ECE352 Computer Organization II
Department of Electrical and Computer Engineering
University of Toronto

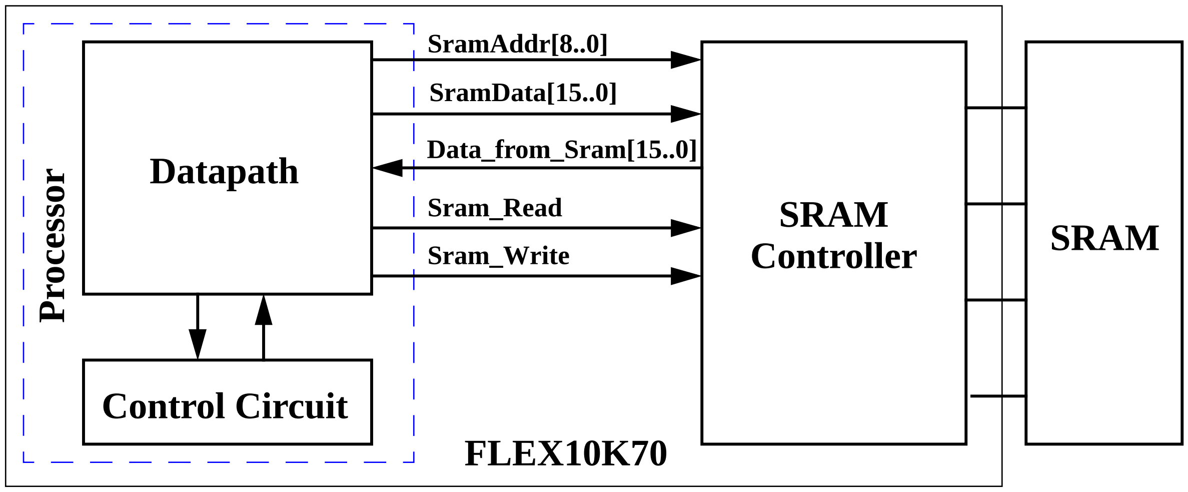
# Project: Simple Processor Implementation

In this lab, you will implement a simple 16-bit processor in the FLEX10k70 FPGA located on the Ultragizmo board. The processor will be able to execute a limited set of instructions which will be stored in the SRAM also located on the Ultragizmo board. In addition to designing the circuitry for the processor, you will also need to design an SRAM controller to access the SRAM.

Your design should have two modes of operation. In the first mode of operation, 68000 accesses to the SRAM are enabled. This mode enables the user to store instructions to the SRAM (that the simple processor will execute) and read data from the SRAM (that the simple processor has modified). In this mode, the simple processor will be idle. The second mode will disable 68000 accesses and start the simple processor. In this mode, the simple processor can access the SRAM. Use one of the switches on the protoboard to set the mode ('0'- m68k, '1'- processor). Call this signal **m68k_master**.

