Compilers and computer architecture

12. The RISC-V architecture

Hsi-Ming Ho

(slides by Niel de Beaudrap)

Email: **hsi-ming.ho@sussex.ac.uk**
Office hours: Online, by email appointment

Autumn 2022

Introduction

Most of the lectures in this module describe computers in broad generalities. In this lecture, we focus on the features of particular processors.

Introduction

Most of the lectures in this module describe computers in broad generalities. In this lecture, we focus on the features of particular processors.

Processors can roughly be classified as

- **CISC** (complex instruction set computer) or
- **RISC** (reduced instruction set computer).

Introduction

Most of the lectures in this module describe computers in broad generalities. In this lecture, we focus on the features of particular processors.

Processors can roughly be classified as

- **CISC** (complex instruction set computer) or
- **RISC** (reduced instruction set computer).

In this lecture, we consider RISC processors, and in particular, the RISC-V line of processors.

Instruction set architecture

The **instruction set architecture** are the features of the CPU which are accessible to the programmer:

Instruction set architecture

The **instruction set architecture** are the features of the CPU which are accessible to the programmer:

- Supported data types (e.g. 32-bit ints, 80-bit floats, *etc.*);

Instruction set architecture

The **instruction set architecture** are the features of the CPU which are accessible to the programmer:

- Supported data types (e.g. 32-bit ints, 80-bit floats, *etc.*);
- Instructions that can be performed on these data types;

Instruction set architecture

The **instruction set architecture** are the features of the CPU which are accessible to the programmer:

- Supported data types (e.g. 32-bit ints, 80-bit floats, *etc.*);
- Instructions that can be performed on these data types;
- Number and kinds of registers;

Instruction set architecture

The **instruction set architecture** are the features of the CPU which are accessible to the programmer:

- Supported data types (e.g. 32-bit ints, 80-bit floats, *etc.*);
- Instructions that can be performed on these data types;
- Number and kinds of registers;
- Addressing modes, memory architecture, interrupts, *etc.*

Instruction set architecture

The **instruction set architecture** are the features of the CPU which are accessible to the programmer:

- Supported data types (e.g. 32-bit ints, 80-bit floats, *etc.*);
- Instructions that can be performed on these data types;
- Number and kinds of registers;
- Addressing modes, memory architecture, interrupts, *etc.*

The underlying **microarchitecture** (how the instruction set is implemented) is usually not visible to the programmer.

Instruction set architecture

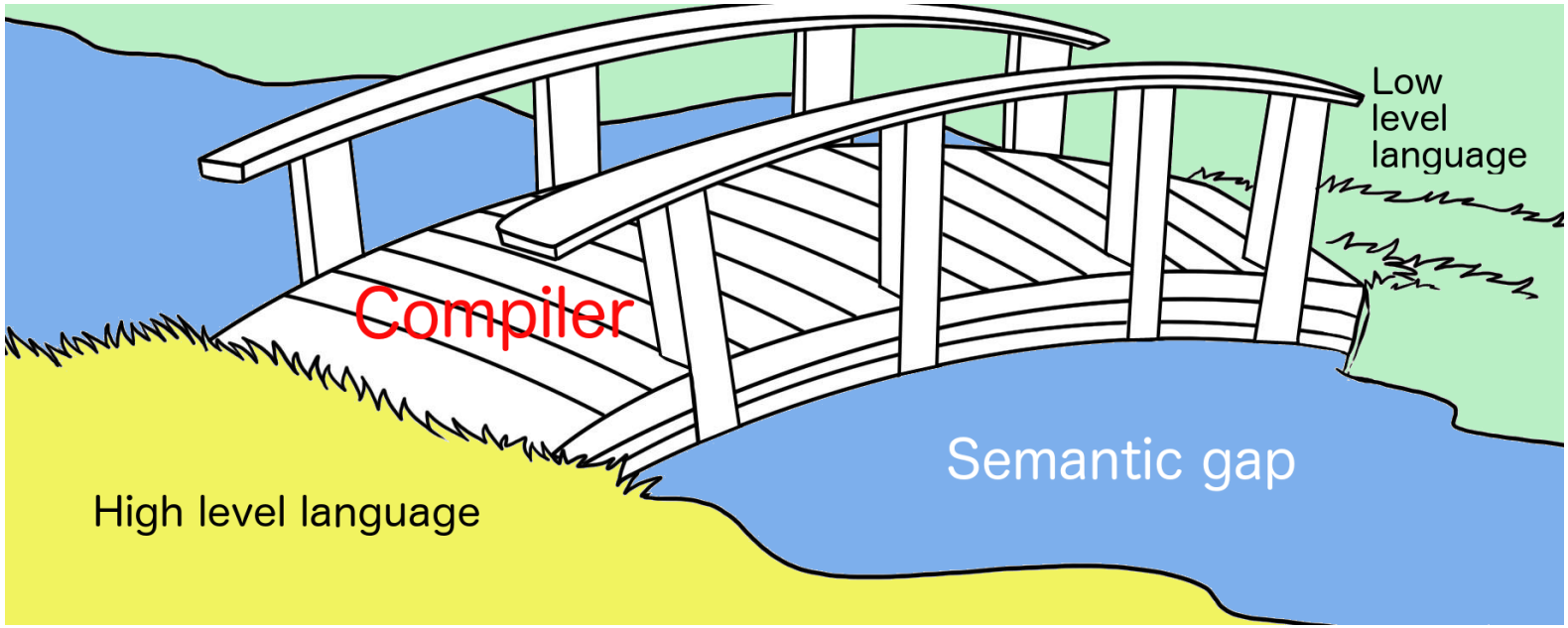
The **instruction set architecture** are the features of the CPU which are accessible to the programmer:

- Supported data types (e.g. 32-bit ints, 80-bit floats, *etc.*);
- Instructions that can be performed on these data types;
- Number and kinds of registers;
- Addressing modes, memory architecture, interrupts, *etc.*

The underlying **microarchitecture** (how the instruction set is implemented) is usually not visible to the programmer.

- CPUs with different microarchitectures can share a common instruction set.
  - Intel and AMD support very similar instruction sets (x86 derived), but have very different microarchitectures.

Recall:



There is a **semantic gap** between high-level programming languages and machine languages: the former have powerful features (e.g. method invocation) that translate to a large number of simple machine instructions.

Instruction set architecture

In the past, it was thought that making machine instructions more powerful would close or narrow the semantic gap, making compiled code faster:

Instruction set architecture

In the past, it was thought that making machine instructions more powerful would close or narrow the semantic gap, making compiled code faster:

- directly enabling constructs such as procedure calls
- single instructions for complicated array access

Instruction set architecture

In the past, it was thought that making machine instructions more powerful would close or narrow the semantic gap, making compiled code faster:

- directly enabling constructs such as procedure calls
- single instructions for complicated array access

CPUs with such instruction sets are called **CISC** (complex instruction set computer), as the instructions do complicated things and are complex to implement in hardware.

Instruction set architecture

In some cases CISC architectures did lead to faster compiled code. But in the 1970s, researchers began to study instruction set architecture and compiled code carefully and noticed few important things.

Instruction set architecture

In some cases CISC architectures did lead to faster compiled code. But in the 1970s, researchers began to study instruction set architecture and compiled code carefully and noticed few important things.

- Compilers rarely make (good) use of the complex instructions provided by CISC machines.

Instruction set architecture

In some cases CISC architectures did lead to faster compiled code. But in the 1970s, researchers began to study instruction set architecture and compiled code carefully and noticed few important things.

- Compilers rarely make (good) use of the complex instructions provided by CISC machines.
- Complex operations tended to be slower than a sequence of simpler operations doing the same thing.

Instruction set architecture

In some cases CISC architectures did lead to faster compiled code. But in the 1970s, researchers began to study instruction set architecture and compiled code carefully and noticed few important things.

- Compilers rarely make (good) use of the complex instructions provided by CISC machines.
- Complex operations tended to be slower than a sequence of simpler operations doing the same thing.
- Often real-world programs spend most of their time executing simple operations.

Instruction set architecture

In some cases CISC architectures did lead to faster compiled code. But in the 1970s, researchers began to study instruction set architecture and compiled code carefully and noticed few important things.

- Compilers rarely make (good) use of the complex instructions provided by CISC machines.
- Complex operations tended to be slower than a sequence of simpler operations doing the same thing.
- Often real-world programs spend most of their time executing simple operations.
- To implement complex operations requires complicated CPU architecture (circuitry etc.) — which causes even simpler operations to be slower than they have to be!

RISC

These empirical insights lead to a radical rethink of instruction set architecture, with the aim of aggressively optimising the few instructions that are most used in practice.

RISC

These empirical insights lead to a radical rethink of instruction set architecture, with the aim of aggressively optimising the few instructions that are most used in practice.

- Unleashes higher performance!
- Makes the work of the compiler (much) harder;
- Note that the compiler has to compile a program only once to obtain the benefit of the more efficient processors.

RISC

These empirical insights lead to a radical rethink of instruction set architecture, with the aim of aggressively optimising the few instructions that are most used in practice.

- Unleashes higher performance!
- Makes the work of the compiler (much) harder;
- Note that the compiler has to compile a program only once to obtain the benefit of the more efficient processors.

