Introducción *Práctica 1*

Simular el procesador RISC-V en versión uniciclo y segmentado

Curso 2022-2023

Arquitectura de Computadores

3º de grado en Ingeniería Informática y

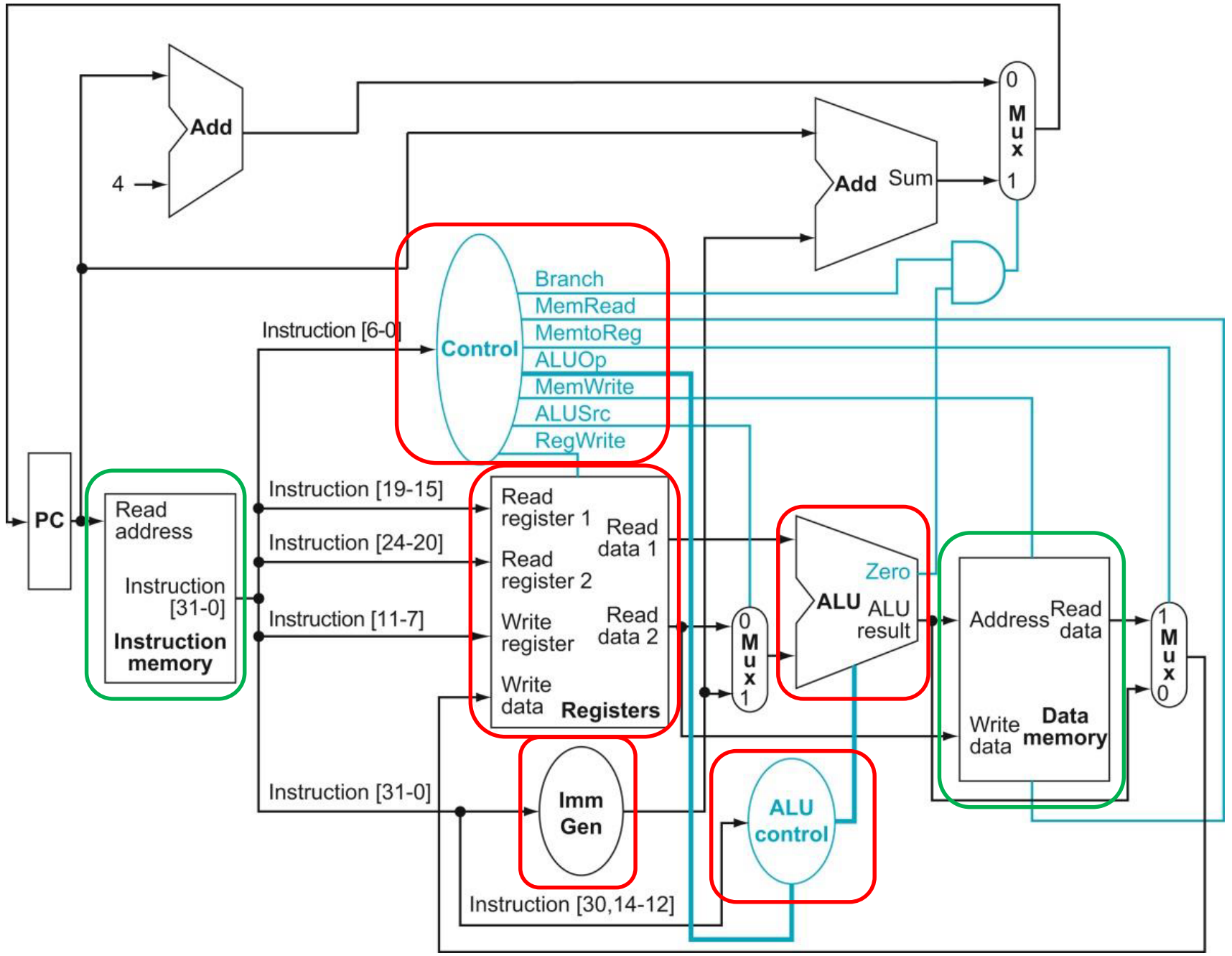
3º de doble grado en Ing. Informática y Matemáticas

