DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

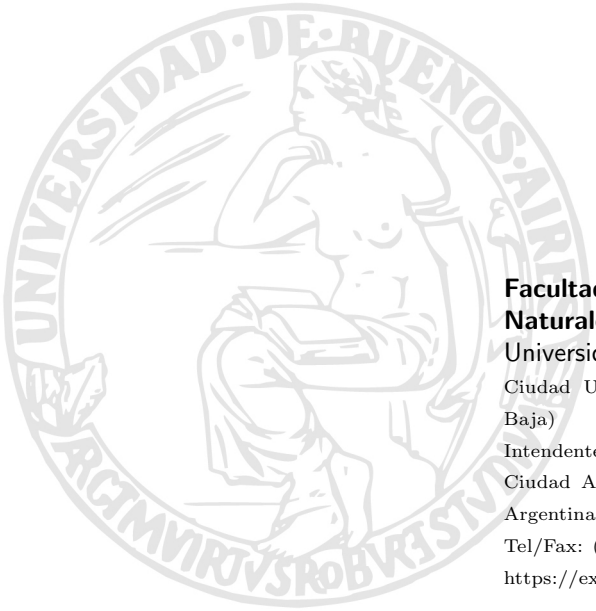
SoftFPGA

Informe

August 7, 2023 Programación de softcores en FPGAs

ret = pop rip

Integrante	LU	Correo electrónico
Losiggio, Ignacio Esteban	751/17	iglosiggio@dc.uba.ar
Demartino, Francisco	348/14	email2@dominio.com



Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

https://exactas.uba.ar

Abstract

El siguiente informe detalla el trabajo de diseño, verificación e implementación llevado a cabo por Ignacio Losiggio y Francisco Demartino bajo el marco de la materia *"Programación de Softcores en FPGAs"*. El informe consiste en cuatro partes: diseño, implementación, evaluación en un simulador y construcción de software auxiliar para ayudar al desarrollo de programas. Aunque trabajo realizado no sucedió con etapas tan marcadas creemos que esta forma de narrarlo da pié a un mejor informe.

Contents

1 ISA

Nuestra arquitectura es de una máquina de 8-bits con dos interfaces de memoria:

- **Datos:** 1kb, direccionable a 8bit (direcciones de 10 bits)
- **Código:** 1kb direccionable a 16bit (direcciones de 9 bits)

La ISA posee 16 registros (r0 a rf). Todos los registros pueden utilizarse para operación aritméticas y de acceso a memoria sin distinción.

Algunos registros son usados implícitamente dentro de varios mecanismos del procesador:

- `r0`: siempre `0x00`
- `r8`: usado como destino implícito de las operaciones aritméticas y la instrucción `setcc`
- `r9`: offset de la próxima instrucción en el segmento de código actual
- `rc`: $\frac{\text{base}}{2}$ del segmento de código actual
- `rd`: $\frac{\text{base}}{4}$ del segmento de datos actual
- `rf`: siempre `0xFF`

Nuestro ensamblador soporta una serie de nombres alternativos que sugieren ciertos usos para los registros.

Registro	Nombre	Uso
`r0`	`bh`	black hole, siempre `0x00`
`r1`	`a`	propósito general
`r2`	`b`	propósito general
`r3`	`c`	propósito general
`r4`	`d`	propósito general
`r5`	`e`	propósito general
`r6`	`f`	propósito general
`r7`	`g`	propósito general
`r8`	`fl`	flags (`S___CNVZ`)
`r9`	`ip rip pc`	instruction pointer
`ra`	`sp`	stack pointer
`rb`	`juanca`	scratch register
`rc`	`cs`	code segment
`rd`	`ds`	data segment
`re`	`lr`	link register
`rf`	`wh`	white hole, siempre `0xFF`

Table 1: Nombres y usos comunes de los registros

1.1 Macros

Nuestro lenguaje ensamblador es procesado por `cpp`, el preprocesador del lenguaje C. En los programas entregados hacemos uso de varios macros "estándar" para ayudar a la legibilidad de los mismos. Dado que el preprocesador elimina todos los comentarios válidos en el lenguaje C (`/* comentario */` y `// comentario`) el uso del mismo nos proporciona una forma sencilla de documentar nuestro código sin tener que implementarla en nuestro ensamblador.

