10 Instructions CPU

Cocis George

30431

Technical University of Cluj-Napoca

11.10.2020

Table of contents

1.Introduction

1.1 Project Proposal

The purpose of this project is to design and implement a CPU (central processing unit) with an instruction set consisting of 10 instructions, using the IDE provided by vivado. The CPU executes sequences of stored instructions, whose results can be stored either in some internal registers, or in the slower but less expensive main memory.

The following set represents the 10 chosen instruction, along with their respective functions:

1. Add → D = A + B
2. Subtract → D = A – B
3. Logical Or → D = A or B
4. Add with Carry → D = A + B + Carry
5. Subtract with Carry → D = A – B - Carry
6. Reverse Subtract → D = B - A
7. Load → D = 32-bit Immediate Value
8. Compare → D = compare (A, B)
9. Store → Memory[A] = D
10. Move → D = A

The set of operations will be implemented using assembly instructions. As seen above, the number of required registers differs from one instruction to another. Generally, we need destination registers, source registers and immediate values.

1.2 Weekly Plan

Week 1: Project choice – CPU with 10 instructions

Week 3: Project proposal, brief documentation about the project

Week 5, 7, 9, 11: Implementation of the CPU instructions and the complementary components with their respective functionalities

Making of documentation – detailed description of the project

Final adjustments

Week 13: Presentation

2. Bibliographic Study

“The CPU performs basic, arithmetic, logic, controlling, and input/output (I/O) operations specified by the instructions in the program. The fundamental operation of most CPUs, regardless of the physical form they take, is to execute a sequence of stored instructions that is called a program. The instructions to be executed are kept in some kind of computer memory. Nearly all CPUs follow the fetch, decode and

execute steps in their operation, which are collectively known as the instruction cycle. " – see Image 4.2; [1]

"The arithmetic logic unit (ALU) is a digital circuit within the processor that performs integer arithmetic and bitwise logic operations. The inputs to the ALU are the data words to be operated on (called operands), status information from previous operations, and a code from the control unit indicating which operation to perform. Depending on the instruction being executed, the operands may come from internal CPU registers or external memory, or they may be constants generated by the ALU itself." – see images 3.1 & 3.2; [1]

The Control Unit fetches the instruction from the main memory and decodes it, converting it into signals that control other components of the CPU, such as the ALU. The opcode of each instruction indicates which operation to be executed, and the remaining fields provide additional information required in order to perform the operation, such as operands. These operands can be specified as a either a constant value (immediate value), or as the location of a value that may be either a register or a memory address, determined by the addressing mode.

Whenever an operation is to be executed, the ALU inputs are connected to a pair of operand sources, and the ALU is configured to specifically execute the required information, with respect to the instruction's opcode. The result of the required operation will appear as the output, which is connected to either the main memory or a register.

3. Analysis

3.1 RTL (Register-transfer level) Expressions

The RTL is an intermediate representation that is used to describe the flow of signals (data) between the registers, and the operations performed on the signals. We need it in order to obtain the data path within the CPU.

- Add: Rd <- Rn + shifter_operand
- Sub: Rd <- Rn – shifter_operand
- Orr: Rd <- Rn OR shifter_operand
- Adc: Rd <- Rn + shifter_operand + Carry Flag
- Sbc: Rd <- Rn – shifter_operand – Carry Flag
- Rsb: Rd <- shifter_operand - Rn
- Cmp: CPSR flags <- Rn – shifter_operand
- Ldr: Rd <- mem32[address]
- Str: mem32[address] <- Rd
- Mov: Rd <- shifter_operand

3.2 Instruction Encoding

The CPU will have the following instruction format:

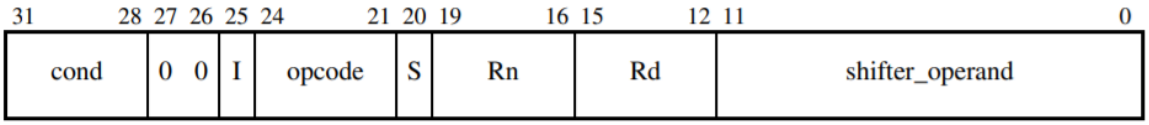


Image 3.1 [1]

The bits within the encoding mean, as follows:

I bit – Distinguishes between immediate value/register form of <shifter_operand>

S bit – Signifies that the instruction updates the condition codes

Rn – First source operand register

Rd – Destination Register

shifter_operand – Second source operand (immediate value / register)

Add: 0100_00_0/1_0100_1_Rn_Rd_shifter.operand

Sub: 0010_00_0/1_0010_0_Rn_Rd_shifter.operand

Orr: 1100_00_0/1_1100_0_Rn_Rd_shifter.operand

Adc: 0101_00_0_0101_1_Rn_Rd_shifter.operand

Sbc: 0110_00_0_0110_1_Rn_Rd_shifter.operand

Rsb: 0011_00_0/1_0011_0_Rn_Rd_shifter.operand

Cmp: 1010_00_0/1_1010_1_Rn_x_shifter.operand

Mov: 1101_00_0/1_1101_0_x_Rd_shifter.operand

As for the LDR and STR instructions, the following format will be used:

