KGP-RISC Documentation

Group - 3

1. *G Rahul Krantikiran - 16CS10018*
2. *Sai Saketh Aluru - 16CS30030*

Instruction description:

Each instruction for the processor contains 32 bits as shown,

Op-code 6-bits	Destination register 5-bits	Source register 5-bits	Shift amount 5-bits	Opcode extension 11-bits

The register usage convention for the available 32 registers is,

Register Numbers	Register Name	Description
0	$zero	Zero value
1-2	$v0 - $v1	Function Return values
3-7	$a0 - $a4	Function parameters
8-21	$t0 - $t13	Temporary registers
22-29	$s0 - $s7	Callee saved registers
30	$sp	Stack Pointer
31	$ra	Function Return address

The Instruction Set Architecture (Also Attached as a **Separate File**) is,

Instruction	Usage	Description	Opcode(6)	R1 (5)	R2 (5)	Shift amount(5)	Immediate value (21)	Opcode Extension(11)
R-type instructions								
Add	add rs,rt	rs <- (rs) + (rt)	000000	rs	rt	XXXXX	*XXXXXXXXXXXXXXXXXXXXX*	00000000000
Complement	comp rs,rt	rs <- 2's complement (rt)	000000	rs	rt	0	*XXXXXXXXXXXXXXXXXXXXX*	00000000001
And	and rs,rt	rs <- (rs) & (rt)	000000	rs	rt	XXXXX	*XXXXXXXXXXXXXXXXXXXXX*	00000000010
Xor	xor rs,rt	rs <- (rs) ⊕ (rt)	000000	rs	rt	XXXXX	*XXXXXXXXXXXXXXXXXXXXX*	00000000011
Shift left logical	shll rs,sh	rs <- (rs) left shifted by sh	000000	rs	XXXXX	shift amount sh	*XXXXXXXXXXXXXXXXXXXXX*	00000000100
Shift right logical	shrl rs,sh	rs <- (rs) right shifted by sh	000000	rs	XXXXX	shift amount sh	*XXXXXXXXXXXXXXXXXXXXX*	00000000101
Shift left logical variable	shllv rs,rt	rs <- (rs) left shifted by (rt)	000000	rs	rt	XXXXX	*XXXXXXXXXXXXXXXXXXXXX*	00000000110
Shift right logical variable	shrlv rs,rt	rs <- (rs) right shifted by (rt)	000000	rs	rt	XXXXX	*XXXXXXXXXXXXXXXXXXXXX*	00000000111
Shift right arithmetic	shra rs, sh	rs <- (rs) arithmetic right shifted by sh	000000	rs	XXXXX	shift amount sh	*XXXXXXXXXXXXXXXXXXXXX*	00000001000
Shift right arithmetic variable	shrav rs,rt	rs <- (rs) arithmetic right shifted by (rt)	000000	rs	rt	XXXXX	*XXXXXXXXXXXXXXXXXXXXX*	00000001001
I-type instructions								
Load Word	lw rt,imm(rs)	rt <- mem[(rs) + imm]	000001	rs	rt	XXXXX	immediate constant value (16 bits)	XXXXXXXXXX
Store Word	sw rt, imm(rs)	mem[(rs) + imm] <- rt	000010	rs	rt	XXXXX	immediate constant value (16 bits)	XXXXXXXXXX
Add immediate	addi rs,imm	rs <- (rs) + imm	001111	rs	XXXXX	XXXXX	immediate constant value	XXXXXXXXXX
Complement immediate	compi rs,imm	rs <- 2's complement of imm	010000	rs	XXXXX	XXXXX	immediate constant value	XXXXXXXXXX
Branch Register	br rs	goto contents(rs)	000100	rs	XXXXX	XXXXX	*XXXXXXXXXXXXXXXXXXXXX*	XXXXXXXXXX
Bit Position(Allocation)			31..26	25..21	20..16	15..11	20..0	10..0

Instruction	Usage	Description	Opcode(6)	Offset (26)
Unconditional branch	b L	goto L	000011	*offset*
Branch on zero	bz L	if (zflag == 1) then goto L	000101	*offset*
Branch on not zero	bnz L	if(zflag == 0) then goto L	000110	*offset*
Branch on Carry	bcy L	if (carryflag == 1) then goto L	000111	*offset*
Branch on No Carry	bncy L	if (carryflag == 0) then goto L	001000	*offset*
Branch on Sign	bs L	if (signflag == 1) then goto L	001001	*offset*
Branch on Not Sign	bns L	if (signflag == 0) then goto L	001010	*offset*
Branch on Overflow	bv L	if (overflowflag == 1) then goto L	001011	*offset*
Branch on No Overflow	bnv L	if (overflowflag == 0) then goto L	001100	*offset*
Call	call L	ra ← (PC)+4 ; goto L	001101	*offset*
Return	Ret	goto contents(ra)	001110	*XXXXXXXXXXXXXXXXXXXXXXXXXX*
Bit Position(Allocation)			31..26	25..0

Instruction Fetch:

In the instruction fetch phase, a Block RAM(BRAM) module is instantiated which acts as the instruction memory. The instructions (source code) is fed to the memory during the initialisation phase of BRAM in 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.

Instruction Decoder:

The instruction decoder in our processor design is a module that takes as input the instruction provided to it by the Instruction Fetch module and it separates it into the following components: -

1) Opcode
2) Opcode ext
3) rs
4) rt
5) Shift Amount
6) Immediate Constant
7) Offset

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.

Datapath:

We have designed our entire data path into a separate module. As can be seen by the schematic diagram of the datapath also attached with this document. In the DataPath, apart from the clk and rst signals, the addresses to the registers(regAddr_1 and regAddr_2), 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 32 bit 2x1 MUXes and one 5 bit 2x1 MUX. (Details can be seen from the Diagram).

- *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(redAddr_1 and regAddr_2) 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 regAddr_1.

- *ALU_constant* MUX:-

  This MUX is controlled by the signal *const_src* and is used to select between the *immediate_constant* and the *shift_amount* 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

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.

  In this module, there is a 10 bit address bus which is the result of the ALU. and a 32-bit data write BUS which is the output from the Register Bank. 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.

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.

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.

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.

- 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 Statistics

1. Adders/Subtractors - 4
    a. 32-bit adder - 3
    b. 32-bit adder carry out - 1
2. Registers - 53
    a. 1-bit register - 4
    b. 32-bit register - 49
3. Latches - 10
    a. 1-bit latch - 9
    b. 4-bit latch - 1
4. Multiplexers - 3
    a. 32-bit 16-to-1 multiplexer - 1
    b. 32-bit 32-to-1 multiplexer - 2
5. Logic shifters - 2
    a. 32-bit shifter logical left - 1
    b. 32-bit shifter logical right - 1
6. Xors - 2
    a. 1-bit xor4 - 1
    b. 32-bit xor2 - 1

Execution Facts

- In the instruction files, due to the nature of the program counter, one needs to add a no-op instrunction after every branch statement and the offset must point to the instruction before the target of the jump.
- Due to the BRAM delay, one needs to use the complment of the clock as an input to the BRAM and one has to add two no-op instructions at the front to accomodate the initials delays.