

# Organization of Digital Computers Lab

## EECS 112L

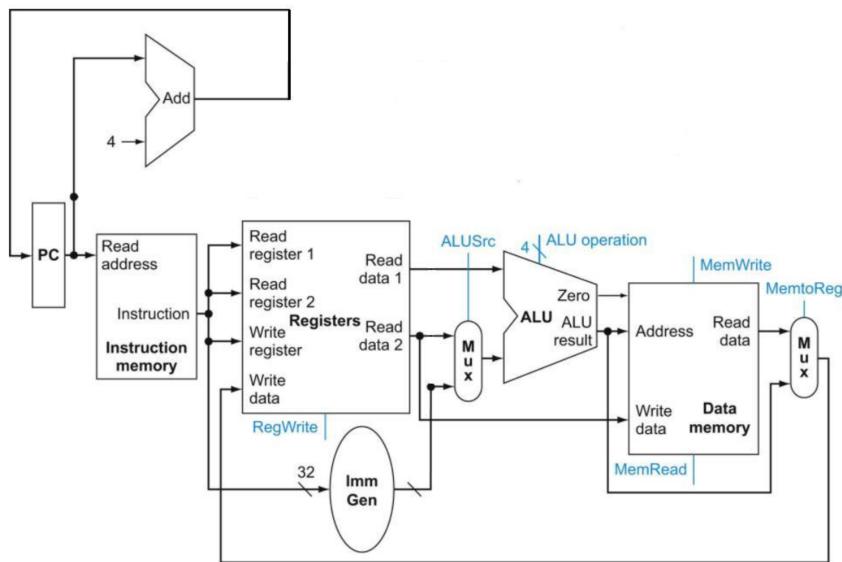
### Lab 2: Single Cycle RISC-V datapath and control

University of California, Irvine

Due date: April 22, 2019

## 1 RISC-V datapath

System-Verilog code for RISC-V processor is given to you. According to the module definition, RISC-V is the top module which includes Controller, ALUController, and Datapath. The provided code for datapath implements the following figure:



RISC-V base instruction formats are as follows extracted from [1]:

31	25 24	20 19	15 14	12 11	7 6	0	
funct7	rs2	rs1	funct3	rd	opcode		R-type
imm[11:0]		rs1	funct3	rd	opcode		I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode		S-type
imm[31:12]			rd	opcode			U-type

Figure 1: RISC-V base instruction formats. Each immediate subfield is labeled with the bit position (imm[x]) in the immediate value being produced, rather than the bit position within the instruction's immediate field as is usually done.

The RISC-V ISA keeps the source rs1 and rs2 and destination register rd in the same position in all the formats. Also, there are a further two variants of the instruction formats (B/J) based on the handling of immediate. RISC-V base instruction formats including immediate variants are represented below:

31	30	25 24	21	20	19	15 14	12 11	8	7	6	0	
funct7		rs2		rs1		funct3		rd		opcode		R-type
imm[11:0]			rs1	funct3		rd		opcode				I-type
imm[11:5]			rs2	rs1		funct3		imm[4:0]				S-type
imm[12]	imm[10:5]			rs2	rs1		funct3	imm[4:1]	imm[11]	opcode		B-type
imm[31:12]						rd		opcode				U-type
imm[20]	imm[10:1]	imm[11]	imm[19:12]		rd		opcode					J-type

To add all the instructions of different types to the datapath, the first step is to modify the Immediate Generator unit. According to the following figure, Imm Gen produces the 32-bit output differently according to the type of instruction. The current code for the Imm Gen only generates output for I-type and S-type instructions.

31	30	20 19	12	11	10	5	4	1	0			
— inst[31] —					inst[30:25]	inst[24:21]	inst[20]	I-immediate				
— inst[31] —					inst[30:25]	inst[11:8]	inst[7]	S-immediate				
— inst[31] —					inst[7]	inst[30:25]	inst[11:8]	0	B-immediate			
inst[31]	inst[30:20]	inst[19:12]	— 0 —					U-immediate				
— inst[31] —			inst[19:12]	inst[20]	inst[30:25]	inst[24:21]	0	J-immediate				

The following table has the values control signals for some of the instructions which might be useful for you.