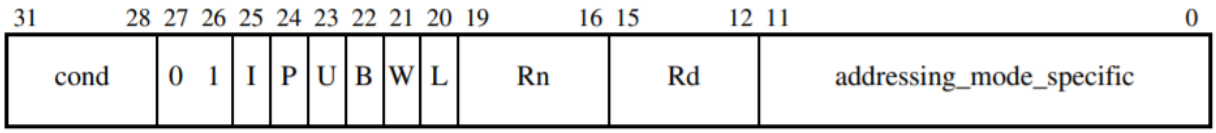


Image 3.2 [1]

The bits within the encoding mean, as follows:

L bit – distinguishes between Load (L = 1) and Store (L = 0)

B bit – distinguishes between an unsigned byte (B = 1) and a word (B = 0) access

Rn – specifies the base register used by addressing_mode

Rd – specifies the register whose contents are to be loaded or stored

I bit – distinguishes between offset sources (0 for immediate, 1 for registers)

P bit == 0 for load & store base

== 1 for load & store base + offset

U bit – indicates whether the offset is added to the base or subtracted from the base

W bit == 0 for base + offset

== 1 for base - offset

Ldr: 1110_01_0/1_0/1_1_0/1_0/1_1_addresing.mode

Str: 1111_01_0/1_0/1_0_0/1_0/1_0_addresing.mode

3.3 Datapath

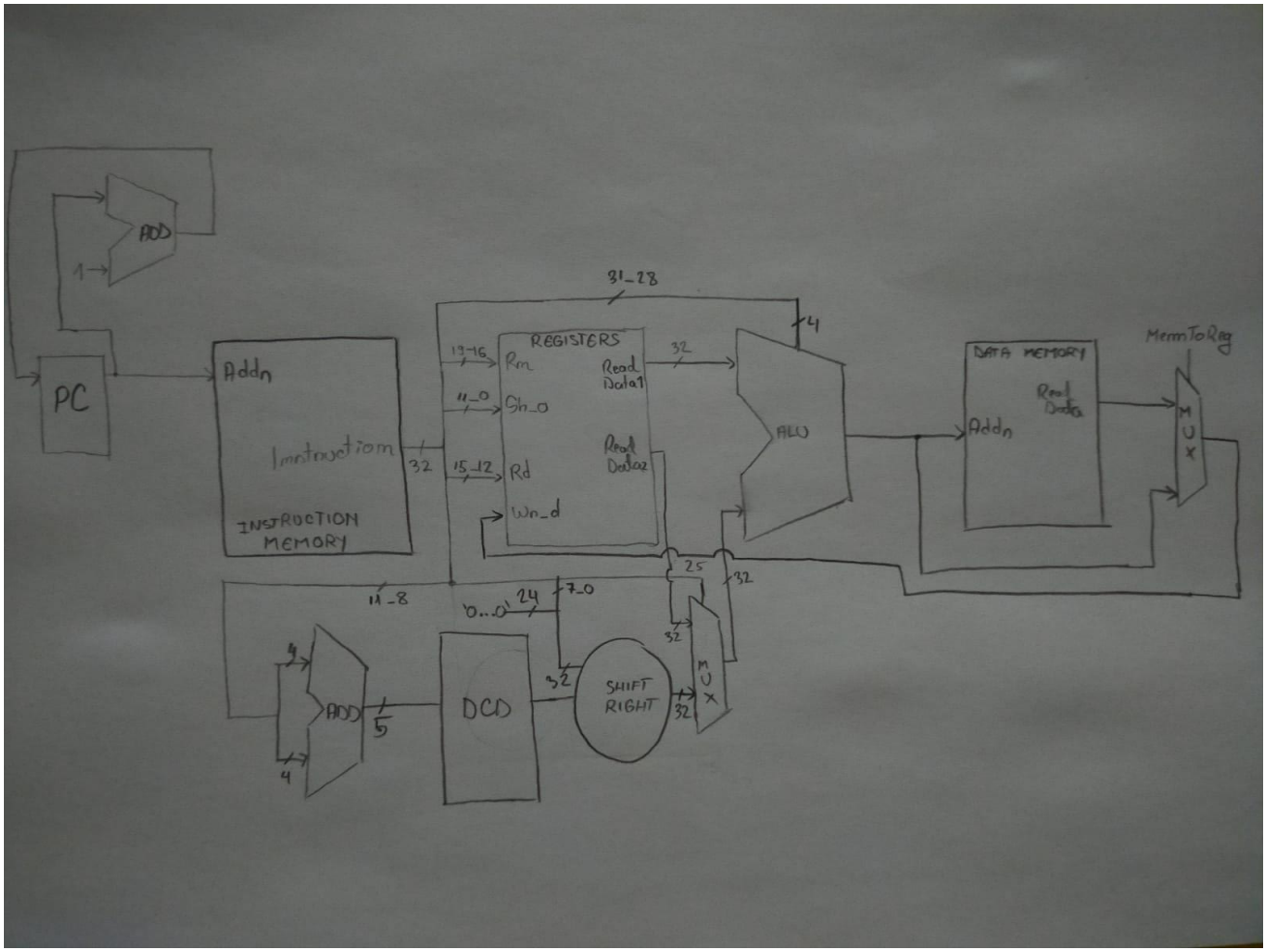


Image 3.3

The program counter will be incremented by 1 at each step. Addr represents the address of the instruction.

