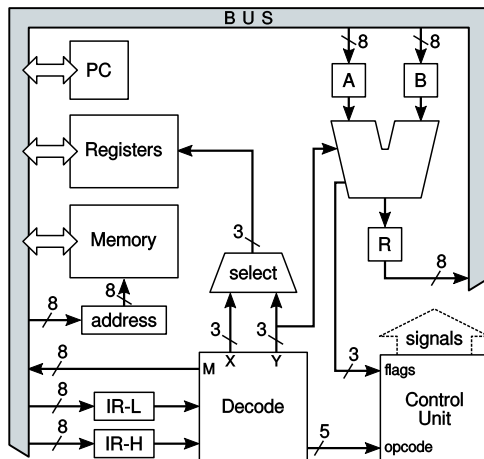


# Cartilla de Referencia - OrgaSmall

## Procesador OrgaSmall

OrgaSmall es un procesador diseñado e implementado sobre la herramienta *Logisim*. Este cuenta con las siguientes características:



- Arquitectura *von Neumann*, memoria de datos e instrucciones compartida.
- 8 registros de propósito general, R0 a R7.
- 1 registro de propósito específico PC.
- Tamaño de palabra de 8 bits e instrucciones de 16 bits.
- Memoria de 256 palabras de 8 bits.
- Bus de 8 bits.
- Diseño microprogramado.

## Instrucciones

Instrucción	CodOp	Formato	Acción
ADD Rx, Ry	00001	A	$Rx \leftarrow Rx + Ry$
ADC Rx, Ry	00010	A	$Rx \leftarrow Rx + Ry + \text{flag\_C}$
SUB Rx, Ry	00011	A	$Rx \leftarrow Rx - Ry$
AND Rx, Ry	00100	A	$Rx \leftarrow Rx \text{ and } Ry$
OR Rx, Ry	00101	A	$Rx \leftarrow Rx \text{ or } Ry$
XOR Rx, Ry	00110	A	$Rx \leftarrow Rx \text{ xor } Ry$
CMP Rx, Ry	00111	A	Modifica <i>flags</i> de Rx - Ry
MOV Rx, Ry	01000	A	$Rx \leftarrow Ry$
STR [M], Rx	10000	D	$\text{Mem}[M] \leftarrow Rx$
LOAD Rx, [M]	10001	D	$Rx \leftarrow \text{Mem}[M]$
STR [Rx], Ry	10010	A	$\text{Mem}[Rx] \leftarrow Ry$
LOAD Rx, [Ry]	10011	A	$Rx \leftarrow \text{Mem}[Ry]$
JMP M	10100	C	$PC \leftarrow M$
JC M	10101	C	Si $\text{flag\_C}=1$ entonces $PC \leftarrow M$
JZ M	10110	C	Si $\text{flag\_Z}=1$ entonces $PC \leftarrow M$
JN M	10111	C	Si $\text{flag\_N}=1$ entonces $PC \leftarrow M$
INC Rx	11000	B	$Rx \leftarrow Rx + 1$
DEC Rx	11001	B	$Rx \leftarrow Rx - 1$
SHR Rx, t	11010	A	$Rx \leftarrow Rx \ll t$
SHL Rx, t	11011	A	$Rx \leftarrow Rx \gg t$
SET Rx, M	11111	D	$Rx \leftarrow M$

Las instrucciones están codificadas en 16 bits. Los primeros 5 bits son el **opcode** de la

instrucción, el resto de los bits indican los parámetros. Existen 4 posibles codificaciones de parámetros.

Formato	Codificación	<b>XXX</b> codifica el número del registro X (Rx), vale entre 0 y 7
A	00000 XXX YYY-----	<b>YYY</b> codifica el número del registro Y (Ry) o un inmediato, vale entre 0 y 7
B	00000 XXX -----	
C	00000 --- MMMMMMMM	<b>MMMMMMMMM</b> Dirección de memoria o valor inmediato, número de 8 bits
D	00000 XXX MMMMMMMM	

Las posiciones indicadas con - deben ser cero. Las instrucciones de **opcode**: 9, 10, 11, 12, 13, 14, 15, 28, 29 y 30 no son utilizados en la máquina.

## Componentes

La arquitectura está compuesta por 6 componentes interconectados. Su representación en Logisim esta dada por un circuito principal llamado **microOrgaSmall** que integra los demás componentes en un **dataPath** (se ve en la parte izquierda del mismo, del lado derecho presenta herramientas para visulizar el estado de la máquina).

Los componentes de la arquitectura son: **Registers** (Banco de Registros), **PC** (Contador de Programa), **ALU** (Unidad Aritmético Lógica), **Memory** (Memoria), **Decode** (Decodificador de Instrucciones) y **ControlUnit** (Unidad de Control)

Cada uno de estos componentes es controlado por medio del conjunto de entradas y salidas descriptas a continuación:

<b>Registers</b> (Banco de Registros)	
inData(8) y outData(8)	Entrada y salida de datos.
RB_enIn(1) y RB_enOut(1)	Habilita entrada y salida.
RB_inSelect(1) y RB_outSelect(1)	Selecciona el índice de X e Y
<b>PC</b> (Contador de Programa)	
inValue(8) y outValue(8)	Entrada y salida del PC.
PC_load(1)	Carga un nuevo valor en el registro.
PC_inc(1)	Incrementa el valor actual.
PC_enOut(1)	Habilita la salida del valor.
<b>ALU</b> (Unidad Aritmético Lógica)	
A(8), B(8), out(8) y flags(3)	Entradas y salidas de la ALU.
ALU_enA(1), ALU_enB(1) y ALU_enOut(1)	Habilitación de entradas y salidas.
ALU_opW(1)	Indica si se deben escribir los <i>flags</i> .
ALU_OP(4)	Indica la operación a realizar por la ALU.
<b>Memory</b> (Memoria)	
inData(8) y outData(8)	Entrada y salida de datos.
MM_addr	Dirección de memoria donde leer.
MM_enOut	Habilitación de la salida.
MM_load	Indica si leer o escribir.
MM_enAddr	Habilita cargar la dirección.

Decode (Decodificador de Inst.)	
halfInst(8)	Entrada de datos (media instrucción)
DE_loadL(8) y DE_loadH(8)	Indica cargar la mitad alta o baja.
opcode(5)	Salida de Opcode decodificado.
indexX(3) y indexY(3)	Salidas de índices de registros.
valueM(8)	Salida de valor inmediato o dirección.
ControlUnit (Unidad de Control)	
inOpcode(5)	Entrada de <i>Opcode</i> .
flags(3)	Entrada de <i>flags</i> .
RB_enIn(1) y RB_enOut(1)	Señales de habilitación para Registros
RB_inSelect(1) y RB_outSelect(1)	Señales de selección para Registros
DM_enOut(1) y DM_load(1)	Señales para la Memoria de Datos
DM_enAddr(1)	Habilita escribir el registros de direcciones de la memoria de datos.
DM_selectAddr(1)	Selecciona si la dirección de la memoria de datos proviene de un valor inmediato o del bus de datos.
ALU_enA(1), ALU_enB(1) y ALU_enOut(1)	Señales de habilitación de la ALU.
ALU_opW(1), ALU_OP(4)	Señales de control de la ALU.
PC_load(1), PC_inc(1) y PC_enOut(1)	Señales de control de PC.
IM_enIn(1)	Habilita la entrada de una dirección a la memoria de direcciones.
DE_enOutImm(1)	Habilita la entrada al bus de un valor inmediato.

Algunas de estas entradas y salidas están conectadas directamente al bus del sistema, mientras que otras son utilizadas como señales de control. Estas últimas están directamente conectadas a la unidad de control (UC).

## Micro-Instrucciones

Las micro-instrucciones corresponden a la secuencia de señales necesarias para resolver una instrucción. En este diseño, tanto la operación de *fetch* como *decode* son ejecutadas por medio de micro-instrucciones. Una vez identificada la instrucción a ejecutar, la operación de *execute* también es realizada por código de micro-instrucciones.

Las micro-instrucciones son almacenadas en una memoria destinada para tal fin, que forma parte del componente UC. Esta memoria contiene palabras de 32 bits y direcciones de 9 bits. Cada uno de los 32 bits de la palabra corresponde a una señal para algún componente según se detalla más adelante. Estas señales pueden estar directamente conectadas a un componente o ser señales internas de la UC.

La UC funciona como un controlador de señales. El registro **microPC** dentro de la UC opera como un contador de programa, que indica la dirección de la próxima micro-instrucción a ejecutar. Toma como entradas el *opcode* de la instrucción a ejecutar y los valores de *flags* desde la ALU.

Inicialmente **microPC** vale cero, posición en la que se almacena la secuencia de pasos correspondientes a la etapa de *fetch*. En esta se carga desde la memoria la próxima instrucción a ser ejecutada indicada por el PC. Como la memoria direcciona a byte, y las instrucciones ocupan dos bytes, el proceso de *fetch* debe cargar dos valores desde memoria. Estos valores son enviados a la unidad de decodificación (*DE*). Esta última genera cuatro salidas M, X, Y y OP. Las primeras tres corresponden a los parámetros de la instrucción y la restante a su *opcode*. Luego la UC carga en el **microPC** el valor *opcode* << 4, es decir, agrega cuatro ceros en

los bits menos significativos del *opcode*. Las siguientes micro-instrucciones corresponden a la secuencia de señales para resolver la instrucción indicada por el *opcode*. Una vez resuelta la instrucción, el ciclo comienza nuevamente seteando **microPC** en cero.

La estructura de la memoria de micro-instrucciones es la siguiente:

Dirección	Inst.	Dirección	Inst.	Dirección	Inst.	Dirección	Inst.
00000xxxx	fetch	01000xxxx	MOV	10000xxxx	STR	11000xxxx	INC
00001xxxx	ADD	01001xxxx	-	10001xxxx	LOAD	11001xxxx	DEC
00010xxxx	ADC	01010xxxx	-	10010xxxx	STR*	11010xxxx	SHR
00011xxxx	SUB	01011xxxx	-	10011xxxx	LOAD*	11011xxxx	SHL
00100xxxx	AND	01100xxxx	-	10100xxxx	JMP	11100xxxx	-
00101xxxx	OR	01101xxxx	-	10101xxxx	JC	11101xxxx	-
00110xxxx	XOR	01110xxxx	-	10110xxxx	JZ	11110xxxx	-
00111xxxx	CMP	01111xxxx	-	10111xxxx	JN	11111xxxx	SET

Donde xxxx indica que son las posiciones de memoria resultantes de variar xxxx desde 0000 hasta 1111.  
(\*) Instrucciones con direccionamiento indirecto a memoria.

Los datos almacenados en la memoria de micro-instrucciones corresponden a señales para los distintos componentes del sistema. La siguiente tabla indica a qué bit de micro-instrucción corresponde cada señal.

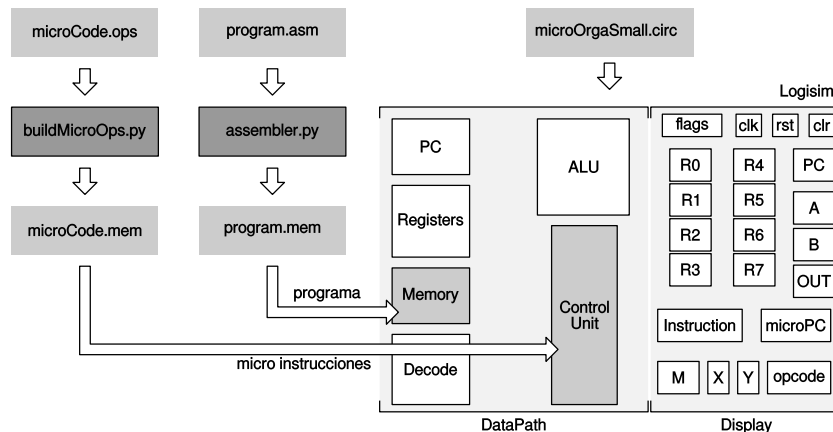
reset_microOp	load_microOp	-	-	-	DE_loadH	DE_loadL	DE_enOutImm	-	PC_enOut	PC_inc	PC_load	-	JN_microOp	JZ_microOp	JC_microOp	ALU_OP	ALU_OP	ALU_OP	ALU_OP	ALU_opW	ALU_enOut	ALU_enB	ALU_enA	-	MM_enAddr	MM_load	MM_enOut	RB_selectIndexOut	RB_selectIndexIn	RB_enOut	RB_enIn
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

La codificación de las operaciones de la unidad aritmético lógica (en **ALU\_OP**) es la siguiente:

Código	Operación	Código	Operación	Código	Operación	Código	Operación
0000	Reservada	0100	AND	1000	SHR	1100	cte0x00
0001	ADD	0101	OR	1001	SHL	1101	cte0x01
0010	ADC	0110	XOR	1010	-	1110	cte0x02
0011	SUB	0111	CMP	1011	-	1111	cte0xFF

## Herramientas

Se cuenta con dos herramientas: un programa ensamblador, que transforma código ASM en código binario, y un generador de micro-instrucciones. Este último genera a partir de un archivo de descripción de señales el código binario de micro-instrucciones.



## Ensamblador (assembler.py)

El ensamblador toma como entrada un archivo de texto con la lista de mnemónicos de instrucciones y genera el código binario del programa. Las instrucciones soportadas son todas las descritas por la arquitectura, además el ensamblador soporta el uso de etiquetas y comentarios. Las etiquetas pueden ser cualquier cadena de caracteres finalizada en “:”. Una vez declarada, puede ser utilizada en cualquier parte del código, incluso como parámetro o valor inmediato. Para declarar un comentario, se utiliza el caracter “;”. Todo el texto luego de la primera aparición de este será considerado comentario. El ensamblador además soporta declarar valores inmediatos. Para esto se utiliza la palabra reservada DB y luego de esta un valor numérico inmediato. Este será incluido directamente en el código generado.

## Generador de Micro-instrucciones (buildMicroOps.py)

El generador de micro-instrucciones toma como entrada un archivo de descripción de señales que respeta la siguiente sintaxis:

```
<binary_opcode>:
<signal_1> <signal_2> <signal_n>
...
<signal_1> <signal_2> <signal_n>
```

donde **<binary\_opcode>** corresponde al valor de *opcode* a codificar en binario y **<signal>** a un nombre de señal. Las señales pueden ser indicadas por el nombre de la misma o como: **<signal>=<x>** donde *x* indica el valor de la señal. Para señales de más de un bit, como ALU\_OP se debe utilizar el número entero decimal correspondiente. En particular, para la señal mencionada, se puede utilizar directamente el nombre de la operación en la ALU. Si no se indica el valor que debe tomar la señal, se utiliza 1.

Por ejemplo, para la codificación de la instrucción ADD:

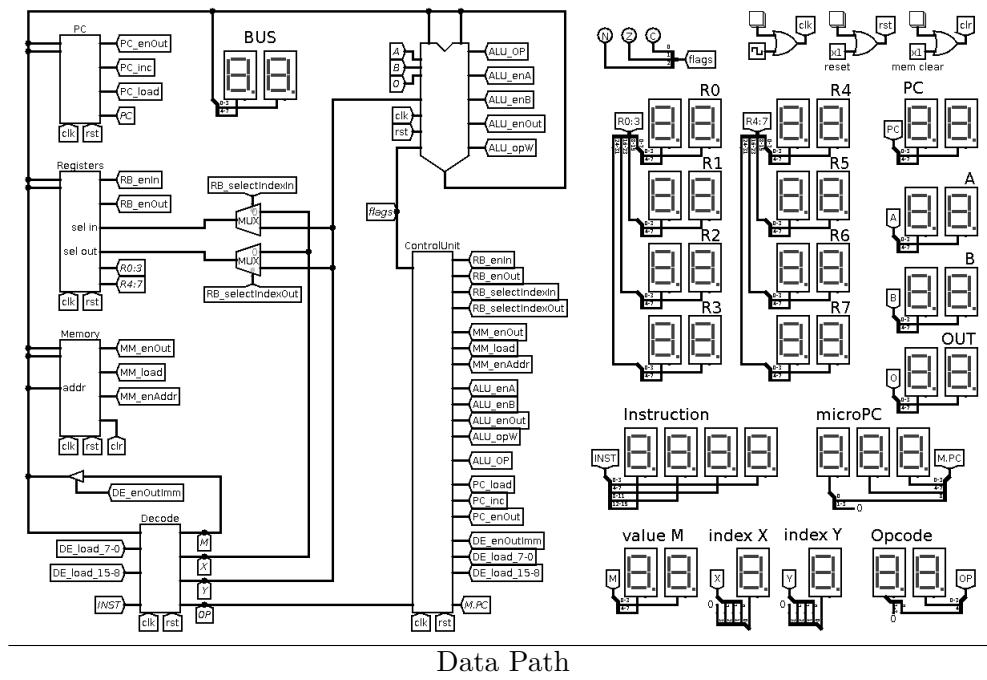
```
00001: ; ADD
RB_enOut ALU_enA RB_selectIndexOut=0 ; ALU_A := Rx
RB_enOut ALU_enB RB_selectIndexOut=1 ; ALU_B := Ry
ALU_OP=ADD ALU_opW ; ALU_ADD
RB_enIn ALU_enOut RB_selectIndexIn=0 ; Rx := ALU_OUT
reset_microOp
```

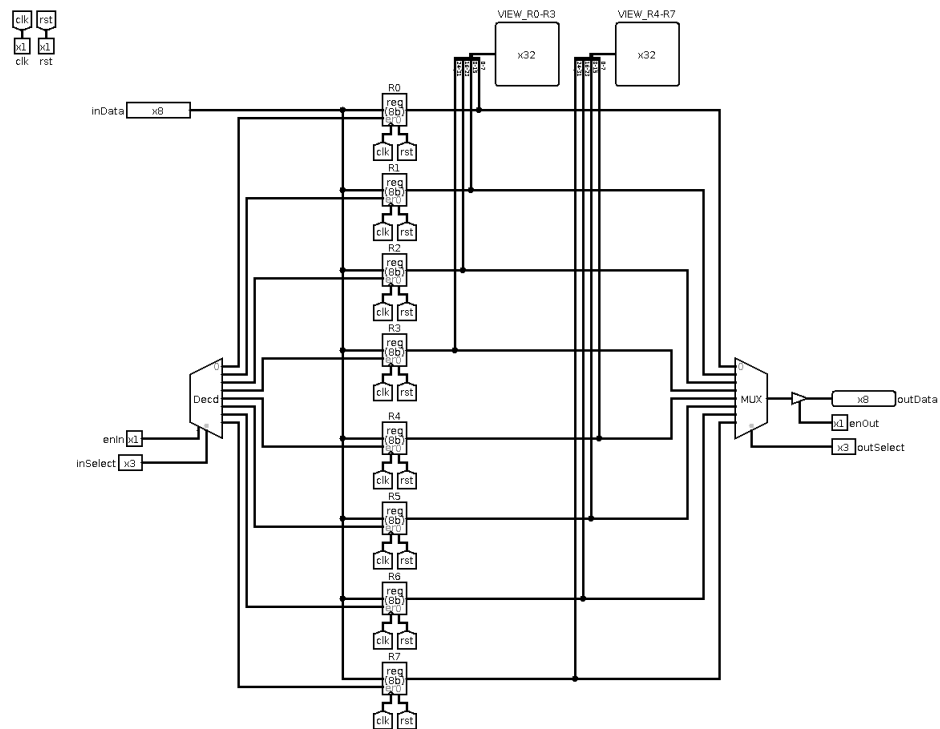
Notar que la señal `ALU_OP` es completada con un texto que indica la operación de la ALU a realizar.

## Uso

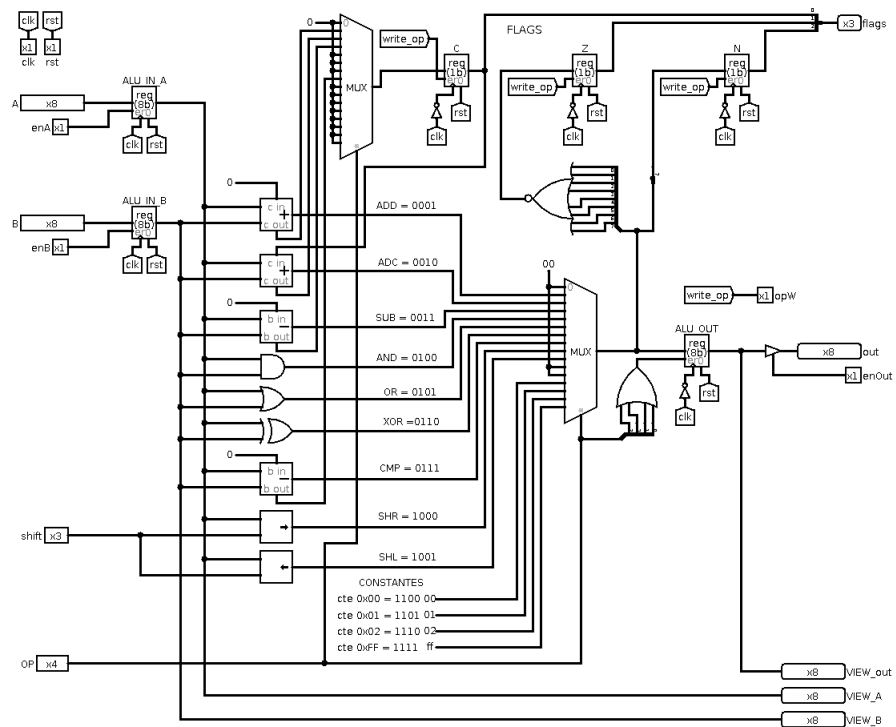
Una vez dentro de *Logisim* se debe cargar el circuito de `OrgaSmall`. Para cargar un programa, se debe entrar en el circuito `memory` y una vez seleccionada la memoria, utilizar el comando `load` para cargar un nuevo programa. Para modificar el código de micro-instrucciones, se debe entrar al circuito `controlUnit` y con el mismo procedimiento anterior, cargar el nuevo código de micro-instrucciones. Recordar que el PC comienza siempre en cero, y esta es la primera instrucción a ejecutar.

## Implementación en *Logisim*





Registros



Unidad Aritmético Lógica

