Arquitectura de Computadores

curso 2022-2023

Intro RARS and RISC-V assembler

3º de grado en Ingeniería Informática y

3º de doble grado en Ing. Informática y Matemáticas

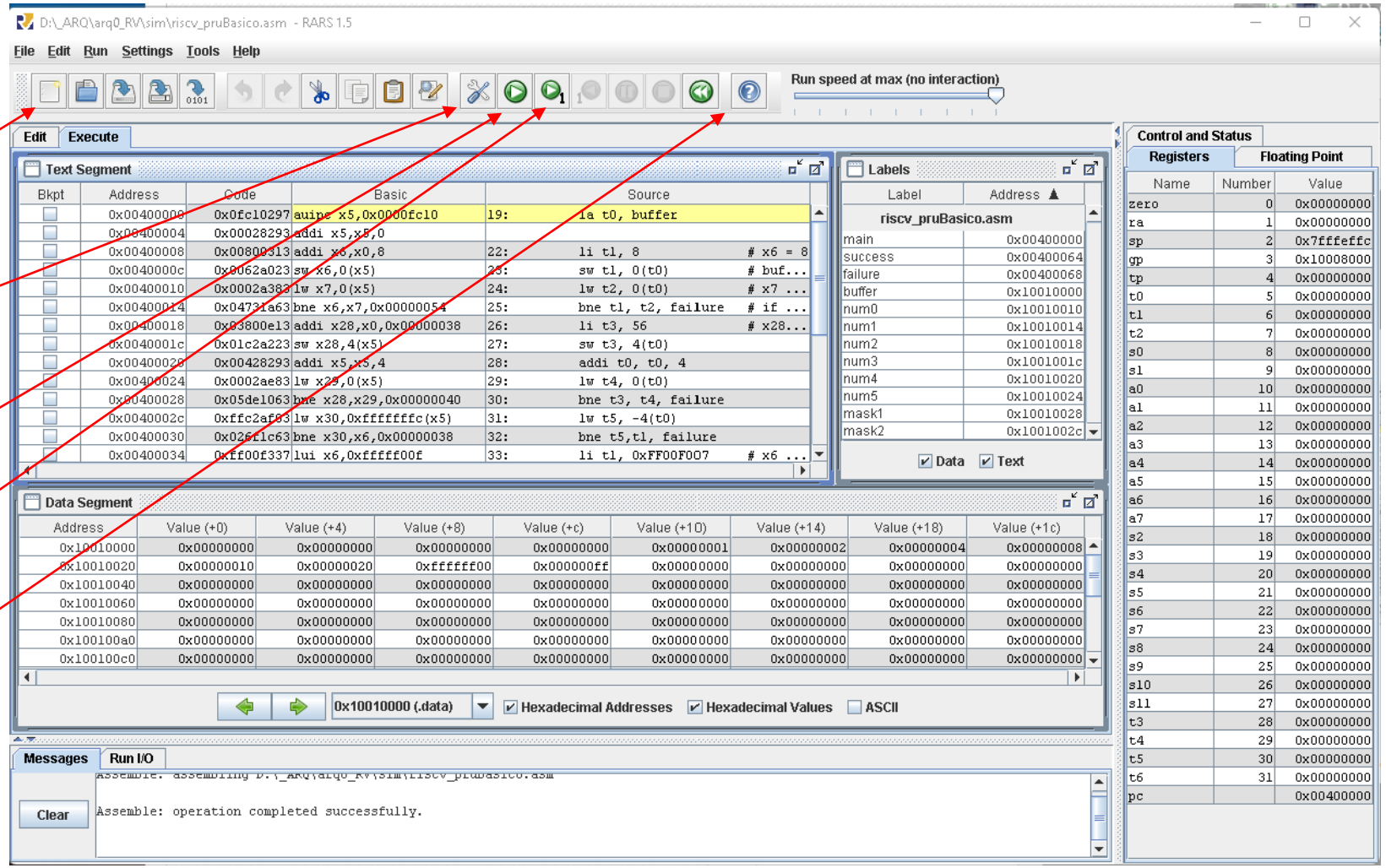
RARS Introduction

- RARS (**R**ISC-V **A**ssembler and **R**untime **S**imulator)
  - Designed to execute for RISC-V assembly language programs
  - supports RISC-V IMFDN ISA base (riscv32 & riscv64).
  - supports debugging using breakpoints and/or ebreak.
  - supports side by side comparison from pseudo-instruction to machine code with intermediate steps.
- You need Java environment to run RARS
- In Moodle, downloaded from: https://github.com/TheThirdOne/rars/
  - Execute the command to start RARS: java -jar <rars jar path>
  - If you associate jar with JAVA in Linux or Windows, you can double click

