55:035 Spring 2013
Computer Architecture and Organization
Simple Instruction Set Computer (SISC) Project

---

1. General Information:

The project consists on building and extending a Simple Instruction Set Computer (SISC) processor using Verilog HDL, and writing machine code programs that correctly execute on this computer. The code to be used for this project is provided in a zip file in ICON. The project is divided into 3 parts described below.

a) Overview of the SISCver3 Processor Architecture

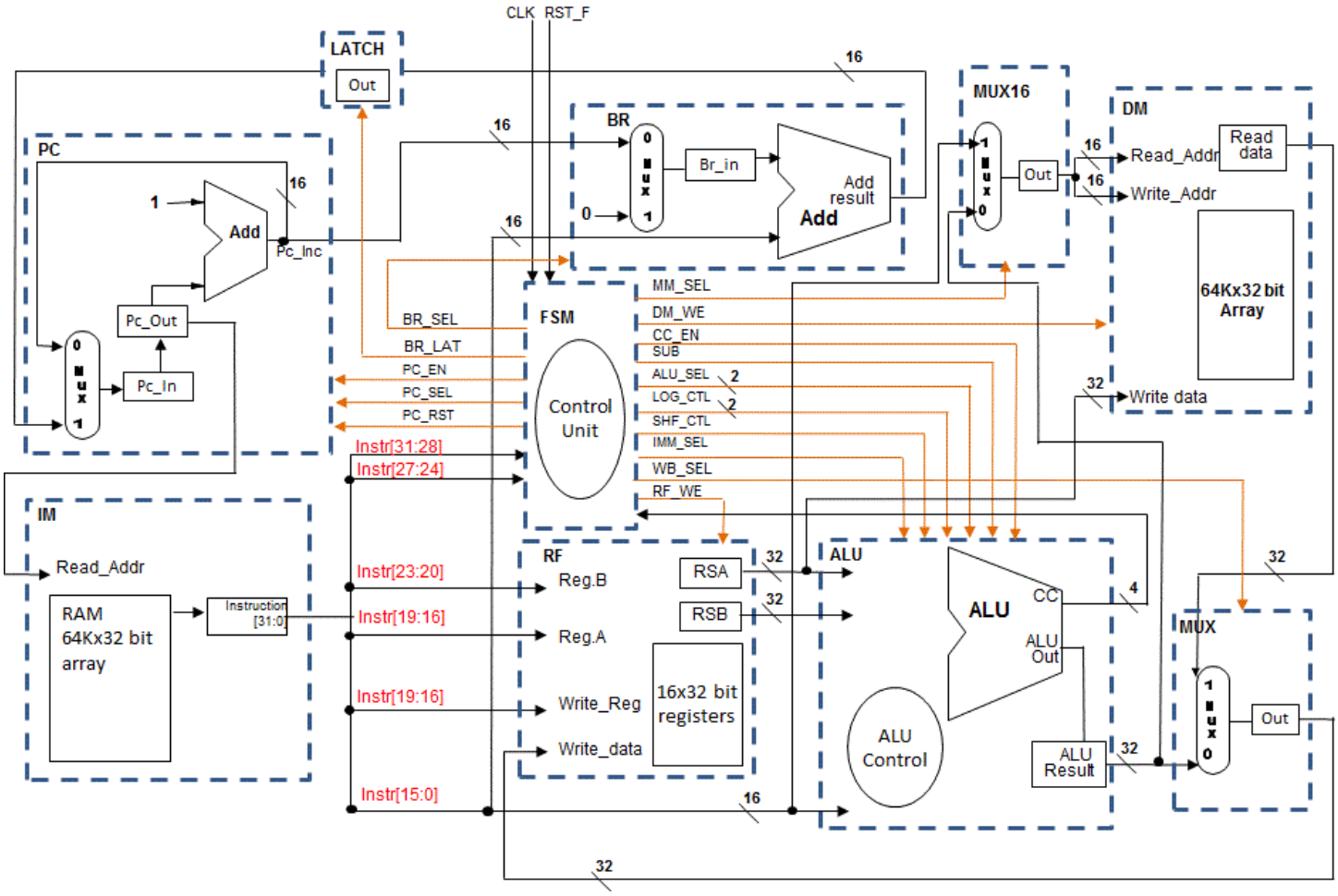
This Simple Instruction Set Computer (SISC) is a multi-cycle RISC computer with separate memory for instructions and data, with the following characteristics:

word length:	32 bit
general purpose registers:	16 x 32 bit
Instruction/data space:	2^16 = 65536 words = 64 KW
addressing resolution:	word
instruction set:	LOAD/STORE-architecture
immediate operand lengths:	16 bit
addressing modes:	immediate
	register-direct
	register-indirect
	index
	absolute