RISC processors like RISC-V are the outcome.

RISC

A key feature of RISC:

- data stored in memory is only accessible to load or store instructions;
- all other instructions are limited to registers

Hence RISC is sometimes called **load/store architecture**.

RISC

A key feature of RISC:

- data stored in memory is only accessible to load or store instructions;
- all other instructions are limited to registers

Hence RISC is sometimes called **load/store architecture**.

This drastically simplifies processor design: allowing instructions to be fixed-length, simplifying pipelines, and isolating the logic for dealing with the delay in completing a memory access (cache miss, etc.) to only two instructions.

RISC vs CISC today

Despite RISC being technically better, still the most popular desktop/server family of chips (Intel x86) is not RISC:

RISC vs CISC today

Despite RISC being technically better, still the most popular desktop/server family of chips (Intel x86) is not RISC:

- Intel x86 architecture has been backwards compatible back to the 8080 processor (1974) — supporting a large amount of legacy x86 code (*e.g.*, Microsoft products), locking PC users into x86 CISC.

RISC vs CISC today

Despite RISC being technically better, still the most popular desktop/server family of chips (Intel x86) is not RISC:

- Intel x86 architecture has been backwards compatible back to the 8080 processor (1974) — supporting a large amount of legacy x86 code (*e.g.*, Microsoft products), locking PC users into x86 CISC.
- Intel earns much more money than manufacturers of RISC chips, and so is able to spend much more on research, design and manufacturing (keeping CISC chips competitive with RISC).

RISC vs CISC today

Despite RISC being technically better, still the most popular desktop/server family of chips (Intel x86) is not RISC:

- Intel x86 architecture has been backwards compatible back to the 8080 processor (1974) — supporting a large amount of legacy x86 code (*e.g.*, Microsoft products), locking PC users into x86 CISC.
- Intel earns much more money than manufacturers of RISC chips, and so is able to spend much more on research, design and manufacturing (keeping CISC chips competitive with RISC).

Note however that:

- Mobile phones are dominated by RISC archs (ARM)

RISC vs CISC today

Despite RISC being technically better, still the most popular desktop/server family of chips (Intel x86) is not RISC:

- Intel x86 architecture has been backwards compatible back to the 8080 processor (1974) — supporting a large amount of legacy x86 code (*e.g.*, Microsoft products), locking PC users into x86 CISC.
- Intel earns much more money than manufacturers of RISC chips, and so is able to spend much more on research, design and manufacturing (keeping CISC chips competitive with RISC).

Note however that:

- Mobile phones are dominated by RISC archs (ARM)
- Mac transition to Apple silicon (2020)!

RISC vs CISC today

Despite RISC being technically better, still the most popular desktop/server family of chips (Intel x86) is not RISC:

- Intel x86 architecture has been backwards compatible back to the 8080 processor (1974) — supporting a large amount of legacy x86 code (*e.g.*, Microsoft products), locking PC users into x86 CISC.
- Intel earns much more money than manufacturers of RISC chips, and so is able to spend much more on research, design and manufacturing (keeping CISC chips competitive with RISC).

Note however that:

- Mobile phones are dominated by RISC archs (ARM)
- Mac transition to Apple silicon (2020)!
- Modern Intel processors are also RISC 'under the hood'

RISC vs CISC today

Despite RISC being technically better, still the most popular desktop/server family of chips (Intel x86) is not RISC:

- Intel x86 architecture has been backwards compatible back to the 8080 processor (1974) — supporting a large amount of legacy x86 code (*e.g.*, Microsoft products), locking PC users into x86 CISC.
- Intel earns much more money than manufacturers of RISC chips, and so is able to spend much more on research, design and manufacturing (keeping CISC chips competitive with RISC).

Note however that:

- Mobile phones are dominated by RISC archs (ARM)
- Mac transition to Apple silicon (2020)!
- Modern Intel processors are also RISC 'under the hood'
- ARM provides dedicated instructions for cryptography (AES), just as x86 does.

Aside: extreme RISC

Some workloads are so extremely skewed towards certain instructions that it makes sense to abandon traditional CPU archs (including RISC) and use special purpose processors.

Aside: extreme RISC

Some workloads are so extremely skewed towards certain instructions that it makes sense to abandon traditional CPU archs (including RISC) and use special purpose processors.

- GPUs (graphics processing unit) which exploit the unusually high degree of SIMD parallelism of most graphics and image processing algorithms.

Aside: extreme RISC

Some workloads are so extremely skewed towards certain instructions that it makes sense to abandon traditional CPU archs (including RISC) and use special purpose processors.

- GPUs (graphics processing unit) which exploit the unusually high degree of SIMD parallelism of most graphics and image processing algorithms.
- Bitcoin ‘mining’ lead to dedicated chips that compute nothing but SHA256.

Aside: extreme RISC

Some workloads are so extremely skewed towards certain instructions that it makes sense to abandon traditional CPU archs (including RISC) and use special purpose processors.

- GPUs (graphics processing unit) which exploit the unusually high degree of SIMD parallelism of most graphics and image processing algorithms.
- Bitcoin 'mining' lead to dedicated chips that compute nothing but SHA256.
- Google's TPU (Tensor Processing Units) for speeding up computation in neural nets (typically 15x - 30x vs GPU), and more energy efficient (70x vs GPU, 200x vs CPU).
  - Basically matrix multiplication and activation function engine. Currently a **lot** of work in this space.

The RISC-V processor

The first successful RISC architecture was MIPS, led by John Hennessy (Stanford). It has since been superseded by **RISC-V**.

The RISC-V processor

The first successful RISC architecture was MIPS, led by John Hennessy (Stanford). It has since been superseded by **RISC-V**.

RISC-V is **open source** and has an extremely clean and simple design. For those reasons it has emerged as a serious competitor to ARM.

The RISC-V processor

The first successful RISC architecture was MIPS, led by John Hennessy (Stanford). It has since been superseded by **RISC-V**.

RISC-V is **open source** and has an extremely clean and simple design. For those reasons it has emerged as a serious competitor to ARM.

Several industrial strength compilers to RISC-V exist, including LLVM and GCC.

This lecture

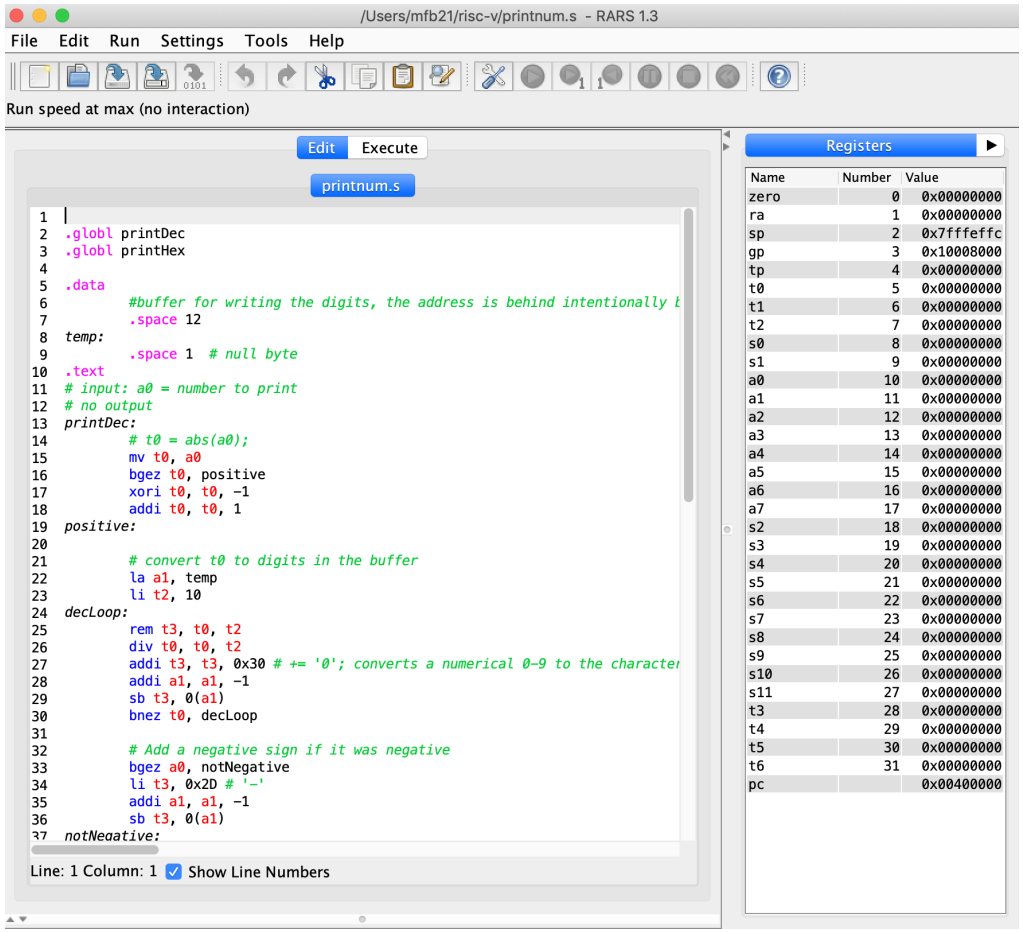
In this lecture, we provide a basic overview of RISC-V which bear on the subject of code generation.

