DiBU: Digilent Based Unit

Pablo Balbi, Rodrigo Parra

Mayo 2023

Índice

1. Introducción

Este documento contiene la descripción del ISA[1] de la máquina DiBU , en conjunto con sus decisiones de diseño y detalles de implementación. A lo largo del documento se referencia múltiples partes del código fuente, las cuales se encuentran documentadas mediante comentarios. Por ello, este documento no debe ser leído aislado del código fuente, sino en conjunto con el mismo.

La máquina DiBU cuenta con las siguientes prestaciones:

- Ocho registros de propósito general, cada uno de 8 bits de ancho.
- Soporte para llamadas a función, con un límite de 8 llamadas anidadas.
- Un puerto de entrada y otro de salida de datos, cada uno con 4 bits.
- Una memoria de datos de 1kB, con direccionamiento a byte.
- Instrucciones de tamaño fijo de 16 bits, con una memoria de código de 1kB.

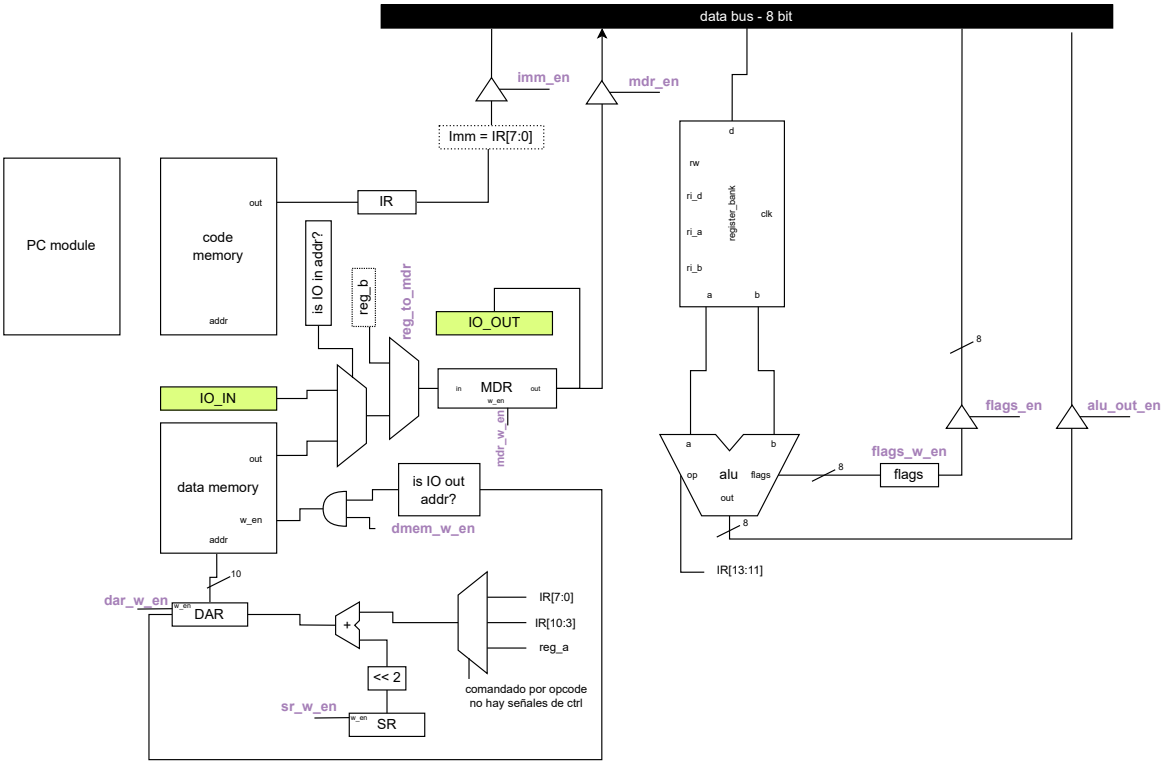


Figura 1: Diagrama del datapath de la máquina.

[1] Instruction Set Architecture.

1.1. Estructura del documento

Este documento primero presenta algunas características generales de la máquina DiBU .

La sección 2 y 3 detalla los diferentes registros expuestos al programador y la organización tanto de la memoria de código como datos.

Luego, la sección de 4 primero describe los diferentes formatos de instrucción mediante los cuales se codifican las instrucciones en la memoria de código. También se presenta una tabla de referencia con todas las instrucciones soportadas y la acción que cada una realiza. Se detalla como se codifica cada tipo de instrucción junto con sus operandos. Por último, la sección 5 detalla la implementación de una serie de instrucciones "virtuales", que facilitan la codificación de programas.

A continuación, una serie de secciones explican algunos detalles de funcionamiento de la máquina: cómo se implementan los llamados a función, en la sección 6 y el funcionamiento interno de la unidad de control en 7.

Finalmente, la sección 8 relata el enfoque utilizado para la validación del funcionamiento de la máquina.

2. Registros

La máquina DiBU expone al programador 8 registros de propósito general, cada uno de 8 bits. Estos pueden ser usados para operaciones lógico-aritméticas, accesos a memoria y entrada / salida. Además, existen dos registros de propósito específico: el registro de segmento, SR, usado para definir la porción de memoria direccionable, y el PC, registro de 9 bits que indica la instrucción siendo ejecutada.

Registro	Tamaño (bits)	Accesible a programador	Descripción
`Rx` $\mid x \in \{0, .., 7\}$	8	Si	Registro de propósito general
`SR`	8	Si	Registro de segmento
`PC`	9	No	Registro contador de programa

3. Memoria

Una máquina DiBU tiene dos memorias de 1kB, una de datos y una de programa. Para el caso de la memoria de instrucciones, las direcciones tendrán un tamaño de 9 bits, ya que las instrucciones tiene un tamaño fijo de 16 bits. En el caso de la memoria de datos, las direcciones tendrán un tamaño de 10 bits, con direccionamiento a byte.

Debido a que los registros de propósito general son de 8 bits, el modo de direccionamiento indirecto no podrá direccionar a la totalidad de la memoria de datos. Para ello, se tendrá un registro adicional llamado SR o **segment register**. De esta manera, dada un dirección local $\mathcal{A}_{local}$ a un segmento, la dirección física de memoria $\mathcal{A}_{fisica}$ se calcula como:

$$\mathcal{A}_{fisica} = \text{SR} \ll 2 + \mathcal{A}_{local} \tag{1}$$

Esto permite direccionar cualquier porción de 255 palabras de memoria, partiendo de una dirección determinada por el registro de segmento.

Cabe de estar que al acceder a los registros de entrada y salida es importante que el registro SR esté configurado para acceder a la sección más alta de la memoria.

4. Instrucciones

En DIBU las instrucciones tienen un tamaño fijo de 16 bits, igual que la palabra direccionable dentro de la memoria de instrucciones. Gracias a esto no se deben realizar verificaciones de la posición de memoria usada en las instrucciones de salto, o llamada a sub-rutinas.

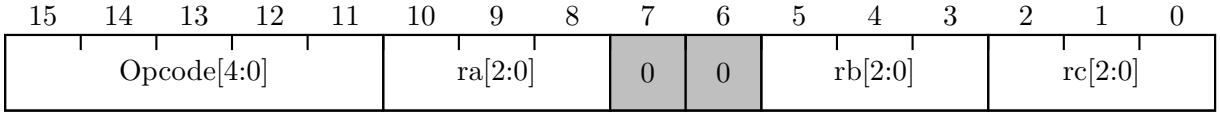
Los formatos de instrucción utilizan de manera fija los 5 bits más significativos para determinar el opcode, y el resto para operandos. En total, DIBU soporta 4 formatos de instrucción definidos en 4.1.

La asignación de códigos de operación no será secuencial, sino que seguirá un árbol de asignaciones siempre y cuando sea posible. Esto nos permite para el caso de las instrucciones lógico-aritméticas diferenciarlas del resto con los primeros 2 bit (Opcode[4:3]), y utilizar el resto como bits de código de función de la ALU (Opcode[2:0]).

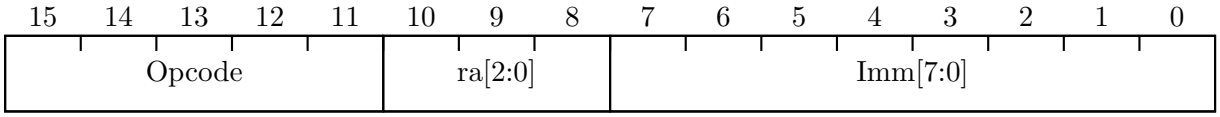
4.1. Formatos

La arquitectura DIBU cuenta con 4 formatos de instrucción los cuales están pensados para minimizar la cantidad de señales de control necesarias en el datapath.

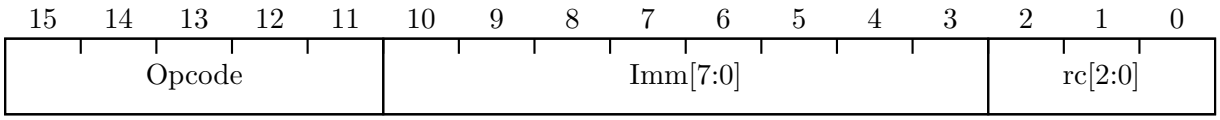
4.1.1. Formato A



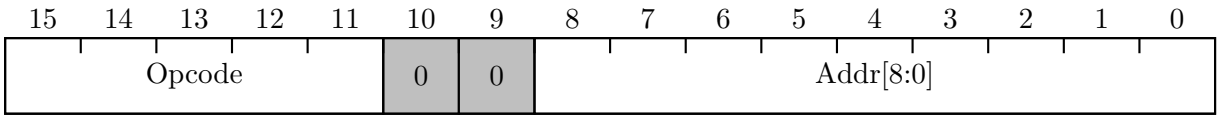
4.1.2. Formato B



4.1.3. Formato C



4.1.4. Formato D



4.2. Tabla de referencia

La siguiente tabla muestra un resumen del ISA soportado por DIBU , junto con una descripción semántica la o las operaciones que lleva a cabo cada instrucción. Al interpretar los operandos, tener en cuenta lo siguiente:

- `Rx` hace referencia a un registro de propósito general
- `Imm` hace referencia a un valor inmediato de 8 bit
- `M` hace referencia a una dirección de memoria de datos de 10 bit, y `[⋆]` a la palabra en memoria referenciada por la dirección $\star$
- `Label` hace referencia a una etiqueta en el código fuente ensamblador, la cual será traducida a una dirección de memoria de código de 9 bit por el programa ensamblador

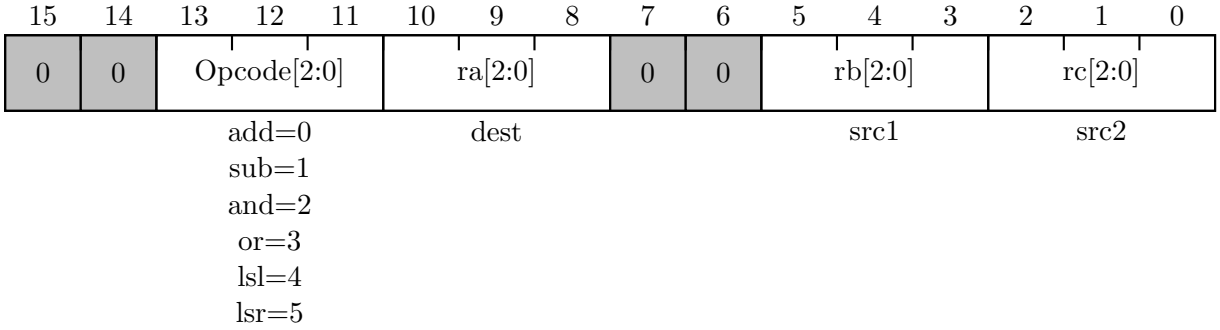
Instrucción	Acción	Formato	Opcode
`ssr Rx`	`SR` $\leftarrow$ `Rx`	A	01100
`mov Rx imm`	`Rx` $\leftarrow$ `Imm`	B	01111
`mov Rx Ry`	`Rx` $\leftarrow$ `Ry`	A	00111
`movf Rx`	`Rx` $\leftarrow$ `Flags`	A	01011
`load Rx [M]`	`Rx` $\leftarrow$ `[`$SR \ll 2 + M$`]`	B	10000
`load Rx [Ry]`	`Rx` $\leftarrow$ `[`$SR \ll 2 + R_x$`]`	A	10010
`str [Rx] Ry`	`[`$SR \ll 2 + R_x$`]` $\leftarrow$ `Ry`	A	10011
`str [M] Rx`	`[`$SR \ll 2 + M$`]` $\leftarrow$ `Rx`	C	10001
`add Ra Rb Rc`	$R_a = R_b + R_c$	A	00000
`sub Ra Rb Rc`	$R_a = R_b - R_c$	A	00001
`and Ra Rb Rc`	$R_a = R_b \wedge R_c$	A	00010
`or Ra Rb Rc`	$R_a = R_b \vee R_c$	A	00011
`lsl Ra Rb Rc`	$R_a = R_b \ll R_c$	A	00100
`lsr Ra Rb Rc`	$R_a = R_b \gg R_c$	A	00101
`not Ra Rb`	$R_a = \neg R_b$	A	00110
`cmp Ra Rb`	$Flags \leftarrow R_a - R_b$	A	00111
`addi Ra Imm`	$R_a = R_a + Imm$	A	Pseudo ins
`rnd Ra`	$R_a = Rand(R_a)$	A	11110
`jmp Label`	$PC \leftarrow Imm$	D	11000
`je Label`	if Z=1 then $PC \leftarrow Imm$ else PC++	D	11001
`jne Label`	if Z=0 then $PC \leftarrow Imm$ else PC++	D	11010
`jn Label`	if N=1 then $PC \leftarrow Imm$ else PC++	D	11011
`call Label`	Ver detalle debajo	D	11100
`ret`	Ver detalle debajo	D	11101
`halt`[2]	$PC \leftarrow 111111111$	D	11111

Para las dos instrucciones de llamadas a subrutina, se describe en detalle su funcionamiento en la sección 6.

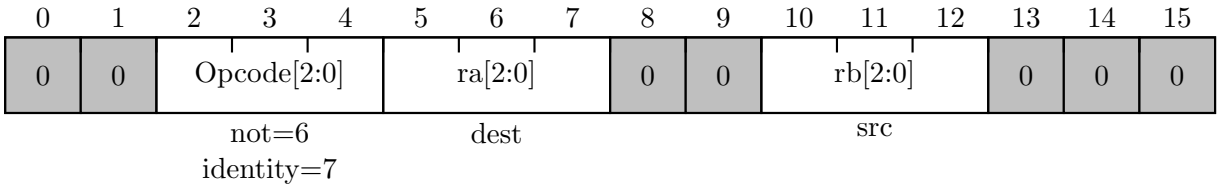
4.3. Codificación de operaciones lógico-aritméticas

Las operaciones aritméticas y lógicas de la forma $c = a \star b$, usan el formato de instrucción A, donde un registro es usado para guardar el resultado de la operación, y los otros dos como operandos.

Como fue mencionado en 4, los opcodes no son asignados secuencialmente para estas instrucciones, sino que siguen el siguiente formato: los dos bit más significativos `Opcode[4:3]` quedan fijos en 0, mientras que los restantes `Opcode[2:0]` se asignan partiendo de 0, uno por operación soportada por la ALU. Esto permite, en el datapath, rutear de manera directa estos bits desde el registro de instrucciones IR, a los puertos de código de función de la ALU.

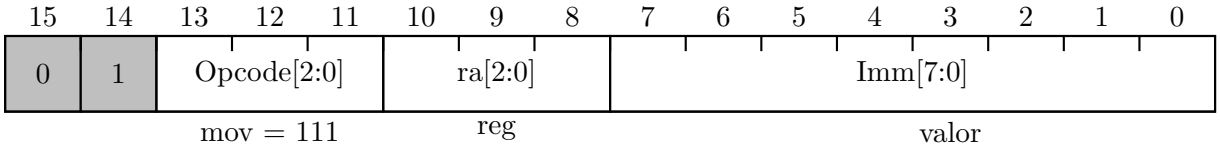


Las operaciones aritméticas y lógicas de la forma $c = \star a$, usan el mismo formato de instrucción, pero ignorando el primer registro en el formato. De esta forma, utiliza RB como único operando, y RA como registro de destino. Un caso particular de estas instrucciones es MOV RA RB, cuya ejecución cumple el mismo flujo que cualquier operación lógico aritmética, pero utilizando la ALU con una función de identidad.

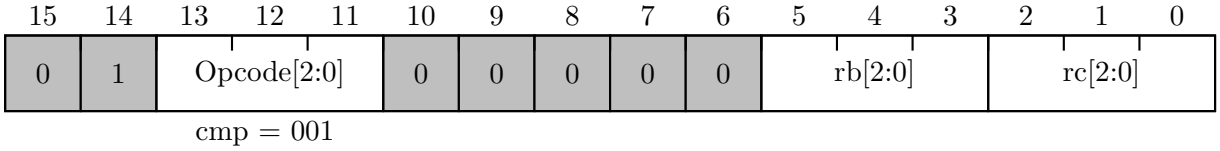


Luego, algunas instrucciones operan con único registro y un valor inmediato, permitiendo operaciones como incrementar asignar un valor inmediato a un registro, MOV RA 0XDE. Estas utilizan el formato B.

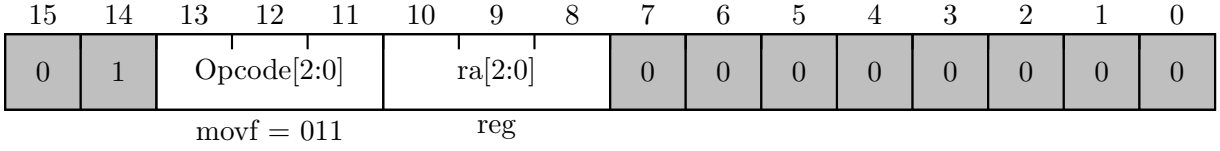
[2]Instrucción virtual, solamente usada por los test benches para detectar que finalizó la ejecución de un programa.



La instrucción CMP (compare) presenta un caso particular. Los dos bits más significativos del op-code la diferencian de las instrucciones de operaciones entre registros, pero los tres restantes tiene el valor del código de función de la ALU correspondiente a la substracción.



Por último, hay una serie de instrucciones que realizan una operación sobre un registro en particular. Las mismas siguen usando el formato A, pero solo teniendo en cuenta el último registro del formato.

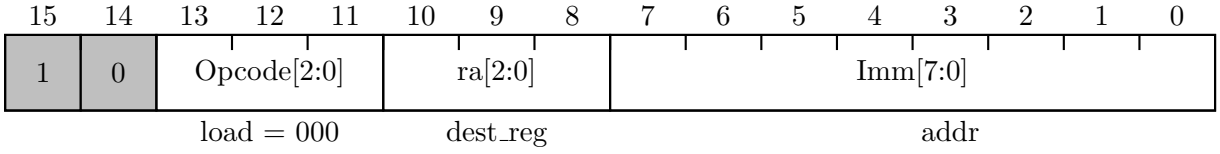


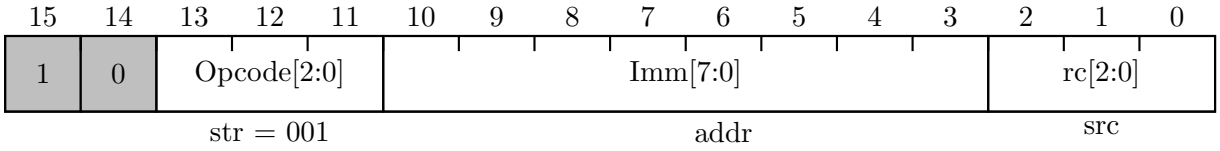
4.4. Codificación de operaciones de accesso a memoria

DIBU es una arquitectura load-store, donde solo las instrucciones de `load` y `store` acceden a memoria, y todas las operaciones lógico-aritméticas operan entre registros. La máquina implemente 3 modos de direccionamiento a memoria:

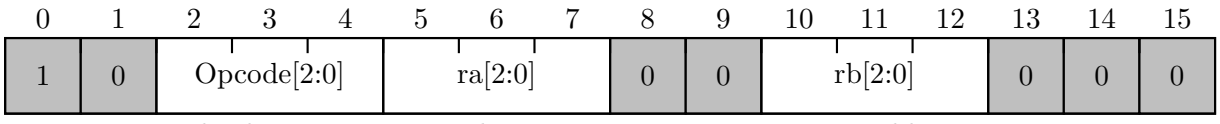
- **Directo:** Las instrucciones contienen la dirección de memoria con la cual leer / escribir.
- **Indirecto a registro:** La dirección de memoria con la cual leer / escribir es leída del valor de un registro.

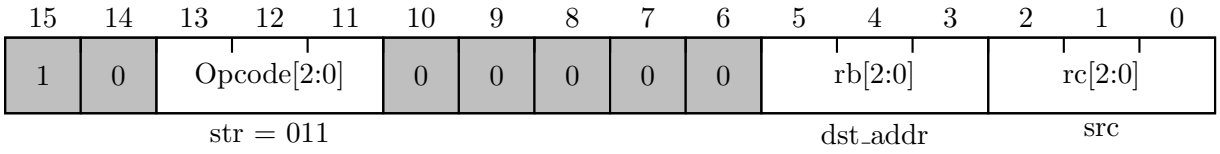
Las instrucciones de load y store con direccionamiento directo se codifican de la siguiente manera:





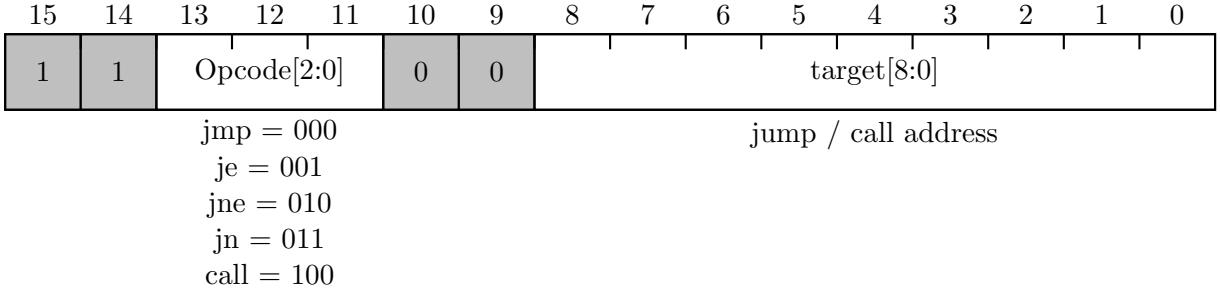
Por otro lado, en el caso de direccionamiento indirecto, la codificación es la siguiente:





4.5. Codificación de operaciones de control de flujo

DiBU provee tres tipos de instrucciones para controlar el flujo de ejecución de un programa: saltos incondicionales, saltos condicionales y llamadas a sub-rutinas, todas codificadas con el mismo formato de instrucción.



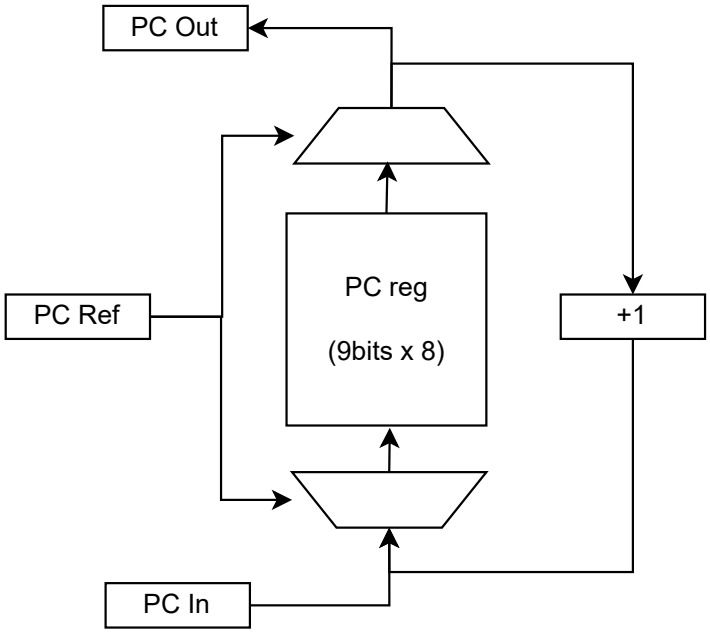
5. Pseudo-instrucciones

DiBU soporta pseudo-instrucciones, instrucciones que no se encuentran implementadas en las estructuras de micro-control de la máquina, sino que son traducidas en tiempo de compilación a una o más instrucciones "reales". Un ejemplo de estas es ADDI RI IMM, instrucción que permite sumar un valor inmediato a un registro. Este soporte fue agregado una vez que casi la totalidad de la máquina este implementada, y surgió de la necesidad de describir ciertas operaciones comunes, como incrementar el valor de un registro usado como contador, de manera más concisa. Por esto, el soporte para estas pseudo-instrucciones fue implementado dentro del programa ensamblador, y usando como registro temporal de ser necesario R7.

La implementación de esta funcionalidad se encuentra dentro de parser.py.

6. Llamadas a sub-rutinas

En base al requerimiento de que la arquitectura debe soportar al menos 5 llamadas a función anidadas se optó por utilizar una serie de 8 registros para almacenar los PC a medida que las llamadas se ejecuten. Cabe destacar que debido a que el contenido de estos registros es de 9 bits, no podían ser implementados en un stack en la memoria de datos. Esta serie de registros serán indexados mediante otro registro de 3 bits que se incrementará y decrementará en la medida en que se realicen llamados a funciones. A su vez, estos registros serán utilizados como PC, por lo que el PC activo se incrementará en la medida en que se encuentre dentro de esa función.



El siguiente pseudocódigo explica que ocurre en DiBU cuando se ejecuta una instrucción de `call atajar_penal`:

Require: $AP \leftarrow$ dirección de memoria `atajar_penal`
Require: R_{PC} apunta al PC activo, con $R_{PC} \in \{0, ..., 7\} \wedge R_{PC} < 7$
Require: M_{PC} banco de 8 registros de 9 bit
$M[R_{PC}] \leftarrow M[R_{PC}] + 1$
$R_{PC} \leftarrow R_{PC} + 1$
$M[R_{PC}] \leftarrow AP$
ReiniciarCicloDeEjecucion

Considerando el psuedo-código, en un ciclo de fetch, la instrucción a leer de memoria de código es $M_{code}[M_{PC}[R_{PC}]]$, siendo M_{code} la memoria de código.

Luego, este otro pseudocódigo explica que ocurre en DiBU cuando se ejecuta una instrucción de `ret`:

Require: R_{PC} apunta al PC activo, con $R_{PC} \in \{0, ..., 7\} \wedge R_{PC} > 0$

Require: M_{PC} banco de 8 registros de 9 bit
  $R_{PC} \leftarrow R_{PC} - 1$
  ReiniciarCicloDeEjecucion

7. Unidad de control

La máquina DiBU cuenta con una organización micro-controlada, donde cada instrucción se implementa cono una serie de micro-instrucciones, que controlan las diferentes señales en el datapath, cuyo diagrama se muestra en la figura 1. Las micro-instrucciones, almacenadas en una memoria ROM dentro de la unidad de control como se puede ver en la figura 2, controlan todo el ciclo de ejecución de la máquina: los ciclos fetch, decode y execute, donde se lee una instrucción de memoria, se decodifica y ejecuta, y las acciones necesarias para ejecutar cada instrucción en particular.

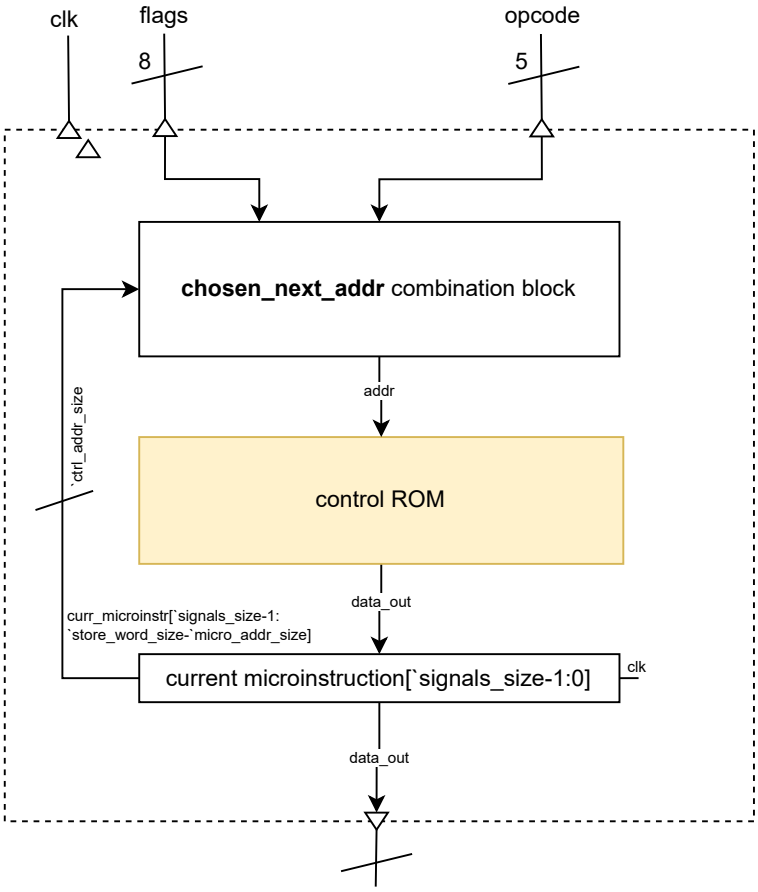


Figura 2: Diagrama de la unidad de control.

El programa denotado por todas las micro-instrucciones disponibles en la máquina, y el orden en que estas son ejecutadas, puede ser interpretado como una máquina de estados finita (FSM, o finite state machine). La figura 4 muestra una porción de esta máquina de estados, y las señales involucradas en cada estado. Cada estado, que se corresponde con una micro-instrucción, indica cual es el siguiente estado a ser ejecutado, a excepción de un caso

particular. Este último, al que se refiere como **estado de decisión**, decide la siguiente micro-instrucción a ser ejecutada a partir de el código de operación de la instrucción actual, y los flags provenientes de la ALU. La totalidad de la lógica que decide el siguiente estado se encuentra contenida en el primer bloque combinacional, etiquetado como "chosen_next_addr combination block" en la figura 2. También se puede apreciar en la figura 3 que cada micro-instrucción codifica todas las señales del datapath (*signals)*, la dirección en la memoria ROM de la siguiente micro-instrucción a ejecutar, o un bit especial, que indica que el estado actual es el estado de decisión.

Debido a que la codificación de la unidad de control en Verilog requería codificar además, el contenido de la memoria ROM, se optó por implementar una herramienta que genera de manera automática las micro-instrucciones codificadas en la manera que se representan en la memoria, y ciertas porciones del código del datapath (mapping entre la salida de la unidad de control y las señales individuales). El código fuente de esta herramienta se encuentra en microprogram.py.

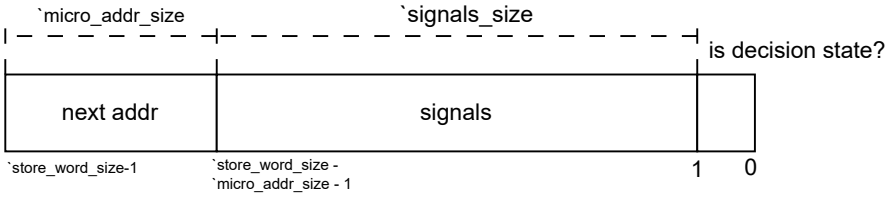
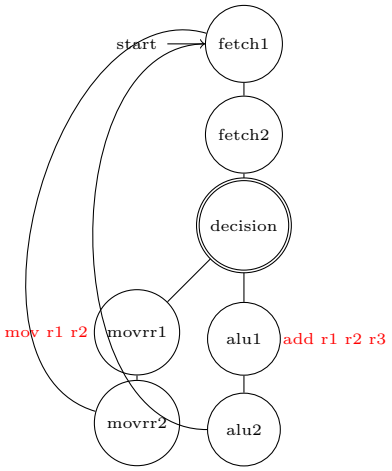


Figura 3: Formato de micro-instrucción.



Estado	Señales
fetch1	MarWEn, PCWEn
fetch2	IRWEn
movrr1	∅
movrr2	RegRW, AluOutEn
alu1	∅
alu2	FlagsWEn, RegRW, AluOutEn

Figura 4: Ejemplo de componente de la máquina de estados finita denotado por el programa de micro-instrucciones. La parte representada corresponde a la ejecución de dos intrucciones que operan sobre registros.

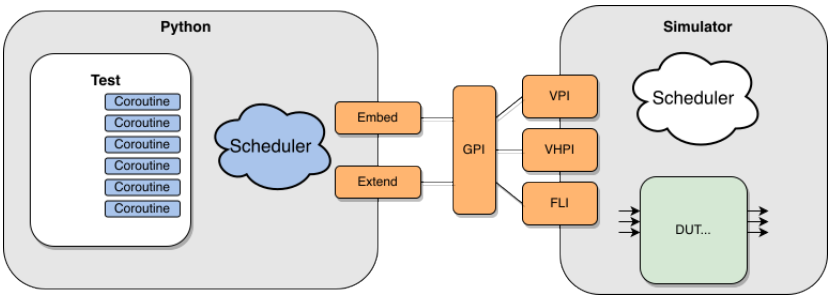


Figura 5: Descripción de alto nivel del funcionamineto de Cocotb, framework utilizado para el desarrollo de los test-benches de cada componente.

8. Testing y validación

Para el desarrollo de test-benches de los componentes se optó por utilizar un enfoque diferente a desarrollarlos en Verilog, debido a que el desarrollo es más complejo y más difícil de automatizar. Para ello, se utilizó el framework Cocotb[3]. Cocotb es una biblioteca de Python que permite escribir test-benches de componentes de hardware programasdos en un lenguaje de descripción de hardware, como Verilog. Los tests desarrollados con esta herramienta siguen un enfoque de co-rutinas. Este consiste en interpretar los tests como dos hilos de ejecución, uno donde ocurren las validaciones, y otro que simula el funcionamiento de la unidad de hardware bajo validación (DUT). Durante la ejecución de una de estas pruebas, el control fluye entre el hilo validador, y el simulador, siguiendo ciertas primitivas que le indican al orquestador por cuánto tiempo ejecutar la simulación. La figura 5 muestra un diagrama de como estos hilos están conectados entre si.

En nuestra máquina, estos tests se encuentran en rtl_tests.

9. Referencias

- Microprocesador OrgaSmall
- RISC-V

[3] `https://www.cocotb.org/`