KGPRISC (Single Cycle CPU)

User Guide

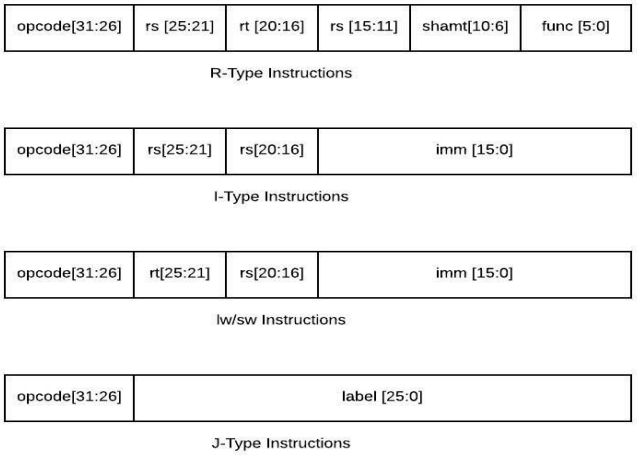
- Nikhil Nayan Jha (16CS30022)
- Sarthak Chakraborty (16CS30044)

GROUP 32

The following document is a guide for the Single Cycle CPU (RISC architecture) named KGPRISC that we designed. This document lists the Instruction Set Architecture (ISA) that the processor supports, format for instructions, instruction encoding and the architecture design.

Class	Instruction	Usage	Meaning
Arithmetic	Add	add rs,rt	$rs \leftarrow (rs) + (rt)$
	Comp	comp rs,rt	$rs \leftarrow$ 2's Complement (rt)
	Add immediate	addi rs,imm	$rs \leftarrow (rs) + imm$
	Complement Immediate	compi rs,imm	$rs \leftarrow$ 2's Complement (imm)
Logic	AND	and rs,rt	$rs \leftarrow (rs) \wedge (rt)$
	XOR	xor rs,rt	$rs \leftarrow (rs) \oplus (rt)$
Shift	Shift left logical	shll rs, sh	$rs \leftarrow (rs)$ left-shifted by sh
	Shift right logical	shrl rs, sh	$rs \leftarrow (rs)$ right-shifted by sh
	Shift left logical variable	shllv rs, rt	$rs \leftarrow (rs)$ left-shifted by (rt)
	Shift right logical variable	shrlv rs, rt	$rs \leftarrow (rs)$ right-shifted by (rt)
	Shift right arithmetic	shra rs, sh	$rs \leftarrow (rs)$ arithmetic right-shifted by sh
	Shift right arithmetic variable	shrav rs, rt	$rs \leftarrow (rs)$ right-shifted by (rt)
Memory	Load Word	lw rt,imm(rs)	$rt \leftarrow mem[(rs) + imm]$
	Store Word	sw rt,imm,(rs)	$mem[(rs) + imm] \leftarrow (rt)$
Branch	Unconditional branch	b L	goto L
	Branch Register	br rs	goto (rs)
	Branch on zero	bz L	if $(zflag == 1)$ then goto L
	Branch on not zero	bnz L	if$(zflag == 0)$ then goto L
	Branch on Carry	bcy L	if $(carryflag == 1)$ then goto L
	Branch on No Carry	bncy L	if $(carryflag == 0)$ then goto L
	Branch on Sign	bs	if $(signflag == 1)$ then goto L
	Branch on Not Sign	bns L	if $(signflag == 0)$ then goto L
	Branch on Overflow	bv L	if $(overflowflag == 1)$ then goto L
	Branch on No Overflow	bnv L	if $(overflowflag == 0)$ then goto L
	Call	Call L	$ra \leftarrow$ (PC)+4 ; goto L
	Return	Ret	goto (ra)

KGPRISC follows a set of instructions that are listed in the ISA shown beside. It can support only these instructions and any instruction different from the ones listed needs to be written in terms of these.
Eg: **sub rs, rt** can be written as **comp rt, rt** followed by **add rs, rt.**



Instruction format that KGPRISC follows are shown here. R-Type instructions include all the arithmetic and the logical and shift instructions. I-Type instructions are addi, and compi. J-Type instructions are the branch instructions.

The instruction encoding followed for the instructions listed in the ISA for KGPRISC are shown below.

Instr	add	comp	and	xor	shll	shrl	shllv	shrlv	shra	shrav
opcode	000000	000000	000000	000000	000000	000000	000000	000000	000000	000000
func	000000	000001	000010	000011	000100	000101	000110	000111	001000	001001

