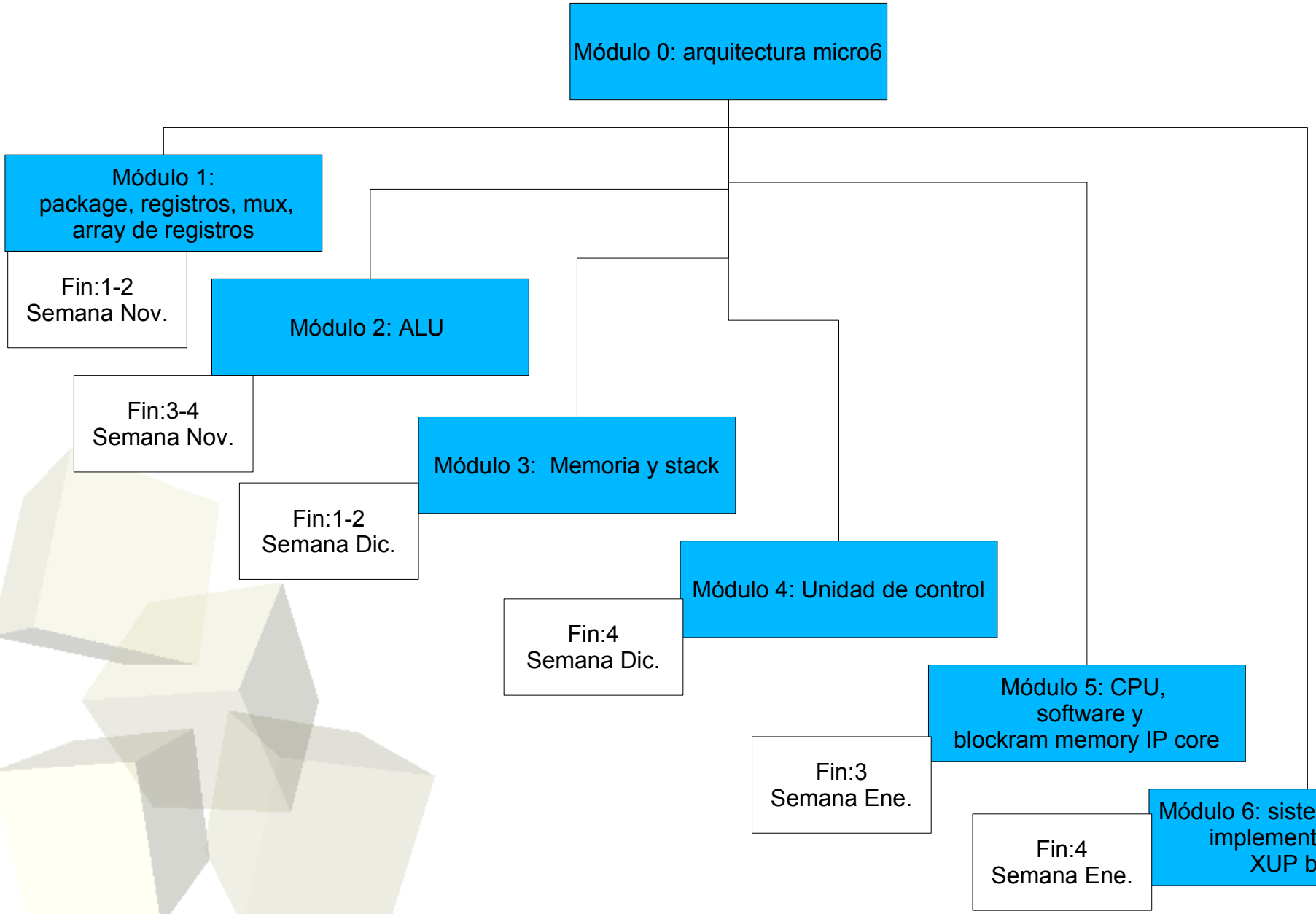
# Desarrollo de los módulos en el diseño del microprocesador MICRO6

Manuel J. Bellido Díaz

Octubre de 2010

# Estructuración del diseño de Micro6

- División del diseño de Micro6 en 1+6 módulos



# Módulo 1

- El Módulo 1 se divide en 5 submódulos: 1a hasta 1e

- Módulo 1a: Completar el fichero micro_pk.vhd.
  - A tener en cuenta: Utilización de atributos de señales

    arg(31 downto 0) ---> arg'left = 31; arg(31) = arg(arg'left)
  - Declaración de arrays sin tamaño definido usar como rango:

    natural range <>

- Módulo 1b: Diseño de registros básicos

  - Solo Proceso secuencial ya que el registro no tiene señal de lectura (lectura incondicional)

  - Diseño Parametrizable: Vectores sin restricciones.
    - registro_in : in std_logic_vector;

- Módulo 1c: Diseño de Contadores (para PC y Stack Pointer)
  - A tener en cuenta: Señales internas tipo unsigned para realizar operaciones de incremento/decremento:

```
signal  cont  unsigned(q'range):
.....
    cont <= cont +1 ;
    cont <= cont -1 ;
  -
```

  - Conversión de tipos entre señales.
  - Completar testbench de contador up-down.
- Módulo 1d: Diseño de multiplexores (para interconexión de componentes -técnica multiplexada, no H.I.-)
  - Aspecto a tener en cuenta: Bloque combinacional. Lista de sensibilidad adecuada a bloques combinacionles.

