Your Name: _____ SID: _____

	EECS151/251A **Spring 2022 Midterm**	
Name of the person on left (or aisle)	Proctor's Name	Name of the person on right (or aisle)

Question	1	2	3	4	**Total**
Max. points	7	13	24	26	**70**

Exam Notes:
You have 80 minutes to work, starting at 12:40PM Pacific Time and ending at 2:00PM Pacific Time.

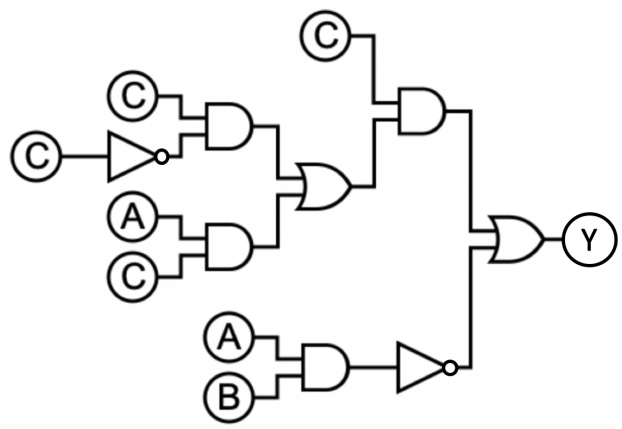
Before 12:40pm Pacific Time, you may write down your name, SID, names of the persons next to you, etc., on the first page but you may NOT begin working.

Please write your name and SID on EVERY page.

Problem 1: Boolean Simplification [7 Points]

Part a)

(1) Please write down the Boolean expression directly for the following circuit (match the logic gates below without simplifying). [2pts]



Answer:

$$Y = C(C\overline{C} + AC) + \overline{AB}$$

(2) Simplify the Boolean expression in part (1). [2pts]

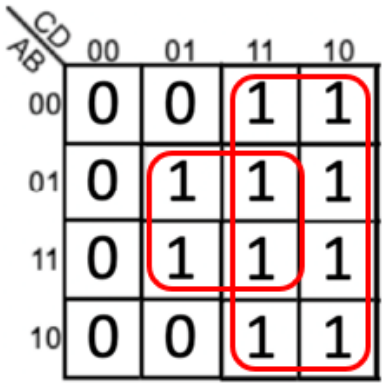
Answer:

$$Y = C + \overline{A} + \overline{B}$$

Part b)

Please translate the following K-map into Boolean expression and simplify. [3pts]

$$Y(A, B, C, D) =$$



Answer:

$$Y(A, B, C, D) = BD + C$$

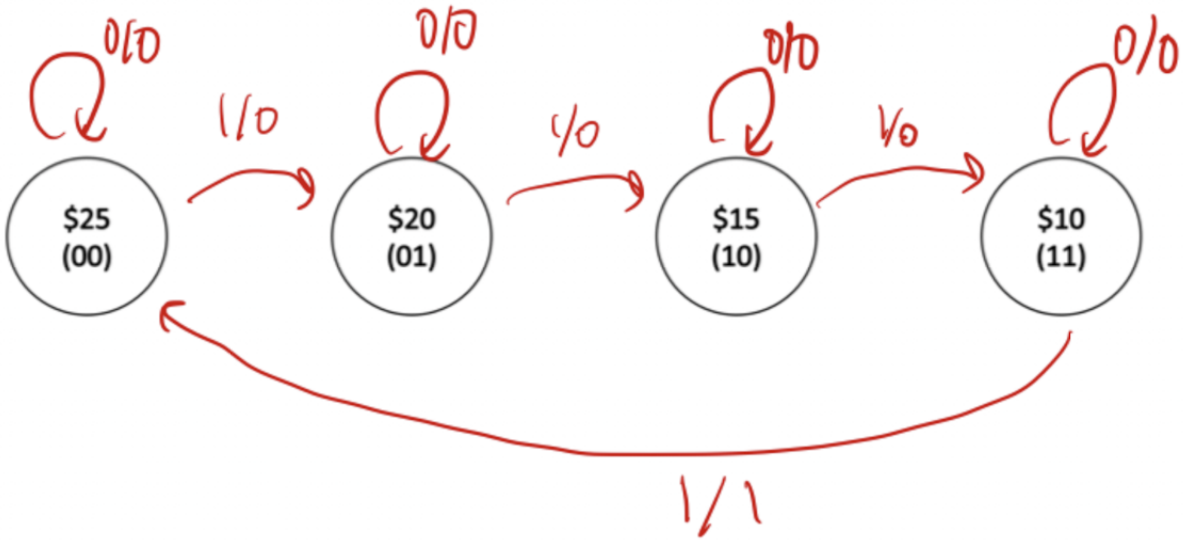
Get full credit if expression is correct (no need to check k-map); Get 1 point as partial credit if circled the k-map right but expression was wrong; Get 2 point as partial credit if the expression is not fully simplified