Input or output	Signal name	R-format	Id	sd	beq
Inputs	I[6]	0	0	0	1
	I[5]	1	0	1	1
	I[4]	1	0	0	0
	I[3]	0	0	0	0
	I[2]	0	0	0	0
	I[1]	1	1	1	1
	I[0]	1	1	1	1
Outputs	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

## 2 Integer Computational Instructions

Integer computational instructions are either encoded as register-immediate operations using the I-type format or as register-register operations using the R-type format. No integer computational instructions cause arithmetic exceptions.

## 3 Integer Register-Immediate Instructions

ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic overflow is ignored. SLTI (set less than immediate) places the value 1 in register rd if register rs1 is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to rd. SLTIU is similar but compares the values as unsigned numbers.

ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register rs1 and the sign-extended 12-bit immediate and place the result in rd.

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM	
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM	

Shifts by a constant are encoded as a specialization of the I-type format. The operand to be shifted is in rs1, and the shift amount is encoded in the lower 5 bits of the I-immediate field. The right shift type is encoded in a high bit of the I-immediate. SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic

right shift (the original sign bit is copied into the vacated upper bits).

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	imm[4:0]	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	shamt[4:0]	src	SLLI	dest	OP-IMM	
0000000	shamt[4:0]	src	SRLI	dest	OP-IMM	
0100000	shamt[4:0]	src	SRAI	dest	OP-IMM	

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest 12 bits with zeros. AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC forms a 32-bit offset from the 20-bit U-immediate, filling in the lowest 12 bits with zeros, adds this offset to the pc, then places the result in register rd.

31	12 11	7 6	0
imm[31:12]	rd	opcode	
20	5	7	
U-immediate[31:12]	dest	LUI	
U-immediate[31:12]	dest	AUIPC	

## 4 Integer Register-Register Instructions

RV32I defines several arithmetic R-type operations. All operations read the rs1 and rs2 registers as source operands and write the result into register rd. The funct7 and funct3 fields select the type of operation.

ADD and SUB perform addition and subtraction respectively. Over flows are ignored. SLT and SLTU perform signed and unsigned compares respectively. OR, and XOR perform bitwise logical operations. SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	
7	5	5	3	5	7	
0000000	src2	src1	ADD/SLT/SLTU	dest	OP	
0000000	src2	src1	AND/OR/XOR	dest	OP	
0000000	src2	src1	SLL/SRL	dest	OP	
0100000	src2	src1	SUB/SRA	dest	OP	

## 5 Control Transfer Instructions

### 5.1 Unconditional Branches

The jump and link (JAL) instruction uses the J-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Plain unconditional jumps (assembler pseudo-op J) are encoded as a JAL with rd=x0.

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd.

31	30	21	20	19	12 11	7 6	0
imm[20]	imm[10:1]	imm[11]	imm[19:12]		rd	opcode	
1	10	1	8		5	7	
		offset[20:1]			dest	JAL	

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	
offset[11:0]	base	0	dest	JALR	

## 5.2 Conditional Branches

All branch instructions use the B-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2, and is added to the current pc to give the target address. The conditional branch range is 4 KiB.

Branch instructions compare two registers. BEQ and BNE take the branch if registers rs1 and rs2 are equal or unequal respectively. BLT and BLTU take the branch if rs1 is less than rs2, using signed and unsigned comparison respectively. BGE and BGEU take the branch if rs1 is greater than or equal to rs2, using signed and unsigned comparison respectively.