1.1.1 `calli(addr)`

```
push(lr)
movi lr addr
xchg lr ip
pop(lr)
```

1.1.2 `callr(reg)`

```
push(lr)
movr lr reg
xchg lr ip
pop(lr)
```

1.1.3 `ret`

```
movr ip lr
```

1.1.4 `jmpi(addr)`

```
movi ip addr
```

1.1.5 `jmpr(reg)`

```
movr ip reg
```

1.1.6 `push(reg)`

```
storer [sp] reg
add sp wh
```

1.1.7 `pop(reg)`

```
sub sp wh
loadr reg [sp]
```

1.2 Instrucciones

La mayoría de las instrucciones de nuestra ISA son *"skipeables"*. Si los flags del procesador tienen el bit de skip en 1 entonces toda instrucción *"skipeable"* que también lo tenga se ejecutará como si fuera un `nop`. La representación del bit de skip en nuestro lenguaje ensamblador es por medio del prefijo `maybe`. El único mecanismo de ejecución condicional que nuestro procesador posee es este (un salto condicional es `maybe add ip <reg>`, por ejemplo).

1.2.1 `storei [imm10] r4`

- **Encoding:** `11 AAAAAAAAAA RRRR`
- **Operación:** Escribe `r4` a la dirección de memoria indicada por `imm10`
- No es skipeable

1.2.2 `loadi r4 [imm10]`

- **Encoding:** `01 AAAAAAAAAA RRRR`
- **Operación:** Escribe en `r4` el valor encontrado en la dirección de memoria indicada por `imm10`
- No es skipeable

1.2.3 `movi r4 imm8`

- **Encoding:** `0111 DDDDDDDD RRRR`
- **Operación:** Escribe en `r4` el valor `imm8`
- No es skipeable

1.2.4 `sl r4 imm3`

- **Encoding:** `0110 111 S _III RRRR`
- **Operación:** Multiplica `r4` por 2 `imm8` veces
- **Flags:**
  - **c:** Valor del último bit que se shifteó *"afuera"* de `r4`
  - **n:** 1 si el bit de signo del resultado es 1, 0 sino
  - **v:** Siempre cero
  - **z:** 1 si el resultado es 0, 0 sino

1.2.5 `sra r4 imm3`

- **Encoding:** `0110 110 S 0III RRRR`
- **Operación:** Divide `r4` por 2 `imm8` veces (realiza división con signo)
- **Flags:**
  - **c:** Siempre creo
  - **n:** 1 si el bit de signo del resultado es 1, 0 sino
  - **v:** Siempre cero
  - **z:** 1 si el resultado es 0, 0 sino

1.2.6 `srl r4 imm3`

- **Encoding:** `0110 110 S 1III RRRR`
- **Operación:** Divide `r4` por 2 `imm8` veces (realiza división sin signo)
- **Flags:**
  - **c:** Siempre cero
  - **n:** 1 si el bit de signo del resultado es 1, 0 sino
  - **v:** Siempre cero
  - **z:** 1 si el resultado es 0, 0 sino

1.2.7 `cmp r4a r4b`

- **Encoding:** `0110 101 S RRRR RRRR`
- **Operación:** Modifica los flags de acuerdo al resultado de `r4a - r4b`
- **Flags:**
  - **c:** 1 si `r4b` era mayor a `r4a` (interpretación sin signo)
  - **n:** 1 si el bit de signo del resultado es 1, 0 sino
  - **v:** 1 si el resultado no es representable en 8 bits (interpretación con signo)
  - **z:** 1 si el resultado es 0, 0 sino

1.2.8 `sub r4a r4b`

- **Encoding:** `0110 100 S RRRR RRRR`
- **Operación:** `r4a = r4a - r4b`
- **Flags:**

