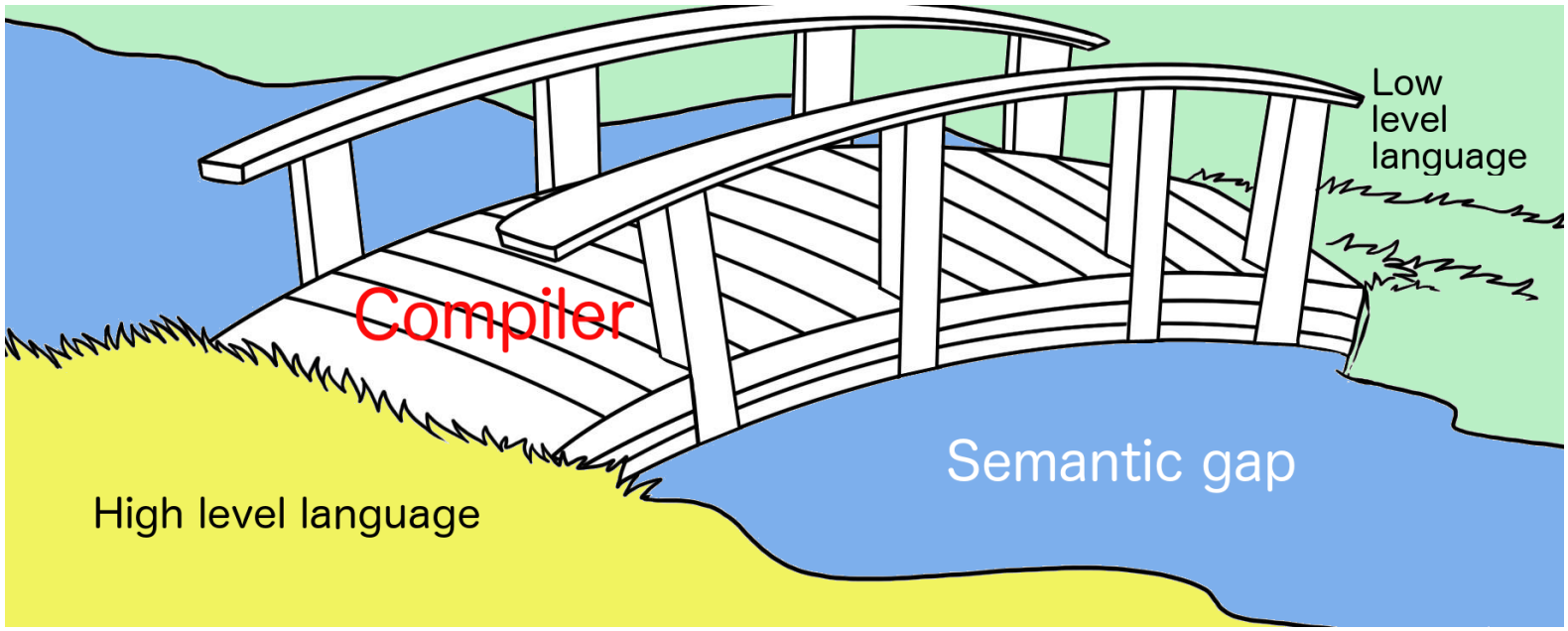
# Compilers and computer architecture: The RISC-V architecture

Martin Berger [1]

November 2019

[1] Email: M.F.Berger@sussex.ac.uk, Office hours: Wed 12-13 in Chi-2R312

# Recall the function of compilers



# Introduction

In previous lectures, we focussed on generating code for simple architectures like the stack machine, or accumulator machines.

Now we want to do something more interesting, generating code for a real CPU.

# Introduction

In previous lectures, we focussed on generating code for simple architectures like the stack machine, or accumulator machines.

Now we want to do something more interesting, generating code for a real CPU.

We focus on the RISC-V (family of) processor(s).

# CISC vs RISC

Processors can roughly be classified as

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

# CISC vs RISC

Processors can roughly be classified as

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

What is the instruction set?

# Instruction set architecture

In a CPU we distinguish between

- **Instruction set architecture**, that is externally visible aspects like the supported data types (e.g. 32 bit Ints, 80 bit floats etc), instructions, number and kinds of registers, addressing modes, memory architecture, interrupt etc.

# Instruction set architecture

In a CPU we distinguish between

- **Instruction set architecture**, that is externally visible aspects like the supported data types (e.g. 32 bit Ints, 80 bit floats etc), instructions, number and kinds of registers, addressing modes, memory architecture, interrupt etc. This is what the programmer uses to get the CPU to do things.

# Instruction set architecture

In a CPU we distinguish between

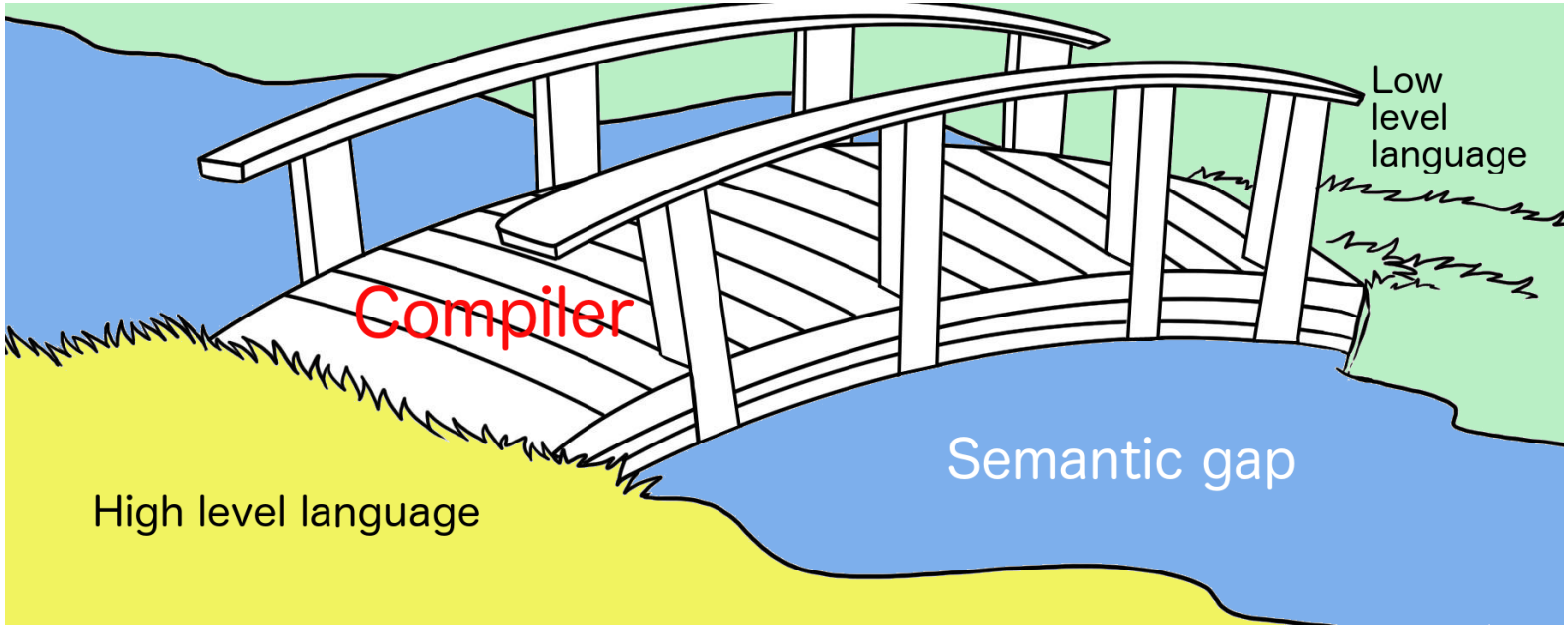
- **Instruction set architecture**, that is externally visible aspects like the supported data types (e.g. 32 bit Ints, 80 bit floats etc), instructions, number and kinds of registers, addressing modes, memory architecture, interrupt etc. This is what the programmer uses to get the CPU to do things.
- **Microarchitecture**, which how the instruction set is implemented. The microarchitecture is not visible to the programmer, and CPUs with different microarchitectures can share a common instruction set. For example Intel and AMD support very similar instruction sets (x86 derived) but have very different microarchitectures.

# Instruction set architecture

In a CPU we distinguish between

- **Instruction set architecture**, that is externally visible aspects like the supported data types (e.g. 32 bit Ints, 80 bit floats etc), instructions, number and kinds of registers, addressing modes, memory architecture, interrupt etc. This is what the programmer uses to get the CPU to do things.
- **Microarchitecture**, which how the instruction set is implemented. The microarchitecture is not visible to the programmer, and CPUs with different microarchitectures can share a common instruction set. For example Intel and AMD support very similar instruction sets (x86 derived) but have very different microarchitectures. The programmer can ignore this, this is for the hardware people.

# Recall:


Low
level
language
Compiler
Semantic gap
High level language

# Instruction set architecture

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

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.

