Simple CPU 1/2: 4-bit data path

Anas Shrinah

November 5, 2021

In this lab, you will design a 4-bit data path that will form the core of our CPU. You will use ModuleSim to implement your design and test it. The labs allow you to implement some of the theoretical concepts from computer architecture.

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

The aim of this lab is to get you started with ModuleSim, help you to understand the role of each component in the data path, help you to learn about the design of instruction sets and how it affects the hardware design in CPUs. Moreover, this lab will develop your design and problem-solving skills.

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 and the lab next week.

2 Simple 4-bit data path

A data path is simply a loop containing an AU and registers. The data path is configurable in the sense that you can control which registers are read and written to in each clock cycle. We also can control which registers are connected to the inputs of the AU.

2.1 Components

To build a 4-bit data path, as described in the lecture slides, you will need the following ModuleSim components:

- 1 AU
- 1 clock - set this to manual
- 1 multiplexer
- 3 registers
- 2 split/merges
- 4 inputs
- 2 fan-out

2.2 Wiring

The AU only needs to support add mode so you can leave its control input empty. Use one of your registers as a holding register - it should be clocked in phase "a" and receive values from the AU. We will name the other registers "A" and "B".

1. For one input of the AU, you should be able to choose whether to take a value from the "A" register or a switch input. Which module does allow you to select from different inputs? How can you control this module?
2. The other AU input can just be connected to the "B" register.
3. The "A" and "B" registers should update their values when the "b" clock phase is high if we choose so. As such, you should be able to enable or disable them at your will. If a register is enabled, it will update its output only when the "b" clock phase is high. On the other hand, the register will not update its output if it is disabled even when the "b" clock phase is high. So, the idea is we need an input switch to enable or disable the register "A" and an input switch to enable or disable the register "B". To enable a register, you need to manually turn its enable switch on, so when the clock phase "b" is high, the register will update its output. Use a switch and split/merge for each of these two registers so that you can choose whether to write to them or not. Read the specification of the registers and split/merge modules to figure out how to connect them correctly. Basically, when the LSB of the switch is turned on, and the clock phase "b" is high, the connected register should copy its input to its output.
4. Do not forget to fan out the holding register output to the inputs of the "A" and "B" registers to complete the data path loop.

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

2.3 Testing

When your data path is set up, you should be able to use it to answer the following questions. For each task, the clock should be set to manual so you can alternate between changing switch inputs and advancing the clock.

1. What are the steps to load a value into the "A" or "B" register?
2. How to clear a register on its own without resting all other registers? I.e., without using the reset button.

xxdc should be connected to clock, and dcba should be connected to switch. Then we can set the second digit of switch to 1 to reset a register.

3. Try to compute 3 + 5 + 6 and store the result in the "A" register.

You have now built a manual CPU that can execute the following instructions in any combination and order.

1. Load value x into the "A" register, let's call it LDAC. LD from the verb load, A from the register "A" and C from the word constant. This instruction takes an operand, the value that need to be loaded to the register "A".
2. Load value x into the "B" register, let's call it LDBC.
3. ADD - add the values in the "A" and "B" registers and place the result in the "A" register.

2.4 First assembly program

Try to mentally figure out what would be the result of executing the following program.

```
LDAC 2
LDBC 3
ADD
LDBC 5
ADD
LDBC 6
ADD
```

A 2 5 10 16
B 3 5 6
3+2+5+6=16

What are the values of the "A" and "B" registers at the end of the execution? You can try to use your data bath to execute this code manually to verify your answers, or you could wait until we build an automated CPU to do the execution on your behalf, your choice!

Save your current design as `4-bit-data-path.modsim`. In the next tasks, we will start with a new file.

3 Advanced 4-bit data path

You can start to see how the CPU is going to work. Instead of setting instructions by hand, our processor will fetch them from the memory, decode them and set the AUs and multiplexers and registers up correctly. Therefore, to automate our data path:

1. We need a place to store the instructions of our program.
2. We need to agree on the binary representation of our instructions.
3. We also need a counter to tell us which instruction we need to execute next.
4. We need to be able to fetch the instructions from the memory.
5. We need a system to decode the opcode of the instructions and to set the AUs and multiplexers and registers up correctly so the instructions can be executed properly.
6. On top of this, we need a clock to enable us to control the registers and other parts in a timely manner.

We will consider these requirements in the following tasks and during the next lab. Our next task is to agree on what instructions do we want our CPU to execute and what are the codes of these instructions.

3.1 Our instruction set

In addition to LDAC, LDBC and ADD, we would like our CPU's data path to support the execution of two instructions to load the "A" and "B" registers with a value from memory, namely LDAM, LDBM. An instruction to store the value of the register "A" into the memory, let's call it STAM. Also, the CPU should be able to execute a subtraction instruction SUB. Finally, we need one more instruction, LDAI, to use the value in the register "A" as an address and to retrieve the data stored in location [A + offset] in the memory and store it in the register "A". These instructions are listed in Table 1

Machine code	Mnemonic	Description	Example	
0000	LDAC	Load register A with the operand value	00000101	Load A with 5
0001	LDBC	Same as LDAC, but for B	00010110	Load B with 6
0010	LDAM	Load register A with the value stored in the memory location that is addressed by the operand	00100101	Load A with the content of the 5th byte.
0011	LDBM	Same as LDAM, but for B	00110110	Load B with the content of the 6th byte.
0100	STAM	Store the value of A in the memory location addressed by the operand and reset the 4 MSBs of this location.	01001010	Store A in the 10th byte of the memory and reset the 4 MSBs of this location.
0101	ADD	A <- A + B	0101****	
0110	SUB	A <- A - B	0110****	
0111	LDAI	Load register A with the content of the memory location at an address calculated by adding the A register and the operand	01110011	Load A with the content of the memory [A + 3]

Table 1: Our instruction set.

3.2 Registers

To implement this instruction set, we need more registers and we need to do some re-wiring. But first, let's get started by getting the core of the CPU ready.

1. Create new design and save it as `4-bit-CPU.modsim`
2. In `4-bit-CPU.modsim`, Add an AU and three registers.
3. Name the registers "A", "B" and "holding".
4. Connect the AU output to the holding register.
5. Leave the "A" and "B" registers unconnected.
6. For all of the following tasks, use the file `4-bit-CPU.modsim`.

Program counter

First, we need a register to point to the next instruction that is due to be fetched and executed. Add a register and call it "PC" the program counter. Later we will work out how to increment the PC register during the instruction execution cycle.

Opcode and operand

As you have noticed, instructions consist of opcode: the binary number that represents the instruction; and operand: the constant value in the LDAC 2 for example. So, we need two more registers to hold the opcode and operand of the instructions when they are fetched from the memory to the CPU. Add two registries and name them "O" for the Operand register and "Opcode" short for operation code. Do not connect them yet.

3.3 Wiring

Fan out the data output from the holding register to the "A", "PC", "B" and, "O" but not to the Opcode register.

Add one multiplexer before each AU input. Connect the multiplexers to the AU inputs, but do not connect the registers to the multiplexers' inputs.

We need to decide how to connect the output of "A", "PC", "B" and "O" registers up to the AU's input multiplexers. This exercise is about working out what should be connected to what; the main effort is theoretical.

Look at the 8 instructions in our instruction set. Each of these will use the AU exactly once to do either an addition, or a subtraction, or just pass a value through unchanged (add that value to zero). In addition to the instructions, there is the case when we need the data path to pass the program counter unchanged to the memory to fetch the current instruction. Additionally, there is another case when we need the AU to increment the program counter which we can imagine as pc <- pc + 1.

For each of the instructions and for passing and incrementing the program counter, decide how the AU should be set up by making a table with 3 columns:

1. AU mode (add without carry, add with carry or subtract).
2. AU left input (A input) - either a register or a constant zero.
3. AU right input (B input) - either a register or a constant zero.

Each row of the table should represent one instruction, add two extra rows for "Fetch" (Pass PC) and "Increment PC". For example, the cell in the "ADD" row and "AU left input" column will indicate what the AU left input should be when it is executing an ADD instruction.