Generalidades

- Simular el procesador RISC-V en versión uniciclo/segmentado
- Simulador Siemens EDA (Mentor) QuestaSim
- Entorno Linux
- Ensamblar código con RARS

- Opcional: para editar VHDL usar Visual Studio Code / Xilinx Vivado / Notepad++ / Atom / Sublime / Vim /…
- En casa: Versión gratuita (**Intel FPGA**) de QuestaSim o Vivado Simulator o Máquina Virtual provista por la asignatura

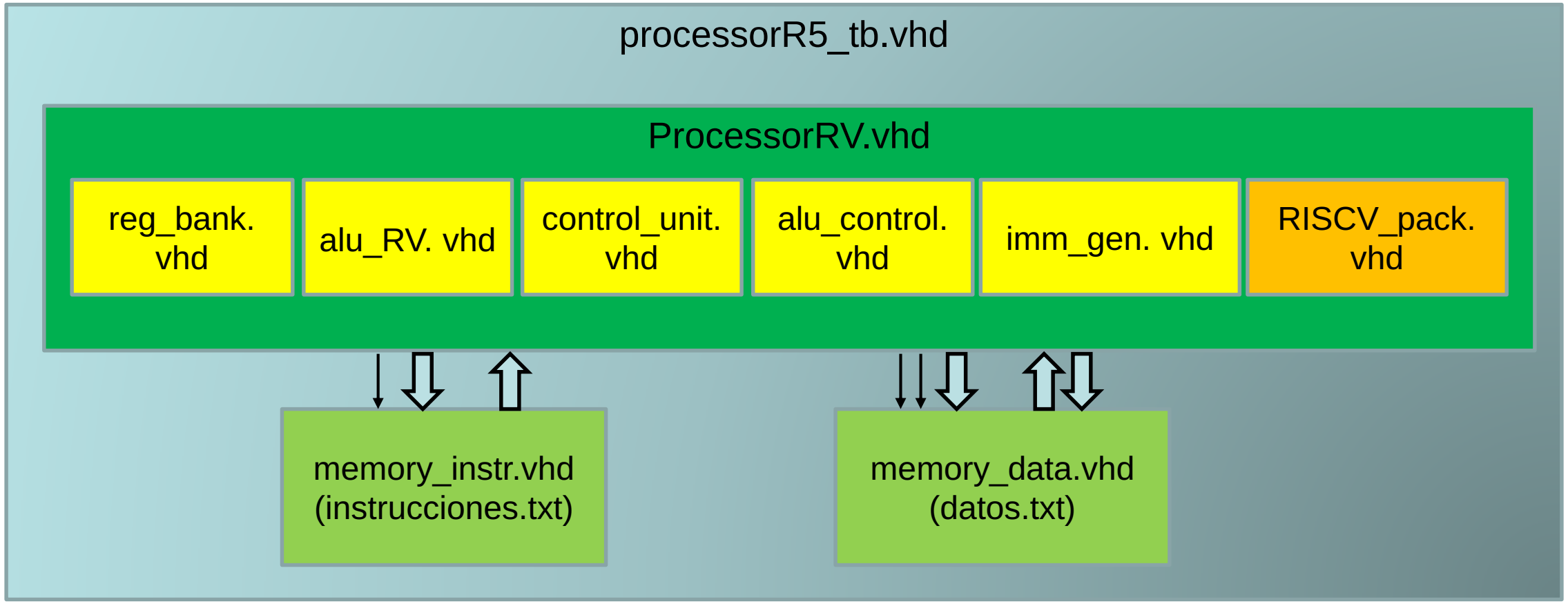
RISC-V uniciclo



Entidades externas al procesador

Entidades ya desarrolladas