In the past it was thought that making machine commands more powerful would close or narrow the semantic gap, making compiled code faster. Examples of powerful machine commands include directly enabling constructs such as procedure calls, or complicated array access in single instructions.

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

# Instruction set architecture

In some cases CISC architectures led to faster compiled code. But in the 1970s researchers began to study instruction set architecture and compiled code carefully and noticed two things.

- Compilers rarely make 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.
- Implementing complex operations leads to complicated CPU architecture that slow down the execution of simple instructions. Worse: simple operations are slower even when you don't use the complex operations.

# RISC

These empirical insights lead to a radical rethink of instruction set architecture.

# RISC

These empirical insights lead to a radical rethink of instruction set architecture.

Optimise as much as possible the few instructions that are used the most in practise. Try and make them exceedingly fast.

# RISC

These empirical insights lead to a radical rethink of instruction set architecture.

Optimise as much as possible the few instructions that are used the most in practise. Try and make them exceedingly fast.

This makes the task of the compiler (much) harder, but the compiler has to compile a program only once, whereas the CPU would have to support complex instructions all the time.

# RISC

RISC processors like RISC-V are the outcome.

# RISC

RISC processors like RISC-V are the outcome.

One key feature of RISC is that external memory was only accessible by a load or store instruction. All other instructions were limited to internal registers. Hence RISC is also called **load/store architecture**.

# RISC

RISC processors like RISC-V are the outcome.

One key feature of RISC is that external memory was only accessible by a load or store instruction. All other instructions were limited to internal registers. Hence RISC is also 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. (Phones are dominated by RISCish architectures (ARM).) Reasons:

- Large amount of legacy x86 code (e.g. Microsoft products), locking PC users into x86 CISC. x86 has been backwards compatible back to the 8080 processor (1974).
- Intel earns much more money than producers of RISC chips so can spend much more on research, design and manufacturing, keeping CISC chips competitive with RISC.
- 'Under the hood' modern Intel processors are also RISC: the complicated x86 machine instructions are translated at run-time into much simpler underlying RISC microcode (which is not user-visible).
- Both ARM and x86 provide dedicated instructions for cryptography (AES).

# RISC vs CISC today

Despite RISC being technically better, still the most popular desktop/server family of chips (Intel x86) is not RISC. (Phones are dominated by RISCish architectures (ARM).) Reasons:

- Large amount of legacy x86 code (e.g. Microsoft products), locking PC users into x86 CISC. x86 has been backwards compatible back to the 8080 processor (1974).
- Intel earns much more money than producers of RISC chips so can spend much more on research, design and manufacturing, keeping CISC chips competitive with RISC.
- 'Under the hood' modern Intel processors are also RISC: the complicated x86 machine instructions are translated at run-time into much simpler underlying RISC microcode (which is not user-visible).
- Both ARM and x86 provide dedicated instructions for cryptography (AES).

# Aside: extreme RISC

Sometimes a workload is so extremely skewed towards certain commands that it makes sense to abandon traditiona CPU architecture (including RISC) and use special purpose processors.

- GPUs (graphics processing unit) which explore the unusually high degree of SIMD parallelism of most graphics and image processing algorithms.
- Bitcoin 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 original RISC processor was MIPS, John Hennessy in Stanford. For various reasons it has been replaced 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.

# The RISC-V processor

In its basic from, RISC-V has 32 integer registers.

RISC-V is a **load-and-store** architecture, meaning:

# The RISC-V processor

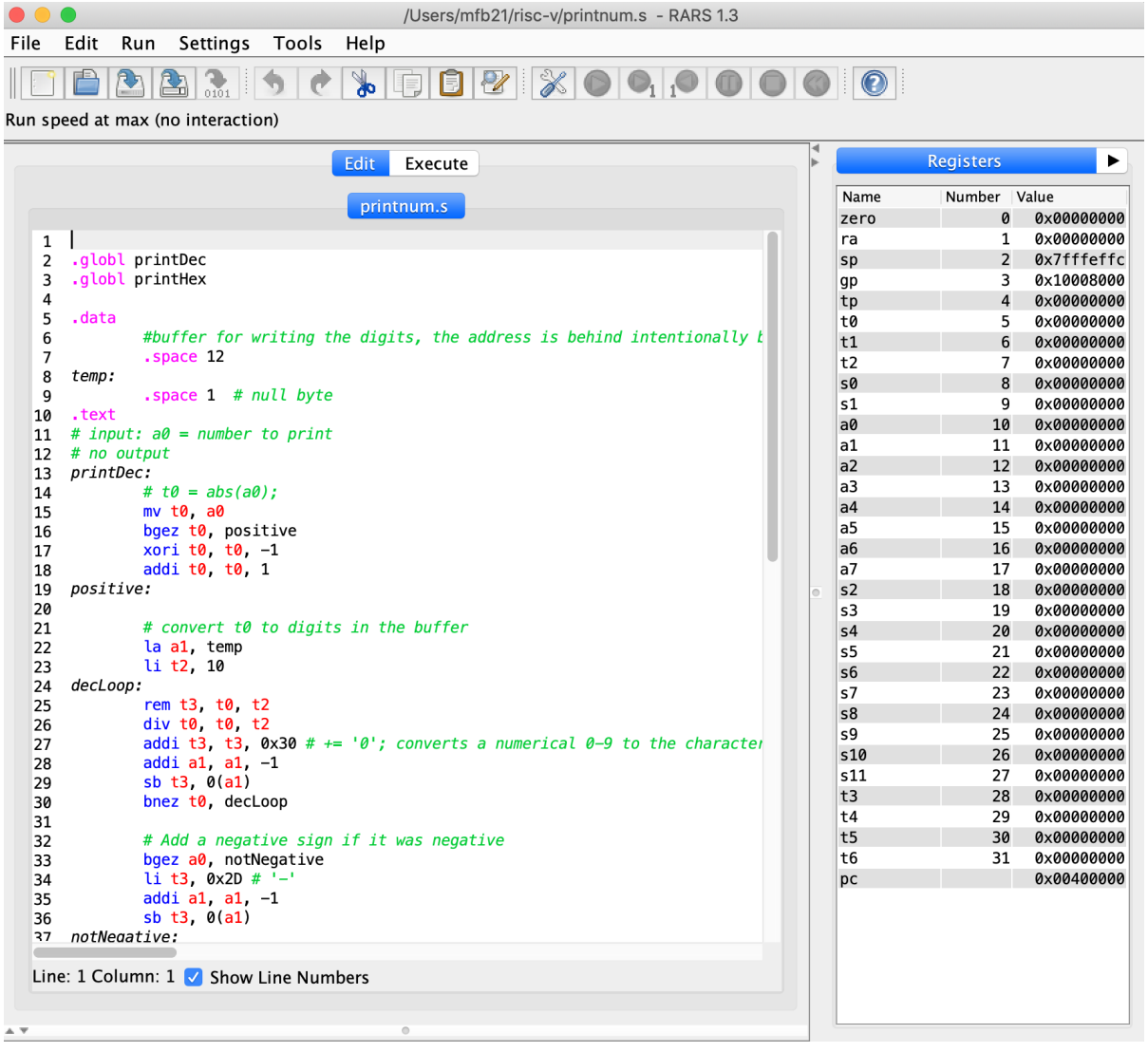
In its basic from, RISC-V has 32 integer registers.

RISC-V is a **load-and-store** architecture, meaning:

Except for memory access instructions, instructions address only registers.

# RARS

I recommend using a RISC-V simulator like RARS for learning RISC-V.



# Getting RARS

Homepage: `https://github.com/TheThirdOne/rars`

# Getting RARS

Homepage: `https://github.com/TheThirdOne/rars`

Download *.jar at `https://github.com/TheThirdOne/rars/releases/download/v1.3.1/rars1_3_1.jar`

# Getting RARS

Homepage: `https://github.com/TheThirdOne/rars`

Download *.jar at `https://github.com/TheThirdOne/rars/releases/download/v1.3.1/rars1_3_1.jar`

Easy to launch: `java -jar rars1_3_1.jar`

# Getting RARS

Homepage: `https://github.com/TheThirdOne/rars`

Download *.jar at `https://github.com/TheThirdOne/rars/releases/download/v1.3.1/rars1_3_1.jar`

Easy to launch: `java -jar rars1_3_1.jar`

Has useful online help.