RARS basic commands

Basic Commands:

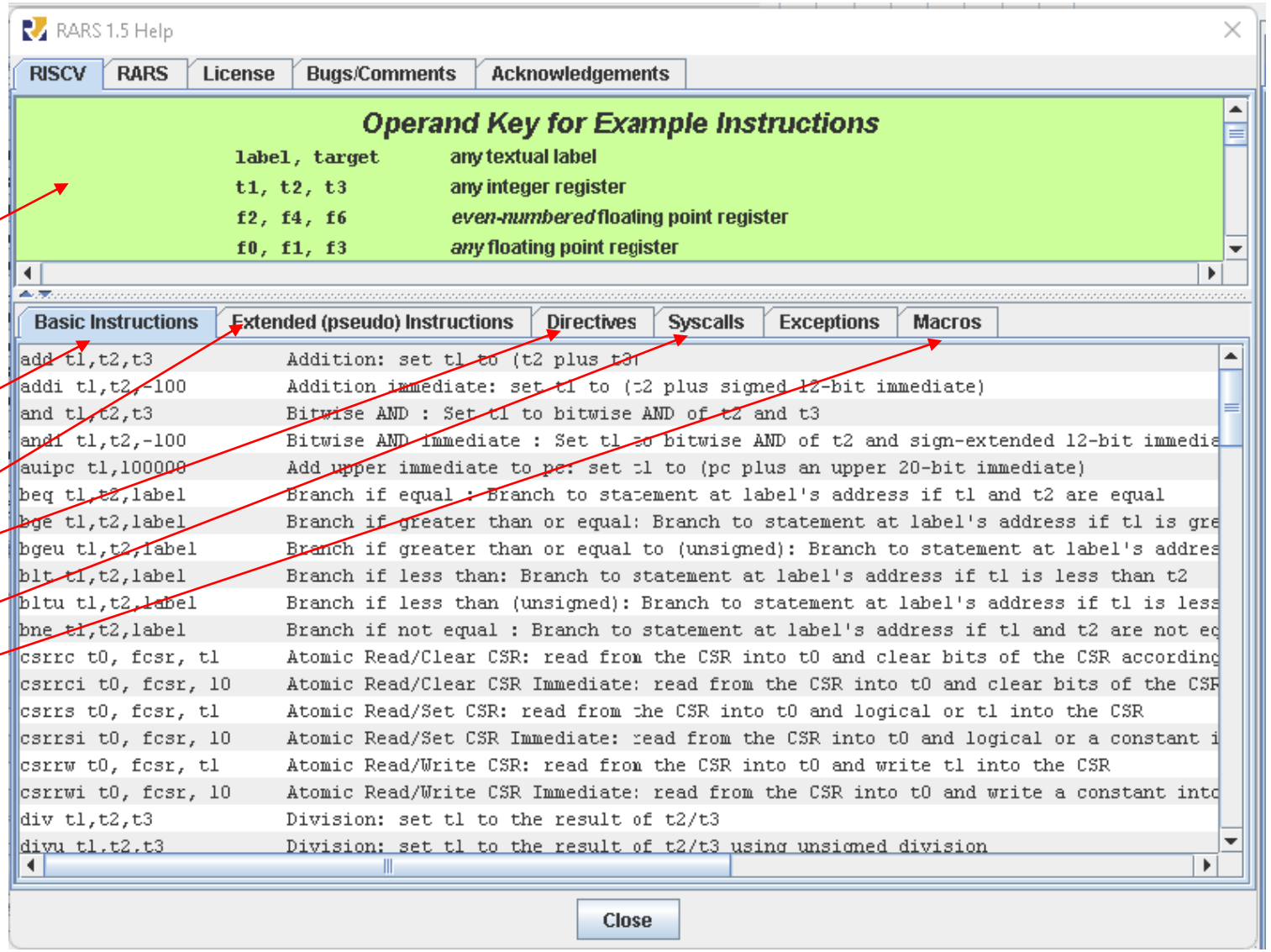
- Create a new source file (Ctrl + N)
- Assemble the source code (F3)
- Execute the current source code (F5)
- Step running (F7)
- Instructions & System call (Help - F1)