RARS

You are expected to familiarise yourself with RISC-V programming on your own — with a RISC-V simulator, this should not be difficult.

RARS

You are expected to familiarise yourself with RISC-V programming on your own — with a RISC-V simulator, this should not be difficult. I recommend RARS:




```
1
2  .globl printDec
3  .globl printHex
4
5  .data
6          #buffer for writing the digits, the address is behind intentionally l
7          .space 12
8  temp:
9          .space 1  # null byte
10 .text
11 # input: a0 = number to print
12 # no output
13 printDec:
14         # t0 = abs(a0);
15         mv t0, a0
16         bgez t0, positive
17         xori t0, t0, -1
18         addi t0, t0, 1
19 positive:
20
21         # convert t0 to digits in the buffer
22         la a1, temp
23         li t2, 10
24 decLoop:
25         rem t3, t0, t2
26         div t0, t0, t2
27         addi t3, t3, 0x30 # += '0'; converts a numerical 0-9 to the character
28         addi a1, a1, -1
29         sb t3, 0(a1)
30         bnez t0, decLoop
31
32         # Add a negative sign if it was negative
33         bgez a0, notNegative
34         li t3, 0x2D # '-'
35         addi a1, a1, -1
36         sb t3, 0(a1)
37 notNegative:
```

Download **`*.jar`** at

`https://github.com/TheThirdOne/rars/releases/download/continuous/rars_27a7c1f.jar`

Easy to launch:

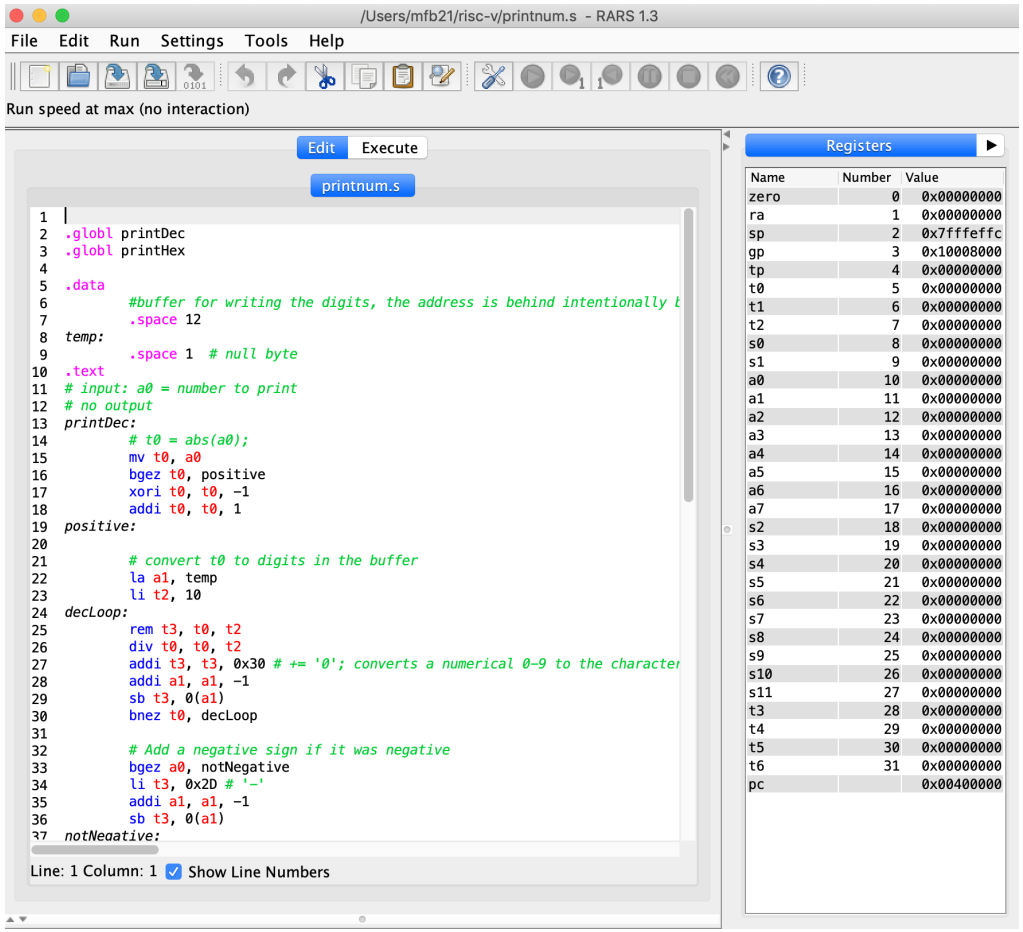
```
java -jar rars_27a7c1f.jar
```

Instructions supported:

`https://github.com/TheThirdOne/rars/wiki/Supported-Instructions`

RARS

You are expected to familiarise yourself with RISC-V programming on your own — with a RISC-V simulator, this should not be difficult. I recommend RARS:




```
1
2  .globl printDec
3  .globl printHex
4
5  .data
6          #buffer for writing the digits, the address is behind intentionally l
7          .space 12
8  temp:
9          .space 1  # null byte
10 .text
11 # input: a0 = number to print
12 # no output
13 printDec:
14         # t0 = abs(a0);
15         mv t0, a0
16         bgez t0, positive
17         xori t0, t0, -1
18         addi t0, t0, 1
19 positive:
20
21         # convert t0 to digits in the buffer
22         la a1, temp
23         li t2, 10
24 decLoop:
25         rem t3, t0, t2
26         div t0, t0, t2
27         addi t3, t3, 0x30 # += '0'; converts a numerical 0-9 to the character
28         addi a1, a1, -1
29         sb t3, 0(a1)
30         bnez t0, decLoop
31
32         # Add a negative sign if it was negative
33         bgez a0, notNegative
34         li t3, 0x2D # '-'
35         addi a1, a1, -1
36         sb t3, 0(a1)
37 notNegative:
```

Download **`*.jar`** at

`https://github.com/TheThirdOne/rars/releases/download/continuous/rars_27a7c1f.jar`

Easy to launch:

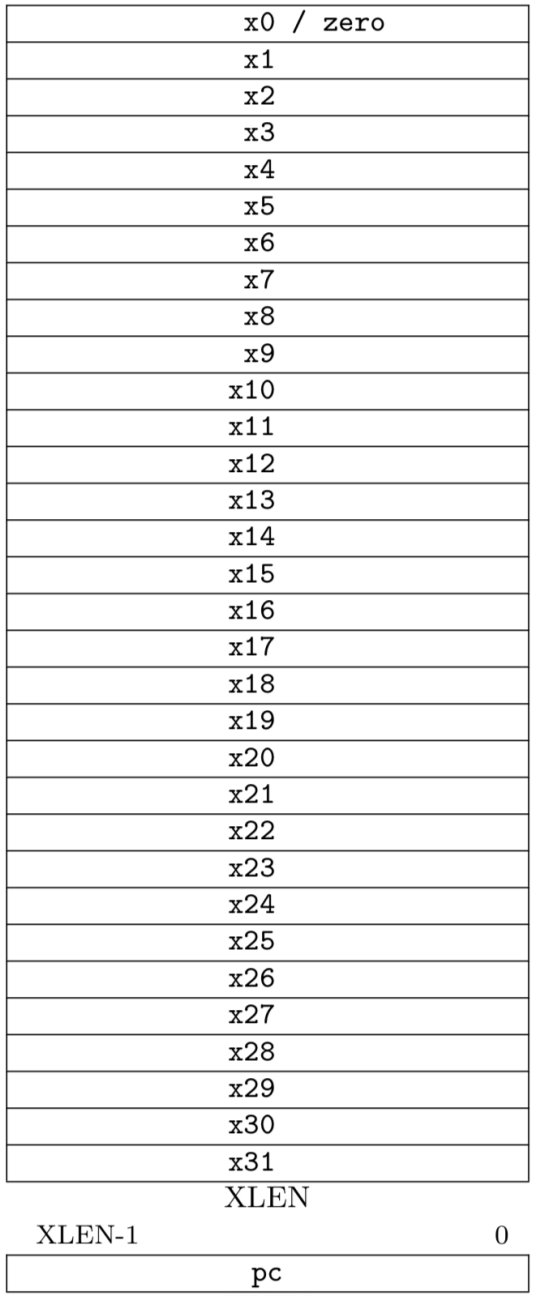
```
java -jar rars_27a7c1f.jar
```

Instructions supported:

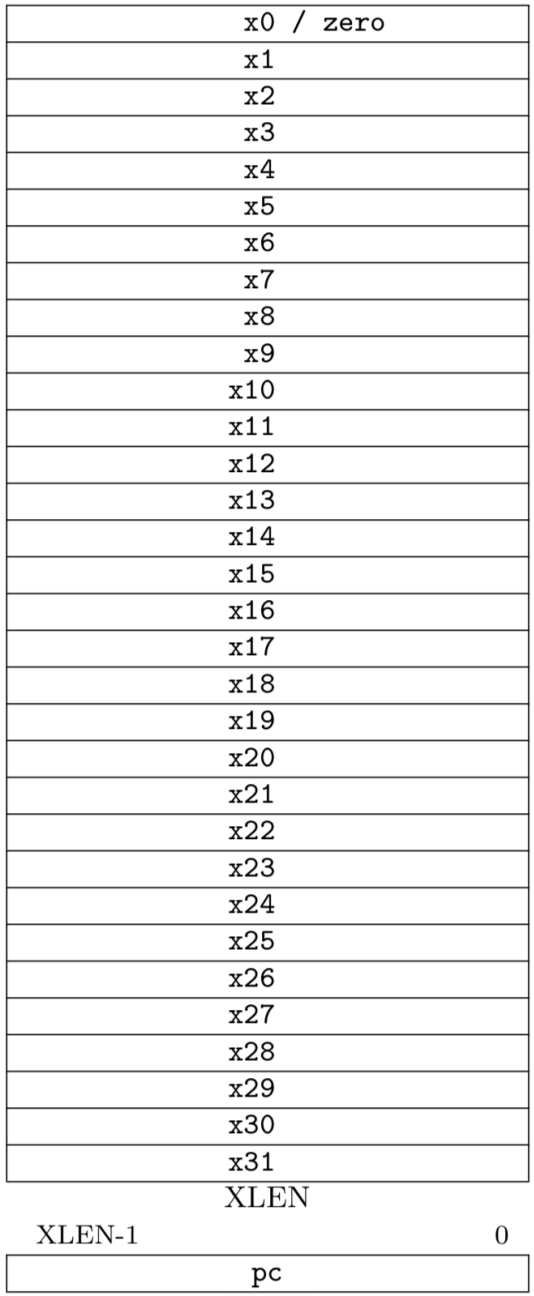
`https://github.com/TheThirdOne/rars/wiki/Supported-Instructions`

The editor also has useful ‘IntelliSense’-style help.

RISC-V registers — all are 32 bits:

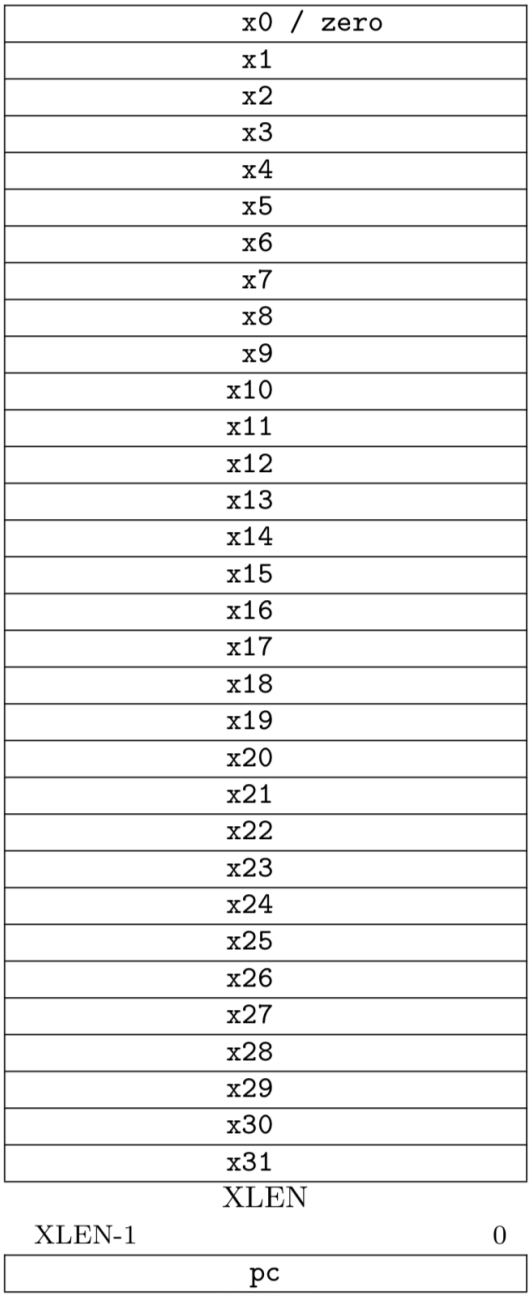


RISC-V registers — all are 32 bits:



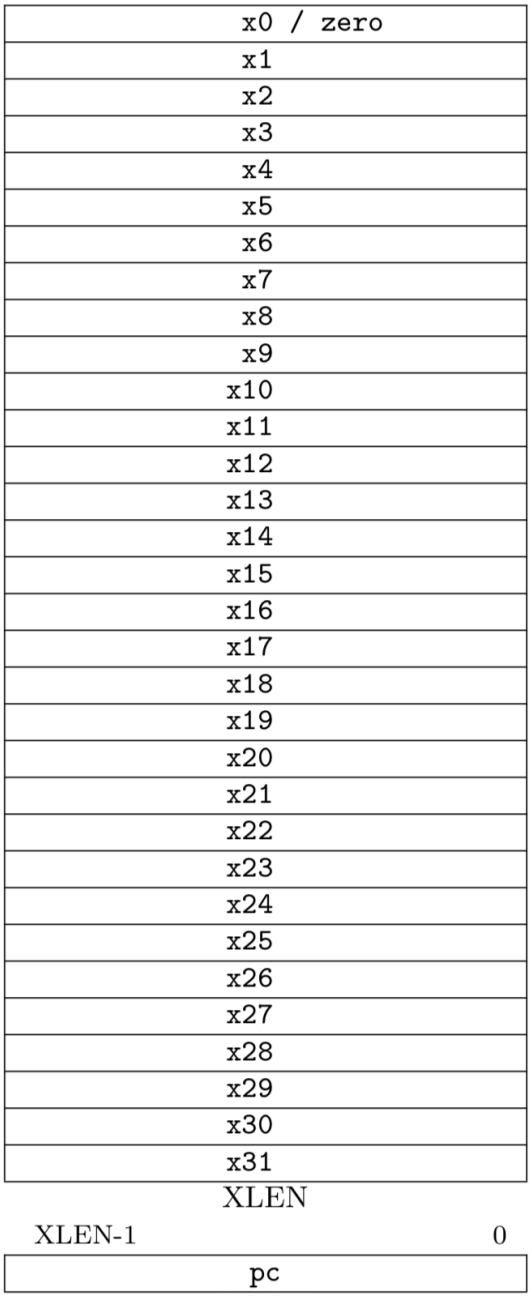
- a register **`x0`** ‘hard-wired’ to 0
  - can be used as target register for any instruction whose result is to be discarded
  - can also be used as a source of the constant 0 if needed

RISC-V registers — all are 32 bits:



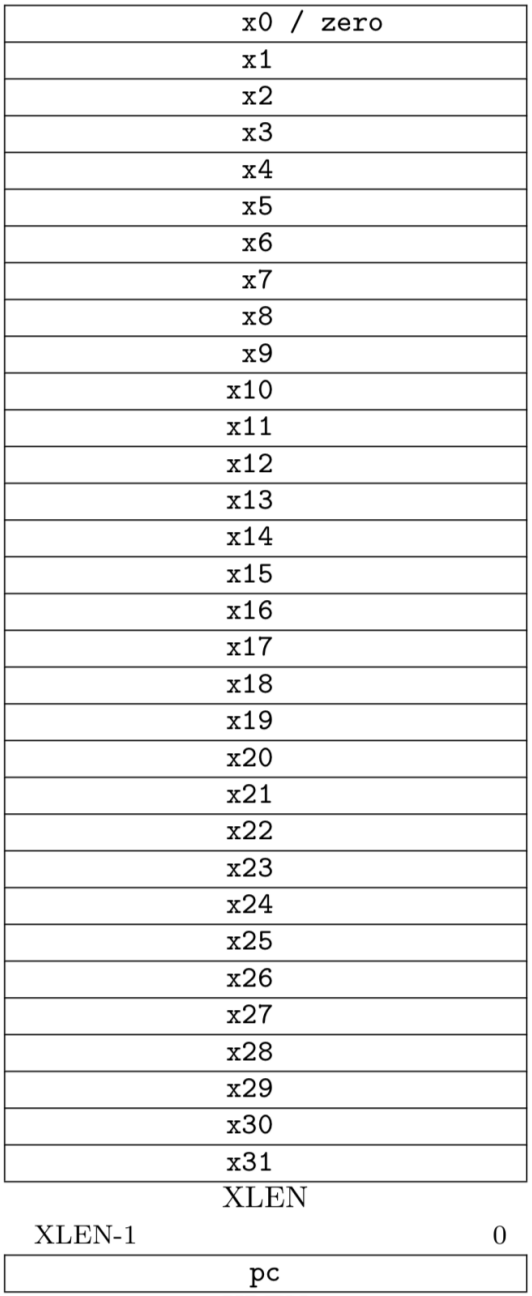
- a register **`x0`** 'hard-wired' to 0
  - can be used as target register for any instruction whose result is to be discarded
  - can also be used as a source of the constant 0 if needed
- general purpose registers **`x1`** – **`x31`**, usable as:
  - boolean flags
  - signed or unsigned integers
  - pointer values

RISC-V registers — all are 32 bits:



- a register **`x0`** 'hard-wired' to 0
  - can be used as target register for any instruction whose result is to be discarded
  - can also be used as a source of the constant 0 if needed
- general purpose registers **`x1`** – **`x31`**, usable as:
  - boolean flags
  - signed or unsigned integers
  - pointer values
- A program counter (PC) register
  - points to instruction to do next
  - cannot directly be written or read
  - only affected through side-effects of certain instructions

RISC-V registers — all are 32 bits:



- a register **`x0`** 'hard-wired' to 0
  - can be used as target register for any instruction whose result is to be discarded
  - can also be used as a source of the constant 0 if needed
- general purpose registers **`x1`** – **`x31`**, usable as:
  - boolean flags
  - signed or unsigned integers
  - pointer values
- A program counter (PC) register
  - points to instruction to do next
  - cannot directly be written or read
  - only affected through side-effects of certain instructions
- SP/FP registers

RISC-V registers: usage conventions

Register name	Symbolic name	Description
32 integer registers		
x0	Zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporary
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function argument / return value
x12–17	a2–7	Function argument
x18–27	s2–11	Saved register
x28–31	t3–6	Temporary

RISC-V registers: usage conventions

Register name	Symbolic name	Description
32 integer registers		
x0	Zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporary
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function argument / return value
x12–17	a2–7	Function argument
x18–27	s2–11	Saved register
x28–31	t3–6	Temporary

Interesting for us: mostly
`x0` (**`zero`**), **`x1`** (**`ra`**),
`x2` (**`sp`**), **`x8`** (**`fp`**),
`x5` (**`t0`**), **`x6`** (**`t1`**), and
`x10` (**`a0`**).