- El Módulo 1e
  - Módulo 1e: Diseño de fichero de registros

    - A tener en cuenta: Utilización de los componentes diseñados entre módulo 1a y 1d: Es un diseño estructural.
    - No es necesario declarar los componentes que van a emplearse ya que existe un “package” donde están declarados todos los componentes del microprocesador MICRO6:
      - **micro_comp_pk.vhd**
    - Instanciación repetitiva mediante bucle:

      for _____ generate.

    - ERROR EN CODIGO REGFILE.VHD:
      - Conversión selX a entero
    - IMPORTANTE: HABRÁ QUE INCORPORAR AL PROYECTO DE SIMULACIÓN TODOS LOS CÓDIGOS QUE SE NECESITEN DE LOS MÓDULOS ANTERIORES

- Módulo 2a: Diseño de la ALU
  - Aspectos a tener en cuenta:
    - Sobre los bits de estado: Se almacenan en biestables
      - Negativo --> es el bit de signo del resultado
      - Overflow--> bit de desbordamiento en las operaciones aritméticas (ADD,SUB, MULT,INC,DEC)
      - Zero --> vale 1 si el resultado es 0; 0 en otro caso.
      -
    - Sobre las operaciones:
      - Se seleccionan con sel que es de tipo alu_op (ver micro_pk.vhd)
      - DIV y REM no se implementan

- Módulo 2a: Diseño de la ALU
  - Sobre la operación MULT:
    - La operación MULT da como resultado el producto de los datos A y B con las siguientes consideraciones:
      - A y B son números con signo (el resultado también)
      - A y B deben ser números representables en DATA_WIDTH/2 bits para que el resultado quepa en DATA_WIDTH bits
      - Hay overflow si algunos de los datos A o B no cabe en DATA_WIDTH/2 bits
      - Para implementar MULT en el código alu.vhd lo mejor es emplear la función MultL que se define en micro_pk.vhd:
        - Result <= MultL(A,B);
      - Esta función incluye la función RESIZE definida en numeric_std.