Diseño Jerárquico



Módulos que se proveen y han de ser desarrollados

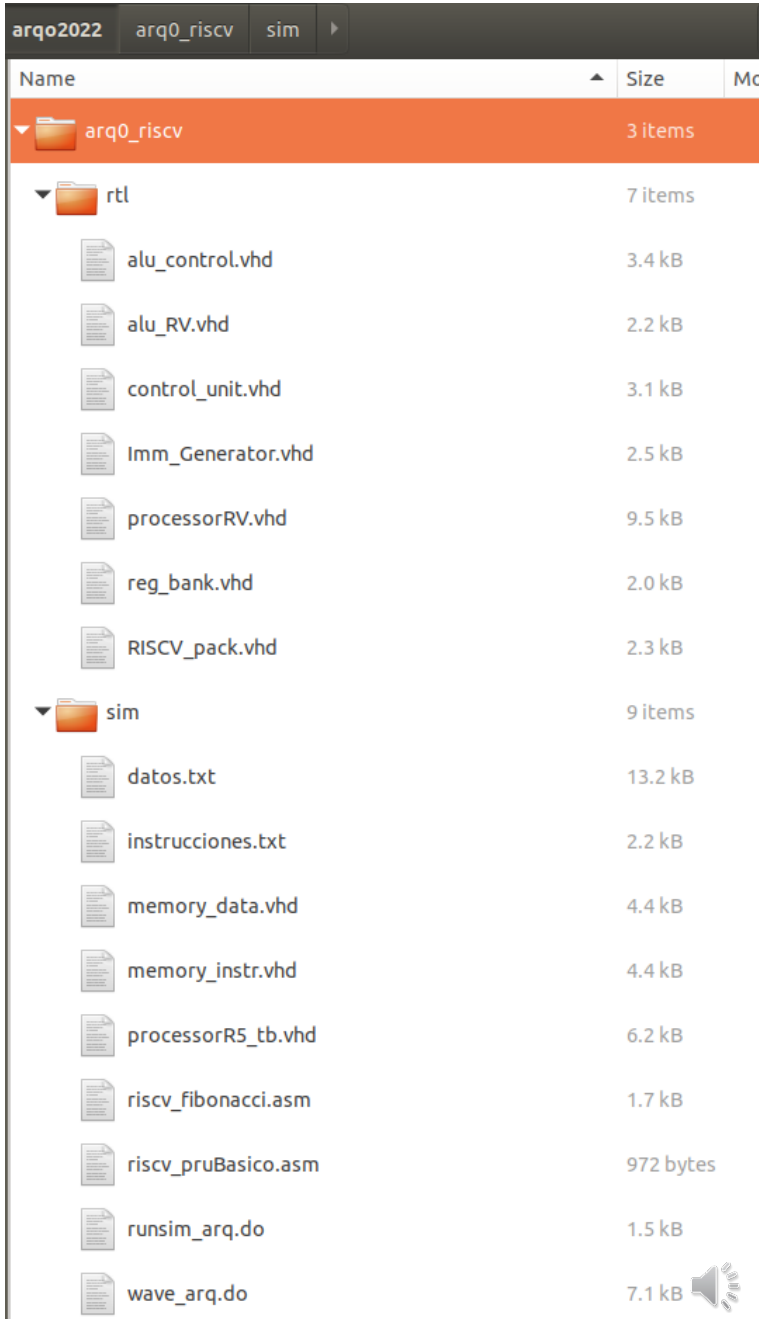
Material Entregado

- directorio ***rtl/*** : contiene el código del procesador

processorRV.vhd	**procesador, versión simple uniciclo**
alu_RV.vhd	ALU para RISC-V, completa
reg_bank.vhd	Banco de registros, completo
control_unit.vhd	Unidad de control, completo
alu_control.vhd	Control de la ALU, completo
imm_gen.vhd	Generador de inmediato, completo
RISCV_pack.vhd	Package con definiciones comunes

