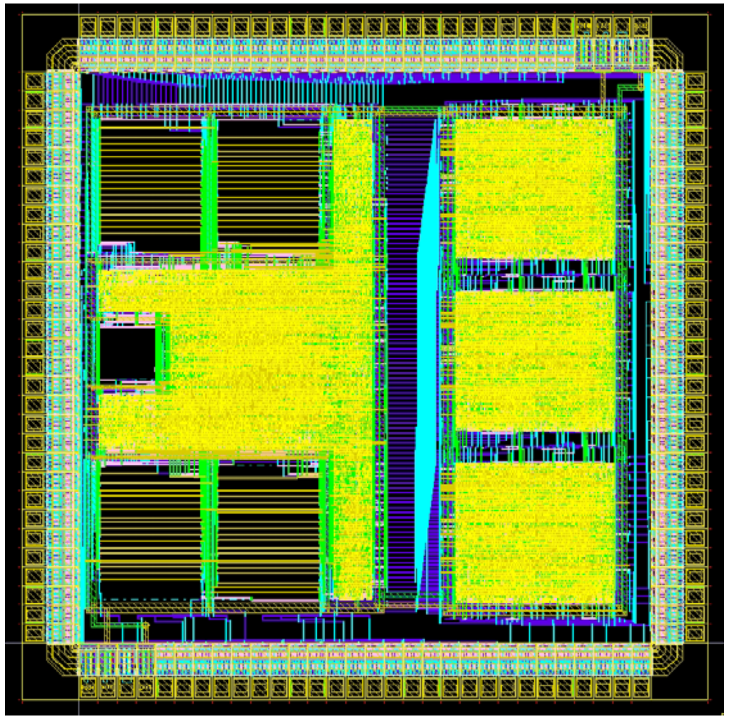
Digital Systems and Microprocessor Design (H7068)

