# Simple CPU 2/2: 4-bit CPU

Anas Shrinah

November 2, 2021

In this lab, you will implement a 4-bit CPU starting from the 4-bit data path designed in the previous lab. You will use ModuleSim to implement your design and test it.

Credits: The content of this worksheet is based on lab materials originally prepared and developed by previous lecturers of COMSM1302 at the University of Bristol.

## Goals of this lab

In this lab:

- You will implement a clock system to produce the required clock phases for your CPU to work.
- You will develop the control unit of your CPU by implementing the logic required to execute the instructions of your instruction set.
- You will test your instructions as you go.
- You will use your CPU to execute the code of finding the difference between two two-digit numbers from the lectures.

This lab aims to help you to improve your understanding of the instruction cycle. The lab will also introduce you to the process of implementing hardware instructions. Additionally, the lab will help you to appreciate how the CPU solve complex problems by executing simple operation.

It is recommended that you work in small informal lab groups, e.g. with two to three students located close to you in the lab. Please note, however, that each of you needs to develop a good understanding of the material so that you can perform the tasks on your own once you have worked through the lab sheets.

## Questions

If you have any questions, please ask. Make use of the TAs, but do not expect them to give you complete designs, they are there to guide you, not do the work for you. Please be patient. You may need several attempts to get something working.

When requesting help, TAs will expect you to show them what you have done so far (no matter how sketchy) and they will ask you to clearly explain your reasoning. This makes it easier for the TAs to help you. For this reason, priority will be given to students who can show and explain how far they have got.

## 1 Starting ModuleSim

ModuleSim is a Java application. It requires a Java installation to run, so you may need to install Java before you can run ModuleSim. (This applies in particular if you want to run ModuleSim on your own computer.)

ModuleSim can be obtained from `https://github.com/uobteachingtechnologist/ModuleSim/releases`. Simply download the JAR file and run it from the terminal.

You may want to place the JAR file into the working directory from which you start your terminal and where you intend to keep your work. Assuming the download is called `ModuleSim.jar`, you can then run ModuleSim as follows:

```
java -jar ModuleSim.jar
```

Or, if your JAR file is in your `Downloads/` directory, you can run it using:

```
java -jar ~/Downloads/ModuleSim.jar
```

You will need to use ModuleSim in order to complete the tasks in this lab.

## 2 The instruction cycle

Instructions pass through four stages. Each stage consists of two phases.

1. Fetch the next instruction
    a) Fetch the next instruction from the memory and store its operand it in the holding register – phase "1a".
    b) Store the instruction in the Opcode and O registers - phase "1b".
2. Increment the program counter
    a) Add one to the value of the PC and store it in the holding register – phase "2a".
    b) Store the new PC value in the PC register – phase "2b".
3. Execute the instruction
    a) Store the result of the execution which can come either from the AU or from the memory for LDAM and LDBM in the holding register – phase "3a".
    b) Store the result of the execution which come from the holding register in its final destination which can be a register or a memory location for STAM– phase "3b".

More detailed explanation will be provided later. In the next task you will design the clock circuit which will expand the clock module phases from "a" and "b" to "1a", "1b", "2a", "2b", "3a", and "3b".

In which phase does the holding register need to update its value?

## 3 Clock circuit

CPUs need a clock to orchestrate their activities. It is like the drummer who leads the paddlers throughout a boat race using the rhythmic drum beat to indicate the frequency and synchronization of all the paddlers' strokes.

Our CPU needs 6-phase clock with phases 1a, 1b, 2a, 2b, 3a, and 3b to coordinate the fetch, increment PC, execute phases of the instruction cycle. However, the clock module offers us only two phases "a" and "b". Therefore, our first task is to design a clock circuit with 6 phases. The first step it to design a counter so we can use its value to indicate the different phases. I.e., the counter value zero indicates the first phase, one indicates the second phase, and three indicates the third phase.

### 3.1 4-bit counter

Build a 4-bit counter circuit containing:

- Clock
- AU
- Two registers
- Input switch

### 3.2 Decode the counter value

Now we have a working counter. We need to decode its value to produce three different signals.

1. How many different outputs can we select if we feed the first two bits from the counter to the address port of a demultiplexer?
2. What should be the input signal to the demultiplexer to get 0001 at its selected output?

Add a demultiplexer and an input switch and connect them as per your answers.

Now we have two clock phases: “a” and “b” from the clock module. We also have four different signals from the outputs of the demultiplexer, though we only need the first three outputs.

In the following task, you will combine the signals from the counter-decoder and the two clock phases to generate the clock phases 1a, 1b, 2a, 2b, 3a, and 3b.

