# Tema 12.2 - Creación de un Periférico Wishbone

Sistemas Electrónicos para Automatización

Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Hipólito Guzmán Miranda

# Contenido

- Motivación
- Creación de un periférico Wishbone
- Otras funcionalidades
- Añadir periférico al diseño
- Modificaciones hardware
- Modificaciones software

# Contenido

- Motivación
- Creación de un periférico Wishbone
- Otras funcionalidades
- Añadir periférico al diseño
- Modificaciones hardware
- Modificaciones software

# Motivación

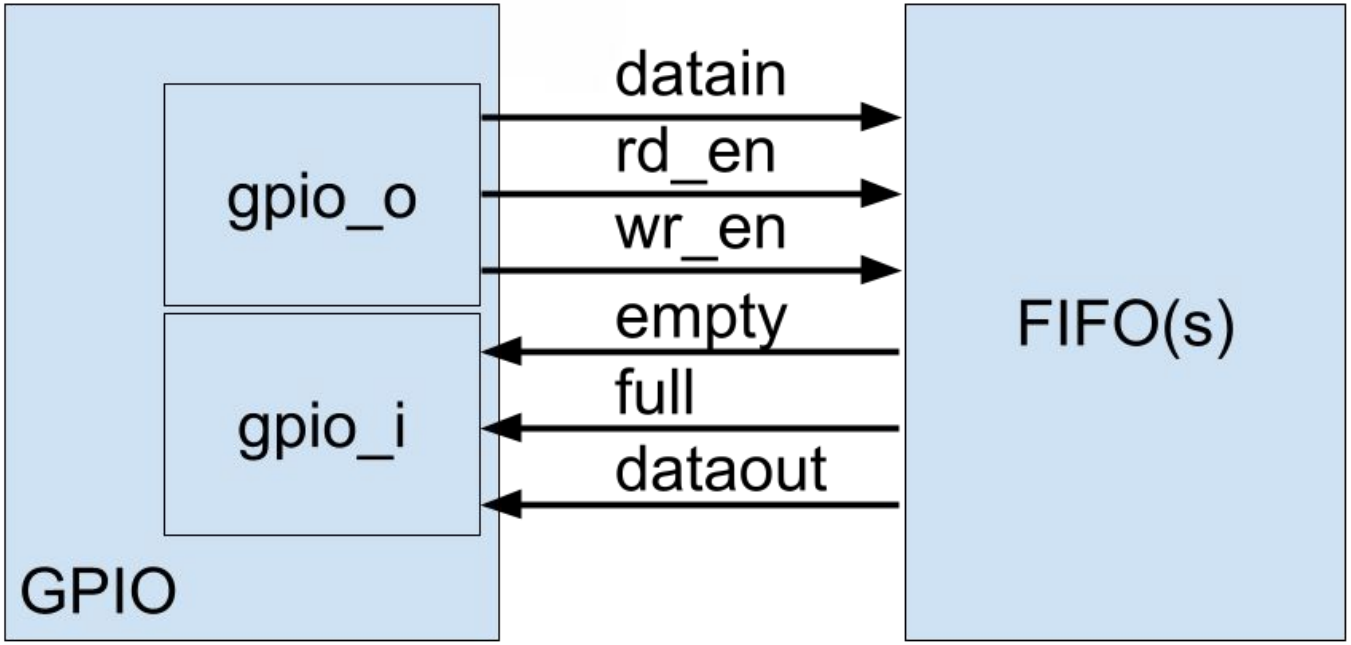
## GPIO = quick and dirty solution :)

- GPIO es fácil y rápido de prototipar
  - ‘Sólo’ tienes que saber de FPGAs, microprocesadores empotrados, arquitectura de microprocesadores, manejo de las herramientas, C, librerías para manejo del GPIO… ;)
- Pero los accesos son lentos
- Y el interfaz es muy sencillo: y si queremos tener interfaz tipo FIFO, regs, memoria?
  - Tendríamos que implementarlos a mano: el acceso será aún más lento

# Ejemplo: acceso a una FIFO desde GPIO

- GPIO con dos canales
- Conectamos al canal de salida del GPIO las señales datain, wren, rden de la FIFO
- Conectamos al canal de entrada del GPIO las señales dataout, full, empty de la FIFO

# Ejemplo: acceso a FIFO desde GPIO



# Ejemplo: acceso a una FIFO desde GPIO

Operación de lectura:

- Comprobar bit empty (lectura GPIO in)
- Si !empty:
  - Activar rd_en (escritura en GPIO out)
  - Desactivar rd_en (escritura en GPIO out)
- Leer datos (lectura GPIO in)

4 operaciones para leer un dato!

# Ejemplo: acceso a una FIFO desde GPIO

Operación de escritura:

- Comprobar bit full (lectura GPIO in)
- Si !full:
  - Escribir dato (escritura en GPIO out)
  - Activar wr_en (escritura en GPIO out)
  - Desactivar wr_en (escritura en GPIO out)

4 operaciones para escribir un dato!