Status register is 4 bits:

3	2	1	0
Even	**Overflow**	**Negative**	**Zero**

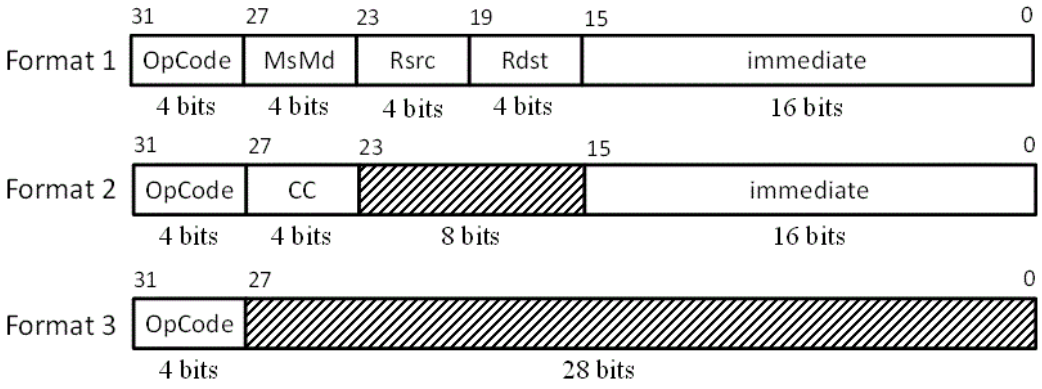
b) Datapath



Dotted boxes represent a .v file in the provided code and correspond to a verilog module. The name of the module is the label in the top left corner of the box. Control lines are highlighted in orange. The sisc.v file implements the datapath by connecting all the components together (part 1).

c) Instruction set

The SISC instruction has three instruction formats:



And support the following instructions:

- Load Register (from indexed (indirect) register or absolute to direct register) – LOD
- Store Register (to indexed (indirect) register or absolute from direct register) – STR
- Addition (Rdst = Rdst + Rsrc) (Rdst = Rsrc + {4hx0000,immediate}) – ADD
- Subtraction (Rdst = Rdst - Rsrc) (Rdst = Rsrc - {4hx0000,immediate}) – SUB
- Bitwise-AND (Rdst = Rdst /\ Rsrc) – AND
- Bitwise-OR (Rdst = Rdst \/ Rsrc) – OR
- Bitwise-XOR (Rdst = Rdst XOR Rsrc) – XOR
- Bitwise Invert (Rdst = ~ Rsrc) – NOT
- Logical Shift (Rdst = Rdst >> or << [Rsrc]) – SHL/SHR[1]
- Logical Rotate (Rdst = Rdst >-> or <-< [Rsrc]) – ROL/ROR[1]
- Branch Absolute (conditional/unconditional) (PC ← immediate) – BRA
- Branch Relative (conditional/unconditional) (PC ← PC ± immediate) – BRR
- No Operation – NOP
- Halt (Stops the processor) – HLT

[1] IR[0] = 1 ? left : right

Bits:	31:28	27:24	23:20	19:16	15:0
NOP	**0**	**x**	**x**	**x**	**xxxx**
BRA	**1**	***cc****	**x**	**x**	**immed**
LOD ind	**2**	**0**	**(R*src*)**	**R*dst***	**immed**
LOD abs	**2**	**8**	**x**	**R*dst***	**(immed)**
STR reg	**3**	**0**	**R*src***	**(R*dst*)**	**(immed)**
STR abs	**3**	**8**	**R*src***	**x**	**(immed)**
ADD	**4**	**0**	**R*src***	**R*dst***	**xxxx**
	4	**8**	**R*src***	**R*dst***	**immed**
SUB	**5**	**0**	**R*src***	**R*dst***	**xxxx**
	5	**8**	**R*src***	**R*dst***	**immed**
NOT	**B**	**0**	**R*src***	**R*dst***	**xxxx**
SHF	**7**	**0**	**R*src***	**R*dst***	**xxxL**
ROT	**8**	**0**	**R*src***	**R*dst***	**xxxL**
BBR	**D**	***cc****	**x**	**x**	**immed**
HLT	**F**	**x**	**x**	**x**	**xxxx**

* Condition codes are produced during the ADD and SUB instructions and used by the BRA and BRR instructions. ****cc*** is a mask field for the condition codes. When executing a BRA or BRR instruction, the mask bits are compared to the condition code results for the most recent ADD or SUB instruction. If

the condition code is true for any mask bit that is set high, the branch is taken. Otherwise the branch is not taken. The ***cc*** bits should be set as follows:

0 unconditional branch
1 branch on **zero bit set** (last add/sub result was zero)
2 branch on **negative bit set** (last add/sub result was negative)
4 branch on **overflow** (last add/sub result overflowed)
8 branch on **carry bit set** (last add/sub result generated a carry or borrow)

All operands are signed 32-bit values except the immediate values used in branch, load, and store instructions, which are signed 16-bit values. For add and sub immediate, the immediate value are zero extended.