Problem 2: Finite State Machines: Fastrak [13 Points]

Part a)

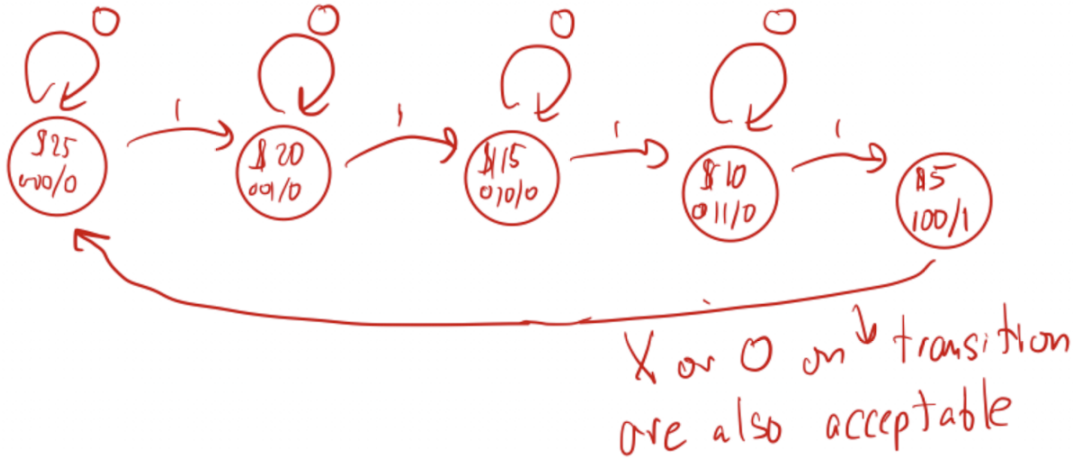
You are designing an FSM for a Fastrak (in-car toll tracking) device for the Bay Area. When you first purchase and place it in your car, you load it with $25. Every use of the toll bridge costs $5 and automatically charges from the device. Once the Fastrak device has reached $5, it reloads itself from your credit card with $20.

Draw a Mealy FSM with an input that is 1 for 1 cycle every time a toll bridge is used and an output that is 1 for 1 cycle every time a reload is needed. You may assume the reload always occurs by one cycle after the output is high. You will not incur tolls in consecutive cycles. [4pts]



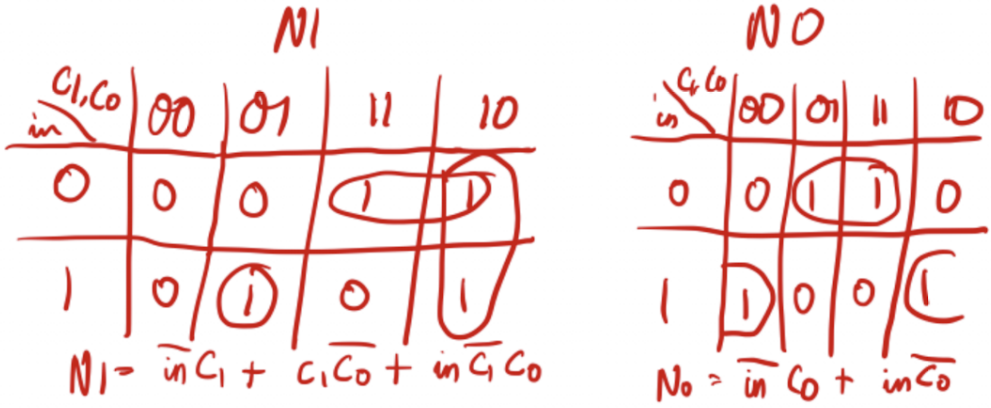
Part b)

Redraw the FSM for a Moore machine. [4pts]



Part c)

Find the minimum logic to determine the new state *n1,n0* where *n0* is the LSB, from the current state *c1,c0* where *c0* is the LSB, and the input *in* based on the Mealy machine. [3pts]



