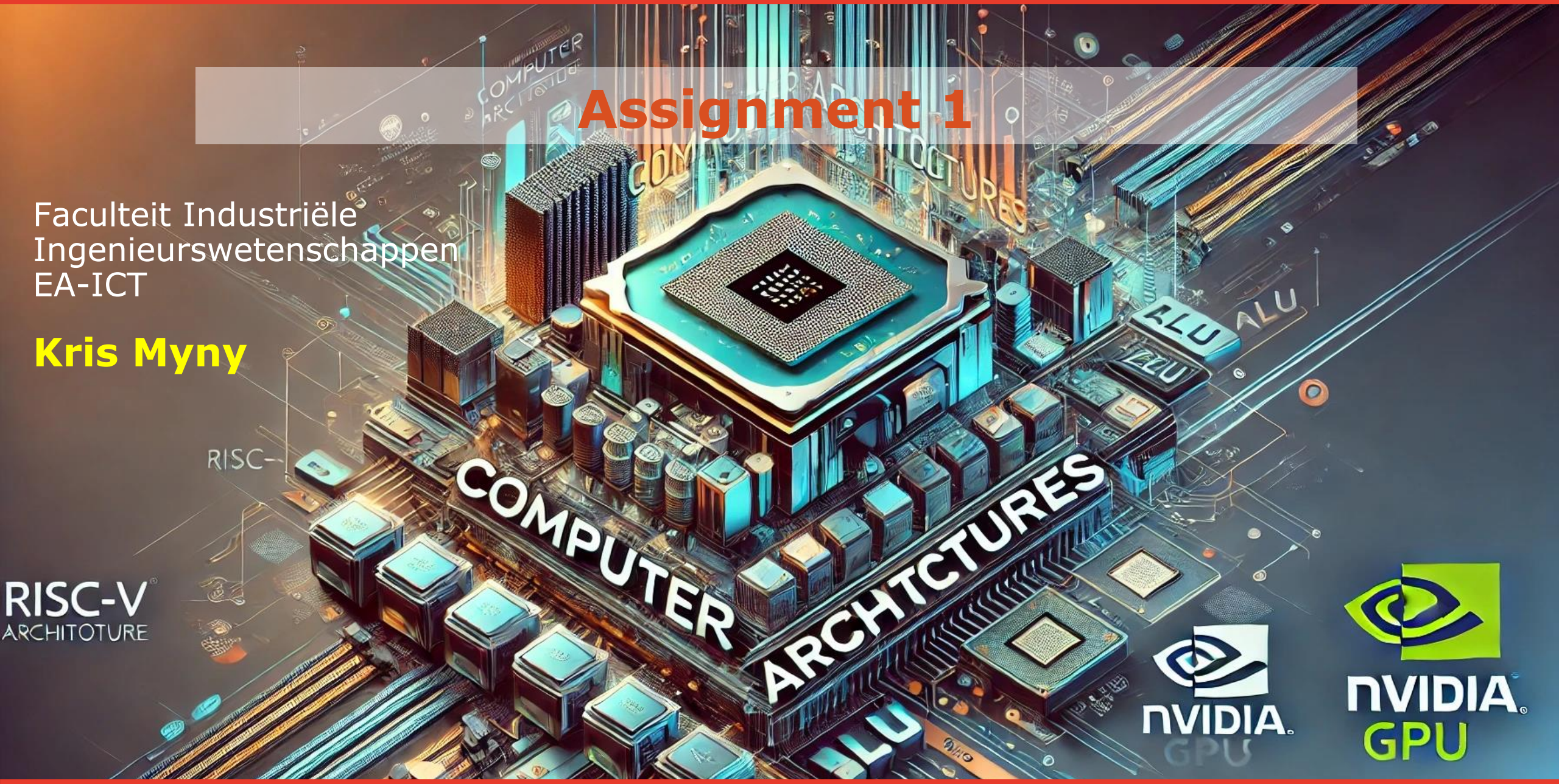
Assignment 1

Faculteit Industriële Ingenieurswetenschappen
EA-ICT

Kris Myny



Assignment 1

- You have to write assember code for three different programs.
- Use sufficient # to explain the program.
- Upload a small report (*Word or Latex*) showing your code and output for the assigment.
- By when? Sunday **September 28, 2025**

- Which programs?
  1. Calculate k = ((a - b) + (c - d) + e) #basic arithmetic
  2. A[6] = (A[5] + A[4]) >> 2 #array with shift
  3. While (save[i] == k) i += 1; #simple while loop

Your assembly programs

- Sufficient commenting
- Typical structure of a program:

```
.data
# this is your data

k: .word 0x0000
a: …

.text

Name of function:
Your assembly code
```

You can choose the value of the data yourself!

.data – store all data in memory

- Single words:

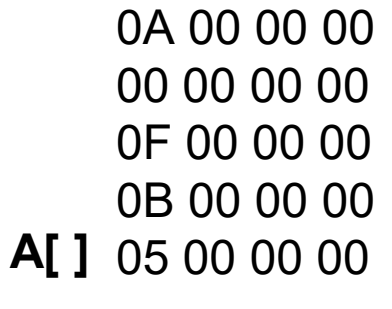
K: .word 0x0011 (hexadecimal)

G: .word 0b01110011 (binary)

H: .word 5 (decimal)

- Array:

A: .word 5, 11, 15, 0, 10 (he will make an array in the memory)




```
la x22, A  # load base address of a in X22
```

- Use la (load address) to load a base address in a register)

- Byte see next slide

.data

- Characters:

lab: .byte ‘h’, ‘e’, ‘l’, ‘l’, …, 0 (0 is marker for end)

G: .word 0b01110011 (binary)
H: .word 5 (decimal)

- Array

A: .word 5, 11, 15, 0, 10 (he will make an array in the memory)

Following .data give the same result in the memory:

- `lab1: .string "abc"`
- `lab: .byte 'a', 'b', 'c', 0`

Print an enter in the console

- .data

```
.data
enter_ascii: .byte 0x0A
```

- .text

```
lb t1, enter_ascii
mv a1, t1
li a0, 11
ecall
```

- Print a space: (ascii table!)
  - ASCII code 32

```
.byte 32
```

Some ecalls (environmental calls) for the simulator

To use an environmental call, load the ID into register `a0`, and load any arguments into `a1` - `a7`. Any return values will be stored in argument registers.