RISC-V registers: usage conventions

Register name	Symbolic name	Description
32 integer registers		
x0	Zero	Always zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / alternate return address
x6–7	t1–2	Temporary
x8	s0/fp	Saved register / frame pointer
x9	s1	Saved register
x10–11	a0–1	Function argument / return value
x12–17	a2–7	Function argument
x18–27	s2–11	Saved register
x28–31	t3–6	Temporary

Interesting for us: mostly `x0` (`zero`), `x1` (`ra`), `x2` (`sp`), `x8` (`fp`), `x5` (`t0`), `x6` (`t1`), and `x10` (`a0`).

These “roles” are just **usage conventions** — but you should adhere to them if you want to interface with other RISC-V software (*e.g.*, an **assembler**).

RISC-V Memory access

RISC-V has address space of 2^{32} bytes, for all memory accesses.

RISC-V Memory access

RISC-V has address space of 2^{32} bytes, for all memory accesses.

- Address space is **circular**, so that the byte at address $2^{32} - 1$ is adjacent to the byte at address zero.

RISC-V Memory access

RISC-V has address space of 2^{32} bytes, for all memory accesses.

- Address space is **circular**, so that the byte at address $2^{32} - 1$ is adjacent to the byte at address zero.
- A word of memory is defined as 32 bits (4 bytes);
  A halfword is 16 bits (2 bytes);
  A byte 8 bits.

RISC-V Memory access

RISC-V has address space of 2^{32} bytes, for all memory accesses.

- Address space is **circular**, so that the byte at address $2^{32} - 1$ is adjacent to the byte at address zero.
- A word of memory is defined as 32 bits (4 bytes);
  A halfword is 16 bits (2 bytes);
  A byte 8 bits.
- Some instructions are fixed-length 32-bit instructions that **must** be aligned on 32-bit boundaries (*i.e.*, at memory locations divisible by 4).

RISC-V Memory access

RISC-V has address space of 2^{32} bytes, for all memory accesses.

- Address space is **circular**, so that the byte at address $2^{32} - 1$ is adjacent to the byte at address zero.
- A word of memory is defined as 32 bits (4 bytes);
  A halfword is 16 bits (2 bytes);
  A byte 8 bits.
- Some instructions are fixed-length 32-bit instructions that **must** be aligned on 32-bit boundaries (*i.e.*, at memory locations divisible by 4).
- Storing or loading data at "misaligned" locations is allowed — but accesses to aligned addresses may be faster
  - simple RISC-V CPUs may implement unaligned accesses with slow software emulation driven from an alignment failure interrupt.

RISC-V instructions

RISC-V instructions are all 32 bits long in memory, and are organised into the following functional groups:

RISC-V instructions

RISC-V instructions are all 32 bits long in memory, and are organised into the following functional groups:

1. Load and store (memory access)
2. Immediates (handling of constants)
3. Computational (e.g. integer arithmetic and boolean logic)
4. Jump and branch (conditional and unconditional)
5. Some others (e.g. SIMD, vectoring)

RISC-V instructions

RISC-V instructions are all 32 bits long in memory, and are organised into the following functional groups:

1. Load and store (memory access)
2. Immediates (handling of constants)
3. Computational (e.g. integer arithmetic and boolean logic)
4. Jump and branch (conditional and unconditional)
5. Some others (e.g. SIMD, vectoring)

Note. RISC-V processors use a simple load/store architecture — all operations (e.g. addition, comparison) are performed on operands held in **registers**

RISC-V instructions

RISC-V instructions are all 32 bits long in memory, and are organised into the following functional groups:

1. Load and store (memory access)
2. Immediates (handling of constants)
3. Computational (e.g. integer arithmetic and boolean logic)
4. Jump and branch (conditional and unconditional)
5. Some others (e.g. SIMD, vectoring)

Note. RISC-V processors use a simple load/store architecture — all operations (e.g. addition, comparison) are performed on operands held in **registers**

- To perform an operation on data stored in memory, it must first be loaded into a register, and later stored back in memory after the operation.
- Main memory is accessed **only** through load and store instructions.

RISC-V instructions

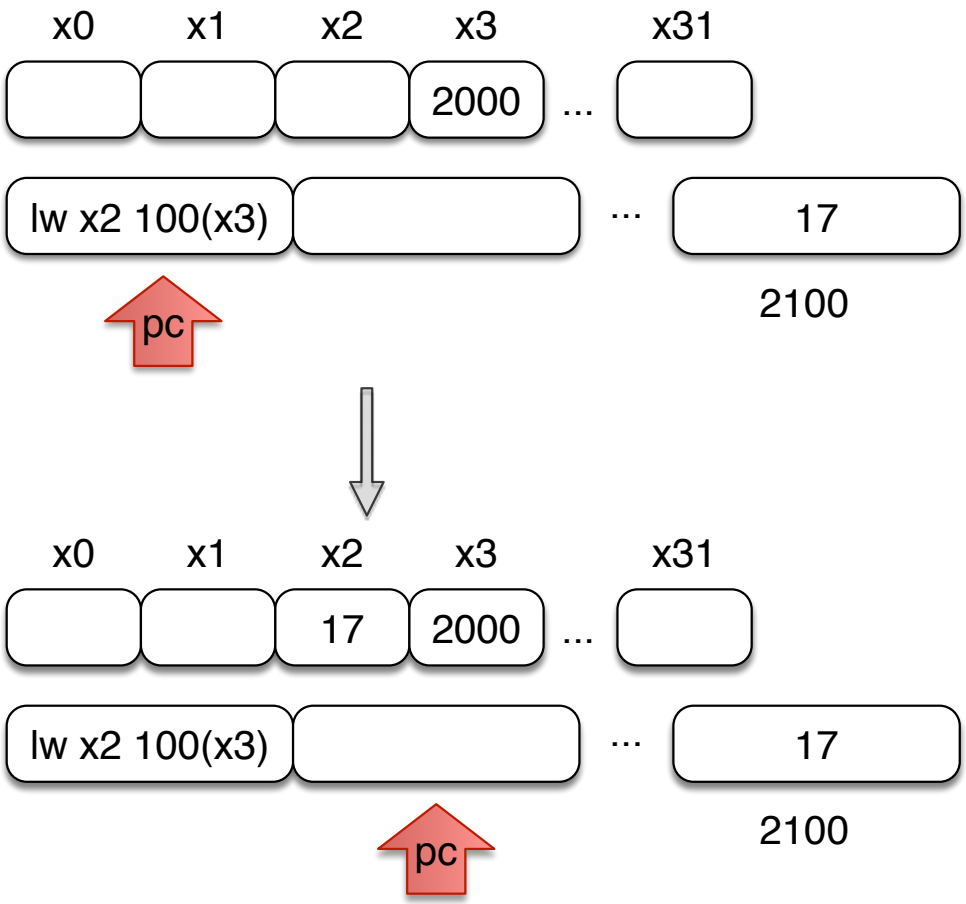
The **load word** instruction

```
lw reg1 offset(reg2)
```

(where `offset` is a 16-bit integer) adds the content $\texttt{reg}_2$ and the 16 bit value `offset`, obtaining a new number `n`, and then looks up the 32 bit value stored in memory at `n`. That value is then loaded into register $\texttt{reg}_1$ as a signed integer.

The sum of $\texttt{reg}_2$ and `offset` must be word aligned (i.e. the two least significant bits must be 0), otherwise an error will occur.

