CS224 - Hardware Lab

Assignment 08

Lab Report

Group 4

230101012 - Annigandla Kameshwara Rao

230101041 - Grandhi Varshith

230101044 - Gunje Manideep Ram

230101067 - Muduguru Mahedar Reddy

Problem Statement:

Design a single cycle 32-bit MIPS implementation that includes a subset of the core MIPS instruction set

• The memory reference instructions: load word (lw) and store words (sw)

• The arithmetic-logical instructions: add, sub, and, or, and set-less-than (slt)

• Control transfer instructions: branch equal (beq) and jump (j)

• Instruction for supporting subroutine: jump and link (jal)

[Note: Refer to the P&H book for detailed instruction formats and meaning]

Design the data-path (Instruction memory (only read access), Data memory (read and write), Register file (32 numbers of 32 bit registers), ALUs, MUXes, Shifter, and Sign Extender, Program counter etc), and controller of the processor (executes one instruction in a single cycle).

Implement all data-path units as separate Verilog modules to design the processor datapath in modular fashion. To test, set a specific value to the data and instruction memories and execute the instruction.

Behavioral level implementation is not allowed in this Assignment. Note that the instruction set is complete i.e., any meaningful program can be written with these 10 instructions. The test cases must have some meaningful programs covering usage of all instructions. No FPGA or breadboard level demo is needed. Only simulation results will be sufficient.

Process followed to obtain the design:

The design and implementation of a simplified MIPS processor is done using Verilog. The MIPS processor supports a subset of instructions, including arithmetic, logical, memory access, and control flow operations. The design

follows a modular approach, the modules for distinct components such as the ALU, control unit, datapath, and memory modules were written.

1. ALU (Arithmetic Logic Unit)

Functionality:

The ALU performs arithmetic and logical operations based on a 3-bit function selector (f_in). It supports the following operations:

- **AND** (f_in = 3'b000)
- **OR** (f_in = 3'b001)
- **ADD** (f_in = 3'b010)
- **SUBTRACT** (f_in = 3'b110)
- **SET LESS THAN** (f_in = 3'b111)

Two's Complement of b_in: For subtraction and set-less-than operations, the second operand (b_in) is conditionally inverted to facilitate two's complement arithmetic, it is done by checking whether f_in[2]==1.

Operation Selection: The operation is selected based on the lower two bits of f_in (f_in[1:0]), with the third bit determining whether to subtract or perform set less than operation by considering 2's complement.

Output Zero : The zero output is asserted when the result (y_out) is zero, which is helpful in branch decisions.

2. ALU Control Unit

Functionality

The ALU Control Unit translates the ALUOp and function field (FnField) from the instruction into a specific control signal (AluCtrl) for the ALU.

Implementation Details

- **Instruction Decoding**: Based on the combination of AluOp and FnField, the unit determines the required ALU operation.
- **Control Signal Mapping**:
  - AluOp = 00: Load/Store operations → AluCtrl = 010 (ADD)

- AluOp = 01: Branch equal → AluCtrl = 110 (SUBTRACT)
- AluOp = 10: R-type instructions → Decoded using FnField

3. Multiplexer Module

Functionality

The multiplexer (mux) selects between two inputs (ina and inb) based on a select signal (sel).

If sel==0 then ina is selected, else inb is selected.

Details of logic:

- We call the mux# depending of our requirement of input by the parameter DATA_LENGTH whose default value is 8.
- We can avoid defining multiple mux for different input sizes.

4. Program Counter Logic

Functionality

The pclogic module manages the Program Counter (PC), determining the next instruction address based on control signals.

Implementation Details

- **Reset Handling**: On reset, the PC is initialized to zero.
- **Jump and Branch Handling**:
  - **Jump**: If jump is asserted, the PC is set to the target address (ain).
  - **Branch**: If pcsel is 1, indicating a branch condition is met, the PC is updated to aout + ain + 1.
  - **Next instruction Execution(Sequential)**: If neither jump nor pcsel is asserted, then PC increments by 1.

5. Register File

Functionality

The registerfile module provides storage for 32 general-purpose 32-bit registers, supporting two read ports and one write port.

Implementation Details

- **Read Operations**: Registers ra and rb are read, with outputs da and db. Reading register 0 always returns 0.
- **Write Operations**: On the rising edge of the clock, if regWrite is 1, data (dc) is written to register rc.