The 32-bit instruction will be taken as follows: bits 19-16 will be the first operand (Rn), bits 11-8 will be the second operand (in case it is a register), and bits 15-12 will be the destination register (Rd).

Bits 31-28 will represent the opcode given to the ALU in order to let it know which operation to execute. Bit 25 will represent whether the second operand is a register or an immediate value.

Bits 11-0 represent the shifter operand, which an either be a register or an immediate value. In case it is a register (specified by bit 25), we only need bits 11-8 to represent it. In case the shifter operand is am immediate value:

The shifter operand is represented on 12 bits. We can only represent 2^12 numbers on 12 bits, meaning the numbers from 0-4095. So instead, we will not use the 12-bit immediate value as a 12-bit number, but instead we will use it as an 8-bit number with a 4-bit rotation, as follows:

Shifter operand:

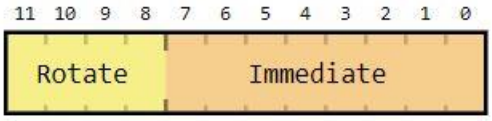
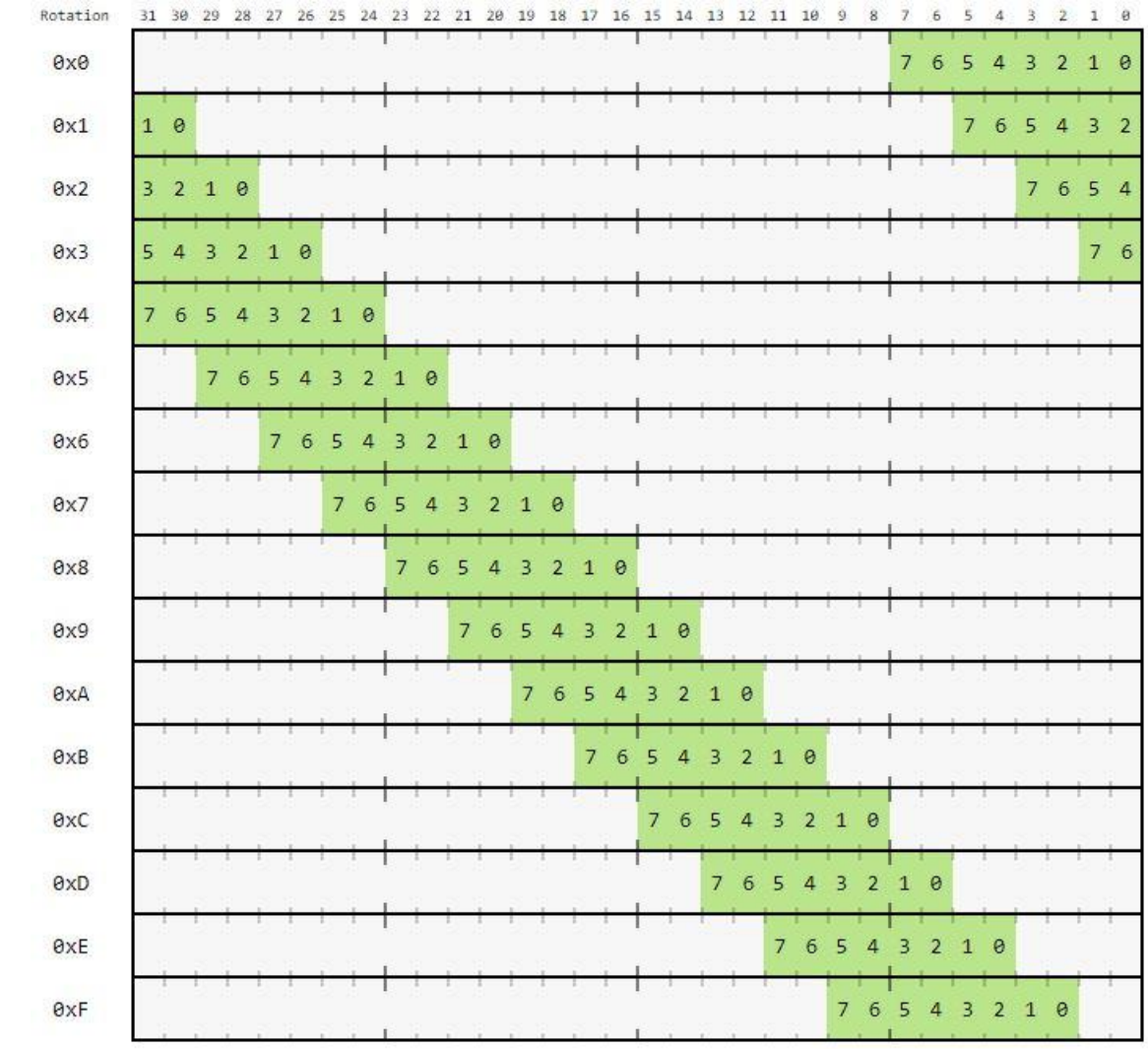


Image 3.4 [6]

The 4-bit rotation has 16 possible values. In order to obtain the most effective possible rotation, we multiply it by 2. So, instead of representing numbers from 0-15, we can represent all the even numbers from 0-30. In order to form our 32-bit value, we extend the 8-bit immediate value with 24 0's. Using the multiplied value of the 4-bit rotation, we can shift the 32-bit immediate value with 0, 2, -> 30 bits, obtaining a much more useful set of numbers than 0-4095.