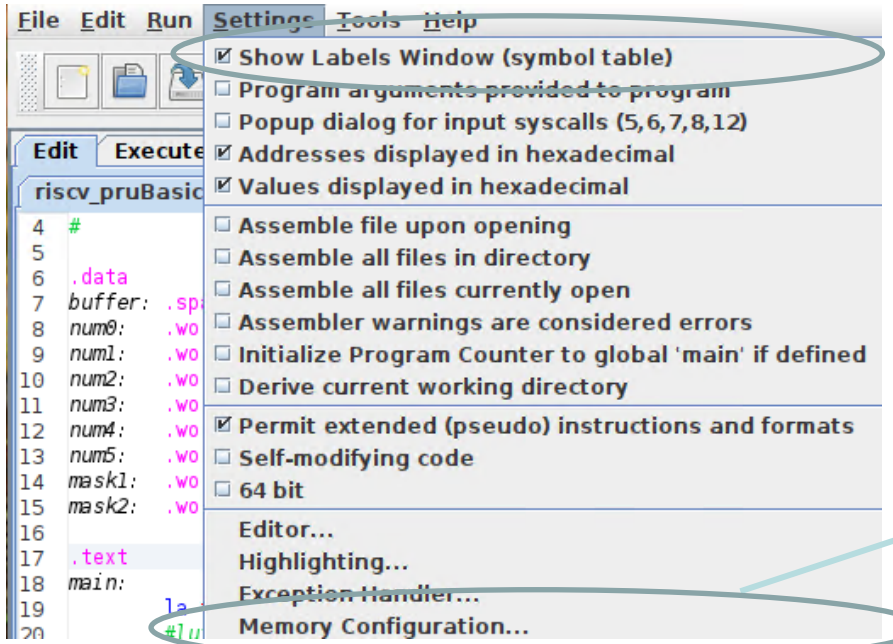
RARS Quick Reference

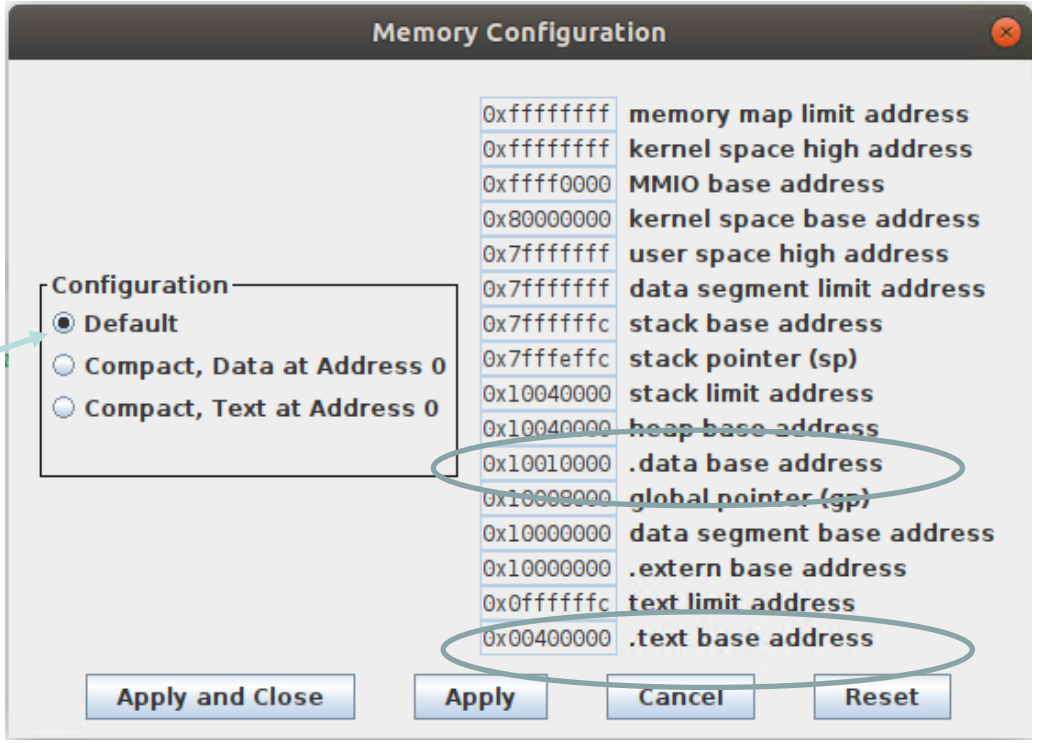
- You can display RISC V Instructions & System call (F1)