### High Level Organization

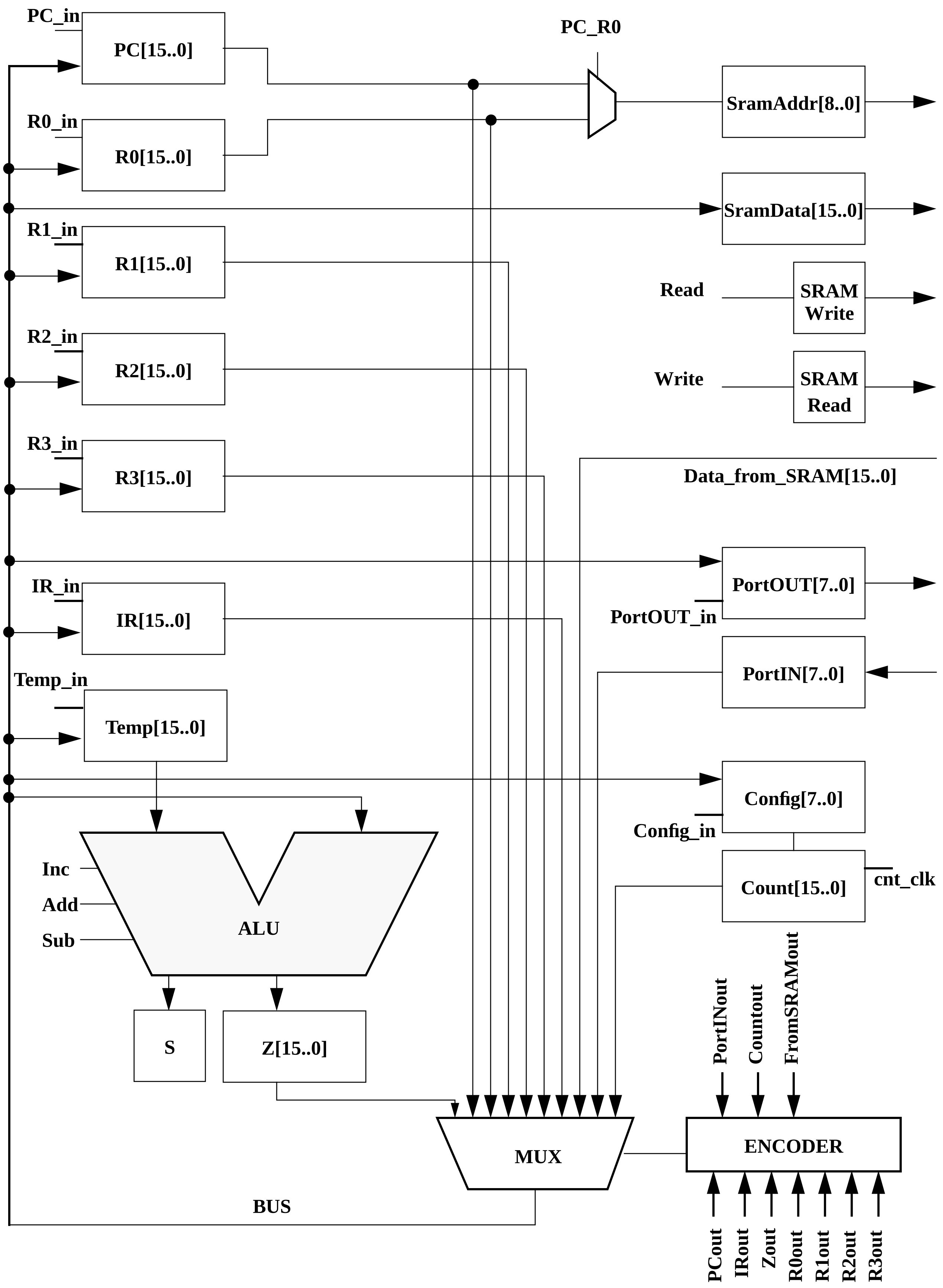
The figure below shows how the processor interacts with the outside world. The processor is able to address 512 words of the SRAM (256 for code, 256 for user data). You should implement the datapath such that all instructions are located in the address space from 0-256. Any data address **A** referenced by a **load/store** instruction should be mapped to **256+A**. This prevents the user from accidently overwriting their code.



Data/Instruction reads are accomplished by asserting the **Sram_Read** signal and latching the value on the **Data_from_Sram** bus after a sufficiently long delay. Data writes are accomplished by asserting the **Sram_Write** signal with the data value on the **SramData** bus.

## 1. The Simple Processor

A block diagram of the simple processor you are to implement is given on the following page.


PC_in
PC[15..0]
PC_R0
SramAddr[8..0]
R0_in
R0[15..0]
SramData[15..0]
R1_in
R1[15..0]
Read
SRAM Write
R2_in
R2[15..0]
Write
SRAM Read
R3_in
R3[15..0]
Data_from_SRAM[15..0]
PortOUT[7..0]
IR_in
IR[15..0]
PortOUT_in
PortIN[7..0]
Temp_in
Temp[15..0]
Config[7..0]
Config_in
Count[15..0]
cnt_clk
Inc
Add
Sub
ALU
S
Z[15..0]
PortINout
Countout
FromSRAMout
MUX
ENCODER
BUS
PCout
IRout
Zout
R0out
R1out
R2out
R3out

A brief description of some of the components in the processor is given in the table. Although not indicated in the block diagram, each register is connected to the same clock signal and global reset. Also note that registers that are connected to the bus have an input enable called *register_name*$_{in}$. It controls when the value on the bus is stored in the register.

| Component | Description |
|---|---|
| **R0, R1, R2, R3** | Four general purpose registers that can be used for simple computations. |
| **PC** | The *program counter* holds the address of the next instruction to be fetched from memory. Upon fetching the instruction, the program counter is incremented by the ALU so that it points to the next instruction. The program counter should be initialized to ‘0’ on start-up. |
| **IR** | The *instruction register* holds the instruction brought in from memory. This allows the processor to read other locations from memory without loss of the current instruction. |
| **ALU** | A simple ALU which can add, subtract and increment the value at its inputs. |
| **Temp** | A register that holds an intermediate or temporary value. It is useful when the processor needs to execute an addition or subtraction operation. |
| **Z** | A register that holds the output value from the ALU. Again this plays an important role in any arithmetic operation. |
| **S** | A register to hold a status bit called “zero”. It indicates whether the result of a subtraction is equal to zero. |
| **PortOUT[7..0]** | A register used for I/O whose outputs are connected to pins on the processor. It is used to write data to external devices (e.g. LEDs). |
| **PortIN[7..0]** | A register used for I/O whose inputs are connected to pins on the processor. It is used to read data from external devices (e.g. switches). |
| **Count** | A 16-bit counter which can be used to time events. Note that the counter is connected to an external clock, *cnt_clock*. |
| **Config [7..0]** | A register to hold the configuration bits for the counter. Currently only one bit will be used to enable the counter. |
| **MUX - Encoder** | Allows data on bus according to which signal is asserted on the encoder. |

The SRAM interface in the block diagram was not included in the table. It is worth mentioning that all signals that are output to the SRAM memory are registered. Providing combinational signals as outputs could cause glitches on these signals which in turn could cause corruption of stored data.