# Ejemplo: acceso a una FIFO desde GPIO

Adicionalmente, no tenemos control sobre cuántos ciclos se mantienen activos rd_en y wr_en, con lo cual escribimos/leemos múltiples veces los datos en la FIFO

# Limitaciones del GPIO

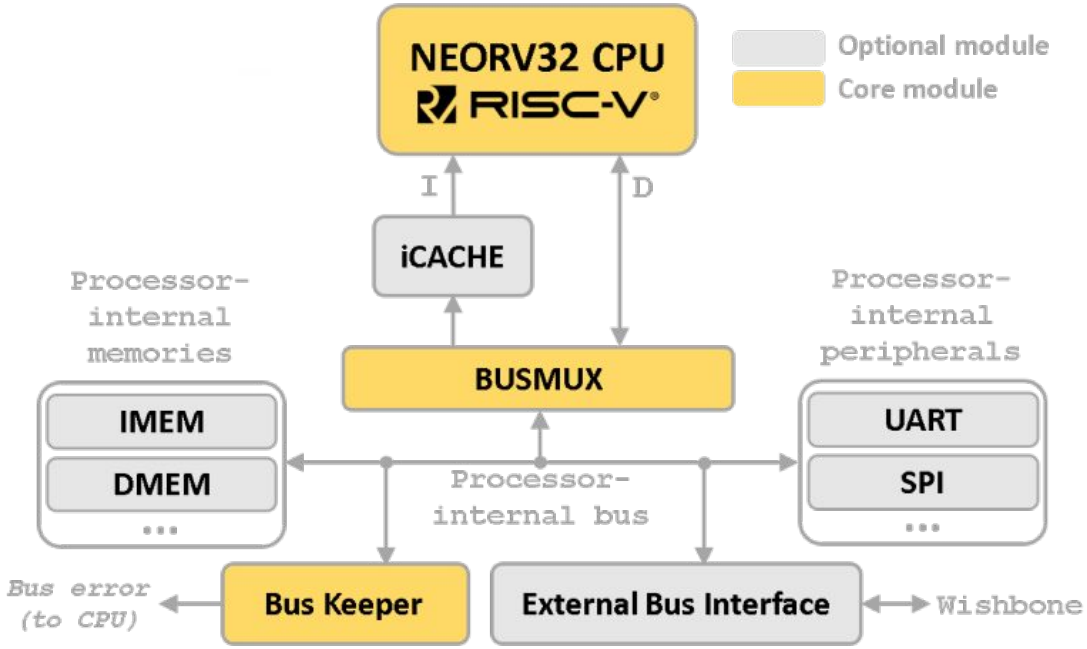
- GPIO se puede usar cuando el interfaz es sencillo y no existen restricciones de tiempo fuertes
- En otro caso, representará un cuello de botella en nuestro sistema
- En neorv32 sólo tenemos 1 GPIO!
  - (de 64 bits)
- No obstante, puede ser muy útil para prototipado rápido de soluciones

# Contenido

- Motivación
- Creación de un periférico Wishbone
- Otras funcionalidades
- Añadir periférico al diseño
- Modificaciones hardware
- Modificaciones software

# Periféricos Wishbone

- Mayores prestaciones de velocidad
- Necesario poner el generic *MEM_EXT_EN* a **true**