- Basic instructions
- Pseudo instructions
- Directives
- System Calls
- Macros



RARS for assembly and debugging





- In settings select show labels
- Leave *Memory Configuration* as *Default*:
  This means data (*.data*) in address *0x1001_0000* and code (*.text*) in address *0x0040_0000*

RISC-V Registers

- We can manipulate 32 general purpose registers in assembly programming directly
- In assembler, we prefer using aliases to indicate registers

Register Names	ABI Names	Description
x0	zero	Hard-wired zero
x1	ra	Return address
x2	sp	Stack pointer
x3	gp	Global pointer
x4	tp	Thread pointer
x5	t0	Temporary / Alternate link register
x6-x7	t1 - t2	Temporary register
x8	s0 / fp	Saved register / Frame pointer
x9	s1	Saved register
x10-x11	a0-a1	Function argument / Return value registers
x12-x17	a2-a7	Function argument registers
x18-x27	s2-s11	Saved registers
x28-x31	t3-t6	Temporary registers

RISC-V Instructions (reminder)

- A typical RISC-V instruction is on this format: **`OPCODE rd, rs1, rs2`**
- In programming terms, we could think as: **`rd = f(rs1, rs2)`**
  - The opcode specifies some function ***f*** to perform on the two source registers **rs1** and **rs2** and produce a result which is stored in the destination register rd.

- Example: For the ADD and SUB instruction this is often written as:

```
ADDI x2, x0, 1      # x2 ← x0 + 1
BLT x0, x1, loop    # IF x0 < x1 GOTO loop
```

* RARS interpreter uses hash (#) symbols for comments.

BLT stands for Branch if Lower Than

RISC-V Instructions (reminder)

- Why Three Operands? In assembly code instructions are encoded in a fixed width format (32-bits in RISC-V). To make it easier for the decoder inside the processor to figure out what an instruction does, it helps to have different parts of the instruction being in fixed locations.

32-bit RISC-V Instruction Formats

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

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

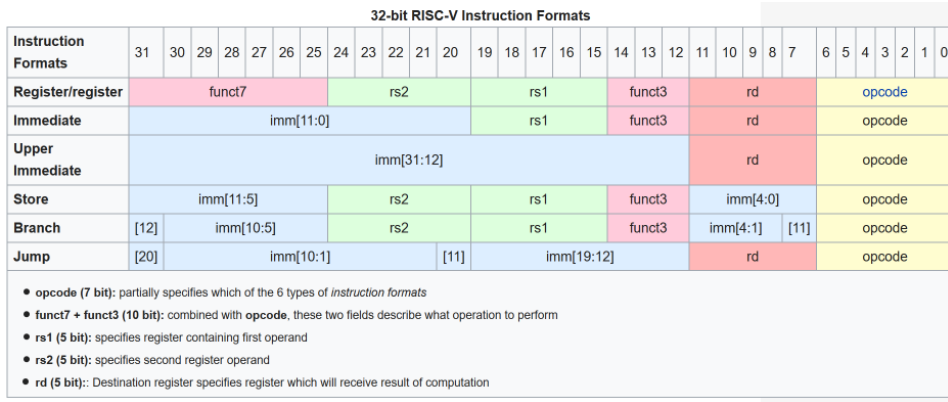
RISC-V Instructions (reminder)

- The diagram shows what the different bits are used for in 32-bit encoding. You can observe the regularity:
  - Opcode at same position and size (bit-0 to bit-6)
  - The selection of the ***rd*** register spans the same bit positions, bit-7 to bit-11
  - This also happens for the fields selecting the source registers ***rs1*** and ***rs2***.
- You can however write a bunch of RISC-V instructions that don't take three arguments, but a lot of these are in fact ***pseudo-instructions.*** (We will introduce them shortly).
  - For example:

```
NEG x2, x4   # x2 ← negative of x4
```

  - It is in fact a shorthand for:

```
SUB x2, zero, x4   # x2 ← zero - x4
```



RISC-V Assembler: Loading and Storing Data

- To load data from memory into registers or store data in registers to memory we use the **L** and **S** instructions. You need to use different suffixes to indicate what you are loading or storing:
  - LB - Load Byte, LW - Load Word, LD - Load Double
  - SB - Store Byte, SW - Store Word, SD - Store Double

We use only LW and SW for simplicity.