++

Image 3.5 [6]

Finally, with respect to the bit 25, we choose between the 32-bit register value and the 32-bit rotated immediate value for our operation, using a multiplexer.

The result of the operation is stored on the Addr in the Data Memory. Then, using a multiplexer, we can choose between arithmetic operations and load/store, by setting the MUX inputs as the address on which we have to store a value, or the value held on that address, which we have to save into a register.

4. Design

The CPU, considered as the brain of the computer, performs all types of data processing operations. It can store data, intermediate results and instructions. The three main components in a CPU's structure are the Memory Unit (or Storage Unit), the Control Unit and the ALU (Arithmetic Logic Unit).

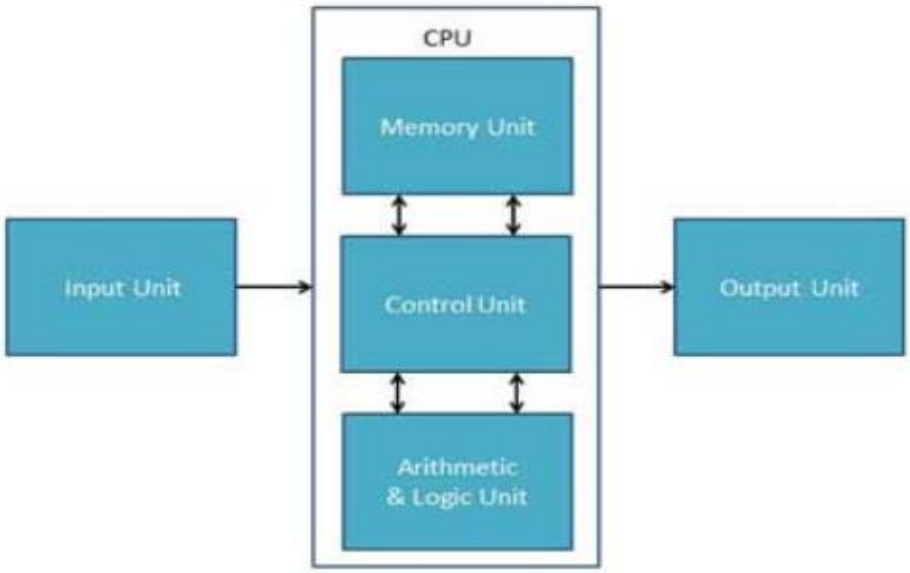


Image 4.1 [3]

4.1 The Memory Unit

The Memory Unit has the purpose of storing all types of information, such as instructions, data or intermediate results. Whenever the information stored in this component is needed in order to perform certain tasks, the CPU fetches the instructions from the memory and executes it. Technically, the Memory Unit is not part of the CPU, they just work together in order to run programs. The Memory type of this project is RAM (Random Access Memory).

The main functions of the memory unit are, as follows:

- ➔ It stores data and instructions required for processing
- ➔ It stores intermediate results, required for future processing
- ➔ It stores the final result of an operation, before it is transmitted to a possible output device
- ➔ It is the place where both inputs and outputs are sent, stored and transmited through

4.2 The Control Unit

The Control Unit (CU) has the purpose of directing operations within the CPU. It lets the other components know how to respond to different instructions.

It receives input information that it converts into signals, checks if the signals have been delivered successfully, and makes sure the data goes to the correct place at the correct time. Once the CPU has fetched the information (moved from the Memory Unit to a register), it is decoded, and the Control Unit lets the ALU know what it has to execute.

The main functions of the Control Unit are, as follows:

- ➔ It is responsible of the data flow throughout the system
- ➔ It obtains instructions from the memory, interprets them, and directs the operation to the ALU, in order to execute it
- ➔ It does not process nor store data
- ➔ Moves data between registers, ALU and memory

4.3 ALU

The ALU (Arithmetic Logic Unit) has the purpose of performing arithmetic and logic operations and is the fundamental building block of the CPU. The ALU receives information from the Control Unit related to what operation it has to perform, and it stores the result in an output register.

4.4 The Instruction Cycle (Machine Cycle)

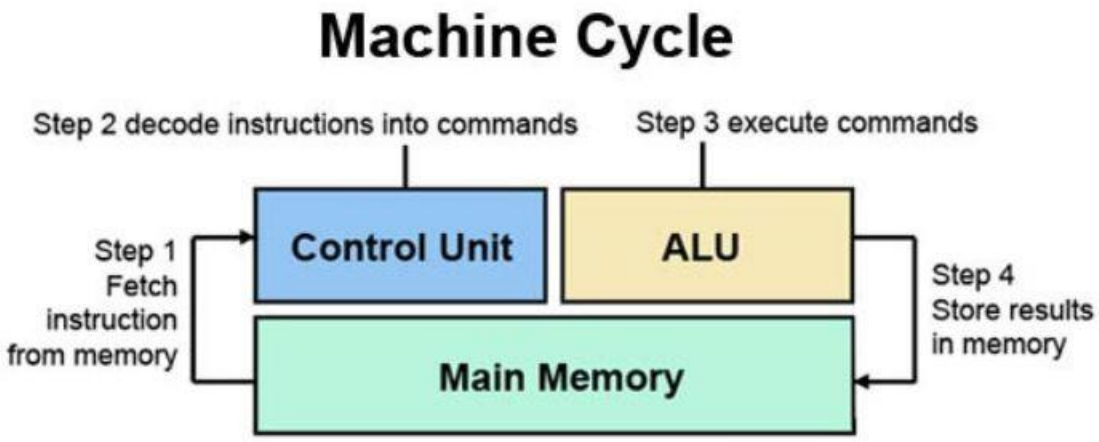


Image 4.2 [4]