# Creación de un periférico Wishbone

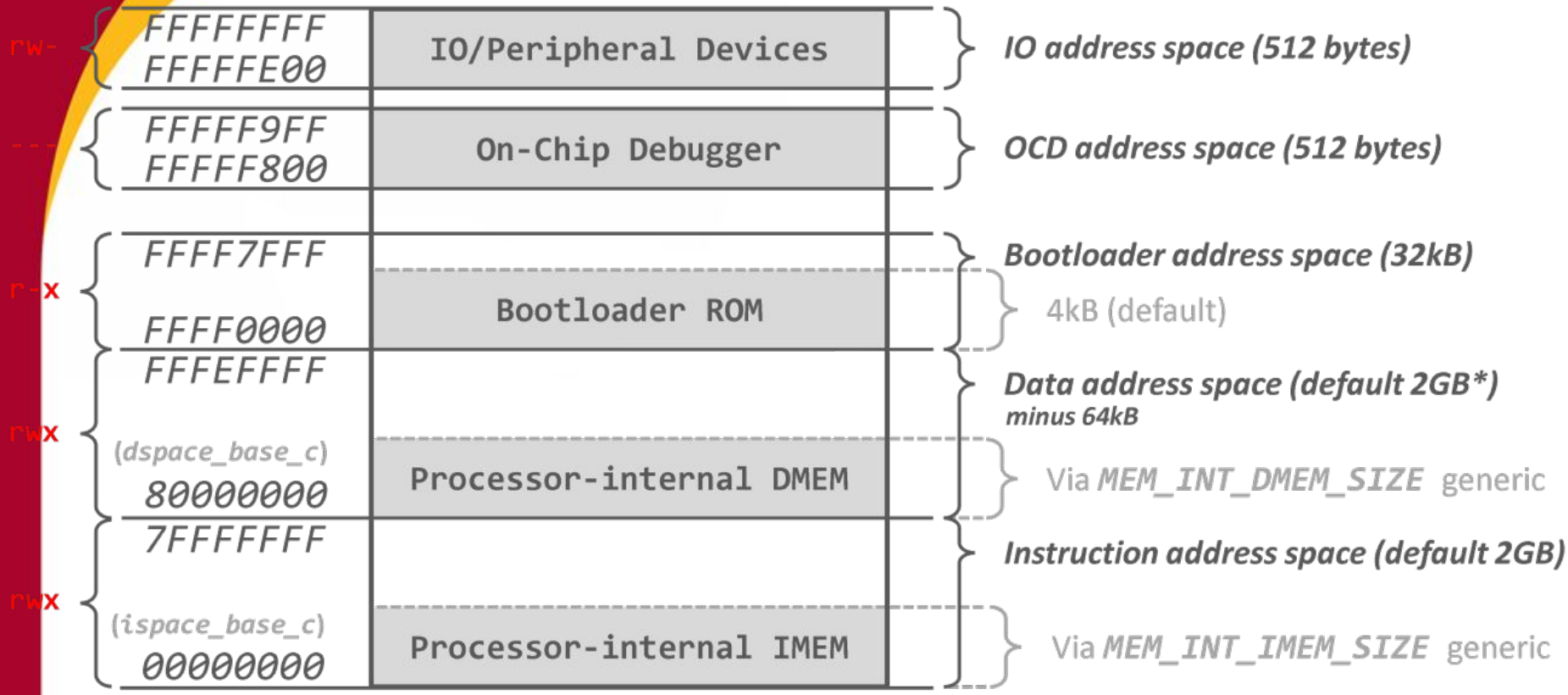
| Generic | Function |
|---|---|
| *MEM_EXT_EN* | enable external memory interface when *true* |
| *MEM_EXT_TIMEOUT* | number of clock cycles after which an unacknowledged external bus access will auto-terminate (0 = disabled) |
| *MEM_EXT_PIPE_MODE* | when *false* (default): classic/standard Wishbone protocol; when *true*: pipelined Wishbone protocol |
| *MEM_EXT_BIG_ENDIAN* | byte-order (Endianness) of external memory interface; true=big, false=little (default) |
| *MEM_EXT_ASYNC_RX* | use registered RX path when *false* (default); use async/direct RX path when *true* |

# Creación de un periférico Wishbone

| Port | Function |
|---|---|
| wb_tag_o | request tag output (3-bit) |
| wb_adr_o | address output (32-bit) |
| wb_dat_i | data input (32-bit) |
| wb_dat_o | data output (32-bit) |
| wb_we_o | write enable (1-bit) |
| wb_sel_o | byte enable (4-bit) |
| wb_stb_o | strobe (1-bit) |
| wb_cyc_o | valid cycle (1-bit) |
| wb_lock_o | exclusive access request (1-bit) |
| wb_ack_i | acknowledge (1-bit) |
| wb_err_i | bus error (1-bit) |
| fence_o | an executed fence instruction |
| fencei_o | an executed fence.i instruction |

