A Simplified MIPS Processor in Verilog

Data Memory

- `module DM(MemRead, MemWrite, ABUS, DIN, DATABUS);`
  - `MemWrite`: Nothing happens if 0. If 1, the memory at location `ABUS` will be written with `DIN`.
  - `ABUS`: At any moment, the data at location `ABUS` will appear at `DATABUS`.
  - `MemRead`: Not used.

Data Memory

- Address bus: 8 bits.
- Data bus: 32 bits. Each memory location holds 32 bits.

Data Memory

- Init the contents in data memory with any value you like:

```
initial
 begin
     for (i=0; i <= DM_ADDR_MAX_m1; i = i + 1)
       ram[i] = i*10 + 1;
 end
```

Instruction Memory

- `module IM(CSB,WRB,ABUS,DATABUS);`
  - `CSB:` chip select. If 0, selected. If 1, not selected.
  - `WRB:` Not used.
  - `ABUS:` Address bus. At any moment, if chip is selected, the data at location `ABUS` will appear at `DATABUS`.

Instruction Memory

- Address bus: 8 bits.
- Data bus: 32 bits. Each memory location holds 32 bits.

Instruction Memory

- The most straightforward way of loading a program:

```
ram[0]  = 32'b00100000000000000000000000000000;  // addi $0, $0, 0
ram[1]  = 32'b00100000001000010000000000000001;  // addi $1, $1, 1
ram[2]  = 32'b00100000010000100000000000000010;  // addi $2, $2, 2
ram[3]  = 32'b00100000011000110000000000000011;  // addi $3, $3, 3
ram[4]  = 32'b00000000000000000000000000000000;  // nop
ram[5]  = 32'b00000000000000000000000000000000;  // nop
ram[6]  = 32'b00000000000000000000000000000000;  // nop
ram[7]  = 32'b00000000000000000000000000000000;  // nop
ram[8]  = 32'b00000000000000000000000000000000;  // nop
ram[9]  = 32'b00000000000000000000000000000000;  // nop
```

Get The Next PC

```
module getNextPC (PCSrc, currPC, offset, out);
  parameter MIPS_PC_WIDTH_m1 = 7;
  input PCSrc;
  input [MIPS_PC_WIDTH_m1:0] offset;
  input [MIPS_PC_WIDTH_m1:0] currPC;
  output reg [MIPS_PC_WIDTH_m1:0] out;

  always @(PCSrc, currPC, offset)
     if (PCSrc == 0)
          out <= currPC + 1;
     else
          out <= currPC + 1 + offset;
  endmodule
```

PC

- `module MIPSPC(clk, newPC, PC);`
- Just an 8-bit D-flip-flop.

Register File

- `module MIPSREG(clk, RegWrite, ReadAddr1, ReadAddr2, WriteAddr, ReadData1, ReadData2, WriteData);`
  - Just as what specified in the book.
  - `RegWrite`: If 0, disabling write. If 1, register `WriteAddr` register `WriteAddr` will be overwritten with `WriteData`.
  - At any time, `ReadData1` is the content of reg `ReadAddr1`, and `ReadData2` is the content of reg `ReadAddr2`.

ALU

```
module MIPSALU (ALUctl, A, B, ALUOut, Zero);
input [3:0] ALUctl;
input [31:0] A,B;
output reg [31:0] ALUOut;
output Zero;
assign Zero = (ALUOut==0); //Zero is true if ALUOut is 0; goes
   anywhere
always @(ALUctl, A, B) //reevaluate if these change
case (ALUctl)
0: ALUOut <= A & B;
1: ALUOut <= A | B;
2: ALUOut <= A + B;
6: ALUOut <= A - B;
7: ALUOut <= A < B ? 1:0;
12: ALUOut <= ~(A | B); // result is nor
default: ALUOut <= 0; //default to 0, should not happen;
endcase
endmodule
```

Sign Extension

```
module SignExtend (in, out);
   input [15:0] in;
   output [31:0] out;

   assign out[15:0] = in[15:0];
   assign out[31:16] = in[15];
endmodule
```

