**ALU design in Verilog using MIPS Instruction Set**

MIPS is a RISC (Reduced Instruction Set Computer) based architecture which is used in MIPS based processors.

Designing a simple ALU in Verilog using few example instructions from the MIPS Instruction Set.

MIPS Instructions are classified into three types: R, I and J. They have the following specified formats:

**R-Type:**

[](https://blogger.googleusercontent.com/img/a/AVvXsEhXczWgdvVHe4Vv8vIw5zzufdaXPR3-jWlcwv02aeK4dcgoyuxz7zVk_qECzkaopYyXygH5vbUlsM2fix7ecSTKy9vk8U9kSvrkdXHJIR-LgongvbkDaJ07FRRTWtMkyaZkD8QwmV1r9aSeemSMBCCOdpHVh-Bfy1E6-1NHyfKTcU6de8ePFDjF_Y-9pw=s605)

op - Opcode  
rs, rt - Source Registers 1 and 2  
rd - Destination Register  
shamt - Shift Amount  
funct - Function Code

**I-Type:**

[](https://blogger.googleusercontent.com/img/a/AVvXsEgqwq4Lm3AsCnyhSXYt4w_3_MuRRuInYQmEzwdRlDLEhjOfavQoINOAYwP-J4Ov0DE_uLZKKfhwBdqP2asTVLx_HoujO_s4ezTTsU7B2pvak80Y124IHH-jWlJcWVHKNKB8cp8qPLduYIua3tGBuBd5VNRpVe9aOiSKrzIxckaglSXZA3zi_nwlI6qggQ=s405)

**J-Type:**

[](https://blogger.googleusercontent.com/img/a/AVvXsEhpERIBFebeWashqhkINQx8OYJNbaPR0d8WlpO5e-rQqhX_aZ-9UqRyaYM7MfLfFGeHIcMQlNCltZUUjMd7gdx5VknS6k5-JrI5eN-CKGwa3h61OIGep4OsaAljnZKlqCAmfpGT4vCYal_xOmSx6meuHW-qk-QapoHtWN4XUmQDz4DdLiO3Q5zhklulSw=s206)

**Step 1:** **Decide the various instructions that ALU must support.**

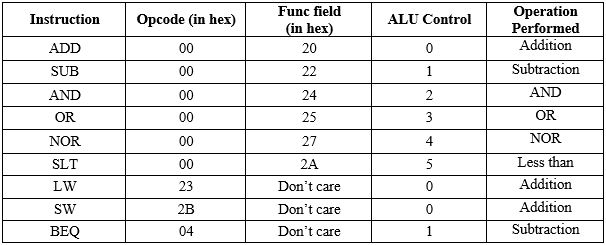
From the MIPS Instruction Set, ALU supports the following instructions:  
AND, OR, ADD, SUB, SLT, NOR, LW, SW, BEQ.

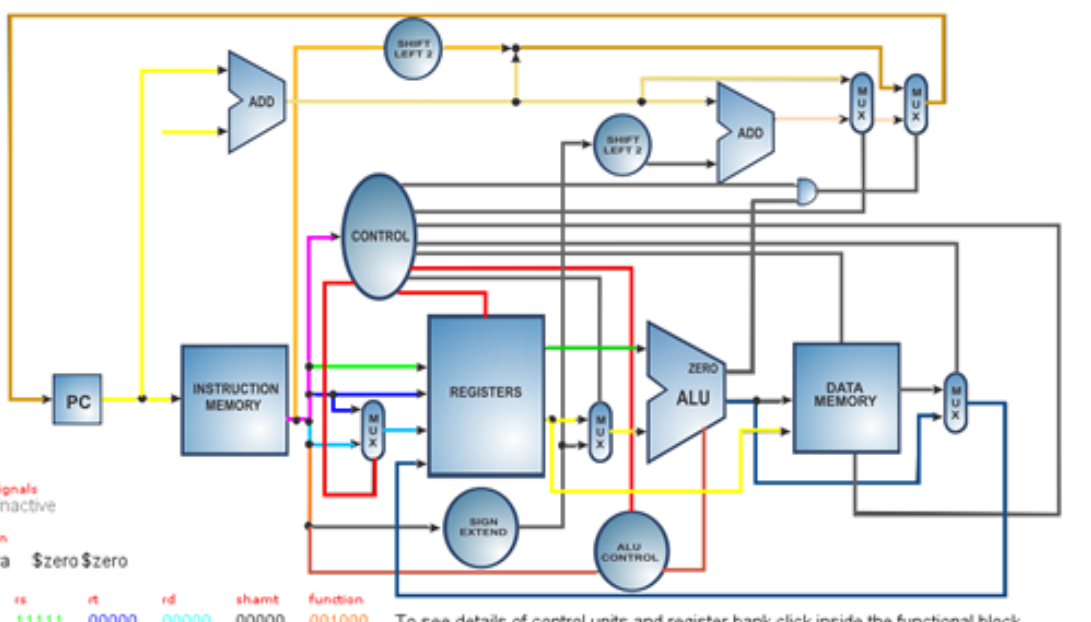
**Step 2: ALU Control Signal Generation**

* For I-Type Instructions, the type of operation to be performed is found by looking at the opcode.
* For R-Type Instructions, the type of operation is determined by the function field (opcode field will remain as all zeroes).