# RARS

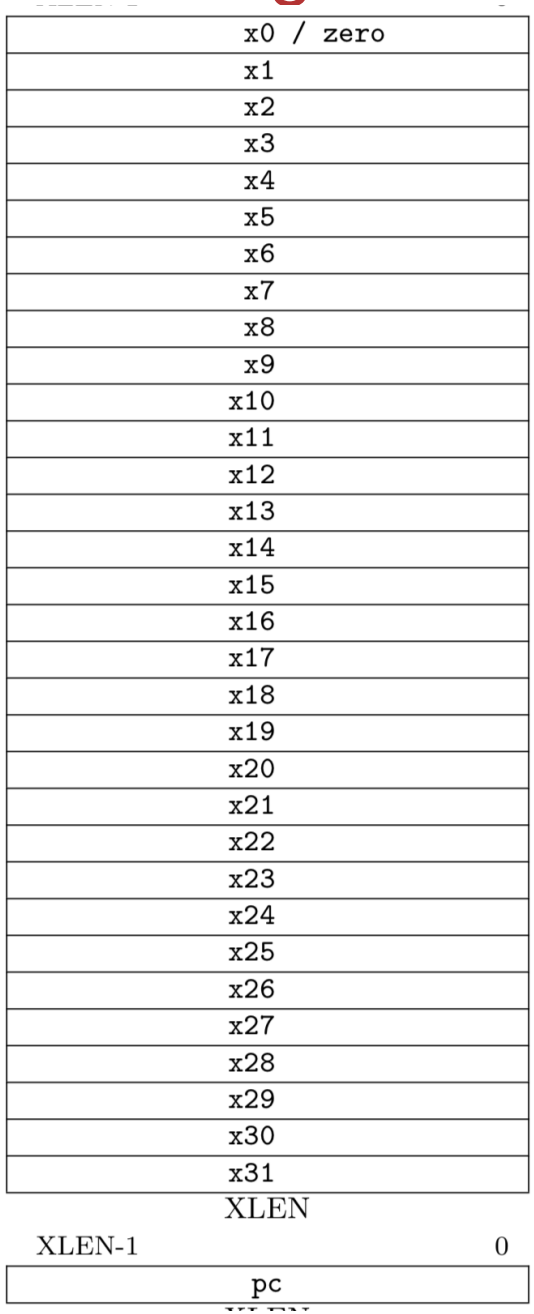
You need to learn RARS by yourself and in tutorials.

# RISC-V

Here is a basic overview of RISC-V. We'll only cover issues that are relevant for code generation. You are expected to familiarise yourself with RISC-V programming on your own.

This should not be difficult with RARS, as RISC-V is an exceptionally clean architecture.

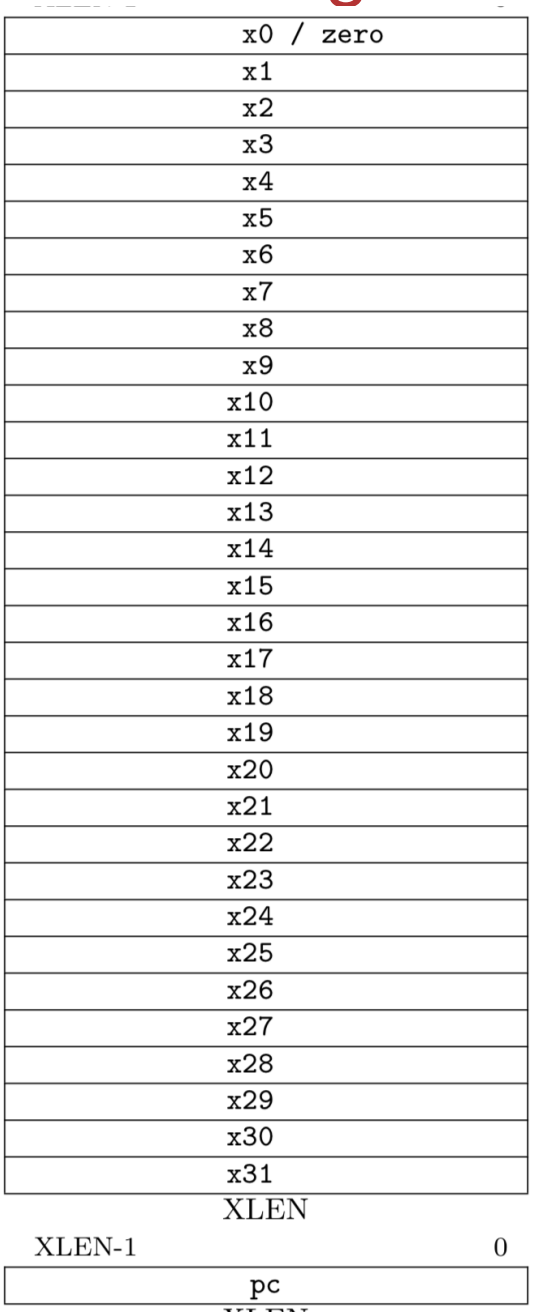
# RISC-V registers




RISC-V has the following registers (all are 32 bits).

- 32 general purpose registers
- A program counter (PC)

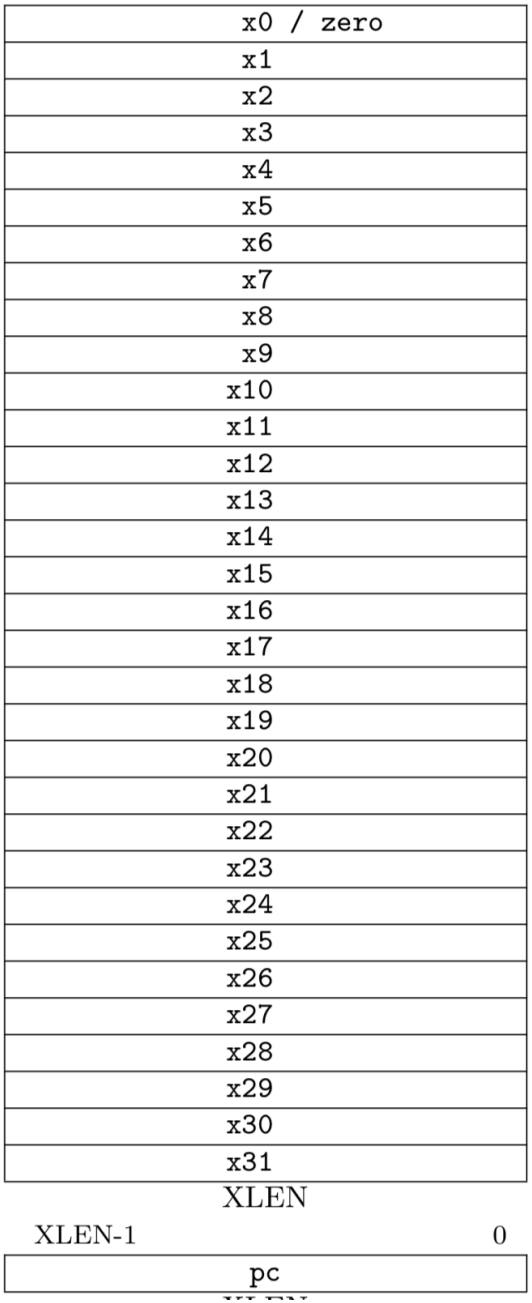
# RISC-V registers




CPU general-purpose registers have assigned functions:

- x0 is hard-wired to 0, and can be used as target register for any instruction whose result is to be discarded. x0 can also be used as a source of 0 if needed.
- x1-x31 are general purpose registers. The 32 bit integers they hold are interpreted, depending on the instruction that access the registers. (Examples: Boolean values, two's complement signed binary integers or unsigned binary integers, stack pointer or return address).

# RISC-V registers: PC




The program counter (PC) register, points to the instruction to be executed next. The PC cannot directly be written or read using load/store instructions. It can only be influenced by executing instructions which change the PC as a side-effect.

# RISC-V registers: do you notice something?

| x0 / zero |
|---|
| x1 |
| x2 |
| x3 |
| x4 |
| x5 |
| x6 |
| x7 |
| x8 |
| x9 |
| x10 |
| x11 |
| x12 |
| x13 |
| x14 |
| x15 |
| x16 |
| x17 |
| x18 |
| x19 |
| x20 |
| x21 |
| x22 |
| x23 |
| x24 |
| x25 |
| x26 |
| x27 |
| x28 |
| x29 |
| x30 |
| x31 |

XLEN

XLEN-1 0

| pc |
|---|

# RISC-V registers: do you notice something?

| x0 / zero |
|---|
| x1 |
| x2 |
| x3 |
| x4 |
| x5 |
| x6 |
| x7 |
| x8 |
| x9 |
| x10 |
| x11 |
| x12 |
| x13 |
| x14 |
| x15 |
| x16 |
| x17 |
| x18 |
| x19 |
| x20 |
| x21 |
| x22 |
| x23 |
| x24 |
| x25 |
| x26 |
| x27 |
| x28 |
| x29 |
| x30 |
| x31 |

XLEN

XLEN-1 0

| pc |
|---|

No explicit SP (and no `push`, no `pop` commands)

# RISC-V registers: do you notice something?

| x0 / zero |
|---|
| x1 |
| x2 |
| x3 |
| x4 |
| x5 |
| x6 |
| x7 |
| x8 |
| x9 |
| x10 |
| x11 |
| x12 |
| x13 |
| x14 |
| x15 |
| x16 |
| x17 |
| x18 |
| x19 |
| x20 |
| x21 |
| x22 |
| x23 |
| x24 |
| x25 |
| x26 |
| x27 |
| x28 |
| x29 |
| x30 |
| x31 |

XLEN

XLEN-1 0

| pc |
|---|

No explicit SP (and no `push`, no `pop` commands)

# RISC-V registers: do you notice something?

| x0 / zero |
|---|
| x1 |
| x2 |
| x3 |
| x4 |
| x5 |
| x6 |
| x7 |
| x8 |
| x9 |
| x10 |
| x11 |
| x12 |
| x13 |
| x14 |
| x15 |
| x16 |
| x17 |
| x18 |
| x19 |
| x20 |
| x21 |
| x22 |
| x23 |
| x24 |
| x25 |
| x26 |
| x27 |
| x28 |
| x29 |
| x30 |
| x31 |

XLEN

XLEN-1 0

| pc |
|---|

No explicit SP (and no `push`, no `pop` commands)


Can be simulated with general purpose register and normal commands.

