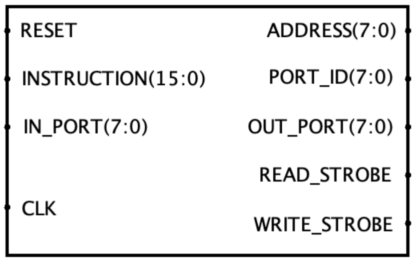
An implementation of the PicoBlaze 8-bit Microcontroller

David-Oliver Șipoș

1 Project description

This document presents an implementation of the PicoBlaze 8-bit Microcontroller for FPGAs based on the architecture and instruction set presented in XAPP213. The microcontroller operates on a 100Mhz clock with each instruction lasting 2 cycles. It has the following black box:



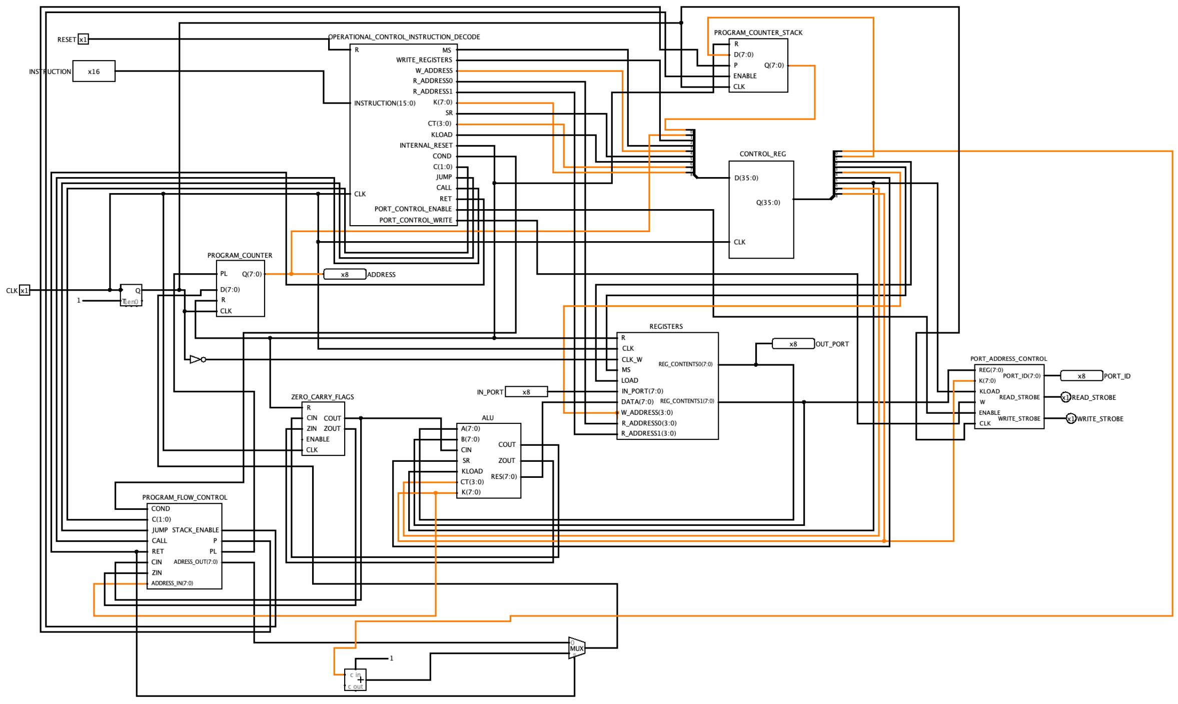
Inputs:

- **RESET** - the microcontroller can be reverted to its initial state using this signal
- **CLK** - clock signal up to 100Mhz
- **INSTRUCTION** - the code of the next instruction to be executed
- **IN_PORT** - external data can be provided to the microcontroller using the `INPUT` instruction

Outputs:

- **ADDRESS** - the address of the next instruction
- **PORT_ID** - it enables the use of multiple I/O ports
- **OUT_PORT** - output from the microcontroller
- **READ_STROBE** - signal indicating the successful capture of data
- **WRITE_STROBE** - signal indicating the successful writing of data

2 Architecture

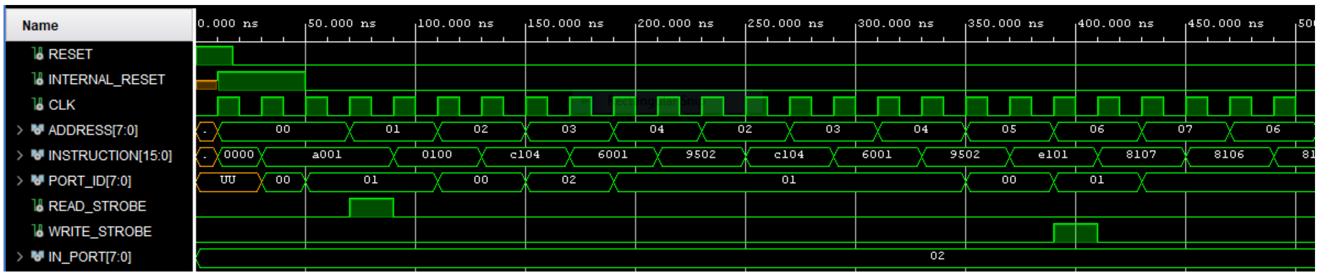


There are 9 principal components each having a well-defined role. The main component is called OPERATIONAL_CONTROL_INSTRUCTION_DECODE and is responsible for computing the control signals for the other components using the bits of the instruction to be executed. The microcontroller has 16 8-bit general purpose registers called `s0...s15` and a stack which allows recursive calls with the maximum depth of 15. Another major component is the ALU built for performing logical operations, additions, substractions, rotate and shift operations. Several components are responsible for the program flow allowing unconditional and conditional jumps and procedure calls. The read and write signals are generated by the PORT_ADDRESS_CONTROL circuit.

