Module 2:

Processor architecture

Introduction to computers II

José Manuel Mendías Cuadros
Dpto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid

Outline

- ✓ The RISC-V architecture.
- ✓ Instructions and data.
- ✓ Memory model.
- ✓ Registers.
- ✓ Addressing modes.
- ✓ Instruction set.
- ✓ Extensions.
- ✓ RISC vs. CISC architectures.

These slides are based on:

- S.L. Harris and D. Harris. *Digital Design and Computer Architecture. RISC-V Edition.*
- D.A. Patterson and J.L. Hennessy. *Computer Organization and Design. RISC-V Edition.*

The RISC-V architecture

- RISC-V ISA (*Instruction Set Architecture*) is an architecture:
  - Open, not proprietary and in evolution.
  - Originally designed at UC Berkeley in 2010.
  - Currently coordinated by the RISC-V International consortium.
- It is a RISC-type architecture (*Reduced Instruction Set Computer*), and thus:
  - It has a reduced set of simple instructions.
  - Only the load and store instructions can access memory.
  - The rest of instructions work with data stored in registers.
  - It has a large number of general-purpose registers.
  - It has a reduced set of addressing modes.
  - Instructions has a fixed size, with a reduced number of formats.
- We will study the RV32I base set with the RVM extension.
  - 32-bit integer data and 32-bit instructions (RV32I).
  - With integer multiplication and division (RVM).

Instructions and data

- All RISC-V instructions have 32 bits.
- RISC-V instructions operate with 32-bit data or addresses.
  - Data are integer numbers (signed) or natural numbers (unsigned), encoded in two's complement or pure binary, respectively.
  - Addresses are natural numbers encoded in pure binary.
- However, it can work with smaller-width numbers:
  - Typically, they are extended to 32 bits before operating with them.
  - Depending on the case, they will be sign-extended (sExt) or zero-extended (zExt).
- The most common data sizes are:
  - Word: 32 bits.
  - Half word: 16 bits.
  - Byte: 8 bits.

Memory model

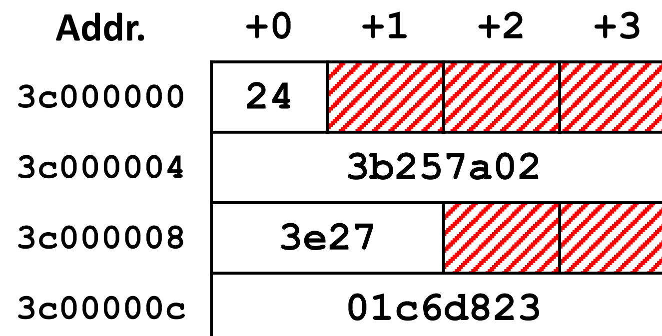
- It consists of a 4-GiB RAM main memory (2^{32}×8b = 2^{30}×32b):
  - 32-bit data and address buses.
  - Byte-addressable (each byte has a unique address).
  - It contains 8, 16 and 32-bit data and 32-bit instructions.
  - All of them are aligned and with little-endian organization.

Memory model

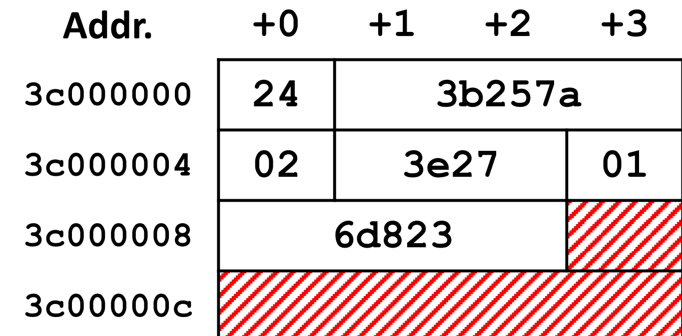
Alignment

- In the RISC-V memory, information is aligned, i.e., there are location constraints depending on its size.
  - Byte: it can be located in any address
  - Half word: it can only be located in multiple-of-2 addresses (even).
  - Word: it can only be located in multiple-of-4 addresses
    - This applies to 32-bit data and instructions.
  - In general, N-byte data must be located in multiple-of-N addresses.
  - When different-size data are stored consecutively, empty gaps are created.

Size	Data
byte	0x24
word	0x3b257a02
½ word	0x3e27
word	0x01c6d823



RISC-V: Aligned data



Not-aligned data

Memory model

Organization

- In the RISC-V memory, the word/half word bytes follow a little-endian organization:
  - The least significant byte is located at the lowest address, i.e., the data address coincides with its least significant byte address.
  - Bits within the byte keep the usual organization.
- Other processors can follow a big-endian organization:
  - The most significant byte is located at the lowest address, i.e., the data address coincides with its most significant byte address.

Size	Data
byte	0x24
word	0x3b257a02
½ word	0x3e27
word	0x01c6d823

Addr.	+0	+1	+2	+3
3c000000	24			
3c000004	02	7a	25	3b
3c000008	27	3e		
3c00000c	23	d8	c6	01

RISC-V: Little-Endian

Addr.	+0	+1	+2	+3
3c000000	24			
3c000004	3b	25	7a	02
3c000008	3e	27		
3c00000c	01	c6	d8	23

Big-Endian

Registers

- All data in a program are stored in memory, but in order to be used by RISC-V, they must be previously loaded in registers.

- A RISC-V has 32 general-purpose registers, each one with 32 bits.
  - They can be used interchangeably.
  - They are numbered from **`x0`** to **`x31`**.
  - The **`x0`** register, always contains constant 0, and any write operation in this register has no effect.
  - However, in order to simplify programming, each register has an alias that allows remembering its most conventional use.

- Besides, RISC-V has a special register, PC (*Program Counter*)
  - It contains the memory address of the instruction in execution.
  - After this instruction is executed, the PC is incremented +4 (each instruction takes 4Bytes)
    - Except if the executed instruction is a branch

Registers

# Reg.	Alias	Description
x0	zero	zero
x1	ra	return address
x2	sp	stack pointer
x3	gp	global pointer
x4	tp	thread pointer
x5…x7	t0…t2	temporary register
x8	s0/fp	saved register / frame pointer
x9	s1	saved register
x10…x17	a0…a7	argument register
x18…x27	s2…s11	saved register
x28…x31	t3…t6	temporary register

alias must be used always ***when programming in assembly***

Addressing modes

- The addressing modes are the mechanisms to indicate where the instruction operands are located
  - They indicate the data location and how to get them.
- The instruction operands can be located in:
  - The instruction itself.
  - A processor register, indicating which one.
  - The computer memory, indicating its memory address.
- There are only 4 addressing modes in RISC-V:
  - Immediate addressing: the operand is a constant located in the instruction.
  - Register addressing: the operand is located in a processor register.
  - Base addressing: the operand is located in the memory.
    - Its address is obtained by adding the content of a base register plus an offset.
  - PC-relative addressing: The operand is an address (branch target).
    - It is obtained by adding the content of the PC plus an offset.

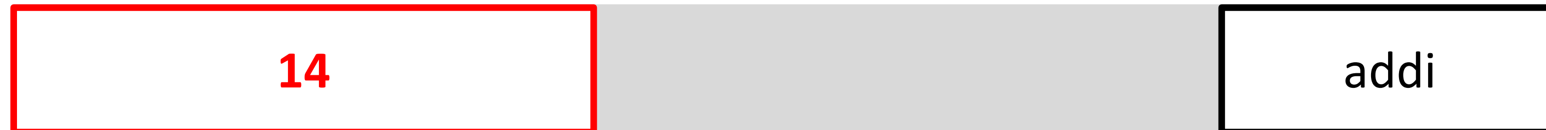
Addressing modes

Immediate addressing

- The operand is a constant contained in the instruction.
  - The constant is explicitly indicated in assembly:

```
addi x1, x1, 14
```

  - The machine instruction has a field where the constant is stored:



- Since instructions take 32b and the immediate operands are contained in them, constants always have a smaller width:
  - Unsigned 5-bit immediate: Used without extension.
  - Signed 12/13-bit immediate: Extended to 32 bits before using them.
    - If the constant has 13 bits, the instruction only stores the 12 most significant bits.
  - 20-bits immediate: Used without extension, but shifted.
  - 21-bit immediate: Extended to 32 bits before using them.
    - The instruction only stores the 20 most significant bits.

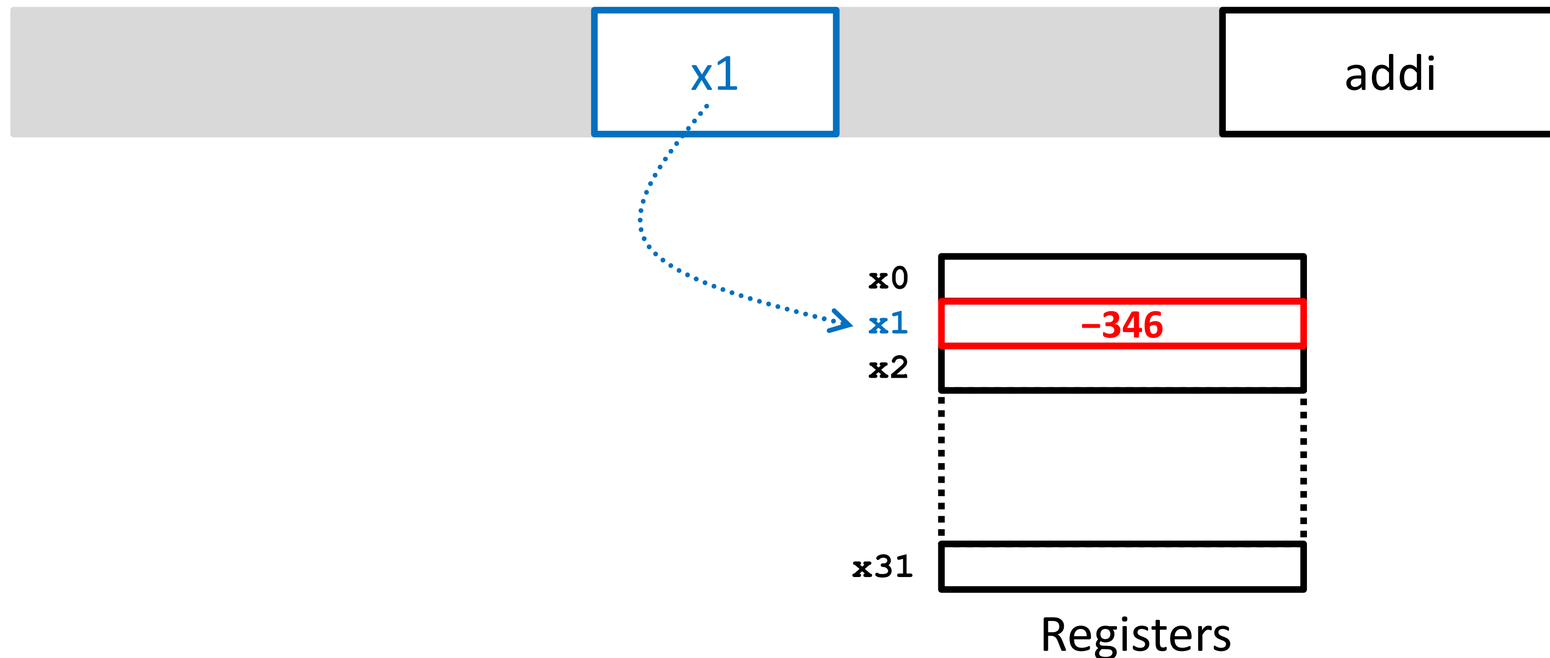
Addressing modes

Register addressing

- The operand is stored in a processor register.
  - The register name is indicated in assembly:

```
addi x1, x1, 14
```