Two-to-one Selector

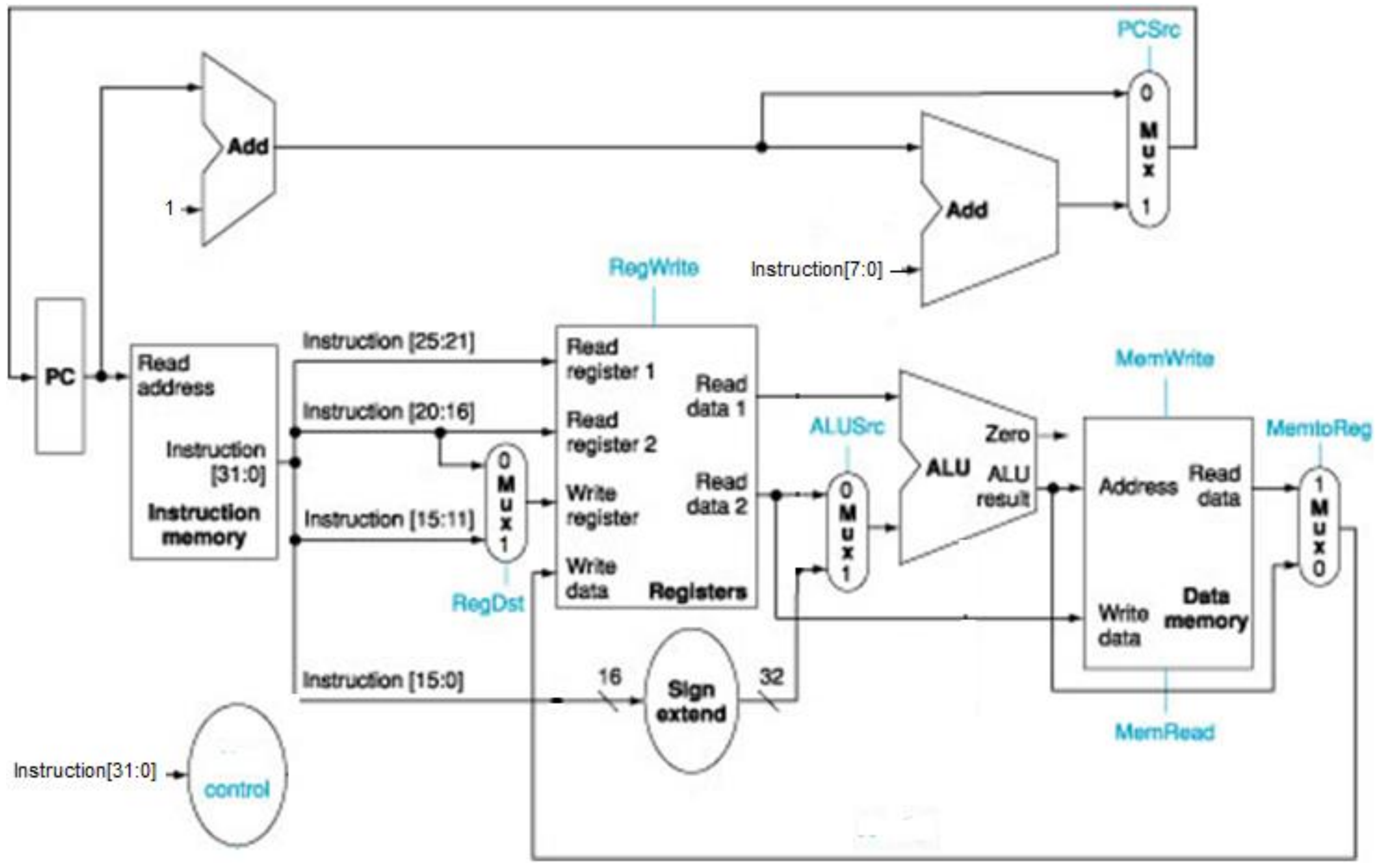
```
module STwoToOne32 (sel, in0, in1, out);
  input sel;
  input [31:0] in0, in1;
  output reg [31:0] out;

  always @(sel, in0, in1)
     if (sel == 0)
          out <= in0;
     else
          out <= in1;
endmodule
```

Control

- `module MIPSCtrl (instr, RegDst, ALUSrc, MemToReg, RegWrite, MemWrite, MemRead, branch, ALUCtrl);`
- Take the 32-bit instruction, generate the control signals.