6. Sign Extension

Functionality

The sign_extend module extends a 16-bit immediate value to a 32-bit signed value, preserving the sign bit.

Implementation Details

- **Sign Preservation**: The most significant bit (MSB) of the 16-bit input (idata[15]) is replicated 16 times to fill the upper bits of the 32-bit output (odata).

7. Data Memory

Functionality

The dmem module simulates data memory, supporting read and write operations.

Implementation Details

- **Memory Array**: A 32-bit wide memory array with 64 entries is defined.
- **Read Operation**: When read_enable is asserted, the data at the specified address (addr) is output on rdata.
- **Write Operation**: On the rising edge of the clock, if write_enable is 1, wdata is written to the specified address.

8. Instruction Memory

Functionality

The imem module simulates instruction memory, preloaded with a program that initializes an array and computes its sum.

Implementation Details

- **Memory Initialization**: The memory is initialized with specific instructions, including:
  - **Array Initialization**: Storing values 0 to 4 at consecutive memory locations.
  - **Sum Computation**: Looping through the array to compute the sum.
  - **Result Storage**: Storing the computed sum at a specific memory location.

9. Datapath

Functionality

The datapath module integrates all components, facilitating instruction fetch, decode, execution, memory access, and write-back stages.

Implementation Details

- **Instruction Fetch**: The PC address (PC_adr) is used to fetch instructions from imem.
- **Instruction Decode**: The opcode and function fields are extracted for control signal generation.
- **Execution**:
  - **ALU Operations**: Operands are selected via multiplexers, and the ALU performs the required operation.
  - **Zero Flag**: Used to determine branch decisions.
- **Memory Access**: Depending on control signals, data is read from or written to dmem.
- **Write-Back**: Results are written back to the register file based on control signals.