- **c:** 1 si `r4b` era mayor a `r4a` (interpretación sin signo)
- **n:** 1 si el bit de signo del resultado es 1, 0 sino
- **v:** 1 si el resultado no es representable en 8 bits (interpretación con signo)
- **z:** 1 si el resultado es 0, 0 sino

1.2.9 `add r4a r4b`

- **Encoding:** `0110 011 S RRRR RRRR`
- **Operación:** `r4a` = `r4a - r4b`
- **Flags:**
  - **c:** 1 si el resultado de la suma no es representable en 8 bits (interpretación sin signo)
  - **n:** 1 si el bit de signo del resultado es 1, 0 sino
  - **v:** 1 si el resultado no es representable en 8 bits (interpretación con signo)
  - **z:** 1 si el resultado es 0, 0 sino

1.2.10 `xor r4a r4b`

- **Encoding:** `0110 010 S RRRR RRRR`
- **Operación:** `r4a` = `r4a ^ r4b`
- **Flags:**
  - **c:** Siempre cero
  - **n:** 1 si el bit de signo del resultado es 1, 0 sino
  - **v:** Siempre cero
  - **z:** 1 si el resultado es 0, 0 sino

1.2.11 `or r4a r4b`

- **Encoding:** `0110 001 S RRRR RRRR`
- **Operación:** `r4a` = `r4a | r4b`
- **Flags:**
  - **c:** Siempre cero
  - **n:** 1 si el bit de signo del resultado es 1, 0 sino
  - **v:** Siempre cero
  - **z:** 1 si el resultado es 0, 0 sino

1.2.12 `and r4a r4b`

- **Encoding:** `0110 000 S RRRR RRRR`
- **Operación:** `r4a` = `r4a` & `r4b`
- **Flags:**
  - **c:** Siempre cero
  - **n:** 1 si el bit de signo del resultado es 1, 0 sino
  - **v:** Siempre cero
  - **z:** 1 si el resultado es 0, 0 sino

1.2.13 `setcc op cond`

- **Encoding:** `0101 OOO S MMMM FFFF`
- **Operación:** `flags[7]` = `op`(`flags[7]`, `cond`)
  - **Códigos de operación:**
    - `000`: Set to zero
    - `001`: NOR
    - `010`: XOR
    - `011`: NAND
    - `100`: AND
    - `101`: XNOR
    - `110`: OR
    - `111`: Set to one
  - **Condición:** La condición se describe con una cadena de texto con la forma `[cC][nN][vV][zZ]`. Las letras minúsculas describen bits de los flags que deben ser cero mientras que las mayúsculas bits que deben ser unos. Sólo se verificará el valor de las letras que formen parte de la condición. *Ejemplo:* `cNv` es una condición que sólo se hará cierta si el carry es 0, el overflow es cero y el flag de negativo es 1.

1.2.14 `storer [r4b] r4a`

- **Encoding:** `0011 ___ S RRRR RRRR`
- **Operación:** `[(ds << 2) + r4b]` = `r4a`

1.2.15 `loadr r4a [r4b]`

- **Encoding:** `0010 ___ S RRRR RRRR`
- **Operación:** `r4a` = `[(ds << 2) + r4b]`

1.2.16 `xchg r4a r4b`

- **Encoding:** `0001 ___ S RRRR RRRR`
- **Operación:** `r4a`, `r4b` = `r4b`, `r4a`

1.2.17 `movr r4a r4b`

- **Encoding:** `0000 ___ S RRRR RRRR`
- **Operación:** `r4a` = `r4b`

2 Implementación

Nuestra implementación se divide en 8 componentes. Cada componente es testeado de manera aislada y luego tres tests nos permiten evaluar el procesador con todas sus partes debidamente integradas.

