Computer Organization and Architecture

KGP-RISC

October 07, 2019

Overview

In this report we go through the design, implementation and testing phases of a custom instruction set architecture called KGP-RISC.

Goals

1. Design.
2. Implementation.
3. Simulation.

Specifications

KGP-RISC has the Instruction Set Architecture (ISA) as shown below.The Processor word size is 32 bits, and the processor uses 32-bit addresses. Also, the processor has a total of 32 registers, of which certain registers (e.g. ra which stores the return address) play similar roles as the corresponding registers in the MIPS-32 ISA. Out of the 32 registers, register no. 19 and reg. no. 20 together can be used to hold the product of a signed/unsigned multiply operation on two 32-bit numbers — reg. no. 19 is used to hold the most significant word and reg. no. 20 for the least significant word, while the opcode determines whether the multiplication is signed or unsigned.

Class	Instruction	Usage	Meaning
Arithmetic	Add	add rs, rt	rs = (rs) + (rt)
	Multiply	mult rs, rt	{reg_19, reg_20} = (rs)*(rt)
	Multiply(Unsigned)	mult rs, rt	{reg_19, reg_20} = (rs)*(rt)
	Comp	comp rs, rt	rs = 2's complement(rt)
	Add Immediate	addi rs, imm	rs = (rs) + imm
	Comp Immediate	compi rs, imm	rs = 2's complement(imm)
Logic	AND	and rs, rt	rs = (rs)&(rt)
	XOR	xor rs, rt	rs = (rs)^(rt)
Shift	Shift Left Logical	shll rs, sh	rs = (rs) << sh
	Shift Right Logical	shrl rs, sh	rs = (rs) >> sh
	Shift Left Logical Variable	shllv rs, rt	rs = (rs) << (rt)
	Shift Right Logical Variable	shrlv rs, rt	rs = (rs) >> (rt)
	Shift Right Arithmetic	shra rs, sh	rs = (rs) >> sh (signed bit preserved)
	Shift Right Arithmetic Variable	shrav rs, rt	rs = (rs) >> (rt)
Memory	Load Word	lw rt, imm(rs)	rt = mem(rs + imm)
	Store Word	sw rt, imm(rs)	mem(rs + imm) = rt
Branch	Unconditional Branch	b L	goto L
	Branch Register	br rs	goto (rs)
	Branch on Zero	bz L	if(zflag==0) goto L
	Branch on Not Zero	bnz L	if(zflag!=0) goto L
	Branch on Carry	bcy L	if(carryflag==0) goto L
	Branch on No Carry	bncy L	if(carryflag!=0) goto L
	Branch on Sign	bs L	if(signflag==0) goto L
	Branch on Not Sign	bns L	if(signflag!=0) goto L
	Branch on Overflow	bv L	if(overflowflag==0) goto L
	Branch on No Overflow	bnv L	if(overflowflag!=0) goto L
	Call	call L	ra = (PC) + 4, goto L
	Return	ret L	goto (ra)

Milestones

1. Decide the registers and the register usage convention.
2. Design a suitable instruction format and instruction encoding.
3. Design and implement the Instruction Decoder.
4. Design and implement the Register Bank.
5. Design and implement the Arithmetic Logic Unit (ALU) in a hierarchical manner, with possibly different modules responsible for different operations.
6. Design and implement the Branching Logic, in conjunction with the ALU.
7. Design and implement the Load-Store unit.
8. Design and implement the Control Unit.
9. Simulate the entire design (pre-and post-synthesis), with proper testbenches.

Register Usage Convention

Register Number	Symbol	Usage
0	$zero	Always contains zero
1	$at	Assembler Temporary
2 to 3	$v0 - $v1	Function return value
4 to 7	$a0 - $a3	Function parameters
8 to 15	$t0 - $t7	Function temporary values
16 to 18	$s0 - $s2	Saved registers across function calls
19 to 20	$lo - $hi	Multiplication results
21 to 26	$s3 - $s8	Saved registers across function calls
27	$t8	Function temporary values
28 to 29	$sp and $gp	Stack pointer and Global pointer
30	$ra	Return address from function call
31	Program Counter	Points at 8 bytes past current instruction

Instruction Formats and Encodings

R-Type Instructions (opcode = 000000)

opcode (6 bits)	rs	rt	Always Zero	shamt	func code (6 bits)

Arithmetic Unit

Instruction	31:26	25:21	20:16	15:11	10:6	5:0
ADD	000000	rs	rt	00000	00000	100000
MULT	000000	rs	rt	00000	00000	011000
MULTU	000000	rs	rt	00000	00000	011001
SHLLV	000000	rs	rt	00000	00000	000100
SHRLV	000000	rs	rt	00000	00000	000110
SHRAV	000000	rs	rt	00000	00000	000111
SHLL	000000	rs	00000	00000	sh	000000
SHRL	000000	rs	00000	00000	sh	000010
SHRA	000000	rs	00000	00000	sh	000011

Logical Unit

Instruction	31:26	25:21	20:16	15:11	10:6	5:0
AND	000000	rs	rt	00000	00000	100100
XOR	000000	rs	rt	00000	00000	100110

Branch (Arithmetic Unit)

Instruction	31:26	25:21	20:16	15:11	10:6	5:0
BR	000000	rs	rt	00000	00000	001000

I-Type Instructions

opcode (6 bits)	rs	Not Used	Immediate Value

Instruction	31:26	25:21	20:16	15:0
ADDI	001000	rs	000000	Immediate Value
COMPI	001100	rs	000000	Immediate Value

Memory Access

opcode (6 bits)	rs	rt	Immediate Value

Instruction	31:26	25:21	20:16	15:0
LW	100011	rs	rt	Immediate Value
SW	101011	rs	rt	Immediate Value

B-Type Instructions

opcode (6 bits)	Label (26 bits)

Instruction	31:26	25:0
B	010000	Label
BZ	010001	Label
BNZ	010011	Label
BCY	010100	Label
BNCY	010101	Label
BS	010110	Label

BNS	010111	Label
BV	011000	Label
BNV	011001	Label
CALL	011010	Label
RET	011011	Label

Instruction Fetch

Block RAM hard macro modules on XILINX FPGA is instantiated in the code to emulate memory.

```
module instruction_memory

(

clk,

address,

instruction

);



      input [31:0]  address;

      output [31:0] instruction;

      blk_mem_gen_v7_3 mem(.clka(clk), .wea(0), .addra(address), .dina(0),
      .douta(instruction));

 endmodule
```

Register File

This Module will assist with all the interactions with the registers.

```
module register_file(

input clk,

input [4:0] raddr0,        // address of register a (to read)

input  [4:0]  raddr1,      // address of register b (to read)

input  [4:0]  waddr,       // address of register c (to write)

input  [31:0] wdata,       // data to be written to register

input wren,                // write enable

output [31:0] rdata0,      // data read from register a

output [31:0] rdata1       // data read from register b

);

reg [31:0] register_file[0:31];  // Instatiating 32 Registers as per Instruction Set

assign rdata0 = register_file[raddr0];  // data read from register a

assign rdata1 = register_file[raddr1];  // data read from register b


// Logic to write in register (Clock Synchronized)

always @ (posedge clk)

if (wren) begin

       register_file[waddr]=wdata;

end

endmodule
```

Control Unit

```
always @(instruction) begin

        if (op == SW && !rst) begin

                data_mem_wren = 4'b1111;      // Get ready to write data into memory

        end else begin

                data_mem_wren = 4'b0000;

        end

        if (rst || op == SW || op == B || op[4]==1 || funct == BR) begin

                reg_file_wren = 0;            // Get ready to write something in a register

        end else begin

                reg_file_wren = 1;

        end

        if (op == LW) begin

                reg_file_dmux_select = 0;

        end else begin

                reg_file_dmux_select = 1;

        end

        if (op == R) begin

                reg_file_rmux_select = 1;

        end else begin

                reg_file_rmux_select = 0;

        end

        if ((op == R || op[4] == 1) && (((op == R) && (funct != SHLL && funct!=SHRL &&
funct!=SHRA))) begin

                alu_mux_select = 0;

        end else begin // I-type
                alu_mux_select = 1;

        end

        if ((op == R && funct == ADD) || op == SW || op == ADDI || op == LW) begin

                alu_control = C_ADD;

        end else if (op == R && funct == MULT) begin

                alu_control = C_MULT;

        end else if (op == R && funct == MULTU) begin

                alu_control = C_MULTU;

        end else if (op == R && funct == AND) begin

                alu_control = C_AND;

        end else if (op == R && funct == XOR) begin

                alu_control = C_XOR;

        end else if (op == R && (funct == SHLL || funct == SHLLV)) begin

                alu_control = C_SHLL;

        end else if (op == R && (funct == SHRL || funct == SHRLV)) begin

                alu_control = C_SHRL;

        end else if (op == R && (funct == SHRA || funct == SHRAV)) begin

                alu_control = C_SHRA;

        end else begin

                alu_control = 4'b1111;

        end

        if (op == R && (funct == SHLL || funct == SHRL || funct == SHRA)) begin

                alu_shamt = instruction[10:6];

        end else begin

                alu_shamt = 5'b00000;

        end
```

```
#0.5

if (op == B) begin

       pc_control = 4'b0001;

end else if (op == R && funct == BR) begin

       pc_control = 4'b0010;

end else if (op == BZ && zflag == 0) begin

       pc_control = 4'b0011;

end else if (op == BNZ && zflag == 1) begin

       pc_control = 4'b0100;

end else if (op == BCY && carryflag == 0) begin

       pc_control = 4'b0101;

end else if (op == BNCY && carryflag != 1) begin

       pc_control = 4'b0110;

end else if (op == BS && signflag == 0) begin

       pc_control = 4'b0111;

end else if (op == BNS && signflag != 1) begin

       pc_control = 4'b1000;

end else if (op == BV && overflowflag == 0) begin

       pc_control = 4'b1001;

end else if (op == BNV && overflowflag != 1) begin

       pc_control = 4'b1010;

end else if (op == CALL) begin

       pc_control = 4'b1011;

end else if (op == RET) begin

       pc_control = 4'b1100;

end else begin
```

```
                // PC = PC+4
                pc_control = 4'b0000;
        end
end
```