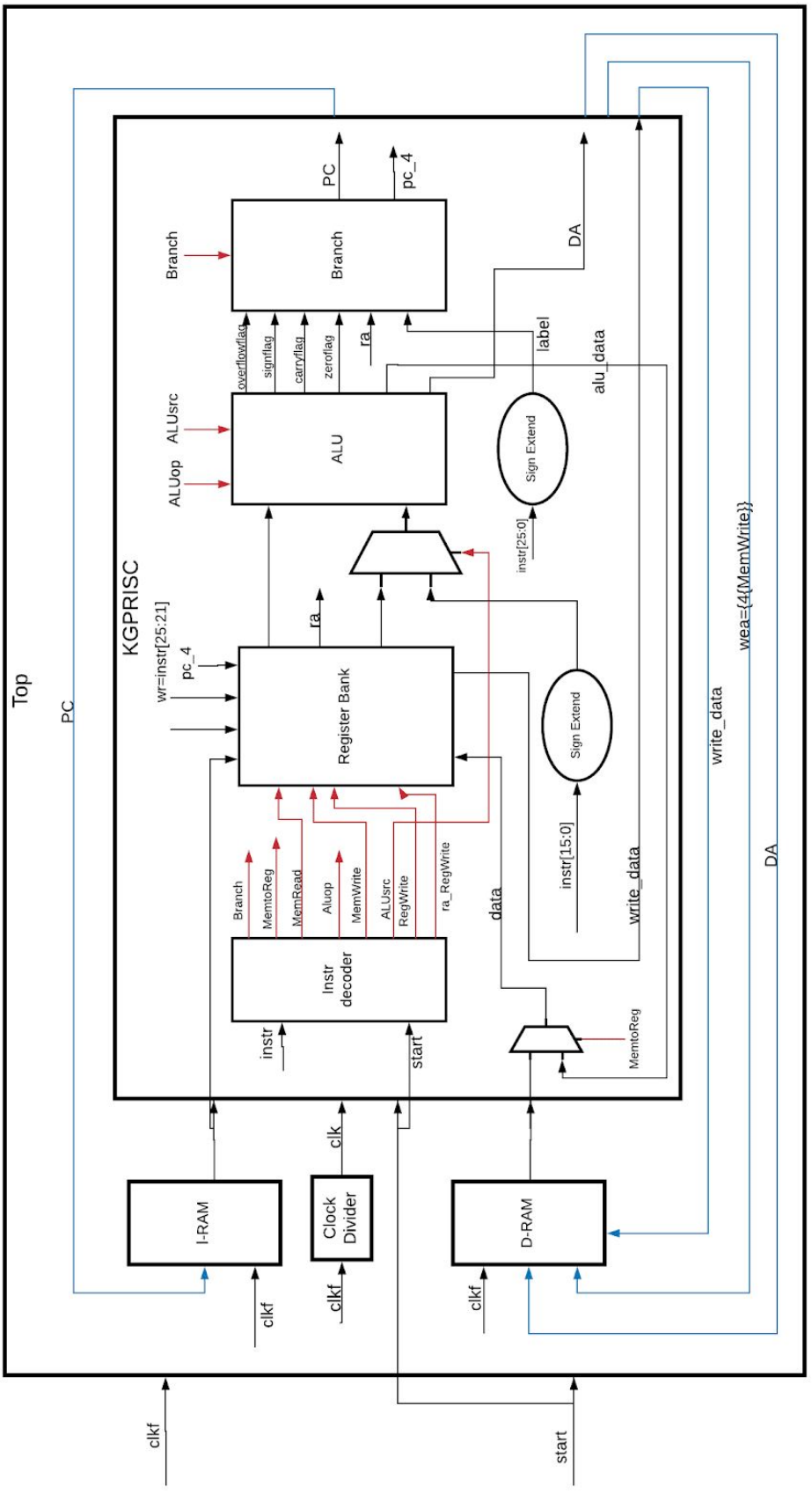
Instr	addi	compi	lw	sw	b	br	bz	bnz
opcode	000100	000101	000010	000011	010000	010001	010010	010011

Instr	bcy	bncy	bs	bns	bv	bnv	Call	Ret
opcode	010100	010101	010110	010111	011000	011001	000110	000111

KGPRISC uses **32 registers**.

Amongst these 32 registers, the last one (i.e., the 31st register, $ra) is reserved for storing, passing and manipulating the return address. This finds its utility when **Call** or **Ret** instructions are used. Even though not mandatory, it must be considered conventional to keep the first register reserved for zero.

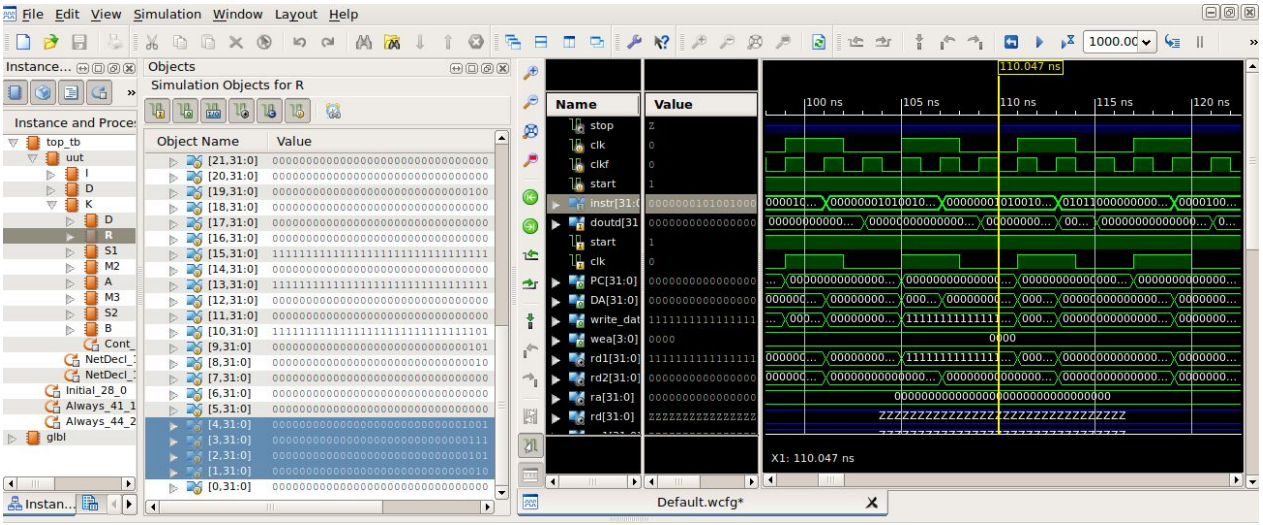
Conventionally the registers are named r0 … r31 starting from the first register. This register naming convention has been followed in our code for bubble sort too (ie the code you will find in our “instr.coe” file).


Top
PC
KGPRISC
wr=instr[25:21]
pc_4
Register Bank
ra
ALUsrc
ALUop
ALU
overflowflag
signflag
carryflag
zeroflag
ra
Branch
Branch
PC
pc_4
DA
label
alu_data
Sign Extend
instr[25:0]
Sign Extend
instr[15:0]
write_data
write_data
wea={4{MemWrite}}
DA
data
Branch
MemtoReg
MemRead
Aluop
MemWrite
ALUsrc
RegWrite
ra_RegWrite
Instr decoder
instr
start
MemtoReg
I-RAM
clkf
clk
Clock Divider
clkf
D-RAM
clkf
clkf
start

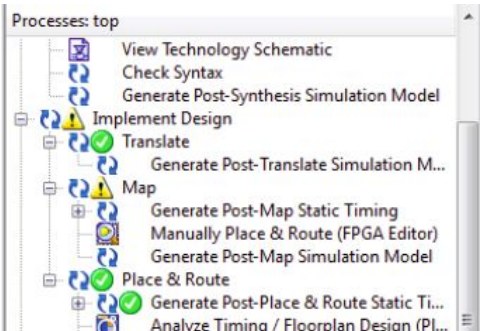
The **modules** used for the implementation of KGPRISC are:

- **Top** - The main module which houses the Block Rams and processor KGPRISC.
- **Clock Divider** - Divides the frequency of the clock.
- **Inst_memory & Data_memory** - B-RAM modules.
- **KGPRISC** - The processor.
- **Instr_decoder** - Control Unit to decode 32 bit instr to switch on/off control signals.
- **RegBank** - Register Bank, houses 32 register.
- **SignExtend** - Sign extends 16 bit to 32 bit.
- **ALU** - Operates arithmetic, logical and shift instructions.
- **Branch** - Checks whether to branch. If yes, goes to label, else increases program counter by 4.

Here we give the observations on implementing bubble sort using KGPRISC. A screenshot of the ISE simulator is attached where the result is shown in the registers highlighted.



In this sample working of **bubble sort** (the code to which is included in "**instr.coe**"), we stored 4 numbers in our data memory. We then allowed the simulation to run. In our scheme of things, the numbers get sorted and stored in a sorted manner in the data memory. For the purpose of demonstrating the correctness of the program, we have allowed the program to run on an initial input (as given in the "**data.coe**") and once the process is done, we have loaded the memory into **register number 1 to 4** in a sequential manner (ie. in the highlighted registers). As can be easily verified, the process worked and the numbers are now sorted.



Post-Place and Route Simulation model was generated in Xilinx ISE. Hence "**top.ucf**" file was generated, which is attached in the submitted file. If this is successful, we can be almost sure that the design is implementable on FPGA.

```
always@(posedge clk)
begin
   if((register[1]<register[2]) && (register[2]<register[3]) && (register[3]<register[4]))
      stop = 1;
   else
      stop = 0;
end
```

This is the simulation output of the program.
We are using the **stop** signal in the following way: After the numbers have been sorted in memory, they are loaded into successive registers, as mentioned. If and only if the numbers are in sorted order in the registers, then the stop signal will go to 1, else it would remain at 0. (see the image attached above). A successful high stop signal is proof that the sorting has worked (see the image below).


```
# PlanAhead Generated IO constraints

NET "clkf" IOSTANDARD = LVCMOS18;
NET "start" IOSTANDARD = LVCMOS18;
NET "stop" IOSTANDARD = LVCMOS18;

# PlanAhead Generated physical constraints

NET "clkf" LOC = F4;
NET "start" LOC = V10;
NET "stop" LOC = V11;
NET "start" CLOCK_DEDICATED_ROUTE = FALSE;
```

This is the **"top.ucf"** file which was used in our successful implementation.

"Every great piece of history must end with a quote"

--Confucius