A 36-bit register exploits the benefits of a pipeline. An instruction is executed as follows:

1. on the first rising edge of the clock the instruction is fetched and decoded
2. the next rising edge triggers instruction execution followed by the update of the program counter with the correct address for the next instruction
3. on the last rising edge the result is saved in the registers and the flags are updated while the next instruction is being fetched and decoded

The following waveform presents the execution of a simple program which calculates the sum of natural numbers from 1 to a number received at the IN_PORT input:

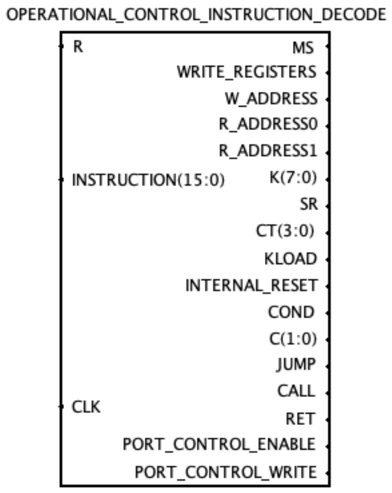


3 Principal components

3.1 OPERATIONAL CONTROL INSTRUCTION DECODE

Each instruction is encoded on 16 bits. There are two main instruction formats: the first one specifies as operands registers and the second one uses a register and an 8-bit constant value. For example, `LOAD s0,s1(C010)` copies the contents of register `s1` into register `s0` and `ADD s0,9A (409A)` adds to register `s0` the value `9A`. Based on bits 15-12 of an instruction one can determine which components are involved in the execution. Depending on the type of the instruction the following four bits can encode a register or a condition to be verified. For the content of the last 8 bits there are several possibilities: a constant value, a register and operation code, or special bits indicating an I/O or `RETURN` instruction. Based on these facts this part of the microcontroller decodes the instructions.

The component has the following black box:



When R becomes high a signal called INTERNAL_RESET is generated synchronously which lasts 2 clock cycles and is used as a reset indicator for other components allowing the microcontroller to reach its initial state. This signal is created using two D flip-flops with synchronous set.

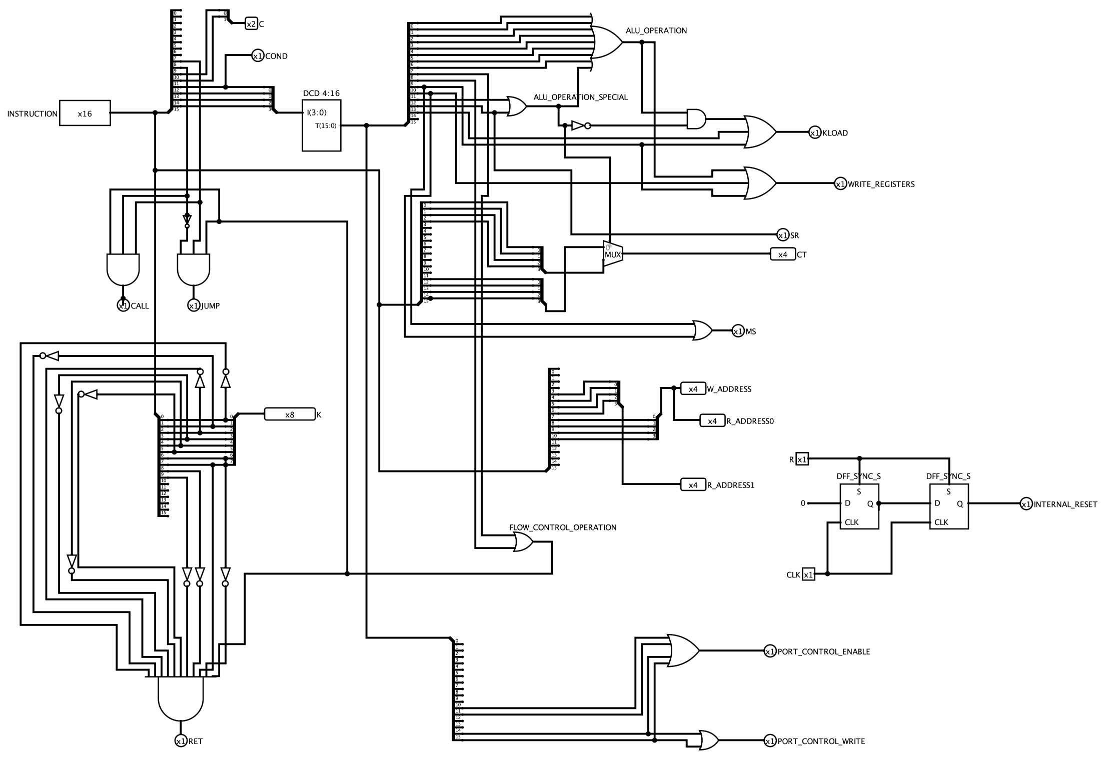
The processing of an instruction begins with a 4:16 DECODER. The outputs of the decoder are then used to determine the instruction type. Instructions starting with the values 0-7, C, D are executed by the ALU, A, B, E, F represent I/O instructions and finally 8, 9 indicate `CALL` and `RETURN` instructions. The output MS(master select) tells whether the value loaded into the registers comes from the ALU or from an external source and WRITE_REGISTERS triggers the update of all the registers and flags of the microcontroller. The pair R_ADDRESS0, R_ADDRESS1 represent the operand registers and the signal W_ADDRESS shows where the result should be written.