# 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 x1 (ra), x2 (sp), x8 (s0)

# 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 x1 (ra), x2 (sp), x8 (s0)

Note that those are **usage conventions**. I recommend adhering to them if you want to interface with other RISC-V software (e.g. assembler).

# RISC-V Datatypes

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

# RISC-V Datatypes

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 Datatypes

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.

Memory is byte-addressable.

# RISC-V Datatypes

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.

Memory is byte-addressable.

A word of memory is defined as 32 bits (4 bytes).

# RISC-V Datatypes

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.

Memory is byte-addressable.

A word of memory is defined as 32 bits (4 bytes).

A halfword is 16 bits (2 bytes).

# RISC-V Datatypes

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.

Memory is byte-addressable.

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 alignment

RISC-V has fixed-length 32-bit instructions that **must** be aligned on 32-bit boundaries (i.e. at memory locations divisible by 4).

# RISC-V memory alignment

RISC-V has fixed-length 32-bit instructions that **must** be aligned on 32-bit boundaries (i.e. at memory locations divisible by 4).

**But** ...

# RISC-V memory alignment

RISC-V has fixed-length 32-bit instructions that **must** be aligned on 32-bit boundaries (i.e. at memory locations divisible by 4).

**But** ... Accessed memory addresses **need not** be aligned, **but accesses to aligned addresses may be faster**; for example, simple CPUs may implement unaligned accesses with slow software emulation driven from an alignment failure interrupt.

# RISC-V memory alignment

RISC-V has fixed-length 32-bit instructions that **must** be aligned on 32-bit boundaries (i.e. at memory locations divisible by 4).

**But** ... Accessed memory addresses **need not** be aligned, **but accesses to aligned addresses may be faster**; for example, simple CPUs may implement unaligned accesses with slow software emulation driven from an alignment failure interrupt.

The assembler will help you with alignment. We come back to this.

# RISC-V instructions

CPU instructions are organized into the following functional groups:

- Load and store (memory access)
- Immediates (handling of constants)
- Computational (e.g. integer arithmetic and boolean logic)
- Jump and branch (conditional and unconditional)
- Many others (e.g. SIMD, vectoring)

Each instruction is 32 bits long in memory.

# RISC-V instructions

CPU instructions are organized into the following functional groups:

- Load and store (memory access)
- Immediates (handling of constants)
- Computational (e.g. integer arithmetic and boolean logic)
- Jump and branch (conditional and unconditional)
- Many others (e.g. SIMD, vectoring)

Each instruction is 32 bits long in memory.

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

Main memory is accessed **only** through load and store instructions.

# RISC-V instructions

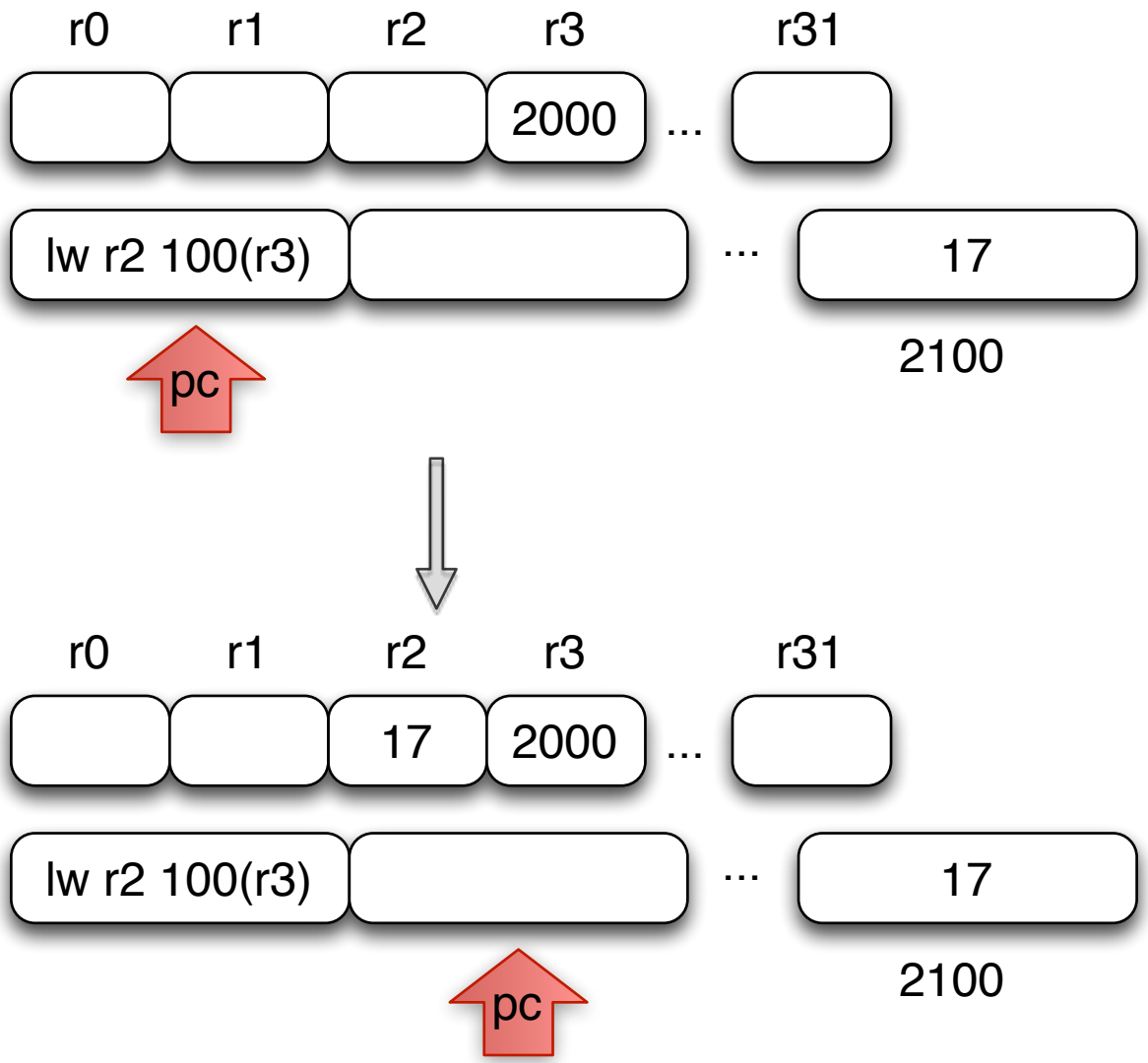
The command

```
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

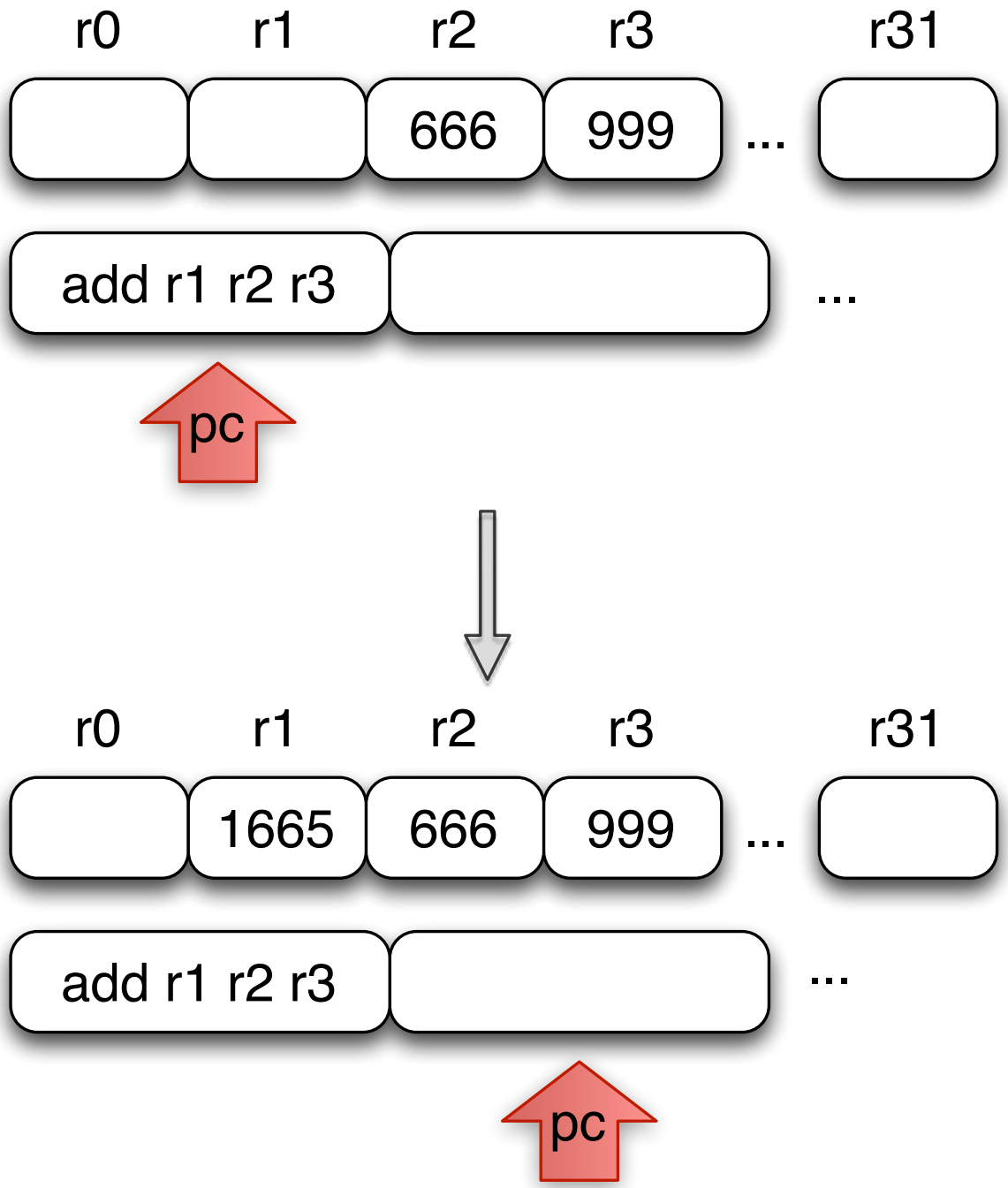
The command