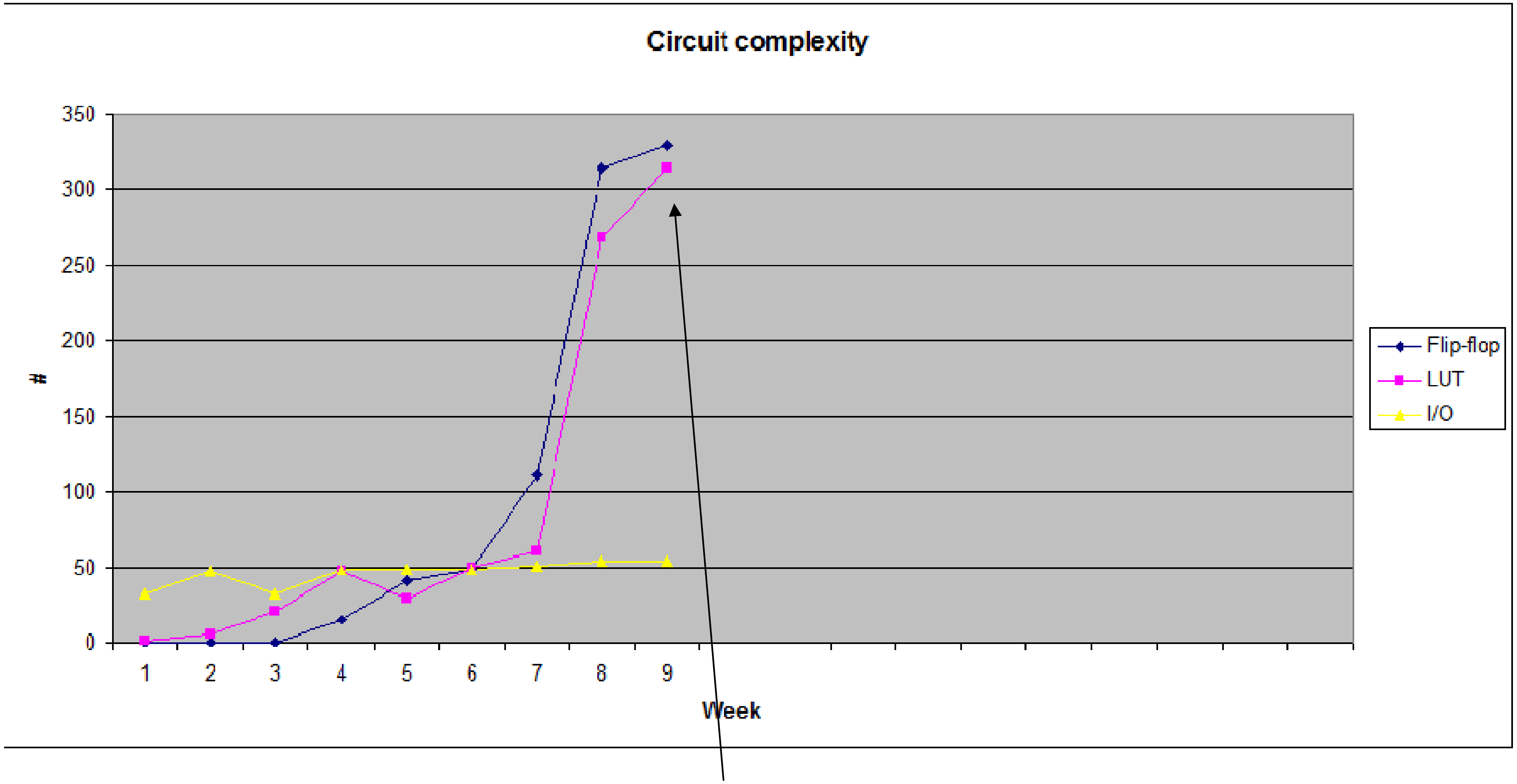


9. Exercises / Laboratory

2014

Daniel Roggen
Richmond 3A13
d.roggen@sussex.ac.uk

Circuit complexity



CPU with custom instructions

Objective / success criteria

- Prerequisite: lab of week 8, knowledge of digital electronics

- The objective of this laboratory is to:
  - Understand how the ALU of the educational processor is implemented
  - Understand how to add instructions to the educational processor (specifically the ALU)

- Success criteria:
  - You have implemented the two custom instructions in the tutorial of this lab
  - You have implemented the random number generator for the coursework assignment

Custom instructions

- The easiest place to add custom instructions is in the ALU
  - The CPU has been designed to have an extensible ALU
  - The instruction decoding to feed the ALU with data and store the result is already in place!
- Modifying or creating instructions unrelated to the ALU may involve significant changes to the control unit and other parts of the processor
  - For instance creating an instruction that takes 3 parameters instead of 2 would require to change the register bank
  - Creating an instruction that moves from memory to memory would require more than 3 clock cycles per instruction...

Custom ALU instruction

- ALU instructions are have opcodes: 001, 010, 011
- The ALU data path is activated for these opcodes
- The general format of the ALU instruction is:

instruction(15..8)								Instruction(7..0)							
opcode			$\overline{R}/I$	ALU op		dreg		src							
0	X	X	X	X	X	r	r	i/-	i/-	i/-	i/-	i/-	i/-	i/r	i/r

- The specific ALU operation is defined by the opcode and the ALUop bits in the instruction: instruction(12..10)
- Instruction(12) is called R'/I, but it can have a different meaning.
  - Single operand ALU instructions use R'/I as part of ALUop

