# Week 6 Lab B: Building a simple CPU

## Objectives

Develop understanding and experience of:

1. How components we have built in previous weeks could be combined to automate running of instructions.

This lab involves building a simulation that uses the same ideas as the one used in the coursework, so understanding this work will really help with understanding the coursework.

## Automation with an ALU

You may download the Logisim circuit file (Week6LabB\_simpleCPU) from the Week 6 lab starter files on Moodle. This contains a completed ALU from week 4.

The ALU has three inputs, two 8-bit data inputs to carry out the operation on and a 2-bit input to indicate which operation to do. The ALU has one 8-bit output, the result of the calculation (or logical operation).

The operations are as follows:

|  |  |  |
| --- | --- | --- |
| ALU OP value | Short format | Output |
| 0 | ADD | The result of adding A and B together |
| 1 | AND | The result of a bitwise A AND B |
| 2 | SUB | The result of subtracting 1 from A |
| 3 | LEFT | The result of shifting the bits in A left by one bit so that the rightmost bit will be filled with a zero. |

This task takes the ideas from the circuit built in part A and replaces the addition that feeds into the accumulator register by an ALU. This means that the ROM will have to hold both the number to be operated on and the operation to be carried out. The circuit is shown below and was talked through in last week’s lecture.

The register on the left is the Program Counter (PC) and works in the same way as the circuit in part A. The register on the right is the accumulator (ACC) and its output feeds into the first input of the ALU (as it previously fed into the addition). This aspect is also seen in the coursework circuit.

A diagram of a circuit

Description automatically generated

Hints for building the computer:

1. Create a new circuit for this computer (in the same project as the complete ALU) and add the ALU as a sub-circuit.
2. Use registers to store 8 bits (as in part A).
3. The ROM should store 16 bits (to allow for extensions).
4. The data coming out of the ROM now needs to be split to feed into input B and the ALU\_Opcode on the ALU.
   * The first splitter has 16 bits going in and fans out to two parts. The bits that correspond to the last two hex digits from the instruction come out as bits 7 to 0 and need to feed in to the ALU.
   * The second splitter takes bits 15 to 8 from the first splitter. This means it has 8 bits going in and needs to fan out into four equal parts. The ALU operation needs only two bits (values 0 to 3), but we are using the first hex digit from the instruction. That first digit will always have two zeros as the first two bits from the left, so we take the next two bits to the ALU operation.

We need to consider how to test that it is working correctly by planning a program and working out the expected values separately. We can load a program into the ROM and step through that program by clicking the clock input (with the hand icon highlighted). Last week’s lecture talked through one possible program including writing the instructions and predicting the results, but you should take the same approach for a different program.

The program for you to write should work as shown in the table below. Use a conversion website to convert between decimal, binary and hex if necessary. You may need some additional rough notes for working out the result of the bitwise AND operation. Remember that the instruction will be the ALU operation as a hex digit, a zero and the immediate value (when needed) as two hex digits.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Program counter | Operation (decimal) | Instruction in hex | Expected value in accumulator (binary) | Expected value in accumulator (hex) |
| 0 | Add 27 | 001b | 0001 1011 |  |
| 1 | Left shift | 3000 |  |  |
| 2 | AND with 60 |  |  |  |
| 3 | Subtract 1 |  |  |  |

Explaining the instructions:

1. Add 27. First digit 0 as the ALU op code for addition, Second digit always zero, Third and Fourth digits are decimal 27 converted to hexadecimal as 1b
   * The expected value in the accumulator will be 1b
2. Left shift doesn’t need an immediate value so we only need the first digit to be the ALU\_Op\_Code from the earlier table and the remaining digits are 0.
   * Use binary to work out the expected value by shifting by 1 bit. Convert that to hex.
3. Bitwise AND with decimal 60. Convert 60 to both binary and hexadecimal. The instruction will need the ALU op code, a zero and the hexadecimal form. Work out the expected value in binary by doing a bitwise AND of the binary for 60 with the previous binary value. Convert to hex.

Add images to show your circuit and testing.

## Extension Exercise

Although you have built a programmable computer, there are still many aspects in which this simulation is not very flexible.