To reset the counter, press the reset button on the clock module. This action will cause all registers that are connected to the clock signals to reset.

**Checkpoint: show your processor design to a TA or staff member for feedback.**

## 4 Instruction cycle stages

### 4.1 Fetch the next instruction

Instructions are first fetched from memory using the PC register as an address. The four high bits of the instruction have to be stored in the Opcode register and the four low bits in the “O” register. The four low bits (the operand) first have to be stored in the holding register, this happens in phase “1a”.

As you will notice, the holding register will always update its value during phases “1a”, "2a", and “3a”. So this register can be clocked by the “a” output of the clock module directly.

Then in the “1b” phase, the Opcode and “O” registers should be enabled, so Opcode stores the instruction’s opcode directly from the four high bits of the memory, whereas the “O” register stores the instruction’s operand which was stored in the holding register in the previous phase.

To get the Opcode and the “O” registers to update their outputs in clock phase “1b”, use a split-merge to combine the “1” signal from the counter-decoder and “b” output from the clock module correctly. If you press the reset button on the clock, then this signal should get to the Opcode and the “O” registers and their value should be cleared.

### 4.2 Increment the program counter

The next step is to increase the PC so it points to the next instruction due to be fetched.

- In clock phase “2a”, the AU should take the PC as input and add one. The result should be stored in the holding register. The holding register is already clocked by the “a” phase.
- In clock phase “2b”, the PC should be enabled to copy the new PC value from the holding register.

To get the PC to store the new value in clock phase “2b”, use a split-merge to combine the “2” and “b” signals correctly.

We still have to figure out how to set the control bits of the AU and its right and left inputs’ multiplexers.

### 4.3 Execute the instruction

We have seen how to control the holding, program counter, Opcode and “O” registers. The control signals to set up the data path and to control the “A” and “B” registers are discussed in the following section.

| Target module | Signal description | Signal name |
|---|---|---|
| Reg A | Enable signal | Write A |
| Reg B | Enable signal | Write B |
| Mux A/PC | Select A or PC as the left input of the AU | Mux A/PC |
| Mux B/O | Select B or O as the right input of the AU | Mux B/O |
| Mux AU/Memory | Select AU or memory as the input of the holding register | Mux AU/Mem |
| AU mode | Control the mode of the AU | AU mode |
| Memory | Write | Write Mem |

**Table 1:** Control signals.

# 5 Setting up the data path for instruction execution:

The data path has a number of components that have different configurations, for example:

- The AU can be in the add or subtract mode, and the carry-in can be on or off.
- The MUXes before the left and right inputs to the AU can be in one of four input configurations each.
- The "A" and "B" registers can be either enabled or disabled.

We need seven control signals to control all configurable components of our system. The control signals in Table 1 assumes A and PC are connected to the first two inputs of the multiplexer of the first input of the AU, and the B and O registers are connected to the first two inputs of the multiplexer of the second input of the AU.

Note, for the rest of this document, if you have chosen a different connection, then you have to adapt the next steps to match your design.

## 5.1 Write A and Write B

The "Write A" signal has to be high to save the result of the execution of any instruction that needs to update the value of register A.

Which instructions do need to update the value in register A?

In the previous lab, we have produced one signal for each instruction using two layers of demultiplexers. If you feed the input of the first demultiplexer with phase 3 from the counter-decoder, the instruction signals will be high during phase 3 when their opcodes are present in the Opcode register.

Add a chained OR and label it "Write A". The output of this chained OR is the control signal "Write A". The inputs of this chained OR must be connected to the signals of all instructions that need to write to register "A".

Updating the value of register "A" happens during phase "3b". Therefore, The signal "Write A" (The output of the chained OR) must be combined with the clock phase "b" to enable the registers "A" to update its value correctly. Use a split/merge to do this.

Repeat the previous step for the control signal "Write B".

## 5.2 Write Memory

STAM is the only instruction that writes to the memory, and it takes the data from the "A" register. Writing to the memory is like writing to a register in terms of the control signal. So, the signal Write Memory, which is in our case the signal of the instruction STAM, has to be combined with the clock phase "b" so the value of the register "A" gets stored into the memory. You also need to turn the memory's write enable switch on. In the simulator, click it and it should turn blue. When the memory is writing data, the write LED comes on.

## 5.3 Mux A/PC and Mux B/O

So far, the control signals we have discussed have only one bit. On the other hand, the multiplexer's control port has two bits. We need Mux A/PC to be able to pass either PC, "A", or zero. Therefore we need to

produce the control patterns 00, 01, and 10.

Why do we need Mux A/PC to be able to pass zero?