Control signal for the ALU is required to determines the type of operation to be performed by looking at either the opcode for I-Type or function field for R-Type instructions.

In this case, it is as simple as: if the opcode field is all zeroes, then look at the function field.

[](https://blogger.googleusercontent.com/img/a/AVvXsEjdfo5If4n1MSqB489kkgHDwOla5FiKIS4oZx3yFvDEJ-HnrPL0yh4rPLZWFXtbYuJRtlbclkcxF-kr9bq0fwU29wCuZB8OJw_JXCkXGwAIrOj37twiebxPZsGPrhpes9agy0ruaLWvMG0zUv9UZmWZXI3Y7u-sWLRqnmbQFiljE8G7ri1myE5Gq2XTiA=s606)



* In LW and SW, perform addition as the offset must be added with the base address.
* ALU Control is the signal which is sent to the ALU core to indicate what type of operation is to be performed.
* For BEQ, subtraction must be performed. If the operands are equal, then the subtraction result will be 0 and the branching condition will be true. ALU will have a separate zero output signal to indicate output zero condition.

Verilog Module: ALU\_Controller  
Inputs: opcode (6 bits), func\_field (6 bits)  
Outputs: alu\_control (3 bits)

**Verilog Code for ALU Control:**

**module** Alu\_Control(opcode, func\_field, alu\_control);

**input** [**5**:**0**] opcode;

**input** [**5**:**0**] func\_field;

**output** **reg** [**2**:**0**] alu\_control;

**reg** [**2**:**0**] func\_code; //A

**always** @ (\*)

**begin //for R type instructions**

**case** (func\_field)

**6'h20**: func\_code = **3'h0**;

**6'h22**: func\_code = **3'h1**;

**6'h24**: func\_code = **3'h2**;

**6'h25**: func\_code = **3'h3**;

**6'h27**: func\_code = **3'h4**;

**6'h2A**: func\_code = **3'h5**;

**default**: func\_code = **3'h0**;

**endcase**

**case** (opcode)

**6'h00**: alu\_control = func\_code;

**6'h04**: alu\_control = **3'h1**;

**6'h23**: alu\_control = **3'h0**;

**6'h2B**: alu\_control = **3'h0**;

**default**: alu\_control = **3'h0**;

**endcase**

**end**

**endmodule**

**Step 3: ALU Core Design**

Now that the control signal tells us the type of operation to be performed, the desired operation can be performed in the ALU core module. Let the input operands be A and B. The computed output is sent out as result.

Verilog module: ALU\_Core  
Inputs: alu\_control (3 bits), A (32 bits), B (32 bits)  
Outputs: result (32 bits), zero (1 bit)

**Verilog Code for ALU Core:**

**module** Alu\_Core(**A**, **B**, alu\_control, result, zero);

**input** [**31**:**0**] **A**;

**input** [**31**:**0**] **B**;

**input** [**2**:**0**] alu\_control;

**output** **reg** [**31**:**0**] result;

**output** **wire** zero;

**assign** zero = !(|result); //oring the entire result to 1bit value then do negation

**always** @ (\*)

**begin**

**case**(alu\_control)

**3'h0**: result = **A** + **B**;

**3'h1**: result = **A** - **B**;

**3'h2**: result = **A** & **B**;

**3'h3**: result = **A** | **B**;

**3'h4**: result = ~(**A** | **B**);

**3'h5**: result = (**A** < **B**);

**default**: result = **A** + **B**;

**endcase**

**end**

**endmodule**

**Step 4:** **Create the ALU top module.**

The top module instantiates and connects both the above modules.

Verilog Module: ALU\_Top  
Inputs: opcode (6 bits), func\_field (6 bits), A (32 bits), B (32 bits)  
Outputs:  result (32 bits), zero (1 bit)

**Verilog Code for ALU Top:**

**module** Alu\_Top(

opcode,

func\_field,

**A**,

**B**,

result,

zero

);

**input** [**5**:**0**] opcode;

**input** [**5**:**0**] func\_field;

**input** [**31**:**0**] **A**;

**input** [**31**:**0**] **B**;

**output** [**31**:**0**] result;

**output** zero;

**wire** [**2**:**0**] alu\_control;

Alu\_Control alu\_ctrlr\_inst (

.opcode (opcode),

.func\_field (func\_field),

.alu\_control (alu\_control)

);

Alu\_Core alu\_core\_inst (

.**A** (**A**),

.**B** (**B**),

.alu\_control (alu\_control),

.result (result),

.zero (zero)

);

**endmodule**

The above verilog code implements an ALU using certain instructions from the MIPS Instruction set.