```
add reg1 reg2 reg3
```

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

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

# RISC-V instructions



# RISC-V instructions

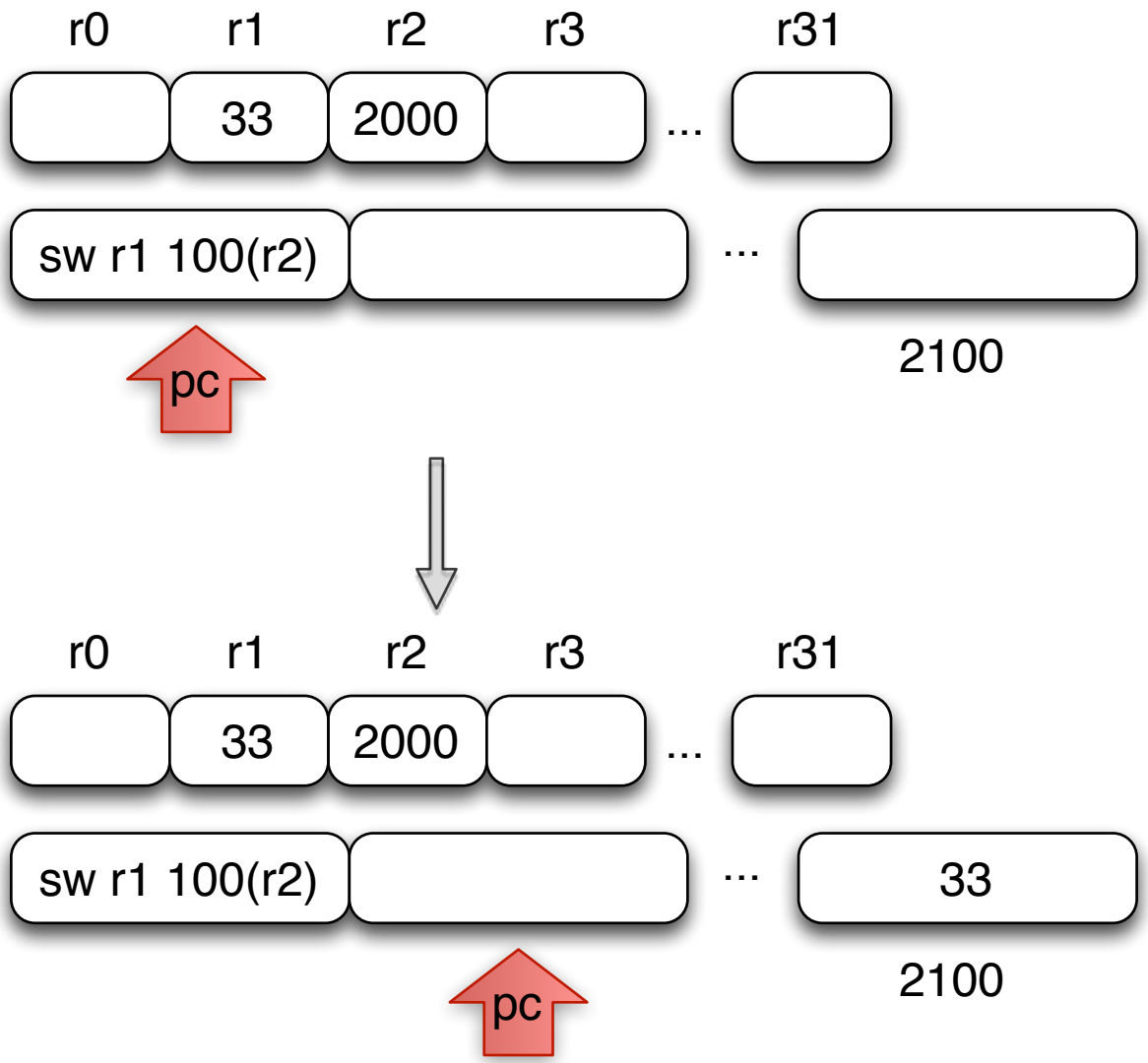
The command

```
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 command

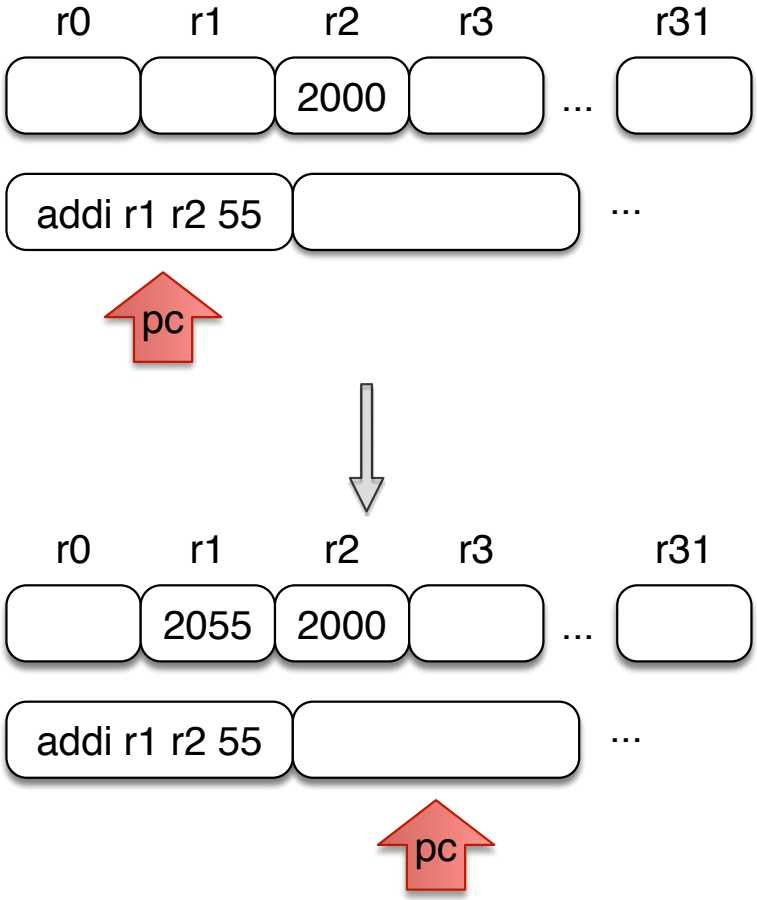
```
addi reg1 reg2 imm
```

Adds the 16 bit **signed** integer `imm` to the word currently in $\texttt{reg}_2$, storing the result in register $\texttt{reg}_1$. Here the 'u' in `addi`**u** means **unsigned**. In first approximation that means overflow is not checked when adding (no error is caused by overflowing).

Not checking overflow is useful e.g. when you want 'wrap around' a sum at 0 or $2^{32} - 1$. You want this e.g. when doing cryptography. In addition we consider e.g. the SP an unsigned integer.

But `imm` is signed, so we can increment and decrement e.g. the SP.