K contains the last 8 bits of the instruction. SR and CT identify the operation for the ALU, KLOAD telling it whether a constant or a register value is used.

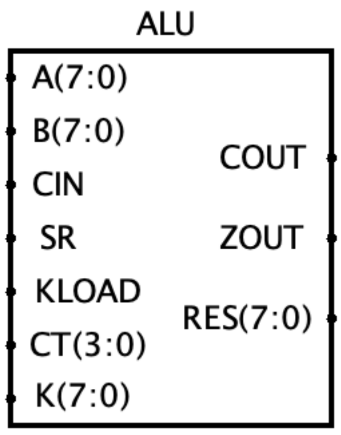
The ports COND, C, JUMP, CALL, RET are sent to the circuit called PROGRAM_FLOW_CONTROL, which in turn verifies the condition and sends additional control signals to the stack and program counter.

PORT_CONTROL_ENABLE triggers the generation of a strobe, PORT_CONTROL_WRITE specifying its type.

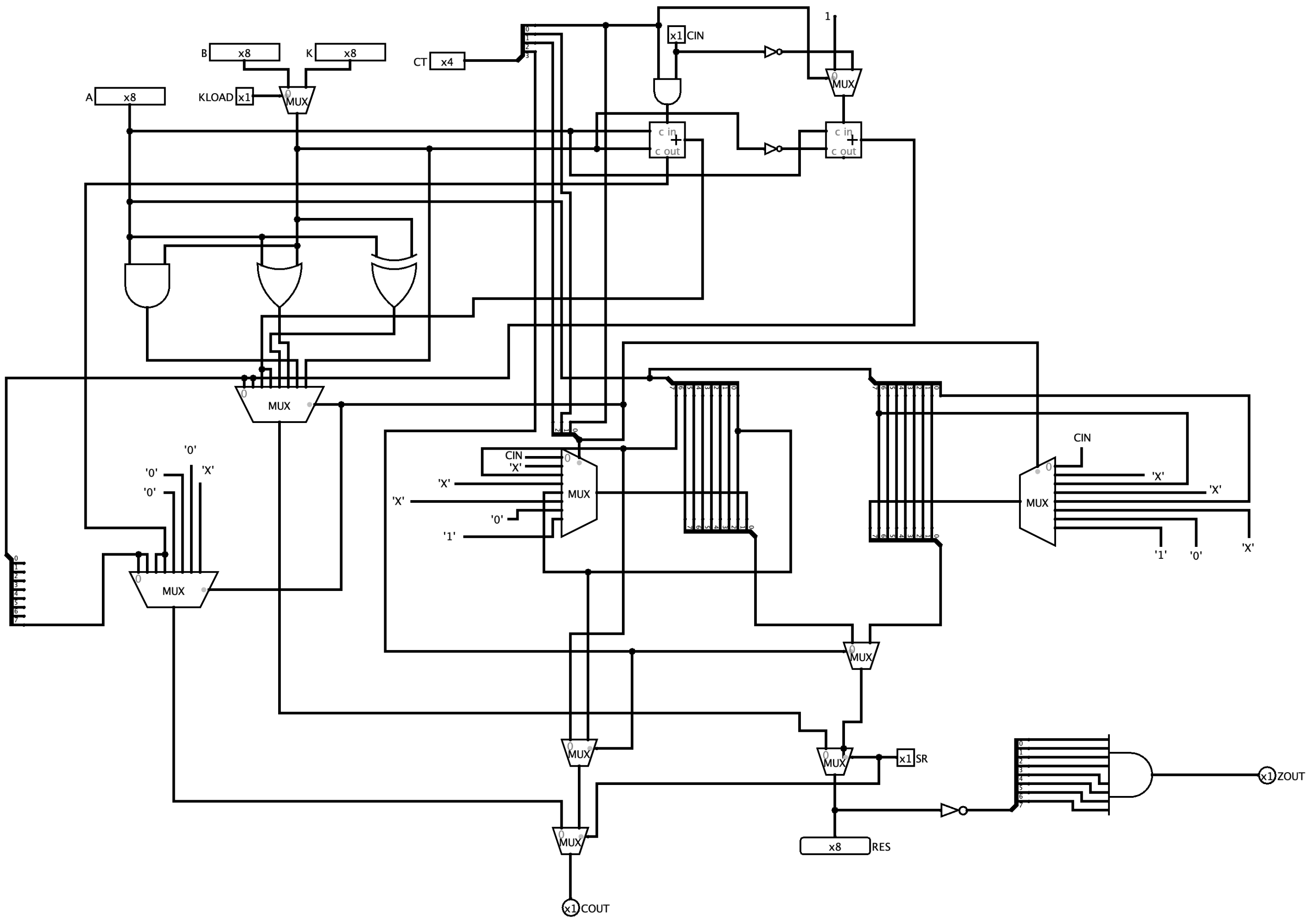
The next block diagram illustrates how this process is implemented:



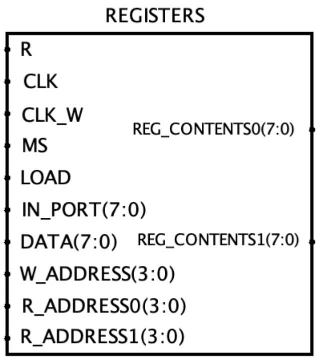
3.2 ARITHMETIC LOGIC UNIT



The ALU has a simple design being a combinational logic circuit. When the input is received all possible results are calculated and with the help of multiplexers the corresponding output is selected based on SR and CT, KLOAD telling whether the second operand is a register or the value K. The carry flag contains the most significant or least significant bit in case of rotates and shifts, the carry bit for additions and in case of substractions it is equal to the sign of the result. The zero flag is set whenever the result is 0. The result and flags are written on the next rising edge. Substraction is performed using an 8-bit full adder in 2's complement. This logic is integrated in the next block diagram:

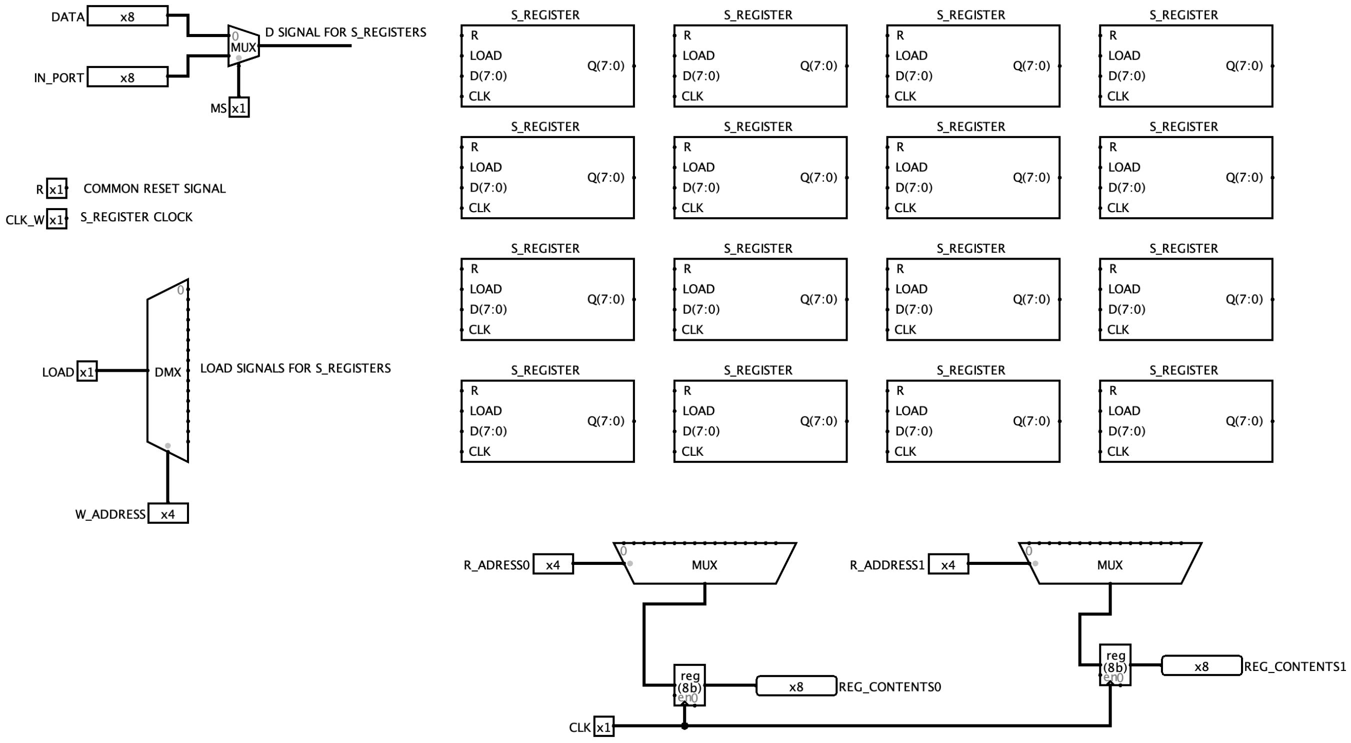


3.3 REGISTERS



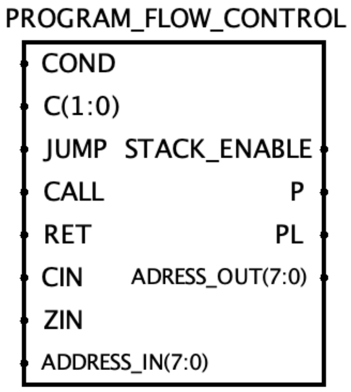
This component allows acces to all of the 16 registers of the microcontroller. Each read address is decoded using a multiplexer. The input MS is the selection of a multiplexer and specifies the source of the data to be written, while the LOAD signal in combination with the W_ADDRESS port triggers the update of the registers on the next rising edge. The outputs will transmit the contents of the registers specified at the input with the signals R_ADDRESS0 and R_ADDRESS1.

