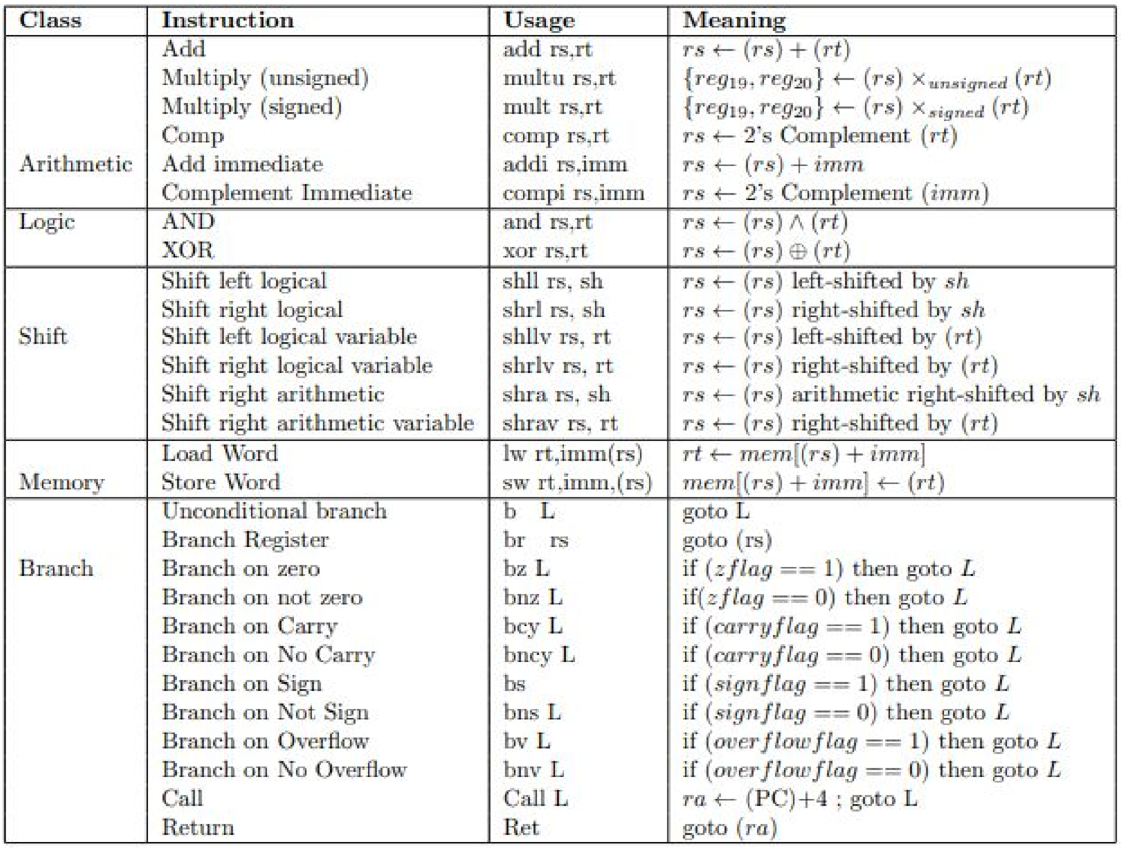
KGP-RISC

This document contains the design report for a 32-bit processor KGP-RISC. Instructions, as well as address, are kept to be 32 bit following RISC philosophy of architecture. Every instruction takes only one cycle to complete. Details regarding the processor have been listed below:

1. Instruction Set Architecture :

Set of all instructions :



Class	Instruction	Usage	Meaning
	Add	add rs,rt	$rs \leftarrow (rs) + (rt)$
	Multiply (unsigned)	multu rs,rt	$\{reg_{19}, reg_{20}\} \leftarrow (rs) \times_{unsigned} (rt)$
	Multiply (signed)	mult rs,rt	$\{reg_{19}, reg_{20}\} \leftarrow (rs) \times_{signed} (rt)$
	Comp	comp rs,rt	$rs \leftarrow$ 2's Complement (rt)
Arithmetic	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 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	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)
	Load Word	lw rt,imm(rs)	$rt \leftarrow mem[(rs) + imm]$
Memory	Store Word	sw rt,imm,(rs)	$mem[(rs) + imm] \leftarrow (rt)$
	Unconditional branch	b L	goto L
	Branch Register	br rs	goto (rs)
Branch	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)

2. Register Usage Convention :

This document summarizes the conventions registers should be used for:

Registers	Functions	Register Number	Register Code
$zero	stores the constant 0	0	00000
$v0 - $v1	stores return value of function calls	1 - 2	00001 - 00010
$a0 - $a3	used to pass function parameters	3 - 6	00011-00110
$s0 - $s11	saves variables preserved during function call	7 - 18	00111-10010
$hi	stores the higher bits for multiplication	19	10011
$lo	stores the lower bits for multiplication	20	10100
$t0 - $t8	stores temporary variables	21 - 29	10101-11101
$sp	stores stack address	30	11110
$ra	stores return address during function call	31	11111

3. Instruction Format and Encoding

All the instructions in our ISA have been divided into 4 basic parts based on their similarities. Instructions that are involved with computation have been grouped into R - format, those which have address present in instruction have been grouped into J - format, those which require transfer of data with memory have been grouped into Memory and those which have immediate value present in instruction have been grouped into I - format.