The **SramAddr** is generated by a multiplexer that selects between the values of **PC** and **R0**. This functionality is needed because the SRAM address can either be provided by the **PC** on instruction fetches or by register **R0** on data accesses.

The **SRAM_Read** and **SRAM_Write** signals are generated registered versions of the **Read** and **Write** signals asserted by your control circuit. The **SramData** signals are registered values of the bus lines. Thus data is constantly being sent to the SRAM memory, however the **SRAM_Write** signal must be asserted for memory values to actually be changed.

Omitted from the block diagram is the control circuit for the processor. It will be described in more detail after the discussion on instructions.

## Instruction Format

All instructions are stored in the following format:

| OP-CODE | X | Y | DATA |
|---|---|---|---|

- **OP-CODE** is a 4-bit field that can describe up to 16 instructions. Only 7 instructions will be needed for this lab.
- **X** is a 2-bit field that indicates the destination register for the operation. For example when X is set to “01” then the destination refers to **R1**.
- **Y** is a 2-bit field that indicates the source register for the operation.
- **DATA** is an 8-bit field used for holding constant values for corresponding to the operation. This field may not be needed by all operations. In these cases, it may be set to any value.

## Instructions

| NAME | OP-CODE | ACTION |
|---|---|---|
| movi | 0000 | $R_X \leftarrow DATA$ |
| move | 0001 | $R_X \leftarrow R_Y$ |
| load | 0010 | $R_X \leftarrow (R_0)$<br>Load the contents of memory location $R_0$ into $R_X$ . |
| store | 0011 | $(R_0) \leftarrow R_Y$ |
| add | 0100 | $R_X \leftarrow R_X + R_Y$ |
| sub | 0101 | $R_X \leftarrow R_X - R_Y$ |
| halt | 0110 | **Stop executing**. Do no fetch any more instructions. |

| NAME | OP-CODE | ACTION |
|---|---|---|
| bne | 0111 | $PC \leftarrow DATA$ if S≠0 |
| mvin | 1000 | $R_X \leftarrow PortIn$ |
| mvout | 1001 | $PortOUT \leftarrow R_Y$ |
| mvcnt | 1010 | $R_X \leftarrow Count$ |
| mvcfg | 1011 | $Config \leftarrow R_Y$ |

**Control Circuitry**

You are responsible for creating control circuitry that works together with the datapath to implement the simple instruction set given. Your control circuit should be implemented as a state machine which controls the input enable signals to the registers, the inputs to the encoder, PC_R0, Read, Write, Inc, Add and Sub. You should begin by drawing a state diagram that details the sequence of control steps needed to implement each instruction.

To start, consider that you will need the 4 steps at the beginning of every instruction to fetch the current instruction from memory, place it in the instruction register (**IR**), and increment the program counter to the next instruction location.

| | |
|---|---|
| T1 | $PCR0 = PC, SRAMout, Read$ |
| T2 | $SRAMout, IRin$ |
| T3 | $PCout, Increment$ |
| T4 | $Zout, PCin$ |

Your control circuit should also use the **OP-CODE**, **X**, and **Y** fields of the instruction register as inputs, so that it can assert the appropriate outputs depending on the instruction fetched from memory.

## 2. Project Milestones

The project will last for 3 lab periods. At the end of each of the three lab periods you will be marked based on the following weekly goals.

### Week 1

Implement the simple processor with the following instructions: *movi*, *move*, *load*, *store*, *add*, *sub* and *halt*. You are not required to implement the counter or the I/O port registers for now. For this first part you also do not need to connect your processor to the SRAM on the Ultragizmo board. To make things simpler initially, it is recommended you use the RAM on the FLEX10K70. This way you can fully test your processor in the simulator. A template for the processor and the top level design file is provided on the webpage. Also, a file called test.mif is provided which will show you how to initialize the RAM in the FLEX10K70.

**Week 2**

Add the counter, configuration register, I/O port registers and the remaining instructions to your implementation. Write some test code (a test.mif file) to exercise each instruction.

**Week 3**

Connect your processor to the SRAM. Add an input signal called **m68k_master** to your control circuit. Your processor should wait until this signal goes low before attempting to fetch and execute any instructions. Note that this is the same signal discussed in the introduction of the lab, which also serves to multiplex address and control lines to the SRAM.

After you have connected your simple processor to the SRAM, you will need to be able to write instructions to the SRAM for your processor to execute. Instead of entering each instruction with monitor commands, download an “assembly program” to the SRAM (just as you would any other assembly program to memory). The assembly program should only contain an *org* statement and *DC directives*.

Write machine code for a reaction timer to be executed on your processor implementation. Demonstrate that it works.