  - The machine instruction has a field that indicates the register number:



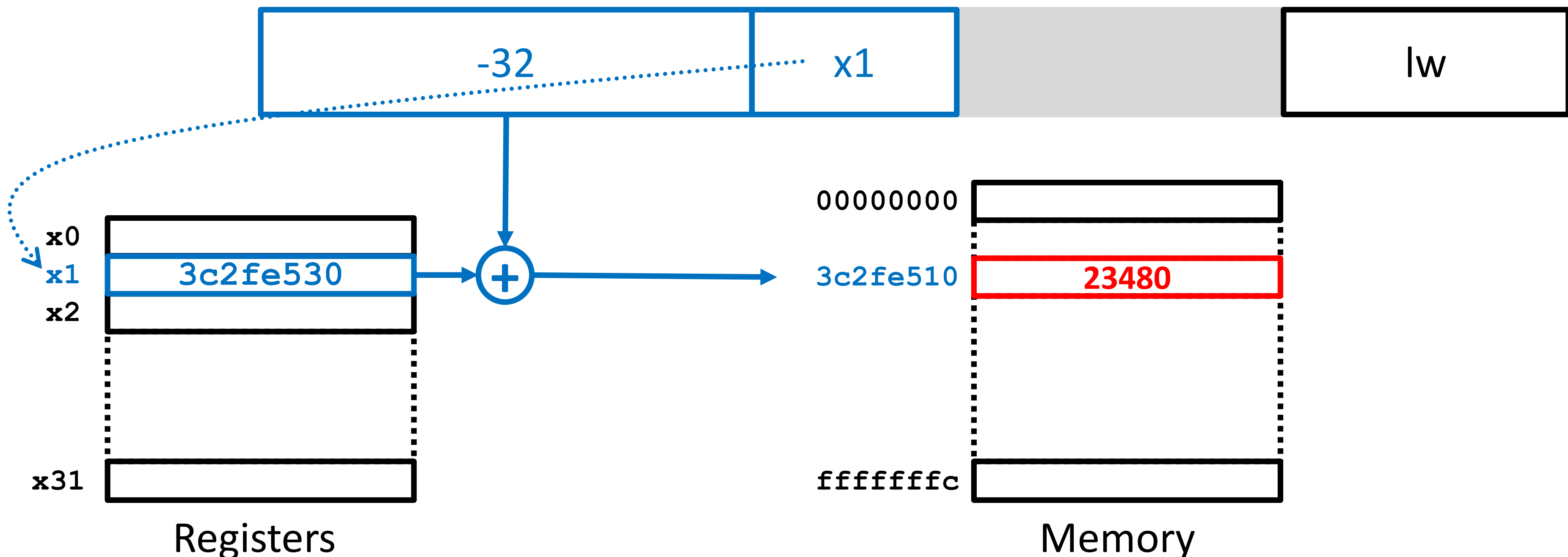
Addressing modes

Base addressing

- The operand is located in a memory position whose address is calculated:
  - Adding the content of a processor register (base register) with an offset (constant) indicated in the instruction
  - The offset and the register are indicated in assembly:

```
lw x3, -32(x1)
```

  - The machine instruction has fields to indicate both elements:

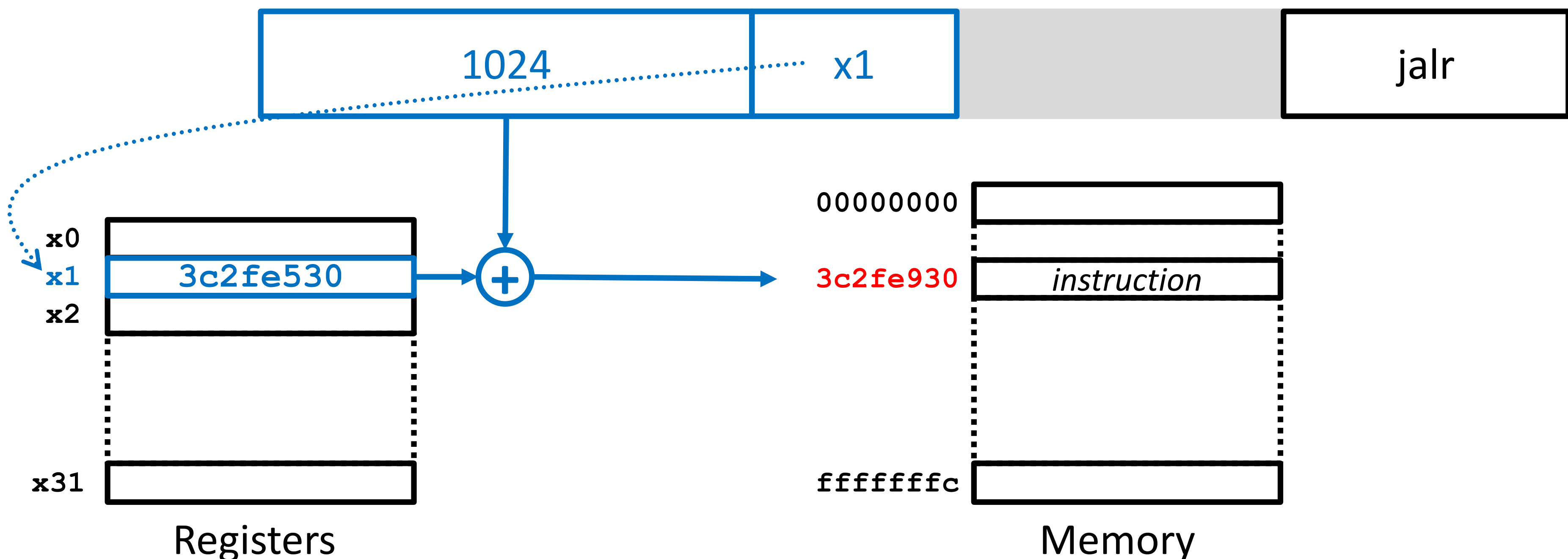


Addressing modes

Base addressing

- A particular case of this addressing mode is when the operand is the calculated address itself.
  - It is used in branch instructions.
  - The branch address is calculated in the same way: adding the content of a register with an offset contained in the instruction.

```
jalr x3, x1, 1024
```



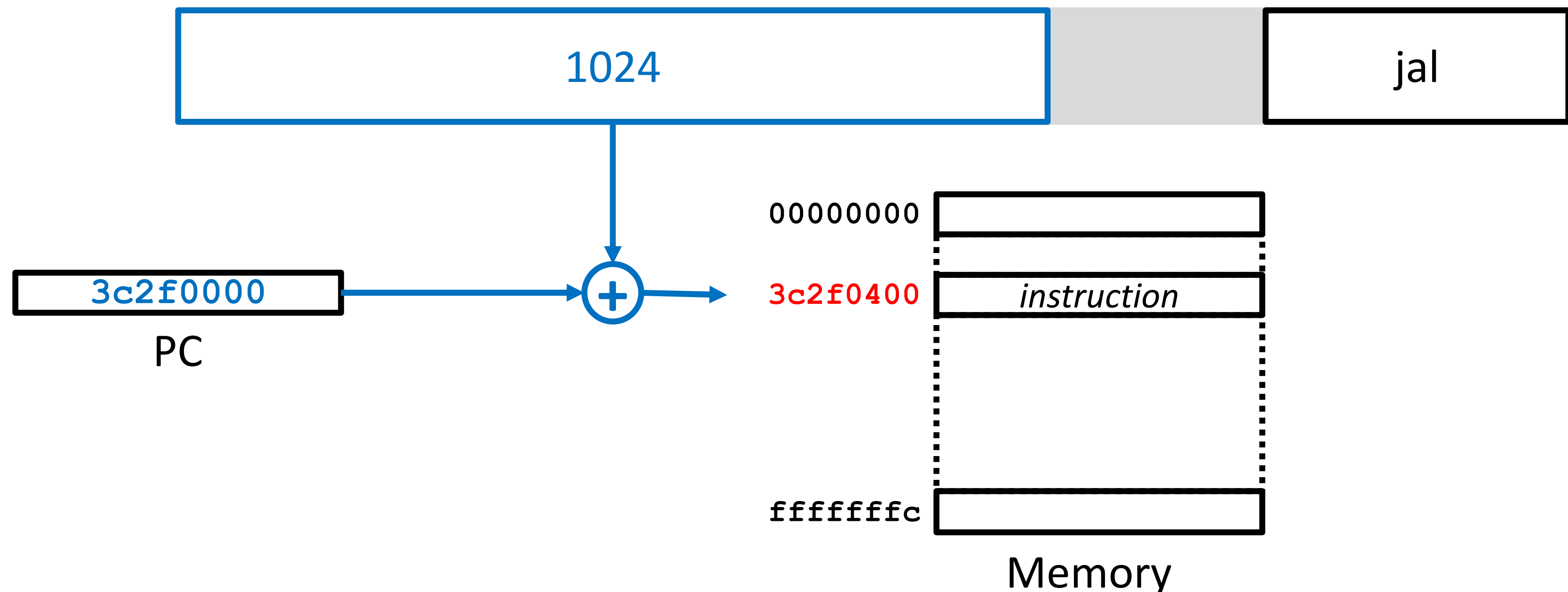
Addressing modes

PC-relative addressing

- The operand is a branch address that is calculated:
  - Adding the content of the PC (address of the instruction in execution) with an offset (constant) indicated in the instruction.
  - Only the offset is explicitly indicated in assembly:

```
jal x3, 1024
```

  - The machine instruction has a field that indicates the offset:



Addressing modes

About relative addressing

- There is no absolute (direct) addressing in RISC-V because relative addressing (PC or base) is more convenient in most cases.
  - Absolute addressing: the instruction contains the explicit memory address where the data/instruction is located.
  - Absolute addressing requires 32 bits to indicate the address.
  - In relative addressing only the difference between two instructions is indicated, which usually requires fewer bits to be encoded.
- Offsets may be short immediate because:
  - For instructions, the usual case is to branch to nearby addresses.
    - Which implies a short offset from the PC
  - For data, these usually gather in an adjacent memory region.
    - If the start address of this region is stored in a base register, all data could be accessed with small offsets relative to that base register
- Besides, PC-relative addressing allows relocatable code:
  - The calculation of branch addresses is always correct, regardless of the memory address in which the program is located.