2.1 `alu.v`

```
module alu(
    input  [2:0] op,
    input  [7:0] a,
    input  [7:0] b,
    input        shift_logical,
    input  [2:0] shift_imm,

    output [7:0] r,
    output [3:0] flags
);
```

En `alu.v` se implementa la unidad aritmético-lógica de nuestro procesador. El componente es completamente combinacional. El módulo recibe la operación a realizar (`op`, `shift_logical`) y sus operandos (`a`, `b`, `shift_imm`) como entrada y emite el valor resultado (`r`) y las flags asociadas (`flags`) como salida. Es importante notar que las flags asociadas son sólo cuatro (carry, negative, overflow y zero) pese a que el registro de flags es de 8 bits.

2.2 `registers.v`

```
module registers(
    input        clk,
    input        rst,
    input  [3:0] sel_read_a,
    input  [3:0] sel_read_b,
    input        exchange_a_b,
    input        en_write_reg,
```

```
    input  [3:0] sel_write_reg,
    input  [7:0] data_write_reg,
    input        en_write_flags,
    input  [7:0] data_write_flags,
    input        en_write_ip,
    input  [7:0] data_write_ip,

    output [7:0] r_read_a,
    output [7:0] r_read_b,
    output [7:0] r_flags,
    output [7:0] r_ip,
    output [7:0] r_cs,
    output [7:0] r_ds
);
```

En `registers.v` se implementa nuestro banco de 16 registros. El mismo posee dos puertos de lectura (`sel_read_a`, `r_read_a` y `sel_read_b`, `r_read_b`) y uno de escritura (`en_write_reg`, `sel_write_reg`, `data_write_reg`). Adicionalmente los registros `r8` (`flags`), `r9` (`ip`), `rc` (`cs`) y `rd` (`ds`) se encuentran siempre expuestos para que el procesador pueda hacer uso de ellos.

Finalmente nuestro banco de registros implementa la instrucción `xchg` por medio de la señal `exchange_a_b` la cual causa un intercambio entre los registros seleccionados por los dos puertos de lectura. Las escrituras por medio de intercambios o del puerto de escritura toman precedencia sobre los puertos de escritura dedicados (`ip`, `flags`).

2.3 setcc_logic.v

```
module setcc_logic(
    input previous_s,
    input [2:0] op,
    input [3:0] mask,
    input [3:0] expected_flags,
    input [3:0] current_flags,

    output s
);
```

`setcc_logic.v` implementa la lógica necesaria para la instrucción `setcc`. Es un componente completamente combinacional.

2.4 ir.v

```
module ir(
    input  [15:0] data,
    input         skip,
```

```
    output [3:0] op_code,
    output [2:0] subop_code,
    output [9:0] addr,
    output [7:0] imm,
    output [3:0] sel_ra,
    output [3:0] sel_rb,
    output       shift_logical,
    output [2:0] shift_imm,
    output [3:0] setcc_mask,
    output [3:0] setcc_expected
);
```

`ir.v` implementa la decodificación de las instrucciones de nuestra ISA. Es un componente completamente combinacional. El mecanismo de ejecución condicional se implementa en este componente: cuando una instrucción *skipeable* es fetcheada el módulo emitirá las señales necesarias para causar un `nop` (`mov bh, bh`) si la señal `skip` se encuentra activa.

2.5 core.v

```
module core(
    input         clk,
    input         rst,

    input  [15:0] instruction,
    input  [7:0]  mem_load,

    output        mem_en_store,
    output        mem_en_load,
    output [7:0]  mem_store,
    output [9:0]  mem_addr,
    output [8:0]  instruction_addr,

    input  [7:0]  io_input,
    output [7:0]  io_output
);
```

`core.v` integra todos los componentes anteriores para construir nuestro núcleo. Las señales de control se encuentran (lamentablemente) implementadas aquí. La interfaz de nuestro núcleo es sencilla:

- Un bus de código de 16 bits: (`instruction_addr`, `instruction`)
- Un bus de datos de 8 bits: (`mem_addr`, `mem_load`, `mem_store`, `mem_en_store`, `mem_en_load`)
- Un puerto de I/O de entrada de 8 bits: `io_input`

- Un puerto de I/O de salida de 8 bits: `io_output`

2.6 mem.v

```
module mem(
    input        clk,

    input        en_store,
    input  [9:0] addr_store,
    input  [7:0] data_store,

    input        en_load,
    input  [9:0] addr_load,
    output [7:0] data_load
);
```

`mem.v` implementa la memoria de datos de 1kb. Posee un puerto de lectura (`en_load`, `addr_load`, `data_load`) y uno de escritura (`en_store`, `addr_store`, `data_store`).