# RISC-V instructions



# RISC-V instructions

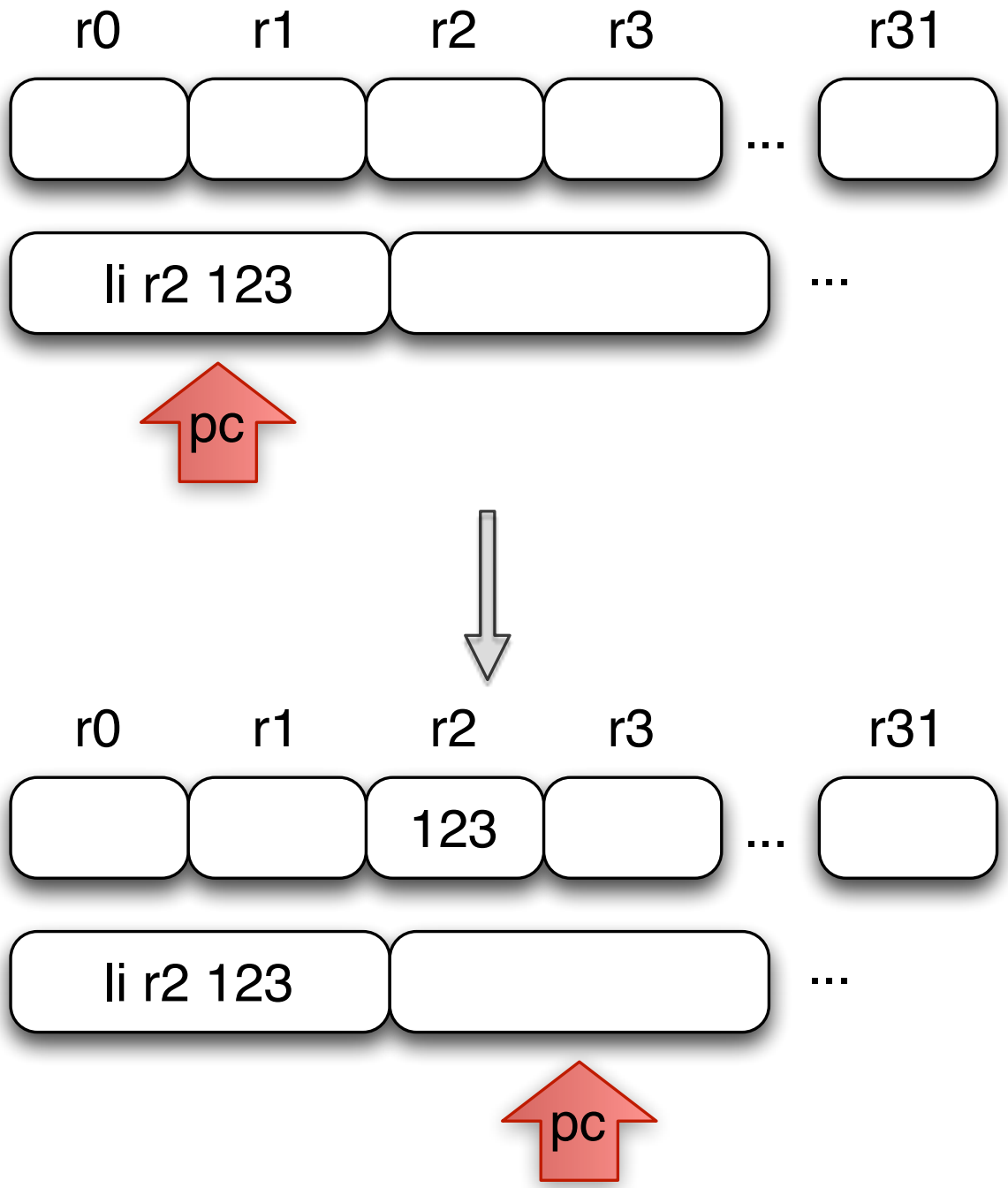
The **pseudo instruction**

```
li reg imm
```

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

It is a pseudo instruction in that there is no RISC-V assembly command that directly implements this (MPIS cannot load 32 bit words directly), instead the RISC-V assembler will automatically expand `li reg imm` into a sequence of real assembler commands. When compiling you can easily treat pseudo instructions as real instructions.

# RISC-V instructions



# Our first RISC-V program

Let's write the program 7+5, we want the result in register `r5`.

```
li r6 7
li r5 5
add r5 r5 r6
```

# Our second RISC-V program

Let's write 7+5, in accumulator machine form.

- One argument is in the accumulator.
- Remaining arguments on the stack.
- Result should be in accumulator.

# Our second RISC-V program

Let's write 7+5, in accumulator machine form.

- One argument is in the accumulator.
- Remaining arguments on the stack.
- Result should be in accumulator.

Recall RISC-V doesn't have an explicit SP.

Also: no explicit accumulator!

# Our second RISC-V program

Let's write 7+5, in accumulator machine form.

- One argument is in the accumulator.
- Remaining arguments on the stack.
- Result should be in accumulator.

Recall RISC-V doesn't have an explicit SP.

Also: no explicit accumulator!

We must simulate both, and that is easy: each register can be used as SP or as accumulator.

# Our second RISC-V program

Let's write 7+5, in accumulator machine form.

# Our second RISC-V program

Let's write 7+5, in accumulator machine form.

Convention: we use register `x2` as stack pointer, and register `x10` as accumulator.

It is customary in RISC-V assembly to write `sp` for the stack pointer (`r29`) and `a0` for register `r4`.

# Our second RISC-V program

Recall that in the accumulator machine model, memory operations work only via the accumulator.

# Our second RISC-V program

Recall that in the accumulator machine model, memory operations work only via the accumulator. With this in mind, here is the program 7+5 we are seeking to translate to RISC-V in pseudo-code.

```
acc <- 7
push acc
acc <- 5
acc <- acc + top of stack
pop
```

# Our second RISC-V program

To translate

```
acc <- 7
push acc
acc <- 5
acc <- acc + top of stack
pop
```

into RISC-V we adhere to the conventions that

# Our second RISC-V program

To translate

```
acc <- 7
push acc
acc <- 5
acc <- acc + top of stack
pop
```

into RISC-V we adhere to the conventions that

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

# Our second RISC-V program

To translate

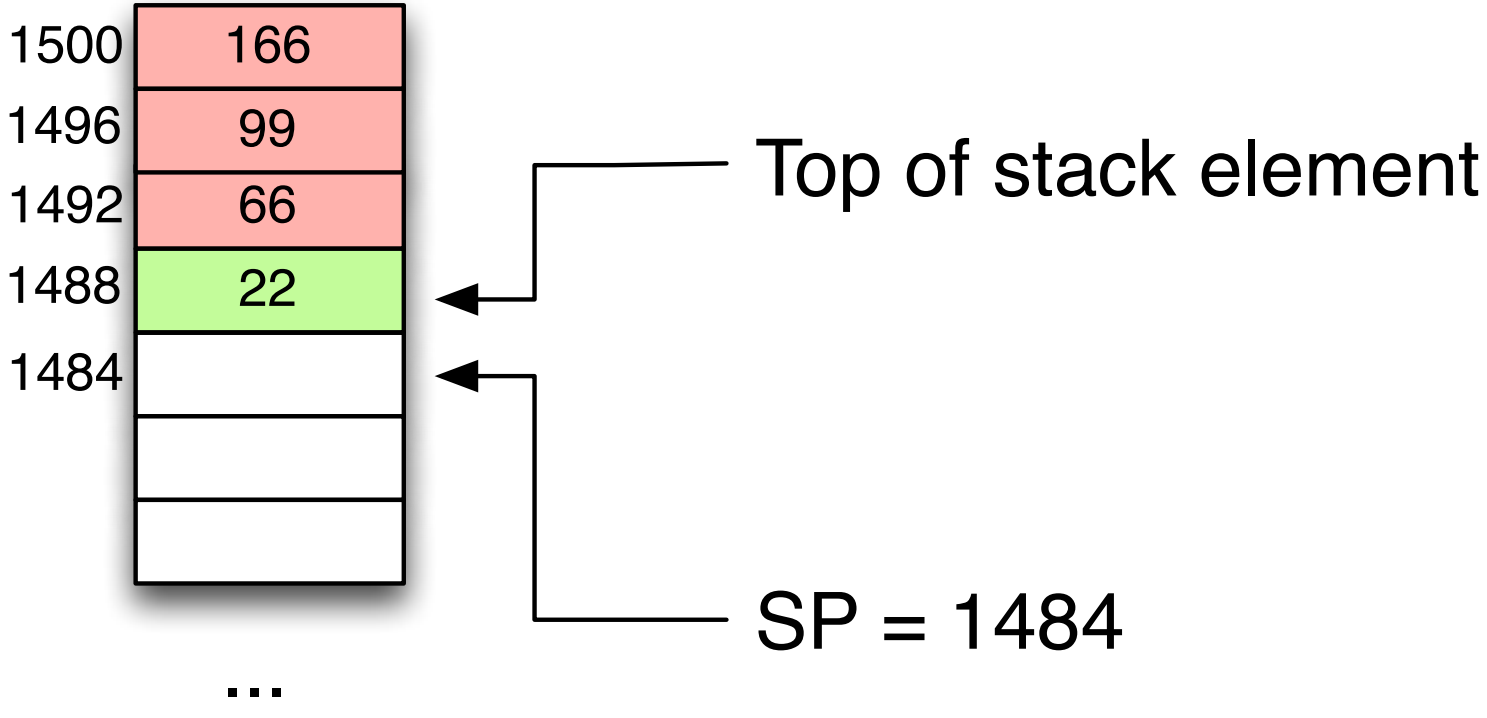
```
acc <- 7
push acc
acc <- 5
acc <- acc + top of stack
pop
```

into RISC-V we adhere to the conventions that

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

# Our second RISC-V program

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



# Our second RISC-V program

```
acc <- 7
push acc
acc <- 5
acc <- acc+topOfStack
pop
```

# Our second RISC-V program

```
                                li a0 7
acc <- 7                        sw a0 0(sp)
push acc                        addi sp sp -4
acc <- 5                        li a0 5
acc <- acc+topOfStack           lw t1 4(sp)
pop                             add a0 a0 t1
                                addi sp sp 4
```