The following environmental calls are currently supported.

ID (`a0`)	Name	Description
1	print_int	prints integer in `a1`
4	print_string	prints the null-terminated string whose address is in `a1`
9	sbrk	allocates `a1` bytes on the heap, returns pointer to start in `a0`
10	exit	ends the program
11	print_character	prints ASCII character in `a1`
13	openFile	Opens the file in the VFS where a pointer to the path is in `a1` and the permission bits are in `a2`. Returns to a0 an integer representing the file descriptor.
14	readFile	Takes in: `a1` = FileDescriptor, `a2` = Where to store the data (an array), `a3` = the amount you want to read from the file. Returns `a0` = Number of items which were read and put to the given array. If it is less than `a3` it is either an error or EOF. You will have to use another ecall to determine what was the cause.
15	writeFile	Takes in: `a1` = FileDescriptor, `a2` = Buffer to read data from, `a3` = amount of the buffer you want to read, `a4` = Size of each item. Returns `a0` = Number of items written. If it is less than `a3` it is either an error or EOF. You will have to use another ecall to determine what was the cause. Also, you need to flush or close the file for the changes to be written to the VFS.
16	closeFile	Takes in: `a1` = FileDescriptor. Returns 0 on success and EOF (-1) otherwise. Will flush the data as well.
17	exit2	ends the program with return code in `a1`
18	fflush	Takes in: `a1` = FileDescriptor. Will return 0 on success otherwise EOF on an error.
19	feof	Takes in: `a1` = FileDescriptor. Returns a nonzero value when the end of file is reached otherwise, it returns 0.
20	ferror	Takes in: `a1` = FileDescriptor. Returns Nnnzero value if the file stream has errors occurred, 0 otherwise.
34	printHex	prints hex in `a1`
0x3CC	vlib	Please check out the vlib page to see what functions you can use!

The environmental calls are intended to be somewhat backwards compatible with SPIM's syscalls.

As an example, the following code prints the integer `42` to the console:

```
addi a0 x0 1        # print_int ecall
addi a1 x0 42       # integer 42
ecall
```