RISC-V instructions



RISC-V instructions

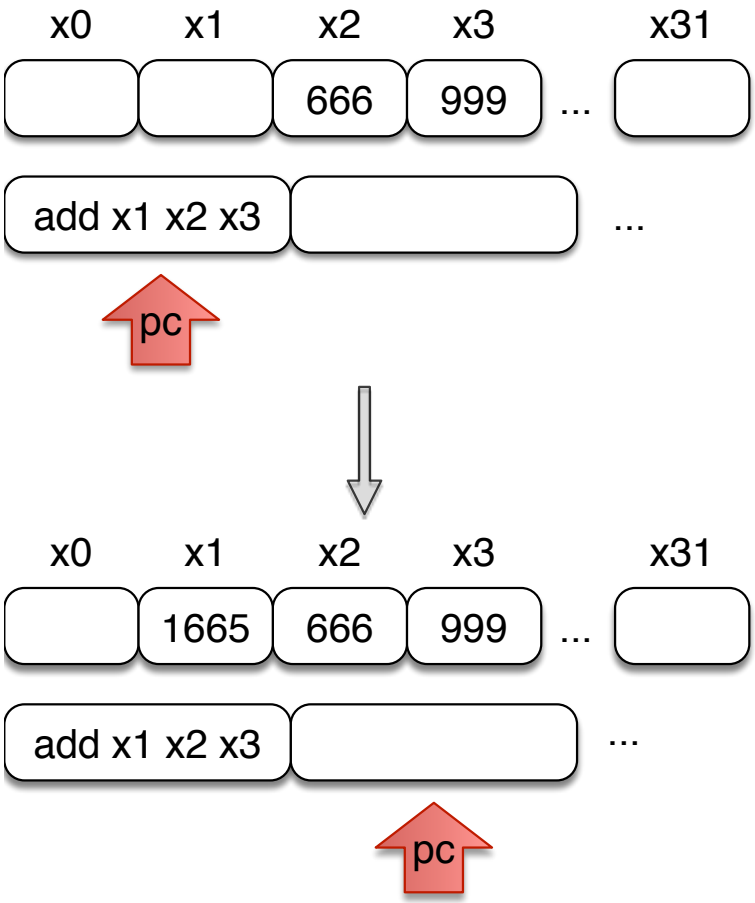
The **add** instruction

add $\texttt{reg}_1$ $\texttt{reg}_2$ $\texttt{reg}_3$

Adds the contents of registers $\texttt{reg}_2$ and $\texttt{reg}_3$, and stores the result in $\texttt{reg}_1$.

Note that the registers $\texttt{reg}_1$, $\texttt{reg}_2$ and $\texttt{reg}_3$ don't have to be distinct.

RISC-V instructions



RISC-V instructions

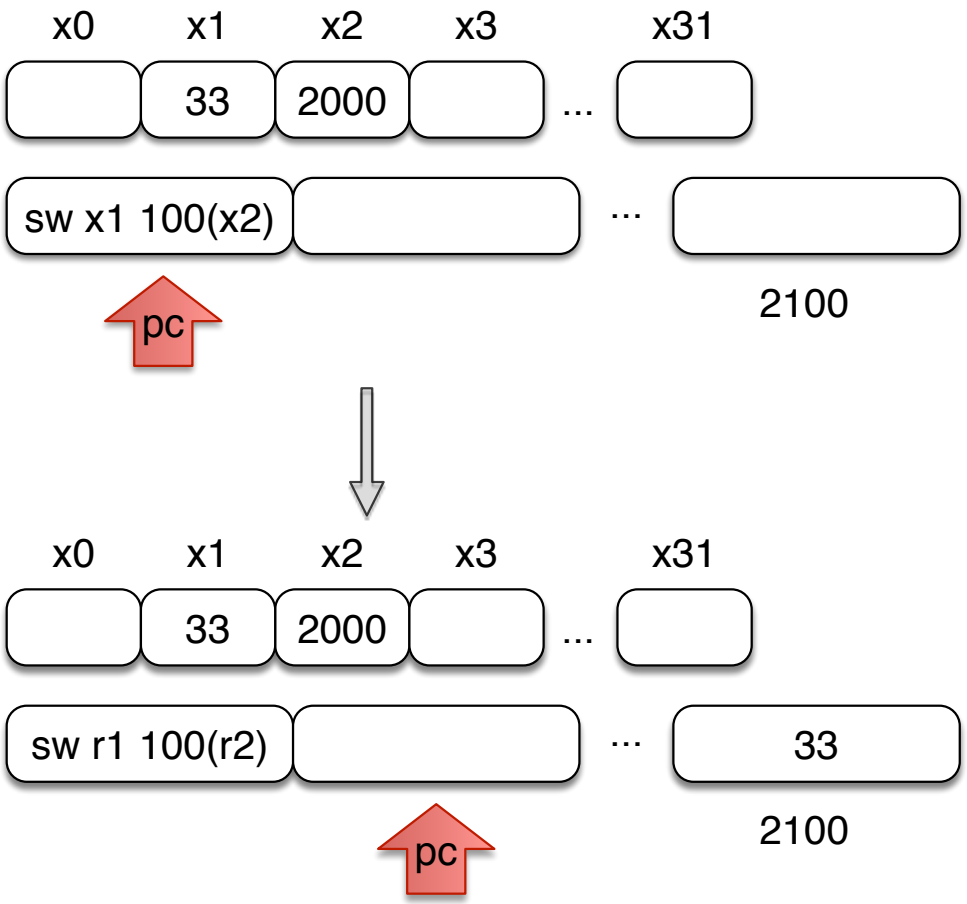
The **store word** instruction

```
sw reg1 offset(reg2)
```

(where `offset` is an integer) stores the 32 bit word currently in $\texttt{reg}_1$ at the address obtained by adding the 16 bit value `offset` to the content of register $\texttt{reg}_2$.

The sum of $\texttt{reg}_2$ and `offset` must be word aligned (*i.e.* the two least significant bits must be 0), otherwise an error will occur.

RISC-V instructions



RISC-V instructions

The **add immediate** instruction

```
addi reg1 reg2 imm
```

Adds the 16 bit **signed** integer `imm` to the word currently in $\texttt{reg}_2$, storing the result in register $\texttt{reg}_1$.

RISC-V instructions

The **add immediate** instruction

```
addi reg1 reg2 imm
```

Adds the 16 bit **signed** integer `imm` to the word currently in $\texttt{reg}_2$, storing the result in register $\texttt{reg}_1$.

- What is “immediate” about this, is the constant `imm` — the value of which is encoded as part of the 32-bit instruction itself (and so is immediately present), rather than being an operand to be retrieved.

RISC-V instructions

The **add immediate** instruction

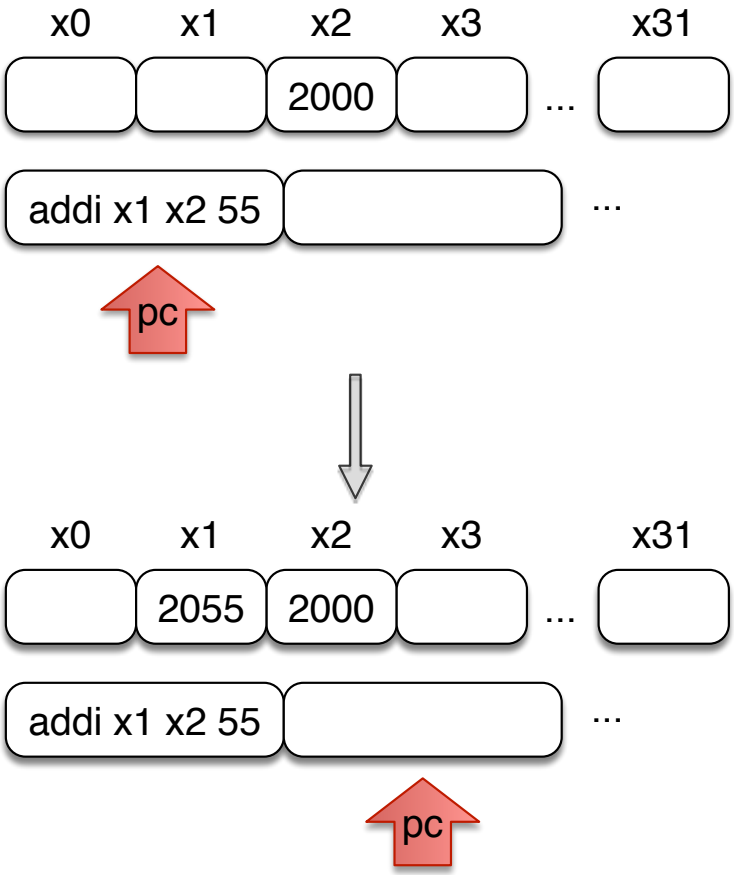
```
addi reg1 reg2 imm
```

Adds the 16 bit **signed** integer `imm` to the word currently in $\texttt{reg}_2$, storing the result in register $\texttt{reg}_1$.