$N_1 = \overline{in}\,C_1 + C_1\overline{C_0} + in\,\overline{C_1}\,C_0$

$N_0 = \overline{in}\,C_0 + in\,\overline{C_0}$

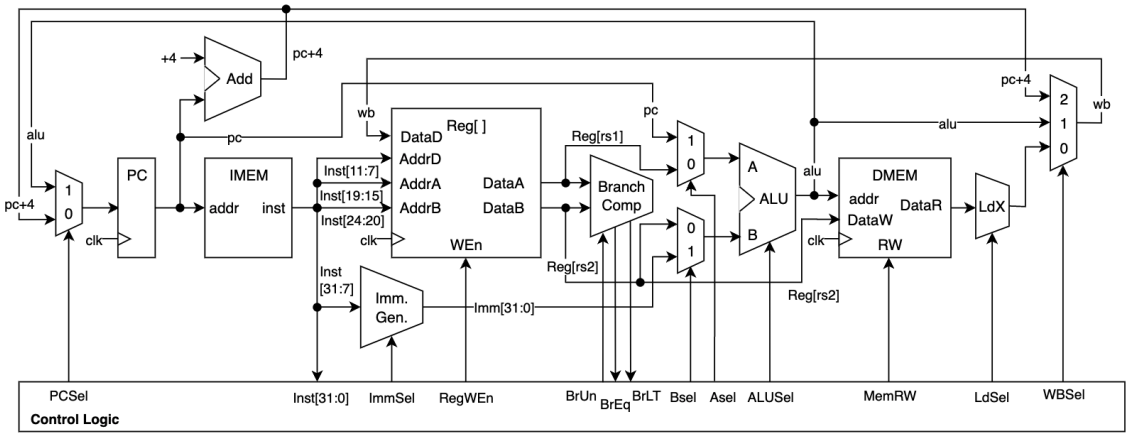
Part d)

Find the minimum logic to determine the output *out* from the current state *c1,c0*, where *c0* is the LSB, and the input *in* based on the Mealy Machine. [2pts]

$out = in\,C_1\,C_0$

Problem 3: RISC-V Instructions/Datapath [24 Points]

You may refer to the RISC-V Green Card on the last pages of this exam.



The single-cycle datapath above implements a subset of the RV32I instruction set.

Part a)

In the fabrication of any digital circuit, there may be manufacturing defects. One type involves a signal being shorted to GND or VDD (stuck-at-zero or stuck-at-one). Among the provided RISC-V instructions, please mark **N** in the table entries to the instructions that will **no longer work for the following stuck signals**, and leave others as blank. [12pts]

-	ASel is GND	BSel is VDD	RegWEn is GND
beq	N		
and		N	N
jal	N		N
auipc	N		N
lw			N
sw			
addi			N
jalr			N

0.5pt per block

Part b)

We want to implement a function called `maxpool_bias`, that pools out maximum value of a given array `arr`, with length of `LEN` and then add with integer `bias` with the final value. The following code describes its operation.

```
def maxpool_bias(arr, bias):
    c = arr[0]
    for i in range(1, LEN):
        c = max(c, arr[i])
    c += bias
```

Assume that intermediate values of `c`, and `arr` elements are stored in register file, so that you do not need to access memory (no need for `sw` or `lw`). The length of the array `LEN` is 3 (each array element pre-loaded in register `rs1, rs2, rs3`), `bias` is pre-loaded in register `rs4`, and result c is written to register `rd`.

For simplicity, we will be using the following unrolled version:

```
c = arr[0]
c = max(c, arr[1])  # Iteration 1
c = max(c, arr[2])  # Iteration 2
c += bias           # Final stage
```

(1) Write the RISCV instruction set flows to complete `maxpool_bias` operation described as the above code block without changing any datapath or control signals. Use only the register `rs1, rs2, rs3, rs4, rd`. You might not need to use all the lines provided. [4pts]

```
                add rd, rs1, x0
   Iteration1:

   _____

   _____

   _____

   Iteration2:

   _____

   _____

   _____

   FinalStage:

   _____

   _____

   _____
```

```
            add rd, rs1, x0
Iteration1: BLT rs2, rd, Iteration2
            add rd, rs2, x0
Iteration2: BLT rs3, rd, FinalStage
            add rd, rs3, x0
FinalStage: add rd, rd, rs4
```

-1pt per wrong instruction and/or factor

(2) Calculate the number of cycles it would take for the single cycle datapath to finish the operation. [2pts]
1pt each. No partial credit for wrong answer due to wrong (1)