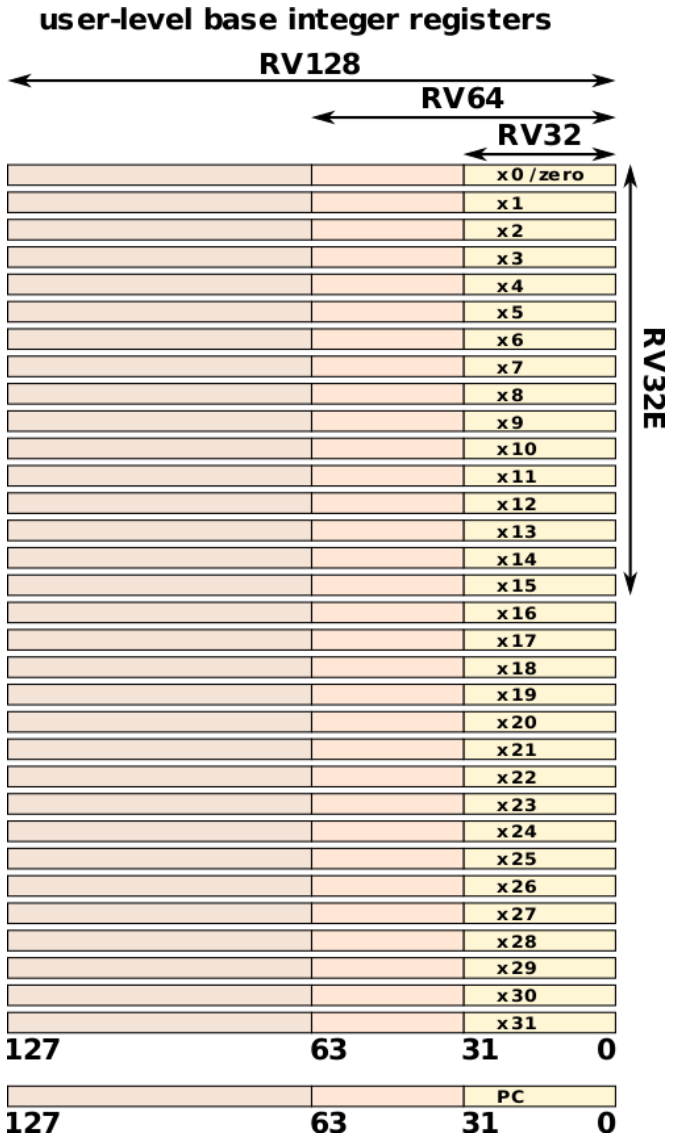
- Print in console in hexadecimal

mv a1, REGISTER WHERE YOU STORED YOUR FINAL DATA
li a0, 34
ecall

- Stop the simulator

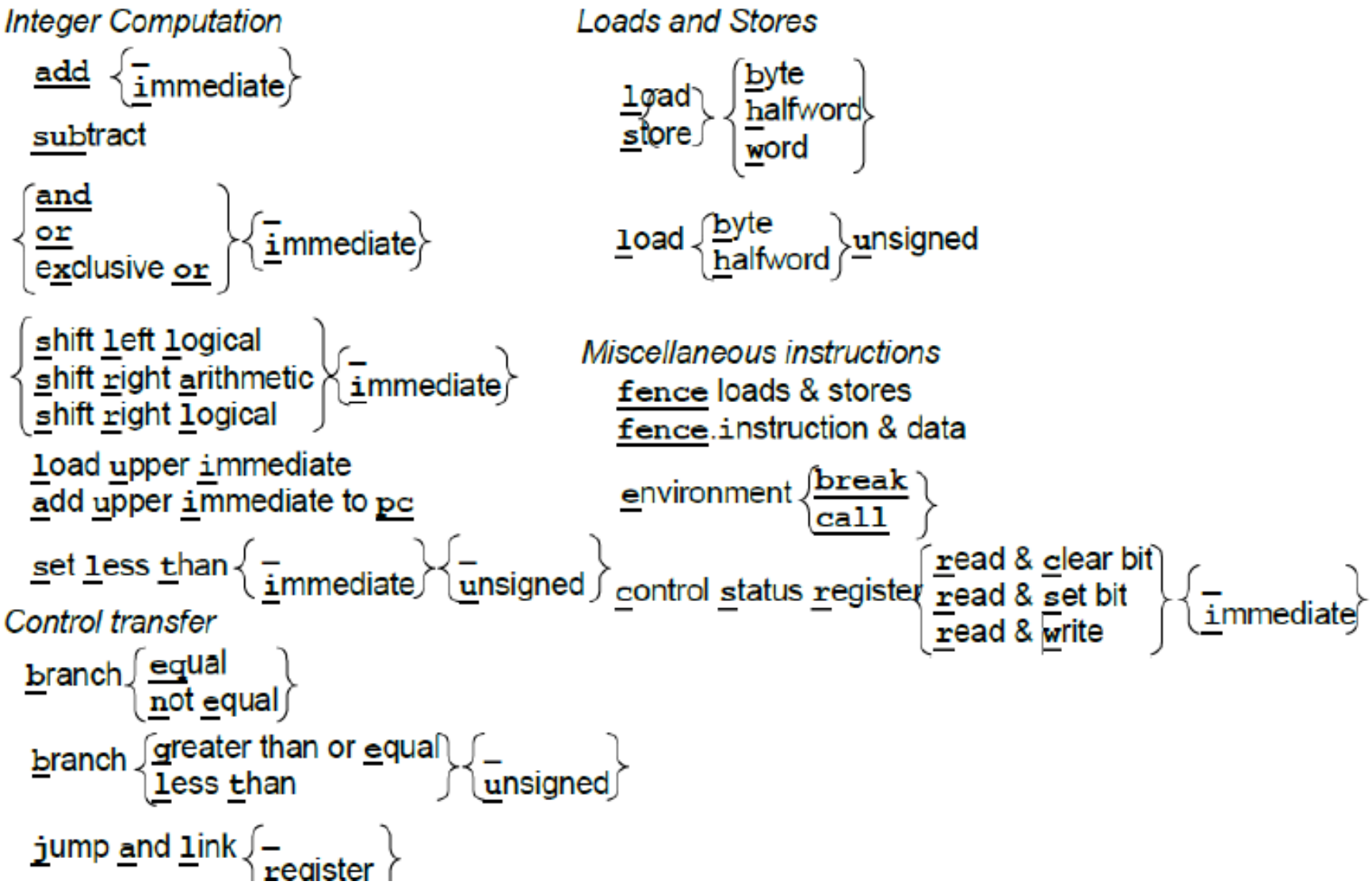
li a0, 10
ecall

Some instructions (RV32I)



- X0 ➔ 0x00000000 (zero)
- PC is a separate register

RV32I instructions diagram



Instructions formats and encoding

32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

- ❑ R: register-register operations (add, sub, xor)
- ❑ I: immediates and loads (addi, lw, srli)
- ❑ S: stores (sw, sb)
- ❑ B: conditional branches (beq, bge)
- ❑ I: long immediates (lui, auipc)
- ❑ J: unconditional jumps (jal)

RV32I instructions

- Basic set to be supported by all further extensions (such as RV32IM, RV64G, …)
- 47 unique instructions
- Eight ecall/ebreak/csrxx instructions can be implemented with a single SYSTEM hardware instruction that always traps (implemented in software); Implement fence and fence.i as nops → 38 instructions
- RV32I only 1/8 of the encoding space → space for ISA extensions

Registers

31 … 0	
x0 / zero	Hardwired zero
x1 / ra	Return address
x2 / sp	Stack pointer
x3 / gp	Global pointer
x4 / tp	Thread pointer
x5 / t0	Temporary
x6 / t1	Temporary
x7 / t2	Temporary
x8 / s0 / fp	Saved register, frame pointer
x9 / s1	Saved register
x10 / a0	Function argument, return value
x11 / a1	Function argument, return value
x12 / a2	Function argument
x13 / a3	Function argument
x14 / a4	Function argument
x15 / a5	Function argument
x16 / a6	Function argument
x17 / a7	Function argument
x18 / s2	Saved register
x19 / s3	Saved register
x20 / s4	Saved register
x21 / s5	Saved register
x22 / s6	Saved register
x23 / s7	Saved register
x24 / s8	Saved register
x25 / s9	Saved register
x26 / s10	Saved register
x27 / s11	Saved register
x28 / t3	Temporary
x29 / t4	Temporary
x30 / t5	Temporary
x31 / t6	Temporary

32

31 … 0
pc

32

- All registers x1…X32
  - Allow same operations
  - Functionally interchangeable
- ABI (application binary interface)
  - Register names: zero, ra, sp, …