Instruction set

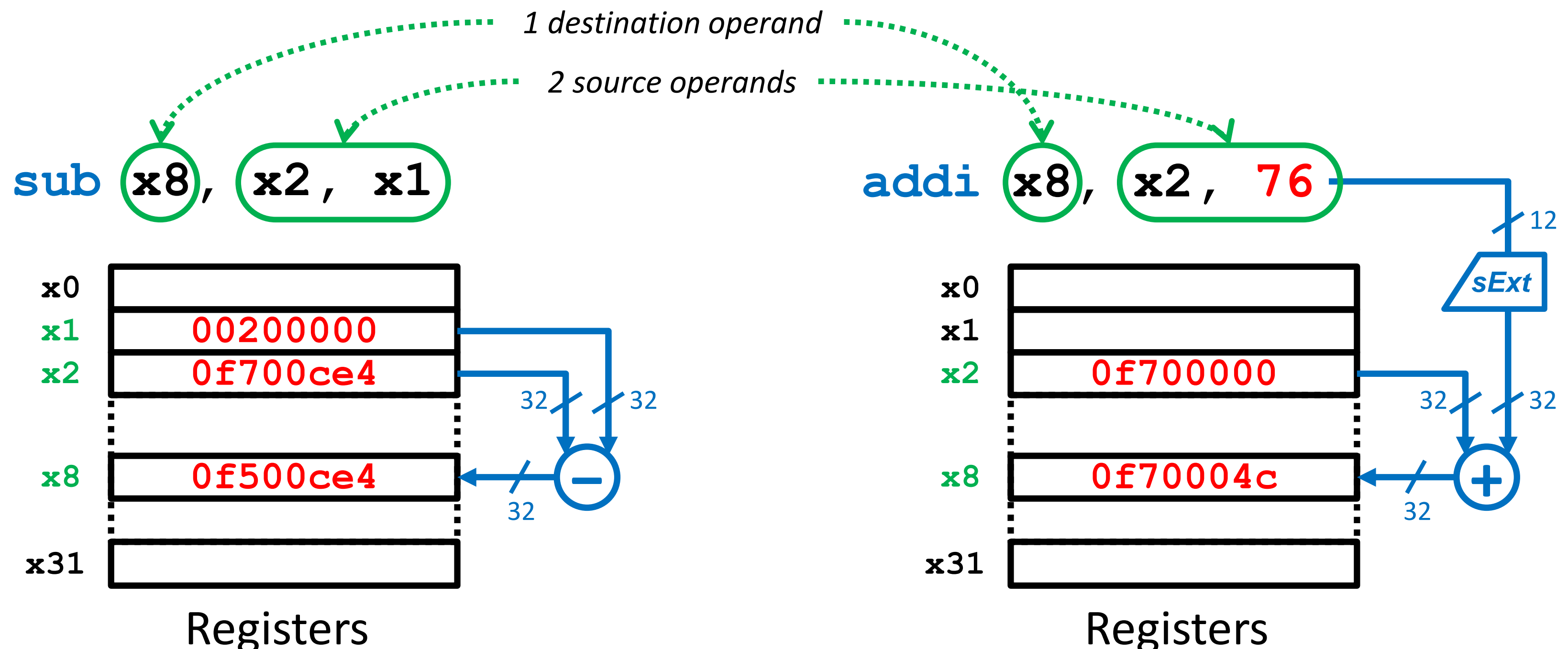
Concepts and types of instructions

- The instruction set is composed of all the instructions that can be executed by a processor.
  - All programs executed by a computer are sequences of instructions that belong to the same set.
- Instructions are classified in different types:
  - Data transfer: they copy data between the registers and the memory.
  - Arithmetic: they perform arithmetic operations.
  - Logical: they perform bitwise logical operations.
  - Shift: they perform bit shift operations.
  - Branch: they break the execution order by modifying the PC.
  - Privileged: they allow access to functionality that controls the system.
- The RISC-V instruction set:
  - Is very reduced, in order to avoid duplicities.
  - Instructions and addressing modes are strongly coupled.

Instruction set

Arithmetic (i)

- They perform arithmetic operations with 2 source operands and 1 destination operand, all of them with 32 bits.
  - The left operand is always in a register.
  - The right operand is either in a register or is a short immediate.
    - The immediate constant takes 12b in C2, in the [−2048, +2047] range, but its sign is extended to 32b before using it.
  - The result is always stored in a register.



Instruction set

Arithmetic (ii)

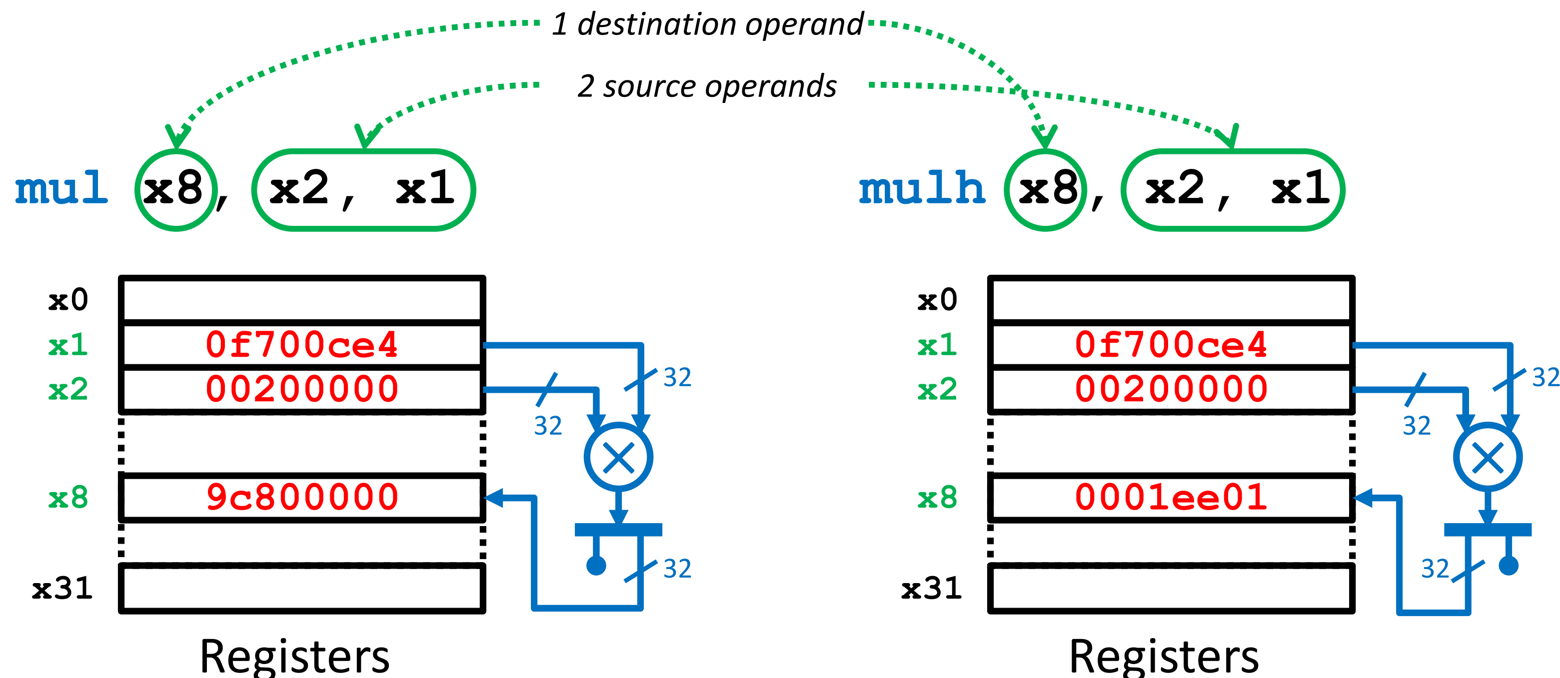
Instruction	Operation	Description
`add rd, rs1, rs2`	rd ← rs1 + rs2	**add**
`sub rd, rs1, rs2`	rd ← rs1 − rs2	**sub**tract
`slt rd, rs1, rs2`	rd ← *if* (rs1 $<_S$ rs2) *then* (1) *else* (0)	**s**et if **l**ess **t**han (signed)
`sltu rd, rs1, rs2`	rd ← *if* (rs1 $<_U$ rs2) *then* (1) *else* (0)	**s**et if **l**ess **t**han **u**nsigned
`addi rd, rs1,` imm_{12b}	rd ← rs1 + sExt(imm)	**add i**mmediate
`slti rd, rs1,` imm_{12b}	rd ← *if* (rs1 $<_S$ sExt(imm)) *then* (1) *else* (0)	**s**et if **l**ess **t**han **i**mmediate (signed)
`sltiu rd, rs1,` imm_{12b}	rd ← *if* (rs1 $<_U$ sExt(imm)) *then* (1) *else* (0)	**s**et if **l**ess **t**han **i**mmediate **u**nsigned

- There are different comparison instructions for signed and unsigned data.
- There is no subtraction with immediate operand, since this can be performed by adding the opposite.

Instruction set

Multiplication and division (i)

- The result of multiplying two 32-bit operands requires 64 bits.
  - There are two different types of multiplication instructions: one to calculate the upper part of the result and another one to calculate the lower part.
  - There are different instructions to obtain the upper part of the result depending on whether the source operands are signed or unsigned.
  - There is only one instruction to obtain the lower part of the result.
  - All the operands in these instructions are located in registers.



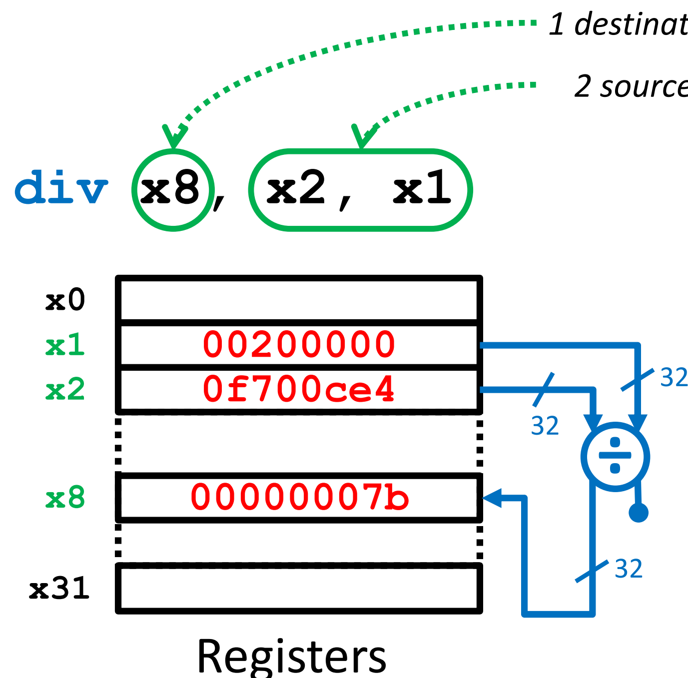
Instruction set

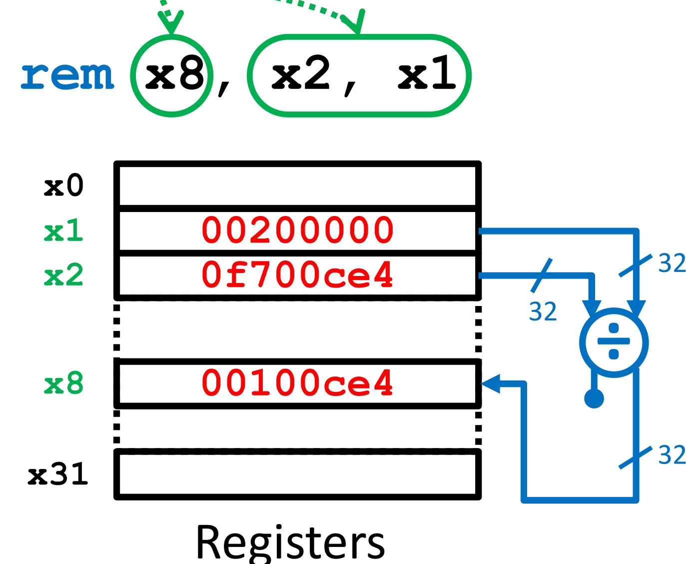
Multiplication and division (ii)

- The integer division of two 32 bits operands produces two results: the quotient and the remainder, both with 32 bits.
  - For that reason, there are two different instructions: one to obtain the quotient and another one to obtain the remainder.
  - Each one has variations to operate with signed and unsigned data.
  - All the operands in these instructions are located in registers.

