General Instructions

- You can download all required source files, that are provided as starting points for the following exercises, from Moodle.
- Each group should submit a zip (or tar.gz) file containing all the necessary source code files via Moodle.
- Each group should submit a report, written in French or English, in the **PDF** file format via Moodle.
- Each student has to be member of a group, where groups should generally consist of 3 students.

\- no bullet points
\- no late submissions
\- quote the text taken from somewhere else

1 RISC-V Instruction Set

Aims: *Understand the instruction set architecture and encoding of the RISC-V processor.*

Consider the following RISC-V program represented by the binary code of a simple function:

```
 0:   00050893
 4:   00068513
 8:   04088063
 c:   04058263
10:   04060063
14:   04d05063
18:   00088793
1c:   00269713
20:   00e888b3
24:   0007a703
28:   0005a803
2c:   01070733
30:   00e62023
34:   00478793
38:   00458593
3c:   00460613
40:   ff1792e3
44:   00008067
48:   fff00513
4c:   00008067
50:   fff00513
54:   00008067
```

- Use the attached cheat sheet (at the end of the assignment) in order to determine which RISC-V instructions appear in the program. Determine the instruction format for each instruction.
- Determine the operands for each instruction. For registers determine both, the register number and the symbolic register name.
- Search on the internet for an explanation of the concept "branch delay slots", which was popular in early implementations of the MIPS architecture. Explain the advantages and disadvantages, if any, of these branch delay slots.
- There are conditional branches in the function. Determine to which instructions they branch.
- What is the function actually doing? What is its return value?

2 RISC-V Tool Chain

Aims: *Understand the interplay between compiler and computer architecture.*

- Write a C program matching the program from above.
- Compile the program using the RISC-V compiler installed on the lab machines using the following command line:

```
riscv64-linux-gnu-gcc -g -O0 -mcmodel=medlow -mabi=ilp32
   -march=rv32im -Wall -c -o se201-prog.o se201-prog.c
```

- Disassemble the compiled program (`se201-prog.o`) with the `objdump` tool using the following command line:

```
riscv64-linux-gnu-objdump -d se201-prog.o
```

- Compare the resulting assembly code obtained from the `objdump` tool with the code from above. Explain why the code looks so differently?
- Try to change the compiler options (enable/disable optimizations using the option `-O0`, `-O`, or `-O3`) and see how this changes the code that you can see using the `objdump` tool.

- should get the same code as at 1. ?

3 RISC-V Architecture

Aims: *Understand RISC-V program execution on a pipelined processor.*

For the following exercises assume a RISC-V implementation as discussed in the lecture:

- The pipeline consists of 5 stages (`IF`, `ID`, `EX`, `MEM`, `WB`).
- Registers are read in the `ID` stage and written in the `WB` stage.
- Memory accesses are performed in the `MEM` stage.
  - The address computation is performed in the `EX` stage.
  - Data hazards between a memory load (in the `MEM` stage) and another instruction immediately using its results (in the `EX` stage) are resolved by stalling in the `ID` stage.
- Branches are performed in the `EX` stage.
  The two instructions following a branch are flushed when the branch is taken.
- For arithmetic instructions forwarding is performed as explained in the lecture.

3.1 Program Flow

- Given the program from Question 1, provide a list of instructions that are executed, along with a brief explanation of the processor/program state.

  Assume that the program starts with the following initial processor state:

  - Registers `a0`, `a1`, and `a2` all have the value `0x200`.
  - Register `a3` has the value `0x2`.
  - All other registers have the value zero (`0x0`).
  - The memory contents at the address range `0x200` through `0x210` is given as follows:

Address	Value
`0x200`	`0x61`
`0x204`	`0x20`
`0x208`	`0x62`
`0x20C`	`0x0`
`0x210`	`0x0`

  - All other memory cells have a value of zero (`0x0`).

