**Introduction**

The processor broadly consists of a Datapath and a Control Unit.

Datapath: refers to all the elements that work with data and process them.

Control Unit: generates control signals to direct the processor operation under different situations.

Difference between single clock cycle and pipelined execution:

All instructions do not take the same time to execute. Hence for single clock cycle execution, we need to ensure that the clock period is long enough to accommodate the execution of the slowest instruction.  
To speed up the execution process, the Pipelining technique is commonly used where multiple instructions are overlapped in execution. Pipelining however, is prone to different 'hazards' and must deal it.

Here, we have considered single clock cycle execution.

**Components of processor datapath:**

1. **Instruction Memory:**

* To store the instructions of a program.
* Given the address, it must supply the instruction located at that address.
* We can load instruction memory using readmemh function from a mem file.

**module** Instruction\_Memory(instrn\_address, instrn);

**input** [**31**:**0**] instrn\_address; //5-bit address holds 8 instructions of 32-bit width

**output** **wire** [**31**:**0**] instrn;

**reg** [**7**:**0**] instrn\_mem [**31**:**0**];

**initial** **begin**

$readmemh("instrn\_memory.mem", instrn\_mem); //load initial values

**end**

**assign** instrn = {instrn\_mem[instrn\_address+**3**],instrn\_mem[instrn\_address+**2**],

instrn\_mem[instrn\_address+**1**],instrn\_mem[instrn\_address]};

**endmodule**

**A diagram of instruction and instruction

Description automatically generated**

1. **Instruction Memory (instrn\_memory.mem)**

Consider the following assembly code that you want to simulate using this processor.  
(The number on the left represents instruction address)

00: add $t1, $t2, $t3          
04: lw $t1, $t2, 16'd4  
08: beq $t1, $t2, offset  
0C: add $t1, $t2, $t3  
10: or $t2, $t3, $t4  
14: sw $t1, $t2, offset

Totally we have 6 instructions. Now let us write them in the number code format based on the type of instruction (R or I-Type)

The code for each MIPS instruction is taken from the [MIPS instruction manual](https://s3-eu-west-1.amazonaws.com/downloads-mips/documents/MD00086-2B-MIPS32BIS-AFP-6.06.pdf).

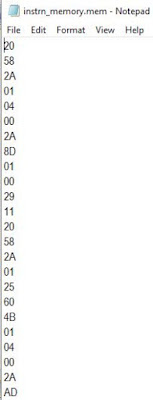
The code now becomes:

00: 6'd0,5'd9,5'd10,5'd11,5'd0,6'h20  
04: 6'h23,5'd9,5'd10,16'd4  
08: 6'h04,5'd9,5'd9,16'd1   
0C: 6'd0,5'd9,5'd10,5'd11,5'd0,6'h20  
10: 6'd0,5'd10,5'd11,5'd12,5'd0,6'h25  
14: 6'h2B,5'd9,5'd10,16'd4

Convert this to hexadecimal representation:

00: 01 2A 58 20  
04: 8D 2A 00 04  
08: 11 29 00 01  
0C: 01 2A 58 20  
10: 01 4B 60 25  
14: AD 2A 00 04

This is the data that we need to load in the instruction memory in a byte-wise manner.

[](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEh_PMt8DFFGE-rBYJnQ0Rf7Ik1X_wliH2cSDJJY3OqZeWX53Mp4p7pY54MR8ZFTxjS4o0gi_jgMKZTkA_IJ_QUUwcGvf_Jocr0XsjmeDPgXB-xChm1HZr-kD5QbW8x-c__HjQroI2_V1wMa/s531/instrn_memory_mem.JPG)

1. **Program Counter (PC):**

* Holds the address of the current instruction.
* For a 32-bit (4-byte) processor, we must increment the address by 4 to fetch the next instruction (as the width of each instruction is 4 bytes). This adder module will be connected to the PC.
* This address increment must happen at every clock cycle and hence, it will be a D Flip-Flop.
* A new instruction is executed every clock cycle.

**Verilog Code:**

**module** Program\_Counter(clk, rst\_n, in\_address, out\_address);

**input** clk, rst\_n;

**input** [**31**:**0**] in\_address;

**output** **reg** [**31**:**0**] out\_address;

**always** @ (**posedge** clk **or** **negedge** rst\_n)

**begin**

**if**(!rst\_n)

out\_address <= **32'd0**;

**else**

out\_address <= in\_address;

**end**

1. **Register File:**

* This module will enclose all the independent registers of the processor, to perform write and read operations. MIPS consists of 32 inbuilt registers.
* R-Format instructions have three operands. So, we will need to read 2 dwords(1dword = 4Bytes) from the register file (2 output read ports) and 1 write port (input port) along with a write enable signal that indicates when the data has to be written.
* Example Instruction: add $t1, $t2, $t3  
  To execute this instruction, we need to read two registers t1 and t2. Add them. Then write the result to register t3.
* In the Verilog code below, combinational read is done from the register memory using assign statement. But usually, read data will appear only after 1 clock cycle (flopped)
* We can load register memory using readmemh function from a mem file.

