	EECS151/251A Fall 2022 Midterm	
Name of the person on left (or aisle)	Your Name	Name of the person on right (or aisle)
	Your SID	

Question	1	2	3	4	5	**Total**
151 Max. points	10	12	14	23	7	**66**
251A Max. points	10	18	14	26	7	**75**

Exam Notes:
You have 110 minutes to work, starting at 8:10 PM Pacific Time and ending at 10 PM Pacific Time.

Before 8:10 pm Pacific Time, you may write down your name, SID, names of the persons next to you, etc., on the first page, but you may NOT begin working.

Please write your name and SID on every page.

The problems are NOT organized in the order of increasing difficulty. If you find yourself taking too long to come up with a solution, consider skipping the problem and moving on to the next one.

Both versions of the RISC-V green card are provided at the end of the document.

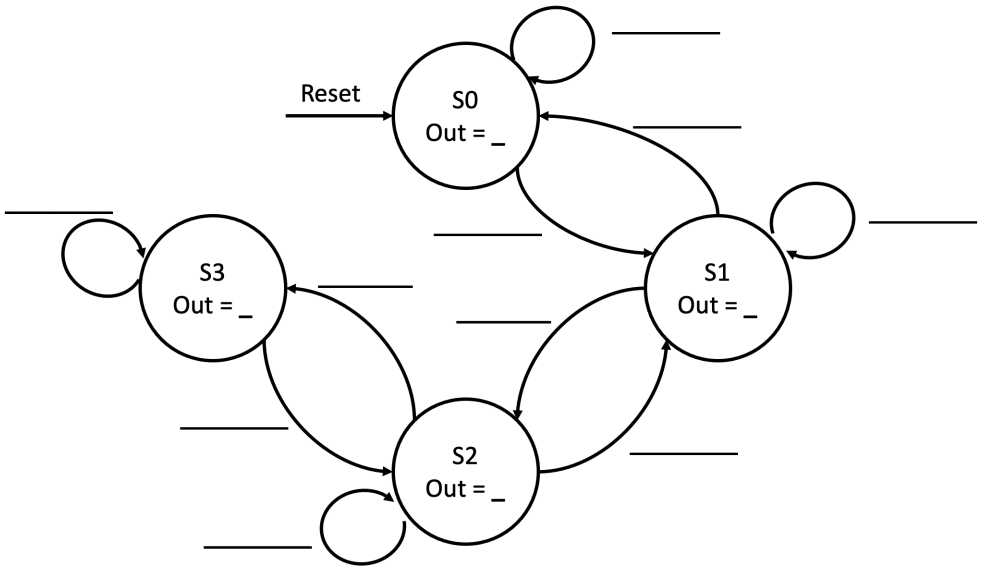
Question 1: Finite State Machine [10 points]

To better manage lab computational resources, we would like to build a resource manager that only allows up to 3 students to be logged onto the same server. Your job is to design this controller as a **Moore FSM** which has...

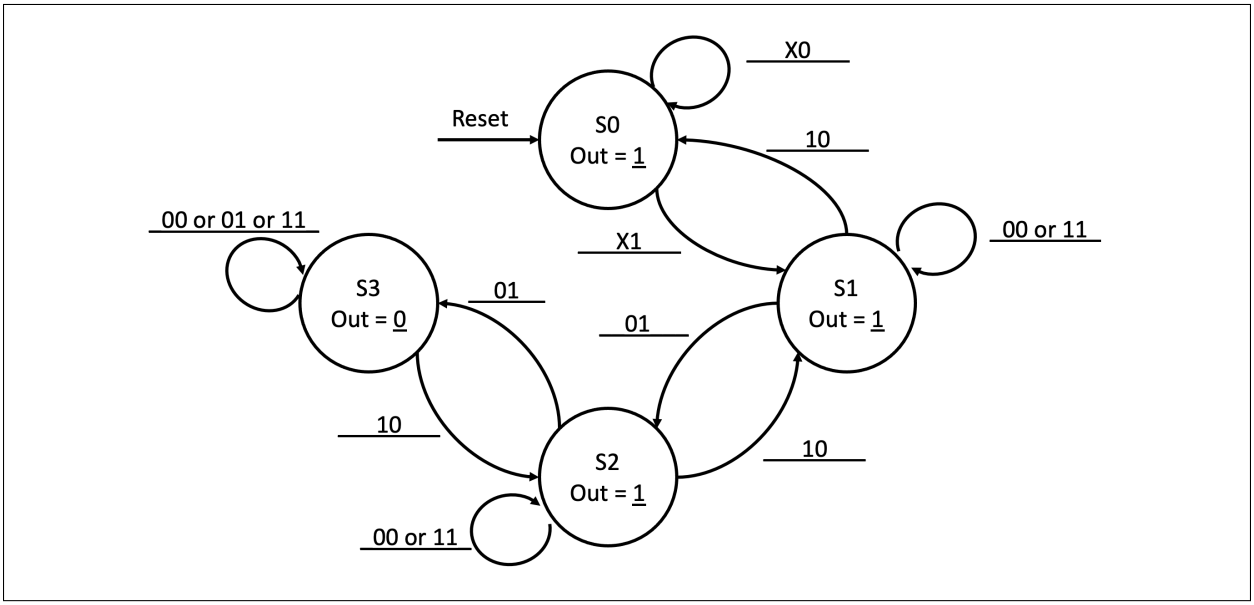
- A 1-bit output `out` that is **0** only if the server should not accept more student logins.
- A 2-bit input `in[1:0]`:
  - `in[0]` is **1** only if a new student *attempts* to log in.
  - `in[1]` is **1** only if a student logs out.
    Note: `in[1]` should be ignored when 0 students are logged into the server.
- A reset signal which will force all students to be logged out.

a) (5 points) A skeleton of the FSM state diagram has been provided for you below. Fill in the output for each state and the input conditions for the state transitions.

- As an example, if a certain state transition will occur if `in[1:0] == 10`, write **10** in the provided space.
- You can use **X** for input bits that don't matter. For example, if the input condition for a state transition is `in[1] == 1`, write **1X**.



Solution:
The FSM has 4 states: S0, S1, S2, S3. A state Sk represents k students that are currently logged into the server.



b) (5 points) Complete the following 2-always block Verilog implementation of the FSM. Note that you are only supposed to write down the first two states;

```
// Assume input and output wire/reg are already declared,
// and the state parameters are already defined properly

reg [_____:0] curr_state, next_state; //Don't miss this line!
always @ (posedge clk) begin
    if(rst) curr_state <= S0;
    else curr_state <= next_state;
end

always @ (*) begin
    //initial conditions to avoid latch

    ____________________________

    ____________________________

    case(curr_state)
        S0: begin
            out = _______________;
            if( ____________________________ ) next_state = S1;
        end
        S1: begin
            out = _______________;
            if( ____________________________ ) next_state = _____;
            else if( ____________________________ ) next_state = _____;
```

```
        end
        // ... the rest of the code
    endcase
end
```

Solution:

```
localparam
    S0 = 2'd0,
    S1 = 2'd1,
    S2 = 2'd2,
    S3 = 2'd3; // This part is skipped in the problem

reg [1:0] curr_state, next_state;

always @ (posedge clk) begin
    if(rst) curr_state <= S0;
    else curr_state <= next_state;
end

always @ (*) begin
    next_state = curr_state; //avoid latch
    out = 0; // this line is optional
    case(curr_state)
        S0: begin
            out = 1;
            if(in[0] == 1'b1) next_state = S1;
        end
        S1: begin
            out = 1;
            if(in == 2'b01) next_state = S2;
            else if(in == 2'b10) next_state = S0;
        end
        S2: begin
            out = 1;
            if(in == 2'b01) next_state = S3;
            else if(in == 2'b10) next_state = S1;
        end
        S3: begin
            out = 0;
            if(in == 2'b10) next_state = S2;
        end
    endcase
end
```

Question 2: Verilog [12/18 points]

Consider the following problem. You have an N-bit binary data input, in which you want to check if an M-bit binary pattern can be found ($N >= M$).

For example, the module should indicate **found** if data input is `16'hcead` and pattern is `4'b0110`. This is because the pattern is found in `16'b1100111010101`**`0110`**`1`.

(You may find the Verilog reduction operators useful in this problem. The reduction operators perform a bit-wise operation on a single operand to produce a single bit result. For example, to perform an `or` operation of all the bits in a operand, we can use: `|4'b1000 = 1`.)

a) (6 points) First, complete the module using only combinational logic.

```
module comb_imp #(parameter N=16, parameter M=4) (
  input [N-1:0] data,
  input [M-1:0] pattern,
  output found
);
  genvar i;

  wire [_____________] match_found;
  generate

    for (i = 0; i < _____________; i += 1) begin

      assign match_found[i] = ____________________________________;
    end
  endgenerate

  assign found = ____________________;
endmodule
```

Solution:

```
N-M:0
N-M+1
(data[i+M-1:i] == pattern)
|(match_found)
```

b) (6 points) Suppose that due to area constraints, we want to transform the design to use only one M-bit comparator for pattern matching (but will take multiple cycles).