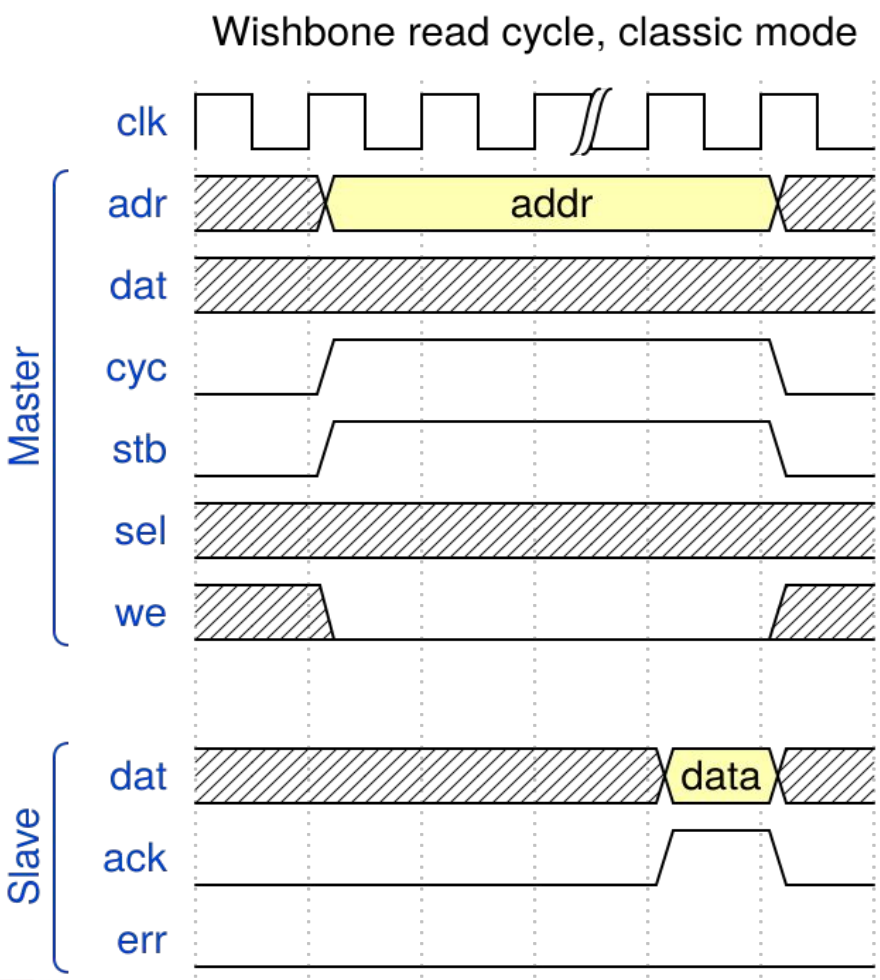
# Creación de un periférico Wishbone

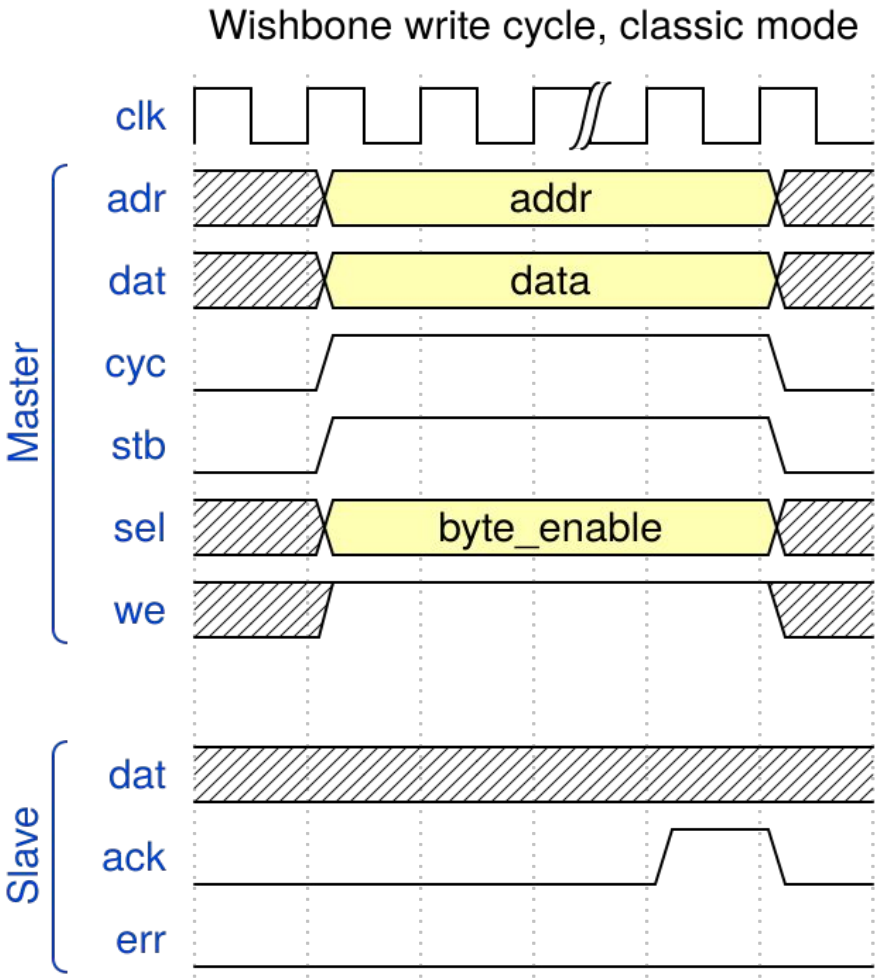
## Espacio de direcciones




Todo acceso a una zona que no esté marcada en gris entre `0x00000000` y `0xffff0000` usará el bus Wishbone

# Ciclos de lectura y escritura



# Conexión

- Si es un único periférico:

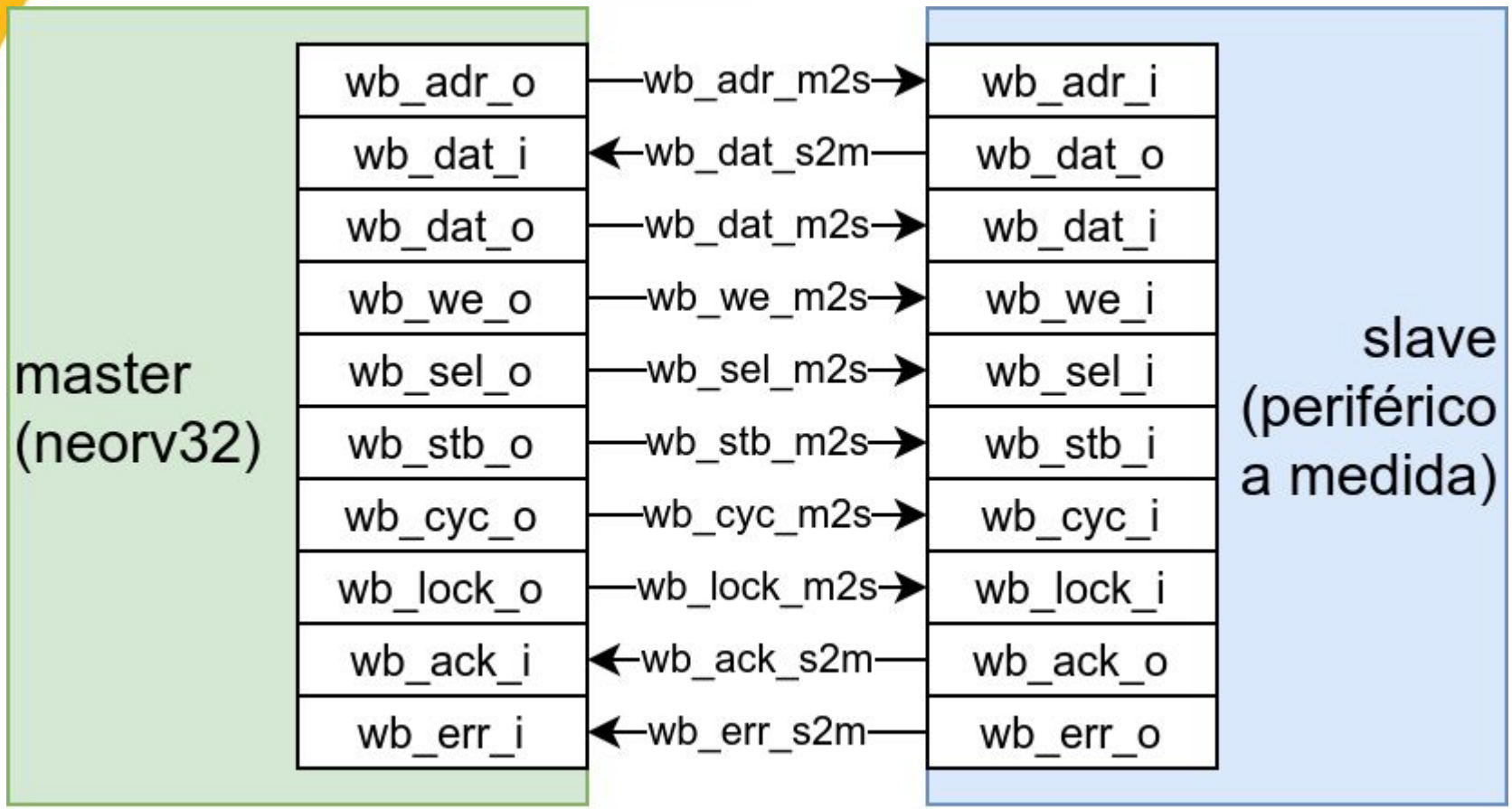
  Conexión directa al master

- Si tienes más de un periférico:

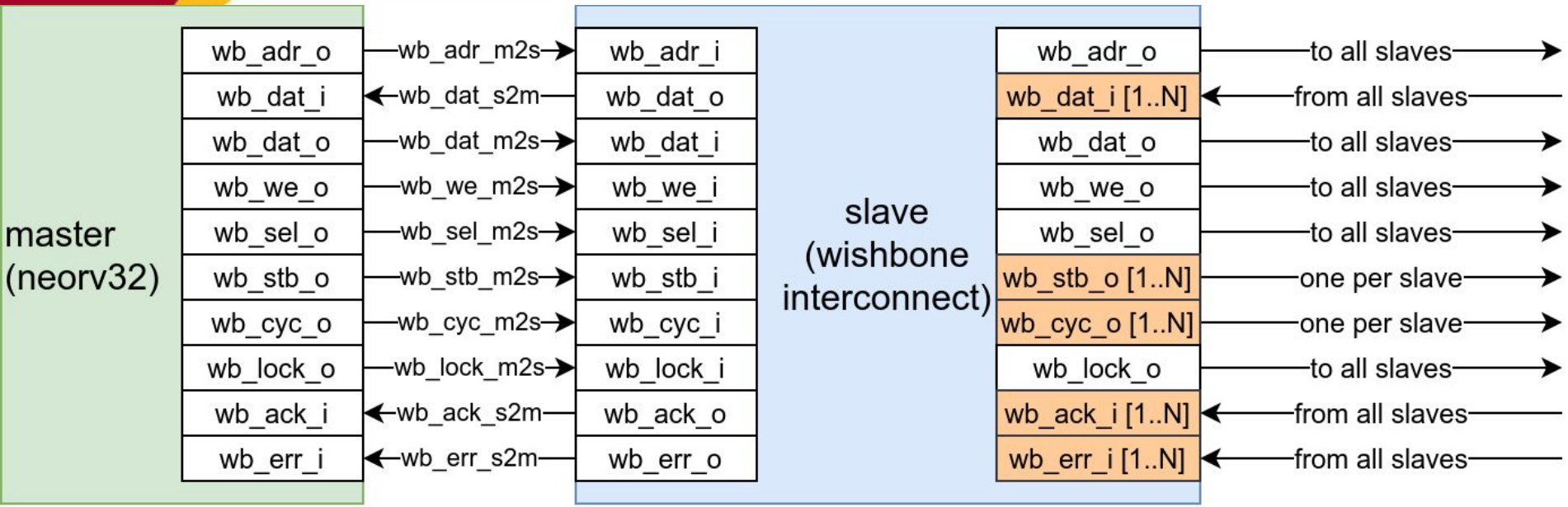
  Uso de un wishbone interconnect (shared bus)