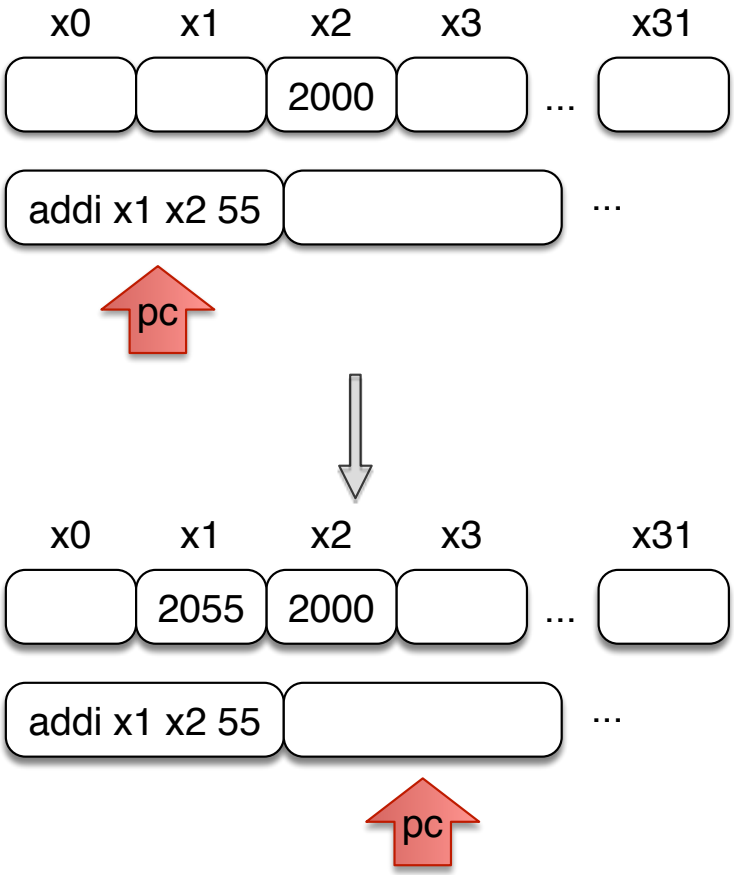
- What is "immediate" about this, is the constant `imm` — the value of which is encoded as part of the 32-bit instruction itself (and so is immediately present), rather than being an operand to be retrieved.

Note `imm` is signed (*i.e.*, can be negative), so we can increment and decrement *e.g.* the **SP**, without also needing a corresponding subtraction operation.

RISC-V instructions



RISC-V instructions



Question. what is the effect of `addi` $\texttt{reg}_1$ $\texttt{reg}_2$ `0`?

RISC-V instructions

The **load immediate** ‘pseudo-instruction’

```
li reg imm
```

stores the **32 bit** integer `imm` in register `reg`.

RISC-V instructions

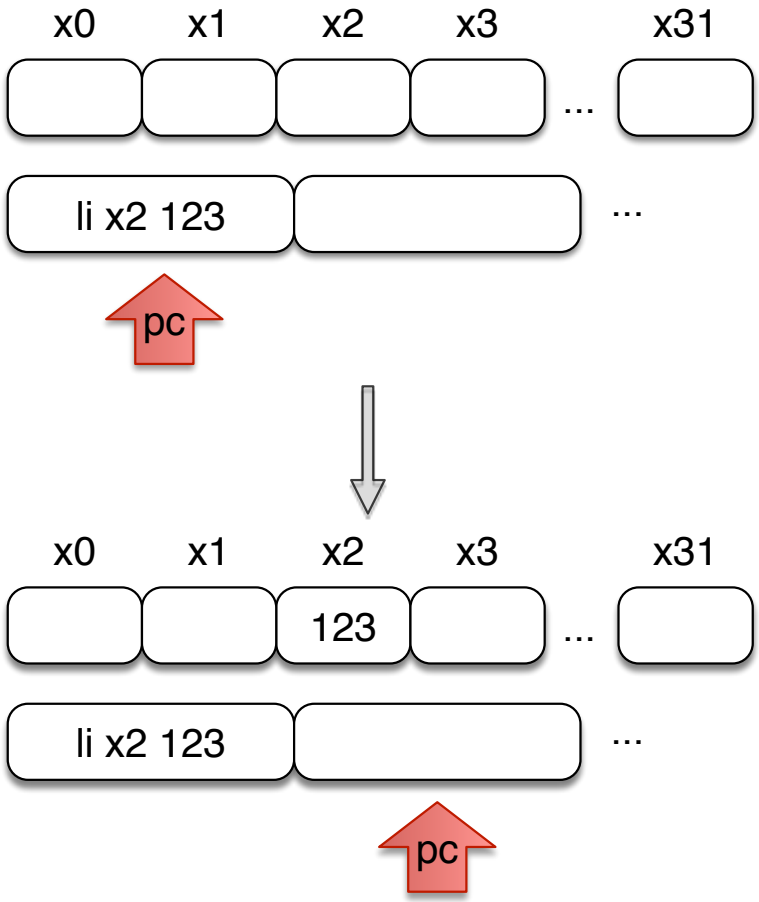
The **load immediate** ‘pseudo-instruction’

```
li reg imm
```

stores the **32 bit** integer `imm` in register `reg`.

- This is a ‘pseudo-instruction’ in that, in fact, RISC-V cannot load 32 bit words directly — there is no RISC-V assembly instruction that directly implements this.
- Instead, the RISC-V assembler will automatically expand “`li reg imm`” into a sequence of real assembler instructions. When compiling you can easily treat pseudo instructions as real instructions.

RISC-V instructions



Our first RISC-V program

Let's write the program `x5` $:=$ `7+5`:

Our first RISC-V program

Let's write the program **`x5`** $:=$ **`7+5`**:

```
li x6 7
```

Our first RISC-V program

Let's write the program **`x5`** $:=$ **`7+5`**:

```
li x6 7
li x5 5
```

Our first RISC-V program

Let's write the program **`x5`** $:=$ **`7+5`**:

```
li x6 7
li x5 5
add x5 x5 x6
```

Our second RISC-V program

Let's write **`x10 := 1+2+3+4`**, in accumulator machine form.

- Drawing elements from the stack
  (*e.g.*, as if they were there previously)
- Adding results successively into
  the accumulator, **`x10`**.
- Reading items from the stack into a
  temporary register, **`x6`** (as RISC-V cannot
  do arithmetic directly on data in memory)

Our second RISC-V program

Let's write **`x10`** := **`1+2+3+4`**, in accumulator machine form.

- Drawing elements from the stack
  (*e.g.*, as if they were there previously)
- Adding results successively into
  the accumulator, **`x10`**.
- Reading items from the stack into a
  temporary register, **`x6`** (as RISC-V cannot
  do arithmetic directly on data in memory)

Simulating a more limited CPU architecture,
or a situation in which the other registers are
already in use storing other important data.

Stack conventions

We adopt some conventions for realising a stack on RISC-V (common elsewhere as well):

Stack conventions

We adopt some conventions for realising a stack on RISC-V (common elsewhere as well):

- The stack grows downwards
  (*i.e.*, from high to low addresses).

Stack conventions

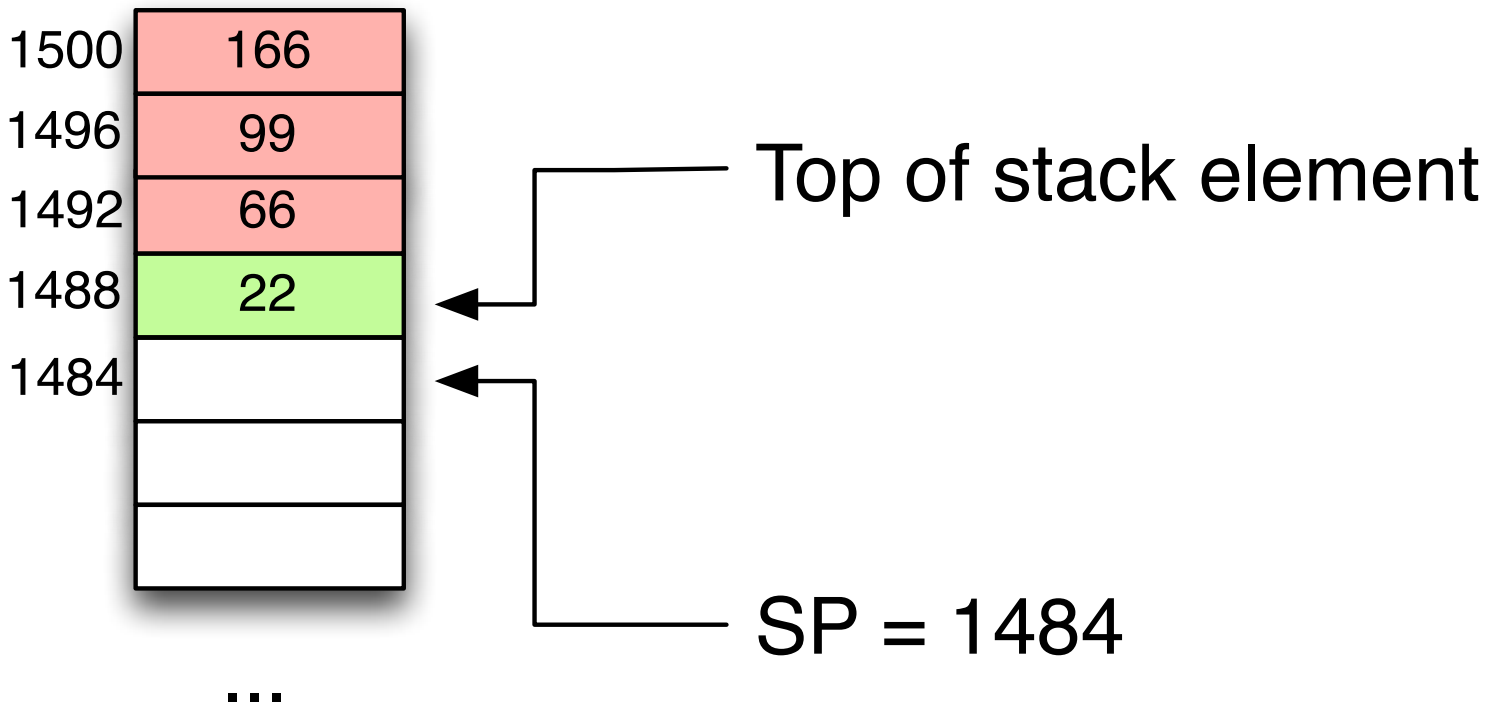
We adopt some conventions for realising a stack on RISC-V (common elsewhere as well):

- The stack grows downwards
  (*i.e.*, from high to low addresses).
- The stack pointer `sp` points to the first **free** memory cell in front of (immediately below) the stack.

