¿Qué precisamos para modelar/implementar/ejecutar los fenómenos observables?
Representación de la información , una forma de representar los datos con los que vamos a trabajar (números)
Conservación de los datos , un mecanismo que conserve el valor de nuestros datos, salvo que indiquemos lo contrario, a través del tiempo (memorias)
Modelo de cómputo , una definición de cómo operar con la información representada (set de instrucciones, microoperaciones)

¿Qué precisamos para modelar/implementar/ejecutar los fenómenos observables?
Operaciones aritmético/lógicas , una implementación de operaciones aritmeticas y logicas basicas sobre nuestra información representada (ALU)
Cómputo en base a valores previos , una forma de resolver operaciones complejas a partir de ejecutar operaciones sencillas (atómicas) en una serie de pasos (circuitos secuenciales, microprogramación)

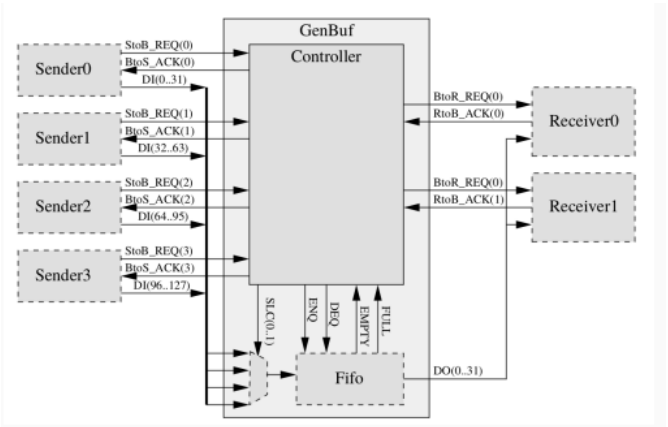
Para poder computar precisamos: **Representar** , **Conservar** , **Operar**.

Para poder computar precisamos: **Enteros acotados** , **Memorias** , **ALU**.

Diseño e implementación de hardware

Al igual que el software, la práctica del diseño y la implementación del hardware depende del siguiente grupo de actividades:
Diseño → Especificación → Implementación → Validación → Verificación

Del diseño a la implementación tenemos:
Diseño Descripción de arquitectura, división por vistas del sistema



Especificación Representación declarativa del comportamiento esperado

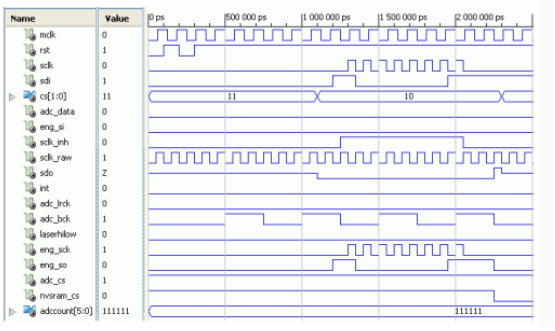
G1	$\forall i : \texttt{always}\ (\text{StoB\_REQ}(i) \rightarrow \texttt{eventually!}\ \text{BtoS\_ACK}(i))$ $\forall i : \texttt{always}\ (\neg\text{StoB\_REQ}(i) \rightarrow \texttt{eventually!}\ \neg\text{BtoS\_ACK}(i))$
G2	$\forall i : \texttt{always}\ (\texttt{rose}(\text{StoB\_REQ}(i)) \rightarrow \neg\text{BtoS\_ACK}(i))$
G3	$\forall i : \texttt{always}\ (\texttt{rose}(\text{BtoS\_ACK}(i)) \rightarrow \texttt{prev}(\text{StoB\_REQ}(i)))$
G4	$\forall i : \texttt{always}\ ((\text{BtoS\_ACK}(i) \wedge \text{StoB\_REQ}(i)) \rightarrow \texttt{next!}\ \text{BtoS\_ACK}(i))$
A1	$\forall i : \texttt{always}\ (\text{StoB\_REQ}(i) \wedge \neg\text{BtoS\_ACK}(i) \rightarrow \texttt{next!}\ \text{StoB\_REQ}(i))$ $\forall i : \texttt{always}\ (\text{BtoS\_ACK}(i) \rightarrow \texttt{next!}\ \neg\text{StoB\_REQ}(i))$
G5	$\forall i \forall i' \neq i : \texttt{always}\ \neg(\text{BtoS\_ACK}(i) \wedge \text{BtoS\_ACK}(i'))$
A2	$\forall j : \texttt{always}\ (\text{BtoR\_REQ}(j) \rightarrow \texttt{eventually!}\ \text{RtoB\_ACK}(j))$ $\forall j : \texttt{always}\ (\neg\text{BtoR\_REQ}(j) \rightarrow \texttt{next!}\ \neg\text{RtoB\_ACK}(j))$
A3	$\forall j : \texttt{always}\ (\text{BtoR\_REQ}(j) \wedge \text{RtoB\_ACK}(j) \rightarrow \texttt{next!}\ \text{RtoB\_ACK}(j))$
A4	$\forall j : \texttt{always}\ (\text{RtoB\_ACK}(j) \rightarrow \texttt{prev}(\text{BtoR\_REQ}(j)))$