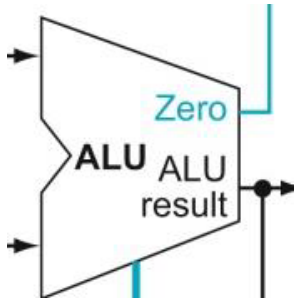
- directorio ***sim/*** : contiene lo necesario para simular

processorR5_tb.vhd	**El banco de pruebas (Testbench)**
datos.txt	Fichero de datos para la memoria de datos
Instrucciones.txt	Fichero de datos para la memoria de instrucciones
memory_datos.vhd	Modelo simple de memoria síncrona de datos
memory_instr.vhd	Modelo simple de memoria síncrona de instrucciones
riscv_pruBasico.asm	Código fuente de un programa ensamblador de prueba
riscv_Fibonacci.asm	Código fuente de un programa ensamblador de prueba
programa_test.asm	Código fuente de un programa ensamblador de prueba
runsim_arq.do	Script de simulación para ModelSim
wave.do	Script de configuración de ondas paras ModelSim



ALU (Arithmetic Logic Unit)

- Debe ser capaz de ejecutar el conjunto de instrucciones de esta versión reducida de RISC-V que se implementarán
- Las señales de control las general el bloque combinacional **“ALU CONTROL”**



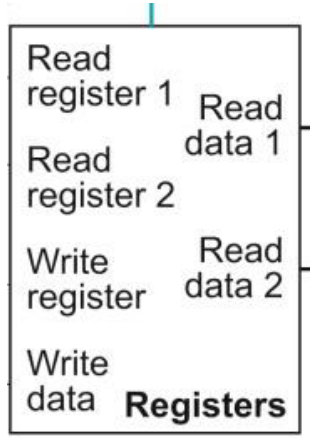
Ya desarrollado
(alu_RV.vhd)

```
entity alu_RV is
    port (
        OpA     : in  std_logic_vector (31 downto 0); -- Operando A
        OpB     : in  std_logic_vector (31 downto 0); -- Operando B
        Control : in  std_logic_vector ( 3 downto 0); -- Codigo de control=op. a ejecutar
        Result  : out std_logic_vector (31 downto 0); -- Resultado
        SignFlag: out std_logic;                      -- Sign Flag
        carryOut: out std_logic;                      -- Carry bit
        ZFlag   : out std_logic                       -- Flag Z
    );
end alu_RV;
```

Banco de registros

- **32 registros**
- **Lectura asíncrona**
- **Escritura síncrona**
  - Flanco de subida (en P1)
  - El resto de los registros del procesador siempre en flanco de subida
- **Registro 0**
  - Siempre vale 0
  - Escrituras sin efecto

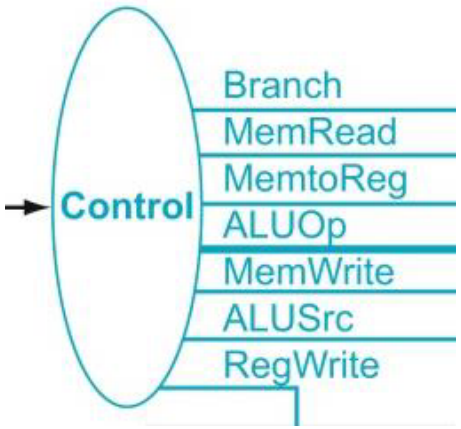
Ya desarrollado (reg_bank.vhd)




```
entity reg_bank is
   port (
      Clk   : in std_logic; -- Reloj activo en flanco de subida
      Reset : in std_logic; -- Reset asíncrono a nivel alto
      A1    : in std_logic_vector(4 downto 0);   -- Dirección para el puerto Rd1
      Rd1   : out std_logic_vector(31 downto 0); -- Dato del puerto Rd1
      A2    : in std_logic_vector(4 downto 0);   -- Dirección para el puerto Rd2
      Rd2   : out std_logic_vector(31 downto 0); -- Dato del puerto Rd2
      A3    : in std_logic_vector(4 downto 0);   -- Dirección para el puerto Wd3
      Wd3   : in std_logic_vector(31 downto 0);  -- Dato de entrada Wd3
      We3   : in std_logic -- Habilitación de la escritura de Wd3
   );
end reg_bank;
```