- On RISC-V a word means 32-bits, while a byte means 8-bits and double 64 bits. These instructions take three operands but the third one is an immediate value. Some examples of usage:

```
LW   x2, 2(x0)   # x2 ← [2], load contents at address 2
LW   x3, 4(x2)   # x3 ← [4 + x2], load content of addr 4 + x2
SW   x1, 8(x0)   # x1 → [8], store x1 at addr 8
```

RISC-V Assembler: Addresses, Jumps and Labels

- For a RISC-V program every instruction takes 32-bits (4 bytes). If first instruction's address is 0, the 2nd's is 4, the 3rd's is 8 and so on.
- The microprocessor keeps track of the next instruction to execute with a special register called the ***Program Counter (PC).*** It gives the address in bytes of the next instruction to execute. Since each instruction is 4 bytes long, the PC gets incremented by 4 each time
- An example: `BEQ x2, x4, 16`
  - The BEQ (branch if equal) means: If `x2 = x4` then the program counter (PC) will be updated to: `PC ← PC + 16`
  - That means jumping 4 instructions forward. (we skip the next 3 instructions).

RISC-V Assembler: Addresses, Jumps and Labels

- Let's look at a count down program. The first column contains the instruction address:

```
00:   ADDI x2, zero, 1     # x2 ← 0  + 1
04:   SUB x1, x1, x2       # x1 ← x1 - 1
08:   SW  x1, 4(x4)        # x1 → [4 + x4]
12:   BLT zero, x1, -8     # 0 < x1 => PC ← PC - 8 = 4
16:   HLT                  # Halt, stop execution
```

Conceptual code. Do not work in RARS as is

- On line 12, we check if x1 is still larger than zero (to see if not end countdown). If it is, we want to jump to line 04 (where we use SUB to subtract 1 from x1).
- However, we don't write BLT zero, x1, 4. Instead we specify -8. That is because jumps are relative (to PC). We jump two instructions backwards.

RISC-V Assembler: Addresses, Jumps and Labels

- The relative branch saves a lot of space. You only have 32-bits to encode an instruction:
  - Encode a register requires 5 bits. 2 registers for branch eats up 10-bits.
  - The opcode and function (funct3) eat up 10-bits.
  - That leaves 12-bits to specify an address to jump to. The maximum number you get with 12-bits is 4096 (2^{12}) -1. Thus, if your program was larger than 4 KB you couldn't perform jumps.

- With relative addressing we can jump 2048 bytes backwards or forwards in the program. Most for-loop, while-loop and if statements will not be larger than that.

RISC-V Assembler: Addresses, Jumps and Labels

- Nevertheless, there is one problem with relative addressing. It is awkward for the programmer to write. To solve it: **address label**.

```
    ADDI x2, x0, 1
loop:
    SUB x1, x1, x2
    SW  x1, 4(x4)
    BLT x0, x1, loop
```

Address	Code	Basic	Source
0x00400000	0x00100113	ADDI x2,x0,1	3: ADDI x2, zero, 1
0x00400004	0x402080b3	SUB x1,x1,x2	5: SUB x1, x1, x2
0x00400008	0x00122223	SW x1,4(x4)	6: SW x1, 4(x4)
0x0040000c	0xfe104ce3	BLT x0,x1,0xfffffff8	7: BLT zero, x1, loop

- You can simply label the location you want to jump. Here we use the label **`loop`**. Use a colon (“:”) to indicate this is a label.
- The assembler will use the label to calculate what offset needs to be used to jump to the given label.