Stack conventions

We adopt some conventions for realising a stack on RISC-V (common elsewhere as well):

- The stack grows downwards (*i.e.*, from high to low addresses).
- The stack pointer `sp` points to the first **free** memory cell in front of (immediately below) the stack.



Our second RISC-V program

What we want to simulate:

Our second RISC-V program

What we want to simulate:

```
setup:
   PushImm 4 ; [ ... 4 ] <- SP
   PushImm 3 ; [ ... 4 3 ] <- SP
   PushImm 2 ; [ ... 4 3 2 ] <- SP
   PushImm 1 ; [ ... 4 3 2 1 ] <- SP
```

Our second RISC-V program

What we want to simulate:

```
setup:
    PushImm 4 ; [ ... 4 ] <- SP
    PushImm 3 ; [ ... 4 3 ] <- SP
    PushImm 2 ; [ ... 4 3 2 ] <- SP
    PushImm 1 ; [ ... 4 3 2 1 ] <- SP
sum:
    Plus  ; [ ... 4 3 3 ] <- SP
    Plus  ; [ ... 4 6 ] <- SP
    Plus  ; [ ... 10 ] <- SP
```

How we simulate it:

```
PushImm n ⟶
  li      t0 n
  sw      t0 0(sp)
  addi    sp sp -4
```

```
Plus  ⟶
    lw          t1, 4(sp)
    addi        sp, sp, 4
    lw          t2, 4(sp)
    addi        sp, sp, 4
    add         t1, t1, t2
    sw          t1, (sp)
    addi        sp, sp, -4
```

Our second RISC-V program

What we want to simulate:

```
setup:
    PushImm 4 ; [ ... 4 ] <- SP
    PushImm 3 ; [ ... 4 3 ] <- SP
    PushImm 2 ; [ ... 4 3 2 ] <- SP
    PushImm 1 ; [ ... 4 3 2 1 ] <- SP
sum:
    Plus  ; [ ... 4 3 3 ] <- SP
    Plus  ; [ ... 4 6 ] <- SP
    Plus  ; [ ... 10 ] <- SP
```

How we simulate it:

```
PushImm n ⟶
  li      t0 n
  sw      t0 0(sp)
  addi    sp sp -4
```

```
Plus  ⟶
    lw          t1, 4(sp)
    addi        sp, sp, 4
    lw          t2, 4(sp)
    addi        sp, sp, 4
    add         t1, t1, t2
    sw          t1, (sp)
    addi        sp, sp, -4
```

By convention we write **`sp`** for the stack pointer, **`t0`** or **`t1`** for the temporary, and **`a0`** for the accumulator.

RISC-V

We will soon write a compiler that compiles a simple language with procedures to RISC-V code.

RISC-V

We will soon write a compiler that compiles a simple language with procedures to RISC-V code.

To understand this, you need to familiarise yourself with RISC-V in the tutorials and in self study.

RISC-V

We will soon write a compiler that compiles a simple language with procedures to RISC-V code.

To understand this, you need to familiarise yourself with RISC-V in the tutorials and in self study.

Fortunately, RISC-V assembly code is really straightforward (though you have to be careful to specify all of the arguments).

Programming in RISC-V assembly

Here is an example of using names:

```
        .text
        .align 2
        .globl main
main:
        sub sp, sp, 32
        sw ra, 20(sp)
        ...
loop:
        lw t6, 28(sp)
        ...
        ble t0, 100, loop
```

Programming in RISC-V assembly

Here is an example of using names:

```
        .text
        .align 2
        .globl main
main:
        sub sp, sp, 32
        sw ra, 20(sp)
        ...
loop:
        lw t6, 28(sp)
        ...
        ble t0, 100, loop
```

- `main` is a **global** name in the sense that other programs can use it.
  - It is the declaration (assembler directive) `.globl main` that makes `main` global.

Programming in RISC-V assembly

Here is an example of using names:

```
        .text
        .align 2
        .globl main
main:
        sub sp, sp, 32
        sw ra, 20(sp)
        ...
loop:
        lw t6, 28(sp)
        ...
        ble t0, 100, loop
```

- `main` is a **global** name in the sense that other programs can use it.
  - It is the declaration (assembler directive) `.globl main` that makes `main` global.
- `loop` is a **local** name: it can only be used (jumped to) inside this program.

Assembler, the program

The assembler processes a source file line by line, translating assembly instructions. In the process, it also keeps track of the size of each instruction.

Assembler, the program

The assembler processes a source file line by line, translating assembly instructions. In the process, it also keeps track of the size of each instruction.

```
loop:
      sub sp, sp, 32
      sw ra, 20(sp)
```

Assembler, the program

The assembler processes a source file line by line, translating assembly instructions. In the process, it also keeps track of the size of each instruction.

```
loop:
        sub sp, sp, 32
        sw ra, 20(sp)
```

- When the assembler encounters a line starting with a label, like `loop:` ··· it calculates what address in memory the instruction just below would be at, and stores the pair of label and address in its symbol table.

Assembler, the program

The assembler processes a source file line by line, translating assembly instructions. In the process, it also keeps track of the size of each instruction.

```
loop:
        sub sp, sp, 32
        sw ra, 20(sp)
```

- When the assembler encounters a line starting with a label, like `loop:` ··· it calculates what address in memory the instruction just below would be at, and stores the pair of label and address in its symbol table.
- If it encounters this label later, *e.g.* `ble t0, 100, loop`, the assembler replaces the label with the address (if local, otherwise the linker does this).

Helpers

Assembly languages typically offer various features making assembly programming easier. Here are some RISC-V examples.

- Data layout directives
- Pseudo instructions
- Alignment instructions

Helpers

Assembly languages typically offer various features making assembly programming easier. Here are some RISC-V examples.

- Data layout directives
- Pseudo instructions
- Alignment instructions

We have already touched on pseudo-instructions, which are like a form of macro expansion. We now describe data-layout directives and alignment instructions.

Data layout directives

Data layout directives describe data in a more concise and natural manner than its binary representation.

Data layout directives

Data layout directives describe data in a more concise and natural manner than its binary representation.Example:

```
.asciz "The sum from 0 .. 100 is %d\n"
```

stores characters from the string in memory.

Data layout directives

Data layout directives describe data in a more concise and natural manner than its binary representation.Example:

```
.asciz "The sum from 0 .. 100 is %d\n"
```

stores characters from the string in memory. Alternatively, we can use the `.byte` directive to obtain the same effect:

Data layout directives

Data layout directives describe data in a more concise and natural manner than its binary representation.Example:

```
.asciz "The sum from 0 .. 100 is %d\n"
```

stores characters from the string in memory. Alternatively, we can use the `.byte` directive to obtain the same effect:

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

Data layout directives

Data layout directives describe data in a more concise and natural manner than its binary representation.Example:

```
.asciz "The sum from 0 .. 100 is %d\n"
```

stores characters from the string in memory. Alternatively, we can use the `.byte` directive to obtain the same effect:

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

— but why would you want to?

Data layout directives

Data layout directives describe data in a more concise and natural manner than its binary representation.Example:

```
.asciz "The sum from 0 .. 100 is %d\n"
```

stores characters from the string in memory. Alternatively, we can use the **`.byte`** directive to obtain the same effect:

```
.byte 84, 104, 101, 32, 115, 117, 109, 32
.byte 102, 114, 111, 109, 32, 48, 32, 46
.byte 46, 32, 49, 48, 48, 32, 105, 115
.byte 32, 37, 100, 10, 0
```

— but why would you want to?

The **`.asciz`** directive is easier to read for text strings. (The **`.byte`** directive is useful in other cases, of course.)

Alignment

Most assembler languages have build in support that aligns (some) data automatically.

Alignment

Most assembler languages have build in support that aligns (some) data automatically.

For example RISC-V assembly language has the `.align` command. Putting `.align n` in a line means that the succeedings lines are aligned on a 2^n byte boundary.

Alignment

Most assembler languages have build in support that aligns (some) data automatically.

For example RISC-V assembly language has the **`.align`** command. Putting **`.align n`** in a line means that the succeedings lines are aligned on a 2^n byte boundary.

```
.align 2
.asciiz "Hello world!"
```

Here, **`.align 2`** aligns the next value (the string "Hello world!") on a word boundary on a 32 bit machine.

Conclusion

- RISC-V is an assembly language, but it won't bite
- You might even come to like it
- Get some practice with RISC-V, in advance of Task 2 of the assessed coursework, to help you to do well.