The aim of this extension is to remove the accumulator (ACC) and instead use a bank of 4 registers, having features in common to the RISC-V architecture you are using for the assembly language programming part of the unit. This will also mean that the instruction needs to include more information and so will need much more by way of “decoding”. This is broken down into two steps. This extension is designed to be quite involved and so would probably benefit from working in small groups. The result is a simulation that is different from the one used in the coursework but should be rewarding to complete and convince you that you could build a full RISC-V computer in Logisim Evolution.

1. A bank of four registers with a source and destination specified in the instruction so that the new instruction has the format as shown below.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Unused (00) | | ALU operation | | Destination register number | | Source register number | | | Immediate value to use as input | | | | | | | |
| Hex digit | | | | Hex digit | | | | | Hex digit | | | | Hex digit | | | |

* Start by copying the simulation as you had it from the main exercise above.
* You should create a set of four general registers that can receive the value from the accumulator or be used to feed a value to the accumulator.
* You already have a splitter that takes the first two hex digits of the instruction and splits into 2-bit pieces. You will now use two of the previously unused outputs from that splitter to represent registers.
  + I will refer to the registers as T0, T1, T2 and T3. You will be referring to registers by letter and number in the RISC-V assembly language programming.
* Remove the accumulator
* The output from the ALU needs to go to the register given by the number for the destination from the instruction.
  + To keep in line with the RISC-V architecture that you will be using, you should not allow register T0 to change from storing 0. Set the WE signal for that register from a constant zero.
  + Think about how you can use a decoder to make sure that the correct register is updated.
* Input A of the ALU that previously came from the accumulator needs to come from the register given by the source register number from the instruction. Remember which component is useful when selecting one from a set of options.

Next you will need to construct a program to test your new CPU simulation. Immediate values should still work in the instruction as in the main part of the lab.

Suggested program (using addition only)

1. Add immediate value 8 to register T0 (source) and store in T1 (destination)

* Remember that T0 will always be zero so this is equivalent to T1 = 0 + 8

1. Add immediate 7 to register T0 and store in T2
2. Add immediate 2 to T2 and store result in T3
3. Add immediate 7 to T1 and store in T1
4. Add immediate 0 to T0 and store in T2

To get the program into the program memory, the bits for the destination and source registers will have to be converted to hex. For example, the first step in the program has destination 01 and source 00 giving hex 4.

Work out the instructions you want to use (in hex) and work out what results you expect to see in the different registers at each step. Remember that you can save the “image” of your program.

When your simulation works, it will be capable of things that look more like programming, for example steps 3 and 4 of the program above would be equivalent to

T3 = T2 + 2

T1 = T1 + 7

1. The simulation could still be more flexible as input B to the ALU is still only taken from the immediate value in the instruction but can’t be taken from a register. So, we wouldn’t be able to do code like

T1 = T2 + T3

The task now is to allow input B to the ALU to be from a numbered register or from an immediate value.

We will need to know whether to use a register or immediate value for input B, so the instruction format now includes a **Mode** (M) which will be 0 when the immediate value should be used and 1 when two bits should be used as the second source register. This means that the processor will be backwards compatible – instructions written for the previous version will still work in the same way as before.

The new instruction format is as follows:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 14 | 13 | 12 | 11 | 10 | | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| M  (0)  (1) |  | ALU operation | | Destination register number | | Source (A) register number | | | Immediate value to use as input  **or** | | | | | | | |
| Source (B) register number | |  | | | | | |
| Hex digit | | | | Hex digit | | | | | Hex digit | | | | Hex digit | | | |

The splitting of the instruction will be more complex now.

The first bit (bit 15) will have to be separated from the instruction so that it can be used to select which value to pass to input B of the ALU. Bits 6 and 7 will need to be separated and used to select which register to read a value from. Note that this reading of a value will have the same approach as reading the value going in to input B of the ALU and will have to happen at the same time so that both inputs are fed in on the same clock cycle.

Again, you need to have a test program where you have worked out all the machine instructions for the program and separately worked out the expected outcomes. This time I will write the program in notation that matches other programming languages (e.g. Java, Python). Remember that we can do subtraction by adding a negative.

Suggested program:

1. T1 = T0 + 8
2. T2 = T1 + 3
3. T3 = T1 + T2
4. T2 = T1 - 5
5. T3 = T3 + T2
6. T3 = T3 + T1

Note that three of the instructions use three registers and need mode to be set to 1. The other instructions use an immediate value as previously.

It would probably be a good idea to create a table of values you expect in each register after each instruction has run.