Data Path Setup According to



Supported instructions

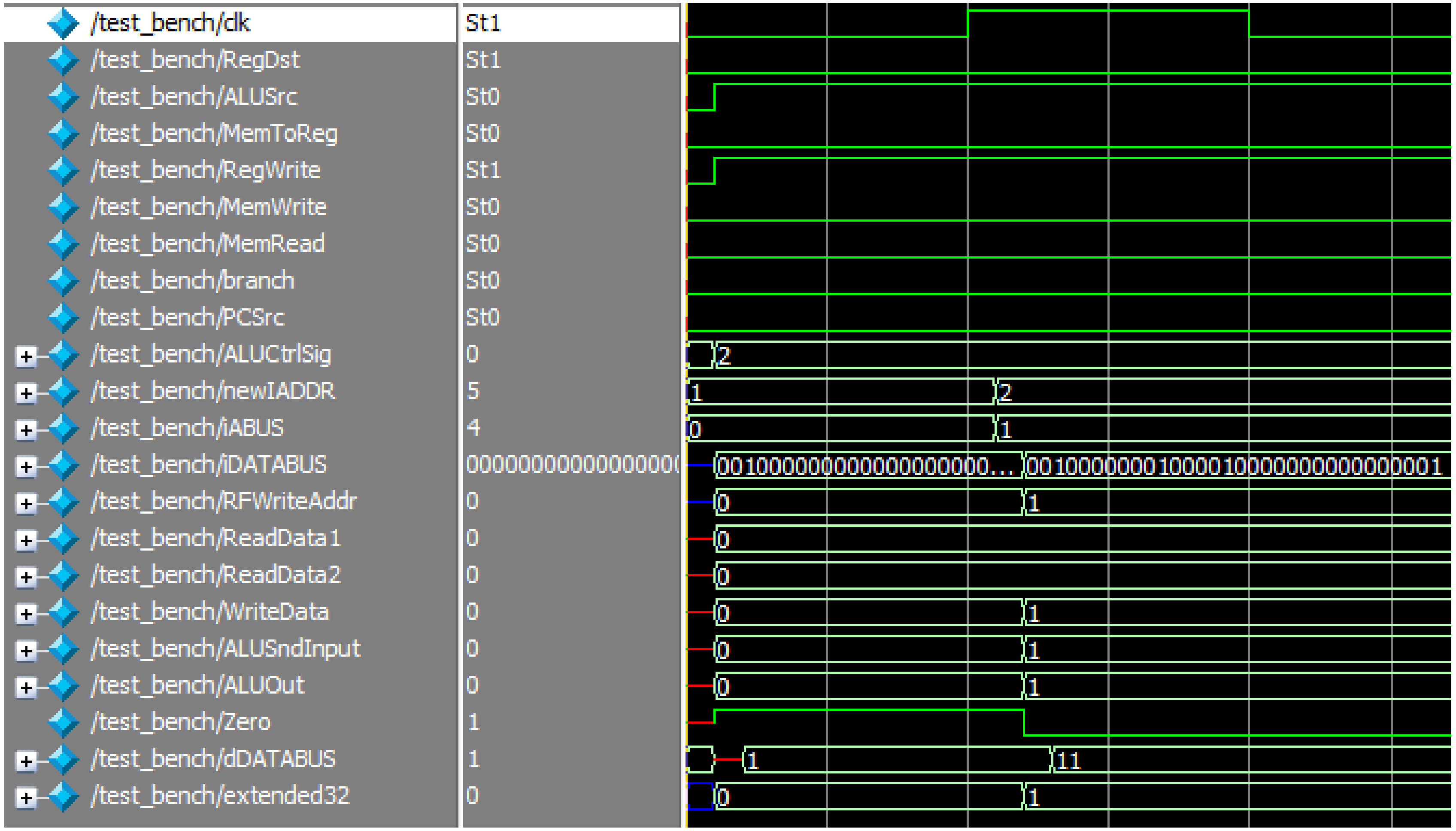
- add, sub, addi, lw, sw, beq.

Stepping through the program

- The clock is set to be 200 time units. Run 200 every time.
- Check the values of the wires in the waveform
- Check the content of the registers and data memories in the memory windows.

The waveform after two instructions

```
00100000000000000000000000000000;   // addi $0, $0, 0
00100000001000010000000000000001;   // addi $1, $1, 1
```



/test_bench/clk	St1
/test_bench/RegDst	St1
/test_bench/ALUSrc	St0
/test_bench/MemToReg	St0
/test_bench/RegWrite	St1
/test_bench/MemWrite	St0
/test_bench/MemRead	St0
/test_bench/branch	St0
/test_bench/PCSrc	St0
/test_bench/ALUCtrlSig	0
/test_bench/newIADDR	5
/test_bench/iABUS	4
/test_bench/iDATABUS	0000000000000000(
/test_bench/RFWriteAddr	0
/test_bench/ReadData1	0
/test_bench/ReadData2	0
/test_bench/WriteData	0
/test_bench/ALUSndInput	0
/test_bench/ALUOut	0
/test_bench/Zero	1
/test_bench/dDATABUS	1
/test_bench/extended32	0

Supporting a new instruction

- With the current data path, can we support a new instruction?

  `lwr $rs, $rt, $rd`
- What it does is to read from data memory at \$rs-\$rt, and write it to \$rd
- Suppose its opcode is 000001.

Supporting a new instruction

```
else if (instr[31:26] == 6'b000001) //lwr
  begin
    RegDst <= 1;
    ALUSrc <= 0;
    MemToReg <= 1;
    RegWrite <= 1;
    MemRead <= 1;
    MemWrite <= 0;
    branch <= 0;
    ALUCtrl <= 4'b0110;
  end
```