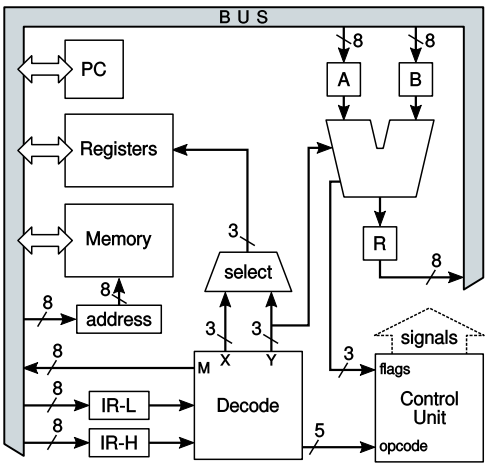
Taller de Lógica Digital

Organización del Computador 1

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

Las instrucciones están codificadas en 16 bits. Los primeros 5 bits indentifican el `opcode` de la instrucción, el resto de los bits indican los parámetros. Existen 4 posibles codificaciones de parámetros.

Caso	Codificación	Parámetros
A	`OOOOO XXXYYY-----`	`XXX` = Registro X, `XXX` = Registro Y o inmediato
B	`OOOOO XXX--------`	`XXX` = Registro X
C	`OOOOO ---MMMMMMMM`	`MMMMMMMM` = Dirección de memoria o Inmediato
D	`OOOOO XXXMMMMMMMM`	`XXX` = Registro X, `MMMMMMMM` = Dir. de memoria o Imm.

Considerando:

- `Rx` o `Ry`: Indices de registros, número entre `0` y `7`.
- `M`: Dirección de memoria o valor inmediato, número de 8 bits.
- `t`: Valor inmediato de desplazamiento, número entre `0` y `7`. Se codifica como `YYY`.
- En la columna de codificación, los bits indicados con `-` son reservados y deben valer cero.
- Las instrucciones de `opcode`: 9, 10, 11, 12, 13, 14, 15, 28, 29 y 30 son instrucciones reservadas.

Las instrucciones soportadas por la arquitectura son las siguientes:

Instrucción	Acción	Codificación
`ADD Rx, Ry`	`Rx` ← `Rx + Ry`	`00001 XXXYYY-----`
`ADC Rx, Ry`	`Rx` ← `Rx + Ry + flag_C`	`00010 XXXYYY-----`
`SUB Rx, Ry`	`Rx` ← `Rx - Ry`	`00011 XXXYYY-----`
`AND Rx, Ry`	`Rx` ← `Rx and Ry`	`00100 XXXYYY-----`
`OR Rx, Ry`	`Rx` ← `Rx or Ry`	`00101 XXXYYY-----`
`XOR Rx, Ry`	`Rx` ← `Rx xor Ry`	`00110 XXXYYY-----`
`CMP Rx, Ry`	Modifica *flags* de `Rx - Ry`	`00111 XXXYYY-----`
`MOV Rx, Ry`	`Rx` ← `Ry`	`01000 XXXYYY-----`
`STR [M], Rx`	`Mem[M]` ← `Rx`	`10000 XXXMMMMMMMM`
`LOAD Rx, [M]`	`Rx` ← `Mem[M]`	`10001 XXXMMMMMMMM`
`STR [Rx], Ry`	`Mem[Rx]` ← `Ry`	`10010 XXXYYY-----`
`LOAD Rx, [Ry]`	`Rx` ← `Mem[Ry]`	`10011 XXXYYY-----`
`JMP M`	`PC` ← `M`	`10100 ---MMMMMMMM`
`JC M`	Si `flag_C=1` entonces `PC` ← `M`	`10101 ---MMMMMMMM`
`JZ M`	Si `flag_Z=1` entonces `PC` ← `M`	`10110 ---MMMMMMMM`
`JN M`	Si `flag_N=1` entonces `PC` ← `M`	`10111 ---MMMMMMMM`
`INC Rx`	`Rx` ← `Rx + 1`	`11000 XXX--------`
`DEC Rx`	`Rx` ← `Rx - 1`	`11001 XXX--------`
`SHR Rx, t`	`Rx` ← `Rx << t`	`11010 XXXYYY-----`
`SHL Rx, t`	`Rx` ← `Rx >> t`	`11011 XXXYYY-----`
`SET Rx, M`	`Rx` ← `M`	`11111 XXXMMMMMMMM`

Componentes

La arquitectura está compuesta por 6 componentes interconectados. El circuito identificado como `microOrgaSmall` los integra en un `dataPath` sobre el lado izquierdo del mismo. El lado derecho presenta la visualización del estado de los registros.

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:

`Registers` (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`

`PC` (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 calor actual.
`PC_enOut(1)`	Habilita la salida del valor.

`ALU` (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 *flags*.
`ALU_OP(4)`	Indica la operación a realizar por la ALU.

Memory (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 *Opcode*.
flags(3)	Entrada de *flags*.
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. Está 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 secuenciador 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, donde las señales generadas 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 parámentros 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

(*) 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.

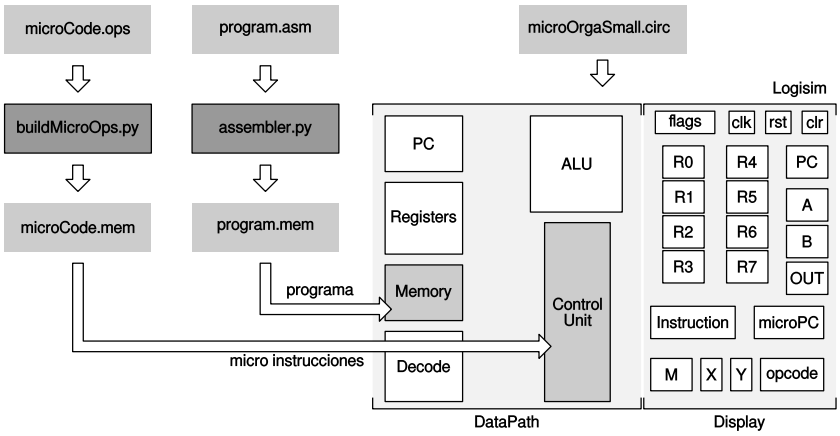
00	RB_enIn	08	ALU_enA	16	JC_microOp	24	DE_enOutImm
01	RB_enOut	09	ALU_enB	17	JZ_microOp	25	DE_loadL
02	RB_selectIndexIn	10	ALU_enOut	18	JN_microOp	26	DE_loadH
03	RB_selectIndexOut	11	ALU_opW	19	-	27	-
04	MM_enOut	12	ALU_OP	20	PC_load	28	-
05	MM_load	13	ALU_OP	21	PC_inc	29	-
06	MM_enAddr	14	ALU_OP	22	PC_enOut	30	load_microOp
07	-	15	ALU_OP	23	-	31	reset_microOp

La codificación de las operaciones de la unidad artimético lógica 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 descriptas 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 númerico 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 instruccón `ADD`:

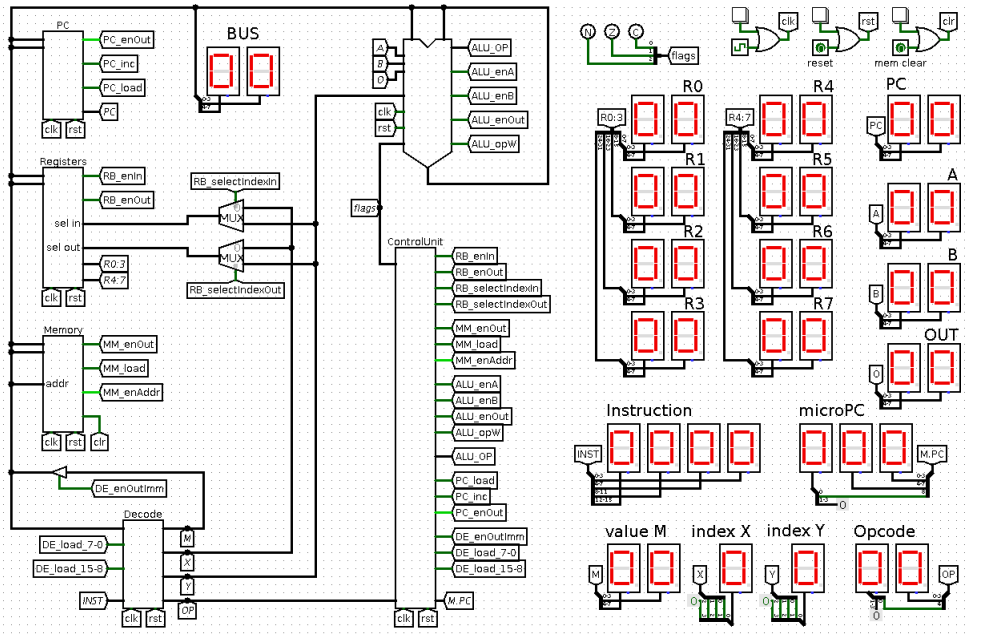
```
00001: ; ADD
    RB_enOut  ALU_enA  RB_selectIndexOut=0   ; ALU_A := Rx
    RB_enOut  ALU_enB  RB_selectIndexOut=1   ; ALU_B := Ry
    ALU_OP=ADD ALU_opW                       ; ALU_ADD
    RB_enIn   ALU_enOut RB_selectIndexIn=0   ; Rx := ALU_OUT
    reset_microOp
```

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

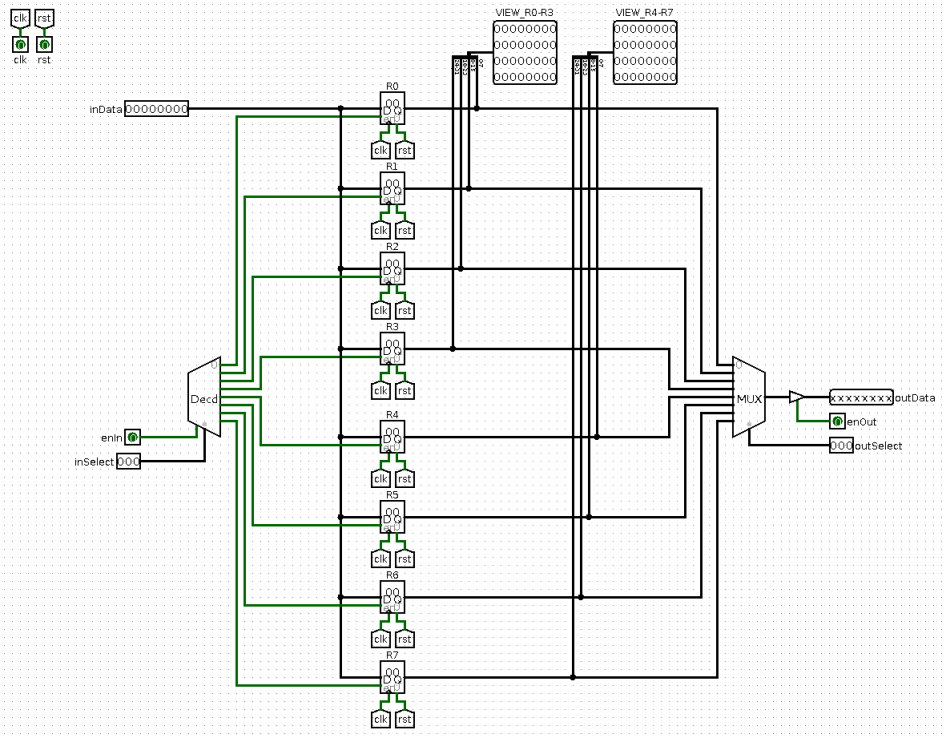
Uso

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.

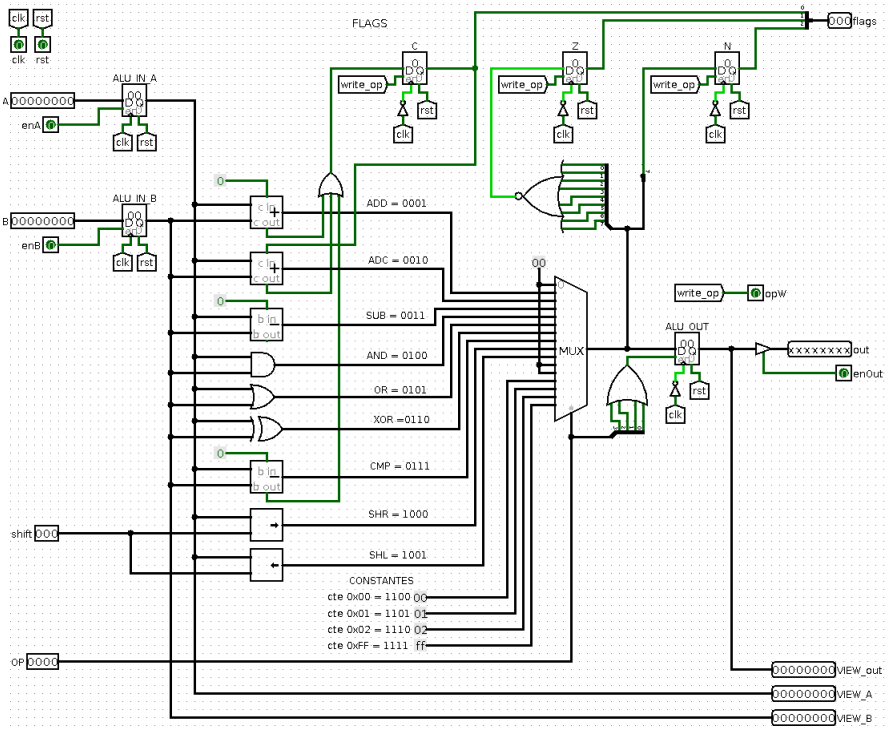
Implementación en *Logisim*



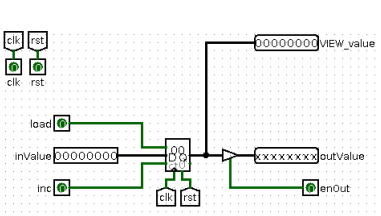
Data Path



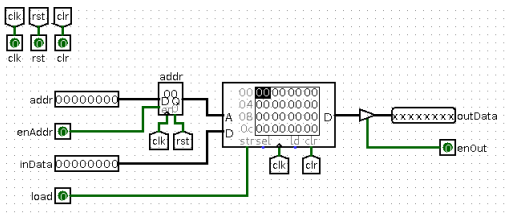
Registros



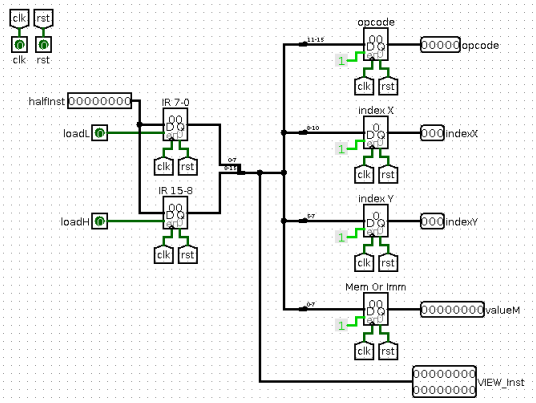
Unidad Aritmético Lógica



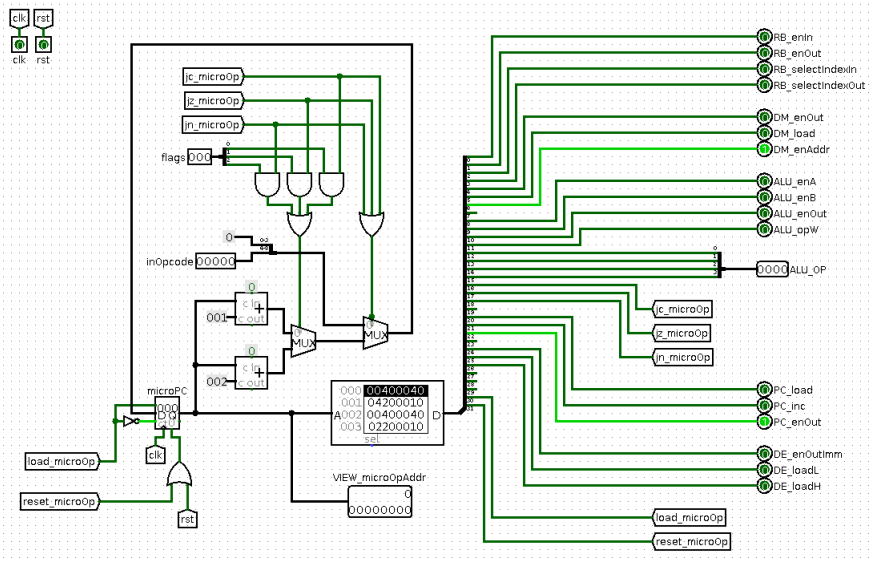
Contador de programa



Memoria de Instrucciones



Decodificador



Unidad de Control