# Our second RISC-V program

```
                                li a0 7
acc <- 7                        sw a0 0(sp)
push acc                        addi sp sp -4
acc <- 5                        li a0 5
acc <- acc+topOfStack           lw t1 4(sp)
pop                             add a0 a0 t1
                                addi sp sp 4
```

Note that the program on the right is really doing almost exactly what we did a few weeks ago when we looked at the accumulator machine, except that

# Our second RISC-V program

```
                                li a0 7
acc <- 7                        sw a0 0(sp)
push acc                        addi sp sp -4
acc <- 5                        li a0 5
acc <- acc+topOfStack           lw t1 4(sp)
pop                             add a0 a0 t1
                                addi sp sp 4
```

Note that the program on the right is really doing almost exactly what we did a few weeks ago when we looked at the accumulator machine, except that

- we use a temporary `t1`

# Our second RISC-V program

```
acc <- 7                        li a0 7
push acc                        sw a0 0(sp)
acc <- 5                        addi sp sp -4
acc <- acc+topOfStack           li a0 5
pop                             lw t1 4(sp)
                                add a0 a0 t1
                                addi sp sp 4
```

Note that the program on the right is really doing almost exactly what we did a few weeks ago when we looked at the accumulator machine, except that

- we use a temporary `t1`
- we use RISC-V assembly

# Our second RISC-V program

```
acc <- 7                        li a0 7
push acc                        sw a0 0(sp)
acc <- 5                        addi sp sp -4
acc <- acc+topOfStack           li a0 5
pop                             lw t1 4(sp)
                                add a0 a0 t1
                                addi sp sp 4
```

Note that the program on the right is really doing almost exactly what we did a few weeks ago when we looked at the accumulator machine, except that

- we use a temporary `t1`
- we use RISC-V assembly
- we have to adjust the stack 'by hand', rather than using built-in `push` and `pop`

# 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.

RISC-V machine code is really straightforward, and not really different from the pseudo machine code we used a few weeks back, except that the assembler syntax is slightly different.

# Interlude on (RISC-V) assembler

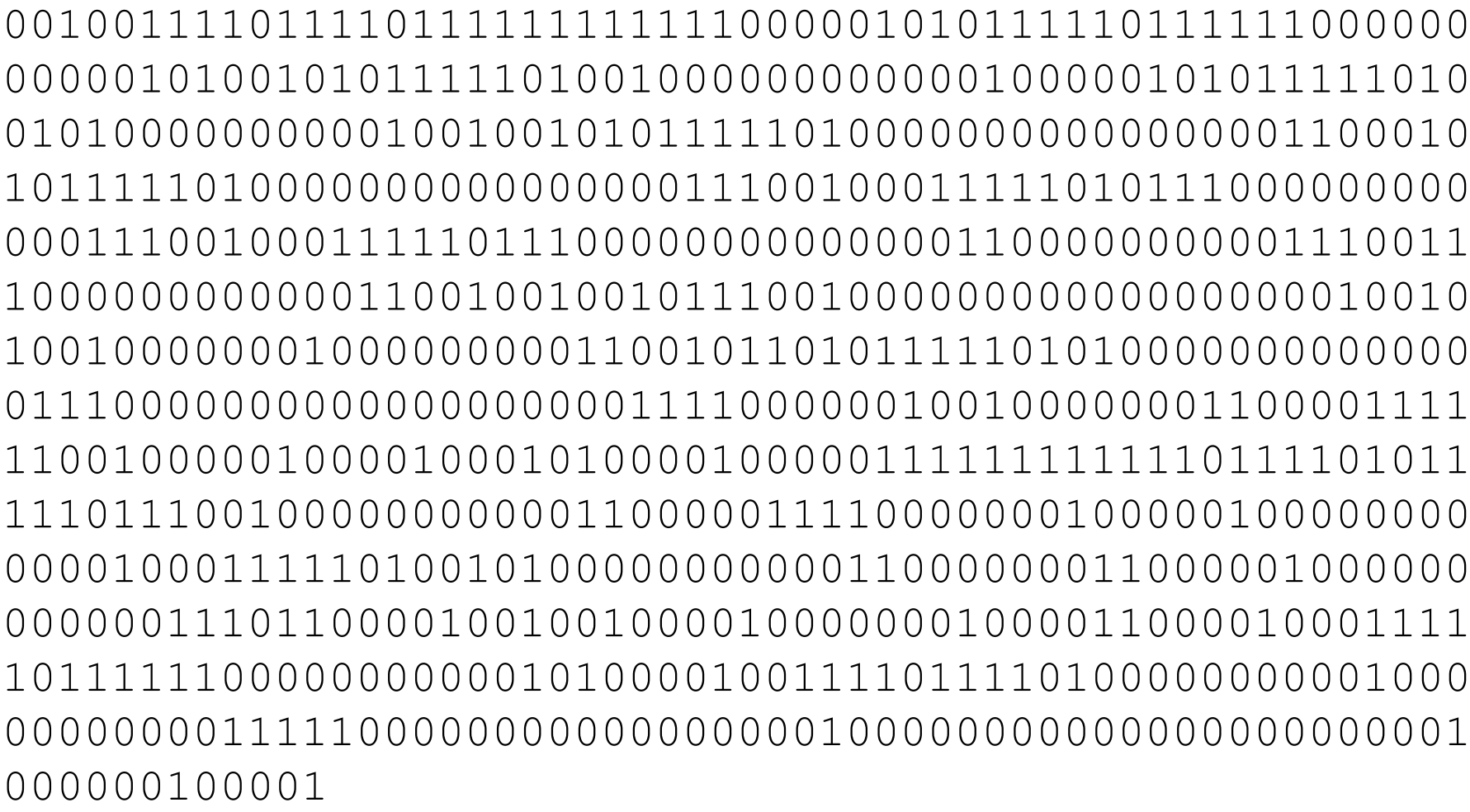
Assembler language is a programming language that is close to machine language but not the same.

# Interlude on (RISC-V) assembler

Assembler language is a programming language that is close to machine language but not the same.

Why bother with yet another language? Why not program straight in machine language?

# That's why



# That's why

Here is same code written in assembly language, but no symbolic labels are used as name of registers or memory locations.

```
addi $29, $29, -32
sw $31, 20($29)
sw $4,  32($29)
sw $5, 36($29)
sw $0,  24($29)
sw $0, 28($29)
lw $14, 28($29)
lw $24, 24($29)
multu $14, $14
addi $8, $14, 1
slti $1, $8, 101
sw $8,  28($29)
```

```
mflo $15
addu $25, $24, $15
bne $1, $0, -9
sw $25, 24($29)
lui $4, 4096
lw $5, 24($29)
jal 1048812
addi $4, $4, 1072
lw $31, 20($29)
addi $29, $29, 32
jr $31
move $2,  $0
```

# That's why

It gets even better with **symbolic** names such as `sp` or `loop`.

```
        .text
        .align 2
        .globl main
main:
        subu sp, sp, 32
        sw ra, 20(sp)
        sd a0, 32(sp)
        sw 0, 24(sp)
        sw 0,  28(sp)
loop:
        lw t6, 28(sp)
        mul t7, t6, t6
        lw t8, 24(sp)
        addu t9, t8, t7
        sw t9, 24(sp)
        addu t0, t6, 1
        sw t0, 28(sp)
        ble t0, 100, loop
        la a0, str
        lw a1, 24(sp)
        jal printf
        move v0, 0
        lw ra, 20(sp)
        addu sp, sp, 32
        jr ra
        .data
        .align 0
str:
        .asciz "The sum from
                  0 .. 100 is %d
```

# Assembler vs assembly language

We must carefully distinguish between

- **Assembly language**, the symbolic representation of a computer's binary machine language.
- **Assembler**, a program (a mini-compiler) that translates assembly language into real machine code (long sequences of 0s and 1s).

# Assembler, the program

The assembler primarily does two things.

- Translate commands in assembly language like `addi t3 t6 t8` into machine code.
- Convert symbolic addresses such as `main` or `loop` into machine addresses such as `10001101001101001101001101010101001`. This task is sometimes deferred to the linker.