Opcode	Binary Representation	Format	Functions
0	00	R - Format	add, multu, mult, comp, and, xor, shll, shrl, shllv, shrlv, shra, shrav
1	01	J - Format	b, br, bz, bnz, bcy, bncy, bs, bns, bv, bnv, Call, Return

2	10	Memory	lw, sw
3	11	I - Format	compi, addi

Next we present instruction codes for all the instructions:

3.1 Opcode 00: R-Format Instructions

Opcode (6 bits)	rs (5 bits)	rt (5 bits)	shamt (5 bits)	Function (11 bits)

Functions :

Function	Function Codes	Binary Format
add	0	0000
comp	1	0001
and	2	0010
xor	3	0011
shll	4	0100
shrl	5	0101
shllv	6	0110
shlrv	7	0111
shra	8	1000
shrav	9	1001
mult	10	1010
multu	11	1011

3.2 Opcode 01: J-Format Instructions

Opcode (6 bits)	L (26 bits)

Functions:

Function	Opcode	Binary Format
b	3	000011
br	4	000100
bz	5	000101
bnz	6	000110
bcy	7	000111
bncy	8	001000
bs	9	001001
bns	10	001010
bv	11	001011
bnv	12	001100
Call	13	001101
Ret	14	001110

3.3 Opcode 10: Memory Access Instructions

Opcode (6 bits)	rs (5 bits)	rt (5 bits)	Immediate (16 bits)

Functions:

Function	Function Code	Binary Format
lw	1	000001

sw	2	000010

3.4 Opcode 11: I-Format Instructions

Opcode (6 bits)	rs (5 bits)	Immediate (21 bits)

Functions :

Function	Code	Binary Format
addii	0	001111
compi	1	010000

4. Implementation of different modules in Verilog

4.1 Instruction Fetch :

In the instruction fetch phase, a Block RAM(BRAM) module is generated using Block Memory Generator tool of Xilinx. The instructions (programs converted to binary bits based on our instruction format) are fed to the memory during the initialisation phase of block ram through a .coe file. The module takes as input a PC (program counter) value and a clock signal, and at every positive edge of the clock, it outputs the instruction (of 32-bits) present in the memory at the location specified by PC value.The BRAM module has the signals clock(clka), reset(rsta), enable(ena), write-enable,address(for reading or writing)(wea), write-data-input(dina) as input signals and read-data-output(douta) as output signals.

4.2 Instruction Decoder :

It is clear from the register allocation document that the above set of bits are overlapping. This is because depending on the opcode, only a certain set will be used at a time. So, this module clubs all of the instructions as per the definitions and depending on the flags set by the Control Unit (which in turn take the opcode as input) various other modules of the processor use the above buses.

4.3 Datapath :

In the DataPath, apart from the **clk** and **rst** signals, the addresses to the registers(**rs** and **rt**), the Write Enable(**regWrite**) flag all the inputs to the ALU (**alu_control**, **ALU_src**, **alu_result**, **sign_flag**, **carry_flag**, **overflow_flag**, **zero_flag**), and all the control signals of the MUXes from the Control Unit. The DataPath includes the RegisterBank, the ALU and the DataMemory modules, along with four 32bit 2x1 MUXes and one 5 bit 2x1 MUX.

● Reg_input MUX: -
This MUX is used to select the input to the Write Port of the Register Bank and it selects among the data coming as a result of the ALU operation and the Data Memory. This MUX is controlled by the **reg_data** signal from the Control Unit.

● regBank_pc_input MUX: -
This MUX selects from the output of the above MUX and the output of the program counter(explicitly for the case of the function call) and is governed by the **reg_to_pc** flag.

● regBank_addr_input MUX: -
This is a 5-bit 2:1 MUX and is governed by the **reg_to_pc** flag. This is used to select the address of the register being written to. This is used explicitly in the case of return and call instructions.
.
● regWriteAddr_select MUX:
This is also a 5-bit 2:1 MUX and is used to select the address for writing the data into the register bank. In case of load and store instructions, the address for writing is same as the address of second read register. In all other cases, the address is the same as that of the first read register.