1 destination operand

2 source operands





Instruction set

Multiplication and division (iii)

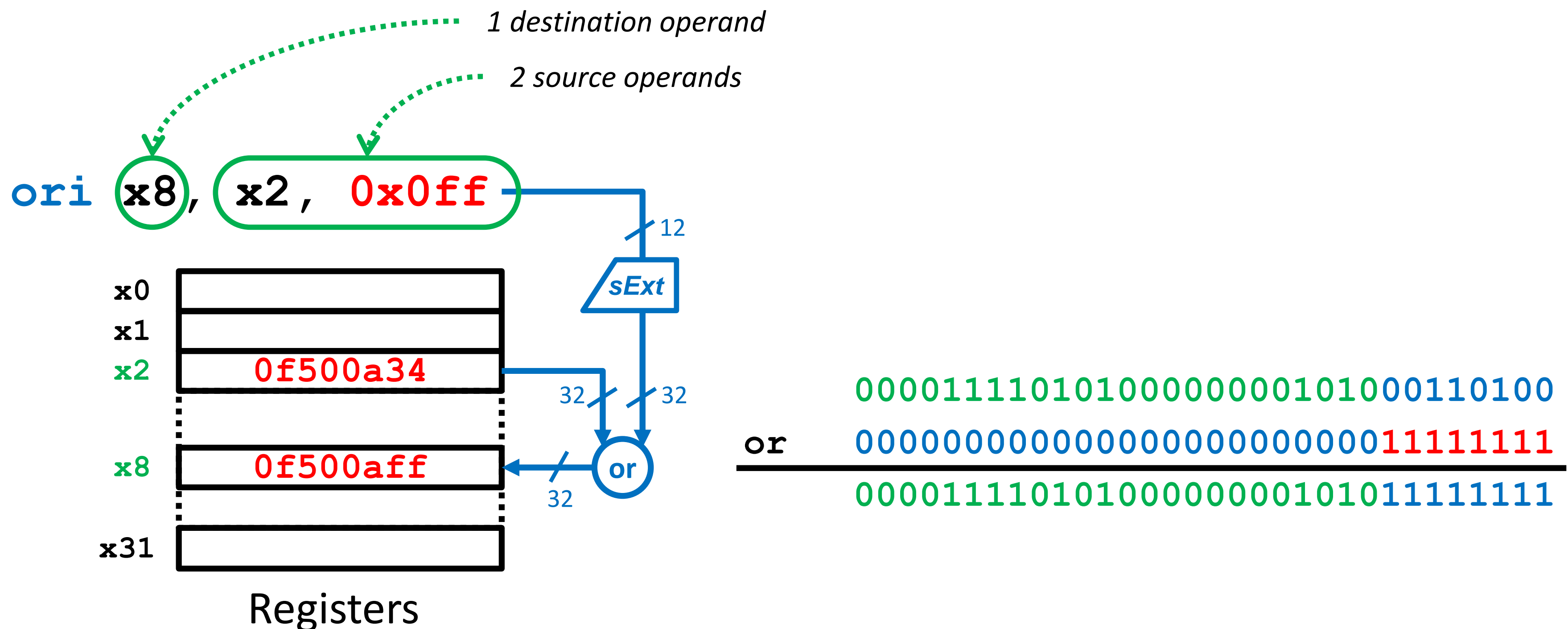
Instruction	Operation	Description
`mul rd, rs1, rs2`	rd ← $(rs1 * rs2)_{31:0}$	**mul**tiply integer multiplication (32 least significant bits)
`mulh rd, rs1, rs2`	rd ← $(rs1_S * rs2_S)_{63:32}$	**mul**tiply **h**igh signed integer multiplication (32 most significant bits)
`mulhsu rd, rs1, rs2`	rd ← $(rs1_S * rs2_U)_{63:32}$	**mul**tiply **h**igh **s**igned **u**nsigned hybrid integer multiplication (32 most significant bits)
`mulhu rd, rs1, rs2`	rd ← $(rs1_U * rs2_U)_{63:32}$	**mul**tiply **h**igh **u**nsigned unsigned integer multiplication (32 most significant bits)
`div rd, rs1, rs2`	rd ← $(rs1 /_S rs2)$	**div**ide signed integer division
`divu rd, rs1, rs2`	rd ← $(rs1 /_U rs2)$	**div**ide **u**nsigned unsigned integer division
`rem rd, rs1, rs2`	rd ← $(rs1 \%_S rs2)$	**rem**ainder signed integer remainder
`remu rd, rs1, rs2`	rd ← $(rs1 \%_U rs2)$	**rem**ainder **u**nsigned unsigned integer remainder

- These instructions are not in the RV32I set, but are part of the RVM extension.

Instruction set

Logical (i)

- They preform bitwise logical operations with 2 source operands and 1 destination operand, all of them with 32 bits.
  - The left operand is always in a register.
  - The right operand is either in a register or is a short immediate.
    - The immediate constant takes 12b in C2, but its sign is extended to 32b .
  - The result is always stored in a register.



Instruction set

Logical (ii)

- The bitwise logical operations, are used to manipulate individual bits within data.
  - One operand contains the data to manipulate.
  - Another operand contains a mask that indicates the bits to change.
  - Different operations are used depending on the required manipulation

```
     00001111010100000000101000110100   <............ data
or   00000000000000000000000011111111   <............ mask
     --------------------------------
     00001111010100000000101011111111   <............ manipulated data
```

*the **or** instruction sets (=1) those data bits whose corresponding mask bits are 1*

```
     00001111010100000000101000110100
and  00000000000000000000000011111111
     --------------------------------
     00000000000000000000000000110100
```

*the **and** instruction resets (=0) those data bits whose corresponding mask bits are 0*

```
     00001111010100000000101000110100
xor  00000000000000000000000011111111
     --------------------------------
     00001111010100000000101011001011
```

*the **xor** instruction toggles (1↔0) those data bits whose corresponding mask bits are 1*

Instruction set

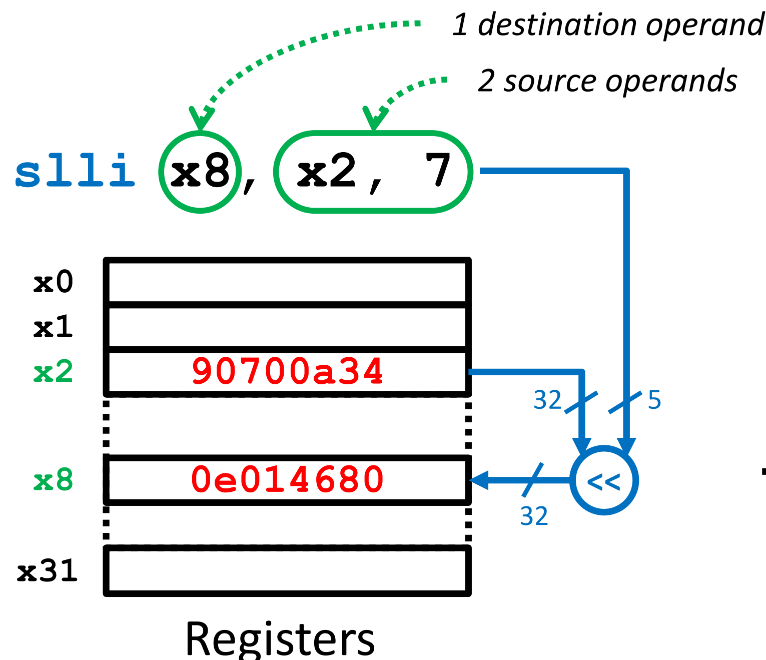
Logical (iii)

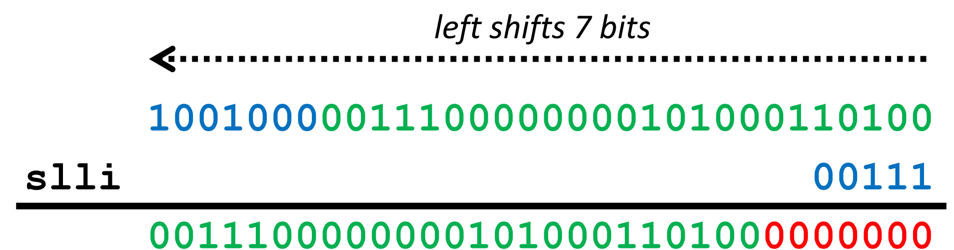
Instruction	Operation	Description
`and rd, rs1, rs2`	rd ← rs1 & rs2	**and**
`or rd, rs1, rs2`	rd ← rs1 | rs2	**or**
`xor rd, rs1, rs2`	rd ← rs1 ^ rs2	**xor**
`andi rd, rs1,` imm_{12b}	rd ← rs1 & sExt(imm)	**and i**mmediate
`ori rd, rs1,` imm_{12b}	rd ← rs1 | sExt(imm)	**or i**mmediate
`xori rd, rs1,` imm_{12b}	rd ← rs1 ^ sExt(imm)	**xor i**mmediate

Instruction set

Shift (i)

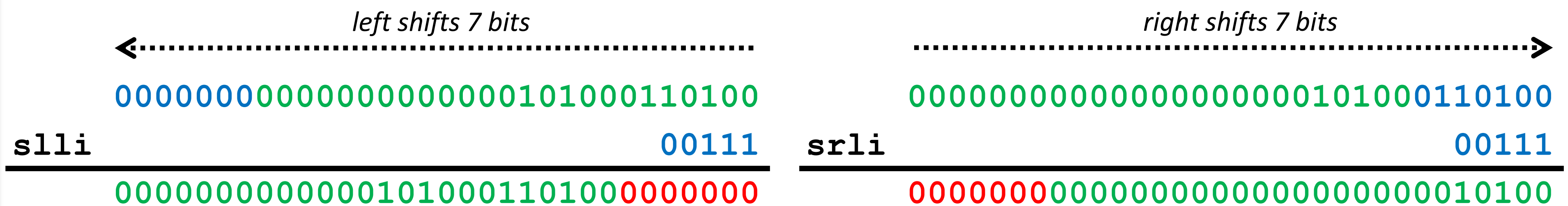
- They shift bits of a source operand a number of positions indicated by another one. Then it writes the result in another operand.
  - The left operand (32b) is always in a register.
  - The right operand (5b) is either in a register or is a short immediate.
    - The 5 least significant bits of the register are taken.
    - The immediate constant takes 5b is pure binary that are not extended.
  - The result is always stored in a register.