Implementación Interpretación imperativa de la especificación que permite producir una instancia operacional del sistema

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

Y en las etapas post implementativas:

Validación Conjunto de pruebas no exhaustivas que prueban el comportamiento de la especificación y/o implementación.



Verificación Prueba de propiedades formales (con garantías basadas en algún mecanismo matemático) de la especificación y/o implementación.

```
import "DwyerPatterns.spectra"

module ElevatorUnrealizable

type Floors = Int(0..3);
sys Floors elevatorLocation;
env Floors request;

gar startOnGroundFloor:
  elevatorLocation=0;

gar moveOneFloorAtATime:
  G (next(elevatorLocation) = elevatorLocation+1) |
     (next(elevatorLocation) = elevatorLocation-1);

gar eventuallyHandleOpenRequests:
  pRespondsToS(request != elevatorLocation,
                              request=elevatorLocation);
```

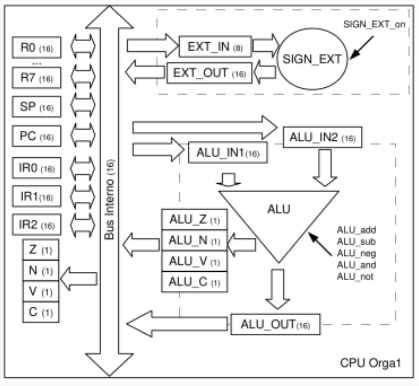
Conceptos principales

Razonamiento composicional describe la práctica de dividir el funcionamiento del sistema en componentes que se encargan de una parte menor del comportamiento
Abstracción es el proceso a través del cual ocultamos detalles de un sistema o componente de acuerdo a las necesidades de la tarea u operación en curso
Diseño por contratos describe la forma en la que vinculamos componentes a partir de derechos de los que el componente goza y obligaciones que debe cumplir.

Arquitectura del microprocesador

Recordemos que queremos diseñar e implementar un modelo de cómputo (microprocesador). Para el alcance de la materia vamos a interpretar a la arquitectura del mismo como:
Diseño de sistema + Set de instrucciones → Microarquitectura

Diseño de sistema Elección de arquitectura y componentes involucrados.



Set de instrucciones Semántica de las capacidades de ejecución de nuestro microprocesador

Formato de instrucción

***Tipo 1*: Instrucciones de dos operandos**

4 bits	*6 bits*	*6 bits*	*16 bits*	*16 bits*
cod. op.	destino	fuente	constante destino (opcional)	constante fuente (opcional)

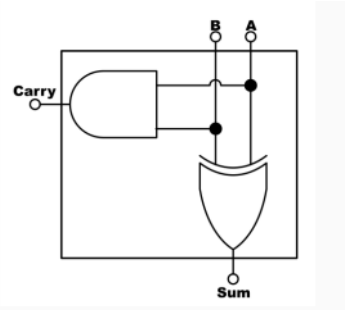
operación	cod. op.	efecto
MOV d, f	0001	$d \leftarrow f$
ADD d, f	0010	$d \leftarrow d + f$ (suma binaria)
SUB d, f	0011	$d \leftarrow d - f$ (resta binaria)
AND d, f	0100	$d \leftarrow d$ and f
OR d, f	0101	$d \leftarrow d$ or f
CMP d, f	0110	Modifica los *flags* según el resultado de $d - f$ (resta binaria)
ADDC d, f	1101	$d \leftarrow d + f +$ *carry* (suma binaria)

Microarquitectura Implementación de nuestro set de instrucciones a través de la lógica de control de la arquitectura y componentes seleccionados.