RV32I instructions - arithmetic

Mnemonic	Instruction	Type	Description
`ADD rd, rs1, rs2`	Add	R	`rd ← rs1 + rs2`
`SUB rd, rs1, rs2`	Subtract	R	`rd ← rs1 - rs2`
`ADDI rd, rs1, imm12`	Add immediate	I	`rd ← rs1 + imm12`
`SLT rd, rs1, rs2`	Set less than	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTI rd, rs1, imm12`	Set less than immediate	I	`rd ← rs1 < imm12 ? 1 : 0`
`SLTU rd, rs1, rs2`	Set less than unsigned	R	`rd ← rs1 < rs2 ? 1 : 0`
`SLTIU rd, rs1, imm12`	Set less than immediate unsigned	I	`rd ← rs1 < imm12 ? 1 : 0`
`LUI rd, imm20`	Load upper immediate	U	`rd ← imm20 << 12`
`AUIP rd, imm20`	Add upper immediate to PC	U	`rd ← PC + imm20 << 12`

RV32I instructions - logical

Mnemonic	Instruction	Type	Description
`AND rd, rs1, rs2`	AND	R	`rd ← rs1 & rs2`
`OR rd, rs1, rs2`	OR	R	`rd ← rs1 | rs2`
`XOR rd, rs1, rs2`	XOR	R	`rd ← rs1 ^ rs2`
`ANDI rd, rs1, imm12`	AND immediate	I	`rd ← rs1 & imm12`
`ORI rd, rs1, imm12`	OR immediate	I	`rd ← rs1 | imm12`
`XORI rd, rs1, imm12`	XOR immediate	I	`rd ← rs1 ^ imm12`
`SLL rd, rs1, rs2`	Shift left logical	R	`rd ← rs1 << rs2`
`SRL rd, rs1, rs2`	Shift right logical	R	`rd ← rs1 >> rs2`
`SRA rd, rs1, rs2`	Shift right arithmetic	R	`rd ← rs1 >> rs2`
`SLLI rd, rs1, shamt`	Shift left logical immediate	I	`rd ← rs1 << shamt`
`SRLI rd, rs1, shamt`	Shift right logical imm.	I	`rd ← rs1 >> shamt`
`SRAI rd, rs1, shamt`	Shift right arithmetic immediate	I	`rd ← rs1 >> shamt`

RV32I instructions – load/store

Mnemonic	Instruction	Type	Description
LD rd, imm12(rs1)	Load doubleword	I	rd ← mem[rs1 + imm12]
LW rd, imm12(rs1)	Load word	I	rd ← mem[rs1 + imm12]
LH rd, imm12(rs1)	Load halfword	I	rd ← mem[rs1 + imm12]
LB rd, imm12(rs1)	Load byte	I	rd ← mem[rs1 + imm12]
LWU rd, imm12(rs1)	Load word unsigned	I	rd ← mem[rs1 + imm12]
LHU rd, imm12(rs1)	Load halfword unsigned	I	rd ← mem[rs1 + imm12]
LBU rd, imm12(rs1)	Load byte unsigned	I	rd ← mem[rs1 + imm12]
SD rs2, imm12(rs1)	Store doubleword	S	rs2 → mem[rs1 + imm12]
SW rs2, imm12(rs1)	Store word	S	rs2(31:0) → mem[rs1 + imm12]
SH rs2, imm12(rs1)	Store halfword	S	rs2(15:0) → mem[rs1 + imm12]
SB rs2, imm12(rs1)	Store byte	S	rs2(7:0) → em[rs1 + imm12]

RV32I instructions – branching

Mnemonic	Instruction	Type	Description
`BEQ rs1, rs2, imm12`	Branch equal	SB	`if rs1 == rs2` `  pc ← pc + imm12`
`BNE rs1, rs2, imm12`	Branch not equal	SB	`if rs1 != rs2` `  pc ← pc + imm12`
`BGE rs1, rs2, imm12`	Branch greater than or equal	SB	`if rs1 >= rs2` `  pc ← pc + imm12`
`BGEU rs1, rs2, imm12`	Branch greater than or equal unsigned	SB	`if rs1 >= rs2` `  pc ← pc + imm12`
`BLT rs1, rs2, imm12`	Branch less than	SB	`if rs1 < rs2` `  pc ← pc + imm12`
`BLTU rs1, rs2, imm12`	Branch less than unsigned	SB	`if rs1 < rs2` `  pc ← pc + imm12 << 1`
`JAL rd, imm20`	Jump and link	UJ	`rd ← pc + 4` `pc ← pc + imm20`
`JALR rd, imm12(rs1)`	Jump and link register	I	`rd ← pc + 4` `pc ← rs1 + imm12`

RV32I instructions – pseudo-instructions

Mnemonic	Instruction	Base instruction(s)
`LI rd, imm12`	Load immediate (near)	`ADDI rd, zero, imm12`
`LI rd, imm`	Load immediate (far)	`LUI rd, imm[31:12]` `ADDI rd, rd, imm[11:0]`
`LA rd, sym`	Load address (far)	`AUIPC rd, sym[31:12]` `ADDI rd, rd, sym[11:0]`
`MV rd, rs`	Copy register	`ADDI rd, rs, 0`
`NOT rd, rs`	One's complement	`XORI rd, rs, -1`
`NEG rd, rs`	Two's complement	`SUB rd, zero, rs`
`BGT rs1, rs2, offset`	Branch if rs1 > rs2	`BLT rs2, rs1, offset`
`BLE rs1, rs2, offset`	Branch if rs1 ≤ rs2	`BGE rs2, rs1, offset`
`BGTU rs1, rs2, offset`	Branch if rs1 > rs2 (unsigned)	`BLTU rs2, rs1, offset`
`BLEU rs1, rs2, offset`	Branch if rs1 ≤ rs2 (unsigned)	`BGEU rs2, rs1, offset`
`BEQZ rs1, offset`	Branch if rs1 = 0	`BEQ rs1, zero, offset`
`BNEZ rs1, offset`	Branch if rs1 ≠ 0	`BNE rs1, zero, offset`
`BGEZ rs1, offset`	Branch if rs1 ≥ 0	`BGE rs1, zero, offset`
`BLEZ rs1, offset`	Branch if rs1 ≤ 0	`BGE zero, rs1, offset`
`BGTZ rs1, offset`	Branch if rs1 > 0	`BLT zero, rs1, offset`
`J offset`	Unconditional jump	`JAL zero, offset`
`CALL offset12`	Call subroutine (near)	`JALR ra, ra, offset12`
`CALL offset`	Call subroutine (far)	`AUIPC ra, offset[31:12]` `JALR ra, ra, offset[11:0]`
`RET`	Return from subroutine	`JALR zero, 0(ra)`
`NOP`	No operation	`ADDI zero, zero, 0`