**Testbench:**

**module** Alu\_Top\_tb;

// Inputs

**reg** [**5**:**0**] opcode;

**reg** [**5**:**0**] func\_field;

**reg** [**31**:**0**] **A**;

**reg** [**31**:**0**] **B**;

// Outputs

**wire** [**31**:**0**] result;

**wire** zero;

// Instantiate the Unit Under Test (UUT)

Alu\_Top uut (

.opcode(opcode),

.func\_field(func\_field),

.**A**(**A**),

.**B**(**B**),

.result(result),

.zero(zero)

);

**initial** **begin**

// Initialize Inputs

opcode = **0**;

func\_field = **0**;

**A** = **0**;

**B** = **0**;

#**30**;

**A**=**32'h2222**; **B**=**32'h1111**;

opcode=**6'h00**;func\_field=**6'h20**;

#**30**;

opcode=**6'h00**;func\_field=**6'h24**;

#**30**;

opcode=**6'h23**;func\_field=**6'h00**;

#**30**;

**A**=**31'h5555**; **B**=**32'h5555**;

opcode=**6'h04**;func\_field=**6'h00**;

#**30**;

**A**=**32'h1111**; **B**=**32'h2222**;

opcode=**6'h00**;func\_field=**6'h2A**;

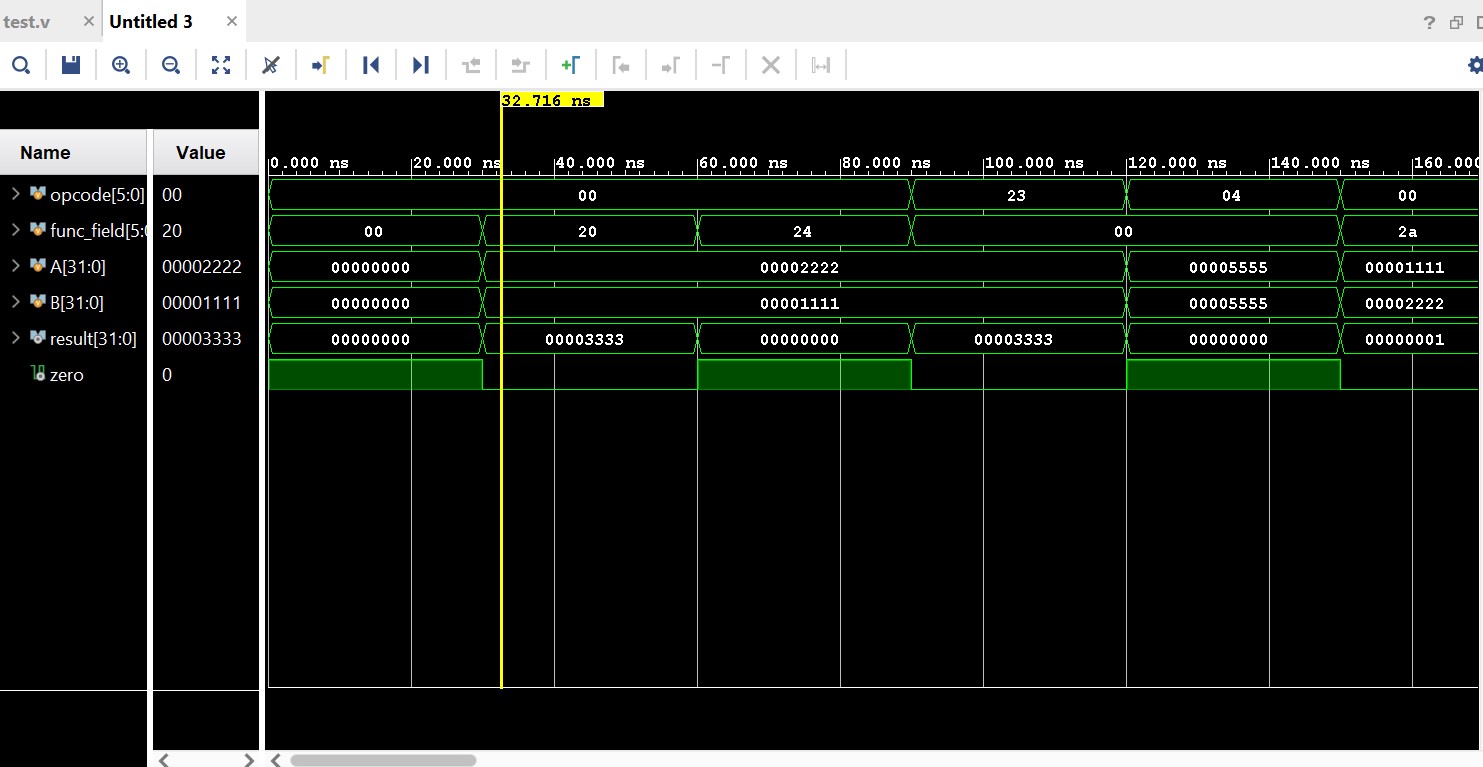
#**30**;

$finish;

**end**

**endmodule**

**Simulation Result:**



As can be seen from simulation,

Different values of opcode, function field, operands A and B are provided (as per our supported MIPS instructions) and the required results are obtained in the result field along with zero condition.

This ALU can be implemented in a simple MIPS instruction set based processor

A diagram of a computer

Description automatically generated