Use a chained OR for the first control bit of Mux A/PC. Call this signal Mux A/PC-0 and another chained OR for the second control bit of Mux A/PC. Name this signal Mux A/PC-1.

Your task is to merge the singles Mux A/PC-0 and Mux A/PC-1 into one signal and use it to control the multiplexer Mux A/PC.

The inputs of the chained ORs Mux A/PC-0 and Mux A/PC-1 have to be connected to the instruction signals as needed. For example, if an instruction, increment PC or Fetch (pass PC) signal needs the Mux A/PC to pass the value of the PC, this signal must be connected to Mux A/PC-0 only. When this signal is high, only the first bit of the control port of Mux A/PC will be high i.e. the control port will receive the signal 01. This will make the Mux A/PC pass the second input which is the PC value.

If some instructions need the value of the "A" register to be passed, then the signals of these instructions should NOT be connected to neither Mux A/PC-0 nor Mux A/PC-1. For these instructions, the signals Mux A/PC-0 and Mux A/PC-1 will be equal to zero and the control port will receive the signal 00. This will make the Mux A/PC pass the first input which is the "A" register value.

Do the same steps as before to create the signals Mux B/O-0 and Mux B/O-1.

## 5.4 Au mode

We need the AU mode to be in add without carry, add with carry or subtract modes. Find out what are the bits pattern required to set the AU in these modes. You will need to control just two bits to choose between these modes.

Add a chained OR for one of the control bits of the AU, and another chained OR for the other control bit of the AU. Your task is to merge the outputs of these two chained ORs into one signal and use it to control the mode of the AU. Note that these two signals do not have to be necessarily connected to the first and second bits of the control input port of the AU.

Make a table with two columns: AU mode and AU control input. Add three rows to this table for each required mode namely: add without carry; add with carry; and subtract modes. Now check what bits of the AU control input's four bits need to be set to one for each of the required modes.

Use a split/merge component to merge the outputs of the chained ORs such that these outputs are linked to the required control bits of the AU.

Now connect the instruction and "Increment PC" signals to the inputs of these two chained ORs are required. For example, the signal of the ADD instruction should NOT be connected to any of these chained ORs, because the ADD instruction needs the AU mode to be in add without carry. This mode is achieved by having the pattern xx00. On the other hand, you need the signal of the SUB instruction to be connected to both chained ORs. This is needed to have the pattern x11x at the control input of the AU.

## 5.5 Mux Au/Memory

The case with this multiplexer is easier than the other multiplexers. Here we just need one control bit to choose between the output of the AU and the data coming from the memory.

If you have connected the memory to the second input of Mux Au/Memory, then any instruction that needs to get data from the memory, like LDAM, would require the Mux Au/Memory to select the memory.

Add a chained OR and label it Mux AU/Mem. Connect all instructions that need data from memory to this OR. Do not forget to connect the signal "Fetch" (Pass PC) to this OR as well, because during the fetch phase we need the operand to be stored in the holding register. Thus, the Mux Au/Memory must select the memory during the fetch phase.

Connect the signal Mux AU/Mem to the control input of the multiplexer Mux Au/Memory.

# 6 Review and test your instructions

## 6.1 Constant loads : LDAC and LDBC

Since instructions execute in clock phase 3 and the current operand is already stored in the "O" register, the function of the instructions LDAC and LDBC is to copy the "O" register into the "A" and "B" registers, respectively. To execute LDAC or LDBC, the data path should pass the contents of the "O" register through the AU. Make sure the signals of the instructions LDAC and LDBC are connected to the chained ORs Mux A/PC-1 (to pass zero) and to Mux B/O-0 (to pass the value of the "O" register). The result is the value of the "O" register gets copied to the holding register in phase 3a.

In phase 3b, you want to clock the "A" register if the instruction is LDAC and the "B" register if the instruction is LDBC. You have taken care of this in Section 5.1

To test your processor, create a text file `ldac.hex` with the following contents and load it to the memory.

```
01 03 07 0f 11 13 17 1f
```

This disassembles to the following instructions:

```
01 LDAC 1
03 LDAC 3
07 LDAC 7
0f LDAC f
11 LDBC 1
13 LDBC 3
17 LDBC 7
1f LDBC f
```

Reset your system by clicking on the reset button of the clock module, all registers will be cleared. Start advancing the clock of your CPU. You should see first the "A" then the "B" register step 0001, 0011, 0111, 1111.

**Checkpoint: show your processor design to a TA or staff member for feedback.**

## 6.2 Memory reads

You have already implemented LDAC: areg <- oreg by setting up the data path so that in phase 3a, the "O" register contents go through the AU into the holding register; in phase 3b you copy the holding register to the "A" register.