# Creación de un periférico Wishbone

## Conexión directa



# Uso de wishbone interconnect




- Calcula cs (chip select) en función de adr_i (y de nuestro memory map)
- Activa sólo un stb y un cyc, el del periférico seleccionado
- Devuelve al neorv32 el dato, ack y error del periférico seleccionado (multiplexa en función de cs)

# Encajando el periférico en el mapa de memoria

- Tu periférico debe conocer su BASEADDR y SIZE para determinar cuándo está seleccionado en función de wb_adr.
  - Esto se puede simplificar cuando usas un wishbone interconnect
- Tiene sentido que BASEADDR y SIZE sean generics en el periférico, así podrás recolocarlo si fuera necesario.

# Configurando neorv32 para poder usar Wishbone

- Al instanciar neorv32 debemos asegurarnos de que
  - `MEM_EXT_EN => true`
- También deberíamos poner
  - `MEM_EXT_TIMEOUT => 255`
- Y
  - `MEM_EXT_PIPE_MODE => false`
- Los otros valores podemos dejarlos por defecto:
  - `MEM_EXT_BIG_ENDIAN => false`
  - `MEM_EXT_ASYNC_RX => false`

# Probando un periférico sencillo con varios registros

- Debemos declarar las señales del bus wishbone
- Debemos conectar las señales del bus al neorv32
- Debemos instanciar el periférico en el top-level, conectándolo a las señales del bus
- El desarrollador de neorv32 proporciona un periférico sencillo que instancia una memoria de tamaño configurable

# Probando un periférico sencillo con varios registros

- Debemos declarar las señales del bus wishbone

```
-- ------------------------------------------------------------------------
-- Signals for Wishbone interface
-- ------------------------------------------------------------------------
-- m2s means master-to-slave
-- s2m means slave-to-master
-- ------------------------------------------------------------------------

signal wb_tag_m2s  : std_ulogic_vector(02 downto 0); -- request tag
signal wb_adr_m2s  : std_ulogic_vector(31 downto 0); -- address
signal wb_dat_s2m  : std_ulogic_vector(31 downto 0); -- read data
signal wb_dat_m2s  : std_ulogic_vector(31 downto 0); -- write data
signal wb_we_m2s   : std_ulogic;                     -- read/write
signal wb_sel_m2s  : std_ulogic_vector(03 downto 0); -- byte enable
signal wb_stb_m2s  : std_ulogic;                     -- strobe
signal wb_cyc_m2s  : std_ulogic;                     -- valid cycle
signal wb_lock_m2s : std_ulogic;                     -- exclusive access request
signal wb_ack_s2m  : std_ulogic;                     -- transfer acknowledge
signal wb_err_s2m  : std_ulogic;                     -- transfer error
```

# Creación de un periférico Wishbone

```
-- Wishbone bus interface (available if MEM_EXT_EN = true) --
wb_tag_o    => wb_tag_m2s,          -- request tag
wb_adr_o    => wb_adr_m2s,          -- address
wb_dat_i    => wb_dat_s2m,          -- read data
wb_dat_o    => wb_dat_m2s,          -- write data
wb_we_o     => wb_we_m2s,           -- read/write
wb_sel_o    => wb_sel_m2s,          -- byte enable
wb_stb_o    => wb_stb_m2s,          -- strobe
wb_cyc_o    => wb_cyc_m2s,          -- valid cycle
wb_lock_o   => wb_lock_m2s,         -- exclusive access request
wb_ack_i    => wb_ack_s2m,          -- transfer acknowledge
wb_err_i    => wb_err_s2m,          -- transfer error
```

- Debemos conectar las señales del bus al neorv32
- Debemos instanciar el periférico en el top-level, conectándolo a las señales del bus
- El desarrollador de neorv32 proporciona un periférico sencillo que instancia una memoria de tamaño configurable

# Creación de un periférico Wishbone

## Probando un periférico se stros

```
-- ------------------------------------------------
-- Instance the Wishbone peripheral
-- ------------------------------------------------

myperiph_inst: entity neorv32.wb_stub
  generic map ( WB_ADDR_BASE => x"90000000",
                WB_ADDR_SIZE => 16 )
  port map (
    wb_clk_i    => std_ulogic(iCEBreakerv10_CLK),
    wb_rstn_i   => std_ulogic(iCEBreakerv10_BTN_N),
    wb_adr_i    => wb_adr_m2s,
    wb_dat_i    => wb_dat_m2s,
    wb_dat_o    => wb_dat_s2m,
    wb_we_i     => wb_we_m2s,
```