Unidad de control

- Genera las señales de control desde el código de operación
  - Entra el código de operación (OpCode) de la instrucción
  - Salen las señales de control: Branch, ResultScr, MemRead, MemWrite, AluScr, AluOP, ins_jalr, RegWrite

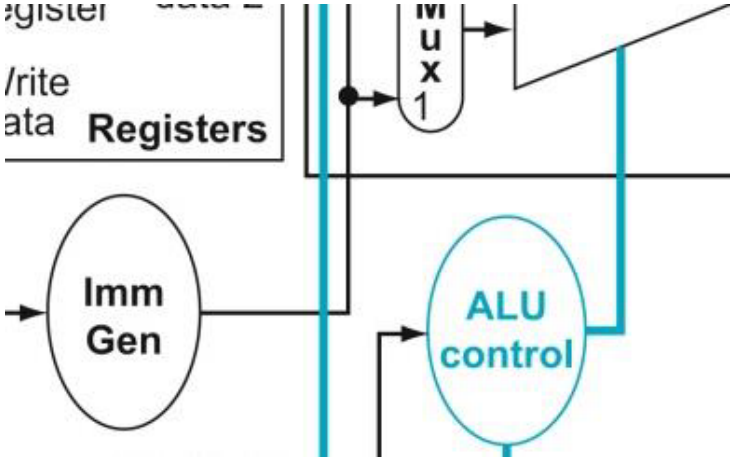


Ya desarrollado (control_unit.vhd)

```
entity control_unit is
   port (
      -- Entrada = codigo de operacion en la instruccion:
      OpCode  : in  std_logic_vector (6 downto 0);
      -- Seniales para el PC
      Branch : out  std_logic; -- 1 = Ejecutandose instruccion branch
      -- Seniales relativas a la memoria
      ResultSrc: out  std_logic_vector(1 downto 0); -- 00 salida Alu; 01 = salida de la mem.; 10 PC
      MemWrite : out  std_logic; -- Escribir la memoria
      MemRead  : out  std_logic; -- Leer la memoria
      -- Seniales para la ALU
      ALUSrc   : out  std_logic;                    -- 0 = oper.B es registro, 1 = es valor inm.
      AuipcLui : out  std_logic_vector (1 downto 0); -- 0 = PC. 1 = zeros, 2 = reg1.
      ALUOp    : out  std_logic_vector (2 downto 0); -- Tipo operacion para control de la ALU
      -- señal generacion salto
      Ins_jalr  : out  std_logic;  -- 0=any instrucion, 1=jalr
      -- Seniales para el GPR
      RegWrite : out  std_logic    -- 1=Escribir registro
   );
end control_unit;
```

ALU control e Immediate Generator

- alu_control: Genera la operación efectiva a realizar por la ALU
- imm_gen: genera el operando inmediato en caso de ser necesario




```vhdl
entity alu_control is
    port (
        -- Entradas:
        ALUOp  : in std_logic_vector (2 downto 0); -- Codigo de control desde la unidad de control
        Funct7 : in std_logic_vector (6 downto 0); -- Campo "funct7" de la instruccion (I(31:25))
        Funct3 : in std_logic_vector (2 downto 0); -- Campo "funct3" de la instruccion (I(14:12))
        -- Salida de control para la ALU:
        ALUControl : out std_logic_vector (3 downto 0) -- Define operacion a ejecutar por la ALU
    );
end alu_control;
```

```vhdl
entity Imm_Gen is
    port (
        instr     : in std_logic_vector(31 downto 0);
        imm       : out std_logic_vector(31 downto 0)
    );
end entity Imm_Gen;
```