10. Control Path

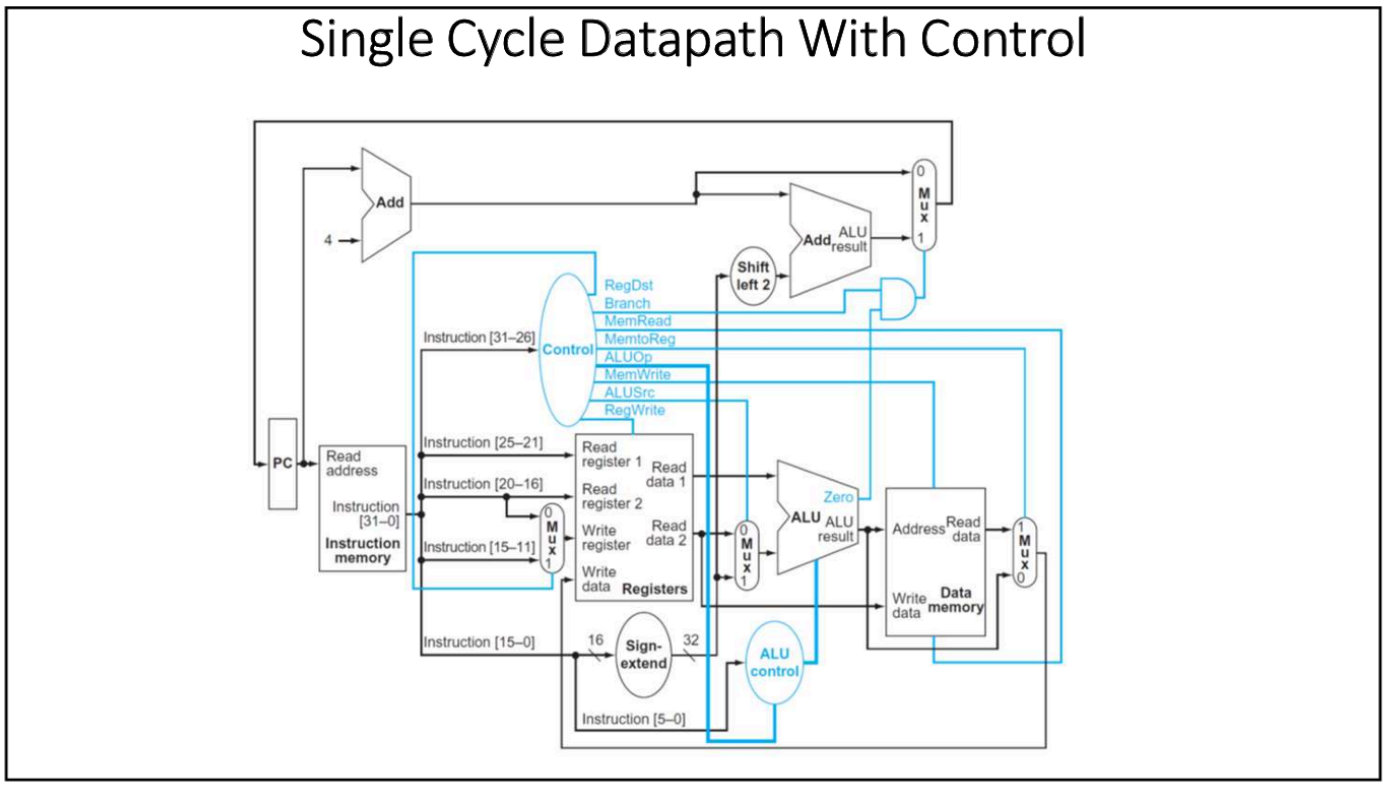
Functionality

The controlpath module generates control signals based on the opcode of the fetched instruction.

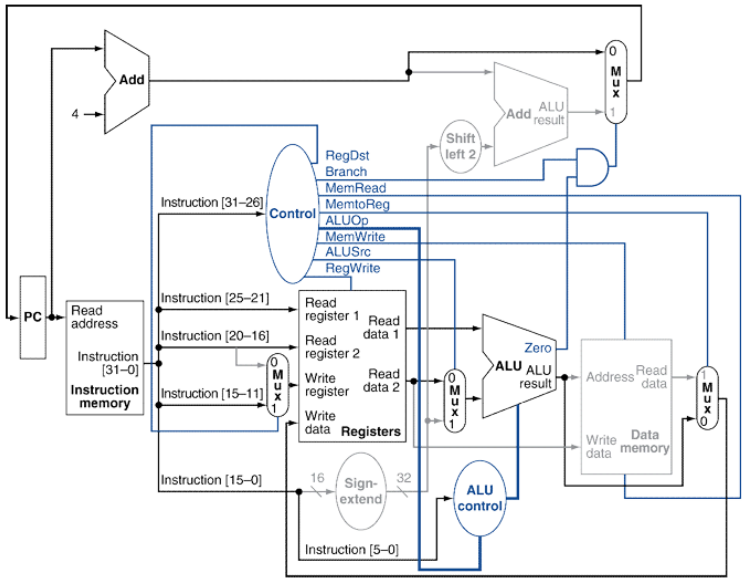
Implementation Details

- **Control Signal Generation**: For each opcode, the module sets the appropriate control signals to guide the datapath.
- **Instruction Support**:
  - **R-type Instructions**: ALU operations determined by the function field.
  - **Load/Store Instructions**: Memory access operations.
  - **Branch Instructions**: Conditional branching based on ALU zero flag.

Diagram Of the Design

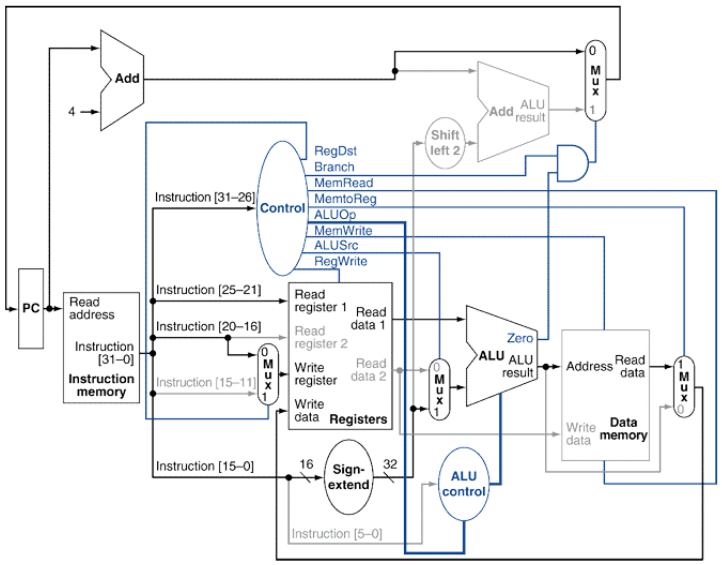


R-Type Instruction



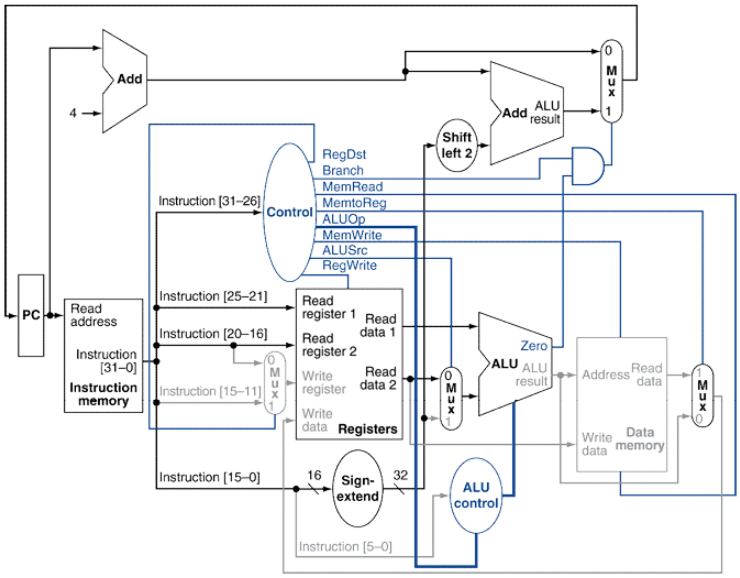
Input or output	Signal name	R-format
Inputs	Op5	0
	Op4	0
	Op3	0
	Op2	0
	Op1	0
	Op0	0
Outputs	RegDst	1
	ALUSrc	0
	MemtoReg	0
	RegWrite	1
	MemRead	0
	MemWrite	0
	Branch	0
	ALUOp1	1
	ALUOp0	0

Store and Load Instruction



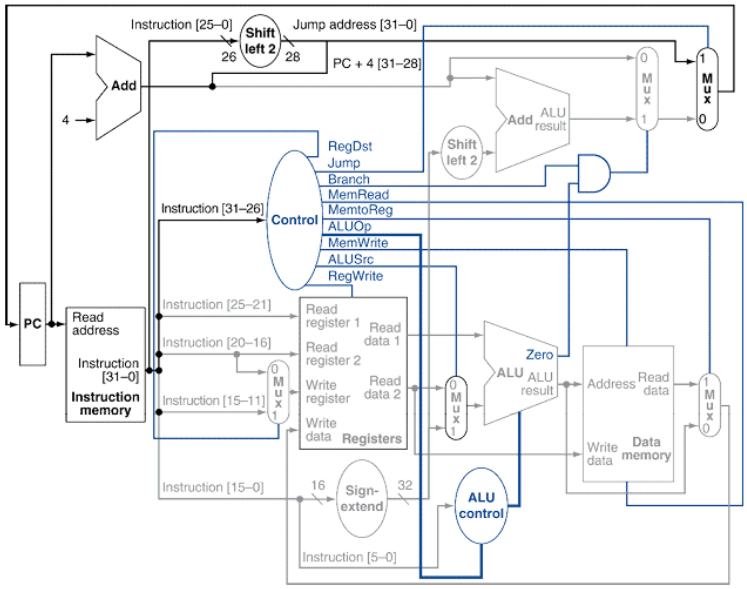
Input or output	Signal name	lw	sw
Inputs	Op5	1	1
	Op4	0	0
	Op3	0	1
	Op2	0	0
	Op1	1	1
	Op0	1	1
Outputs	RegDst	0	X
	ALUSrc	1	1
	MemtoReg	1	X
	RegWrite	1	0
	MemRead	1	0
	MemWrite	0	1
	Branch	0	0
	ALUOp1	0	0
	ALUOp0	0	0

Branch-on-Equal Instruction



Input or output	Signal name	beq
Inputs	Op5	0
	Op4	0
	Op3	0
	Op2	1
	Op1	0
	Op0	0
Outputs	RegDst	X
	ALUSrc	0
	MemtoReg	X
	RegWrite	0
	MemRead	0
	MemWrite	0
	Branch	1
	ALUOp1	0
	ALUOp0	1