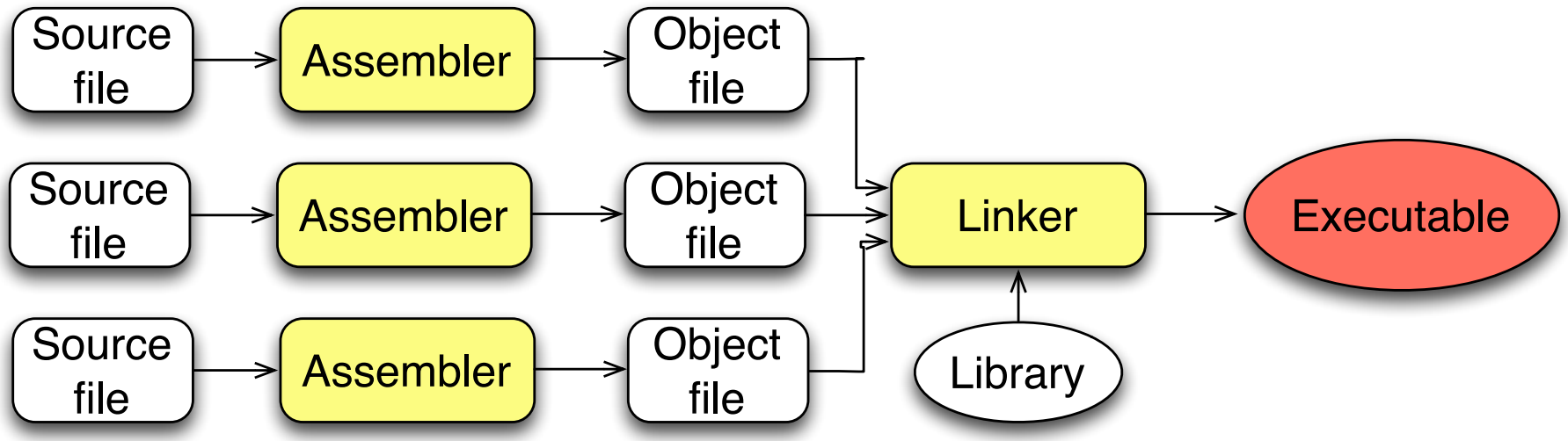
# Assembler, the program

The assembler primarily does two things.

- Translate commands in assembly language like `addi t3 t6 t8` into machine code.
- Convert symbolic addresses such as `main` or `loop` into machine addresses such as `10001101001101001101001101010101001`. This task is sometimes deferred to the linker.

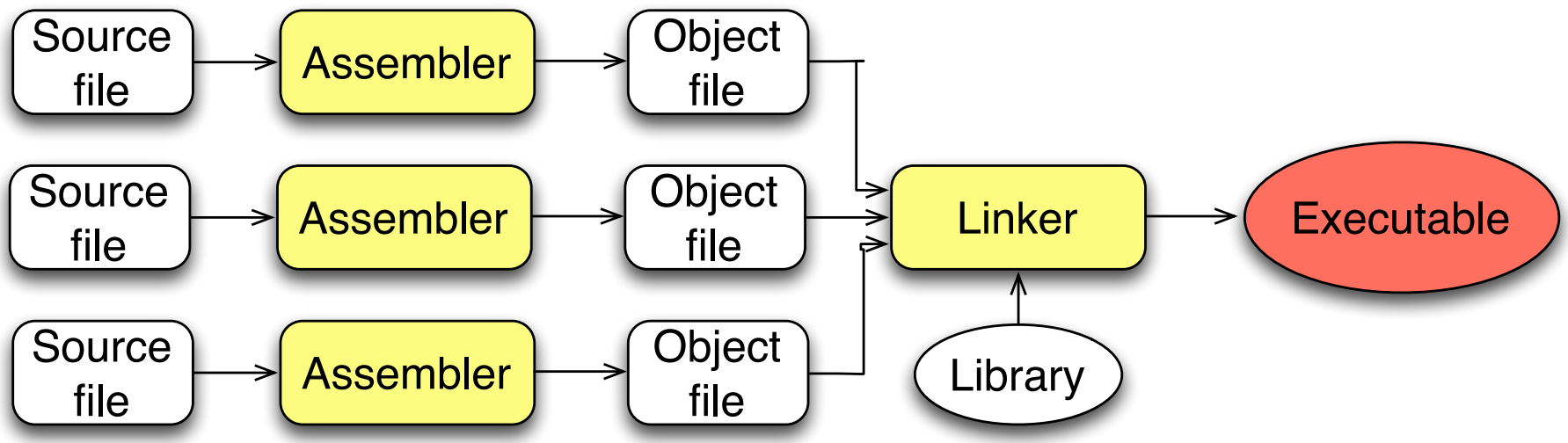
The symbolic addresses in assembly language name commonly occurring bit patterns, such as opcodes and register names, so humans can read and remember them. In addition, assembly language permits programmers to use labels to identify and name particular memory words that hold instructions or data, or that the program can jump to.

# Assembler, the program



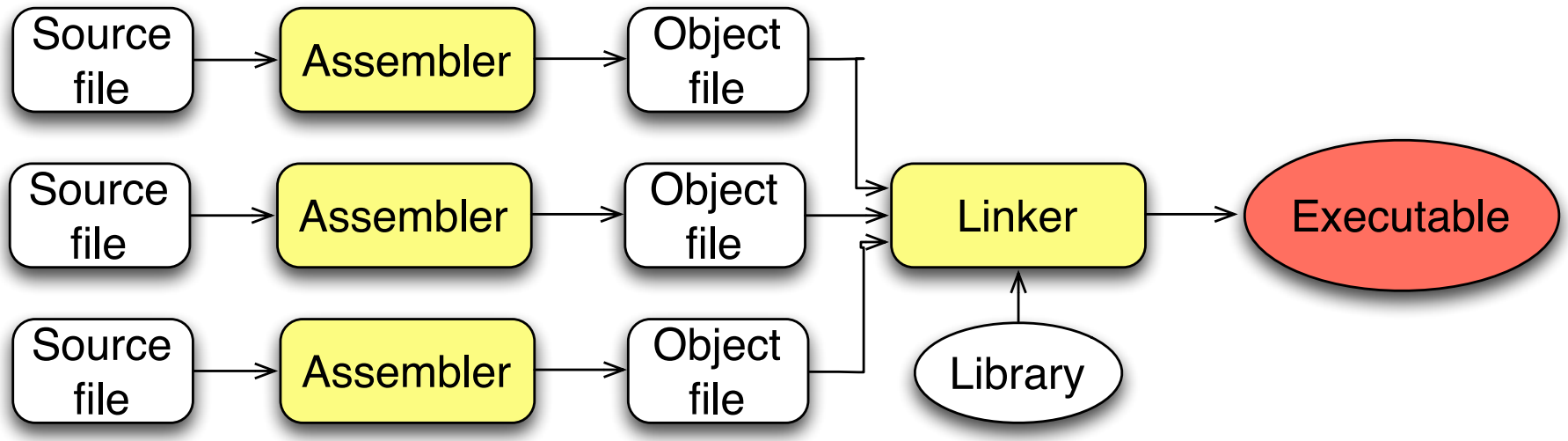

Source files are produced by a compiler. They may contain labels that are not defined in the source file, reference to external code (e.g. print).

# Assembler, the program




Assembler translates source files to object files, which are machine code, but contains 'holes' (basically references to external code). Because of holes, object files cannot be executed directly. The holes arise because the assembler translates each file separately.

# Assembler, the program




The linker gets all object files and libraries and puts the right addresses into holes, yielding an executable.

# Assembler, the program

Here is an example of using names: `main` is a **global** name in the sense that other programs can use it. OTOH `loop` is a **local** name: it can only be used (jumped to) inside this program.

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

It is the declaration (assembler directive) `.globl main` that makes `main` global.

# Assembler, the program

The assembler processes a source file line by line, translating assembly commands. It keeps track of the size of each command.

```
loop:
     subu 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 command 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

# 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
```

The `.asciz` directive is easier to read for text strings.

# Pseudo instructions

You remember

```
li reg imm
```

?

# Pseudo instructions

You remember

```
li reg imm?
```

Turns out that `li` is not a RISC-V assembly command.

# Pseudo instructions

You remember

```
li reg imm?
```

Turns out that `li` is not a RISC-V assembly command.

RISC-V cannot load a 32 bit word in one instruction. Two instructions are needed (one for the lower 16 bits of `imm`, and another one for the upper 16 bits.)

# Pseudo instructions

You remember

```
li reg imm?
```

Turns out that `li` is not a RISC-V assembly command.

RISC-V cannot load a 32 bit word in one instruction. Two instructions are needed (one for the lower 16 bits of `imm`, and another one for the upper 16 bits.)

Instead it is a pseudo command: the assembler replaces each occurrency of `li` appropriately.

# Pseudo instructions

You remember

```
li reg imm?
```

Turns out that `li` is not a RISC-V assembly command.

RISC-V cannot load a 32 bit word in one instruction. Two instructions are needed (one for the lower 16 bits of `imm`, and another one for the upper 16 bits.)

Instead it is a pseudo command: the assembler replaces each occurrency of `li` appropriately.

The RARS simulator shows pseudo instructions and the instructions that the former translate to together.