2.7 plaquita.v

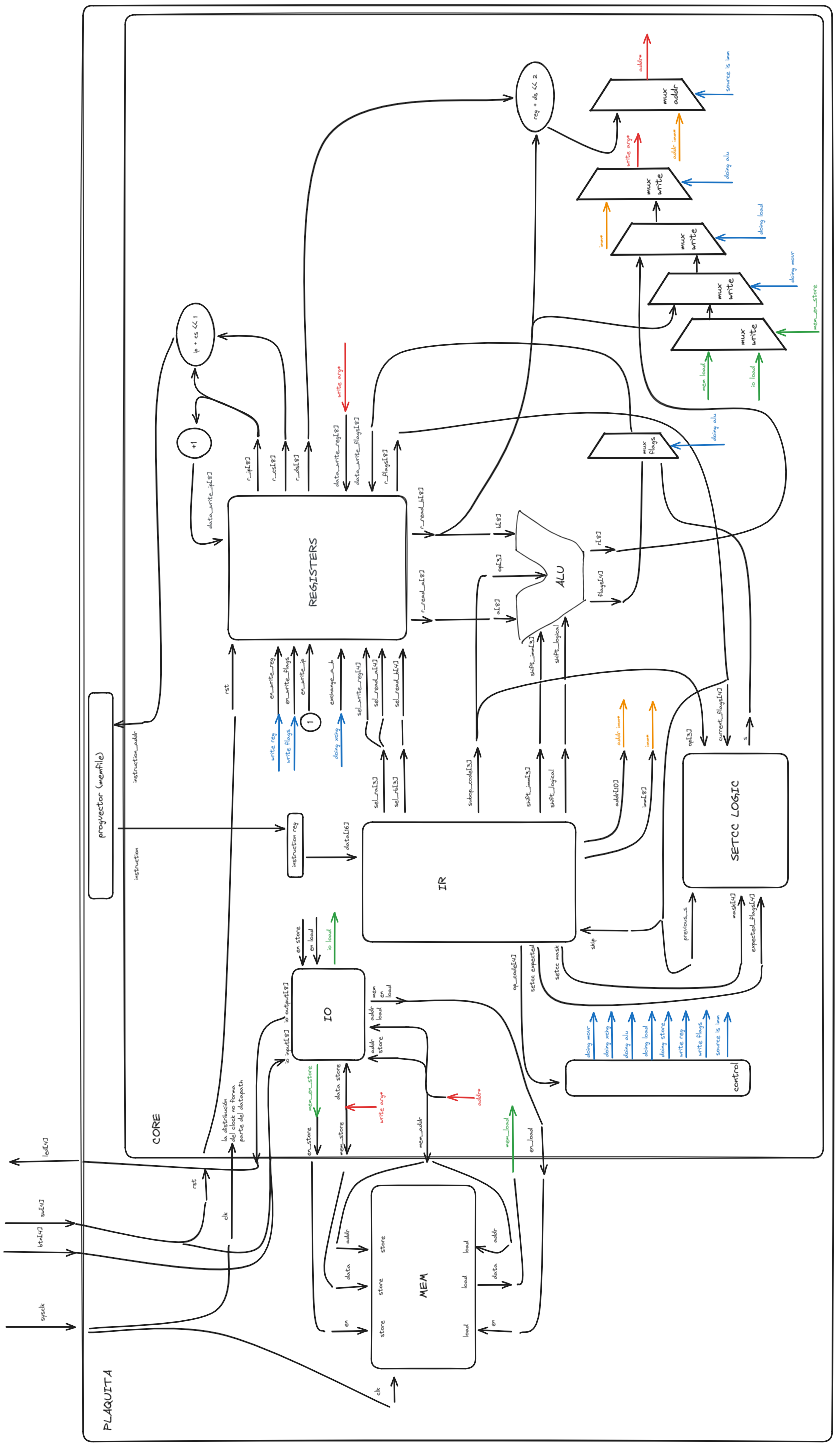
```
module plaquita(
    input        sysclk,
    input  [3:0] sw,
    input  [3:0] btn,
    output [3:0] led
);
```