  Provide a full list of instructions until the function terminates by executing a **`ret`** instruction. Explicitly mention hazards and how they are resolved, explain the address computations of branches and memory accesses. You can follow the example below:

PC	Instruction	a0	a1	a2	a3	a4	a5	a6	a7	Explanation
0x0	**mv** a7, a0	0x200	0x200	0x200	0x2	0x0	0x0	0x0	**0x200**	Copy value of a0 into a7
			...							

- explain in a table like this the instructions

3.2 Pipeline Diagram

- Draw a pipeline diagram showing all the instructions executed by the function as determined above. Assume a processor implementation as described above. Highlight all forms of hazards that occur and graphically distinguish resolution mechanisms (e.g., forwarding, stalls, flushing).

4 Processor Design - open format (design instruction set and processor)

Aims: *Explain and understand the instruction set of a processor and its implementation using a simple pipeline.*

4.1 Instruction Set Architecture

Describe the instruction set and the binary representation of the instructions of a simple processor. Your processor should respect the following list of characteristics:

- All instructions should be encoded in 16 bits. Apart from the instruction width, you are free to define the binary format yourself.
- Your processor should have 16 registers, i.e., encoding a register operand requires 4-bits. Assume that each register is 32-bit wide.
- The `PC` of your processor should be 32-bit wide.
- Your processor has separate instruction and data memories.
- You may chose whether the values of immediate operands of the various instructions are sign-extended or not.
- Define at least three different arithmetic/logic instructions operating on 3 register operands (reading 2 registers and writing 1).
- Define an instruction to read a 32-bit value from data memory (load). The instruction should take two register operands (reading 1 and writing 1) and a 5-bit immediate operand. The address used to access the memory is derived by adding the value of the read register to the immediate.
- Define an instruction to write a 32-bit value to data memory (store). The instruction should take two register operands (reading 2) and a 5-bit immediate operand. The address computation is the same as for loads.
- Define an instruction to copy an immediate value into a register.
- Define a conditional branch instruction having 1 register operand (read) and a 10-bit immediate operand. The branch is taken when the the register operand is non-zero. The new `PC` value is then computed as follows: $\mathtt{PC}_{\mathtt{new}} = \mathtt{PC}_{\mathtt{old}} + \mathtt{imm} * 2$. Untaken branches simply continue straight.
- Define an unconditional jump instruction having 1 register operand (read). The new `PC` value is obtained by copying the register operand's value into the `PC` register. The jump is always taken.
- Define a call instruction having 1 immediate operand with at least 9 bits. The old `PC` value should be stored in a fixed register of your choosing, i.e., this is not an operand! The new `PC` value is obtained by copying the immediate operand's value into the `PC` register. The call instruction behaves like a jump that is always taken.

- Conditional branches, unconditional jumps, and calls in your instruction set architecture have a branch delay slot for a single instruction.

Using the instruction set you just defined, please complete the following exercises and include your replies in the report:

- Define how the 16 registers have to be used by the programmer. In particular define how arguments are passed on function calls for functions with up to 4 arguments. Define how a to return from a function call and how the returned result of the function call can be retrieved. Which registers are preserved/or potentially modified during a function call.
- Describe each instruction of your processor. Explain what the instruction is doing, how it can be written in human readable form (assembly), and how it is encoded in binary form.
- Group instructions into binary formats, similar to the I-, R-, $\ldots$, and SB-format discussed for RISC-V in the lecture. Illustrate the formats using figures in your report.
- Define a no-operation instruction (similar to the **nop** instruction of RISC-V using one of the above instructions. This instruction should be a pseudo instruction that does not modify any registers.
- Provide the assembly code of a function that takes two arguments and returns the sum of those arguments. In addition, provide the code of a function which does not take any arguments and calls your previously defined function in order to compute the sum of 65408 and 134. Try to make good use of the branch delay slot and recall to save the return address!
- Translate the C-code from Question 1 to corresponding instructions of your processor. Arguments and the return value of the function are communicated through registers. The return address is likewise stored in a register. Your code should respect the register usage conventions that you have defined in the previous exercise from above. This may also require saving/register register values on the stack – depending on your register usage convention. Try to use the instructions of your processor as good as possible in order to minimize the number of instructions.