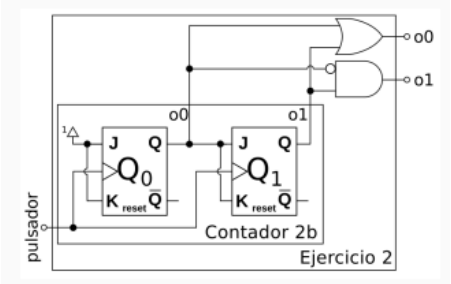
- MOV R5,R1
  1. R5 := R1
- AND R7, R1
  1. ALU_IN1 := R7
  2. ALU_IN2 := R1
  3. ALU_{and}
  4. R7 := ALU_OUT
  5. Z := ALU_Z
  6. N := ALU_N
  7. C := ALU_C
  8. V := ALU_V
- JE 0xFF
  1. IF Z = 1
  2. EXT_IN := IR0[7:0]
  3. SIGN_EXT_on
  4. ALU_IN_1 := PC
  5. ALU_IN_2 := EXT_OUT
  6. ALU_add
  7. PC := ALU_OUT

Los contenidos que vamos a presentar durante las clases prácticas son los siguientes:
Lógica Combinatoria y Secuencial Diseño de un set de instrucciones
Microprogramación
Manejo de memoria, Interrupciones, Caché Y Buses

Lógica Combinatoria presenta los principios fundamentales para construir circuitos que implementen en un soporte electrónico la semántica de la lógica proposicional

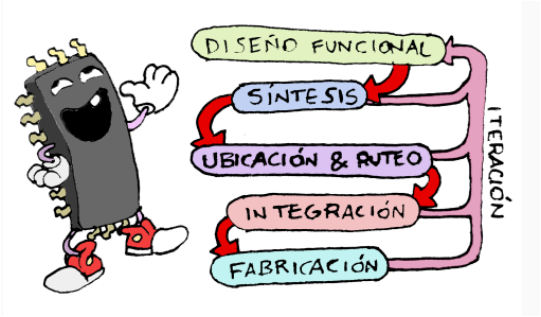


Lógica Secuencial introduce los elementos básicos para mantener el valor de un dato a lo largo del tiempo, los mecanismos de sincronización de circuitos y junto con estos la capacidad y técnicas que nos permiten descomponer e implementar un cómputo complejo en una secuencia de pasos atómicos.



El diseño de un set de instrucciones es el paso necesario para implementar nuestro microprocesador, donde vamos a definir el nombre y significado de las operaciones que deseamos poder ejecutar como parte de nuestros programas.
La microprogramación va describir la forma en que nuestros componentes sincrónicos interactúan para implementar las operaciones descritas en el set de instrucciones previamente definido.

Fabricación del hardware



Hardware description language (HDL)
Del diseño a la implementación tenemos:
Un lenguaje de descripción de hardware, o HDL por sus siglas en inglés (hardware description language) es un lenguaje que describe la estructura y el comportamiento de un circuito digital. Los dos lenguajes más utilizados en la industria son VHDL y Verilog. En Orga 1 vamos a usar **VHDL**.

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

Estructura de un documento VHDL

Un documento VHDL va estar dividido en una descripción de interfaz y otra de comportamiento. La descripción de interfaz se define como **entity**, da un nombre al componente y enumera los tipos de señales expuestas, indicando su tamaño y si se trata de una entrada o una salida.

```
entity signed_adder is
  port
  (
    aclr : in   std_logic;
    clk  : in   std_logic;
    a    : in   std_logic_vector;
    b    : in   std_logic_vector;
    q    : out  std_logic_vector
  );
end signed_adder;
```

La descripción de comportamiento se define como architecture y define el vínculo funcional y/o lógico entre las señales de entrada, salida y cualquier elemento de representación interna, que se puede definir de forma estática o, por ej., en base al evento de flanco de reloj.

```
architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert(a'length >= b'length)
    report "Port A must be the longer vector if different siz
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```