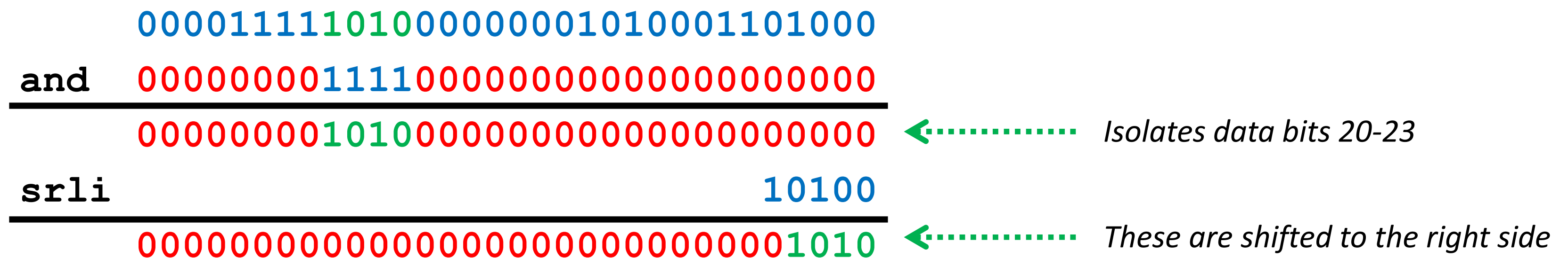


Instruction set
Shift (ii)

- Logical shift instructions insert 0s through one side of the data and discard the same number of bits at the other side.
  - This allows rescaling unsigned data:
    - Left shifting n bits is the same as multiplying by 2^n
    - Right shifting n bits is the same as dividing by 2^n



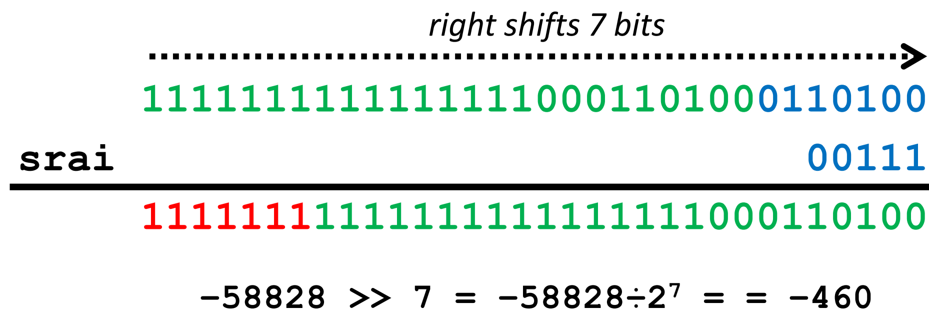
  - After a bitwise logic operation, this allows extracting fields from data:



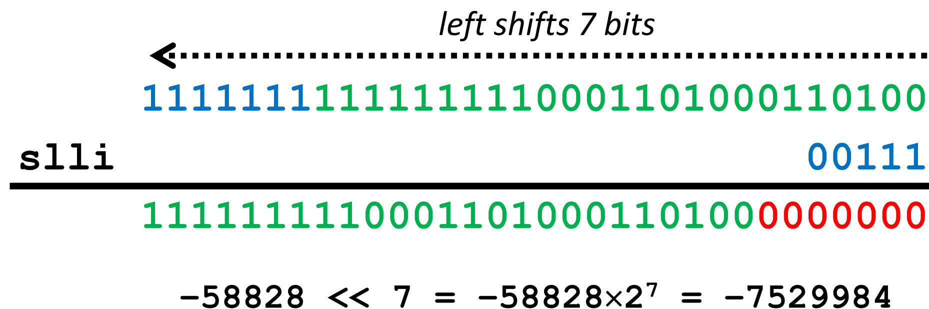
Instruction set

Shift (iii)

- Right arithmetic shift instructions propagate the sign bit on the left and discard the rightmost bits.
  - This allows rescaling signed data :



- There is no instruction for left arithmetic shift
  - For valid results (the rescaled signed data can be represented with 32b), the logical shift is equivalent.



Instruction set

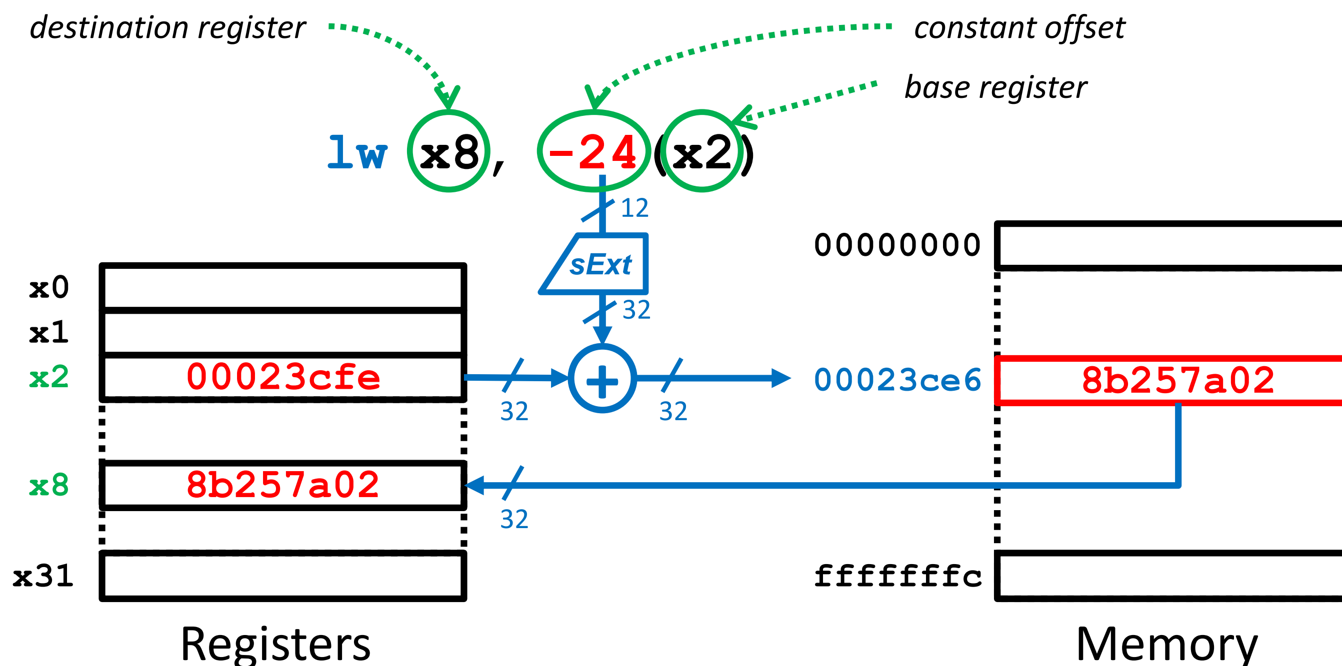
Shift (iv)

Instruction	Operation	Description
`sll rd, rs1, rs2`	rd ← rs1 << $rs2_{4:0}$	shift left logical
`srl rd, rs1, rs2`	rd ← rs1 >> $rs2_{4:0}$	shift right logical
`sra rd, rs1, rs2`	rd ← rs1 >>> $rs2_{4:0}$	shift right arithmetical
`slli rd, rs1,` imm_{5b}	rd ← rs1 << imm	shift left logical immediate
`srli rd, rs1,` imm_{5b}	rd ← rs1 >> imm	shift right logical immediate
`srai rd, rs1,` imm_{5b}	rd ← rs1 >>> imm	shift right arithmetical immediate

Instruction set

Data transfer: load

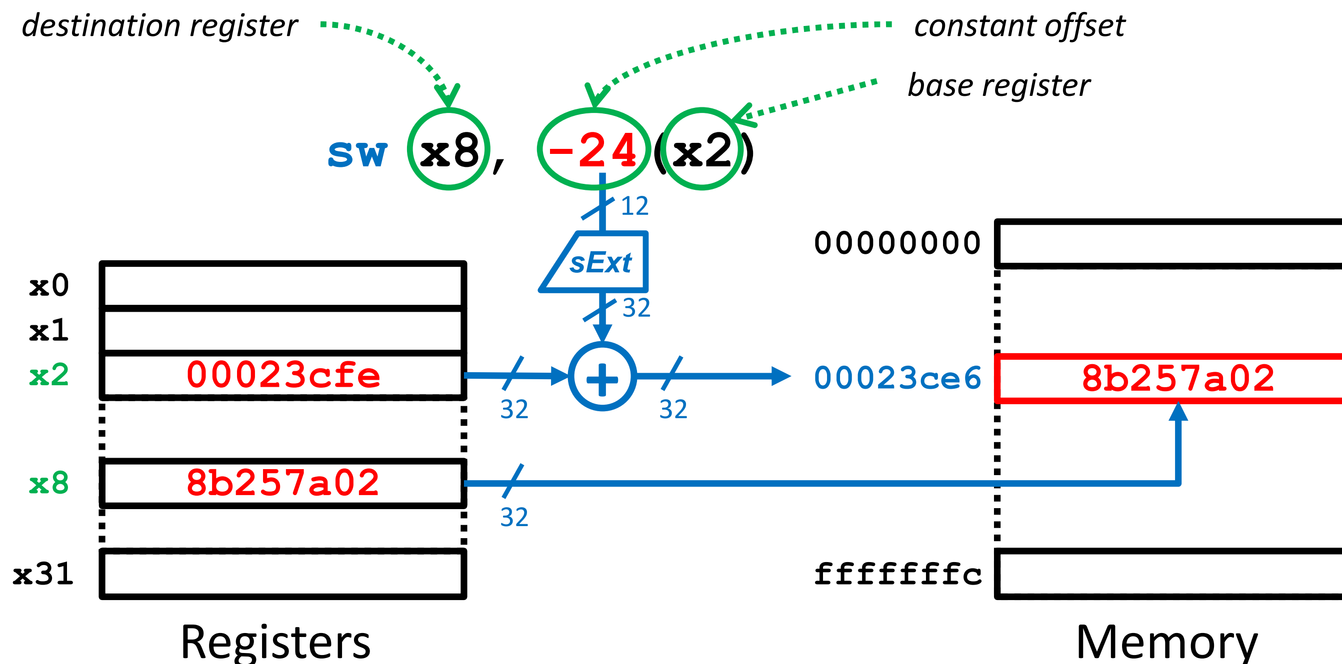
- They copy data from memory to a register.
  - It uses base addressing to indicate the memory address of the data
    - This address is the sum of a base address and an offset.
    - The base address is in a register.
    - The offset is a C2 12b immediate, whose sign is extended to 32b.
  - The data read from memory is loaded in a register.



Instruction set

Data transfer: store

- They copy data from a register to memory.
  - The data is in a register.
  - It uses base addressing to indicate the memory address to store the data
    - This address is the sum of a base address and an offset.
    - The base address is in a register.
    - The offset is a C2 12b immediate, whose sign is extended to 32b.