4.2 Pipelining

Now define the pipeline of your processor, while respecting the following characteristics:

- Your processor should have three pipeline stages: instruction fetch (`IF`), instruction decode (`ID`), and execute (`EX`).
- For arithmetic the three pipeline stages correspond, except for minor differences, to the pipeline stages of the RISC-V processor discussed in the lecture.
- Memory accesses are, however, different. The address computation and the memory access are both performed in the `EX` stage.
- Conditional branches, unconditional jumps, and calls should be executed in the `ID` stage.
- Assume that the processor registers are written at the beginning of the `EX` stage and read at the end of the `ID` stage.

Using the instruction set of your processor from the previous exercise and your pipeline design, please complete the following exercises and include your replies in the report:

- Draw a diagram of your processor's design. Use registers, pipeline registers, multiplexers, ALUs, . . . , as you need them. Describe relevant parts of the diagram. The diagram should contain everything that is necessary to execute **all** instructions that you have defined!
- Make a copy of your drawing that specifically highlights how a call instruction is executed by your pipeline design. Explain what happens in each pipeline stage. Notably, explain which control signals are used to control multiplexers, read/write registers, the ALU, et cetera. Also explain when and how these control signals are computed.
- Which kinds of hazards (data, control, or structural) can you encounter for your processor? Explain under which circumstances these hazards occur. How are these hazards resolved?
- Does your processor need logic to *flush* instructions from the pipeline (as discussed in the lecture)? Explain why this logic is needed or why it is not needed.

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

RISC-V Reference Data ①

RV32I BASE INTEGER INSTRUCTIONS, in alphabetical order

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
add	R	ADD	R[rd] = R[rs1] + R[rs2]	
addi	I	ADD Immediate	R[rd] = R[rs1] + imm	
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	
lb	I	Load Byte	R[rd] = {24'bM[](7),M[R[rs1]+imm](7:0)}	3) 4)
lbu	I	Load Byte Unsigned	R[rd] = {24'b0,M[R[rs1]+imm](7:0)}	
lh	I	Load Halfword	R[rd] = {16'bM[](15),M[R[rs1]+imm](15:0)}	
lhu	I	Load Halfword Unsigned	R[rd] = {16'b0,M[R[rs1]+imm](15:0)}	4)
lui	U	Load Upper Immediate	R[rd] = {imm, 12'b0}	
lw	I	Load Word	R[rd] = {M[R[rs1]+imm](31:0)}	
or	R	OR	R[rd] = R[rs1] | R[rs2]	
ori	I	OR Immediate	R[rd] = R[rs1] | imm	4)
sb	S	Store Byte	M[R[rs1]+imm](7:0) = R[rs2](7:0)	
sh	S	Store Halfword	M[R[rs1]+imm](15:0) = R[rs2](15:0)	
sll	R	Shift Left	R[rd] = R[rs1] << R[rs2]	
slli	I	Shift Left Immediate	R[rd] = R[rs1] << imm	
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	
sltu	R	Set Less Than Unsigned	R[rd] = (R[rs1] < R[rs2]) ? 1 : 0	
sra	R	Shift Right Arithmetic	R[rd] = R[rs1] >> R[rs2]	
srai	I	Shift Right Arith Imm	R[rd] = R[rs1] >> imm	2)
srl	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	2)
srli	I	Shift Right Immediate	R[rd] = R[rs1] >> imm	5)
sub,subw	R	SUBtract (Word)	R[rd] = R[rs1] – R[rs2]	5)
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) Operation assumes unsigned integers (instead of 2's complement)
2) The least significant bit of the branch address in jalr is set to 0
3) (signed) Load instructions extend the sign bit of data to fill the 32-bit register
4) Replicates the sign bit to fill in the leftmost bits of the result during right shift
5) Multiply with one operand signed and one unsigned
6) The Single version does a single-precision operation using the rightmost 32 bits of a 64-bit F register
7) Classify writes a 10-bit mask to show which properties are true (e.g., –inf, -0,+0, +inf, denorm, ...)
8) 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