A diagram of a computer code

Description automatically generated

**Verilog Code:**

**module** Register\_File(clk, rst\_n, read\_addr1, read\_addr2, write\_en, write\_addr, write\_data, read\_data1,read\_data2);

**input** clk;

**input** rst\_n;

**input** [**4**:**0**] read\_addr1;

**input** [**4**:**0**] read\_addr2;

**input** write\_en;

**input** [**4**:**0**] write\_addr;

**input** [**31**:**0**] write\_data;

**output** **wire** [**31**:**0**] read\_data1;

**output** **wire** [**31**:**0**] read\_data2;

**reg** [**31**:**0**] reg\_mem [**31**:**0**];

**initial** **begin**

$readmemh("reg\_memory.mem", reg\_mem); //Load initial values

**end**

**assign** read\_data1 = reg\_mem[read\_addr1];

**assign** read\_data2 = reg\_mem[read\_addr2];

**always** @ (**posedge** clk **or** **negedge** rst\_n)

**begin**

**if** (!rst\_n)

**begin**

reg\_mem[write\_addr] <= reg\_mem[write\_addr];

**end**

**else**

**begin**

reg\_mem[write\_addr] <= write\_en ? write\_data : reg\_mem[write\_addr];

**end**

**end**

**endmodule**

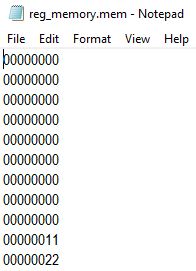
1. **Register Memory (reg\_memory.mem)**

This is the register memory where the memory location corresponds to the particular register address.

We are using the following registers $t1, $t2, $t3 and $t4 in our code which corresponds to the register addresses 09, 10, 11 and 12.

Let us write some initial data 11 in $t1 and 22 in $t2. (locations 09 and 10)

Then the reg\_memory.mem will look like this:

[](https://blogger.googleusercontent.com/img/b/R29vZ2xl/AVvXsEigrurNRAyisu7_mfpHRspFUtoiq7ILZmvCWI8s-GyaY2z3q5kDoIXhJaBMiThPk_TmPK8CA7Ga7QZGL_LMCYoftqU5-61IzntBJqzWO4YLMrA4-RR8JOUHCfH-N4xprUHF2uOJokv2CXNs/s271/reg_memory_mem.JPG)

1. **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.**

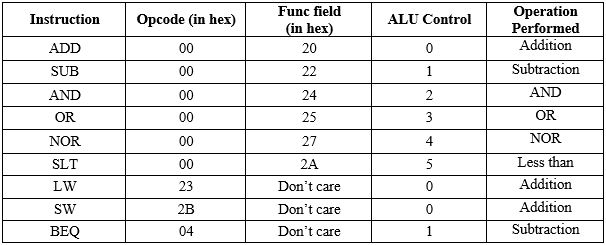
From the MIPS Instruction Set, ALU supports the following instructions:  
AND, OR, ADD, SUB, SLT, NOR, LW, SW, BEQ.

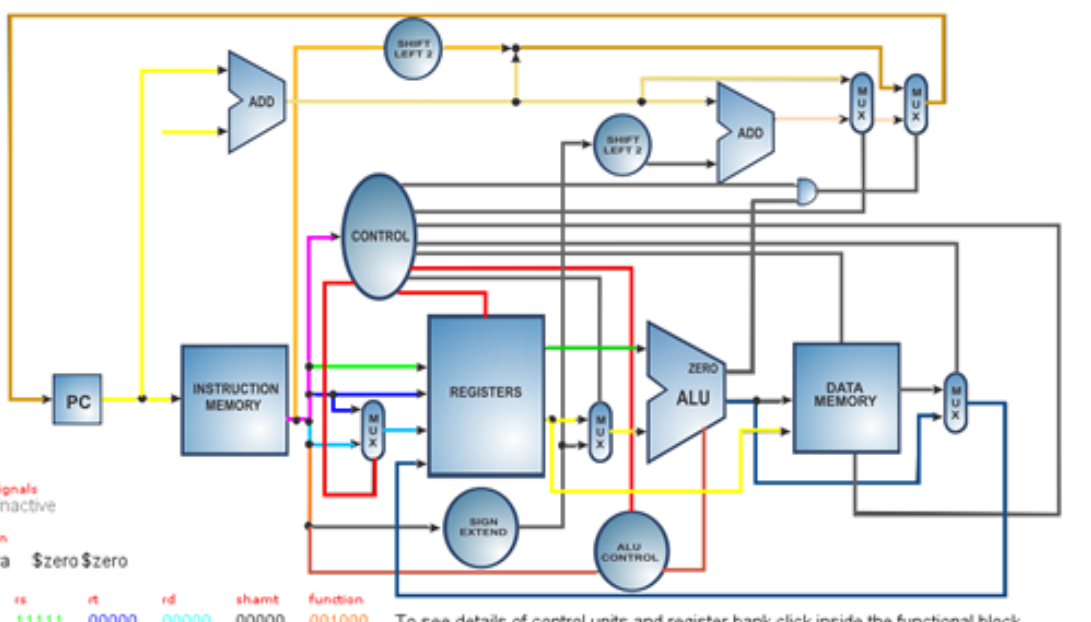
**Step 2: ALU Control Signal Generation**

* 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.

1. **Sign Extension Unit:**

The format of the lw and sw instructions is as follows.

* lw $t1, $t2, offset
* sw $t1, $t2, offset  
  where the offset\_value is a signed 16-bit value.

Since we are working with 32-bit values, we will need to sign extend the 16-bit offset value to bring it to 32-bits, and so we require a Sign Extension Unit.

**Verilog Code:**

**module** Sign\_Extension(

bits16\_in,

bits32\_out

);

**input** [**15**:**0**] bits16\_in;

**output** **wire** [**31**:**0**] bits32\_out;

**assign** bits32\_out = {{**16**{bits16\_in[**15**]}} , bits16\_in[**15**:**0**]};

**endmodule**