- Debe ous wishb
- Debe ous al neorv
- Debemos instanciar el periférico en el top-level, conectándolo a las señales del bus
- El desarrollador de neorv32 proporciona un periférico sencillo que instancia una memoria de tamaño configurable

# Añadir source al proyecto

- No debemos olvidar indicar a las herramientas de implementación que deben incluir el (los) fichero(s) .vhd nuevo(s)
- Para esto, modificaremos `neorv32/setups/osflow/filesets.mk`

# Añadir source al proyecto

```
  $(RTL_CORE_SRC)/neorv32_xirq.vhd

NEORV32_PERIPH_SRC := \
  $(RTL_CORE_SRC)/../periph/myperiph.vhd \
  $(RTL_CORE_SRC)/../periph/wb_stub.vhd

# Before including this partial makefile, NEORV32_MEM_SRC ne
# (containing two VHDL sources: one for IMEM and one for DME

NEORV32_SRC := ${NEORV32_PKG} ${NEORV32_APP_SRC} ${NEORV32_M

NEORV32_SRC += ${NEORV32_PERIPH_SRC}

ICE40_SRC := \
  devices/ice40/sb_ice40_components.vhd
```

```
  $(RTL_CORE_SRC)/neorv32_top.vhd \
  $(RTL_CORE_SRC)/neorv32_trng.vhd \
  $(RTL_CORE_SRC)/neorv32_twi.vhd \
  $(RTL_CORE_SRC)/neorv32_uart.vhd \
  $(RTL_CORE_SRC)/neorv32_wdt.vhd \
  $(RTL_CORE_SRC)/neorv32_wishbone.vhd \
  $(RTL_CORE_SRC)/neorv32_xirq.vhd

# Before including this partial makefile, NEORV32_MEM_SRC ne
# (containing two VHDL sources: one for IMEM and one for DME

NEORV32_SRC := ${NEORV32_PKG} ${NEORV32_APP_SRC} ${NEORV32_M

ICE40_SRC := \
  devices/ice40/sb_ice40_components.vhd
```

- Cambios marcados en verde a la izquierda
- En este ejemplo se han añadido 2 ficheros (separados por ‘/’) por si quieren añadir periféricos más complejos, pero para empezar, con `wb_stub.vhd` es suficiente
- Los sources se pueden poner (por ejemplo) en neorv32/rtl/periph (tendréis que crear esa carpeta)

# Reimplementar

- Ya que se ha modificado el hardware, será necesario volver a hacer la implementación del microprocesado (síntesis + PnR + bitstream)
- `make BOARD=iCEBreaker MinimalBoot`

# Y el software?

- Ahora debemos modificar nuestro programa en C para que pueda acceder a los registros de este periférico
- Es fundamental mirar la documentación de `neorv32_cpu.h`, en particular las funciones:
  - `neorv32_cpu_load_unsigned_word`
  - `neorv32_cpu_load_unsigned_half`
  - `neorv32_cpu_load_unsigned_byte`
  - `neorv32_cpu_store_unsigned_word`
  - `neorv32_cpu_store_unsigned_half`
  - `neorv32_cpu_store_unsigned_byte`

# Modificaciones hardware

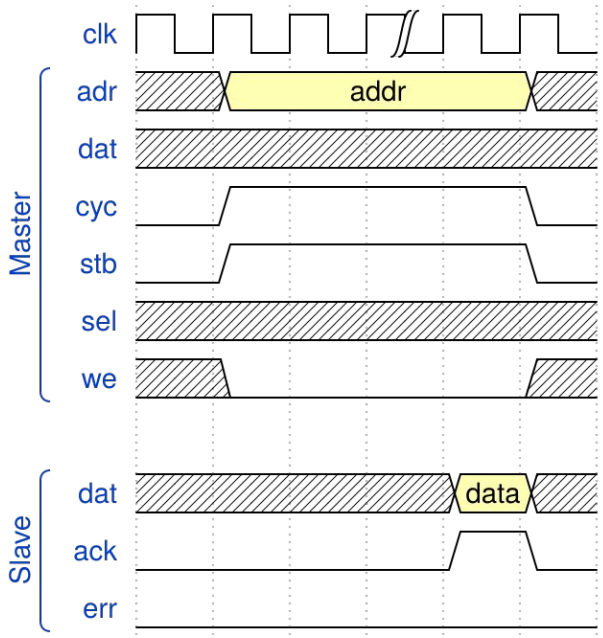
- En el fichero `wb_stub.vhd` aparece la implementación del periférico
- Se puede modificar el VHDL para añadir nuevas funcionalidades

