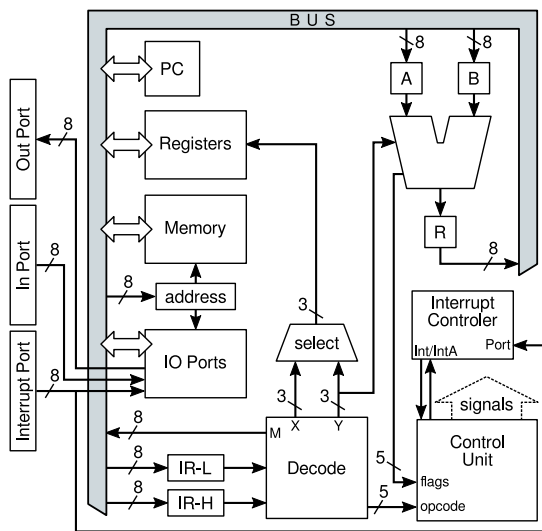


DataSheet: Sistema OrgaSmall

David Alejandro Gonzalez Marquez

Procesador OrgaSmall

OrgaSmall es un procesador diseñado e implementado con fines didácticos sobre la herramienta *Logisim*. Se cuenta además con una implementación en *verilog* denominada *OrgaSmallSystem*. El sistema 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.
- Soporta 3 puertos, 2 de entrada y 1 de salida mapeados a memoria.
- Una interrupción asociada a cambios sobre uno de los puertos de entrada.

Instrucciones

Las instrucciones están codificadas en 16 bits, es decir dos palabras de memoria. Los primeros 5 bits indentifican el **opcode** de la instrucción, el resto de los bits indican sus parámetros. Existen 4 posibles codificaciones de parámetros.

Caso	Codificación	Parámetros
A	00000 XXXYYY-----	XXX = Registro X, XXX = Registro Y o inmediato
B	00000 XXX-----	XXX = Registro X
C	00000 ---MMMMMMMM	MMMMMMMM = Dirección de memoria o Inmediato
D	00000 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 0 y 31 son instrucciones reservadas para *fetch* e interrupciones.
- La instrucción de opcode 15 es libre para codificar nuevas instrucciones.
- El operador | se utiliza para indentificar el registro usado como tope de la pila (ej. |Rx|).

Las instrucciones soportadas por la arquitectura son las siguientes:

Instrucción	Acción	Codificación
Reservada	Codifica el <i>Fetch</i> de instrucciones	00000 -----
ADD Rx, Ry	$Rx \leftarrow Rx + Ry$	00001 XXXYYY-----
ADC Rx, Ry	$Rx \leftarrow Rx + Ry + \text{Flag_C}$	00010 XXXYYY-----
SUB Rx, Ry	$Rx \leftarrow Rx - Ry$	00011 XXXYYY-----
AND Rx, Ry	$Rx \leftarrow Rx \text{ and } Ry$	00100 XXXYYY-----
OR Rx, Ry	$Rx \leftarrow Rx \text{ or } Ry$	00101 XXXYYY-----
XOR Rx, Ry	$Rx \leftarrow Rx \text{ xor } Ry$	00110 XXXYYY-----
CMP Rx, Ry	Modifica flags de $Rx - Ry$	00111 XXXYYY-----
MOV Rx, Ry	$Rx \leftarrow Ry$	01000 XXXYYY-----
PUSH Rx , Ry	$\text{Mem}[Rx] \leftarrow Ry$; $Rx \leftarrow Rx-1$	01001 XXXYYY-----
POP Rx , Ry	$Rx \leftarrow Rx+1$; $Ry \leftarrow \text{Mem}[Rx]$	01010 XXXYYY-----
CALL Rx , Ry	$\text{Mem}[Rx] \leftarrow PC$; $Rx \leftarrow Rx-1$; $PC \leftarrow Ry$	01011 XXXYYY-----
CALL Rx , M	$\text{Mem}[Rx] \leftarrow PC$; $Rx \leftarrow Rx-1$; $PC \leftarrow M$	01100 XXXMMMMMMMM
RET Rx	$Rx \leftarrow Rx+1$; $PC \leftarrow \text{Mem}[Rx]$	01101 XXX-----
RETI Rx	$Rx \leftarrow Rx+1$; $PC \leftarrow \text{Mem}[Rx]$; $Rx \leftarrow Rx+1$; $\text{Flags} \leftarrow \text{Mem}[Rx]$	01110 XXX-----
Libre		11111 -----
STR [M], Rx	$\text{Mem}[M] \leftarrow Rx$	10000 XXXMMMMMMMM
LOAD Rx, [M]	$Rx \leftarrow \text{Mem}[M]$	10001 XXXMMMMMMMM
STR [Rx], Ry	$\text{Mem}[Rx] \leftarrow Ry$	10010 XXXYYY-----
LOAD Rx, [Ry]	$Rx \leftarrow \text{Mem}[Ry]$	10011 XXXYYY-----
JMP M	$PC \leftarrow M$	10100 ---MMMMMMMM
JC M	Si $\text{flag_C}=1$ entonces $PC \leftarrow M$	10101 ---MMMMMMMM
JZ M	Si $\text{flag_Z}=1$ entonces $PC \leftarrow M$	10110 ---MMMMMMMM
JN M	Si $\text{flag_N}=1$ entonces $PC \leftarrow M$	10111 ---MMMMMMMM
JO M	Si $\text{flag_O}=1$ entonces $PC \leftarrow M$	11000 ---MMMMMMMM
SHRA Rx, t	$Rx \leftarrow Rx \ggg t$	11001 XXXYYY-----
SHR Rx, t	$Rx \leftarrow Rx \gg t$	11010 XXXYYY-----
SHL Rx, t	$Rx \leftarrow Rx \ll t$	11011 XXXYYY-----
READF Rx	$Rx \leftarrow \text{Flags}$	11100 XXX-----
LOADF Rx	$\text{Flags} \leftarrow Rx$	11101 XXX-----
SET Rx, M	$Rx \leftarrow M$	11111 XXXMMMMMMMM
Reservada	Codifica el llamado a interrupciones	11110 -----

Las intrucciones PUSH, POP, CALL, RET y RETI son las unicas que operan con la pila, esta es direccionada desde un registro pasado por parámetro. El registro utilizado como tope de la pila, apunta a la primer dirección libre en la pila.

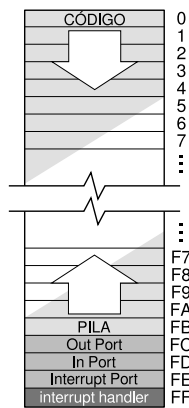
Si bien es posible utilizar cualquier registro como pila, el mecanismo de interrupciones utiliza siempre el registro R7 como pila.

La palabra de estado es almacenada en la **ALU**, permitiendo ser modificada mediante dos operaciones específicas. El orden de los bits de la palabra de estado es el siguiente: 000I 0NZC (desde más significativo a menos significativo). Donde **I** es el *flag* de habilitación de interrupciones, **0** el bit *overflow*, **N** el bit de *negative*, **Z** el bit de *zero* y **C** el bit de *carry*.

Puertos de Entrada-Salida e Interrupciones

El sistema cuenta con 3 puertos de 8 bits, mapeados a memoria en las siguientes direcciones:

- 0xFC : OutPort : Puerto de salida.
- 0xFD : InPort : Puerto de entrada.
- 0xFE : InterruptPort : Puerto de entrada sensible a interrupciones.



El puerto **InterruptPort** genera una interrupción por cada cambio de estado en alguno de sus bits, siempre que el *flag* de habilitación de interrupciones este activado. Una vez que se detecta la interrupción el sistema ejecuta la rutina indicada en la dirección 0xFF de memoria. Para esto, se carga en la pila los *flags*, luego el *PC*, se inihiven las interrupciones modificando el *flag* de interrupciones y por último se salta a la rutina de atención de interrupciones. Para todo este proceso, se utiliza como *stack pointer* el registro R7 que en el caso general se debe cargar en 0xFB.

Componentes

La arquitectura está compuesta por 8 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), **IOPorts** (Puertos de Entrada/Salida), **Decode** (Decodificador de Instrucciones), **ControlUnit** (Unidad de Control) y **InterruptControl** (Controlador de Interrupciones)

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
RB.selectSP(1)	Selecciona el índice del registro R7 para Stack Pointer
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 valor actual.
PC.enOut(1)	Habilita la salida del valor.

ALU (Unidad Aritmético Lógica)	
A(8), B(8) y out(8)	Entradas y salida de la ALU.
shift(3)	Indica la cantidad de bits para el operador <i>shift</i>
ALU_OP(4)	Indica la operación a realizar por 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> .
flags(5)	Salida de <i>flags</i> almacenados por la ALU.
Memory (Memoria)	
inData(8) y outData(8)	Entrada y salida de datos.
addr(8)	Dirección de memoria donde leer.
MM_enOut(1)	Habilitación de la salida.
MM_load(1)	Indica si leer o escribir.
MM_enAddr(1)	Habilita cargar la dirección.
addrOut(8)	Salida del registro de dirección.
IOports (Puertos de Entrada/Salida)	
inData(8) y outData(8)	Entrada y salida de datos.
addr(8)	Entrada de direcciones (conectada a addrOut).
enOut(1)	Habilitación de la salida (conectada a MM_enOut).
load(1)	Indica si leer o escribir (conectada a MM_load).
portOutput(8)	Puerto de salida.
portInput(8)	Puerto de entrada.
portInterrupt(8)	Puerto de interrupciones.
Decode (Decodificador de Instrucciones)	
halfInst(8)	Entrada de datos (media instrucción)
DE_loadL(1) y DE_loadH(1)	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.
InterruptControl (Controlador de Interrupciones)	
portInterrupt(8)	Entrada puerto de interrupciones.
INT_Ack(1)	Señal de interrupción resuelta.
INT_Req(1)	Señal de interrupción.
ControlUnit (Unidad de Control)	
inOpcode(5)	Entrada de <i>Opcode</i> .
flags(5)	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.
RB_selectSP(1)	Señal de selección del registro para Stack Pointer.
MM_enOut(1) y MM_load(1)	Señales para la Memoria de Datos.
MM_enAddr(1)	Habilita escribir el registro de dirección de la memoria.
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.
DE_enOutImm(1)	Habilita la entrada al bus de un valor inmediato.
DE_loadL(1) y DE_loadH(1)	Selecciona la parte de la instrucción a decodificar.
INT_Req(1) y INT_Ack(1)	Señales del controlador de interrupciones.