`plaquita.v` implementa nuestro core en la FPGA. El puerto de entrada del core es la combinación de los botones (`btn`) y los switches (`sw`) disponibles en la placa mientras que a la salida sólo conectamos los cuatro bits menos significativos a los leds (`led`) de la placa.

2.8 Ciclo de instrucción

La máquina está construida de forma que `core.v` ejecuta una instrucción por ciclo. Por problemas del diseño actual el clock máximo soportado es menor al clock de la placa que utilizamos en la materia. Por esto la implementación entregada hace uso del *"Clocking wizard"* provisto por Xilinx para generar un clock mucho menor (10MHz) en el cuál nuestro diseño funcione.

2.9 Datapath



13
Figure 1: Datapath

3 Evaluación

Cada componente de nuestro procesador (con excepción de `ir.v` y `plaquita.v`) tiene un testbench asociado. Cada testbench prueba el componente de manera aislada simulando los estímulos externos que fueran necesarios. Para la mayoría de los componentes la primera estrategia es adecuada pero `core.v` requiere de trabajo adicional. La evaluación de `core.v` se divide en dos testbenchs:

- **core:** Un testbench en el cual se prueba cada instrucción en una cantidad reducida de escenarios. Este testbench todos los estímulos externos (memoria de código y datos) por lo que resulta muy tedioso agregarle tests.
- **progrunner:** Un testbench que realiza una ejecución simulada de un programa dado utilizando una memoria real y una secuencia entradas de I/O esperadas. Este testbench es más fácil de utilizar para verificar el funcionamiento del core, pero no resulta útil como test automatizado ya que no verifica las salidas de I/O.

4 Tooling y ensamblador

Para escribir programas en nuestra ISA desarrollamos un pequeño ensamblador disponible en `asm/`. Este ensamblador es un tanto limitado, por lo que utilizamos `make(1)` para describir un proceso de compilación más útil en el que hacemos uso de `cpp` como herramienta de preprocesamiento de nuestros archivos. El preprocesamiento nos permite renombrar los registros de forma que el código fuente resulte más fácil de leer. Como parte de la entrega se incluyen cuatro programas escritos en ensamblador:

- `counter.asm`: Un contador binario que es incrementado luego de apretar un botón de la placa
- `fib.asm`: Cálculo iterativo de números de fibonacci (se visualizan en binario por medio de los leds)
- `random.asm`, `random-with-sleep.asm`: Generador de números pseudoaleatorio por medio de un `xorshift` de 16 bits (los números son visibles por medio de los leds)
- `simon.asm`: El juego solicitado por parte de la cátedra

5 Conclusiones

Para cerrar este informe queremos mostrar algunas partes interesantes del código que escribimos durante el desarrollo del trabajo. No consideramos que el diseño (para la ISA, tooling o demos) propuesto sea *"bueno"* pero sí creemos que fué divertido de pensar y construir.