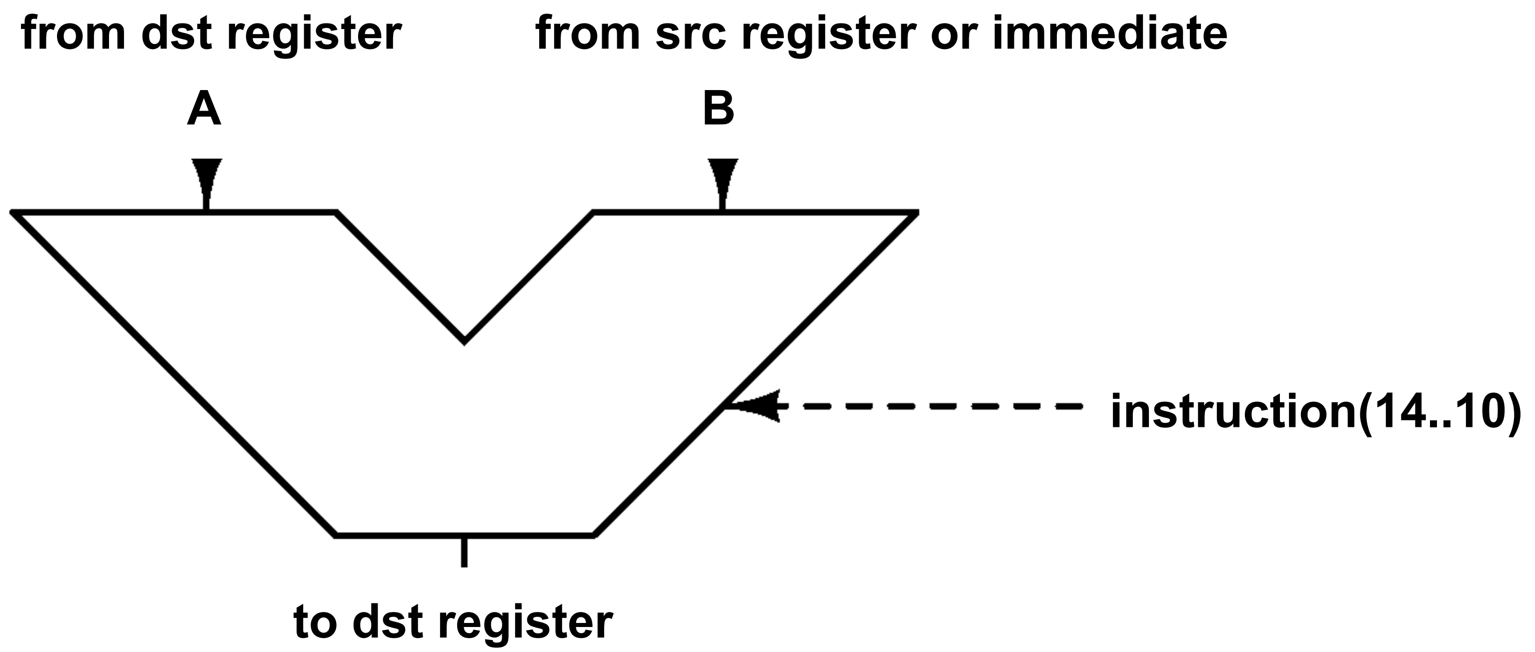
Default ALU data path

instruction(15..8)								Instruction(7..0)							
opcode			$\overline{R}/I$	ALU op		dreg		src							
0	X	X	X	X	X	r	r	i/-	i/-	i/-	i/-	i/-	i/-	i/r	i/r

- For ALU instructions, the control unit does the following:
  - Input A of the ALU receives the data from register dreg
  - Input B of the ALU receives either the immediate src or the value of the register src, depending on instruction(12)
  - The output of the ALU is stored in register dreg
  - The ALU receives instruction(14..10) to indicate the operation to perform.
- Providing data to the ALU and storing the result is done automatically by the control unit
- We simply need to add new function to the ALU - the rest is taken care of!

Default ALU data path

- The only part of the processor we look at is the ALU in cpualu.vhd
- It looks like this:



Tutorial I

- The instruction table shows that the processor has only 3 2-input logic functions: AND, OR, XOR
- The objective is to add a new logic instruction: NAND
- The ALU is defined in cpualu.vhd

Tutorial I

- Step 1: define which instruction code corresponds to the new function
  - The opcode has to be 001, 010, 011 (ALU operation)
  - Check the instruction table to find which opcode has "space" to contain a new instruction
  - Opcode 001 is full: all combinations instruction(12..10) are used
  - Opcode 010 is not full: only 4 combinations of instruction(12..10) are used for the xor and cmp. This could be used.
  - Opcode 011 is not full: only 5 combinations of instruction(12..10) are used. This could be used.
  - To keep some logic in the instruction set we decide to use opcode 010, as the existing instructions with opcode 010 are two-operand instructions (xor and cmp)
  - However nothing prevents us from using opcode 011!

Tutorial I

- Step 2: Decide which ALUop to use for the selected opcode
  - ALUop 00 is xor
  - ALUop 01 is cmp
  - Choose a value of ALUop for this instruction. Let's say 10!
  - We can update our instruction table:

Custom instruction	opcode			$\overline{R}/I$	ALU op		dreg		immedite / reg							
nand r, r	0	1	0	0	1	0	r	r	-	-	-	-	-	-	r	r
nand r, i	0	1	0	1	1	0	r	r	i	i	i	i	i	i	i	i

Tutorial I

- Step 3: Prepare to modify cpualu.vhd
  - The ALU is realized by a multiplexer selecting one function:

```
r <=                a+b when op(4 downto 3)="01" and op(1 downto 0)="00" else
                          sub(7 downto 0) when op(4 downto 3)="01" and op(1 downto 0)="01" else
                          a and b when op(4 downto 3)="01" and op(1 downto 0)="10" else
                          a or b when op(4 downto 3)="01" and op(1 downto 0)="11" else
                          a xor b when op(4 downto 3)="10" and op(1 downto 0)="00" else
                          not a when op(4 downto 0)="11000" else
                          '0'&a(7 downto 1) when op(4 downto 0)="11001" else
                          a(0)&a(7 downto 1) when op(4 downto 0)="11010" else
                          a(7)&a(7 downto 1) when op(4 downto 0)="11011" else
                          a(6 downto 0)&a(7) when op(4 downto 0)="11100" else
                          "00000000";
```