● Register Bank:-
This is a sequential module and takes **clk** and **rst** as input. This module holds 32, 32bit registers. It has three inputs of which two(rs and rt) are 5-bit inputs as they are used to locate the registers whose values will be given as output.
The values of the registers given by the above two addresses are given as output. If the **regWrite** flag(from Control Unit) is enabled then the 32-bit input over the **regWriteData** bus will be written to the register pointed to by rt.

● ALU_constant MUX:-
This MUX is controlled by the signal **const_src** and is used to select between the **immediate_constant** and the **shift_amoun**t depending on the instruction.

● ALU_Input MUX: -
This MUX is used to select between the data coming from the output to the Register Band and the Output to the **ALU_constant** MUX. It is governed by the **ALU_src** flag form the Control Unit.

● ALU:-
This is a purely combinational module that takes one input from the register Bank, one from the above MUX and gives a 32-bit output after performing an operation which is controlled by the **alu_control** select instruction which is of four bits and the default output is zero. The ALU performs the following operations in our case:-
1) Addition
2) Subtraction
3) Compliment
4) Left Shift
5) Right Shift
6) Sign Preserved Right Shift
7) Pass-Through of Operand 1
8) Pass-Through of Operand 2
9) Signed multiplication
10) Unsigned multiplication
And at each instance, sets the following flags: - Sign, Carry, Overflow and Zero.

● Data Memory: -
This is the module simulating the RAM but in the given implementation, we have an array of 1024 registers which are 32-bits each. Also the implementation of these two dimensional arrays are same as that generated by block memory generator. In case the MemWrite flag is true then data from the output of the register bank is written to the data. If the MemRead flag is true then the Data Memory outputs the contents of the location pointed to by the address input to the Data Memory.

4.4 Control - Unit :

The control unit takes as input the 6-bit opcode and the 11-bit opcode extension and outputs the values of various control signals that determine the operations taken in other modules. The control signals given as output by this unit are:

1. ALU control signal - alu_control
Determines the operation to be performed by the ALU.
2. Branch type instruction signal - Branch
Is high for control flow instructions like unconditional or conditional jumps, function calls, and return statements and is low otherwise.
3. Register Bank write enable signal - regWrite
Is high when data needs to be written or updated in register bank, for e.g. load word, arithmetic operations, etc and low otherwise.
4. Memory unit write enable signal - MemWrite
Is high when data needs to be written to the memory unit, i.e. for store word instructions and low otherwise.
5. Memory unit read enable signal - MemRead
Is high when data needs to be read from the memory unit, i.e. for load word instructions and low otherwise.
6. Select line of MUX for ALU's second input (either from register bank or constant value) - ALU_src Is high when data is ALU's second input comes as a constant value in instruction, for e.g. addi, compi, constant shifts or jump offsets. It is low when the second input of ALU is also from register bank like the first input, for e.g. arithmetic operations, variable shift amounts, etc.
7. Select line of MUX for constant input (either immediate constant or shift amount) - const_src Is high when the constant value for previous MUX comes from the instruction's shift amount encoding location, i.e. for constant shift instructions and the value of the signal is low when the constant value is the immediate constant, i.e. for addi or compi instructions.
8. Select line of MUX for register bank's input source (either from ALU or from Memory Unit) reg_data Is high when the input to be written in the register bank is coming from ALU's result, for e.g. arithmetic operations, shift operations, logical operations, etc. and it is low the input data is coming from the memory unit, i.e. for load word instruction.
9. Select line of MUX for register bank's input source (either from previous MUX i.e. ALU/Memory or from PC) - reg_to_pc

This mux is in cascade to the previous mux. This select line is low when the input to be written into the register bank comes the previous MUX, i.e. either ALU or memory unit. The signal is high when the input is equal to (PC+4) to be written into register (ra) in

case of function calls. The alu_control signal is a 4-bit signal that dictates the operation to be performed by the ALU. All the other signals are 1-bit signals.
The control unit also an asynchronous reset that sets all the output signals to zero when high. When reset is low, the control signals are set for every change in opcode or opcode_ext.

4.5 PC_next module :

The PC_next module computes the program counter location for the next instruction depending on the current value of PC and offset value if any. The branch signal coming from the branching logic module specifies if the current instruction is a branch type and if the branching condition (for conditional jumps) is satisfied or if it is an unconditional jump, in which case the next pc is computed by adding the offset to the current pc+4 value. Else, if there is no successful branch, the next pc is set to pc+4.

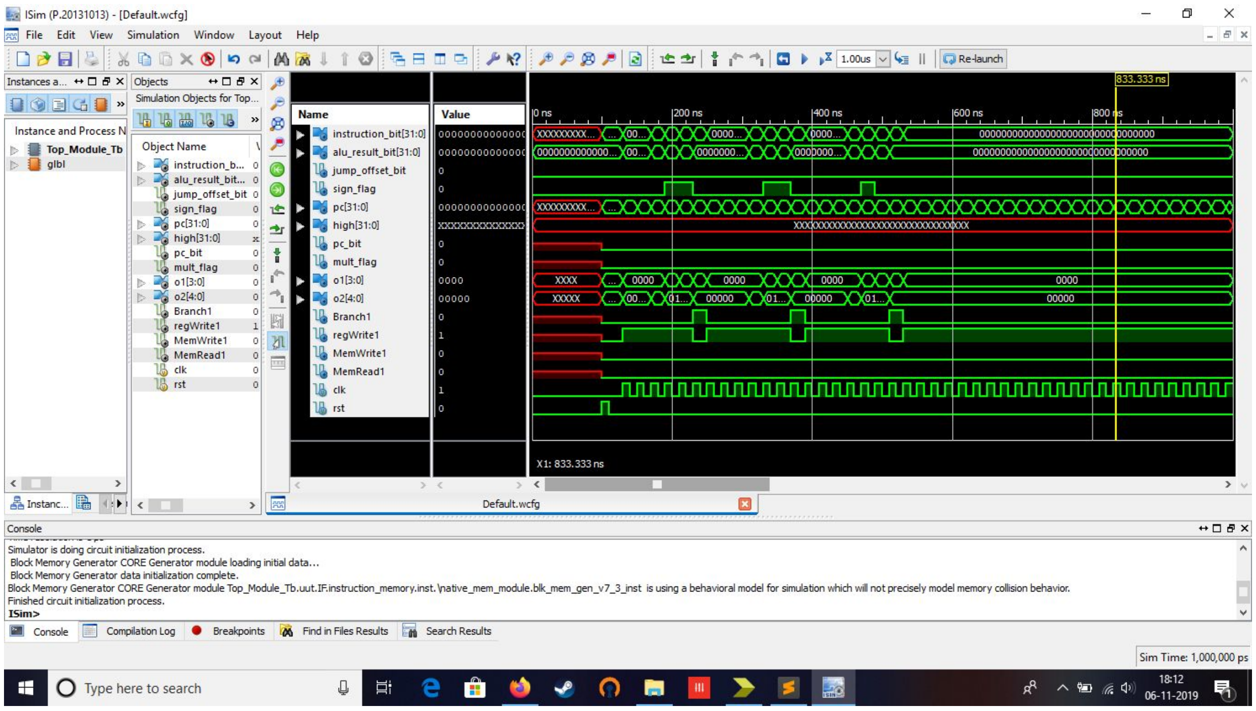
4.6 Branching logic :

The branching logic determines if the current instruction is a branch type instruction and assigns the offset to the offset_out signal in case the branch is a successful branch.
1. In case of unconditional jumps (either constant label or register label), the offset_out is set equal to the offset_in signal, which is taken directly from the instruction encoding and the branch signal is high, indicating a successful branch.
2. In case of jumps to the label specified in a register, like br instruction or return instruction, the offset_out is set from the rs_value, and the branch signal is high, indicating a successful branch.
3. In the case of conditional jumps, depending on the opcode and the flags supplied as input, the offset and branch signals are set accordingly. If the branch is successful, offset_out is equal to offset_in which is taken directly from the instruction encoding and branch signal is high. If the branch condition is not true, then the offset is set to the default value of 0, and the branch signal low, indicating no branch.

4.7 D-Flip Flops :

The input flags: **zero flag**, **carry flag**, **sign flag**, and **overflow flag** obtained from the ALU are stored in D-flip-flops inside this module, so that the flag values from previous clock cycle when they were computed can be used in the next clock cycle when the conditional jump instruction is given. Without the flip-flops, the flags that are computed in previous clock cycles are overwritten with some garbage values in the current clock cycles and condition checking wouldn’t be possible.

Synthesis:



```
Simulator is doing circuit initialization process.
 Block Memory Generator CORE Generator module loading initial data...
 Block Memory Generator data initialization complete.
Block Memory Generator CORE Generator module Top_Module_Tb.uut.IF.instruction_memory.inst.\native_mem_module.blk_mem_gen_v7_3_inst  is using a behavioral model for simulation which will not precisely model memory collision behavior.
Finished circuit initialization process.
ISim>
```

Sim Time: 1,000,000 ps

Submission of group - 2:

Kanishk Singh - 17CS30018

Shrey Shrivastava - 17CS30034