Instruction set

Data transfer (i)

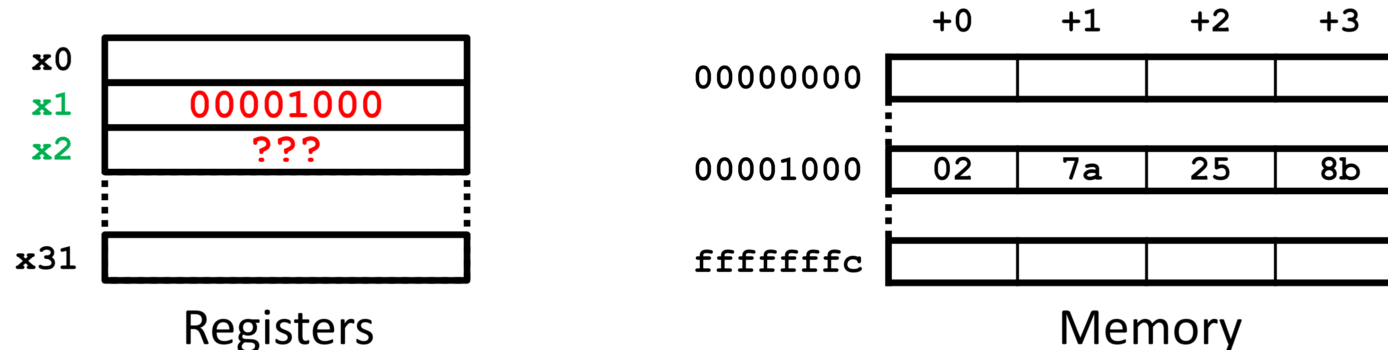
Instruction	Operation	Description
lw *rd*, imm_{12b}(*rs1*)	rd ← Mem[rs1 + sExt(imm)]	load word
lh *rd*, imm_{12b}(*rs1*)	rd ← sExt(Mem[rs1 + sExt(imm)]$_{15:0}$)	load half signed
lhu *rd*, imm_{12b}(*rs1*)	rd ← zExt(Mem[rs1 + sExt(imm)]$_{15:0}$)	load half unsigned unsigned
lb *rd*, imm_{12b}(*rs1*)	rd ← sExt(Mem[rs1 + sExt(imm)]$_{7:0}$)	load byte signed
lbu *rd*, imm_{12b}(*rs1*)	rd ← zExt(Mem[rs1 + sExt(imm)]$_{7:0}$)	load byte unsigned unsigned
sw *rs2*, imm_{12b}(*rs1*)	Mem[rs1 + sExt(imm)] ← rs2	store word
sh *rs2*, imm_{12b}(*rs1*)	Mem[rs1 + sExt(imm)]$_{15:0}$ ← rs2$_{15:0}$	store half
sb *rs2*, imm_{12b}(*rs1*)	Mem[rs1 + sExt(imm)]$_{7:0}$ ← rs2$_{7:0}$	store byte

- There are different instructions to copy 8b, 16b or 32b data.
- Also for sign extension (signed) or zero extension (unsigned).

Instruction set

Data transfer (ii)

- Data in memory is aligned, with Little-Endian organization

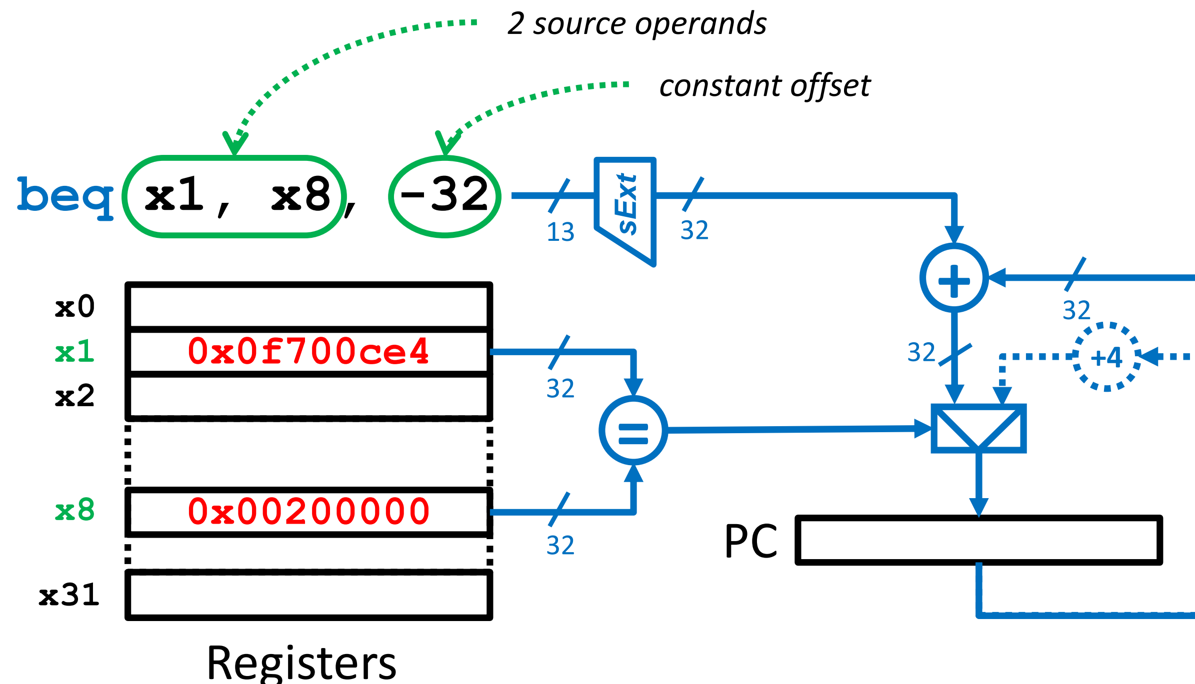


Instruction	Operation	Result
`lw x2, 0(x1)`	x2 ← Mem[x1 + sExt(0)]	load **8b257a02** in x2
`lhu x2, 0(x1)`	x2 ← zExt(Mem[x1 + sExt(0)]$_{15:0}$)	load **00007a02** in x2 (7a02 =$_2$ +31234)
`lhu x2, 2(x1)`	x2 ← zExt(Mem[x1 + sExt(2)]$_{15:0}$)	load **00008b25** in x2 (8b25 =$_2$ +35621)
`lh x2, 0(x1)`	x2 ← sExt(Mem[x1 + sExt(0)]$_{15:0}$)	load **00007a02** in x2 (7a02 =$_{C2}$ +31234)
`lh x2, 2(x1)`	x2 ← sExt(Mem[x1 + sExt(2)]$_{15:0}$)	load **ffff8b25** in x2 (8b25 =$_{C2}$ −29915)
`lbu x2, 3(x1)`	x2 ← zExt(Mem[x1 + sExt(3)]$_{7:0}$)	load **0000008b** in x2 (8b =$_2$ +139)
`lb x2, 3(x1)`	x2 ← sExt(Mem[x1 + sExt(3)]$_{7:0}$)	load **ffffff8b** in x2 (8b =$_{C2}$ −177)
`lh x2, 3(x1)`	x2 ← sExt(Mem[x1 + sExt(3)]$_{7:0}$)	***alignment error***

Instruction set

Conditional branch (i)

- They allow breaking the execution sequence branching to a nearby address when a certain condition is met.
  - It compares 2 two source operands located in registers.
  - It uses PC-relative addressing to indicate the new PC address in the case the condition is met.
    - This address is the sum of the PC content and a short offset.
    - The offset is a C2 13b constant, whose sign is extended to 32b



Instruction set

Conditional branch (ii)

- Conditional branch instructions are used to implement control structures e.g. *if*, *while*, *for*... in assembly.

C/C++

```
...
if( a != b )
  a = a + 1;
c = c - b;
...
```

ASM

```
        ...
dir     beq   x5, x6, 8
dir+4   addi  x5, x5, 1
dir+8   sub   x7, x7, x6
        ...
```

when **beq** *is executed, the PC contains its address; adding 8 to the PC, the program would branch to* **sub** *(only if the comparison is true)*

a → x5   b → x6   c → x7   *Assignment of the C variables to the RSIC-V registers*

- The immediate offset (that is added to the PC to perform the branch):
  - It is signed, and therefore it allows forward or backward branches.
  - Since it has 13 bits and each instruction takes 4B, it allows branching up to 1024 instructions backwards and 1023 forwards, from the branch instruction.
    - A C2 13b constant is in the [−4096, +4095] range.
  - Its 2 least significant bits are 0, because all the instructions are aligned.
    - In fact, the least significant bit is not stored in the instruction.

Instruction set

Conditional Branch (iii)

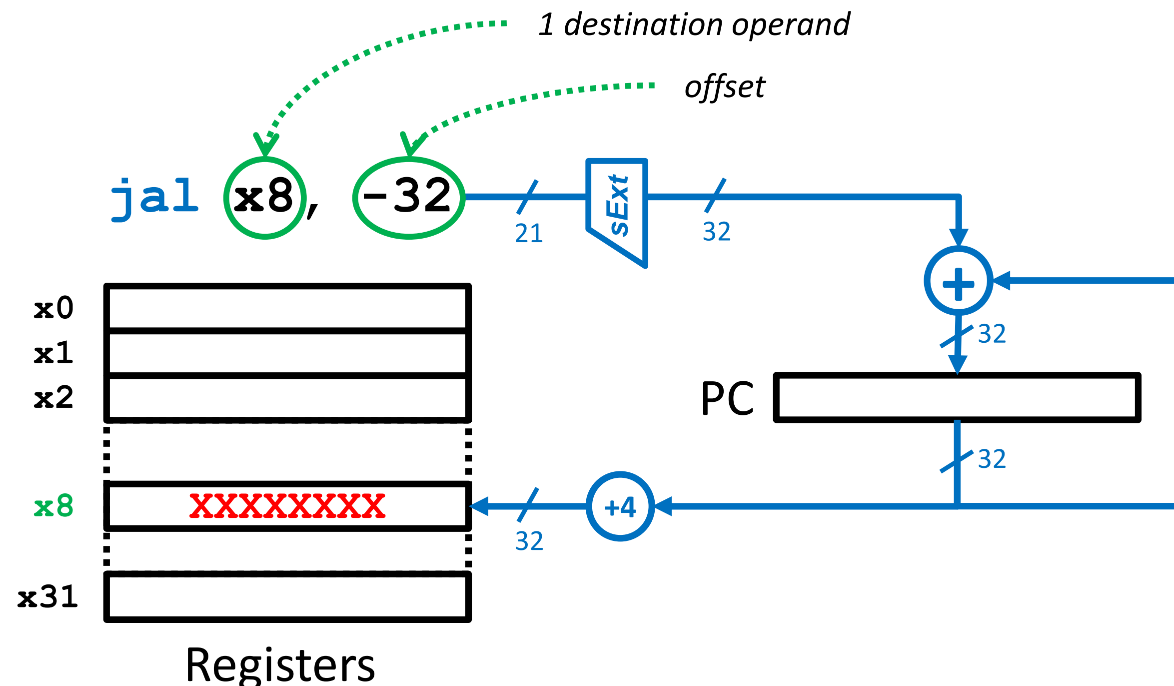
Instruction	Operation	Description
beq *rs1, rs2,* imm_{13b}	*if* (rs1 = rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	branch if equal
bne *rs1, rs2,* imm_{13b}	*if* (rs1 ≠ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	branch if not equal
blt *rs1, rs2,* imm_{13b}	*if* (rs1 $<_S$ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	branch if less than signed
bge *rs1, rs2,* imm_{13b}	*if* (rs1 $\geq_S$ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	branch if greater than or equal signed
bltu *rs1, rs2,* imm_{13b}	*if* (rs1 $<_U$ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	branch if less than unsigned unsigned
bgeu *rs1, rs2,* imm_{13b}	*if* (rs1 $\geq_U$ rs2) *then* (PC ← PC + sExt($imm_{12:1}$ << 1))	branch if greater than or equal unsigned unsigned

- There are different instructions to compare signed and unsigned data.
- There are no branch instructions with an immediate operand.
- There are no “greater than” or “less than or equal”, because these can be implemented using the other instructions and changing the order of the operands.