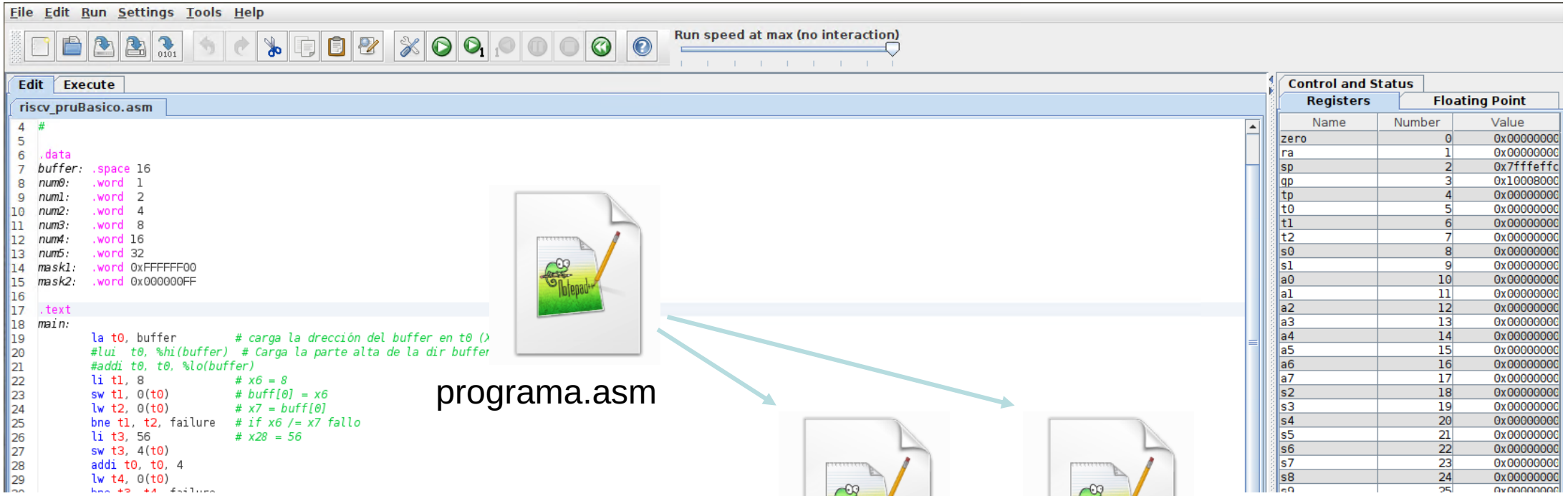
Ya desarrollados
(alu_control.vhd)
(imm_gen)

Simular el procesador

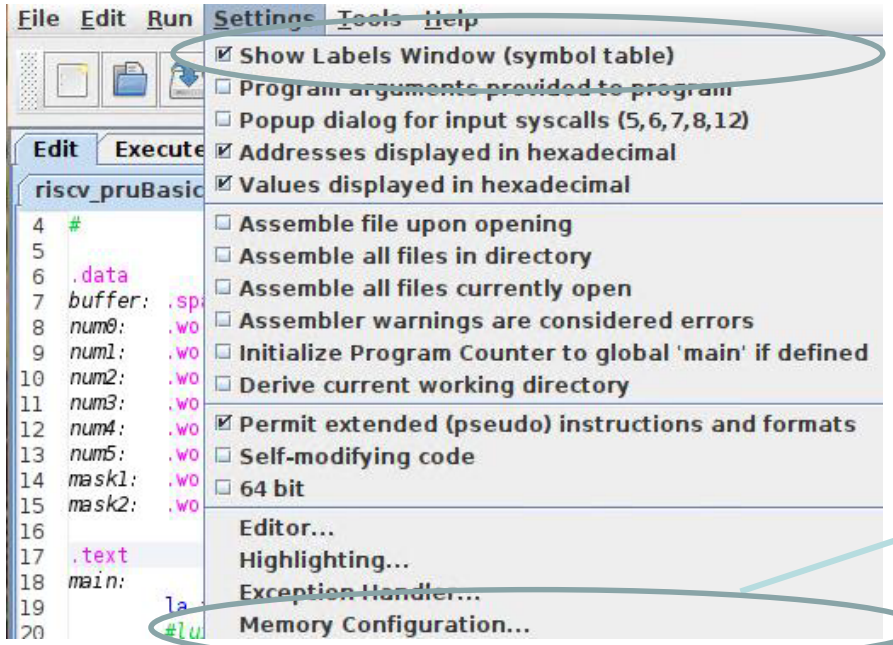
- Se utilizará un simulador HDL
  - En los laboratorios QuestaSim en Linux
  - Fuera de la EPS, QuestaSim (Intel FPGA) o Máquina Virtual
- Para simular el sistema es necesario
  - Código del procesador (y sus componentes)
  - Testbench (procesorRV_tb.vhd) y modelos de las memorias de instrucciones y datos (memory_instr.vhd y memory_data.vhd)
  - Contenido de las memorias (ficheros de texto plano "instrucciones.txt" y "datos.txt"). Ver como generar a continuación.