  - We must add an input to this multiplexer to get the NAND of inputs a and b when the opcode is 010 and ALUop is 10
  - The current ALU uses op(4..0) to select the operation.
  - Where does op come from? Read cpu.vhd to find out!
  - op is actually instruction(14..10)

Tutorial I

- Step 4: Modify cpualu.vhd
  - We want to execute the NAND when instruction(15..10) is "010010"..... (is this correct??!)
  - So op(4..0)="10010" for a NAND
  - Thus modify the ALU as follows:

```
r <=                a+b when op(4 downto 3)="01" and op(1 downto 0)="00" else
                            sub(7 downto 0) when op(4 downto 3)="01" and op(1 downto 0)="01" else
                            a and b when op(4 downto 3)="01" and op(1 downto 0)="10" else
                            a or b when op(4 downto 3)="01" and op(1 downto 0)="11" else
                            a xor b when op(4 downto 3)="10" and op(1 downto 0)="00" else
                            a nand b when op(4 downto 0)="10010" else
                            not a when op(4 downto 0)="11000" else
                            '0'&a(7 downto 1) when op(4 downto 0)="11001" else
                            a(0)&a(7 downto 1) when op(4 downto 0)="11010" else
                            a(7)&a(7 downto 1) when op(4 downto 0)="11011" else
                            a(6 downto 0)&a(7) when op(4 downto 0)="11100" else
                            "00000000";
```

Tutorial I

- Step 5: create a test program
  - We want to test the register mode and the immediate mode!
  - We use the instruction table to find the encoding

Custom ALU instruction	opcode			$\overline{R} / I$	ALU op		dreg		immedite / reg							
nand r, r	0	1	0	0	1	0	r	r	-	-	-	-	-	-	r	r
nand r, i	0	1	0	1	1	0	r	r	i	i	i	i	i	i	i	i

```
-- register mode
mov    ra,a5h        10a5
mov    rb,aah        11aa
nand   ra,rb         4801
-- expected result: 5F
mov    rc,83h        1283
nand   rc,84h        5a84
-- expected result: 7F
```

Tutorial I

- Step 6: test the program
  - Edit the memory to fill it with the instructions:

Address	Data
00	10
01	A5
02	11
03	AA
04	48
05	01
06	12
07	83
08	5A
09	84

Tutorial I

- Step 7: fix bugs
  - The program works with register source, but not with immediate source!
  - The error is in this line in the multiplexer in cpualu.vhd:

```
a nand b when op(4 downto 0)="10010" else
```

  - Can you fix this?

Tutorial II

- The previous instruction was combinational. We will now see how to make an ALU instruction whose internal function is clock dependent
- Let's say we want an instruction "tick" that returns the number of clock cycles since the processor was turned on.
- We will implement this with a circuit counting up at each clock cycle, with a synchronous reset
- The counter output will be connected to one input of the ALU multiplexer and the "tick" instruction will select that input

Tutorial II

- Step 1: define which instruction code corresponds to the new function
  - As before the opcode has to be 001, 010, 011 (ALU operation), and there is space in the opcodes 010 and 011
  - To keep some logic in the instruction set we use opcode 011 this time: all instructions with this opcode have a single operand, as does "tick"
- Step 2: Decide which ALUop to use for the selected opcode
  - ALUop 000 is not, 001 is shr, 010 is ror, 011 is asr, 100 is rol.
  - Let's choose the next free ALUop: 101
  - We can update our instruction table:

Instructions	instruction(15..8)								Instruction(7..0)							
Custom ALU instructions	opcode			ALU op			dreg									
tick	0	1	1	1	0	1	r	r	-	-	-	-	-	-	-	-

Tutorial II

- Step 3: Create the tick counter: a counter with reset
  - We could create a dedicated VHDL component (in a separate file), but we are lazy here and put everything in cpualu.vhd
  - Add a signal in cpualu.vhd: tick as an 8-bit std_logic_vector
  - Add the following counter code somewhere in the architecture:
    - We use a handy VHDL operation for the addition!
    - This is a D flip-flop with synchronous reset and an increment

```
process(clk)
begin
     if rising_edge(clk) then
             if rst='1' then
                     tick<="00000000";
             else
                     tick<=tick+1;
             end if;
     end if;
end process;
```