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).

El sistema está diseñado respetando las siguientes decisiones:

- Los puertos de entrada/salida están mapeados a memoria, para implementar este funcionamiento la memoria almacena la dirección que leer o escribir y reporta a los puertos de entrada/salida dicha dirección por medio de la señal **addrOut**. Las señales de control **enOut** y **load** llegan tanto a la memoria como a los puertos de entrada/salida. Dentro de cada uno se decide si la dirección a leer o escribir está en el rango de memoria o de alguno de los puertos.
- El banco de registros permite seleccionar un registro donde escribir y un registro donde leer al mismo tiempo por medio de dos entradas para tal fin. Las instrucciones codifican los índices **x** e **y**. Estos pueden ser utilizados tanto como entrada o como salida en el banco de registros por medio de las señales de selección **RB_inSelect** y **RB_outSelect**.
- El resultado de la decodificación de una instrucción no es escrito directamente en el bus, en cambio, las cuatro señales generadas se conectan directamente a distintas unidades del sistema. Solo el campo **M** puede ser escrito sobre el bus, mediante la señal **DE_enOutImm**, esto permite codificar instrucciones con constantes.

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 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 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	JO
00001xxxx	ADD	01001xxxx	PUSH	10001xxxx	LOAD	11001xxxx	SHRA
00010xxxx	ADC	01010xxxx	POP	10010xxxx	STR*	11010xxxx	SHR
00011xxxx	SUB	01011xxxx	CALL	10011xxxx	LOAD*	11011xxxx	SHL
00100xxxx	AND	01100xxxx	CALL*	10100xxxx	JMP	11100xxxx	READF
00101xxxx	OR	01101xxxx	RET	10101xxxx	JC	11101xxxx	LOADF
00110xxxx	XOR	01110xxxx	RETI	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.

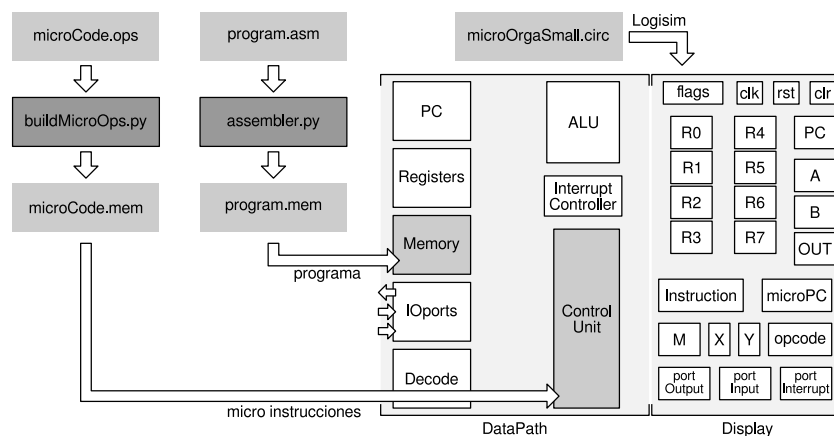
00	RB_enIn	08	ALU_enA	16	JC_microOp	24	DE_enOutImm
01	RB_enOut	09	ALU_enB	17	JZ_microOp	25	DE_loadL
02	RB_selectIndexIn	10	ALU_enOut	18	JN_microOp	26	DE_loadH
03	RB_selectIndexOut	11	ALU_opW	19	JO_microOp	27	INT_ack
04	RB_selectSP	12	ALU_OP	20	PC_load	28	-
05	MM_enOut	13	ALU_OP	21	PC_inc	29	load_Int_microOp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

La codificación de las operaciones de la unidad aritmético 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	READ_FLAGS	1110	cte0x02
0011	SUB	0111	SHRA	1011	LOAD_FLAGS	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
    ALU_enA RB_enOut RB_selectIndexOut=0 ; A <- Rx
    ALU_enB RB_enOut RB_selectIndexOut=1 ; B <- Ry
    ALU_OP=ADD ALU_opW                ; ALU_ADD
    RB_enIn RB_selectIndexIn=0 ALU_enOut ; 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*