Datapath With Unconditional Jumps



2	address
31:26	25:0

Function

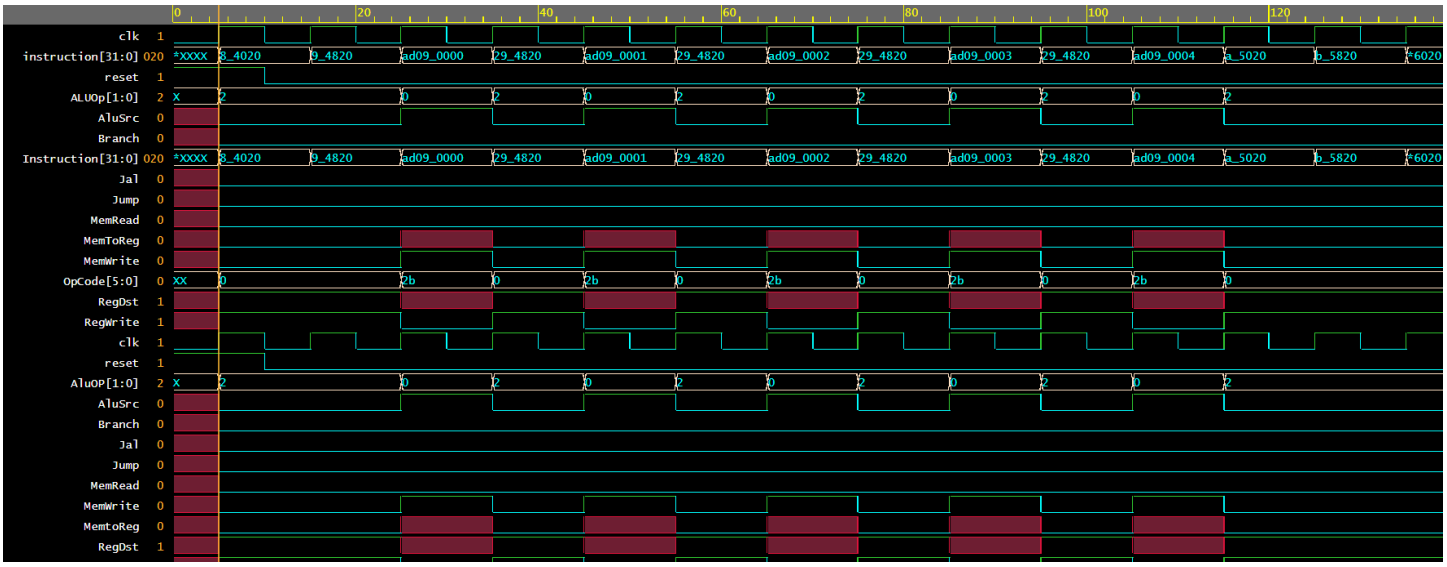
C:

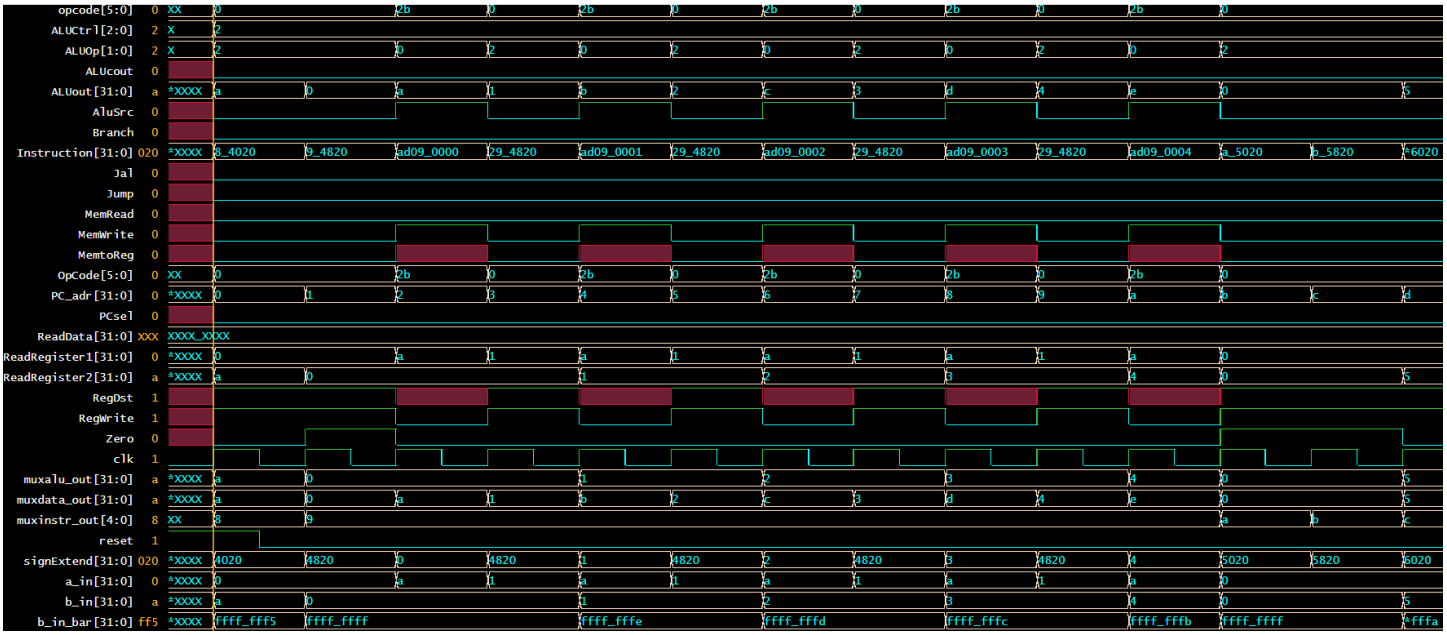
```c
int main() {
    int array[5] = {0, 1, 2, 3, 4};  // Array in memory
    int sum = 0;
    for (int i = 0; i < 5; i++) {
        sum += array[i];  // Load array[i]
    }
    array[0] = sum;       // Store sum back to array[0]
    return sum;
}
```

Assembly:

```
main:
    # Initialize array in memory (address 0x1010)
    addi $t0, $0, 0x100    # Base address = 0x1010
    addi $t1, $0, 1        # array[0] = 0
    sw   $t1, 0($t0)
    addi $t1, $0, 2        # array[1] = 1
    sw   $t1, 4($t0)
    addi $t1, $0, 3        # array[2] = 2
    sw   $t1, 8($t0)
    addi $t1, $0, 4        # array[3] = 3
    sw   $t1, 12($t0)
    addi $t1, $0, 5        # array[4] = 4
    sw   $t1, 16($t0)

    # Sum loop
    addi $t2, $0, 0        # sum = 0
    addi $t3, $0, 0        # i = 0
    addi $t4, $0, 5        # loop limit = 5

loop:
    slt  $t5, $t3, $t4     # i < 5?
    beq  $t5, $0, exit     # Exit if i >= 5
    lw   $t6, 0($t0)       # Load array[i]
    add  $t2, $t2, $t6     # sum += array[i]
    addi $t0, $t0, 4       # Move to next array element
    addi $t3, $t3, 1       # i++
    j    loop

exit:
    sw   $t2, 0x1010($0)    # Store sum back to array[0] (address 0x1010)
```

Simulations


0
20
40
60
80
100
120
clk 1
instruction[31:0] 020
reset 1
ALUOp[1:0] 2
AluSrc 0
Branch 0
Instruction[31:0] 020
Jal 0
Jump 0
MemRead 0
MemToReg 0
MemWrite 0
OpCode[5:0] 0
RegDst 1
RegWrite 1
clk 1
reset 1
AluOP[1:0] 2
AluSrc 0
Branch 0
Jal 0
Jump 0
MemRead 0
MemWrite 0
MemtoReg 0
RegDst 1


opcode[5:0] 0
ALUCtrl[2:0] 2
ALUOp[1:0] 2
ALUcout 0
ALUout[31:0] a
AluSrc 0
Branch 0
Instruction[31:0] 020
Jal 0
Jump 0
MemRead 0
MemWrite 0
MemtoReg 0
OpCode[5:0] 0
PC_adr[31:0] 0
PCsel 0
ReadData[31:0] XXX
ReadRegister1[31:0] 0
ReadRegister2[31:0] a
RegDst 1
RegWrite 1
Zero 0
clk 1
muxalu_out[31:0] a
muxdata_out[31:0] a
muxinstr_out[4:0] 8
reset 1
signExtend[31:0] 020
a_in[31:0] 0
b_in[31:0] a
b_in_bar[31:0] ff5