RISC-V Assembler: Jumps and Branch

- Let us look at different types of jumps. An instruction that makes an unconditional jump starts with a ***J***. Jumps which are conditional start with a ***B*** for Branch.
- **Branching (Conditional Jumps).** Composed by mnemonic **B** and a two or three letter combination describing the condition such as:
  - EQ (= EQual); NE (≠ Not Equal); LT < (Less Than); GE (≥ Greater or Equal).
- A few examples of what that would translate to:

```
BEQ x2, x4, offset    # x2 = x4   => PC ← PC + offset
BNE x2, x4, offset    # x2 ≠ x4   => PC ← PC + offset
BLT x2, x4, offset    # x2 < x4   => PC ← PC + offset
BGE x2, x4, offset    # x2 ≥ x4   => PC ← PC + offset
```

RISC-V Assembler: Jumps and Branch

- **Unconditional Jumps.** Often, we need to jump around in code without checking if a condition is true or false. Examples of this are:
  - Calling a function. That means setting registers as function inputs and doing an unconditional jump to a location in memory where the function resides.
  - Returning from a function. When we are done executing code in a function we need to return to the instruction after the call-site.
  - But often you simply need to make unconditional jumps.

RISC-V Assembler: Jumps and Branch

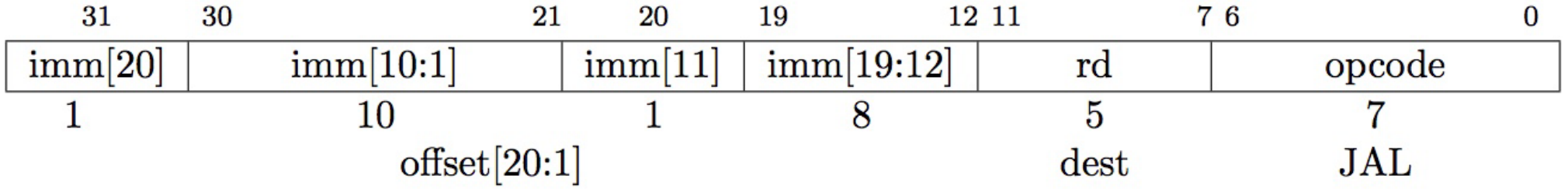
- **Jump and Link (JAL).** The JAL instruction can be used for both calling functions or just making a simple unconditional jump.
  - JAL makes a relative jump (relative to PC) just like the conditional branch.
  - However, the provided register argument is not used for comparisons but to store return address.
  - If you don't need the return address, you can simply provide the zero reg. **x0**.

```
JAL rd, offset      # rd ← PC + 4, PC ← PC + offset
```

- The convention used with RISC-V is that the return address should be stored in the return address register **ra** (which is **x1)**.

RISC-V Assembler: Jumps and Branch

- **Jump and Link (JAL).** It is a J-Format Instruction



- JAL saves PC+4 in register **rd** (the return address)
- Set PC = PC + offset (PC-relative jump: offset = signed immediate * 2)
- Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart (i.e $\pm 2^{18}$ 32-bit instructions, $\pm 2^{20}$ bytes)
- Comments:
  - Assembler "**j**" jump is a pseudo-instruction, uses JAL but sets rd=x0 to discard return addr.
  - The Immediate value is encoded optimized, similarly to the branch instruction, to reduce hardware cost

RISC-V Assembler: Jumps and Branch

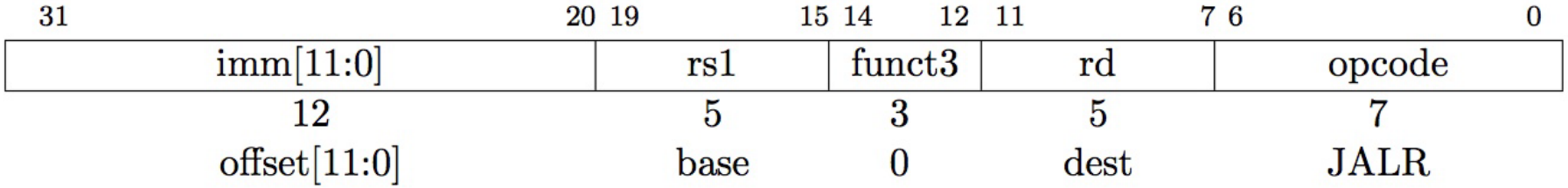
- **Jump and Link Register (JALR).** This is really the same instruction but with the difference that we use an offset from register? What is the point of that?
  - In JAL there is simply not enough space to encode a full 32-bit address. That means you cannot jump anywhere in the code if you are in a larger program. But if you use an address contained in a register, you can jump to any address.
  - JALR works almost the same as JAL. It stores the return address in rd (x1).

```
JALR rd, offset(rs1) # rd ← PC + 4, PC ← rs1 + offset
```

  - The big difference is that JALR jumps are not relative to PC, Instead they are relative to **rs1** .