# Módulo 2b:Testbench ALU

- ESQUEMA DE PEROCESO DE TEST DE LA ALU BASADO EN LEER PATRONES DE UN FICHERO ASCII CON LOS DATOS

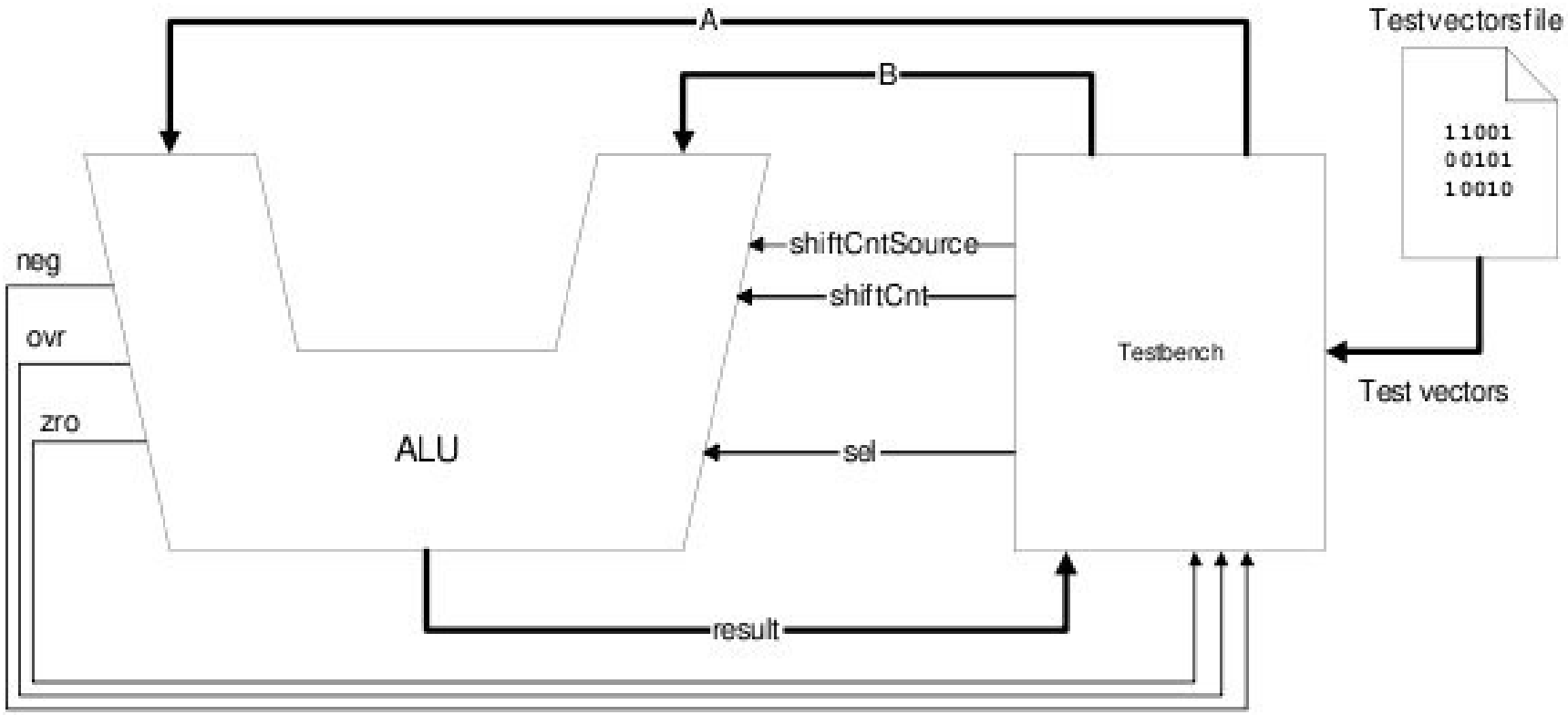



**Figure 1: ALU + Testbench**

# Módulo 2b:Testbench ALU

- Aspectos tener en cuenta:
  - Se incluye la librería STD_textio
  - Se necesitan los ficheros: micro_pk.vhd, micro_com_pk.vhd, alu.vhd, alu_testvectors.txt
  - Seguir la guías del documento pdf del modulo 2b para abrir el fichero alu_testvectors.txt en el código de test.
  - Del fichero alu_testvectors.xls podemos obtener que datos hay en cada uno de los campos del fichero alu_testvectors.txt
  - Tener cuidado con los tipos de los datos (p.ej., los datos de a y b son enteros. Para leerlos habrá que definir una variable del tipo correcto, que posteriormente habrá que asociar a la correspondiente señal. Ejemplo para leer el dato a:
    - variable a_integer : integer;
    - read (vector_actual, a_integer);
    - A <= (std_logic_vector(signed(a_integer)));

# Módulo 2b: Testbench ALU

- Realizar la comparación de los valores esperados y los resultados de simulacion de los bits neg, ovf y zro y del datos result (Solo si el bit de chequeo de resultado vale 1). Ejemplo de neg:
  - assert neg = bit_neg report "Neg" & std_logic'image(neg) & "no coincide con el valor esperado en el vector"&integer'image(num_vec)
  - severity ERROR;

# Módulo 3a: Memoria

- En este modulo 3a se desarrolla el código que “simula” a la memoria del sistema.
- Cuando se implemente el sistema completo en la FPGA (módulo 6a) este código va a ser sustituido por otro que incluya un IP core de memoria (es decir, por una memoria real)
- La memoria que va a implementarse es síncrona y realiza una operación (write o read) en 1 solo ciclo de reloj. Por este motivo la señal ready puede sobrar (ready <='1';)
- Aspectos a tener en cuenta:
  - Para simular la memoria de 4k*32 se declara una variable de tipo memContents_t (revisar micro_pk.vhd).
  - Para que los procesos de simulación funcionen correctamente es necesario que la memoria este cargada con algunos valores. Por esto se inicializa con los contenidos de RAM_CONTENTS que están definidos en el fichero micro_ram_pk.vhd

# Modulo 3b: Controlador de memoria

- En este modulo se diseña el controlador de memoria que interconecta la memoria con los posibles usuarios de la memoria que son: bloque de FETCH de la CPU, bloque de datapath de la CPU y periféricos (posible DMA)
- Se implementa como una maquina finita de estados (FSM). Código VHDL estándar para FSMs
- Aspectos a tener en cuenta:
  - Es importante tener en cuenta que mientras la memoria no haya completado la operación, la máquina de estados debe permanecer en el mismo estado. Esto es para que el bloque que este usando la memoria lo haga hasta que se complete la operación.
  - Es importante revisar las listas de sensibilidad de los procesos que implementan el bloque combinacional (las dos estructuras de case)