5.1 Calcular 2^k para k pequeño

Durante el desarrollo del simon quisimos utilizar un generador de números pseudoaleatorios masomenos realista. Decidimos utilizar un xorshifter con un estado de 16 bits. Un problema de esto es que nosotros queríamos elegir un led al azar (`0b1000`, `0b0100`, `0b0010`, `0b0001`) pero el generador de números nos otorgaba un número al azar (`0b00`, `0b01`, `0b10`, `0b11`). En un entorno de desarrollo normal hubiéramos hecho `1 << n`, pero carecíamos de una operación de shift arbitrario. El poder utilizar el `ip` como registro de propósito general jugó a nuestro favor, por lo que pudimos escrbir lo siguiente:

```
dos_a_la_k:
// input: juanca = k (entre 0 y 7)
// output: juanca = 2^k
    sl juanca 1
    add ip juanca   // saltamos a uno de los movi
    movi juanca 1   // juanca = 0
    ret
    movi juanca 2   // juanca = 1
    ret
    movi juanca 4   // juanca = 2
    ret
    movi juanca 8   // juanca = 3
    ret
    movi juanca 16  // juanca = 4
    ret
    movi juanca 32  // juanca = 5
    ret
    movi juanca 64  // juanca = 6
    ret
    movi juanca 128 // juanca = 7
    ret
```

5.2 Un tokenizer es una bolsa de regex

Para nuestro ensamblador el rendimiento no era muy importante, por lo que decidimos usar los mecanismos más sencillos que encontráramos. A la hora de escribir nuestro *tokenizer* lo caracterizamos por medios de dos tablas (*keywords* y *actionable tokens*. El lexer prueba los patrones uno por uno y ejecuta la acción relacionada al primer match para procesarlo de ser necesario (parseando números, por ejemplo).

```
TOKENS_TEXT = {
    # INSTRUCTIONS
    "storei",
    "loadi",
    "movi",
```

```
    ...

    # REGISTERS
    "r0",
    ...

    # SIGILS
    "[",
    "]",

    # SETCC
    "zero",
    ...

    # CONDITIONALS
    "maybe",
}

TOKENS_PARSE = [
    # NUMBERS
    (r"0x([0-9a-fA-F]+)",            lambda m: int(m, 16)),
    (r"0b([0-1]+)",                  lambda m: int(m,  2)),
    (r"(\d+)",                       lambda m: int(m, 10)),

    # LABEL DEFS
    (r"([a-zA-Z-_][0-9a-zA-Z-_]+):", lambda m: ('labeldef', m)),

    # SETCC SHENANIGANS
    (r"([cC]?[nN]?[vV]?[zZ]?)",      lambda m: {c.lower(): c == c.upper() for c in m}),

    # LABEL NAMES
    (r"([a-zA-Z-_][0-9a-zA-Z-_]+)", lambda m: ('labelname', m)),
]
```

5.3 Parsing por medio de pattern-matching

Una vez tuvimos el texto tokenizado decidimos parsearlo haciendo uso del pattern-matching presente en las últimas versiones de python. Esto nos permitió expresar las reglas de parseo de forma muy directa.

```
def asm_ins(line, labels, is_conditional):
    match line:
        case 'storei', '[', imm10, ']', r4a:
            return asm_store(imm10, r4a)
        case 'loadi', r4a, '[', imm10, ']':
            return asm_load(imm10, r4a)
```

```
        case 'movi', r4a, imm8:
            return asm_r4_imm8(0b0111, r4a, imm8, labels=labels)

        case 'srl', r4a, imm3:
            return asm_shift(False, True, is_conditional, imm3, r4a)

        ...

        case _:
            raise NotImplementedError(line)
```

5.4 La dificultad de escribir loops

El diseño que dimos para la instrucción `setcc` no es ideal. La falta de una operación *set* complejiza el código, dado que nos fuerza a ejecutar `setcc zero` ó `setcc one` antes de empezar una comparación. Por otra parte, que las instrucciones marcadas con `maybe` no se ejecuten al setear el bit S resultó poco intuitivo (dado que S=0 nos parecía el estado natural de ese bit). Luego de programar un poco en nuestra ISA la mayoría de los loops tomó la siguiente forma:

```
setcc one            // Clear last comparison
loop:
    ...              // Do something
    ...              // Perform a comparison
    setcc and Z      // Check if flags have the proper shape to keep looping
    maybe add ip ONE // If they DONT then exit the loop
    jmpi(loop)       // Otherwise keep looping
//  ^^^^^^^^^^ we cannot ‘maybe‘ this because ‘movi‘ is not skippable!
```