RISC-V reference card (I)

Free & Open RISC-V Reference Card ①

Base Integer Instructions: RV32I, RV64I, and RV128I

Category	Name	Fmt	RV32I Base	+RV{64,128}
Loads	Load Byte	I	`LB rd,rs1,imm`	
	Load Halfword	I	`LH rd,rs1,imm`	
	Load Word	I	`LW rd,rs1,imm`	`L{D|Q} rd,rs1,imm`
	Load Byte Unsigned	I	`LBU rd,rs1,imm`	
	Load Half Unsigned	I	`LHU rd,rs1,imm`	`L{W|D}U rd,rs1,imm`
Stores	Store Byte	S	`SB rs1,rs2,imm`	
	Store Halfword	S	`SH rs1,rs2,imm`	
	Store Word	S	`SW rs1,rs2,imm`	`S{D|Q} rs1,rs2,imm`
Shifts	Shift Left	R	`SLL rd,rs1,rs2`	`SLL{W|D} rd,rs1,rs2`
	Shift Left Immediate	I	`SLLI rd,rs1,shamt`	`SLLI{W|D} rd,rs1,shamt`
	Shift Right	R	`SRL rd,rs1,rs2`	`SRL{W|D} rd,rs1,rs2`
	Shift Right Immediate	I	`SRLI rd,rs1,shamt`	`SRLI{W|D} rd,rs1,shamt`
	Shift Right Arithmetic	R	`SRA rd,rs1,rs2`	`SRA{W|D} rd,rs1,rs2`
	Shift Right Arith Imm	I	`SRAI rd,rs1,shamt`	`SRAI{W|D} rd,rs1,shamt`
Arithmetic	ADD	R	`ADD rd,rs1,rs2`	`ADD{W|D} rd,rs1,rs2`
	ADD Immediate	I	`ADDI rd,rs1,imm`	`ADDI{W|D} rd,rs1,imm`
	SUBtract	R	`SUB rd,rs1,rs2`	`SUB{W|D} rd,rs1,rs2`
	Load Upper Imm	U	`LUI rd,imm`	
	Add Upper Imm to PC	U	`AUIPC rd,imm`	
Logical	XOR	R	`XOR rd,rs1,rs2`	
	XOR Immediate	I	`XORI rd,rs1,imm`	
	OR	R	`OR rd,rs1,rs2`	
	OR Immediate	I	`ORI rd,rs1,imm`	
	AND	R	`AND rd,rs1,rs2`	
	AND Immediate	I	`ANDI rd,rs1,imm`	
Compare	Set <	R	`SLT rd,rs1,rs2`	
	Set < Immediate	I	`SLTI rd,rs1,imm`	
	Set < Unsigned	R	`SLTU rd,rs1,rs2`	
	Set < Imm Unsigned	I	`SLTIU rd,rs1,imm`	
Branches	Branch =	SB	`BEQ rs1,rs2,imm`	
	Branch ≠	SB	`BNE rs1,rs2,imm`	
	Branch <	SB	`BLT rs1,rs2,imm`	

RV Privileged Instructions

Category	Name	RV mnemonic
CSR Access	Atomic R/W	`CSRRW rd,csr,rs1`
	Atomic Read & Set Bit	`CSRRS rd,csr,rs1`
	Atomic Read & Clear Bit	`CSRRC rd,csr,rs1`
	Atomic R/W Imm	`CSRRWI rd,csr,imm`
	Atomic Read & Set Bit Imm	`CSRRSI rd,csr,imm`
	Atomic Read & Clear Bit Imm	`CSRRCI rd,csr,imm`
Change Level	Env. Call	`ECALL`
	Environment Breakpoint	`EBREAK`
	Environment Return	`ERET`
Trap Redirect	to Supervisor	`MRTS`
	Redirect Trap to Hypervisor	`MRTH`
	Hypervisor Trap to Supervisor	`HRTS`
Interrupt	Wait for Interrupt	`WFI`
MMU	Supervisor FENCE	`SFENCE.VM rs1`

Optional Compressed (16-bit) Instruction Extension: RVC

Category	Name	Fmt	RVC	RVI equivalent
Loads	Load Word	CL	`C.LW rd',rs1',imm`	`LW rd',rs1',imm*4`
	Load Word SP	CI	`C.LWSP rd,imm`	`LW rd,sp,imm*4`
	Load Double	CL	`C.LD rd',rs1',imm`	`LD rd',rs1',imm*8`
	Load Double SP	CI	`C.LDSP rd,imm`	`LD rd,sp,imm*8`
	Load Quad	CL	`C.LQ rd',rs1',imm`	`LQ rd',rs1',imm*16`
	Load Quad SP	CI	`C.LQSP rd,imm`	`LQ rd,sp,imm*16`
Stores	Store Word	CS	`C.SW rs1',rs2',imm`	`SW rs1',rs2',imm*4`
	Store Word SP	CSS	`C.SWSP rs2,imm`	`SW rs2,sp,imm*4`
	Store Double	CS	`C.SD rs1',rs2',imm`	`SD rs1',rs2',imm*8`
	Store Double SP	CSS	`C.SDSP rs2,imm`	`SD rs2,sp,imm*8`
	Store Quad	CS	`C.SQ rs1',rs2',imm`	`SQ rs1',rs2',imm*16`
	Store Quad SP	CSS	`C.SQSP rs2,imm`	`SQ rs2,sp,imm*16`
Arithmetic	ADD	CR	`C.ADD rd,rs1`	`ADD rd,rd,rs1`