LDAM: areg <- mem[oreg] is almost identical to LDAC, there is only one component on the data path that you have to set differently. Can you see it? mem[oreg] means the value in the memory at address oreg, which you get by feeding the "O" register into the memory's address input and reading the result of the memory's data output. So we need the store the data from the memory (not from the AU) into the "A" register. Make sure LDAM, LDBM, LDAI and "Fetch" (Pass PC) signals are connected to the chained OR Mux Au/Mem as explained in Section 5.5. Also, ensure these signals are connected to Mux A/PC-1 (to pass zero), and Mux B/O-0 (to pass the value of the "O" register).

You should come up with your own test program to test that these three instructions are working correctly. This involves both writing a test program and documenting what effects you expect to see when that will tell you whether your processor is working correctly. This also applies to the following exercises, where it will no longer be mentioned

**Checkpoint: show your processor design to a TA or staff member for feedback.**

## 6.3 Storing to memory

STAM means to take the value in the "A" register and store it in memory at the location pointed to by the "O" register. To do this, in phase 3a we want to copy the "O" register to the AU, which feeds the memory's address input.

Like the memory reads instructions, the STAM signal has to be connected to Mux A/PC-1 (to pass zero), and Mux B/O-0 (to pass the value of the "O" register). This will pass the value of the "O" register to the output of the AU which feeds the address port of the memory. Writing to the memory was done in Section 5.2.

**Checkpoint: show your processor design to a TA or staff member for feedback.**

### 6.4 Arithmetic : ADD and SUB

ADD and SUB instructions needs the AU inputs to be the value of the "A" and "B" registers. Make sure you have connected the signals of these instructions such as these signals are not connected to the chained ORs of Mux A/PC and Mux B/O. Setting up the AU mode was discussed in Section 5.4

ADD and SUB instructions write to the "A" register. Therefore, ensure these signals are connected to the "Write A" chained OR.

**Checkpoint: show your processor design to a TA or staff member for feedback.**

## 7 Test your CPU

Assemble and run the code presented in Lecture 10 of calculating the difference between two two-digits numbers in BCD. This involves:

1. In ModuleSim, pause the simulation.
2. Reset the clock.
3. Translate the assembly instructions into machine codes as per your instruction set.
4. Save this machine code in `diff.hex` file.
5. Load this file to the memory.
6. Advance the clock by performing step by step simulation.
7. Check values of the registers are as expected.
8. When the execution of the instructions of the program, you should find the difference value stored in the memory location 0x9

**Checkpoint: show your processor design to a TA or staff member for feedback.**

## 8 Extra tasks

Please only attempt these tasks when you have done all other sections.

### 8.1 Unconditional branching: BR

Your task is to implement the unconditional branching instruction BR. BR loads the PC with the sum of the "O" register and the PC. PC <- PC + O. Basically, it performs a jump. The PC value after executing this instruction will jump to a new location that is far from the PC value by the value specified in the operand.

There is nothing new about this instruction. It just loads, computes and stores data in a register, except now it stores to PC instead of the "A" register. There is, however, something new about when you enable the PC. It should write:

- Always, in phase 2b (for increment), or
- Again in phase 3b, but only if the instruction is a BR.

Note: if instruction 4 is BR 2 then the PC ends up at $7 = 4 + 2 + 1$, not 6. This is the correct behaviour! The reason is that in phase 2 of instruction 4 you already incremented the PC to 5; in phase 3 you add the 2 from the operand to it again to get 7. It is the programmer's job to take this into account.

### 8.1.1 Halt the processor

At the end of your code, use BR with operand to perform a jump across the memory to reach back to the address of this BR instruction.

Write a program to do some calculation, then insert BR instruction at the end of your code with the correct operand to enter an infinite loop. Observe and describe what happens to the values of the registers "A", "B", and "O" after executing this instruction.

## 8.2 Conditional branching BRZ

The conditional branching instruction BRZ loads the PC with the sum of the O register and the PC if the value of the "A" register is equal to zero. if A = 0 then pc <- pc + oreg.

First, place a new 8-bit AU pair in your design; you'll use it for the "if" tests. The only thing we want to test is the "A" register value, so connect the "A" register to the new AU left inputs.

Secondly, set up the control grid. The PC enable condition is now:

- Always, in phase 2b (for increment);
- In phase 3b, if the instruction is a BR; or
- In phase 3b, if the instruction is a BRZ and the result of the comparison AU shows the content of the "A" register is equal to zero.

### 8.2.1 For loop

Write an assembly code using our instruction set to perform a for loop.

## *Congratulations* you have built a complete processor!