# Módulo 3c: Diseño de la pila interna

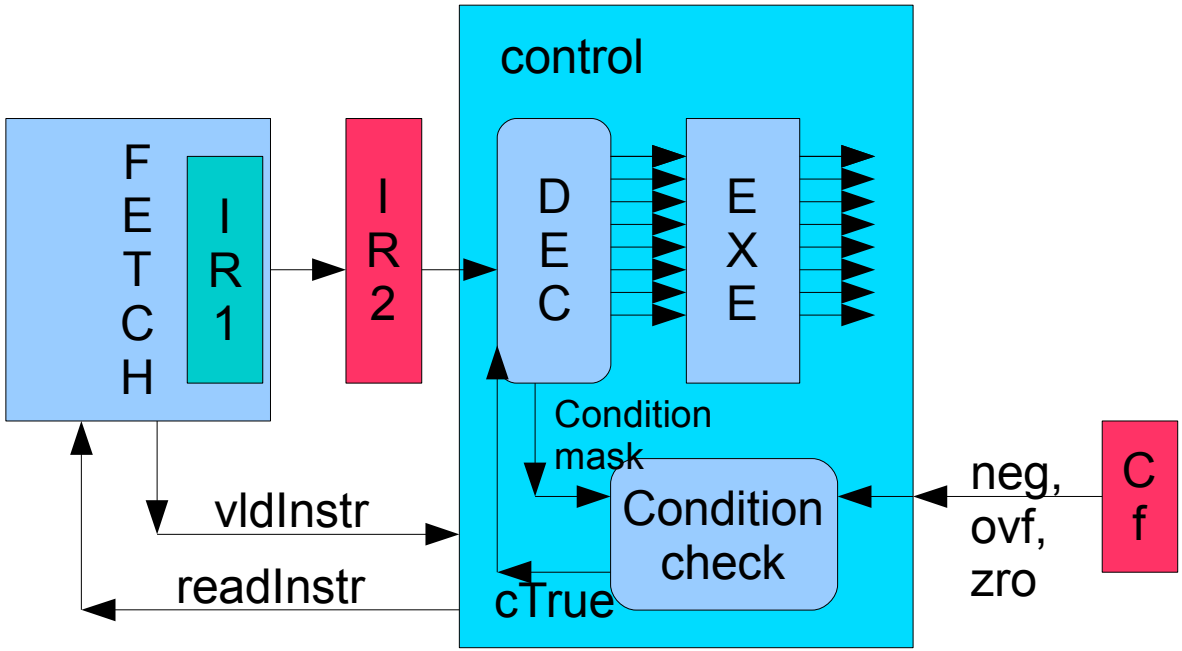
- Micro6 va a tener dos pilas:
  - Una en la memoria RAM para almacenar datos en forma de PILA (existe un registro que es el 31 de regfile que es el puntero de pila. Se diseño en el modulo 1e). Se emplea en las instrucciones de push y pull del microprocesador
  - Otra interna (de nombre stack) que se va a diseñar en este modulo 3c y sirve para guardar las direcciones de retorno de subrutinas. Se empleará en las instrucciones de salto a subrutina y retorno de subrutina
- Aspectos a tener en cuenta:
  - El puntero de pila (**pointer** en el código vhdl) esta apuntando a la primera dirección vacía de la pila. Como la profundidad es de 16 datos significa que cuando este vacía pointer=0 y cuando este llena pointer=16. El tamaño de pointer sera de 5 bits para poder representar desde el 0 hasta el 16.

# Módulo 4: Unidad de Control

- Operaciones que realiza la Unidad de Control de Micro6:
  - Búsqueda de la Instrucción en memoria (FETCH)
    - En el hardware existe un bloque especial dedicado a esta tarea (FETCH UNIT, módulo 4c)
  - Decodificación de la instrucción. Cada instrucción de 32 bits es separada en campos de operación según el formato de cada instrucción (p.ej., opcode (IR(31 downto 27))
    - En el hardware se implementara como un bloque combinacional mediante una función en VHDL (módulo 4a)
  - Bloque de control de la ejecución de las instrucciones (execute)
    - En el hardware se implementara como una máquina de estados de moore en la que se irán activando las señales de control adecuadas de la ruta de datos para la correcta ejecución de una instrucción (módulo 4b)

# Módulo 4: Unidad de Control

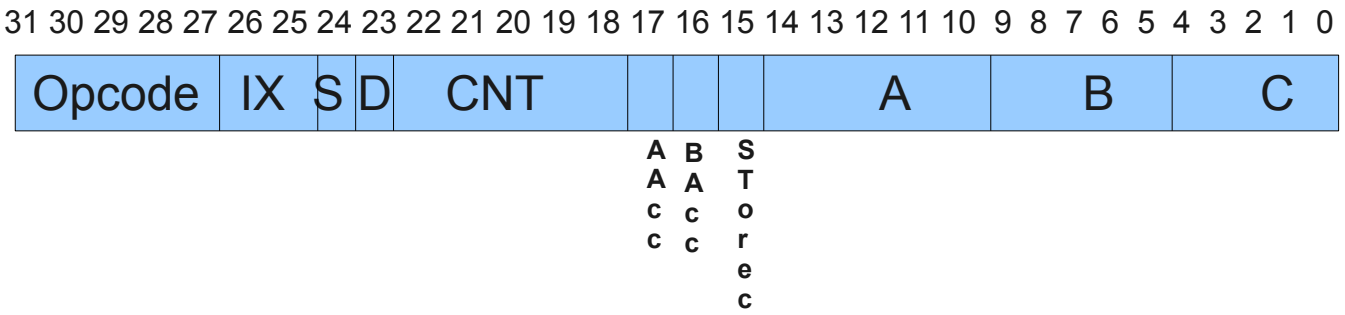
- Diseño de la UC:
  - Arquitectura de pipeline de dos fases: FETCH-(DEC-EXEC)




  - Handchake entre control y fetch para evitar problemas de dependencia en pipeline con la instrucción de salto
  - Condition Check evalua si hay (cTrue=1) o no salto (cTrue=0)

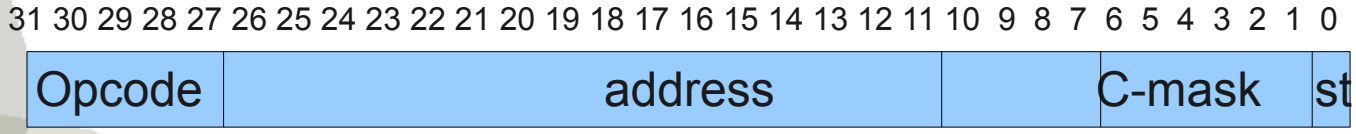
# Módulo 4: Unidad de Control

- Formatos de instrucciones en Micro6:
  - Operación con datos (ADD,SUB, etc): la mayoría de las instrucciones




  - Instrucciones de I/O y LDM (Load memory- sustituye dir. inmedi.)