Instruction set

Branch to function: `jal`

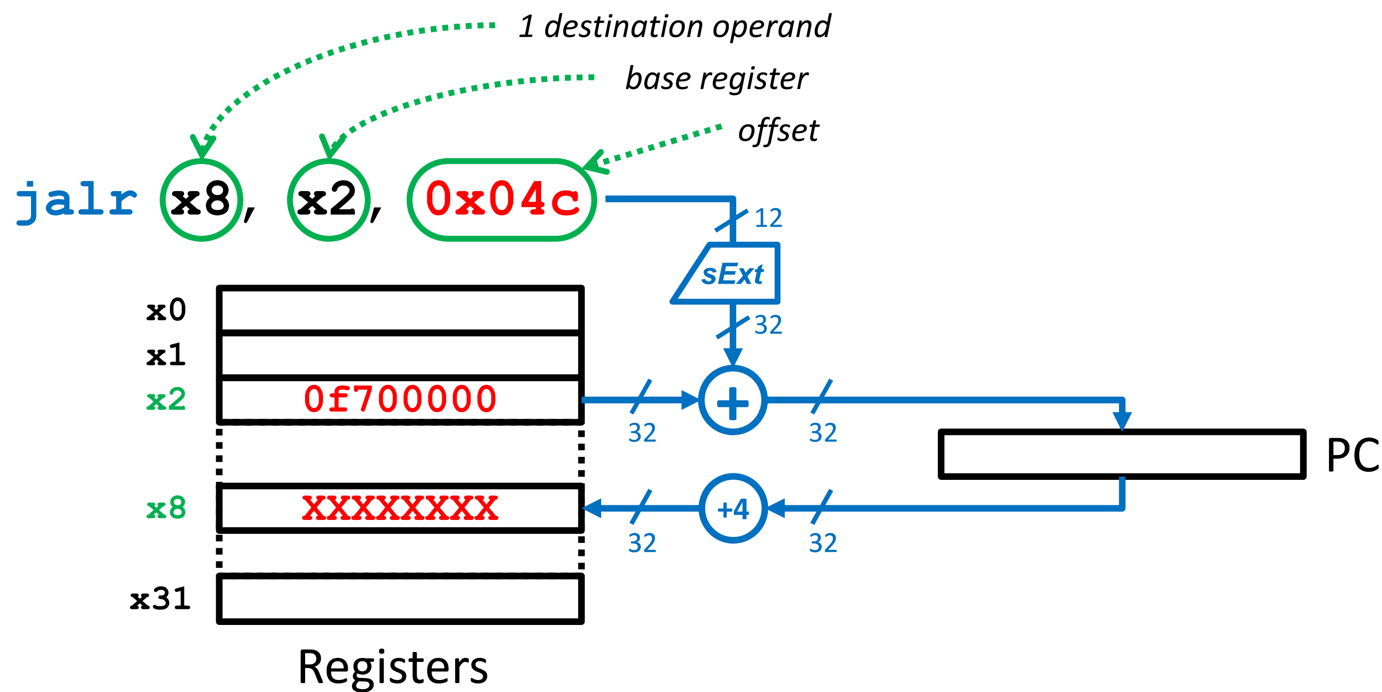
- They allow breaking the execution sequence by branching to a faraway address, but saving the return address.
  - It uses PC-relative addressing to indicate the new PC address.
    - This address is the sum of the PC content and a long offset.
    - The offset is a C2 21b constant, whose sign is extended to 32b.
  - The next instruction address (return) is saved in a register.



Instruction set

Branch to function: `jalr`

- They allow breaking the execution sequence by branching to a faraway address, but saving the return address.
  - It uses base addressing to indicate the new PC address.
    - This address is the sum of the PC content and a short offset.
    - The offset is a C2 12b constant, whose sign is extended to 32b.
  - The next instruction address (return) is saved in a register.



Instruction set

Branch to function (i)

- Branch to function instructions are used to implement function calls in assembly.
  - Each of the functions in a program is located in a different memory address.
  - To call a function means branching to the address of its first instruction.
  - Returning from a function means branching to the address of the instruction following the one that made the function call.

C language

```
...
a = b - c;
foo();
c = c + 1;
...
```

RISC-V assembly

```
       ...
dir-4  sub   x5, x6, x7
dir    jal   x1, 1000
dir+4  addi  x7, x7, 1
       ...
```

*when **jal** is executed, the PC contains its address; PC+4 is saved in **x1** and 1000 is added to the PC in order to branch to the function.*

a → x5   b → x6   c → x7   *Assignment of the C variables to the RSIC-V registers*

Instruction set

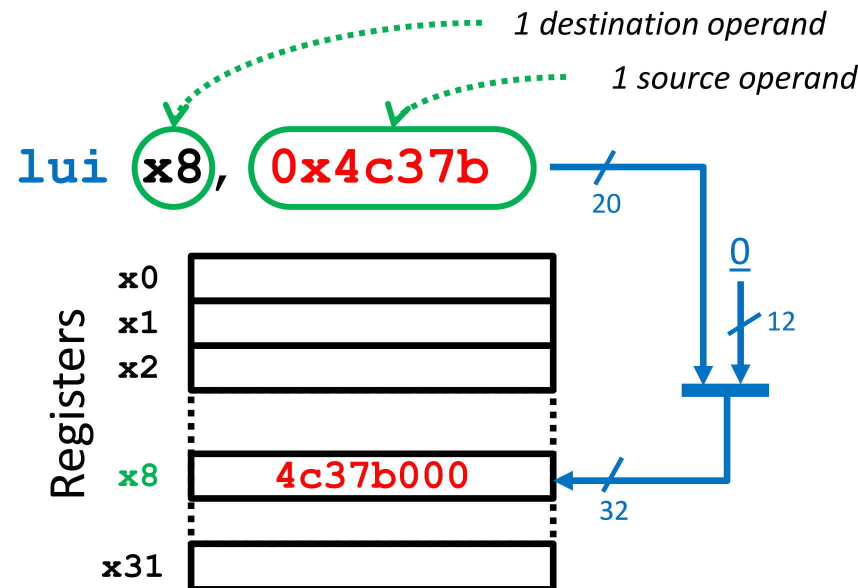
Branch to function (ii)

Instruction	Operation	Description
`jalr rd, rs1, imm`$_{12b}$	PC ← rs1 + sExt(imm), rd ← PC+4	jump and link register branch to function with base addressing
`jal rd, imm`$_{21b}$	PC ← PC + sExt(imm$_{20:1}$ << 1), rd ← PC+4	jump and link branch to function with PC-relative addressing

- In the RISC-V instruction set there are no return or unconditional branch instructions, but these can be implemented using `jal` and `jalr`:
  - Assuming that the return address is stored in register ***xn***, the return can be performed as: `jalr x0, xn, 0`
  - An unconditional branch (PC-relative) to the address of a certain instruction can be performed as: `jal x0, imm`$_{21b}$

Instruction set
lui instruction (i)

- It loads a constant in the upper part of a register.
  - The source operand is a 20b immediate constant.
    - It is zero-extended to 32b, adding 12 0s on the right.
  - The result is a stored in a register.



Instruction	Operation	Description
lui *rd*, imm_{20b}	rd ← imm << 12	load upper immediate

Instruction set

lui instruction (ii)

- The lui instruction is used to operate with 32b long constants.
  - Immediate operands in arithmetic-logic instructions are short (12b).

C:

```
...
a = b + 0xabcde123;
...
```

a → x5 b → x6

Assignment of the C variables to registers

ASM:

```
...
lui  x7, 0xabcde        <-- x7 = abcde000
addi x7, x7, 0x123      <-- x7 = abcde000
                               + 00000123
                                 abcde123
add  x5, x6, x7
...
```

- Since the **addi** instruction extends the sign of the immediate value, if bit 11 of the long constant is 1, then the lui constant has to be increased by 1

C:

```
...
a = b + 0xabcde987;
...
```

a → x5 b → x6

Assignment of the C variables to registers

ASM:

```
...
lui  x7, 0xabcdf        <-- x7 = abcdf000
addi x7, x7, 0x987      <-- x7 = abcdf000
                               + ffff987
                                 abcde987
add  x5, x6, x7
...
```

Instruction set

lui instruction (iii)

- Although it is not very common, the `lui` instruction can also be used to work with 32b absolute memory addresses.
  - To transfer data located in any absolute memory address.

```
ASM
...
lui  x6,  0x76543
lw   x5,  0x210(x6)
...
```

loads the data located in address 0x76543210 into x5

```
ASM
...
lui  x6,  0x76543
sw   x5,  0x210(x6)
...
```

stores the data located in x5 into address 0x76543210

  - To branch to functions located in any absolute memory address.

```
ASM
...
lui    x6,  0x76543
jalr   x1,  x6,  0x210
...
```

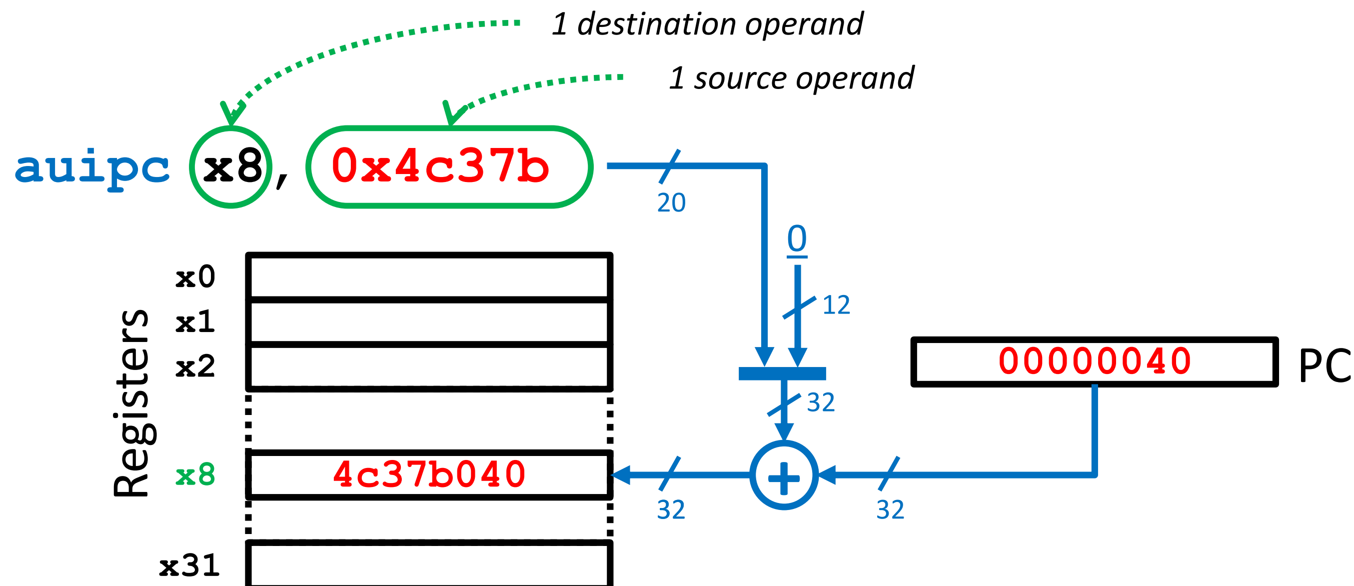
branches to the instruction located in address 0x76543210

  - Since the **`lw`**, **`sw`** and **`jalr`** instructions also extend the sign of the immediate offset, when bit 11 of the absolute address is 1, the `lui` constant has to be increased by 1.

Instruction set

auipc instruction (i)

- It adds a constant to the upper part of the PC.
  - The source operand is a 20b immediate constant.
    - It is zero-extended to 32b, adding 12 0s on the right.
  - The result is a stored in a register.