A high-level view of the component has the following the structure:



3.4 PROGRAM FLOW CONTROL

The interface of this crucial has the form:



This part of the microcontroller verifies various conditions and decides the address of the next instruction. The condition is encoded using COND and C, the first one indicating the existence of a condition which needs to be satisfied while the second one encodes on two bits one of the following conditions:

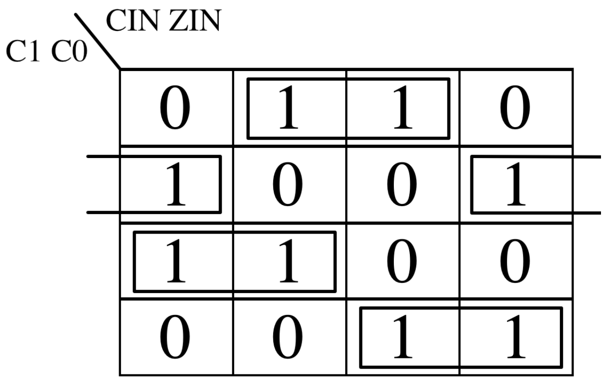
- if ZERO
- if NOT ZERO
- if CARRY
- if NOT CARRY

The conditions are evaluated using the flags. In order to achieve a simple CLC a Karnaugh map was created and minimized.

The inputs JUMP, CALL, RET indicate the course of action to be taken if the condition is satisfied.

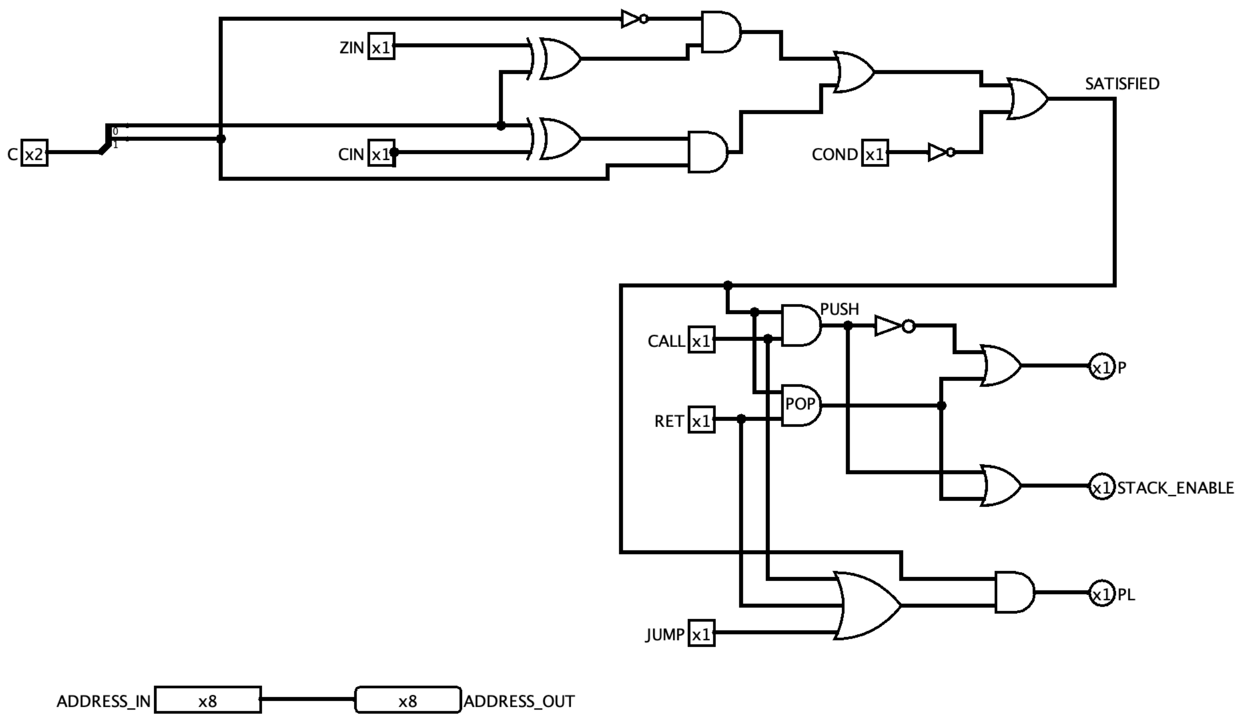
STACK_ENABLE and P control the PROGRAM_COUNTER_STACK triggering together a PUSH or POP operation. The signal PL triggers a parallel load at the level of the PROGRAM_COUNTER such that the value from ADDRESS_OUT will be loaded into the counter breaking program's normal flow.