# ¿Y el modo pipeline?

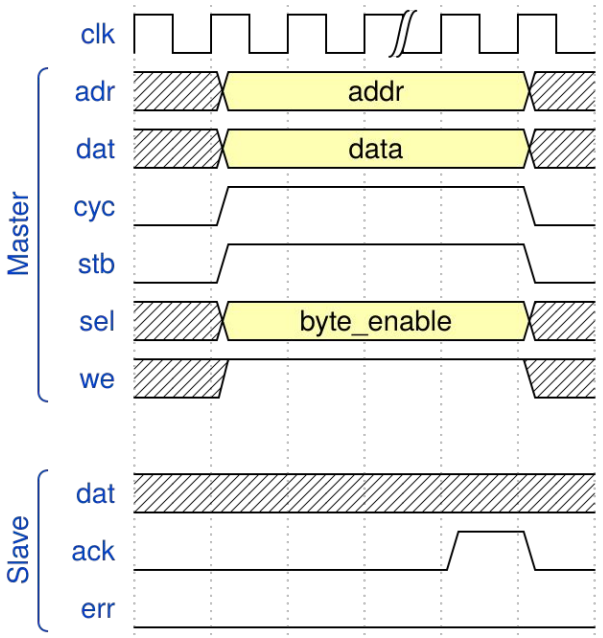
Para empezar estamos usando el modo clásico, pero el modo pipeline puede tener cierta ventaja en algunos casos

Veamos la diferencia entre los ciclos de bus en cada modo

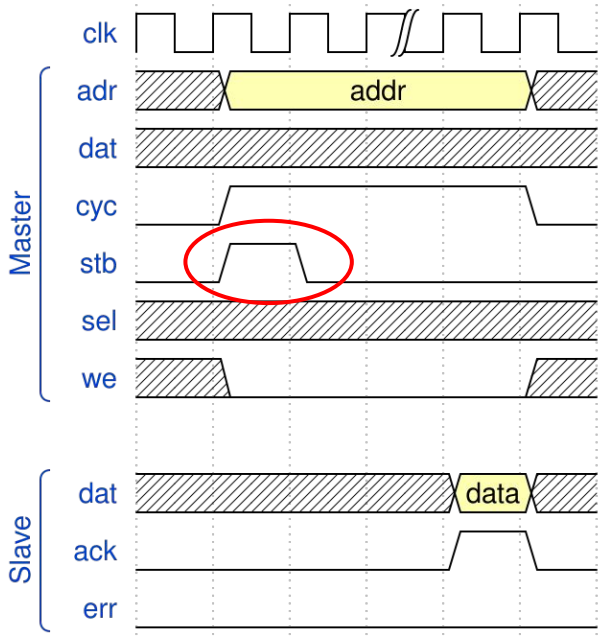
Wishbone read cycle, classic mode



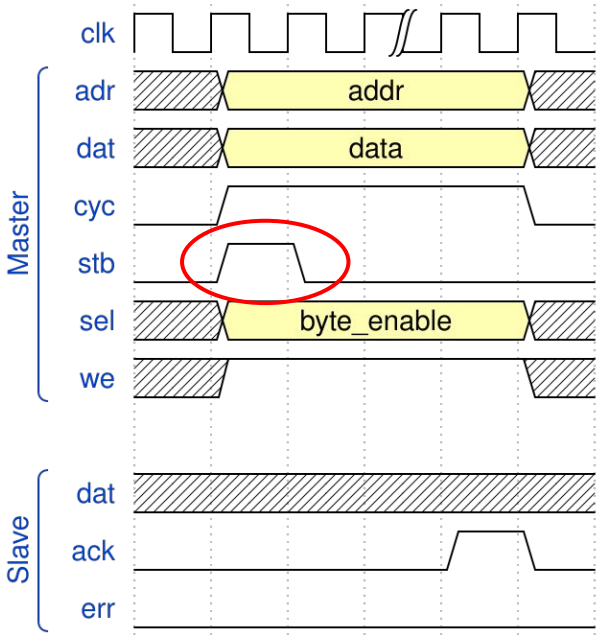

Wishbone write cycle, classic mode




Wishbone read cycle, pipelined mode




Wishbone write cycle, pipelined mode



# ¿Y el modo pipeline?

- Útil para cuando tienes periféricos que necesitan que una condición se cumpla sólo durante UN ciclo de reloj
- Por ejemplo un contador que cuenta si enable = ‘1’, puede usar ‘stb AND cyc’ de enable
- Por ejemplo una FIFO necesita que write_enable = ‘1’ sólo un ciclo por cada dato que vayas a escribir
  - Y lo mismo, con read_enable, al leer

# ¿Y el modo burst?

- A fecha de diciembre 2021, el modo ráfaga NO está implementado en el `neorv32`
- Esto significa que no necesitaremos la señal **stall** de wishbone (de hecho ni siquiera existe en la implementación de wishbone del micro)

# Referencias


- Stefan Nolting, The NEORV32 RISC-V Processor: Datasheet
- Stefan Nolting, The NEORV32 RISC-V Processor: User Guide
- Opencores, Wishbone B4: WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores