CHIPXPRT

Module ID : CX-202
Problem Solving Skills using C
Instructor : Dr. Abid Rafique

Project : Design of Assembler and Simulator for K2 Microprocessor

Version 1.1

Document History

The changes and versions of the document are outlined below:

Version	State / Changes	Date	Author
1.0	Initial Draft	September, 2023	Dr. Abid Rafique
1.1	Modifications	October, 2024	Qamar Moavia

Table of Contents

1 Administrivia

1.1 Learning Objectives

This project will enable you to,

- Understand basic computer architecture
- Understand the basic principle of computing
- Understand how computer processors are built with logic elements
- Understand how assembler works
- Understand how microprocessor simulator can be developed in C
- Understand bare metal programming stack

1.2 Introduction

In this project, you will be working in groups of two. Where, groups will be assigned by the instructors. This project involves

- Understanding the ISA for 4-bit K2 Microprocessor
- Understanding the micro-architecture of the K2 Microprocessor
- Development of Assembler for the K2 ISA
- Development of Simulator for K2 Microprocessor
- Utilizing a Git repository for version control
- Providing a Makefile for building the C code

2 K2 Microprocessor Instruction Set Architecture (ISA)

There are 9 functions we can perform with this machine. The functions and the set of control signals for them are,

Function	Remarks
$R_A = R_A + R_B$	Addition of values in R_A and R_B registers and result to be stored in R_A
$R_B = R_A + R_B$	Addition of values in R_A and R_B registers and result to be stored in R_B
$R_A = R_A - R_B$	Subtraction of value in R_B from R_A and result to be stored in R_A
$R_B = R_A - R_B$	Subtraction of value in R_B from R_A and result to be stored in R_B
$R_O = R_A$	Move the data from R_A to R_O
$R_A = imm$	Load the *imm* value in R_A
$R_B = imm$	Load the *imm* value in R_B
JC = imm	Jump to the address pointed by *imm* if there is carry out from previous operation
J = imm	Unconditional jump to *imm*

3 Micro-architecture of K2 Microprocessor

The design of the K2 Microprocessor is divided into three parts: Arithmetic Logic Unit (ALU), Data Registers and Control Logic.

3.1 Arithmetic Logic Unit

The Arithmetic Logic Unit (ALU) is the most basic and important part of any computing machine. All the arithmetic or logical operations are performed through it. The ALU used in this project will be able to perform binary addition as well as 2's complement binary subtraction.

3.2 Combinational Logic

The circuit requirement is to add or subtract two 4-bit numbers and generate a carry. In order to choose between add and subtract operations, we will be using a selection bit. The boolean equation for such a circuit can be realized as,

$$Out = (A + B)S^0 \ OR \ (A - B)S$$

Here "*A*" and "*B*" are the two inputs, "*Out*" is the output and "*S*",the selection bit.

Subtraction will be carried out by inverting the bits (i.e. taking 1's complement) of *B* and raising the "carry in" of adder to logic 1, in order to add an extra bit which will eventually generate 2's complement of *B*.

$$A - B = A + (1^0 s \ complement \ of \ B) + 1(carry \ in)$$

$$A - B = A + (2^0 s \ complement \ of \ B)$$

For inverting the bits we will use XOR gate with one input tied to the selection bit *S*, and the other to the input bit, such that when selection bit goes 1, the property of XOR, $B \oplus 1 = B^0$ can be used and when it is 0 the input passes unaffected $B \oplus 0 = B$.

3.3 Circuit Assembly

The block diagram of ALU is shown in Figure 1.

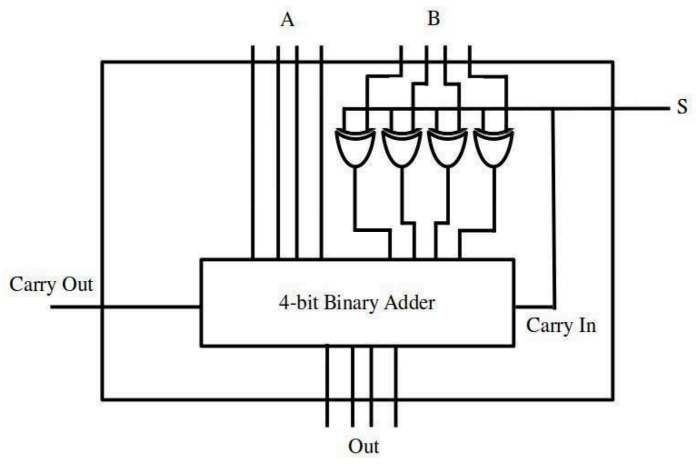


Figure 1: Arithmetic Logic Unit (ALU)

Selection bit (S)	**Function**
0	*Out = A + B*
1	*Out = A - B*

3.4 Data Registers

At least two 4-bit registers are required to hold the data for ALU. The output of these two registers, say R_A and R_B, are directly connected to the two inputs of the ALU, *A* and *B* respectively. The output of these registers is always enabled i.e. they are always channeling data into the ALU. However the input to these registers is controlled and data can only enter into them when the input enable bit of R_A and R_B is at logic 1.

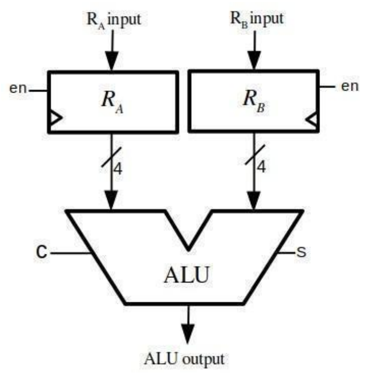


Figure 2: Register-ALU Assembly

3.5 Functionality and Time Synchronization

To start the computation, we need to load some initial values to R_A and R_B. Moreover, we would also like to utilize these registers to save the output of ALU too. In order to achieve this dual functionality, we need to add a 4-bit 1-out-of-2 data MUX before the input of registers. One of the MUX inputs would be connected to ALU's output and the other one to custom input. The selection bit of this MUX, say S_{reg}, must be 0 to let the ALU's data pass through and 1 for custom value. The updated datapath is shown in Figure 3.

A small D flip-flop is also placed adjacent to ALU to store the carry bit into it on the positive edge of the clock so that we can examine the status of the arithmetic operation performed by ALU.

Here the concept of clock is important. We know that registers are made of flip-flops that only store data on the positive edge of the clock (assuming the enable signal is 1, otherwise clock edges are in-effective). The small triangle on R_A and R_B in Figure 3 symbolizes input clock.

Let's see an example. Suppose through custom input (in the past), 2 was stored in R_A and 3 in R_B. Now, enable of R_A is 1, enable of R_B is 0, selection bit (S) of ALU is 0 and S_{reg} is also 0. At the output of ALU, the sum 5 is present. As soon as the positive edge of the clock comes, 5 gets stored in R_A (as its enable pin was high) and appears at the output of R_A, the output of ALU becomes 8. But due to "internal gate delays", 8 appears a few nano-seconds after the positive edge has passed and now it cannot enter any register until the next positive edge arrives. We can turn down the enable pins of registers to zero, to make clock edges in-effective so that the incoming 8 may not replace previously stored 5.

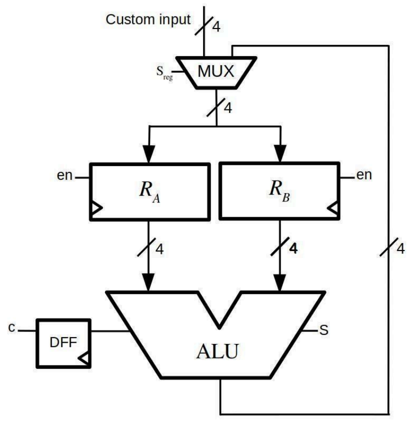


Figure 3: Modified Register-ALU Assembly

3.6 Testing

Let's test our design with a small "swapping" example. Usually when we swap the values of two variables, we use another temporary variable to hold data for sometime,

```
Initial state: $x = a, y = b$
{
  $temp = x$
   $x = y$
  $y = temp$
}
Final state: $x = b, y = a$
```

However, there is another smart algorithm to swap data of two registers without using any third register,

```
Initial state: $x = a, y = b$
{
  $x = x + y$
  $y = x - y$
  $x = x - y$
}
Final State: $x = b, y = a$
```

There are four control signals (S_{reg}, en_{RA}, en_{RB} *and* S) and the 4-bit custom input in our hands. First we shall load "a" to R_A and "b" to R_B, after which we will execute the algorithm. Here "a" and "b" are the 4-bit custom inputs.

Function	Control(S_{reg} en_{RA} en_{RB} S)	Result(atposedgeclk)
Load a to R_A	110 x	a stored in R_A
Load b to R_B	101 x	b stored in R_B
$R_A = R_A + R_B$	0100	a+b stored in R_A
$R_B = R_A - R_B$	0011	a stored inR_B
$R_A = R_A - R_B$	0101	b b stored R_A

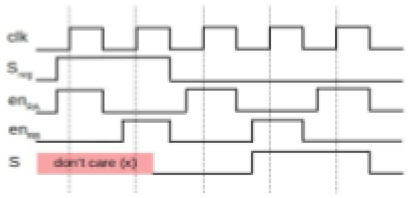
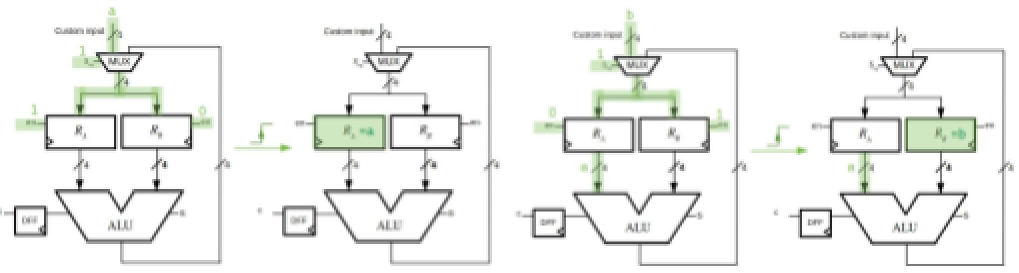
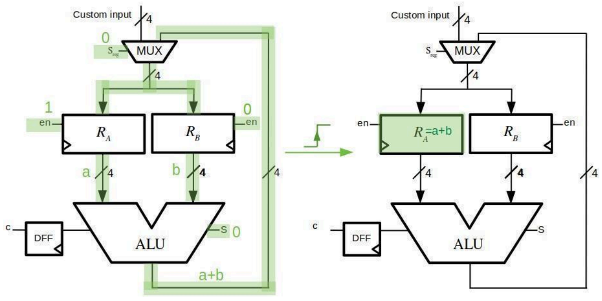


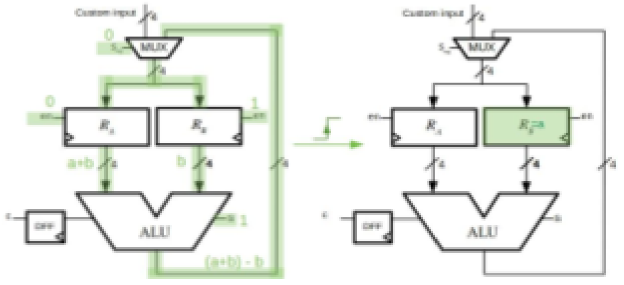
Figure 4: "Swapping" Control Signals Timing diagram



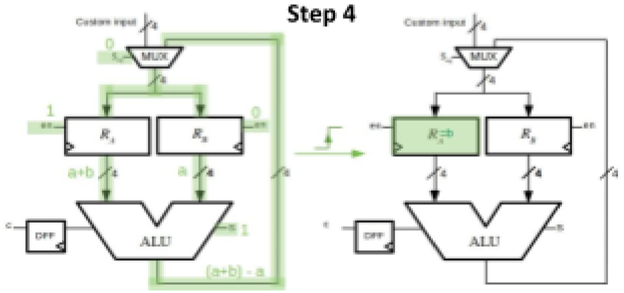
Step 1 Step 2



Step 3



Step 4



Step 5

3.7 Output Register

A small addition to our design, the output register R_O. We shall use this register to store the final result after all the processing for the user to refer. The register is directly connected with R_A.

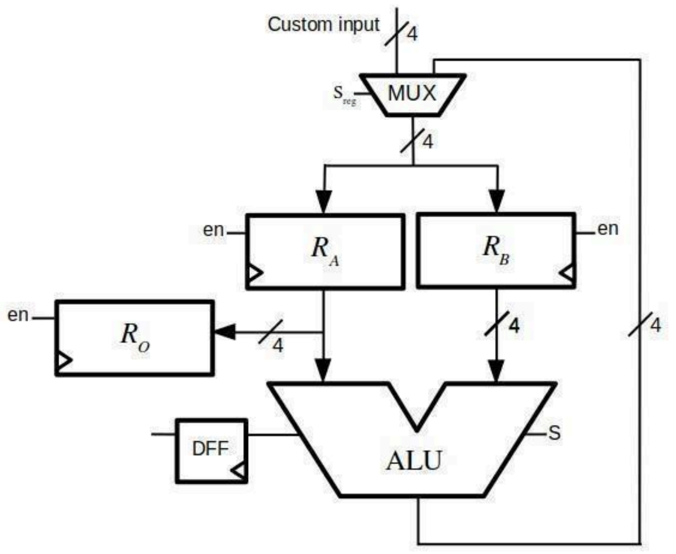


Figure 5: Output Register

3.8 Register De-multiplexing

There are now three signals associated with the registers, en_A, en_B and en_O. In order to reduce these signals we will use a 2-to-4 decoder (two inputs and four outputs) with a truth table shown below,

Input (D_0, D_1)	**Output** (O_0, O_1, O_2, O_3)
00	1000
01	0100
10	0010
11	0001

Now the outputs of decoder are connected such that O_0 to en_A, O_1 is to en_B and O_2 to en_O

Input (D_0,D_1)	**Output**
00	R_A input enabled
01	R_B input enabled
10	R_O input enabled
11	no operation

Keep in mind, from now onward we will call the value of D_0D_1, "**address**" of the corresponding register.

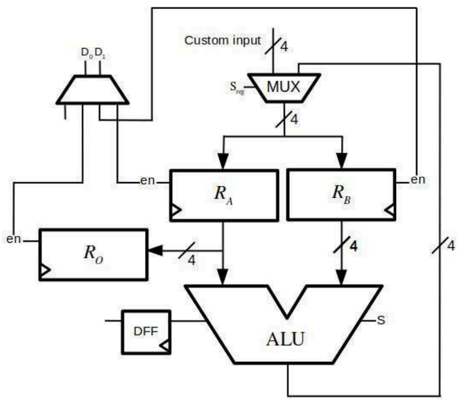


Figure 6: Design with Decoder Incorporated

3.9 Control Logic

We cannot apply logic values on the control signals manually, all the time. In order to avoid this inconvenience, we can store the values of control signals in a non-volatile memory (ROM or EEPROM) and connect its output lines to our design. Let's call the values of the control signal stored in the memory as Instructions.

3.10 Instruction Memory

The instruction memory comprises machine instructions in the form of decoded signals (See Section 3.13).

3.11 Program Counter

In order to set the signals (or instruction) on the control lines of our design before the positive edge of clock arrives, we will be using a 4-bit counter connected to the address line of instruction memory, such that initially the value of the counter is zero with zeroth row activated. As soon as the positive edge of the clock comes, zeroth instruction is executed by our design as well as the counter gets incremented. Now the value of the counter is one and row 1 of instruction memory is activated. The incoming positive edge will execute this instruction and the process goes on.

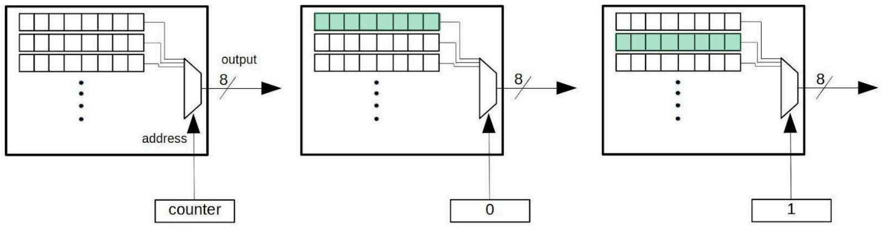


Figure 7: Memory and Program Counter

The program counter must have a 4-bit input and a load control such that when some custom input is applied to 4-bit input of counter and load control is kept high on the arrival of the positive edge, the counter stores that custom input into it and start counting from that specific value on the next edge of the clock. We wish to design a combinational logic around this feature of the counter.

$$Load = J \quad OR \quad (C \quad AND \quad \text{“}Carry\ out^{00})$$

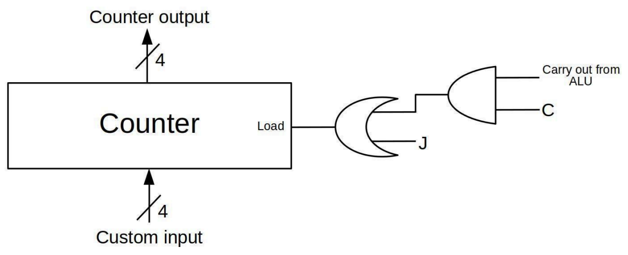


Figure 8: Jump Logic

Here “J” and “C” are control signals in our hand such that when we apply some custom input to the counter and raise J to 1, the counter starts counting from the custom input. Similarly, when the carry out from ALU is 1 and we raise C to 1, the counter starts counting from custom input. This phenomenon is known as **“JUMP”** or **“BRANCH”** in computing language.

3.12 Instruction Memory and Instruction Encoding

Since the rows of instruction memory are 8-bits wide, we wish to adjust all our control signals to these 8-bits. Moreover, our custom input is also supposed to be a part of this instruction. One scheme to adjust all these signals is,

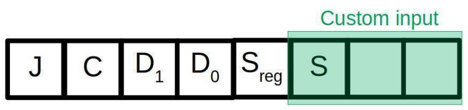


Figure 9: 8-bit wide instruction

1. **Jump (J):** The seventh bit or MSB of the instruction is fixed for Jump. Whenever we set it to 1, the custom input will be loaded in the program counter.

2. **Carry jump (C):** The sixth bit is fixed for “jump if ALU operation produces a carry”. Whenever we set it to 1, the custom input will be loaded in the program counter.

3. **Register Address (D_1D_0):** The fifth and fourth bit of instruction is fixed for register address. The values 00, 01, 10 and 11 corresponds to R_A, R_B, R_O and no register, respectively.

4. **ALU or Custom input (S_{reg}):** The third bit is reserved for multiplexing ALU or custom input. When it is 0, ALU's output is directed to registers' input otherwise, the custom input.

5. **Custom input (immediate):** The last three bits, zeroth, first and second are reserved for custom input. Due to adjustment problem, we have to compromise custom input length (to 3-bits only). The fourth bit is hard wired to zero (ground), which means, we can only load values ranging from 0 to 7 (either in program counter or registers).

6. **ADD or SUB (S):** The second bit of the instruction has dual function. Apart from being part of custom input, it is also connected to ALU selection bit S. This decision is taken based on the observation that when ALU is performing either addition or subtraction, there is no interference of custom input. Similarly, when we are loading some custom input to either registers or counters, we don't care about ALU processing.

3.13 Microprogrammed Controller

There are 9 functions we can perform with this machine. The functions and the set of control signals for them are,

Function	J	C	D_1	D_0	S_{reg}	S		
						Custom Input		
$R_A = R_A + R_B$	0	0	0	0	0	0	0	0
$R_B = R_A + R_B$	0	0	0	1	0	0	0	0
$R_A = R_A - R_B$	0	0	0	0	0	1	0	0
$R_B = R_A - R_B$	0	0	0	1	0	1	0	0
$R_O = R_A$	0	0	1	0	0	0	0	0
$R_A = imm$	0	0	0	0	1	imm[2]	imm[1]	imm[0]
$R_B = imm$	0	0	0	1	1	imm[2]	imm[1]	imm[0]
JC = imm	0	1	1	1	0	imm[2]	imm[1]	imm[0]
J = imm	1	0	1	1	0	imm[2]	imm[1]	imm[0]

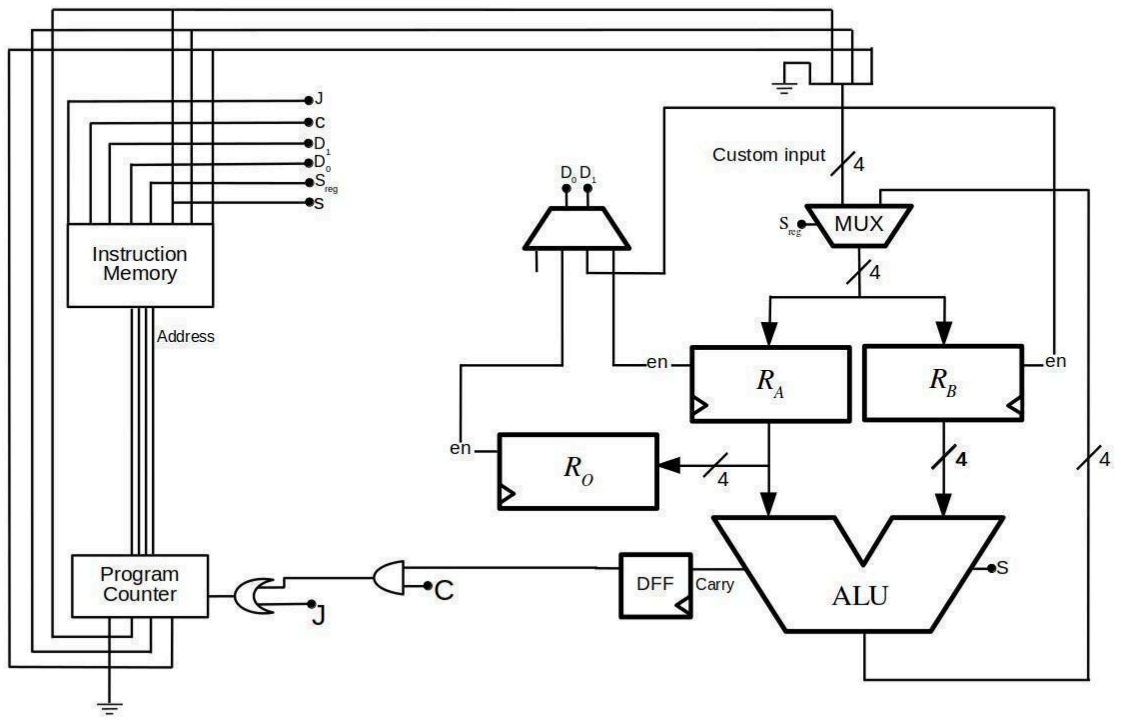


Figure 10: 4-bit K2 Microprocessor

4 Tasks

4.1 K2 Assembler

- Design an Assembler for K2 Microprocessor
- Upon starting, the assembler should take the input assembly filename as a **command line argument** (e.g., **fibonacci.asm**)
- The assembler should take the specified **.asm** file written in K2 Assembly Language as a text file (e.g. a K2 program for Fibonacci numbers generation is shown in Section 5 will be stored as **fibonacci.asm**).
- The assembler then should convert the specified K2 Assembly Program into machine code based on the table shown in Section 3.13, and store the output in a **.bin** file with the same name (e.g., **fibonacci.bin**)
- The assembler should clearly generate error messages or warning messages for the assembly instructions that are not written in the correct format according to the K2 instruction set.

4.2 K2 Simulator

- Write the K2 Microprocessor simulator shown in Figure 10.
- The complete datapath should be modeled in C (*e.g.* multiplexers can be modeled as function **int mux2_1(int iA, int iB, bool bSel**) - Same goes for other building blocks such as decoder, ALU, program counter and Instruction Memory.
- The simulator should take the filename input as a command line argument, allowing the user to load any machine code (**.bin**) file into memory.
- There should be multiple selections available to the user using keys such as
    - **R -** Run the Simulator in continuous mode

    - **S -** Run the Simulator step-by-step, executing one instruction per user input (for line-by-line execution)
- The simulator should print the values from the **R0** register, whenever any value is written to the **R0** register.

4.3 Git Repository and Makefile

The project should be managed using Git. Students must first create a Git repository and push all their work to the repository, maintaining a complete commit history throughout the project.

The project will be completed in groups of two students, with each student working from their own branch. Changes should be merged into the main branch only after review and approval by the other group member.

A **Makefile** should be included with the project, allowing the compilation of C code for both the assembler and the simulator. Main targets in the **Makefile** should include:

- **assemble** to run the assembler executable file. Incase the **assembler. c** is modified, only then compile the assembler program (**assembler.c**) first and then run the assembler executable. The assembler should be executed with the input file specified as **FILENAME=fibonacci.asm**, which the C code will read as a command line argument.
- **simulate** to compile the simulator program (**simulator.c**). The simulator should take the binary filename as a command line argument (e.g., **FILENAME=fibonacci.bin**).
- **all** to compile both programs.
- **clean** to remove any compiled files.
- **help** to display help about use of Makefile with it's targets

The C code for both the assembler and the simulator should read the input filename directly from the command line arguments provided via the **Makefile**. For example, executing **make assemble FILENAME=fibonacci.asm** will pass **fibonacci.asm** as an argument to the assembler program, and similarly for the simulator, using **make simulate FILENAME=fibonacci.bin** will pass **fibonacci.bin** to the simulator executable program.

5 Application

5.1 Fibonacci Numbers

Below is a Program that will produce a Fibonacci sequence in the output register.

0	$R_A = 0$	00001000	load zero to A
1	$R_B = 1$	00011001	load 1 to B
2	$R_O = R_A$	00100000	push A to output
3	$R_B = R_A + R_B$	00010000	Add A to B
4	JC = 0	01110000	If A+B produce carry jump to start
5	$R_A = R_A + R_B$	00000000	Swap A and B
6	$R_B = R_A - R_B$	00010100	Swap A and B
7	$R_A = R_A - R_B$	00000100	Swap A and B
8	J = 2	10110010	Jump to 3rd instruction

The above program should be stored as fibonacci.asm.

5.2 Sample Run

Here is the sample run of the assembler and then simulator

```
# 1. Run Assembler for fibonacci.asm

$ make assemble FILENAME=fibonacci.asm

# Expected Output (Assembler)
Starting Assembler...
Reading file: fibonacci.asm
Line 1: RA=0 -> Machine Code: 00001000
Line 2: RB=1 -> Machine Code: 00011001
Line 3: RO=RA -> Machine Code: 00100000
Line 4: RB=RA+RB -> Machine Code: 00010000
Line 5: JC=0 -> Machine Code: 01110000
Line 6: RA=RA+RB -> Machine Code: 00000000
Line 7: RB=RA-RB -> Machine Code: 00010100
Line 8: RA=RA-RB -> Machine Code: 00000100
Line 9: J=2 -> Machine Code: 10110010
Successfully generated output file: fibonacci.bin

# 2. Run Simulator for fibonacci.bin

$ make simulate FILENAME=fibonacci.bin

# Simulator Prompt
Select one of the following mode
R - Run in continuous mode
S - Run step-by-step
Select mode:S # Choose Step-by-Step Mode (S)


# Expected Output (Step-by-Step Execution with RO printed only on RO=RA):
Loading binary file: fibonacci.bin
Starting Simulator in step-by-step mode...

Instruction 0: RA=0              [Press Enter to continue]
Instruction 1: RB=1              [Press Enter to continue]
Instruction  2:  RO=RA  ->  RO=0      [Press  Enter  to  continue]   #  Printed
because RO=RA
Instruction 3: RB=RA+RB          [Press Enter to continue]
Instruction 4: JC=0 (No Jump)    [Press Enter to continue]
Instruction 5: RA=RA+RB          [Press Enter to continue]
Instruction 6: RB=RA-RB          [Press Enter to continue]
Instruction 7: RA=RA-RB          [Press Enter to continue]
Instruction 8: J=2 (Jump to Instruction 2)

... (continues until the end of the sequence based on the program, with RO
printed only when RO=RA is executed)
```

# Choose Continuous Mode (Fast Execution)

```
Select one of the following mode
R - Run in continuous mode
S - Run step-by-step
Select mode:R
```

# Expected Output (Continuous Mode Execution with RO updates only when RO=RA):

```
Loading binary file: fibonacci.bin
Starting Simulator in continuous mode...

Execution (Register RO output):
RO=0  # Only displayed when RO=RA instruction is executed
RO=1
RO=1
RO=2
RO=3
RO=5
RO=8
RO=13
R0=0 # Note that sequence will restart after 13, as register size is 4 bit
R0=1
R0=1
R0=2
...
```

# Output (Fibonacci sequence in RO): 0, 1, 1, 2, 3, 5, 8, 13, 0,

6 Concluding Remarks

The design of the K2 Microprocessor is presented in this document. The student is asked to understand the ISA, the micro-architecture, leverage the concepts studied in C Programming Module and develop an Assembler as well as a Simulator for K2 Microprocessor. You can implement some other interesting programs like multiplying two numbers or generating tables of 1, 2, 3, …, etc.

In short, if you are able to successfully complete this project and can extend or adapt it according to your need, you have gained a strong hold on the basics of Processors, writing an Assembler and modeling any complex system using C.

Good Luck 🙂