Tutorial II

- Step 4: Modify the multiplexer
  - Select "tick", the output of the counter, when instruction(14..10) is "11101"

```
r <=            a+b when op(4 downto 3)="01" and op(1 downto 0)="00" else
                    sub(7 downto 0) when op(4 downto 3)="01" and op(1 downto 0)="01" else
                    a and b when op(4 downto 3)="01" and op(1 downto 0)="10" else
                    a or b when op(4 downto 3)="01" and op(1 downto 0)="11" else
                    a xor b when op(4 downto 3)="10" and op(1 downto 0)="00" else
                    not a when op(4 downto 0)="11000" else
                    '0'&a(7 downto 1) when op(4 downto 0)="11001" else
                    a(0)&a(7 downto 1) when op(4 downto 0)="11010" else
                    a(7)&a(7 downto 1) when op(4 downto 0)="11011" else
                    a(6 downto 0)&a(7) when op(4 downto 0)="11100" else
                    tick when op(4 downto 0)="11101" else
                    "00000000";
```

Tutorial II

- Step 5: Create a test program.
  - We will create a loop with register RB that goes from 0 to 4
  - We will put the number of clock cycles to complete the look in d (we can verify later that this is correct!)
  - Then the program continuously loads c with the tick

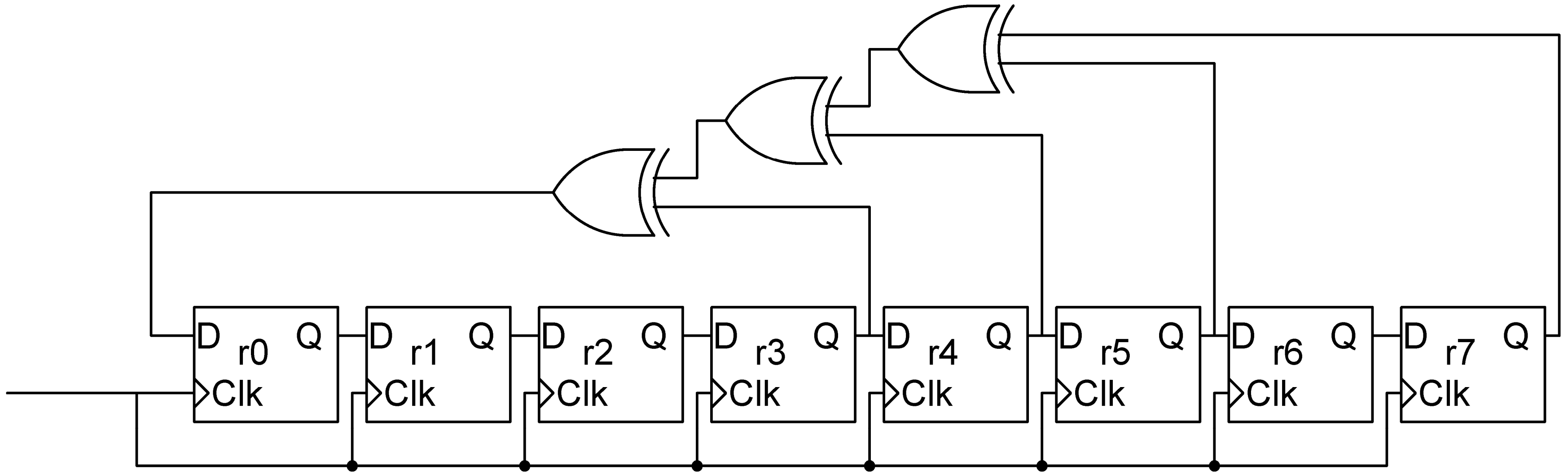
```
0 mov rb,00          1100
2 add rb,1           3101
4 cmp rb,4           5504
6 jb 2               BB02
8 tick rd            7700
A tick rc            7600
C jmp A              B00A
```

Tutorial II

- Step 6: Fill in the memory, reset the CPU and test the program
  - What do you obtain in register C eventually?
  - Is this what you expect from the program? Verify this by counting the number of instructions executed in the program! Hint: the program loops 4 times the instructions between address 2 and 6....
  - You may find that your count is off by one cycle... can you explain this?

Problem 1: Random Number Generator (coursework assignment)

- Creating a pseudo-random number can be realized with a linear feedback shift register:



- The 8-bit number represented by bits r7..r0 is pseudo-random
- In reality the sequence is periodical
- If the taps of the XOR are well chosen (as in this circuit) the periodicity can be of maximal length (here 255 clocks)
- Note that this circuit must be initialized properly: upon reset, at least one bit must be one. Let's say r7=1 on reset

Problem 1: Random Number Generator (coursework assignment)

- The objective is to add an instruction to the ALU that returns a random number using the LFSR circuit and demonstrate that it works with a test program
- You are free to select any unused instruction encoding
- Check the coursework assignment for details on what you must include in the report.
- Hint: you can create a standalone VHDL module to test the LFSR independently of the processor, and then instantiate the circuit in the ALU