  - Saltos



- Formatos de instrucciones en Micro6: nombre de los campos
  -

| Field | Description |
|---|---|
| OPCODE | Opcode |
| IX | Index register |
| S | Shift count source |
| D | Shift direction |
| CNT | Shift count |
| AACC | ALU port A is ACC (Accumulator) |
| BACC | ALU port B is ACC (Accumulator) |
| StoreC | Store the result in C |
| A | Register file port A selection lines |
| B | Register file port B selection lines |
| C | Register file input port selection lines |
| Page-0 address | Address of a location in the first memory page |
| C-MASK | Condition mask |
| ST | Enable stack |

# Módulo 4a: Dec. de la instrucción

- Aspectos a tener en cuenta:
  - Solo se completa un fichero de package donde se define una función para la decodificación que va a ser incluida dentro del código control.vhd que se diseña en el módulo 4b
  - ERROR EN EL DOCUMENTO:
    - AUNQUE EN EL PDF DEL MÓDULO 4A INDIQUE QUE memAddr es de tamaño 16 bits (15:0) REALMENTE ES DE TAMAÑO 12 (11:0)
  - IMPORTANTE: en el case_1 hay que incluir el valor de cFen e instructgroup para todas las instrucciones según una tabla que aparece al comenzar el case. En el código ya aparecen las instrucciones que son de tipo tanto de tipo G1 como de tipo G2 según el valor de storec y nosotros tenemos que incluir el resto.  Es importante tener en cuenta que existen un conjunto de instrucciones de tipo G2 (INC,DEC,NOT,ZRO,CPR) que tenemos que incluir y que suelen pasar desapercibidas.

# Módulo 4b: Diseño de control

- En este módulo se diseña el bloque principal de la unidad de control. Incluye el bloque de decodificación (mediante la función desarrollada en el módulo anterior) el bloque de chequeo de la condición de salto y la máquina de estados que genera las señales de control adecuadas para la correcta ejecución de las instrucciones.
- La FSM de control de la ejecución de las instrucciones tiene un total de 64 estados. Es una máquina de estados compleja en la que el proceso de diseño para llegar a obtenerla pasa por la descomposición en microoperaciones de todas y cada una de las instrucciones.
  - Ejemplo: instrucción **ADD Ra Rb Rc** (Rc <-- Ra + Rb)

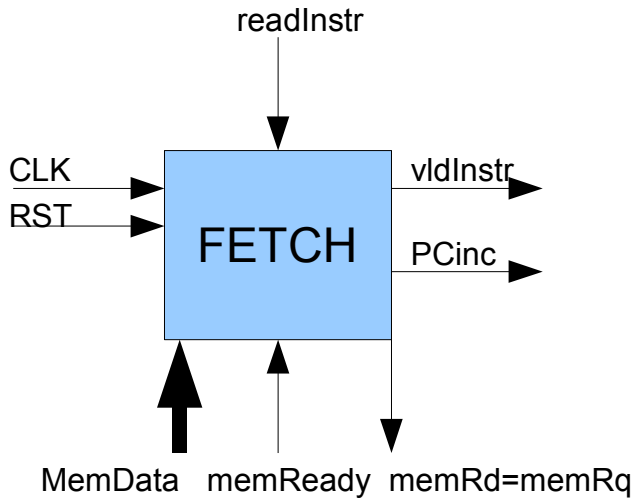
| **Microoperaciones** | **Estado** | **Señales de Control** |
|---|---|---|
| ACC <-- Ra + Rb | g2s1 | ACCen, ALUsel=ADD |
| Rc <-- ACC | g2s2 | RegFileWr, DATAsel=ACC |

# Módulo 4b: Control

- Aspectos atener en cuenta:
  - La FSM no habrá que diseñarla. Ya esta implementada en el código control.vhd
  - Hay que incluir y diseñar:
    - El bloque de decodificación (incluyendo la función creada en el módulo 4a)
    - El circuito de chequeo de condición de salto
      - Puede implementarse tal y como comenta el documento PDF del módulo 4b, mediante un circuito combinacional a nivel de puertas o bien describiendo el comportamiento del bloque mediante un proceso.
  - ERRORES EN EL CÓDIGO:
    - El alias “cmask” se obtiene de la instrucción en los bits que van del 6 al 1 y no del 26 al 21.

# Módulo 4c: Unidad de Fetch

- FETCH: Máquina de Estados (FSM) con 3 estados:
  - Idle : Espera la activación de readInstr (mantiene vldInstr activa)
  - Read_memory: activas las señales para leer la instrucción de la memoria y la almacena en un registro interno. También mantiene desactivado vldInstr para indicarle al módulo control que no se ha completado aun la lectura.
  - Incrementa PC: para apuntar a la siguiente instrucción. También se activa vldInstr para indicarle al módulo de control que se ha completado el proceso de lectura de la memoria

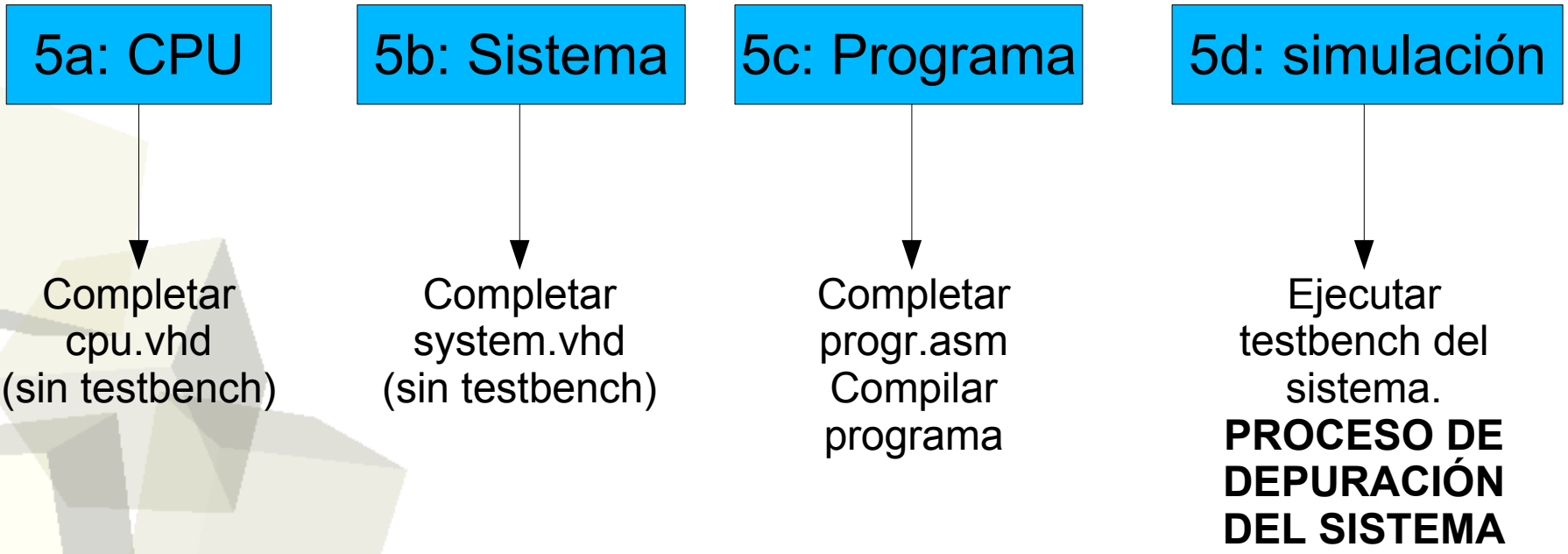


# Módulo 4c: Unidad de Fetch

- Aspectos a tener en cuenta:
  - En el documento PDF se menciona que el código control.vhd que se necesita para realizar la simulación incluye algún código extra para facilitar el debug del diseño de FETCH.
  - Sin embargo, el  código que se va a usar es el que hemos generado nosotros mismos en el módulo 4b que no incluye este código extra.

# Módulo 5: Integración CPU y Sistema

- El módulo 5 esta dedicado a realizar la integración de la CPU, en primer lugar, posteriormente unir la CPU con la memoria a través del controlador de memoria, creando, así un sistema computador sin periféricos (solo CPU y Memoria). Por último, se va a verificar el correcto funcionamiento de este sistema mediante el ensamblado y la ejecución de un programa a través de simulación.



# Módulo 5a-b: Integración CPU y Sistema

- En el módulo 5a se integran los componentes que se han diseñado desde el módulo 1a hasta el 4c para completar la CPU
- En el módulo 5b se unen la CPU, la memoria y el controlador de memoria dando lugar a un sistema.
- Aspectos a tener en cuenta:
  - Sería conveniente copiar todos los códigos generados y/o usados en esos módulos a la carpeta modulo5a/input/ y a la carpeta modulo5b/input/
  - Solo se chequea la sintaxis del código cpu.vhd (5a) y system.vhd (5b) que son los que hay que completar respectivamente. No existen testbench.
  - Esto significa que puede haber interconexiones erróneas. Estas saldrán a la luz en la simulación del sistema completo (módulo 5d)

# Módulo 5c: Programa y ensamblador

- El objetivo del módulo 5c es conocer las características básicas del lenguaje ensamblador de Micro6 y de su transformación en código máquina mediante un ensamblador.
- Tareas a desarrollar: 1.- completar el programa ensamblador que implementa un algoritmo de ordenación de números. 2.- Ensamblar el programa para generar el código máquina.
- Aspectos a tener en cuenta:
  - Existen una serie de errores en el programa (fichero prog.asm) que hay que corregir además de completar el programa:

| Instrucciones erróneas | Instrucción corregida |
|---|---|
| INC Rx | INC Rx Rx |
| DEC Rx | DEC Rx Rx |
| ZRO Rx | ZRO Rx Rx |
| CMP Rx Ry | CMP Rx Ry Rz |