The main purpose of the CPU is to execute instructions using the Instruction Cycle (Machine Cycle / fetch-decode-execute cycle).

The process takes place as follows:

- ➔ The processor fetches the instruction value from the memory location within the Memory Unit
- ➔ Once the instruction has been fetched, it is decoded
- ➔ Once decoded, the instruction is sent to the ALU in order to be executed
- ➔ The ALU executes the instruction and places the result in the memory
- ➔ The cycle is repeated until the program ends

5. Implementation

5.1 Instruction Fetch

The Instruction Fetch holds all the instructions that need to be executed, within an array of std_logic_vector(31 downto 0).

Each operation works with either two registers and the destination register, or one register, an immediate value and the destination register. For each operation that works with a register, the value of the said register will be set using a MOV operation before the specific operation. So, we have the following operations:

1. ADD : 0100_00_0_0100_1_0001_0011_000000000010
   This operation adds 8 and 6 (from two registers) and saves 14 (Rd)
   We have the following MOV operations:
   1.1: 1101_00_1_1101_0_0000_0001_000000001000 – moves 8 in r1
   1.2: 1101_00_1_1101_0_0000_0010_000000000110 – moves 6 in r2

2. SUB: 0010_00_1_0010_1_0001_0011_111000000001
   This operation subtracts the immediate value 16 from 252 (Rn) and saves 236 (Rd)

We have the following MOV:
2.1: 1101_00_1_1101_0_0000_0001_000011111100 – moves 252 in Rn

3. ORR: 1100_00_0_1100_1_0001_0011_000000000010
   This operation performs a bitwise or on 128 and 64 and saves the result 192
   3.1: 1101_00_1_1101_0_0000_0001_000010000000 – moves 128 in r1
   3.2: 1101_00_1_1101_0_0000_0010_000001000000 – moves 64 in r2

4. ADC: 0101_00_0_0101_1_0001_0011_000000000010
   This operation performs an add with carry operation on 64 and 32 and saves the result 96. As the addition does not overflow, the carry flag is not set
   4.1: 1101_00_1_1101_0_0000_0001_000001000000 – moves 64 in r1
   4.2: 1101_00_1_1101_0_0000_0010_000000100000 – moves 32 in r2

5. SBC: 0110_00_0_0110_1_0001_0011_000000000010
   This operation performs a subtraction with carry operation on 193 and 128 and saves the result 65.
   5.1: 1101_00_1_1101_0_0000_0001_000011000001 – moves 193 in r1
   5.2: 1101_00_1_1101_0_0000_0010_000010000000 – moves 128 in r2

6. RSB: 0011_00_0_0011_1_0001_0011_000000000010
   This operation performs a reverse subtraction (sh_o – Rn instead of Rn – sh_o). In our case, we subtract 16 from 32 and save the result 16.
   6.1: 1101_00_1_1101_0_0000_0001_000000010000 – moves 16 in Rn
   6.2: 1101_00_1_1101_0_0000_0010_000000100000 – moves 32 in sh_o

7. CMP: 1010_00_0_1010_1_0001_0011_000000000010
   This operation compares the values 128 and 128, and sets the EQ flag. EQ flag represents flags(2), so the result is ‘0100’ (4).
   7.1: 1101_00_1_1101_0_0000_0001_000010000000 – moves 128 in r1
   7.2: 1101_00_1_1101_0_0000_0010_000010000000 – moves 128 in r2

8. MOV: This operation has been illustrated above.

On each clock event, the counter is incremented by one. If the reset signal is active, then the counter is reset. In order to obtain the correct instruction, we convert the counter value into an integer, and save the value within the array (reg_file) at that position.

5.2 Register File

We have a register file consisting of 16 registers, each of them being able to store 32 bits. Each register can be accessed, but for the sake of the example, we only use registers 1, 2 (operands) and 3 (result).

On each clock event, the register file converts Rd (the destination register) into an integer, and saves wr_d (the data) on that position. For example, if wr_d is “00000000000000000000000000000001” and Rd is “0001”, we will have the value 1 saved in the first register.

Furthermore, the register file saves the values within the given Rn and Sh_o (registers holding the operands) in rd1 and rd2, which are the two register file outputs. These outputs shall be used by the ALU.

5.3 Rotate Unit

This unit is responsible with the rotation of the immediate value, as mentioned above. Instead of using an adder and a decoder as mentioned in the data path, we simply save the rotation value, the immediate value on bits 7->0 and 0's on bits 31->8. As mentioned above, the rotation is represented on 4 bits, so in order to obtain 30bits possible rotations, we have to double it. But, instead of doubling the rotation, we simply rotate the rot value twice, in order to obtain the final rotated value.