Min: 4 cycles

Max: 6 cycles

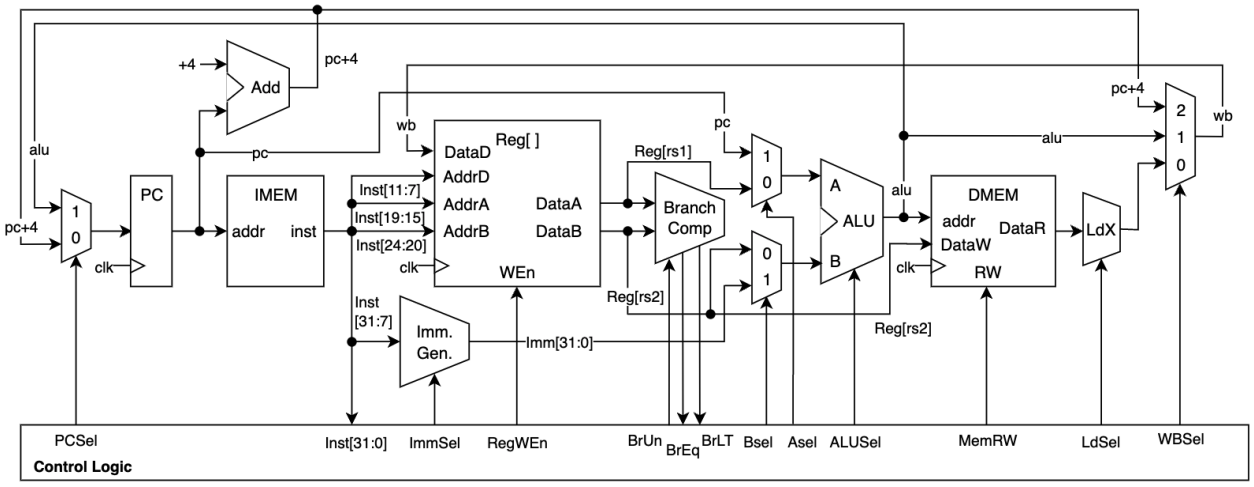
Part c)

This time, we implement a new instruction, `max` that pools out maximum value between two registers and returns it to the destination register by only modifying existing hardware components and control signals.

(1) What type of instruction is the new instruction? [1pt]

Answer: R type

(2) Describe the hardware and control signal change to support this new instruction. You can mark your changes in the following datapath diagram. [4pts]



It is an open ended question, but here I propose R type instruction for new ISA and changing ALU (1pt) to have max operation. The control signal change would be ALUSel (1pt). Need to mention either how the selection signal would be derived or which selection signal would be selected (1pt) e.g. ALUSel = max when ISA's opcode == MAX. Also, need to describe hardware change (1pt) either by drawing on the diagram or using verilog code, pseudo code etc. e.g. add max operation to ALU logic, or out = A > B ? A : B; to ALU logic

1pt for selecting control signal, 1pt for selecting hardware component, 1pt each for elaboration for change of control signal and hardware component

(3) Calculate the number of cycles it would take for the `maxpool_bias` operation in part b with the new instruction and updated datapath. [1pt]

Answer: 4 cycles

Problem 4: Pipelining/Hazards [26 Points]

Part a)

Sasha wants to improve the performance of their single-cycle RISC-V datapath. After analyzing their design, Sasha breaks down the delay of each stage of their datapath. Additionally, Sasha calculates that adding a pipeline register would add 0.2ns of additional delay to the previous pipeline stage. They then consider potential three-stage and five-stage pipeline implementations, depicted in Figure 1. In these designs the register file is synchronously written to at the end of a cycle, and read asynchronously (*i.e.* a write from the writeback stage is only available on the following cycle.)

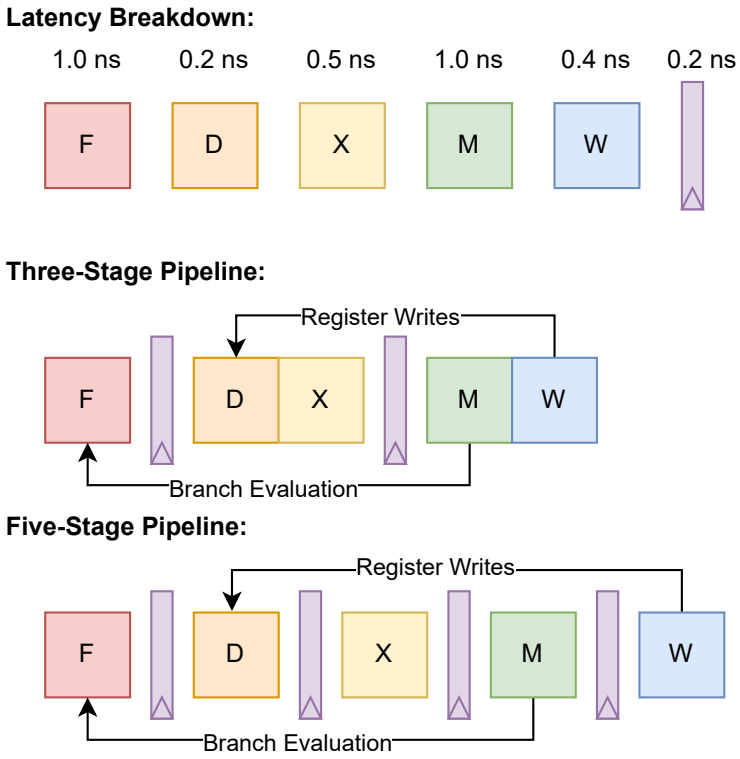


Figure 1: Potential CPU Pipelines

(1) Sasha wants to select an optimal CPU design. Assuming no hazards, what is the instruction latency (*i.e.* from fetch to writeback) for a single instruction for each design? What about the minimum clock period? [6pts]

CPU Implementation	**Instruction Latency (ns)**	**Minimum Clock Period (ns)**
Single Cycle	3.1	3.1
Three-Stage	3.5 or 4.2	1.4
Five-Stage	3.9 or 6.0	1.2

(2) If Sasha wants to optimize their design to maximize instruction throughput (*i.e.* instructions executed per unit time) assuming no hazards, which design should they choose and why? [2pts]

Sasha should use the five-stage pipelined design because it has the lowest clock period, and all have a CPI of 1.

Part b)

(1) Consider the following assembly program. Fill in the pipeline tables below for both the three-stage and five-stage implementation, factoring in hazards and assuming no forwarding/branch prediction. [12pts]

Some columns of the table may be left blank. You may abbreviate the combined decode/execute stage as "D" and the combined memory/writeback stage as "M".

```
add  x1, x2, x3
sub  x4, x5, x1
xor  x2, x3, x1
bne  x1, x2, not_t // branch is not taken
ori  x1, x2, 5
```

Your Name: _____ SID: _____

Three-Stage Pipeline:

Instruction\Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`add x1, x2, x3`	F	D	M																
`sub x4, x5, x1`		F	D	D	M														
`xor x2, x3, x1`			F	F	D	M													
`bne x1, x2, not_t`					F	D	D	M											
`ori x1, x2, 5`									F	D	M								

Five-Stage Pipeline:

Instruction\Cycle	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`add x1, x2, x3`	F	D	X	M	W														
`sub x4, x5, x1`		F	D	D	D	D	X	M	W										
`xor x2, x3, x1`			F	F	F	F	D	X	M	W									
`bne x1, x2, not_t`							F	D	D	D	D	X	M	W					
`ori x1, x2, 5`														F	D	X	M	W	

(2) For each implementation, how many cycles occur between the completion of the first instruction to the last? How much time elapses? [2pts]

CPU Implementation	**Cycles**	**Time (ns)**
Three-Stage	9	12.6
Five-Stage	14	16.8

(3) If Sasha wants to optimize their design to maximize instruction throughput for the given program, which design should they choose and why? [2pts]

Sasha should use the three-stage pipeline because it executes the program in the shortest time. This is because the lowered CPI of the five-stage pipeline offsets the increased clock frequency.

(4) Would this answer change given full forwarding/bypassing for data dependencies and a branch-not-taken predictor (assuming there is no implementation cost for either feature)? [2pts]

Sasha should use the five-stage pipeline because it has the greatest instruction throughput as the CPI is restored to 1 after resolving all hazards, making the design with the highest clock frequency better.

Also valid if the students consider the startup time of the program (though previously we measured from the completion of the first instruction to the last) and claim the three-stage design is still better.

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

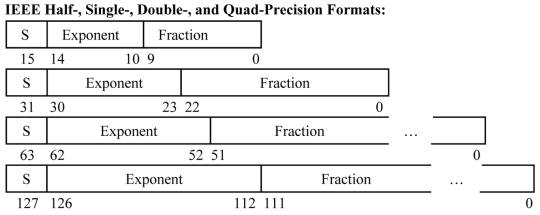
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return 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

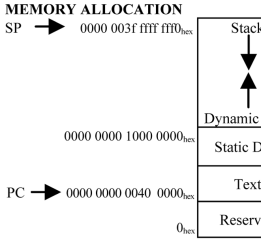
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

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

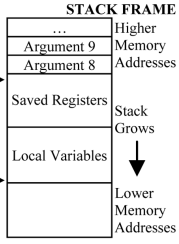
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



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