# Módulo 5c: Programa y ensamblador

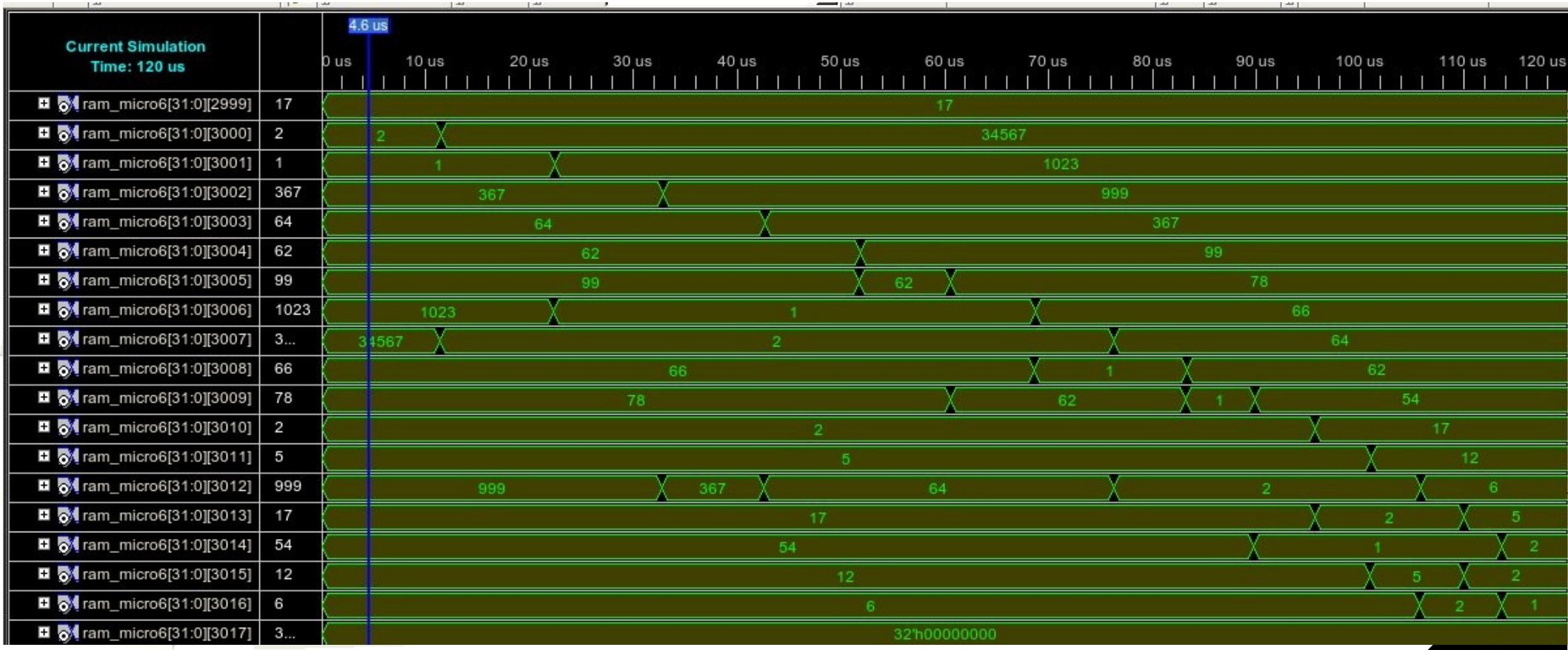
- Aspectos a tener en cuenta:
  - El programa ensamblador esta desarrollado en VHDL (no es lo más apropiado pero es válido)
  - Para generar el código máquina hay que ejecutar una simulación del ensamblador con los ficheros que incluyan los siguientes ficheros del programa:
    - prog.asm — Programa en lenguaje ensamblador Micro6
    - data.asm — datos que necesita el programa (se cargan en memoria)
  - Como resultado de la simulación se obtienen dos ficheros con el programa en código maquina (en la carpeta del proyecto):
    - micro_ram_pk.vhd — Código VHDL que da valor a la constante RAM_CONTENTS. Se usará para simulación del sistema (módulo 5d)
    - ram.coe — Fichero con el contenido de la memoria que se utilizará para inicializar el IP core de la memoria en el proceso de implementación (módulo 6a)

# Módulo 5d: Simulación sistema

- El objetivo del módulo 5d es comprobar la correcta funcionalidad del diseño realizado de Micro6. Para ello se realiza la simulación del sistema del módulo 5b con el programa ensamblado en el módulo 5d.
- El módulo esta preparado para utilizar el simulador MODELSIM, sin embargo se puede realizar la simulación con ISEsimulator (simulador del entorno ISE).
- Cuando se emplea ISEsimulator hay que obviar todo lo que cuenta el documento pdf del módulo respecto de modelsim.
- Para comprobar la correcta ejecución del programa en el sistema hay que realizar un proceso de análisis de las formas de onda de las señales adecuadas.
- Lo habitual es que el sistema no funcione correctamente a la primera (si es así, enhorabuena al que lo consiga)
- Si no funciona correctamente, habrá que realizar un muy laborioso proceso de análisis y corrección de errores que se hayan cometido a lo largo de todo el proceso de diseño

# Módulo 5d: Resultado correcto

- Para visualizar los datos de la memoria en el código memory.vhd hay que transformar la variable interna en señal interna
- Contenido de la RAM durante la ejecución del programa de ordenación de números:
  - Posición 2999: nº de datos
  - Posiciones 3000 a 3016: los datos



# Módulo 6: Implementación On-chip Micro6

- Una vez que el procesador Micro a funcionado correctamente mediante simulación (módulo 5d), el siguiente paso consiste en prepararlo para su implementación final (sobre una FPGA tipo VirtexII-Pro) y su verificación on-chip. Esta tareas se realizan en el Módulo 6 que tiene dos partes:
  - Módulo 6a: El código memory.vhd no es sintentizable ya que ocuparía gran cantidad de recursos de la FPGA. Para resolver el problema de la memoria de 4kx32 que direciona la versión de Micro6 diseñada vamos a sustituir este código VHDL por un IP Core de XILINX. Además, para poder interactuar con Micro6 se necesita algún periférico a través del cual un usuario pueda mandar y recibir datos. En este Módulo se emplea una UART (RS232) para poder conectar Micro con un PC a través del puerto serie.
  - Módulo 6b: Implementación final en la Placa XUP-VitexII-Pro

# Módulo 6a: Sustituir mem por IP Core

- NOTA PARA EL CURSO 09/10: Para facilitar el desarrollo del Módulo 6 se han incluido dos códigos VHDL en la carpeta input/system_vhdl que sustituyen a dos códigos empleados en el módulo 5d:
  - system-6a.vhd Sustituye a system-vhd
  - micro_comp_pk.vhd Sustituye a micro_comp_pk.vhd del módulo 1.e

  El resto de componentes y packages de Micro6 usados en el módulo 5d si habrá que usarlos en este módulo
- Aspectos a considerar en el desarrollo del Módulo 6a:
  - Se aconseja copiar todos los códigos vhd del diseño de Micro6 usados en el módulo 5d (NO LOS DEL TESTBENCH) en la carpeta system_vhdl menos system.vhd y micro_comp_pk.vhd
  - MUY IMPORTANTE: Aquel que trabaje con el ISEWebPack (los que han instalado la herramienta en un ordenador propio) pueden seguir el documento PDF del módulo 6a. Los que trabajen con ISE 10.1 deben seguir el recetario que viene en las siguientes transparencias

# Módulo 6a: Sustituir mem por IP Core

- 1- Compilar el programa prog.asm que se encuentra en la carpeta input/assambler/ según instrucciones de modulo5c
  - Nota: Recordar que hay que modificar algunas instrucciones en el código para que funcione correctamente (INC,DEC,etc)
- 2- Crear un proyecto en el que se incorporen todos los ficheros vhd del sistema (input/system_vhdl/)
  - Nota: memory.vhd se debe borrar; De los dos system se debe usar system-6a.vhd (borrar system.vhd)
  - Atención: El fichero micro_comp.pk.vhd que debe usarse es el que se encuentra en la carpeta input/system_vhdl/ del módulo6a y no el usado en los módulos anteriores
- 3- Incorporar los códigos VHDL de la carpeta input/testbench/
- 4- Incorporar el fichero micro_ram_pk.vhd generado en el punto 1 (Atención: Habrá que borrar cualquier otro micro_ram_pk.vhd)

# Módulo 6a: Sustituir mem por IP Core

- 5- Obsérvese en el proyecto de ISE que solo falta el componente de nombre micro6_ram (será en IP Core)
- 6- En proyecto Añadir New Source --> IP (Core Generator..
  - Atención: Para que el IP Core se asocie al componente que falta debemos darle el nombre “micro6_ram”
  - Buscar el componente en:
    - Basic Elements --> Memory Elements
  - Seleccionar el IP Core: Block Memory Generator
  - Se lanzará una aplicación para generar una memoria
    - Primera Pantalla: Seleccionar Single Port y minimum area: NEXT
    - Segunda Pantalla: Poner el tamaño correcto de la Memoria: 4Kx32
      - Seleccionar Write First y Use ENA Pin: Next
    - Tercera Pantalla: Seleccionar Load Init File
      - Browse: Buscar fichero ram.coe en input/assembler/<Dir_SIM>/ : Next
      - Obsérvese que la señal de escritura se define como wea(0:0)
        Esto significa que, aunque de 1 solo bit será de tipo std_logic_vector
    - Ultima Pantalla: Finish. Se debe generar el IP Core.

# Módulo 6a: Sustituir mem por IP Core

- 7- Una vez generado el IP Core de nombre micro6_ram, seleccionandolo en la ventana de sources podemos darle a la opción “View HDL functional model”
  - Esto genera un código VHDL del IP Core que nos sirve para ver cual es su estructura desde elpunto de vista de señales de entrada y salida.
  - Como el IP Core generado no es el mismo IP Core para el que estaba preparado el diseño de Micro6 original hay que hacer cambios en dos ficheros para adaptar el nuevo IP CORE al System-6a.vhd
    - Modificar main_mem.vhd
      - Cambiar el componente micro6_ram; resolver las salidas rfd y ready; resolver la conexión wea(0:0) (std_logic_vector) con we (std_logic)
    - Modificar micro_comp_pk.vhd: Cambiar componente micro6_ram
- A estas alturas del curso los alumnos deberías ser capaces de realizar las modificaciones necesarias para que el sistema funcione correctamente