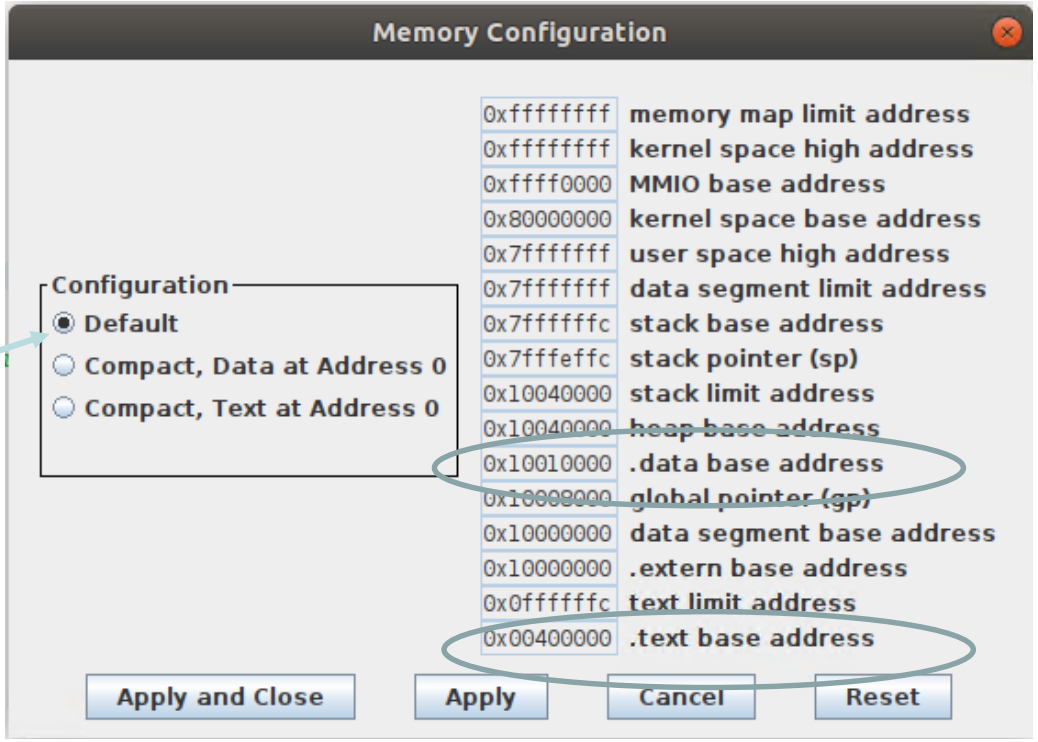
Simulación: generar instrucciones y datos

En Linux (o Windows) también utilizaremos el simulador ***RARS*** *(RISC-V Assembler and Runtime Simulator)* para generar el contenido de las memorias



RARS como ensamblador y compilador





- Seleccionar ver las etiquetas (show labels)
- *Dejar la configuración de memoria por defecto:*
  Esto es datos (.data) en dirección 0x1001_0000 y código (.text) en dirección 0x0040_0000