RISC-V reference card (II)

Branches	Branch =	SB	BEQ	rs1,rs2,imm
	Branch ≠	SB	BNE	rs1,rs2,imm
	Branch <	SB	BLT	rs1,rs2,imm
	Branch ≥	SB	BGE	rs1,rs2,imm
	Branch < Unsigned	SB	BLTU	rs1,rs2,imm
	Branch ≥ Unsigned	SB	BGEU	rs1,rs2,imm
Jump & Link	J&L	UJ	JAL	rd,imm
	Jump & Link Register	UJ	JALR	rd,rs1,imm
Synch	Synch thread	I	FENCE	
	Synch Instr & Data	I	FENCE.I	
System	System CALL	I	SCALL	
	System BREAK	I	SBREAK	
Counters	ReaD CYCLE	I	RDCYCLE	rd
	ReaD CYCLE upper Half	I	RDCYCLEH	rd
	ReaD TIME	I	RDTIME	rd
	ReaD TIME upper Half	I	RDTIMEH	rd
	ReaD INSTR RETired	I	RDINSTRET	rd
	ReaD INSTR upper Half	I	RDINSTRETH	rd

	Store Quad	CS	C.SQ rs1',rs2',imm	SQ rs1',rs2',imm*16
	Store Quad SP	CSS	C.SQSP rs2,imm	SQ rs2,sp,imm*16
Arithmetic	ADD	CR	C.ADD rd,rs1	ADD rd,rd,rs1
	ADD Word	CR	C.ADDW rd,rs1	ADDW rd,rd,imm
	ADD Immediate	CI	C.ADDI rd,imm	ADDI rd,rd,imm
	ADD Word Imm	CI	C.ADDIW rd,imm	ADDIW rd,rd,imm
	ADD SP Imm * 16	CI	C.ADDI16SP x0,imm	ADDI sp,sp,imm*16
	ADD SP Imm * 4	CIW	C.ADDI4SPN rd',imm	ADDI rd',sp,imm*4
	Load Immediate	CI	C.LI rd,imm	ADDI rd,x0,imm
	Load Upper Imm	CI	C.LUI rd,imm	LUI rd,imm
	MoVe	CR	C.MV rd,rs1	ADD rd,rs1,x0
	SUB	CR	C.SUB rd,rs1	SUB rd,rd,rs1
Shifts	Shift Left Imm	CI	C.SLLI rd,imm	SLLI rd,rd,imm
Branches	Branch=0	CB	C.BEQZ rs1',imm	BEQ rs1',x0,imm
	Branch≠0	CB	C.BNEZ rs1',imm	BNE rs1',x0,imm
Jump	Jump	CJ	C.J imm	JAL x0,imm
	Jump Register	CR	C.JR rd,rs1	JALR x0,rs1,0
Jump & Link	J&L	CJ	C.JAL imm	JAL ra,imm
	Jump & Link Register	CR	C.JALR rs1	JALR ra,rs1,0
System	Env. BREAK	CI	C.EBREAK	EBREAK

32-bit Instruction Formats

	31	30 25	24 21	20 19	15 14 12	11 8	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]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11] opcode
U	imm[31:12]						rd	opcode
UJ	imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd	opcode

16-bit (RVC) Instruction Formats

	15 14 13 12	11 10 9 8 7	6 5 4 3 2	1 0
CR	funct4	rd/rs1	rs2	op
CI	funct3 imm	rd/rs1	imm	op
CSS	funct3	imm	rs2	op
CIW	funct3	imm	rd′	op
CL	funct3 imm	rs1′	imm rd′	op
CS	funct3 imm	rs1′	imm rs2′	op
CB	funct3 offset	rs1′	offset	op
CJ	funct3	jump target		op

RISC-V Integer Base (RV32I/64I/128I), privileged, and optional compressed extension (RVC). Registers x1-x31 and the pc are 32 bits wide in RV32I, 64 in RV64I, and 128 in RV128I (x0=0). RV64I/128I add 10 instructions for the wider formats. The RVI base of <50 classic integer RISC instructions is required. Every 16-bit RVC instruction matches an existing 32-bit RVI instruction. See risc.org.

RISC-V reference card (III)

Load	Load	I	`FL{W,D,Q}`	`rd,rs1,imm`
Store	Store	S	`FS{W,D,Q}`	`rs1,rs2,imm`
Arithmetic	ADD	R	`FADD.{S|D|Q}`	`rd,rs1,rs2`
	SUBtract	R	`FSUB.{S|D|Q}`	`rd,rs1,rs2`
	MULtiply	R	`FMUL.{S|D|Q}`	`rd,rs1,rs2`
	DIVide	R	`FDIV.{S|D|Q}`	`rd,rs1,rs2`
	SQuare RooT	R	`FSQRT.{S|D|Q}`	`rd,rs1`
Mul-Add	Multiply-ADD	R	`FMADD.{S|D|Q}`	`rd,rs1,rs2,rs3`
	Multiply-SUBtract	R	`FMSUB.{S|D|Q}`	`rd,rs1,rs2,rs3`
	Negative Multiply-SUBtract	R	`FNMSUB.{S|D|Q}`	`rd,rs1,rs2,rs3`
	Negative Multiply-ADD	R	`FNMADD.{S|D|Q}`	`rd,rs1,rs2,rs3`
Sign Inject	SiGN source	R	`FSGNJ.{S|D|Q}`	`rd,rs1,rs2`
	Negative SiGN source	R	`FSGNJN.{S|D|Q}`	`rd,rs1,rs2`
	Xor SiGN source	R	`FSGNJX.{S|D|Q}`	`rd,rs1,rs2`
Min/Max	MINimum	R	`FMIN.{S|D|Q}`	`rd,rs1,rs2`
	MAXimum	R	`FMAX.{S|D|Q}`	`rd,rs1,rs2`
Compare	Compare Float =	R	`FEQ.{S|D|Q}`	`rd,rs1,rs2`
	Compare Float <	R	`FLT.{S|D|Q}`	`rd,rs1,rs2`
	Compare Float ≤	R	`FLE.{S|D|Q}`	`rd,rs1,rs2`
Categorization	Classify Type	R	`FCLASS.{S|D|Q}`	`rd,rs1`
Configuration	Read Status	R	`FRCSR`	`rd`
	Read Rounding Mode	R	`FRRM`	`rd`
	Read Flags	R	`FRFLAGS`	`rd`
	Swap Status Reg	R	`FSCSR`	`rd,rs1`
	Swap Rounding Mode	R	`FSRM`	`rd,rs1`
	Swap Flags	R	`FSFLAGS`	`rd,rs1`
	Swap Rounding Mode Imm	I	`FSRMI`	`rd,imm`
	Swap Flags Imm	I	`FSFLAGSI`	`rd,imm`