31	30	25 24	20 19	15 14	12 11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode		
1	6	5	5	3	4	1		7	
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH		
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH		
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH		

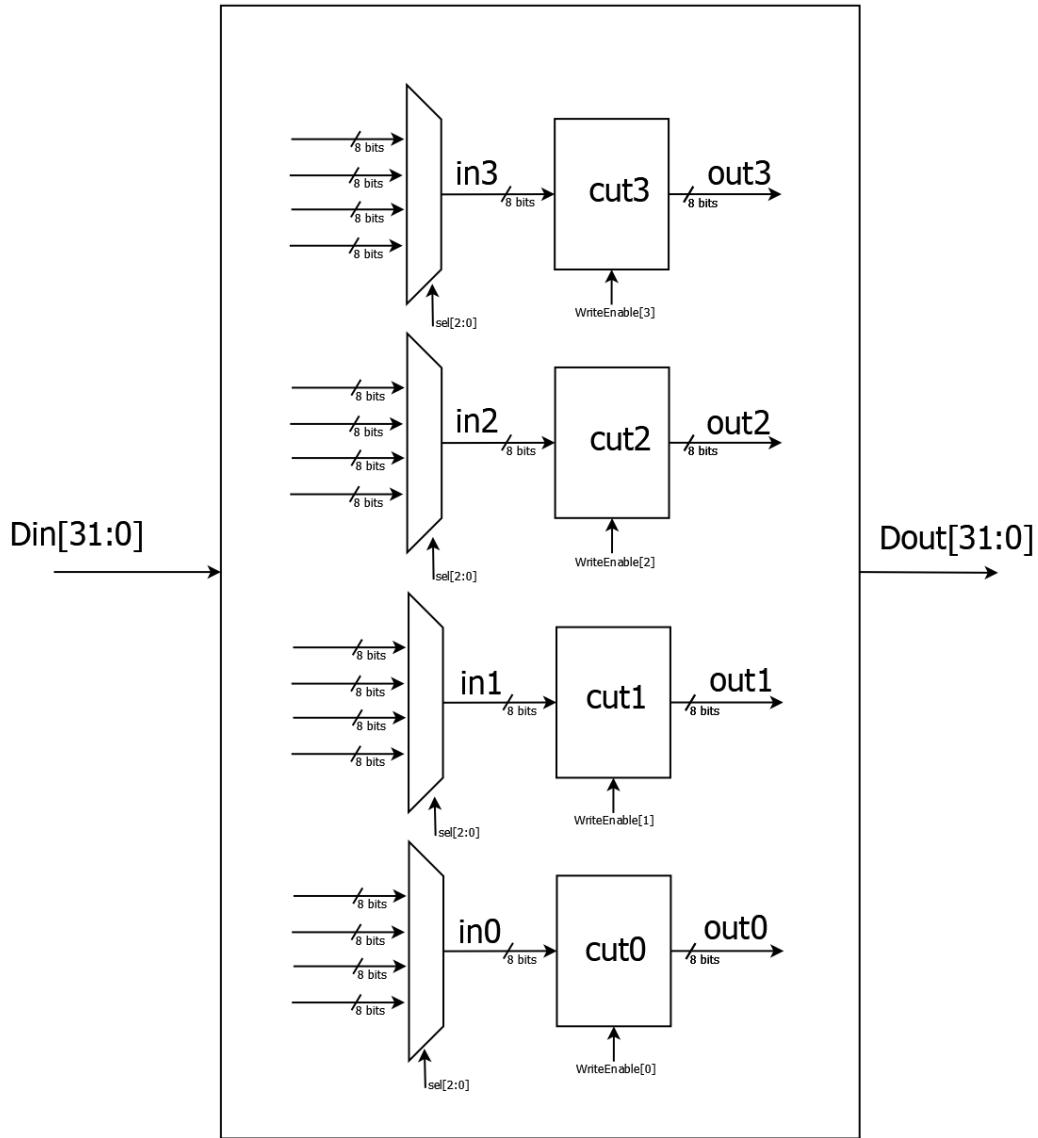
## 6 Load and Store Instructions

The provided RISC-V processor implements Load and Store instructions. Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd.

Stores copy the value in register rs2 to memory. The LW instruction loads a 32-bit value from memory into rd. LH loads a 16-bit value from memory, then sign-extends to 32-bits before storing in rd. LHU loads a 16-bit value from memory but then zero extends to 32-bits before storing in rd. LB and LBU are defined analogously for 8-bit values. The SW, SH, and SB instructions store 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory.