5.4 ALU

This unit is as simple as it gets. Our first operand is a register (Rn), and our second operand (alu_snd_in) is either a second register or an immediate value, depending on the selection value (Bit 25 of the instruction, which differentiates between immediate value and register).

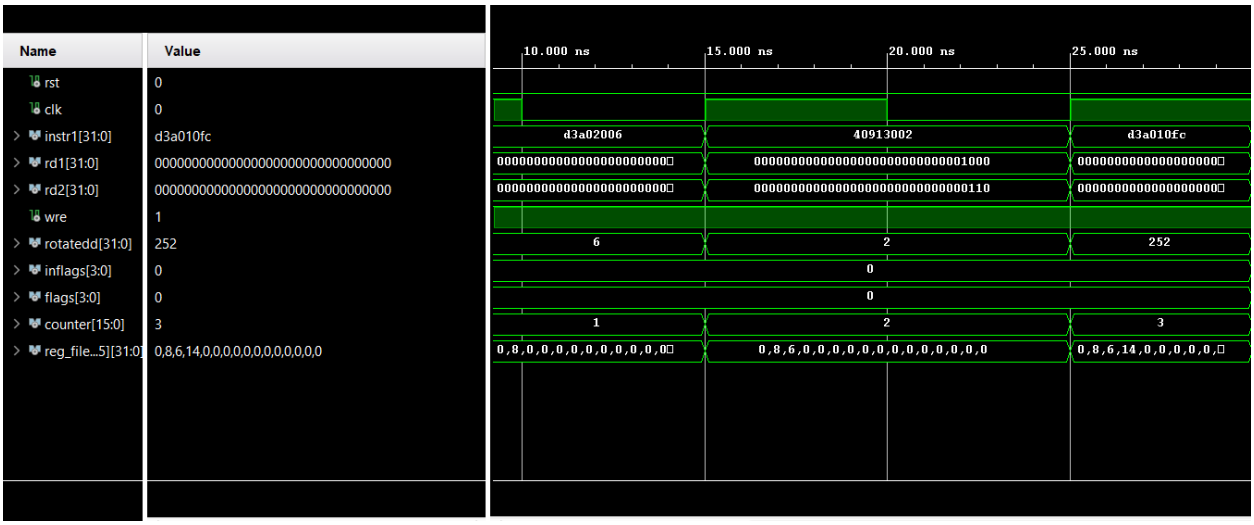
Then, depending on the opcode (4 bits, different for each instruction), the unit performs the correct operation and sets the corresponding values.

6. Testing

This section contains a screenshot after each operation, showing the operand values (register 1 and 2 or 1 and immediate <rotated value>), and the result (register 3).

Using the values specified in section 5.1, we have the following:

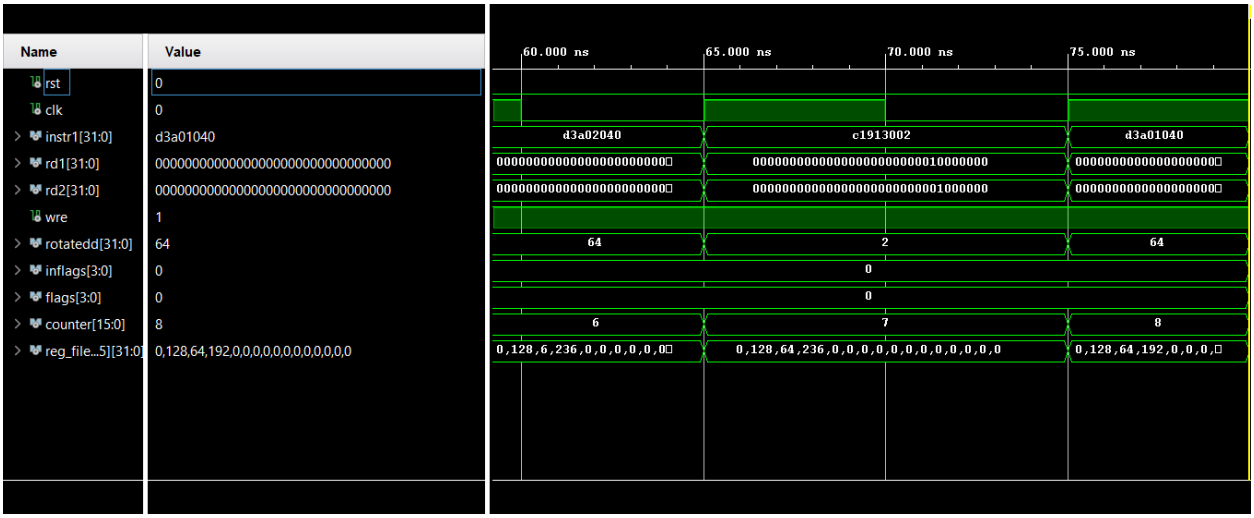
ADD



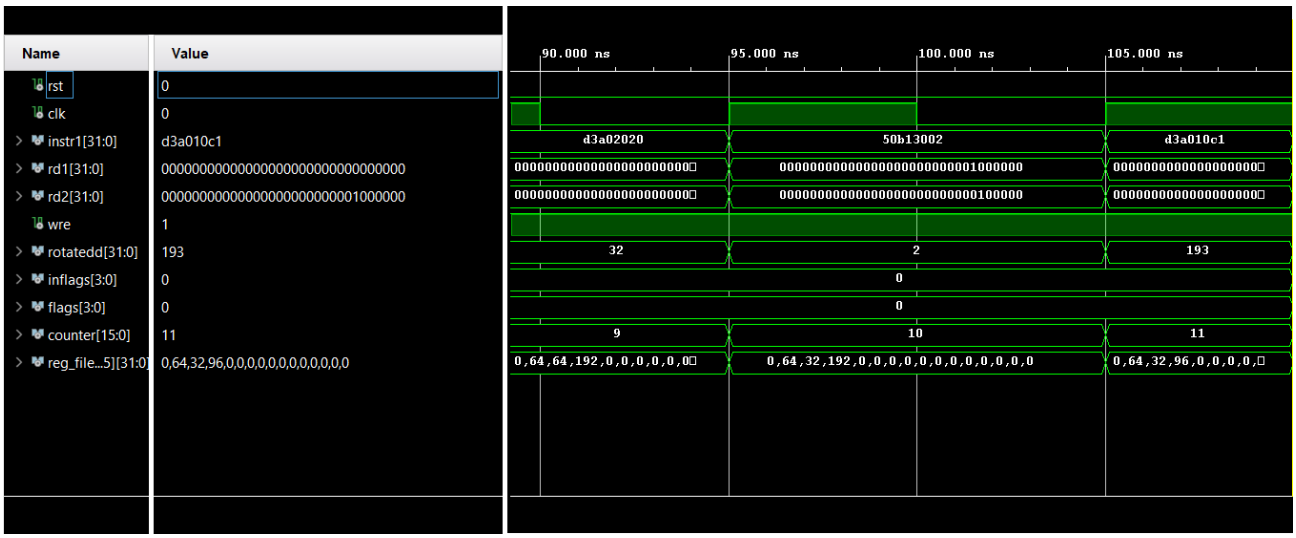
SUB

We have 252 in register 1, and 16 as the immediate value on rotated. The answer will be saved in R3:

ORR


Name
Value
rst
0
clk
0
instr1[31:0]
d3a01040
rd1[31:0]
00000000000000000000000000000000
rd2[31:0]
00000000000000000000000000000000
wre
1
rotatedd[31:0]
64
inflags[3:0]
0
flags[3:0]
0
counter[15:0]
8
reg_file...5][31:0]
0,128,64,192,0,0,0,0,0,0,0,0,0,0,0,0
60.000 ns
65.000 ns
70.000 ns
75.000 ns
d3a02040
c1913002
d3a01040
64
2
64
6
7
8
0,128,6,236,0,0,0,0,0,0
0,128,64,236,0,0,0,0,0,0,0,0,0,0,0,0
0,128,64,192,0,0,0,

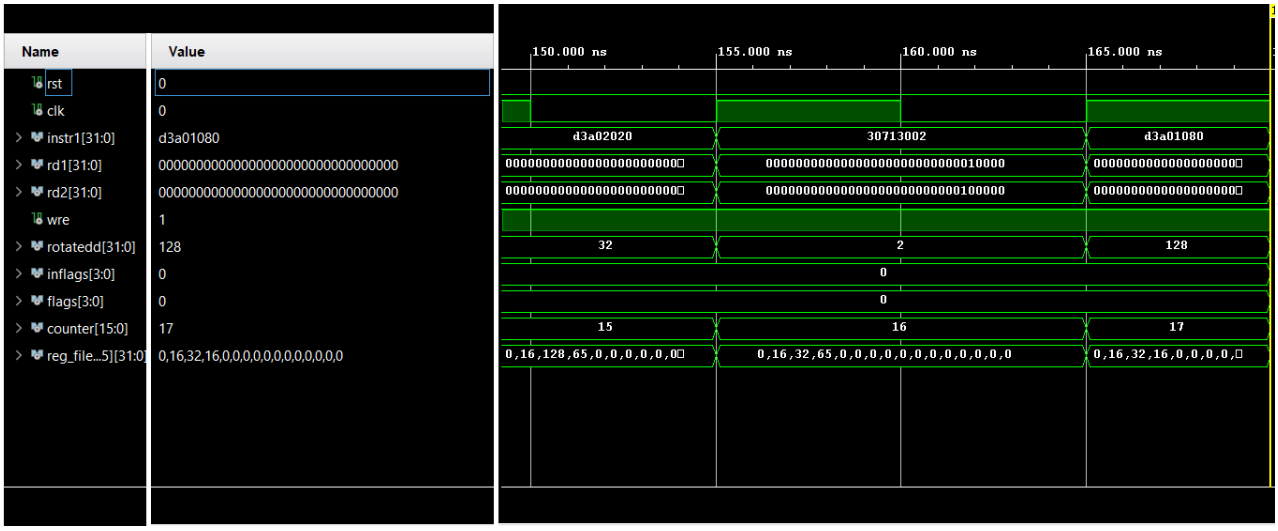
ADC


Name
Value
rst
0
clk
0
instr1[31:0]
d3a010c1
rd1[31:0]
00000000000000000000000000000000
rd2[31:0]
00000000000000000000000001000000
wre
1
rotatedd[31:0]
193
inflags[3:0]
0
flags[3:0]
0
counter[15:0]
11
reg_file...5][31:0]
0,64,32,96,0,0,0,0,0,0,0,0,0,0,0,0
90.000 ns
95.000 ns
100.000 ns
105.000 ns
d3a02020
50b13002
d3a010c1
32
2
193
9
10
11
0,64,64,192,0,0,0,0,0
0,64,32,192,0,0,0,0,0,0,0,0,0,0,0,0
0,64,32,96,0,0,0,0,