At the start of the computation, we capture the input data. When the results are ready, `done` should go high along with the `found` output for at least 1 cycle. Note: you should not index into `data_reg` with `counter`, as this will likely be an expensive operation.

```
module n_cycle_imp #(parameter N=16, parameter M=4) (
  input clk,
  input start,
  input [N-1:0] data,
  input [M-1:0] pattern,
  output reg found,
  output reg done
);
  reg [$clog2(N-M+1):0] counter;
  reg [N-1:0] data_reg;
  wire current_match;

  assign current_match = ____________________________________;

  always @(posedge clk) begin
    if (start) begin
      data_reg <= data;
      counter <= 'd0;
      found <= 1'b0;
      done <= 1'b0;

    end else if (counter == _____________) begin
      done <= 1;
    end else begin

      data_reg <= ________________________________;
      counter <= counter + 'd1;

      found <= ________________________________;
    end
  end
endmodule
```

Solution:

```
(data_reg[M-1:0] == pattern)
N-M+1
{1'b0, data_reg[N-1:1]}
found | current_match
```

c) (6 points) *(251A Only)* Suppose we want to achieve a good balance between area and timing. Complete the following design that takes $(N - M + 1)/2$ cycles and 2 comparators for matching. Be sure to handle boundary conditions. For simplicity, assume $N - M + 1$ is even.

```
module n_div_2_cycle_imp #(parameter N=16, parameter M=4) (
  input clk,
  input start,
  input [N-1:0] data,
  input [M-1:0] pattern,
  output reg found,
  output reg done
);
  reg [$clog2(N-M+1):0] counter;
  reg [N-1:0] data_reg;
  wire [1:0] match;

  assign match[0] = ____________________________________________;

  assign match[1] = ____________________________________________;

  always @(posedge clk) begin
    if (start) begin
      data_reg <= data;
      counter <= 'd0;
      found <= 1'b0;
      done <= 1'b0;

    end else if (counter == ____________________) begin
      done <= 1;
    end else begin

      data_reg <= ____________________________________;
      counter <= counter + 'd1;

      found <= ____________________________________________;
    end
  end
endmodule
```

Solution: There are 2 solutions (and potentially more):
1. Put comparators 1 bit apart and shift 2 bits at a time;
2. Put comparators (N-M+1)/2 apart and shift 1 bit at a time.
To earn full credit, student must handle boundary conditions (i.e. matches at the first or last cycle).

```
(data_reg[M-1:0] == pattern)
(data_reg[M:1] == pattern)
(N-M+1)/2
```

```
{2'b0, data_reg[N-1:2]}
found | match[0] | match[1];
```

or

```
(data_reg[M-1:0] == pattern)
(data_reg[M+(N-M-1)>>1:(N-M+1)>>1] == pattern)
(N-M+1)/2
{1'b0, data_reg[N-1:1]}
found | match[0] | match[1];
```

Question 3: RISC-V ISA and Datapath [14 points]

In this question, we will evaluate a sequence of RISC-V instructions in a single-cycle datapath. We will also explore opportunities to add new instructions (also in the single-cycle datapath).

a) (8 points) Assume that t5 represents a valid memory address. Use the following RISC-V program to answer parts i) through iii).

```
li  t0, 0xAAAAAAAA
li  t1, 0xEEC5251A

sw  t0, 0(t5) #assume this is valid memory
lh  t2, 0(t5)
and t3, t1, t2
blt t3, x0, label
```

i) (2 points) What value is stored in t3?

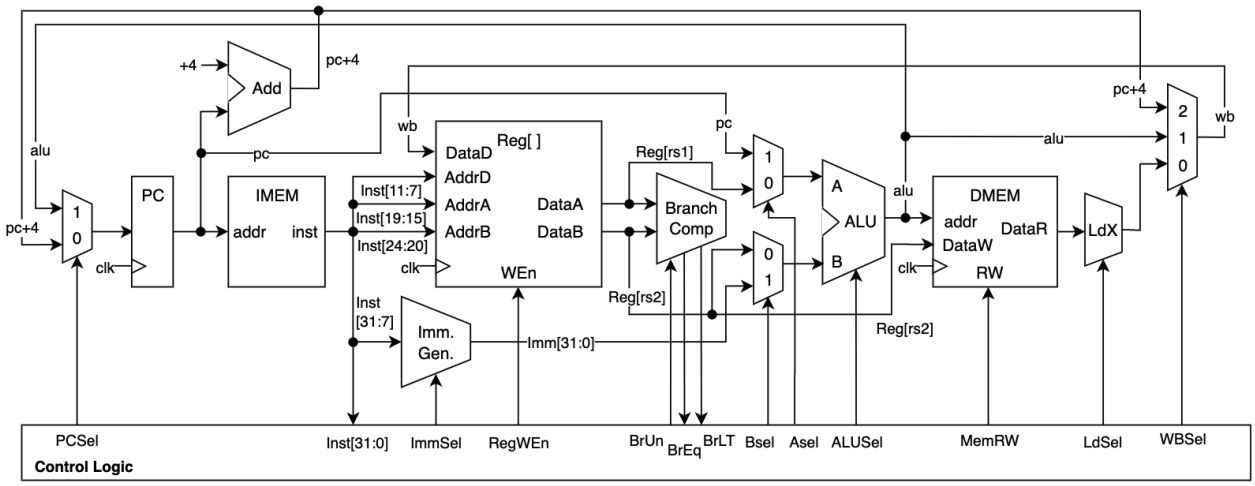
Solution:
Answer: 0xEEC5200A
Explanation: We save 0xAAAAAAAA into the memory address at sp, and then load the lowest 16 bits back into t1 with sign extension. So t1 stores 0xFFFFAAAA. Then we and 0xFFFFAAAA with 0xEEC5251A to get 0xEEC5200A, and store that in t3.

ii) (2 points) Is the branch taken or not taken?

Solution:
Answer: Taken
Explanation: This is a signed branch instruction, so the value in t3 is interpreted as a two's complement negative value due to the MSB of 1. Any negative number is less than 0, so the branch is taken.

iii) (4 points) Using the datapath diagram provided on the next page, fill in the control signals when executing the following instructions according to the values calculated in the previous subparts. The first add is an example. The rest is a subset of the program given above. Assume t0 and t1 have the immediates loaded respectively. Use * to denote don't care.

Instruction	PCSel	RegWEn	Asel	BSel	MemRW	WBSel
EX: add rd, rs1, rs2	0	1	0	0	READ	1
sw t0 0(t5)						
lh t2 0(t5)						
and t3, t1, t2						
blt t3 x0 label						



Solution: See the table below.

Instruction	PCSel	ImmSel	RegWEn	Asel	BSel	ALUSel	MemRW	WBSel
sw t0 0(sp)	0	S	0	0	1	ADD	WRITE	*
lh t2 0(sp)	0	I	1	0	1	ADD	READ	0
and t3, t1, t2	0	*	1	0	0	AND	READ	1
blt t3 x0 label	1	B	0	1	1	ADD	READ	*

b) (6 points) If we want to add some new instructions, what modifications would be required? Select all that apply and explain your choice. The only allowed datapath modifications are adding wires connecting existing units, and adding muxes and adders.

i) (2 points) Description: $R[rd] = \sim (R[rs1] \& R[rs2])$

```
nand rd rs1 rs2
```

- ☐ Control Logic
- ☐ Datapath
- ☐ ALU
- ☐ Immediate Generator

Explain:

Solution:
Answer: ALU, Control Logic. We want to have the ALU deal with another type of instruction.

ii) (2 points) Description: R[rd] = M[rs1 + imm] + 4

```
lw_add_4 rd rs1 imm
```

- ☐ Control Logic
- ☐ Datapath
- ☐ ALU
- ☐ Immediate Generator

Explain:

Solution:
Answer: Control Logic, Datapath. After we regularly load out a value, use a new adder in the datapath to add 4 and a new input to the writeback mux.

iii) (2 points) Description: M[rs1 + imm2] = imm1

```
sw_immediate rs1 imm1 imm2
```

- ☐ Control Logic
- ☐ Datapath
- ☐ ALU
- ☐ Immediate Generator

Explain:

Solution:
Answer: Control Logic, Datapath, Immediate Generator. Edit the immediate generator to output 2 immediates and add a new mux right before the input to DMEM.

Question 4: Pipelining [23/26 points]

We are interested in exploring different strategies to support data forwarding in our **5-stage pipelined** RISC-V datapath covered in the lecture. Similar to our assumptions so far, our datapath assumes:

- The register file is a synchronous-write, asynchronous-read design, where it is capable of reading and writing to the same register inside one cycle.
- Both the IMEM and DMEM blocks are synchronous-write and asynchronous-read designs. Assume all memory operations complete within one cycle.

We have already calculated the critical path of the *Fetch* (**500ps**), *Memory* (**600ps**), and *Writeback* (**450ps**) stages. Next, we want to calculate the delay of the *Decode* and *Execute* stages, where their delays could be affected by the implementation of the forwarding logic.

To support our calculations, Figure 1 shows the delays of different components.

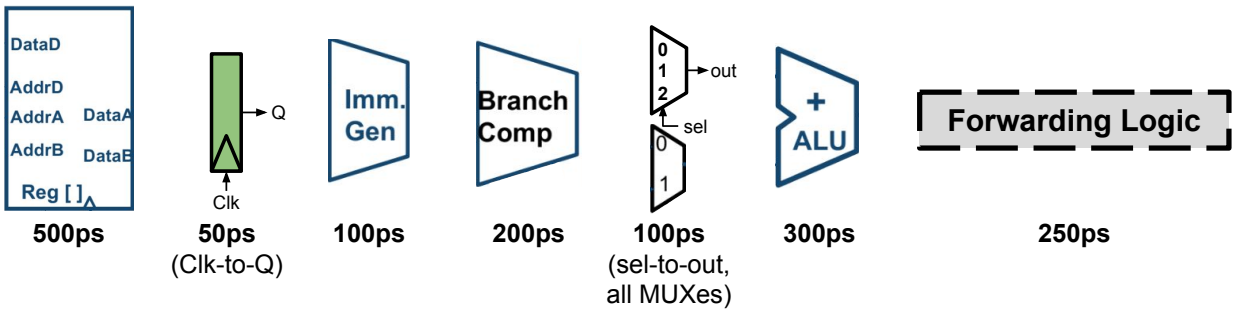


Figure 1: Propagation delay of different logic components in the *Decode* and *Execute* stage.

a) (8 points) To avoid stalling for data hazards, we want to add forwarding logic to the datapath. We start by adding **ALU-to-ALU** forwarding for just *rs1*. The forwarding logic block is purely combinational.

i) (3 points) One way to add the forwarding logic is to use the instructions from *Execute* and *Memory* stages, as shown in Figure 2. $\text{Inst}_{\text{Stage}}$ signal denotes the instruction currently in flight at the stage.
Considering the delays provided previously, what is the minimum clock period achievable (i.e., the critical path) in this design? Show your calculation.

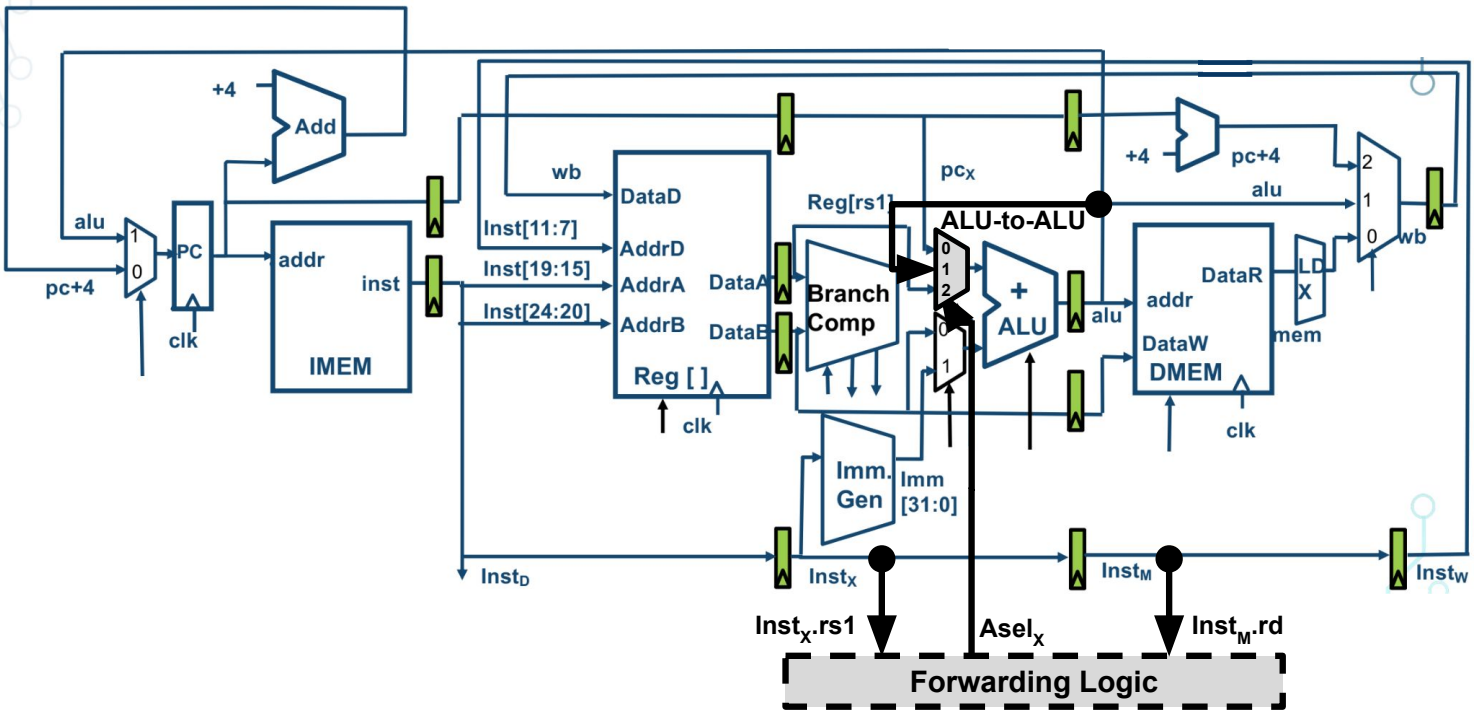


Figure 2: ALU-to-ALU forwarding logic added to the 5-stage pipelined RISC-V datapath.

Solution: 700ps. Forwarding logic lengthens the critical path of the Execute stage, because the signal has to go through the forwarding logic before arriving at the ALU operand MUX. This means PipeReg+Forwarding+MUX+ALU = 50+250+100+300 = 700ps. Immgen is done in parallel to the forwarding logic in shorter time so it doesn't affect the critical path. This makes Execute the longest stage, determining the overall clock period.
Note that the branch comparator does not contribute to the critical path because its output is not used anywhere in the execute stage.

ii) (3 points) Another way to add forwarding logic is to use instructions from the *Decode* and *Execute* stages, as shown in Figure 3. Here, we need to use a pipeline register to stage the Asel_D control signal so that we can feed it to the MUX with the right timing. What is the minimum clock period achievable (i.e., the critical path) for this new design?

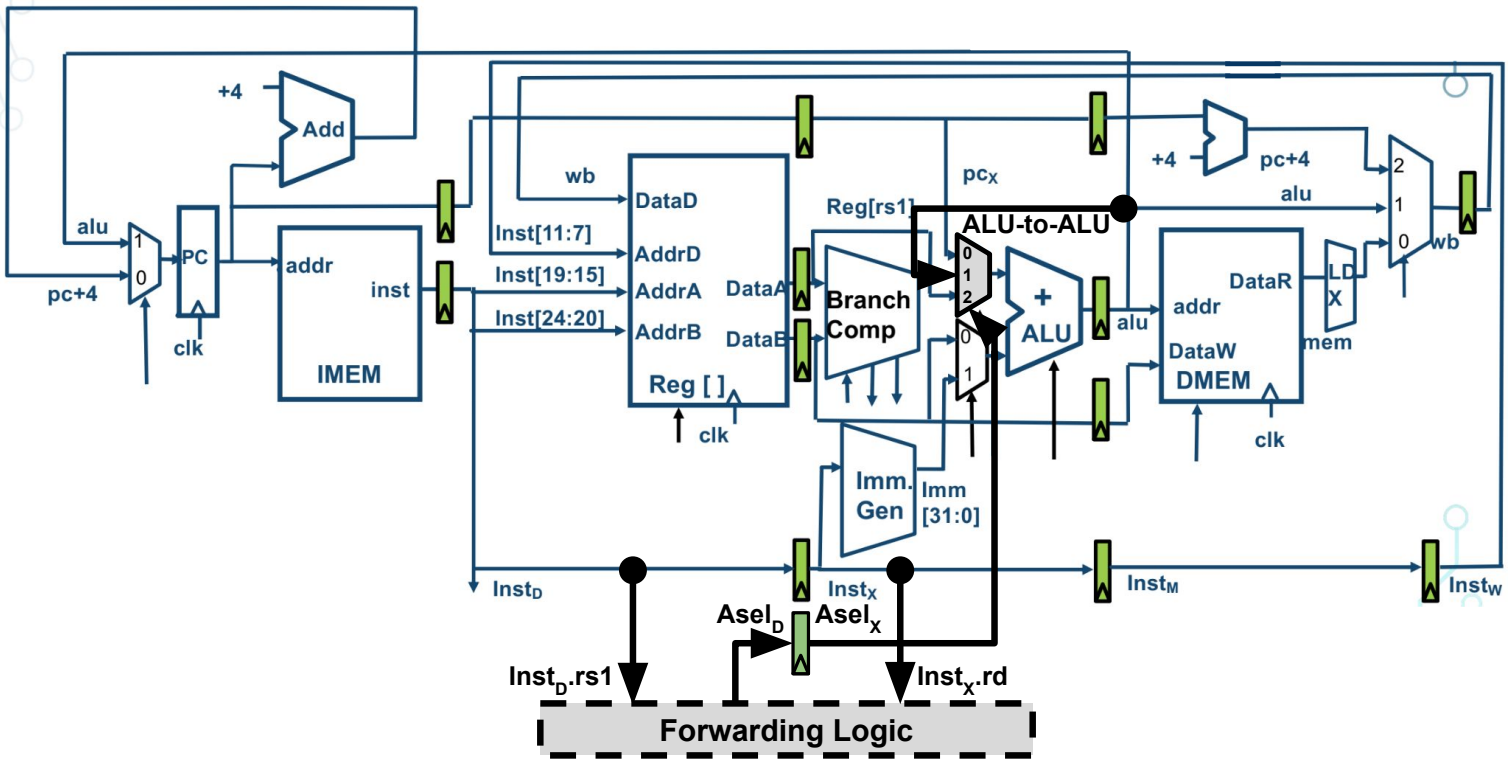


Figure 3: Alternative datapath for the ALU-to-ALU forwarding logic.

Solution: The delay path of the forwarding logic is now cut up by the pipeline register, and this shortens the critical path of Execute to PipeReg-Imm-MUX-ALU = 50+100+100+300 = 550ps (note that now we have to include immgen.) This makes the Memory stage become the new critical path, and the minimum clock period is **600ps. This design achieves higher clock frequency** than i), because it prevents the forwarding logic from worsening the critical path by overlapping its latency with RegFile in the D stage. The cost is the area/power of an additional pipeline register, which should be significantly cheaper than the performance benefit we get.

iii) (2 points) Compare the designs between **i)** and **ii)**, discuss which design achieves a higher clock frequency, and why?

b) (7 points) Now we add **MEM-to-ALU** forwarding to the datapath, as shown in Figure 4.

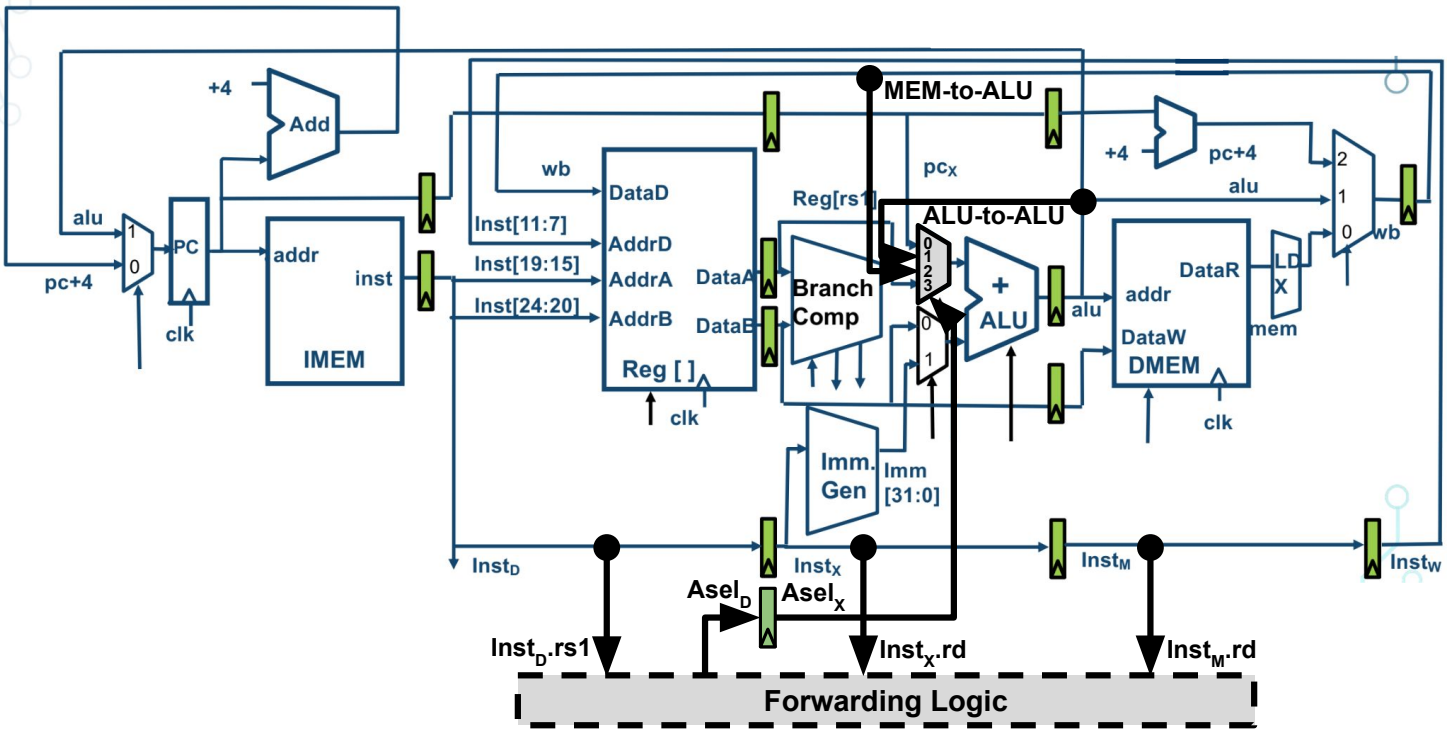


Figure 4: Datapath with both ALU-to-ALU and MEM-to-ALU forwarding implemented.

i) (2 points) Would the MEM-to-ALU forwarding path completely eliminate stalls in the following code sequence? Explain why.

```
ld  x2, 0(x1)
addi x4, x2, 151
```

ii) (5 points) How can we implement the forwarding logic module in Figure 4 in Verilog? Complete the following code by filling in the blanks.
Note: Assume we are only handling R/I-type instructions so that the *rs1* and *rd* bits are valid.

```
// constants used for the Asel MUX selection signal
`define ASEL_PC    2'd0 // select PC for jump/branches
`define ASEL_FORWARD_ALU 2'd1 // select rs1 from ALU-to-ALU
   forwarding
`define ASEL_FORWARD_MEM 2'd2 // select rs1 from MEM-to-ALU
   forwarding
`define ASEL_REGFILE 2'd3 // select rs1 from register file
// X0 needs to be handled differently
`define X0         5'd0 // denote register x0

module forwarding_logic (
  input is_jump_or_branch, // assume this is supplied from decode
     stage
  input [4:0] inst_d_rs1, // rs1 part of InstD
  input [4:0] inst_x_rd, // rd part of InstX
  input [4:0] inst_m_rd, // rd part of InstM
  output reg [1:0] asel_d // selection signal for the Asel MUX
);

always @(*) begin
   if (is_jump_or_branch) begin
      asel_d = `ASEL_PC;

   end else if (_____________________________________) begin

      asel_d = _____________________________________;

   end else if (_____________________________________) begin

      asel_d = _____________________________________;

   end else begin
      asel_d = `ASEL_REGFILE;
   end
end

endmodule
```

Solution: The key thing is that when both ALU-to-ALU and MEM-to-ALU are eligible, we need to forward the newest data. This is done by having a multi-clause if-else chain where we prioritize the ALU-to-ALU case over MEM-to-ALU.
Another thing is to filter out the case where `rs1 == x0`. This is required because some instructions deliberately set the destination register to `x0` to discard the result (ex: `jalr x0, 0(x1)`). If we forward the destination register in those cases, the data will not get discarded and propagate wrong results down the pipeline. This is easy to miss so we leave in `` `define X0 `` as a hint.

```
always @(*) begin
    if (is_jump_or_branch) begin
        asel_d = `ASEL_PC;
    end else if (inst_d_rs1 != `X0 && inst_d_rs1 == inst_x_rd) begin
        asel_d = `ASEL_FORWARD_ALU;
    end else if (inst_d_rs1 != `X0 && inst_d_rs1 == inst_m_rd) begin
        asel_d = `ASEL_FORWARD_MEM;
    end else begin
        asel_d = `ASEL_REGFILE;
    end
end

endmodule
```

1 point for each blank.

c) (8/11 points) Now, we want to run some real code on our implemented design to see how it performs. Suppose we have a small RISC-V program that reads:

```
    addi t1, x0, 0xEEC51510
    bnez t1, normal
fault:
    addi t1, x0, 0xEEC5251A
normal:
    lw   t2, 0(t1)
    xori t4, t2, 1
    lw   t3, 0(t2)
    slli t3, t3, 1
    xor  t5, t3, t4
```

Assume that branches are always predicted **not-taken** and mispredictions are handled by pipeline flushes. Assume all memory accesses are valid.

i) (8 points) Consider that we have implemented both **ALU-to-ALU** and **MEM-to-ALU** forwarding, for both **ALU and branch comparator**, for both **rs1 and rs2**. Fill out the table on the next page and specify how many cycles this program takes to run. Count all cycles from *Fetch* stage of the first instruction to *Writeback* stage of the last instruction. For the instruction column, you only need to write the operator. You may not use all the rows and columns.

Solution:
16 cycles.
The catch is the load-to-use sequences (`lw-addi` and `lw-slli`), where even with MEM-to-ALU forwarding we cannot avoid one stall cycle.

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
addi	F	D	X	M	W																	
bnez		F	D	X	M	W																
addi			F	D	X	~~M~~																
lw				F	D	~~X~~																
xori					F	~~D~~																
lw						F	D	X	M	W												
xori							F	D	D	X	M	W										
lw								F	F	D	X	M	W									
slli										F	D	D	X	M	W							
xor											F	F	D	X	M	W						

ii) (3 points) **(251A Only)** Show how you can reorder instructions in the program in a way that eliminates some of the data hazard stalls in part **i)**, without changing the final values of the registers and data memory. Do not reorder the instructions across the labels (`fault` and `normal`).

Name: Student ID:

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
`addi`	F	D	X	M	W																	

Write down the assembly after you reorder them, and explain how this reordering can eliminate data hazard.

Solution: We start from the fact that we cannot avoid one stall cycle for back-to-back load-to-use chains. So the trick is to keep the load-to-use distance longer by "prefetching" `t2`:

```
lw   t2, 0(t1)
lw   t3, 0(t2)
xori t4, t2, 1
slli t3, t3, 1
```

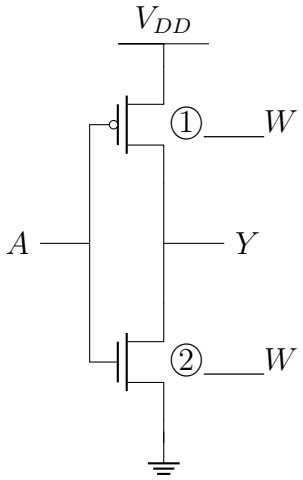
Then we can eliminate the two load stalls. However, we introduce one new stall between `lw`s because there is still an immediate load-to-use dependency on `t2`. The total number of cycles becomes 15 (not graded).

Question 5: Logical Effort and Path Delay [7 points]

In this question, unless otherwise specified, assume
1) $\lambda = 1$ is the ratio between drain and gate capacitance.
2) No internal capacitance (i.e., drain capacitance is only relevant at the output node).

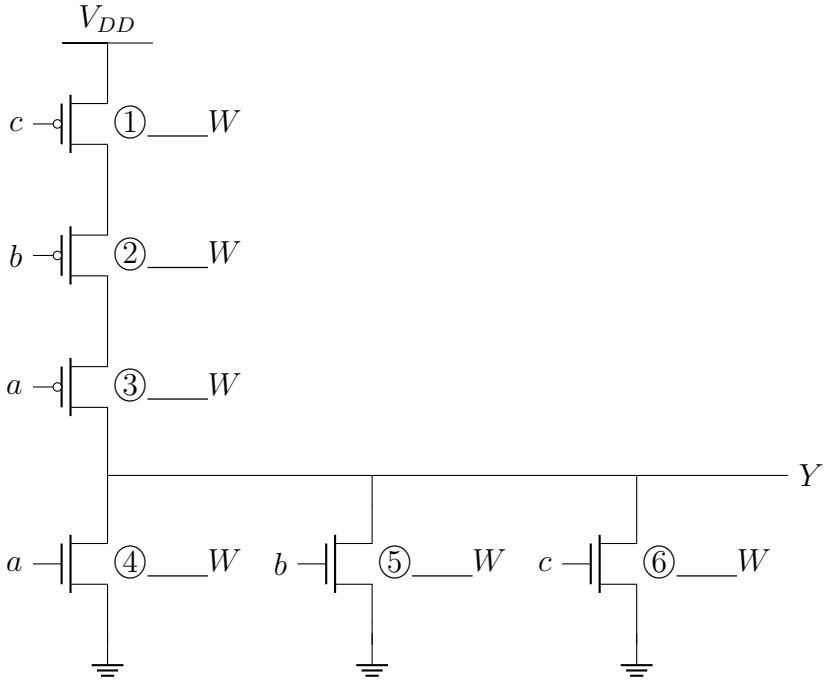
Assume that we are using a different technology from what is taught in class, where PMOS transistors are 2 times stronger than NMOS transistors (i.e., with the same width, $R_{eq,n} = 2R_{eq,p}$). The minimum width of a transistor in this technology is given as W.

a) (2 points) Implement a minimum-sized inverter in our new technology using static CMOS logic. Resize the transistors so that the pull-up and pull-down paths have the same resistance R_{on}.



Solution: ① $1W$ ② $2W$
Since PMOS has a lower resistance, we can make PMOS the smallest size. Then, to match the PMOS resistance, we have to make NMOS 2 times wider.

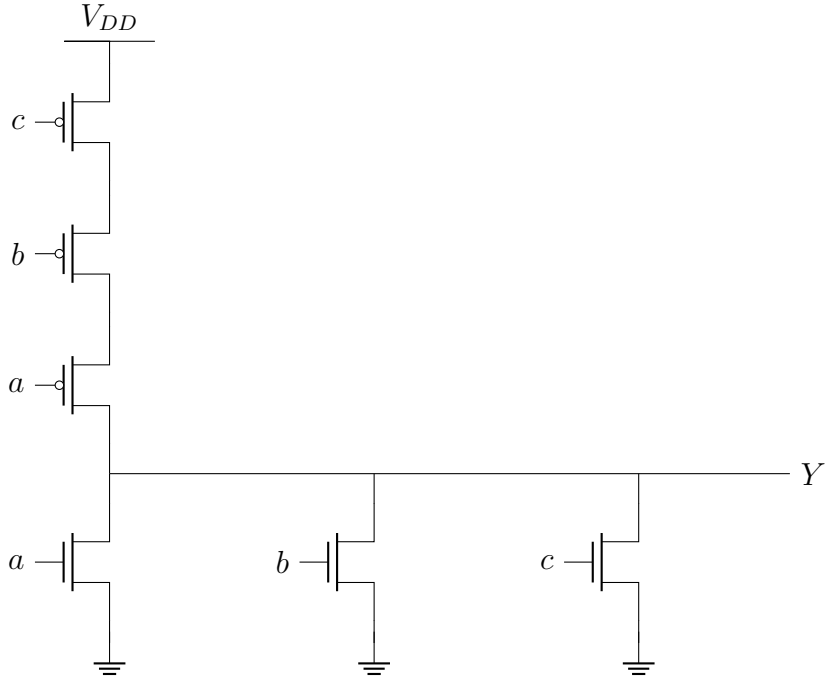
b) (3 points) Now, implement a 3-input NOR in our new technology. Again, resize the transistors so that the worst-case pull-up and pull-down paths have the same resistance R_{on}. (*Don't forget that PMOS transistors are 2 times stronger than NMOS transistors.*)



Solution: ① $3W$ ② $3W$ ③ $3W$ ④ $2W$ ⑤ $2W$ ⑥ $2W$

The pull-up path has 3 PMOS in serial, so the width of each should be 3. The pull-down path has 1 NMOS in the worst case, so each NMOS should have a width of 2.

c) (2 points) Given the transistor sizings above, compute the logical effort and intrinsic delay of the 3-input NOR gate in our new technology. Show your work. Partial credits will be given for correct intermediate values.



$LE =$ _____

$P =$ _____

Solution: $LE = \frac{5}{3}, P = 3t_{inv}$
$R_{eq} = 0.5R_P, C_g = 6 + 4 = 10C_P$. To get an inverter of equal strength, we must widen the min-inverter by 2x. $C_{inv} = 2 + 4 = 6C_P$. Then, $LE = \frac{C_g}{C_{inv}} = \frac{5}{3}$.
P is defined as $\frac{C_{drain}}{C_{drain,inv}} t_{inv}$. $C_{drain} = 6 + 4 + 4 + 4 = 18C_P, C_{drain,inv} = 2 + 4 = 6C_P$. Then $P = \frac{18}{6} t_{inv} = 3t_{inv}$

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

① RISC-V Reference Data

RV64I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add,addw	R	ADD (Word)	R[rd] = R[rs1] + R[rs2]	1)
addi,addiw	I	ADD Immediate (Word)	R[rd] = R[rs1] + imm	1)
and	R	AND	R[rd] = R[rs1] & R[rs2]	
andi	I	AND Immediate	R[rd] = R[rs1] & imm	
auipc	U	Add Upper Immediate to PC	R[rd] = PC + {imm, 12'b0}	
beq	SB	Branch EQual	if(R[rs1]==R[rs2) PC=PC+{imm,1b'0}	
bge	SB	Branch Greater than or Equal	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	
bgeu	SB	Branch ≥ Unsigned	if(R[rs1]>=R[rs2) PC=PC+{imm,1b'0}	2)
blt	SB	Branch Less Than	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	
bltu	SB	Branch Less Than Unsigned	if(R[rs1]<R[rs2) PC=PC+{imm,1b'0}	2)
bne	SB	Branch Not Equal	if(R[rs1]!=R[rs2) PC=PC+{imm,1b'0}	
csrrc	I	Cont./Stat.RegRead&Clear	R[rd] = CSR;CSR = CSR & ~R[rs1]	
csrrci	I	Cont./Stat.RegRead&Clear Imm	R[rd] = CSR;CSR = CSR & ~imm	
csrrs	I	Cont./Stat.RegRead&Set	R[rd] = CSR; CSR = CSR | R[rs1]	
csrrsi	I	Cont./Stat.RegRead&Set Imm	R[rd] = CSR; CSR = CSR | imm	
csrrw	I	Cont./Stat.RegRead&Write	R[rd] = CSR; CSR = R[rs1]	
csrrwi	I	Cont./Stat.Reg Read&Write Imm	R[rd] = CSR; CSR = imm	
ebreak	I	Environment BREAK	Transfer control to debugger	
ecall	I	Environment CALL	Transfer control to operating system	
fence	I	Synch thread	Synchronizes threads	
fence.i	I	Synch Instr & Data	Synchronizes writes to instruction stream	
jal	UJ	Jump & Link	R[rd] = PC+4; PC = PC + {imm,1b'0}	
jalr	I	Jump & Link Register	R[rd] = PC+4; PC = R[rs1]+imm	3)
lb	I	Load Byte	R[rd] = {56'bM[](7),M[R[rs1]+imm](7:0)}	4)
lbu	I	Load Byte Unsigned	R[rd] = {56'b0,M[R[rs1]+imm](7:0)}	
ld	I	Load Doubleword	R[rd] = M[R[rs1]+imm](63:0)	
lh	I	Load Halfword	R[rd] = {48'bM[](15),M[R[rs1]+imm](15:0)}	4)
lhu	I	Load Halfword Unsigned	R[rd] = {48'b0,M[R[rs1]+imm](15:0)}	
lui	U	Load Upper Immediate	R[rd] = {32b'imm<31>, imm, 12'b0}	
lw	I	Load Word	R[rd] = {32'bM[](31),M[R[rs1]+imm](31:0)}	4)
lwu	I	Load Word Unsigned	R[rd] = {32'b0,M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sd	S	Store Doubleword	M[R[rs1]+imm](63:0) = R[rs2](63:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll,sllw	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
slli,slliw	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
slt	R	Set Less Than	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
slti	I	Set Less Than Immediate	R[rd] = (R[rs1] < imm) ? 1 : 0	
sltiu	I	Set < Immediate Unsigned	R[rd] = (R[rs1] < imm) ? 1 : 0	2)
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	2)
sra,sraw	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
srai,sraiw	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
srl,srlw	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
srli,srliw	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	1)
sw	S	Store Word	M[R[rs1]+imm](31:0) = R[rs2](31:0)	
xor	R	XOR	R[rd] = R[rs1] ^ R[rs2]	
xori	I	XOR Immediate	R[rd] = R[rs1] ^ imm	

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
2) Operation assumes unsigned integers (instead of 2's complement)
3) The least significant bit of the branch address in jalr is set to 0
4) (signed) Load instructions extend the sign bit of data to fill the 64-bit register
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift
6) Multiply with one operand signed and one unsigned
7) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
8) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)
9) Atomic memory operation; nothing else can interpose itself between the read and the write of the memory location
The immediate field is sign-extended in RISC-V

② **ARITHMETIC CORE INSTRUCTION SET**

RV64M Multiply Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
mul,mulw	R	MULtiply (Word)	R[rd] = (R[rs1] * R[rs2])(63:0)	1)
mulh	R	MULtiply High	R[rd] = (R[rs1] * R[rs2])(127:64)	
mulhu	R	MULtiply High Unsigned	R[rd] = (R[rs1] * R[rs2])(127:64)	2)
mulhsu	R	MULtiply upper Half Sign/Uns	R[rd] = (R[rs1] * R[rs2])(127:64)	6)
div,divw	R	DIVide (Word)	R[rd] = (R[rs1] / R[rs2])	1)
divu	R	DIVide Unsigned	R[rd] = (R[rs1] / R[rs2])	2)
rem,remw	R	REMainder (Word)	R[rd] = (R[rs1] % R[rs2])	1)
remu,remuw	R	REMainder Unsigned (Word)	R[rd] = (R[rs1] % R[rs2])	1,2)

RV64F and RV64D Floating-Point Extensions

fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	1)
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	1)
fadd.s,fadd.d	R	ADD	F[rd] = F[rs1] + F[rs2]	7)
fsub.s,fsub.d	R	SUBtract	F[rd] = F[rs1] – F[rs2]	7)
fmul.s,fmul.d	R	MULtiply	F[rd] = F[rs1] * F[rs2]	7)
fdiv.s,fdiv.d	R	DIVide	F[rd] = F[rs1] / F[rs2]	7)
fsqrt.s,fsqrt.d	R	SQuare RooT	F[rd] = sqrt(F[rs1])	7)
fmadd.s,fmadd.d	R	Multiply-ADD	F[rd] = F[rs1] * F[rs2] + F[rs3]	7)
fmsub.s,fmsub.d	R	Multiply-SUBtract	F[rd] = F[rs1] * F[rs2] - F[rs3]	7)
fnmadd.s,fnmadd.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] + F[rs3])	7)
fnmsub.s,fnmsub.d	R	Negative Multiply-SUBtract	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fsgnj.s,fsgnj.d	R	SiGN source	F[rd] = { F[rs2]<63>,F[rs1]<62:0>}	7)
fsgnjn.s,fsgnjn.d	R	Negative SiGN source	F[rd] = { (~F[rs2]<63>), F[rs1]<62:0>}	7)
fsgnjx.s,fsgnjx.d	R	Xor SiGN source	F[rd] = {F[rs2]<63>^F[rs1]<63>, F[rs1]<62:0>}	7)
fmin.s,fmin.d	R	MINimum	F[rd] = (F[rs1] < F[rs2]) ? F[rs1] : F[rs2]	7)
fmax.s,fmax.d	R	MAXimum	F[rd] = (F[rs1] > F[rs2]) ? F[rs1] : F[rs2]	7)
feq.s,feq.d	R	Compare Float EQual	R[rd] = (F[rs1]== F[rs2]) ? 1 : 0	7)
flt.s,flt.d	R	Compare Float Less Than	R[rd] = (F[rs1]< F[rs2]) ? 1 : 0	7)
fle.s,fle.d	R	Compare Float Less than or =	R[rd] = (F[rs1]<= F[rs2]) ? 1 : 0	7)
fclass.s,fclass.d	R	Classify Type	R[rd] = class(F[rs1])	7,8)
fmv.s.x,fmv.d.x	R	Move from Integer	F[rd] = R[rs1]	7)
fmv.x.s,fmv.x.d	R	Move to Integer	R[rd] = F[rs1]	7)
fcvt.s.d	R	Convert to SP from DP	F[rd] = single(F[rs1])	
fcvt.d.s	R	Convert to DP from SP	F[rd] = double(F[rs1])	
fcvt.s.w,fcvt.d.w	R	Convert from 32b Integer	F[rd] = float(R[rs1](31:0))	7)
fcvt.s.l,fcvt.d.l	R	Convert from 64b Integer	F[rd] = float(R[rs1](63:0))	7)
fcvt.s.wu,fcvt.d.wu	R	Convert from 32b Int Unsigned	F[rd] = float(R[rs1](31:0))	2,7)
fcvt.s.lu,fcvt.d.lu	R	Convert from 64b Int Unsigned	F[rd] = float(R[rs1](63:0))	2,7)
fcvt.w.s,fcvt.w.d	R	Convert to 32b Integer	R[rd](31:0) = integer(F[rs1])	7)
fcvt.l.s,fcvt.l.d	R	Convert to 64b Integer	R[rd](63:0) = integer(F[rs1])	7)
fcvt.wu.s,fcvt.wu.d	R	Convert to 32b Int Unsigned	R[rd](31:0) = integer(F[rs1])	2,7)
fcvt.lu.s,fcvt.lu.d	R	Convert to 64b Int Unsigned	R[rd](63:0) = integer(F[rs1])	2,7)

RV64A Atomtic Extension

amoadd.w,amoadd.d	R	ADD	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] + R[rs2]	9)
amoand.w,amoand.d	R	AND	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] & R[rs2]	9)
amomax.w,amomax.d	R	MAXimum	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amomaxu.w,amomaxu.d	R	MAXimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] > M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amomin.w,amomin.d	R	MINimum	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	9)
amominu.w,amominu.d	R	MINimum Unsigned	R[rd] = M[R[rs1]], if (R[rs2] < M[R[rs1]]) M[R[rs1]] = R[rs2]	2,9)
amoor.w,amoor.d	R	OR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] | R[rs2]	9)
amoswap.w,amoswap.d	R	SWAP	R[rd] = M[R[rs1]], M[R[rs1]] = R[rs2]	9)
amoxor.w,amoxor.d	R	XOR	R[rd] = M[R[rs1]], M[R[rs1]] = M[R[rs1]] ^ R[rs2]	9)
lr.w,lr.d	R	Load Reserved	R[rd] = M[R[rs1]], reservation on M[R[rs1]]	
sc.w,sc.d	R	Store Conditional	if reserved, M[R[rs1]] = R[rs2], R[rd] = 0; else R[rd] = 1	

CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

PSEUDO INSTRUCTIONS ③

MNEMONIC	NAME	DESCRIPTION	USES
beqz	Branch = zero	if(R[rs1]==0) PC=PC+{imm,1b'0}	beq
bnez	Branch ≠ zero	if(R[rs1]!=0) PC=PC+{imm,1b'0}	bne
fabs.s,fabs.d	Absolute Value	F[rd] = (F[rs1]< 0) ? –F[rs1] : F[rs1]	fsgnx
fmv.s,fmv.d	FP Move	F[rd] = F[rs1]	fsgnj
fneg.s,fneg.d	FP negate	F[rd] = –F[rs1]	fsgnjn
j	Jump	PC = {imm,1b'0}	jal
jr	Jump register	PC = R[rs1]	jalr
la	Load address	R[rd] = address	auipc
li	Load imm	R[rd] = imm	addi
mv	Move	R[rd] = R[rs1]	addi
neg	Negate	R[rd] = –R[rs1]	sub
nop	No operation	R[0] = R[0]	addi
not	Not	R[rd] = ~R[rs1]	xori
ret	Return	PC = R[1]	jalr
seqz	Set = zero	R[rd] = (R[rs1]== 0) ? 1 : 0	sltiu
snez	Set ≠ zero	R[rd] = (R[rs1]!= 0) ? 1 : 0	sltu

OPCODES IN NUMERICAL ORDER BY OPCODE

MNEMONIC	FMT	OPCODE	FUNCT3	FUNCT7 OR IMM	HEXADECIMAL
lb	I	0000011	000		03/0
lh	I	0000011	001		03/1
lw	I	0000011	010		03/2
ld	I	0000011	011		03/3
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
lwu	I	0000011	110		03/6
fence	I	0001111	000		0F/0
fence.i	I	0001111	001		0F/1
addi	I	0010011	000		13/0
slli	I	0010011	001	0000000	13/1/00
slti	I	0010011	010		13/2
sltiu	I	0010011	011		13/3
xori	I	0010011	100		13/4
srli	I	0010011	101	0000000	13/5/00
srai	I	0010011	101	0100000	13/5/20
ori	I	0010011	110		13/6
andi	I	0010011	111		13/7
auipc	U	0010111			17
addiw	I	0011011	000		1B/0
slliw	I	0011011	001	0000000	1B/1/00
srliw	I	0011011	101	0000000	1B/5/00
sraiw	I	0011011	101	0100000	1B/5/20
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
sd	S	0100011	011		23/3
add	R	0110011	000	0000000	33/0/00
sub	R	0110011	000	0100000	33/0/20
sll	R	0110011	001	0000000	33/1/00
slt	R	0110011	010	0000000	33/2/00
sltu	R	0110011	011	0000000	33/3/00
xor	R	0110011	100	0000000	33/4/00
srl	R	0110011	101	0000000	33/5/00
sra	R	0110011	101	0100000	33/5/20
or	R	0110011	110	0000000	33/6/00
and	R	0110011	111	0000000	33/7/00
lui	U	0110111			37
addw	R	0111011	000	0000000	3B/0/00
subw	R	0111011	000	0100000	3B/0/20
sllw	R	0111011	001	0000000	3B/1/00
srlw	R	0111011	101	0000000	3B/5/00
sraw	R	0111011	101	0100000	3B/5/20
beq	SB	1100011	000		63/0
bne	SB	1100011	001		63/1
blt	SB	1100011	100		63/4
bge	SB	1100011	101		63/5
bltu	SB	1100011	110		63/6
bgeu	SB	1100011	111		63/7
jalr	I	1100111	000		67/0
jal	UJ	1101111			6F
ecall	I	1110011	000	000000000000	73/0/000
ebreak	I	1110011	000	000000000001	73/0/001
CSRRW	I	1110011	001		73/1
CSRRS	I	1110011	010		73/2
CSRRC	I	1110011	011		73/3
CSRRWI	I	1110011	101		73/5
CSRRSI	I	1110011	110		73/6
CSRRCI	I	1110011	111		73/7

REGISTER NAME, USE, CALLING CONVENTION ④

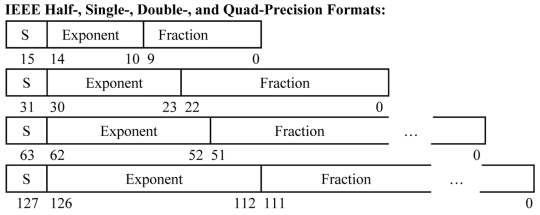
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return values	Caller
x12-x17	a2-a7	Function arguments	Caller
x18-x27	s2-s11	Saved registers	Callee
x28-x31	t3-t6	Temporaries	Caller
f0-f7	ft0-ft7	FP Temporaries	Caller
f8-f9	fs0-fs1	FP Saved registers	Callee
f10-f11	fa0-fa1	FP Function arguments/Return values	Caller
f12-f17	fa2-fa7	FP Function arguments	Caller
f18-f27	fs2-fs11	FP Saved registers	Callee
f28-f31	ft8-ft11	R[rd] = R[rs1] + R[rs2]	Caller

IEEE 754 FLOATING-POINT STANDARD

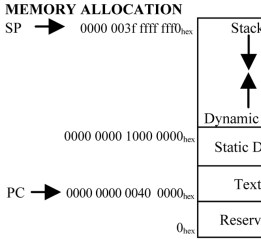
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

where Half-Precision Bias = 15, Single-Precision Bias = 127, Double-Precision Bias = 1023, Quad-Precision Bias = 16383

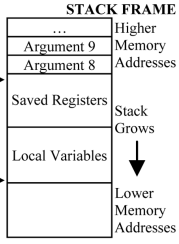
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi
10^{-3}	milli-	m	10^{-15}	femto-	f
10^{-6}	micro-	μ	10^{-18}	atto-	a
10^{-9}	nano-	n	10^{-21}	zepto-	z
10^{-12}	pico-	p	10^{-24}	yocto-	y

RISC-V Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

CS 61C Reference Card

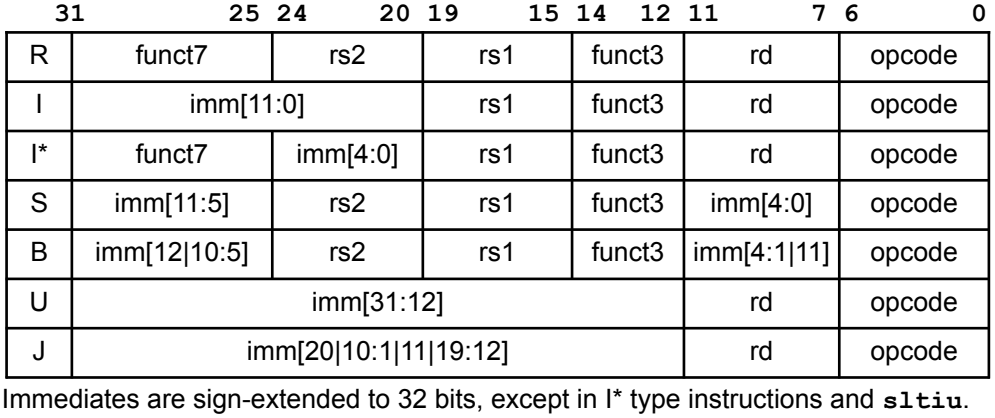
Version 1.5.0

	Instruction	Name	Description	Type	Opcode	Funct3	Funct7
Arithmetic	`add rd rs1 rs2`	ADD	`rd = rs1 + rs2`	R	`011 0011`	`000`	`000 0000`
	`sub rd rs1 rs2`	SUBtract	`rd = rs1 - rs2`	R	`011 0011`	`000`	`010 0000`
	`and rd rs1 rs2`	bitwise AND	`rd = rs1 & rs2`	R	`011 0011`	`111`	`000 0000`
	`or rd rs1 rs2`	bitwise OR	`rd = rs1 | rs2`	R	`011 0011`	`110`	`000 0000`
	`xor rd rs1 rs2`	bitwise XOR	`rd = rs1 ^ rs2`	R	`011 0011`	`100`	`000 0000`
	`sll rd rs1 rs2`	Shift Left Logical	`rd = rs1 << rs2`	R	`011 0011`	`001`	`000 0000`
	`srl rd rs1 rs2`	Shift Right Logical	`rd = rs1 >> rs2` (Zero-extend)	R	`011 0011`	`101`	`000 0000`
	`sra rd rs1 rs2`	Shift Right Arithmetic	`rd = rs1 >> rs2` (Sign-extend)	R	`011 0011`	`101`	`010 0000`
	`slt rd rs1 rs2`	Set Less Than (signed)	`rd = (rs1 < rs2) ? 1 : 0`	R	`011 0011`	`010`	`000 0000`
	`sltu rd rs1 rs2`	Set Less Than (Unsigned)		R	`011 0011`	`011`	`000 0000`
	`addi rd rs1 imm`	ADD Immediate	`rd = rs1 + imm`	I	`001 0011`	`000`	
	`andi rd rs1 imm`	bitwise AND Immediate	`rd = rs1 & imm`	I	`001 0011`	`111`	
	`ori rd rs1 imm`	bitwise OR Immediate	`rd = rs1 | imm`	I	`001 0011`	`110`	
	`xori rd rs1 imm`	bitwise XOR Immediate	`rd = rs1 ^ imm`	I	`001 0011`	`100`	
	`slli rd rs1 imm`	Shift Left Logical Immediate	`rd = rs1 << imm`	I*	`001 0011`	`001`	`000 0000`
	`srli rd rs1 imm`	Shift Right Logical Immediate	`rd = rs1 >> imm` (Zero-extend)	I*	`001 0011`	`101`	`000 0000`
	`srai rd rs1 imm`	Shift Right Arithmetic Immediate	`rd = rs1 >> imm` (Sign-extend)	I*	`001 0011`	`101`	`010 0000`
	`slti rd rs1 imm`	Set Less Than Immediate (signed)	`rd = (rs1 < imm) ? 1 : 0`	I	`001 0011`	`010`	
	`sltiu rd rs1 imm`	Set Less Than Immediate (Unsigned)		I	`001 0011`	`011`	
Memory	`lb rd imm(rs1)`	Load Byte	`rd` = 1 byte of memory at address `rs1 + imm`, sign-extended	I	`000 0011`	`000`	
	`lbu rd imm(rs1)`	Load Byte (Unsigned)	`rd` = 1 byte of memory at address `rs1 + imm`, zero-extended	I	`000 0011`	`100`	
	`lh rd imm(rs1)`	Load Half-word	`rd` = 2 bytes of memory starting at address `rs1 + imm`, sign-extended	I	`000 0011`	`001`	
	`lhu rd imm(rs1)`	Load Half-word (Unsigned)	`rd` = 2 bytes of memory starting at address `rs1 + imm`, zero-extended	I	`000 0011`	`101`	
	`lw rd imm(rs1)`	Load Word	`rd` = 4 bytes of memory starting at address `rs1 + imm`	I	`000 0011`	`010`	
	`sb rs2 imm(rs1)`	Store Byte	Stores least-significant byte of `rs2` at the address `rs1 + imm` in memory	S	`010 0011`	`000`	
	`sh rs2 imm(rs1)`	Store Half-word	Stores the 2 least-significant bytes of `rs2` starting at the address `rs1 + imm` in memory	S	`010 0011`	`001`	
	`sw rs2 imm(rs1)`	Store Word	Stores `rs2` starting at the address `rs1 + imm` in memory	S	`010 0011`	`010`	

	Instruction	Name	Description	Type	Opcode	Funct3
Control	`beq rs1 rs2 label`	Branch if EQual	`if (rs1 == rs2) PC = PC + offset`	B	`110 0011`	`000`
	`bge rs1 rs2 label`	Branch if Greater or Equal (signed)	`if (rs1 >= rs2) PC = PC + offset`	B	`110 0011`	`101`
	`bgeu rs1 rs2 label`	Branch if Greater or Equal (Unsigned)		B	`110 0011`	`111`
	`blt rs1 rs2 label`	Branch if Less Than (signed)	`if (rs1 < rs2) PC = PC + offset`	B	`110 0011`	`100`
	`bltu rs1 rs2 label`	Branch if Less Than (Unsigned)		B	`110 0011`	`110`
	`bne rs1 rs2 label`	Branch if Not Equal	`if (rs1 != rs2) PC = PC + offset`	B	`110 0011`	`001`
	`jal rd label`	Jump And Link	`rd = PC + 4` `PC = PC + offset`	J	`110 1111`	
	`jalr rd rs1 imm`	Jump And Link Register	`rd = PC + 4` `PC = rs1 + imm`	I	`110 0111`	`000`
Other	`auipc rd imm`	Add Upper Immediate to PC	`rd = PC + (imm << 12)`	U	`001 0111`	
	`lui rd imm`	Load Upper Immediate	`rd = imm << 12`	U	`011 0111`	
	`ebreak`	Environment BREAK	Asks the debugger to do something (`imm = 0`)	I	`111 0011`	`000`
	`ecall`	Environment CALL	Asks the OS to do something (`imm = 1`)	I	`111 0011`	`000`
Ext	`mul rd rs1 rs2`	MULtiply (part of mul ISA extension)	`rd = rs1 * rs2`	(omitted)		

#	Name	Description	#	Name	Desc
`x0`	`zero`	Constant 0	`x16`	`a6`	*Args*
`x1`	`ra`	*Return Address*	`x17`	`a7`	
`x2`	`sp`	Stack Pointer	`x18`	`s2`	Saved Registers
`x3`	`gp`	Global Pointer	`x19`	`s3`	
`x4`	`tp`	Thread Pointer	`x20`	`s4`	
`x5`	`t0`	*Temporary Registers*	`x21`	`s5`	
`x6`	`t1`		`x22`	`s6`	
`x7`	`t2`		`x23`	`s7`	
`x8`	`s0`	Saved Registers	`x24`	`s8`	
`x9`	`s1`		`x25`	`s9`	
`x10`	`a0`	*Function Arguments or Return Values*	`x26`	`s10`	
`x11`	`a1`		`x27`	`s11`	
`x12`	`a2`	*Function Arguments*	`x28`	`t3`	*Temporaries*
`x13`	`a3`		`x29`	`t4`	
`x14`	`a4`		`x30`	`t5`	
`x15`	`a5`		`x31`	`t6`	
Caller saved registers					
Callee saved registers (except `x0`, `gp`, `tp`)					

Pseudoinstruction	Name	Description	Translation
`beqz rs1 label`	Branch if EQuals Zero	`if (rs1 == 0) PC = PC + offset`	`beq rs1 x0 label`
`bnez rs1 label`	Branch if Not Equals Zero	`if (rs1 != 0) PC = PC + offset`	`bne rs1 x0 label`
`j label`	Jump	`PC = PC + offset`	`jal x0 label`
`jr rs1`	Jump Register	`PC = rs1`	`jalr x0 rs1 0`
`la rd label`	Load absolute Address	`rd = &label`	`auipc`, `addi`
`li rd imm`	Load Immediate	`rd = imm`	`lui` (if needed), `addi`
`mv rd rs1`	MoVe	`rd = rs1`	`addi rd rs1 0`
`neg rd rs1`	NEGate	`rd = -rs1`	`sub rd x0 rs1`
`nop`	No OPeration	do nothing	`addi x0 x0 0`
`not rd rs1`	bitwise NOT	`rd = ~rs1`	`xori rd rs1 -1`
`ret`	RETurn	`PC = ra`	`jalr x0 x1 0`



Immediates are sign-extended to 32 bits, except in I* type instructions and `sltiu`.

Selected ASCII values

HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR	HEX	DEC	CHAR
0x20	32	SPACE	0x30	48	0	0x40	64	@	0x50	80	P	0x60	96	`	0x70	112	p
0x21	33	!	0x31	49	1	0x41	65	A	0x51	81	Q	0x61	97	a	0x71	113	q
0x22	34	"	0x32	50	2	0x42	66	B	0x52	82	R	0x62	98	b	0x72	114	r
0x23	35	#	0x33	51	3	0x43	67	C	0x53	83	S	0x63	99	c	0x73	115	s
0x24	36	$	0x34	52	4	0x44	68	D	0x54	84	T	0x64	100	d	0x74	116	t
0x25	37	%	0x35	53	5	0x45	69	E	0x55	85	U	0x65	101	e	0x75	117	u
0x26	38	&	0x36	54	6	0x46	70	F	0x56	86	V	0x66	102	f	0x76	118	v
0x27	39	'	0x37	55	7	0x47	71	G	0x57	87	W	0x67	103	g	0x77	119	w
0x28	40	(	0x38	56	8	0x48	72	H	0x58	88	X	0x68	104	h	0x78	120	x
0x29	41	)	0x39	57	9	0x49	73	I	0x59	89	Y	0x69	105	i	0x79	121	y
0x2A	42	*	0x3A	58	:	0x4A	74	J	0x5A	90	Z	0x6A	106	j	0x7A	122	z
0x2B	43	+	0x3B	59	;	0x4B	75	K	0x5B	91	[	0x6B	107	k	0x7B	123	{
0x2C	44	,	0x3C	60	<	0x4C	76	L	0x5C	92	\	0x6C	108	l	0x7C	124	|
0x2D	45	-	0x3D	61	=	0x4D	77	M	0x5D	93	]	0x6D	109	m	0x7D	125	}
0x2E	46	.	0x3E	62	>	0x4E	78	N	0x5E	94	^	0x6E	110	n	0x7E	126	~
0x2F	47	/	0x3F	63	?	0x4F	79	O	0x5F	95	_	0x6F	111	o	0x00	0	NULL

C Format String Specifiers

Specifier	Output
d or i	Signed decimal integer
u	Unsigned decimal integer
o	Unsigned octal
x	Unsigned hexadecimal integer, lowercase
X	Unsigned hexadecimal integer, uppercase
f	Decimal floating point, lowercase
F	Decimal floating point, uppercase
e	Scientific notation (significand/exponent), lowercase
E	Scientific notation (significand/exponent), uppercase
g	Use the shortest representation: %e or %f
G	Use the shortest representation: %E or %F
a	Hexadecimal floating point, lowercase
A	Hexadecimal floating point, uppercase
c	Character
s	String of characters
p	Pointer address

IEEE 754 Floating Point Standard

	Sign	Exponent	Significand
Single Precision	1 bit	8 bits (bias = -127)	23 bits
Double Precision	1 bit	11 bits (bias = -1023)	52 bits
Quad Precision	1 bit	15 bits (bias = -16383)	112 bits

Standard exponent bias: $-(2^{E-1}-1)$ where E is the number of exponent bits

SI Prefixes

Size	Prefix	Symbol	Size	Prefix	Symbol	Size	Prefix	Symbol
10^{-3}	milli-	m	10^{3}	kilo-	k	2^{10}	kibi-	Ki
10^{-6}	micro-	µ	10^{6}	mega-	M	2^{20}	mebi-	Mi
10^{-9}	nano-	n	10^{9}	giga-	G	2^{30}	gibi-	Gi
10^{-12}	pico-	p	10^{12}	tera-	T	2^{40}	tebi-	Ti
10^{-15}	femto-	f	10^{15}	peta-	P	2^{50}	pebi-	Pi
10^{-18}	atto-	a	10^{18}	exa-	E	2^{60}	exbi-	Ei
10^{-21}	zepto-	z	10^{21}	zetta-	Z	2^{70}	zebi-	Zi
10^{-24}	yocto-	y	10^{24}	yotta-	Y	2^{80}	yobi-	Yi

Laws of Boolean Algebra

$$x \cdot \overline{x} = 0 \qquad x + \overline{x} = 1 \qquad (xy)z = x\,(yz)$$
$$x \cdot 0 = 0 \qquad x + 1 = 1 \qquad (x + y) + z = x + (y + z)$$
$$x \cdot 1 = x \qquad x + 0 = x \qquad x\,(y + z) = xy + xz$$
$$x \cdot x = x \qquad x + x = x \qquad x + yz = (x + y)\,(x + z)$$
$$x \cdot y = y \cdot x \qquad x + y = y + x \qquad \overline{x \cdot y} = \overline{x} + \overline{y}$$
$$xy + x = x \qquad (x + y)x = x \qquad \overline{(x + y)} = \overline{x} \cdot \overline{y}$$