Register 0 (R0) is always 0.

Some hints to write your programs: The pseudo instruction MOVE will translate into ADD Rx, R0, immed (Rx ← zero_ext(immed)). For negative numbers you can compute the 2's complement of the number by negating (NOT) and adding 1 to the register (ADD) or you can use SUB Rx, R0, immed to store the 2's complement of zero_ext(immed) into Rx.

As always, state your assumptions clearly.

Part 1: Implementing the Datapath. Due Apr. 15 by 11:59pm

- The SISC processor should be built using Verilog. Using the Xilinx ISE Webpack tool, create a new project using the source files provided for this project. Your project must implement the top-level CPU interface including the Clock and Reset signals. A baseline testbench for the project is included, modify it as you need to test your changes or individual instructions. Memory I/O for is initialized using two different files: one for the instruction memory (IM) module (**`"imem.data")`** which contains the instructions to be executed and one for the data memory (DM) module (**`"datamemory.data")`** that contains the values stored in memory. The content of the memory is expressed as hexadecimal numbers. _ are ignored and are just used to improve the readability of the code. Comments are preceded with //. The first non-commented line corresponds to address 0 and incremented by 1 with each line. .data files need to include a new line at the end of the file.

What you need to do:

a) Complete the top level module (sisc.v) to implement the datapath for this computer depicted in 1b). The code given in sisc.v is already connecting the PC, IM, and RF modules as examples. File sisc_tb.v is given as a baseline testbench. Write a testbench for all the instructions. Sample code and data are also provided in the imem.data and datamemory.data files.
b) Implement a program that computes (A × B) + (C × D) (as in problem 2 of homework 3) in machine code for this computer. Note that our computer does not support the Multiply

instruction. You should write your program using additions and branches. For this part you will replace the content of imem.data and datamemory.data for your own code and test data.

What to turn in:

- Submit to ICON all .v and .data files compressed in a zip folder.

Part 2: Due Apr. 29 by 11:59pm

For this task you are asked to modify the SISCver3 computer:

a) Add the instruction Branch on not equal (BNE) (PC ← #immed if the condition code is not set).

Bits:	31:28	27:24	23:20	19:16	15:0
BNE	**E**	***cc****	**X**	**X**	**Immed**

***** Just as with BRA and BRR the ****cc*** bits should be set as follows:

0 unconditional branch
1 branch on not **zero bit set** (last add/sub result was not zero)
2 branch on not **negative bit set** (last add/sub result was not negative)
4 branch on not **overflow** (last add/sub result did not overflow)
8 branch on not **carry bit set** (last add/sub result did not generate a carry or borrow)

b) Implement the assembly code given for Problem 2 in Homework 4 into machine code executable for the new SISC version using the branch on not equal instruction. Change the references to R0 in the code for R6 (as R0 is always 0 for our SISC). You will need to replace the content of imem.data and datamemory.data with your code and test data.

What to turn in:

- Submit to ICON all .v and .data files compressed in a zip folder.

Part 3: Due May 10 by 11:59pm

a) Add the autoincrement and immediate addressing modes for the LOAD instruction.

Bits:	31:28	27:24	23:20	19:16	15:0
LOAD autoinc	**2**	**4**	**Rsrc**	**R*dst***	**immed**

You can implement the autoincrement addressing mode making any changes you see fit to the SISC computer as long as all other instructions continue to execute correctly.

Here are some ideas to get you started: in addition to the normal load operation this instruction will increment Rsrc by 1 and write this value back to the register file. For increment the register you

could add an extra adder, use the already existing PC adder without overwriting PC, or use the ALU by adding a multiplexer that adds 1 as the second operand. This Load instruction will have to write two values to the register file, one for the register destination to store the data from memory, and the increment of the register Rsrc. You could do this in two different cycles and have multiplexers to select the write address and data to be written, or in the same cycle by modifying the register file to include two write addresses and data ports. You will need to add appropriate control lines.
After you are done, run the testbench from Part 1 on your new datapath to make sure that all other instructions still execute correctly. The program written for Part 1 should still execute correctly after these changes.

b) Implement the assembly code given for Problem 1 in Homework 3 into machine code executable for the new SISC version using the auto increment mode. You will need to replace the content of imem.data and datamemory.data with your code and test data.

What to turn in:

- Submit to ICON all .v and .data files compressed in a zip folder.