Once you have created the table, wire up the registers to the multiplexers' inputs. If you have decided that in some row of your table, the left input to the AU will come from the "A" register then you will need cables from the "A" register to one input of the multiplexer for the AU left input.

Our CPU is very simple that you should not need to connect any register to the right AU's input for some instructions and to the left AU's input for other instructions,

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

3.4 Memory

Now it is the time to add the memory. The requirements of the memory are as follows.

1. The memory should be addressed by the AU's output.
2. The low bits of the memory output data should reach the holding register (instruction operand).
3. The high bits of the memory output data should reach the Opcode register (instruction Opcode).
4. The low bits of the memory data input should be wired in a such way that only instruction STAM can write to the memory.
5. The high bits of the memory data input should be left unconnected.

The low bits of the memory output should be connected to the holding register input, but the holding register input is already connected to the AU's output. How can we select the input of the holding register to be either connected to the memory output or the AU output? What component should we use to achieve this functionality?

The output of the AU needs to reach both the memory address port and the holding register. What module should we use to achieve this connectivity?

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

3.5 Instruction Decoding

Instruction opcodes will be fetched from memory and stored in the Opcode register. The Opcode tells the CPU what operations to perform. For the CPU to be able to execute our instructions, it needs one signal for each instruction.

To convert the 4-bit value in the Opcode register into unique signals, we need to use demultiplexers. Decoding means taking the opcode, which is one 4-bit signal, and using it to activate one signal for each instruction. The relation between the Opcode and the control signals of the instructions is shown in Table 2.

Opcode	LDAC	LDBC	LDAM	LDBM	STAM	ADD	SUB	LDAI
0000	**0001**	0000	0000	0000	0000	0000	0000	0000
0001	0000	**0001**	0000	0000	0000	0000	0000	0000
0010	0000	0000	**0001**	0000	0000	0000	0000	0000
0011	0000	0000	0000	**0001**	0000	0000	0000	0000
0100	0000	0000	0000	0000	**0001**	0000	0000	0000
0101	0000	0000	0000	0000	0000	**0001**	0000	0000
0110	0000	0000	0000	0000	0000	0000	**0001**	0000
0111	0000	0000	0000	0000	0000	0000	0000	**0001**

Table 2: Instruction control signals and codes.

We need eight signals for our eight instructions. Therefore, you need to decode the value form the Opcode register into eight different signals. This way, whenever an instruction is being executed, only the signal associated with this instruction will be high. In this scene, a signal is high if its least significant bit is high. Then, this signal can be used to set up the data path appropriately.

Think about splitting the 4-bit signal from the Opcode into two 2-bit signals (dcba -> xxdc , xxba). You need the outputs of two demultiplexers to form eight outputs. Control these demultiplexers with one part of the Opcode. Add another demultiplexer and connect its input to a constant 0001 and its control input to the other part of the Opcode. Connect the first two outputs of this demultiplexer to the inputs of the first two demultiplexers.

Test your instruction decoding design to see if it matches the behaviour in Table 2.

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

connect ba to control the first dmux, and dc to control other 2 dmuxs.

4 Extra tasks

These are the two design problems posed in the lecture. Here you can test your design. Please only attempt these tasks when you have done all other sections.

4.1 Transfer data between memories without a change

You have two memory components. Design a circuit to copy the content of the first 16 bytes from the first memory to the first 16 bytes of the second memory.

4.2 Transfer data between memories with a change

Extra challenge. Design a circuit to move the data from the first memory to the second memory. Treat each four bits of the data separately. When moving the data, if the the value of any four bits (low or high) is equal to zero, then move 0xf instead of zero. Otherwise, copy the data as it is.

For example:

- 0x02 should be stored in the second memory as 0xf2
- 0x13 should be stored in the second memory as 0x13
- 0x60 should be stored in the second memory as 0x6f
- 0x00 should be stored in the second memory as 0xff

Hint: you might want to use the AU component in comparison mode. For more information about this mode, check the description of the ModuleSim components in Blackboard.

Well done for completing the first Simple CPU lab.