In order to support Byte/Half word load store operation, follow the following image and tables:

# Memory



Instruction	address[1:0]	Dout[31:24]	Dout[23:16]	Dout[15:8]	Dout[7:0]
LW	xx	out3	out2	out1	out0
LH	x0	sign extend		out1	out0
LH	x1	out3	out2	zeros	
LB	00	sign extend			out0
LB	01	sign extend		out1	zeros
LB	10	sign extend	out2	zeros	
LB	11	out3		zeros	

Instruction	address[1:0]	WriteEnable[3:0]	in3	in2	in1	in0
SW	xx	1111	Din[31:24]	Din[23:16]	Din[15:8]	Din[7:0]
SH	x0	0011			Din[15:8]	Din[7:0]
SH	x1	1100	Din[15:8]	Din[7:0]		
SB	00	0001				Din[7:0]
SB	01	0010			Din[7:0]	
SB	10	0100		Din[7:0]		
SB	11	1000	Din[7:0]			

The table below summarizes the newly added instructions with the desired opcodes.

RV32I Base Instruction Set					
imm[31:12]			rd	0110111	LUI
imm[31:12]			rd	0010111	AUIPC
imm[20:10:1 11 19:12]			rd	1101111	JAL
imm[11:0]	rs2	rs1	000	rd	JALR
imm[12:10:5]		rs1	000	imm[4:1 11]	BEQ
imm[12:10:5]		rs1	001	imm[4:1 11]	BNE
imm[12:10:5]		rs1	100	imm[4:1 11]	BLT
imm[12:10:5]		rs1	101	imm[4:1 11]	BGE
imm[12:10:5]		rs1	110	imm[4:1 11]	BLTU
imm[12:10:5]		rs1	111	imm[4:1 11]	BGEU
imm[11:0]		rs1	000	rd	LB
imm[11:0]		rs1	001	rd	LH
imm[11:0]		rs1	010	rd	LW
imm[11:0]		rs1	100	rd	LBU
imm[11:0]		rs1	101	rd	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	SW
imm[11:0]		rs1	000	rd	ADDI
imm[11:0]		rs1	010	rd	SLTI
imm[11:0]		rs1	011	rd	SLTIU
imm[11:0]		rs1	100	rd	XORI
imm[11:0]		rs1	110	rd	ORI
imm[11:0]		rs1	111	rd	ANDI
0000000	shamt	rs1	001	rd	SLLI
0000000	shamt	rs1	101	rd	SRLI
0100000	shamt	rs1	101	rd	SRAI
0000000	rs2	rs1	000	rd	ADD
0100000	rs2	rs1	000	rd	SUB
0000000	rs2	rs1	001	rd	SLL
0000000	rs2	rs1	010	rd	SLT
0000000	rs2	rs1	011	rd	SLTU
0000000	rs2	rs1	100	rd	XOR
0000000	rs2	rs1	101	rd	SRL
0100000	rs2	rs1	101	rd	SRA
0000000	rs2	rs1	110	rd	OR
0000000	rs2	rs1	111	rd	AND

## 7 Material to be submitted

### 7.1 Report:

Your reports should be in pdf format. You can use any text editor to write, however Latex is preferred. If your report is written in LateX, the tex and pdf files (both) are expected to be inside the folder you submit. The report should contain your name and student ID (on the cover page).

- You report should include block diagram of the design with the detailed explanation of how you modified the design to make the processor work for the new instructions. Your block diagram should contain all your design modules.
- Put a screenshot from the simulated waveform for each instruction. A text file is provided in Github which includes some RiscV instruction words. You can use those instruction words to verify your code. Please add them to “inst.bin”.
- After synthesis, please report the frequency, area, and power of your design. You can find that information in syn/112L-RISCV/reports. Code: Please include your design code.

### 7.2 code:

Please put your code and report inside one single folder and submit the folder in zip format. The name of uploaded zip file should be your name.

## 8 References

Waterman, Andrew, et al., The RISC-V Instruction Set Manual, volume I: User-level ISA, version 2.0, EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54 (2014).