RISC-V Assembler: Jumps and Branch

- **Jump and Link Register (JALR).** It is a I-Format Instruction



- JALR saves PC+4 in register **rd** (the return address). Same as JAL
- Set PC = rs + offset (12 bits, sign extend). Offset has less range than JAL.
- Uses same immediates as arithmetic and loads
  - Unlike branches, **no** multiplication by two before adding to rs to form the new PC
  - Byte offset **NOT** halfword offset as in branches and JAL

RISC-V Assembler: Use of JAL and JALR

- Uses of JAL
  - Jump (j) is a pseudo-instruction

```
j Label = jal x0, Label

# Discards return address
```

  - In order to Call function within 2^{18} instructions of PC

```
jal ra, FuncName
```

- Uses of JALR
  - For return (ret) and jr pseudo-instr.

```
ret = jr ra = jalr x0, ra, 0
```

  - Call function at any 32-bit absolute address

```
lui x1, <hi20bits>
jalr ra, x1, <lo12bits>
```

  - Jump PC-relative with 32-bit offset

```
auipc x1, <hi20bits>
jalr x0, x1, <lo12bits>
```

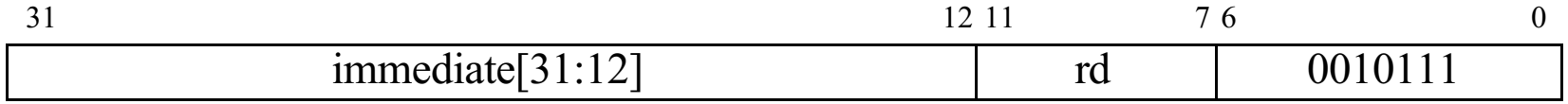
Lui – load upper immediate. Saves the immediate in the upper part of a registers (20 most significant bits)
Auipc – add upper immediate to PC. Saves the PC + immediate in a registers.

RISC-V Assembler: AUIPC

- Uses of AUIPC
  - Add U-Immediate with PC
  - Used in several pseudo-instructions to store upper part (j , la – load address, etc.)

auipc rd, immediate `x[rd] = pc + sext(immediate[31:12] << 12)`

Add Upper Immediate to PC. Tipo U, RV32I y RV64I.
Suma el *inmediato* sign-extended de 20 bits, corrido a la izquierda por 12 bits, al *pc*, y escribe el resultado en x[*rd*].



RISC-V Assembler: Constants

- The following example shows loading a constant using the %hi and %lo assembler functions.

```
.equ UART_BASE, 0x40003080

lui  a0, %hi(UART_BASE)
addi a0, a0, %lo(UART_BASE)
```

RISC-V Assembler: data and text segment

- You need to define what goes to data and instruction memory.
- Define data segment (.data)
  - You can initialize the data memory content
- Define code segment (.text)
  - The code to be executed.

```
.data
argument:   .word    8
fib_seq:    .space   16 # guarda hasta 16 valores en mem
str1:       .string "Fibonacci: 0 1"
space:      .string " "

.text
main:                                   # initial value
        lw    t0, argument              # n = 10
        jal   printHeader               # imprime string

        xor   t2, t2, t2                # t1 indice posiciï¿½n
        la    t1, fib_seq               # apuntador a seq fib mem datos
        sw    t2, (t1)                  # fib[0] = 0
        addi  t2, t2, 1
        sw    t2, 4(t1)                 # fib[1] = 2
        addi  t1, t1, 4                 # t1 apunta a fib[1]

f_loop: addi  t1, t1, 4                 # t1 apunta a fib[n+1]
        addi  t2, t2, 1                 # indice de iteraciï¿½n
        lw    t3, -8(t1)                # load fib[n-2]
```

RISC-V Assembler executed in RARS

- Analyze the effective code (pseudo-instruction and real instruction)
- Review the data in Data Segment
- Observe label address
- Review the registers and memory change in step by step