ARITHMETIC CORE INSTRUCTION SET ②

RV64M Multiply Extension

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

RV64F and RV64D Floating-Point Extensions

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
fld,flw	I	Load (Word)	F[rd] = M[R[rs1]+imm]	
fsd,fsw	S	Store (Word)	M[R[rs1]+imm] = F[rd]	
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)
fmnsub.s,fmnsub.d	R	Negative Multiply-ADD	F[rd] = –(F[rs1] * F[rs2] – F[rs3])	7)
fmnadd.s,fmnadd.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.d.s	R	Convert from SP to DP	F[rd] = single(F[rs1])	
fcvt.s.d	R	Convert from DP to 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 Atomic Extension

MNEMONIC	FMT	NAME	DESCRIPTION (in Verilog)	NOTE
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

can use in assembly - but not an actual instr

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
lbu	I	0000011	100		03/4
lhu	I	0000011	101		03/5
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
sb	S	0100011	000		23/0
sh	S	0100011	001		23/1
sw	S	0100011	010		23/2
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
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

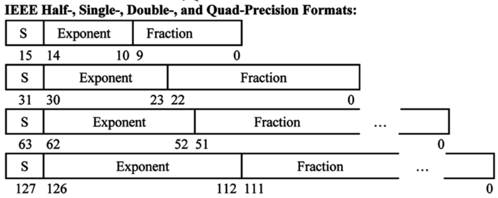
REGISTER	NAME	USE	SAVER
x0	zero	The constant value 0	N.A.
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	--
x4	tp	Thread pointer	--
x5-x7	t0-t2	Temporaries	Caller
x8	s0/fp	Saved register/Frame pointer	Callee
x9	s1	Saved register	Callee
x10-x11	a0-a1	Function arguments/Return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

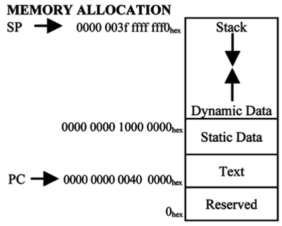
$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent - Bias})}$

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

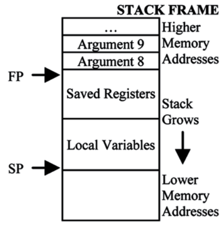
IEEE Half-, Single-, Double-, and Quad-Precision Formats:



MEMORY ALLOCATION



STACK FRAME



SIZE PREFIXES AND SYMBOLS

SIZE	PREFIX	SYMBOL	SIZE	PREFIX	SYMBOL
1000^1	Kilo-	K	2^{10}	Kibi-	Ki
1000^2	Mega-	M	2^{20}	Mebi-	Mi
1000^3	Giga-	G	2^{30}	Gibi-	Gi
1000^4	Tera-	T	2^{40}	Tebi-	Ti
1000^5	Peta-	P	2^{50}	Pebi-	Pi
1000^6	Exa-	E	2^{60}	Exbi-	Ei
1000^7	Zetta-	Z	2^{70}	Zebi-	Zi
1000^8	Yotta-	Y	2^{80}	Yobi-	Yi
1000^9	Ronna-	R	2^{90}	Robi-	Ri
1000^{10}	Quecca-	Q	2^{100}	Quebi-	Qi
1000^{-1}	milli-	m	1000^{-5}	femto-	f
1000^{-2}	micro-	µ	1000^{-6}	atto-	a
1000^{-3}	nano-	n	1000^{-7}	zepto-	z
1000^{-4}	pico-	p	1000^{-8}	yocto-	y
			1000^{-9}	ronto-	r
			1000^{-10}	quecto-	q

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