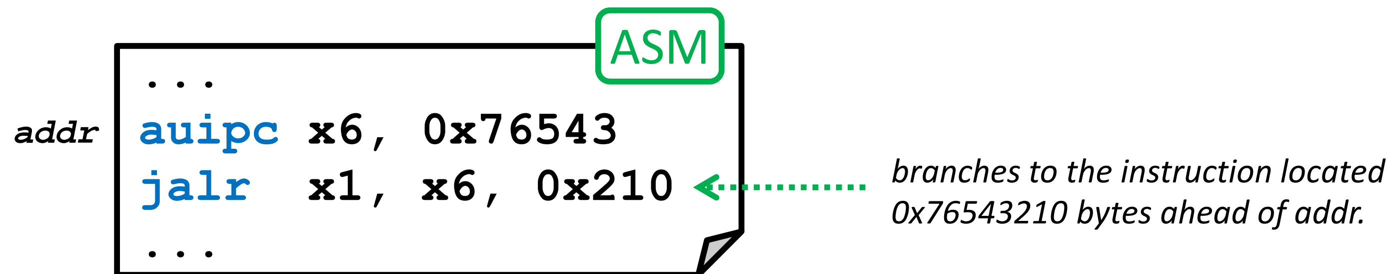


Instruction	Operands	Descriptions
auipc *rd*, *imm*$_{20b}$	rd ← PC + (imm << 12)	add upper immediate to PC

Instruction set

auipc instruction (ii)

- The jal instruction, even using a long offset, only allows branching to addresses in the ±1MiB range, which sometimes is not enough.
  - Since the offset has 21b and each instruction takes 4B, it only allows branching up to 262,144 instructions backwards and 262,143 forwards.
- The auipc instruction, together with jalr, are used to perform PC-relative branches to functions located in any memory address.
  - This covers the full ±4GiB address space.

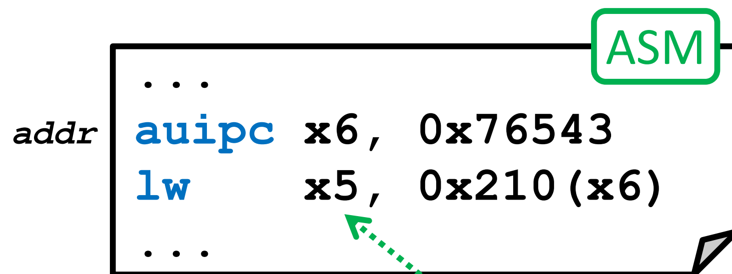


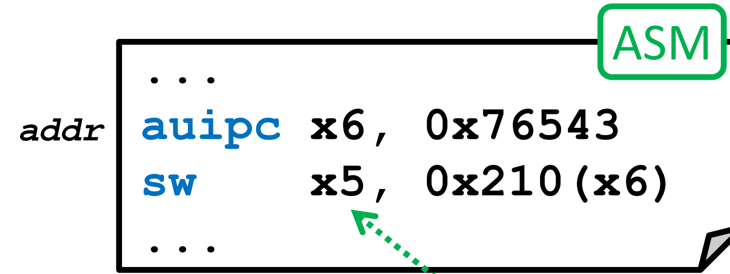
  - Since the jalr instruction extends the sign of the immediate value, when bit 11 of the 32b offset is 1, the auipc constant has to be increased by 1.

Instruction set

auipc instruction (iii)

- The load and store instructions use base addressing with a short offset
  - This covers a ±2KiB range respect the base address .
- Combining these instructions with **auipc**, any memory address (PC-relative) can be accessed.





  - As in previous cases, when bit 11 of the PC-relative 32b offset is 1, the **auipc** constant has to be increased by 1.

Instruction set

Most popular instructions

- There are instructions more frequently used than others
  - This can be measured counting how many times each instruction is executed in a set of standard programs (SPEC CPU2006)

Instruction	Description	Frequency	Accumulated
`lw`	*load*	19.48%	19.48%
`addi`	*add immediate*	17.22%	36.70%
`sw`	*store*	8.05%	44.75%
`add`	*add*	7.57%	52.32%
`bne`	*branch if not equal*	4.14%	56.46%
`slli`	*shift left logical immediate*	3.65%	60.11%
`beq`	*branch if equal*	3.27%	63.38%
`mul`	*multiply*	2.02%	65.40%

source (adapted): D.A. Patterson and J.L. Hennessy. Computer Organization and Design. RISC-V Edition (2021)

RISC-V: ISA and extensions

- RISC-V is an open and flexible architecture.
  - It defines instruction sets architectures (ISA) and extensions.
  - All RISC-V processor must support one of the ISA and optionally some of the extensions.
- ISA:
  - RV32I: 32-bit instructions and data/addresses.
  - RV32E: RV32I version with only 16 registers.
  - RV64I: 32-bit instructions and 64-bit data/addresses.
  - RV128I: 32-bit instructions and 128-bit data/addresses.
- Extensions:
  - RVM: includes integer multiplication, division and remainder.
  - RVF: includes 32 floating point data registers, as well as floating point arithmetic, relational and conversion operations.
  - RVD: 64-bit floating point data version (double precision).
  - RVQ: 128-bit floating point data version (quadruple precision).
  - RVC: extension with 16-bit compressed instructions.

RISC-V: ISA and extensions

Example: RV64I ISA (i)

- RV64I ISA.
  - 64-bit integer data and 32-bit instructions.
  - It has 32 registers, with 64 bits.
  - Includes 64b memory transfer instructions.
  - Redefines the 32b transfer instructions.
  - Includes a new instruction for 32b unsigned data load.

Instruction	Operation	Description
`ld rd, imm(rs1)`	rd ← Mem[rs1 + sExt(imm_{12b})]	load double word
`sd rs2, imm(rs1)`	Mem[rs1 + sExt(imm_{12b})] ← rs2	store double word

Instruction	Operation	Description
`lw rd, imm(rs1)`	rd ← sExt(Mem[rs1 + sExt(imm_{12b})]$_{31:0}$)	load signed word
`lwu rd, imm(rs1)`	rd ← zExt(Mem[rs1 + sExt(imm_{12b})]$_{32:0}$)	load unsigned word
`sw rs2, imm(rs1)`	Mem[rs1 + sExt(imm_{12b})] ← $rs_{31:0}$	store word

RISC-V: ISA and extensions

Example: RV64I ISA (ii)

- RV64I ISA.
  - Arithmetic-logic instructions work with 64b data (immediate operands still have 12b but extended to 64b)
  - Redefines the shift instructions to work with 64b data (they require 6b to indicate the number of bits to shift)

Instruction	Operation	Description
`sll` *rd, rs1, rs2*	rd ← rs1 << $rs2_{5:0}$	shift left logical
`srl` *rd, rs1, rs2*	rd ← rs1 >> $rs2_{5:0}$	shift right logical
`sra` *rd, rs1, rs2*	rd ← rs1 >>> $rs2_{5:0}$	shift right arithmetic
`slli` *rd, rs1, imm*	rd ← rs1 << imm_{6b}	shift left logical immediate
`srli` *rd, rs1, imm*	rd ← rs1 >> imm_{6b}	shift right logical immediate
`srai` *rd, rs1, imm*	rd ← rs1 >>> imm_{6b}	shift right arithmetic immediate

RISC-V: ISA and extensions

Example: RV64I ISA (iii)

- RV64I ISA.
  - Includes arithmetic-logic and shift instructions to work with 32b data (w suffix).

Instruction	Operation	Description
`addw rd, rs1, rs2`	rd ← sExt($rs1_{31:0}$ + $rs2_{31:0}$)	add word
`subw rd, rs1, rs2`	rd ← sExt($rs1_{31:0}$ − $rs2_{31:0}$)	subtract word
`addiw rd, rs1, imm`	rd ← sExt($rs1_{31:0}$ + sExt(imm_{12b})$_{31:0}$)	add immediate word

Instruction	Operation	Description
`sllw rd, rs1, rs2`	rd ← sExt($rs1_{31:0}$ << $rs2_{4:0}$)	shift left logical word
`srlw rd, rs1, rs2`	rd ← sExt($rs1_{31:0}$ >> $rs2_{4:0}$)	shift right logical word
`sraw rd, rs1, rs2`	rd ← sExt($rs1_{31:0}$ >>> $rs2_{4:0}$)	shift right arithmetic word
`slliw rd, rs1, imm`	rd ← sExt($rs1_{31:0}$ << imm_{5b})	shift left logical immediate word
`srliw rd, rs1, imm`	rd ← sExt($rs1_{31:0}$ >> imm_{5b})	shift right logical immediate word
`sraiw rd, rs1, imm`	rd ← sExt($rs1_{31:0}$ >>> imm_{5b})	shift right arithmetic immediate word

RISC vs. CISC architectures

- RISC-V is a clear example of a RISC architecture.
  - Other RISC architectures are: PowerPC, DEC Alpha, MIPS, ARM, SPARC...
  - It is the predominant architecture in mobile devices.
  - 75% of the processors have an ARM architecture.

- But there are also other architectures with a different paradigm, called CISC architectures (*Complex Instruction Set Computer*).
  - They have a large set of complex instructions.
  - Instructions can work both with data stored in memory as well as data in registers
  - They have a reduced number of registers, some of them with a specific purpose.
  - They have a large number of addressing modes.
  - Instructions have a variable size and many different formats.
  - Example of CISC architectures are: Motorola 68K, Intel x86, AMD x86-64...
  - It is the predominant architecture in personal computers.

RISC vs. CISC architectures

x86 architecture (i)

- The x86 architecture is the main example of a CISC architecture
  - Introduced by Intel in 1978 in the 8086 y 8088 microprocessors, it has evolved through several generations.
  - Used by personal computers since the launch of the IBM-PC in 1981.

Feature	RISC-V (RV32I)	x86
Number of registers	32 general purpose	8, with some use restrictions
Number of operands	3 (2 source, 1 destination)	2 (1 source, 1 source/destination)
Operand location	Registers or immediate	Registers, immediate or memory
Operand size	32 bits	8, 16 or 32 bits
Condition flags	No	Yes
Number of instructions	Reduced	Large
Type of instructions	Simple	Simple and complex
Instruction encoding	Fixed: 4 bytes/instruction	Variable: from 1 to 15 bytes/instruction

RISC vs. CISC architectures

x86 architecture (ii)

Instruction	Operation	Adds with...
`add AH, BL`	AH ← AH + BL	8b registers
`add AX, -1`	AH ← AH + 0xffff	16b immediate
`add EAX, EBX`	EAX ← EAX + EBX	32b register
`add EAX, 42`	EAX ← EAX + 0x0000002a	32b immediate
`add EAX, [20]`	EAX ← EAX + Mem[20]	absolute addressing
`add EAX, [ESP]`	EAX ← EAX + Mem[ESP]	base addressing
`add EAX, [EDX+40]`	EAX ← EAX + Mem[EDX+40]	base addressing with offset
`add EAX, [60+EDI*4]`	EAX ← EAX + Mem[60+EDI*4]	scaled index register with offset
`add EAX, [EDX+80+EDI*4]`	EAX ← EAX + Mem[EDX+80+EDI*4]	base register, offset and scaled index register
`add [20], EAX`	Mem[20] ← Mem[20] + EAX	32b register stored in memory
`add [20], 42`	Mem[20] ← Mem[20] + 42	immediate stored in memory

About *Creative Commons*

- CC license (Creative Commons)

  - This license enables reusers to distribute, remix, adapt, and build upon the material in any medium or format for noncommercial purposes only, and only so long as attribution is given to the creator. If you remix, adapt, or build upon the material, you must license the modified material under identical terms:

Attribution:
Credit must be given to the creator.

Non commercial:
Only noncommercial uses of the work are permitted.

Share alike:
Adaptations must be shared under the same terms.

More information: https://creativecommons.org/licenses/by-nc-sa/4.0/