The Karnaugh map and the expressions it yielded are the following:



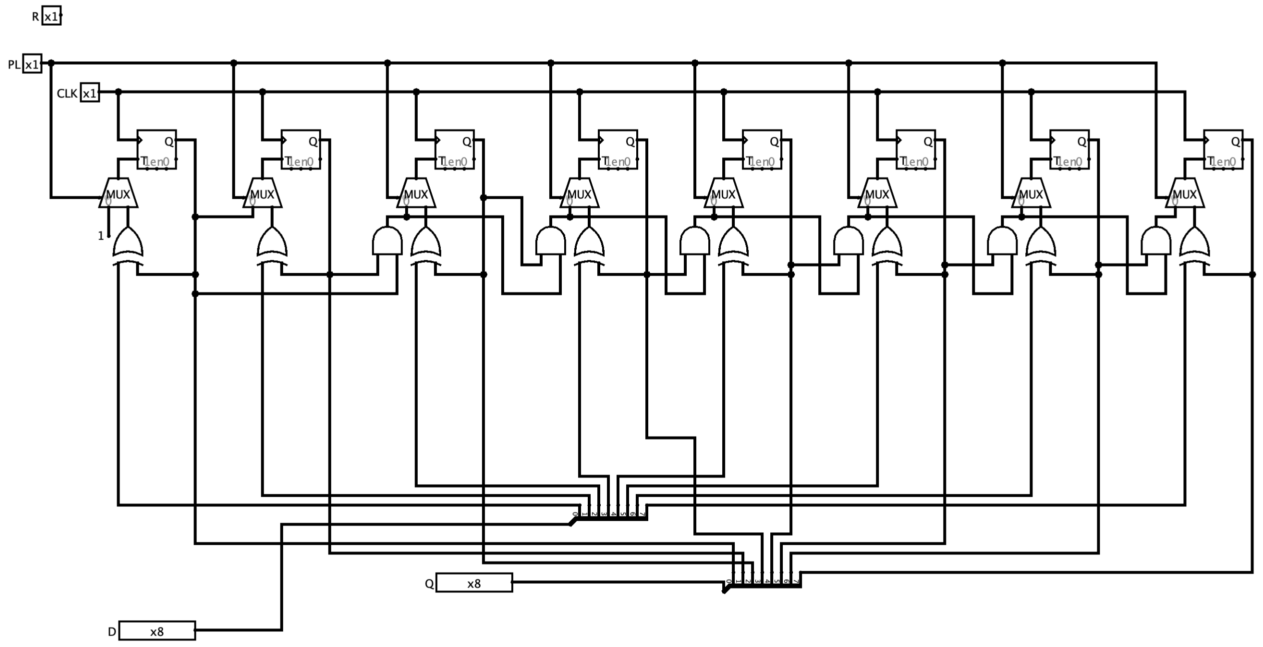
$$\text{SATISFIED} = \overline{\text{COND}} + \text{f}$$
$$\text{f} = \overline{\text{C}_1} \cdot (\text{C}_0 \oplus \text{ZIN}) + \text{C}_1 \cdot (\text{C}_0 \oplus \text{CIN})$$
$$\text{PUSH} = \text{CALL} \cdot \text{SATISIFED}$$
$$\text{POP} = \text{RETURN} \cdot \text{SATISFIED}$$
$$\text{PL} = (\text{CALL} + \text{JUMP} + \text{RETURN}) \cdot \text{SATISFIED}$$
$$\text{P} = \overline{\text{PUSH}} + \text{POP}$$

Implementing the equations using logic gates gives the following circuit:



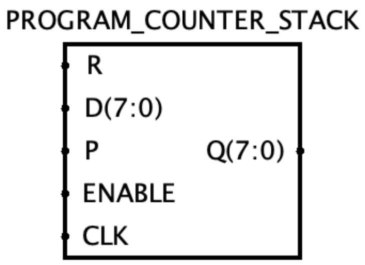
3.5 PROGRAM COUNTER

The program counter is a synchronous 8-bit binary counter with reset and parallel load. It has been implemented using T flip-flops with the help of multiplexers having as selections the input PL. When a parallel load is triggered the XOR gates compute the correct input for the flip-flops based on the current output and the data to be loaded.



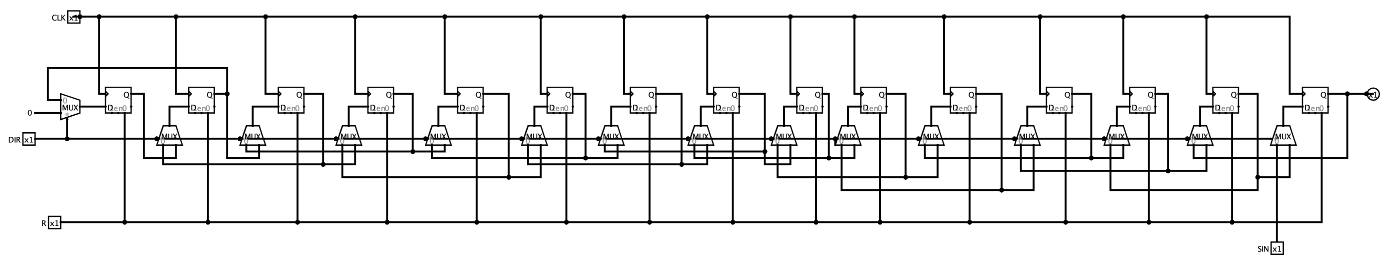
3.6 PROGRAM COUNTER STACK

The stack has the following interface:

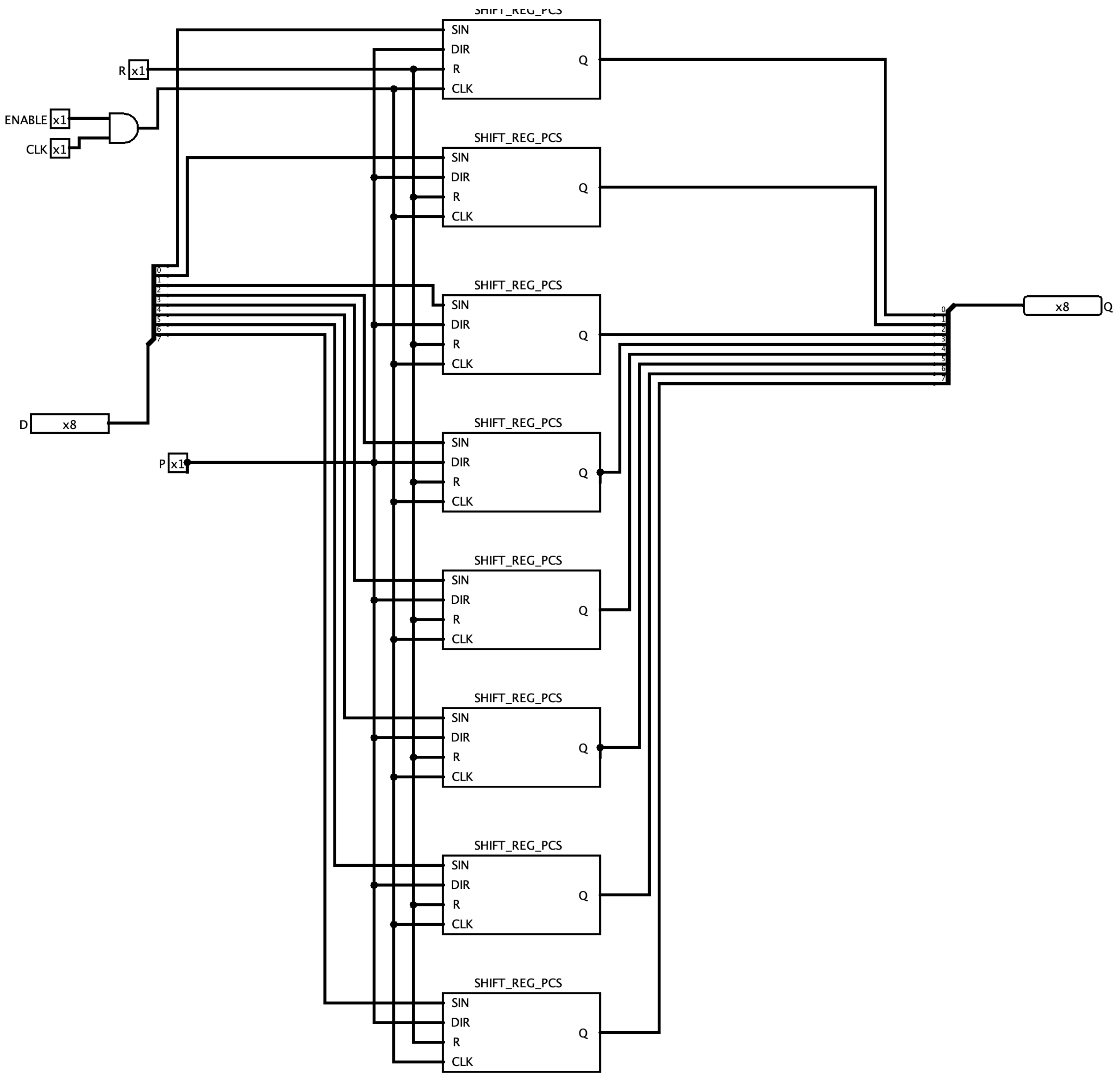


The stack is based on a component called SHIFT_REG_PCS which is a 15-bit shift register. Using eight of these register one can create a stack of size 15 with a `PUSH` operation being a right shift and a `POP` operation corresponding to a left shift. When the stack is enabled and PL is low a `PUSH` operation will be performed using the input D, on the other hand when PL is high a `POP` will be executed. The output Q will indicate the top value in the stack.

The shift register is implemented with D flip-flops as presented in the drawing:

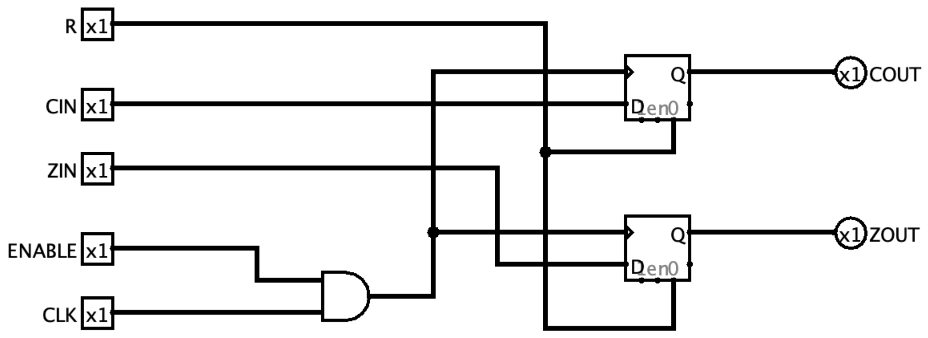


Using these registers one can obtain the following design:

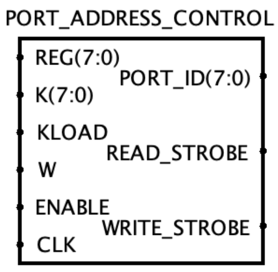


3.7 FLAGS

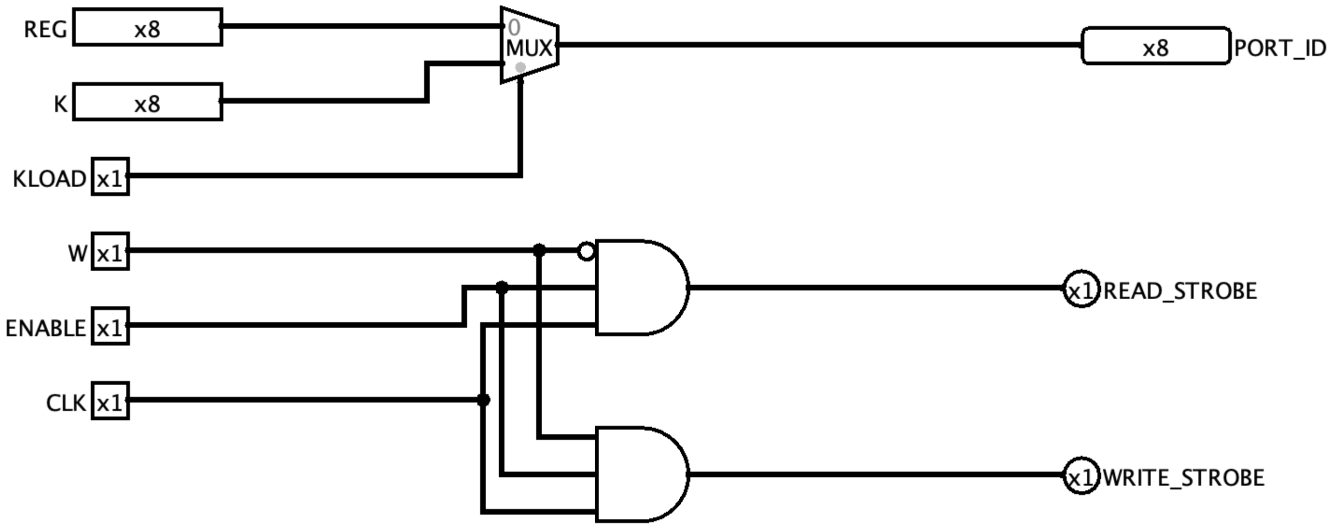
The carry and zero flags are implemented using two D flip-flops, when ENABLE is high on the next rising edge the flags are updated:



3.8 PORT ADDRESS CONTROL



The output PORT_ID is generated based on the signal KLOAD. The output strobes are obtained with the help of AND gates.



4 Program examples

4.1 Product of two numbers

```
00: INPUT sE, 01
01: INPUT sF, 00
02: CALL AA
03: OUTPUT sD, 01
04: OUTPUT sC, 00
05: JUMP 06
06: JUMP 05
AA: ADD sE, 00
AB: RETURN Z
AC: ADD sF, 00
AD: RETURN Z
AE: XOR sD, sD
AF: XOR sC, sC
B0: ADD sC, sF
B1: ADDCY sD, 00
B2: SUB sE, 01
B3: JUMP NZ, B0
B4: RETURN
```

4.2 Sum of natural numbers

```
00: INPUT s3, 01
01: XOR s0, s0
02: SUB s3, 00
03: CALL NZ 35
04: OUTPUT s0, 03
05: JUMP 06
06: JUMP 05
35: ADD s0, s3
36: SUB s3, 01
37: CALL NZ 35
38: RETURN
```

4.3 Euclid’s algorithm

```
00: INPUT s0, 01
01: LOAD s1, 00
02: INPUT s2, 00
03: LOAD s3, 00
04: LOAD sB, s1
05: LOAD sA, s0
06: LOAD sD, s3
07: LOAD sC, s2
08: XOR sD, FF
09: XOR sC, FF
0A: ADD sC, 01
0B: ADDCY sD, 00
0C: ADD sA, sC
0D: ADDCY sB, sD
0E: LOAD sF, sB
0F: AND sF, 80
10: JUMP NZ, 14
11: LOAD s1, sB
12: LOAD s0, sA
13: JUMP 1A
14: XOR sB, FF
15: XOR sA, FF
16: ADD sA, 01
17: ADDCY sB, 00
18: LOAD s3, sB
19: LOAD s2, sA
1A: LOAD sB, s1
1B: LOAD sA, s0
1C: LOAD sC, s2
1D: LOAD sD, s3
1E: XOR sD, FF
1F: XOR sC, FF
20: ADD sC, 01
21: ADDCY sD, 00
22: ADD sA, sC
23: ADDCY sB, sD
24: ADD sA, 00
25: JUMP NZ, 04
26: OUTPUT s0, 01
27: JUMP 28
28: JUMP 27
```

5 Future improvements

The ALU can be optimized reducing the number of components. It is possible to remove a full adder and a few multiplexers by adding additional logic. The microcontroller's registers are using a different clock from the main one and this is a bad design practice. The problem could be solved by redesigning the whole component. The architecture takes up many LUTs on the FPGA and there are ways to refine it thus reducing the number of resources needed for implementation. In the original XAPP213 documention there are instructions for handling interrupts. The addition of interrupts would be a major upgrade extending the range of practical applications.