47        Operand2Select = 0;
48        ALUControl = 1;
49
50        memEnable = 0;
51        regEnable = 0;
52
53        nextstate = decodeInstruction;
54      end
55
56      decodeInstruction: begin //Load Instruction
57        IR1Enable = 0;
58        IR2Enable = 1;
59
60        //Decode Instruction
61        if (opcode == 4'b1011) begin
62          nextstate = halt; //hlt
63        end
64        else if (opcode[3:2] == 2'b00) begin
65          nextstate = loadStore; //lw, sw
66        end
67        else if(opcode[3:2] == 2'b01) begin
68          nextstate = compute; //add, sub
69        end
70        else if(opcode[3:2] == 2'b10) begin
71          nextstate = jump; //j, jcz, jco
72        end
73      end
74
75      loadStore: begin //Loadword, Storeword
76        PCEnable = 0;
77        IR1Enable = 0;
78        IR2Enable = 0;
79
80        AdrSelect = 1;
81        RegSelect = 1;
82        WDSelect = 1;
83        if (opcode[0] == 0) regEnable = 1;
84        else memEnable = 1;
85        nextstate = loadInstruction;
86      end
```

```
87
88      compute: begin //Compute Sum or Difference and store in ALUOut
89        PCEnable = 0;
90        IR1Enable = 0;
91        IR2Enable = 0;
92        ALUOutEnable = 1;
93
94        RegSelect = 0;
95        Operand1Select = 1;
96        Operand2Select = 1;
97        regEnable = 1;
98        WDSelect = 0;
99
100       if (opcode[0] == 0) ALUControl = 1;
101       else ALUControl = 0;
102
103       nextstate = loadInstruction;
104     end
105
106     jump: begin //jump
107       PCEnable = 0;
108
109       if (opcode[1:0] == 2'b00) PCEnable = 1;
110       else if (opcode[1:0] == 2'b01 && zeroFlag == 1) PCEnable = 1;
111       else if (opcode[1:0] == 2'b10 && overflowFlag == 1) PCEnable = 1;
112
113       IR1Enable = 0;
114       IR2Enable = 0;
115
116       PCSelect = 1;
117       nextstate = loadInstruction;
118     end
119
120     halt: begin
121       PCEnable = 0;
122       IR1Enable = 0;
123       IR2Enable = 0;
124       ALUOutEnable = 0;
125     end
126
127   endcase
128 end
129
130 endmodule
```

FibonacciNrs

```
0 0000_0010 //lw 2, 200
1 1100_1000
2 0100_0011 //add 3, 1, 2
```

```
 3 0001_0010
 4 0100_0001 //mov 1, 2
 5 0010_0000
 6 0100_0010 //mov 2, 3
 7 0011_0000
 8 1000_0000 //j 3
 9 0000_0010
10
11 @c8
12 0000_0001
```

nthPrime

```
 0 0000_0001 //lw 1 0 (1)
 1 0000_0000
 2 0000_0010 //lw 2 200
 3 1100_1000
 4 0000_0011 //lw 3 2 (2)
 5 0000_0010                //initialization
 6 0101_0010 //sub 2 2 1
 7 0010_0001
 8 1001_0000 //jcz 12
 9 0000_1100
10 1000_0000 //j 14
11 0000_1110
12 1011_0000 //hlt
13 0000_0000                //check if finished
14 0100_0011 //add 3 3 1 //check next prime candidate
15 0011_0001
16 0000_0101 //lw 5 0 // initialize mod = 1
17 0000_0000
18 0100_0100 //add 4 3 0 //mov prime to check position
19 0011_0000
20 0100_0101 //add 5 5 1 // mod++
21 0101_0001
22 0101_0000 //sub 0 5 4 //check if mod == prime
23 0101_0100
24 1001_0000 //jcz 6
25 0000_0110
26 0101_0100 //sub 4 4 5
27 0100_0101
28 1001_0000 //jcz 14
29 0000_1110
30 1010_0000 //jco 18
31 0001_0010
32 1000_0000 //j 24
33 0001_1000
34
35 @c8
36 0000_0111 //n
```

Declaration

Hereby I declare that I have completed my work solely and only with the help of the references mentioned.

Zürich, December 17, 2018

Benjamin Unger