RISC-V Calling Convention

Register	ABI Name	Saver	Description
x0	zero	---	Hard-wired zero
x1	ra	Caller	Return address
x2	sp	Callee	Stack pointer
x3	gp	---	Global pointer
x4	tp	---	Thread pointer
x5-7	t0-2	Caller	Temporaries
x8	s0/fp	Callee	Saved register/frame pointer
x9	s1	Callee	Saved register
x10-11	a0-1	Caller	Function arguments/return values
x12-17	a2-7	Caller	Function arguments
x18-27	s2-11	Callee	Saved registers
x28-31	t3-t6	Caller	Temporaries
f0-7	ft0-7	Caller	FP temporaries
f8-9	fs0-1	Callee	FP saved registers
f10-11	fa0-1	Caller	FP arguments/return values
f12-17	fa2-7	Caller	FP arguments
f18-27	fs2-11	Callee	FP saved registers
f28-31	ft8-11	Caller	FP temporaries

RISC-V calling convention and five optional extensions: 10 multiply-divide instructions (RV32M); 11 optional atomic instructions (RV32A); and 25 floating-point instructions each for single-, double-, and quadruple-precision (RV32F, RV32D, RV32Q). The latter add registers f0-f31, whose width matches the widest precision, and a floating-point control and status register fcsr. Each larger address adds some instructions: 4 for RVM, 11 for RVA, and 6 each for RVF/D/Q. Using regex notation, { } means set, so `L{D|Q}` *is both* `LD` *and* `LQ`*. See risc.org. (8/21/15 revision)*

RISC-V reference card (IV)

Free & Open RISC-V Reference Card (riscv.org) ②

Optional Multiply-Divide Instruction Extension: RVM

Category	Name	Fmt	RV32M (Multiply-Divide)		+RV{64,128}	
Multiply	MULtiply	R	MUL	rd,rs1,rs2	MUL{W|D}	rd,rs1,rs2
	MULtiply upper Half	R	MULH	rd,rs1,rs2		
	MULtiply Half Sign/Uns	R	MULHSU	rd,rs1,rs2		
	MULtiply upper Half Uns	R	MULHU	rd,rs1,rs2		
Divide	DIVide	R	DIV	rd,rs1,rs2	DIV{W|D}	rd,rs1,rs2
	DIVide Unsigned	R	DIVU	rd,rs1,rs2		
Remainder	REMainder	R	REM	rd,rs1,rs2	REM{W|D}	rd,rs1,rs2
	REMainder Unsigned	R	REMU	rd,rs1,rs2	REMU{W|D}	rd,rs1,rs2

Optional Atomic Instruction Extension: RVA

Category	Name	Fmt	RV32A (Atomic)		+RV{64,128}	
Load	Load Reserved	R	LR.W	rd,rs1	LR.{D|Q}	rd,rs1
Store	Store Conditional	R	SC.W	rd,rs1,rs2	SC.{D|Q}	rd,rs1,rs2
Swap	SWAP	R	AMOSWAP.W	rd,rs1,rs2	AMOSWAP.{D|Q}	rd,rs1,rs2
Add	ADD	R	AMOADD.W	rd,rs1,rs2	AMOADD.{D|Q}	rd,rs1,rs2
Logical	XOR	R	AMOXOR.W	rd,rs1,rs2	AMOXOR.{D|Q}	rd,rs1,rs2
	AND	R	AMOAND.W	rd,rs1,rs2	AMOAND.{D|Q}	rd,rs1,rs2
	OR	R	AMOOR.W	rd,rs1,rs2	AMOOR.{D|Q}	rd,rs1,rs2
Min/Max	MINimum	R	AMOMIN.W	rd,rs1,rs2	AMOMIN.{D|Q}	rd,rs1,rs2
	MAXimum	R	AMOMAX.W	rd,rs1,rs2	AMOMAX.{D|Q}	rd,rs1,rs2
	MINimum Unsigned	R	AMOMINU.W	rd,rs1,rs2	AMOMINU.{D|Q}	rd,rs1,rs2
	MAXimum Unsigned	R	AMOMAXU.W	rd,rs1,rs2	AMOMAXU.{D|Q}	rd,rs1,rs2

Three Optional Floating-Point Instruction Extensions: RVF, RVD, & RVQ

Category	Name	Fmt	RV32{F|D|Q} (HP/SP,DP,QP Fl Pt)		+RV{64,128}	
Move	Move from Integer	R	FMV.{H|S}.X	rd,rs1	FMV.{D|Q}.X	rd,rs1
	Move to Integer	R	FMV.X.{H|S}	rd,rs1	FMV.X.{D|Q}	rd,rs1
Convert	Convert from Int	R	FCVT.{H|S|D|Q}.W	rd,rs1	FCVT.{H|S|D|Q}.{L|T}	rd,rs1
	Convert from Int Unsigned	R	FCVT.{H|S|D|Q}.WU	rd,rs1	FCVT.{H|S|D|Q}.{L|T}U	rd,rs1
	Convert to Int	R	FCVT.W.{H|S|D|Q}	rd,rs1	FCVT.{L|T}.{H|S|D|Q}	rd,rs1
	Convert to Int Unsigned	R	FCVT.WU.{H|S|D|Q}	rd,rs1	FCVT.{L|T}U.{H|S|D|Q}	rd,rs1