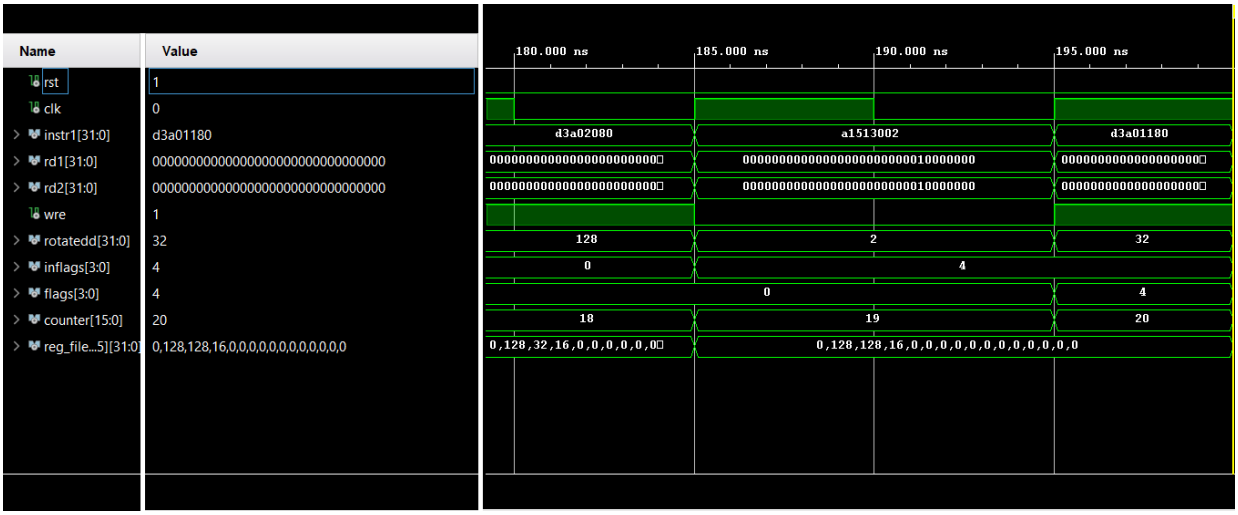
SBC


Name
Value
rst
0
clk
0
instr1[31:0]
d3a01010
rd1[31:0]
00000000000000000000000000000000
rd2[31:0]
00000000000000000000000000000000
wre
1
rotatedd[31:0]
16
inflags[3:0]
0
flags[3:0]
0
counter[15:0]
14
reg_file...5][31:0]
0,193,128,65,0,0,0,0,0,0,0,0,0,0,0,0
120.000 ns
125.000 ns
130.000 ns
135.000 ns
d3a02080
60d13002
d3a01010
128
2
16
12
13
14
0,193,32,96,0,0,0,0,0,0
0,193,128,96,0,0,0,0,0,0,0,0,0,0,0,0
0,193,128,65,0,0,0,

RSB


Name
Value
rst
0
clk
0
instr1[31:0]
d3a01080
rd1[31:0]
00000000000000000000000000000000
rd2[31:0]
00000000000000000000000000000000
wre
1
rotatedd[31:0]
128
inflags[3:0]
0
flags[3:0]
0
counter[15:0]
17
reg_file...5][31:0]
0,16,32,16,0,0,0,0,0,0,0,0,0,0,0,0
150.000 ns
155.000 ns
160.000 ns
165.000 ns
d3a02020
30713002
d3a01080
32
2
128
15
16
17
0,16,128,65,0,0,0,0,0
0,16,32,65,0,0,0,0,0,0,0,0,0,0,0,0
0,16,32,16,0,0,0,0,

CMP



7. Bibliography

[1] *"Central Processing Unit",* at https://en.wikipedia.org/wiki/Central_processing_unit

[2] *"Arm Architecture Reference Manual",* at http://cs107e.github.io/readings/armv6.pdf

[3] *"Computer – CPU",* at https://www.tutorialspoint.com/computer_fundamentals/computer_cpu.htm?fbclid=IwAR3RcaeWdady6KDAQei3LZjQLIxjNIfRY0QgGaemCVyh7dueKsybDB1PRCA

[4] *"Machine Cycle",* at https://www.computerhope.com/jargon/m/machcycl.htm

[5] Yung-Yu Chuang, *"ARM Instruction Set"*, at https://www.csie.ntu.edu.tw/~cyy/courses/assembly/10fall/lectures/handouts/lec09_ARMisa.pdf

[6] Alisdair McDiarmid, "ARM immediate value encoding", at https://alisdair.mcdiarmid.org/arm-immediate-value-encoding/