Assembler pseudo-instructions

Pseudoinstruction	Base Instruction(s)	Meaning
`nop`	`addi x0, x0, 0`	No operation
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`snez rd, rs`	`sltu rd, x0, rs`	Set if $\neq$ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if $<$ zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if $>$ zero
`beqz rs, offset`	`beq rs, x0, offset`	Branch if $=$ zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if $\neq$ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if $\leq$ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if $\geq$ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if $<$ zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if $>$ zero
`j offset`	`jal x0, offset`	Jump
`jr rs`	`jalr x0, rs, 0`	Jump register
`ret`	`jalr x0, x1, 0`	Return from subroutine
`tail offset`	`auipc x6, offset[31:12]` `jalr x0, x6, offset[11:0]`	Tail call far-away subroutine
`rdinstret[h] rd`	`csrrs rd, instret[h], x0`	Read instructions-retired counter
`rdcycle[h] rd`	`csrrs rd, cycle[h], x0`	Read cycle counter
`rdtime[h] rd`	`csrrs rd, time[h], x0`	Read real-time clock
`csrr rd, csr`	`csrrs rd, csr, x0`	Read CSR
`csrw csr, rs`	`csrrw x0, csr, rs`	Write CSR
`csrs csr, rs`	`csrrs x0, csr, rs`	Set bits in CSR
`csrc csr, rs`	`csrrc x0, csr, rs`	Clear bits in CSR
`csrwi csr, imm`	`csrrwi x0, csr, imm`	Write CSR, immediate
`csrsi csr, imm`	`csrrsi x0, csr, imm`	Set bits in CSR, immediate
`csrci csr, imm`	`csrrci x0, csr, imm`	Clear bits in CSR, immediate
`frcsr rd`	`csrrs rd, fcsr, x0`	Read FP control/status register
`fscsr rs`	`csrrw x0, fcsr, rs`	Write FP control/status register
`frrm rd`	`csrrs rd, frm, x0`	Read FP rounding mode
`fsrm rs`	`csrrw x0, frm, rs`	Write FP rounding mode
`frflags rd`	`csrrs rd, fflags, x0`	Read FP exception flags
`fsflags rs`	`csrrw x0, fflags, rs`	Write FP exception flags

Pseudoinstruction	Base Instruction(s)	Meaning
`lla rd, symbol`	`auipc rd, symbol[31:12]` `addi rd, rd, symbol[11:0]`	Load local address
`la rd, symbol`	*PIC*: `auipc rd, GOT[symbol][31:12]` `l{w|d} rd, rd, GOT[symbol][11:0]` *Non-PIC*: Same as `lla rd, symbol`	Load address
`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]` `l{b|h|w|d} rd, symbol[11:0](rd)`	Load global
`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `s{b|h|w|d} rd, symbol[11:0](rt)`	Store global
`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fl{w|d} rd, symbol[11:0](rt)`	Floating-point load global
`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]` `fs{w|d} rd, symbol[11:0](rt)`	Floating-point store global
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if $=$ zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if $>$
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if $\leq$
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if $>$, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if $\leq$, unsigned
`jal offset`	`jal x1, offset`	Jump and link
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`call offset`	`auipc x1, offset[31:12]` `jalr x1, x1, offset[11:0]`	Call far-away subroutine
`fence`	`fence iorw, iorw`	Fence on all memory and I/O
`fscsr rd, rs`	`csrrw rd, fcsr, rs`	Swap FP control/status register
`fsrm rd, rs`	`csrrw rd, frm, rs`	Swap FP rounding mode
`fsflags rd, rs`	`csrrw rd, fflags, rs`	Swap FP exception flags

UHASSELT KU LEUVEN

Assembler directives

Directive	Description
`.text`	Subsequent items are stored in the text section (machine code).
`.data`	Subsequent items are stored in the data section (global variables).
`.bss`	Subsequent items are stored in the bss section (global variables initialized to 0).
`.section .foo`	Subsequent items are stored in the section named `.foo`.
`.align n`	Align the next datum on a 2^n-byte boundary. For example, `.align 2` aligns the next value on a word boundary.
`.balign n`	Align the next datum on a n-byte boundary. For example, `.balign 4` aligns the next value on a word boundary.
`.globl sym`	Declare that label sym is global and may be referenced from other files.
`.string "str"`	Store the string str in memory and null-terminate it.
`.byte b1,..., bn`	Store the n 8-bit quantities in successive bytes of memory.
`.half w1,...,wn`	Store the n 16-bit quantities in successive memory halfwords.
`.word w1,...,wn`	Store the n 32-bit quantities in successive memory words.
`.dword w1,...,wn`	Store the n 64-bit quantities in successive memory doublewords.
`.float f1,..., fn`	Store the n single-precision floating-point numbers in successive memory words.
`.double d1,..., dn`	Store the n double-precision floating-point numbers in successive memory doublewords.
`.option rvc`	Compress subsequent instructions (see Chapter 7).
`.option norvc`	Don't compress subsequent instructions.
`.option relax`	Allow linker relaxations for subsequent instructions.
`.option norelax`	Don't allow linker relaxations for subsequent instructions.
`.option pic`	Subsequent instructions are position-independent code.
`.option nopic`	Subsequent instructions are position-dependent code.
`.option push`	Push the current setting of all `.options` to a stack, so that a subsequent `.option pop` will restore their value.
`.option pop`	Pop the option stack, restoring all `.options` to their setting at the time of the last `.option push`.