System-on-Chip (SoC)

Hipólito Guzmán Miranda
Departamento de Ingeniería Electrónica
Universidad de Sevilla
hguzman@us.es

Learning objectives

- Know how to plan the architecture of a System-on-Chip
- Understand which tasks are better suited to be solved by software and which are better suited to be solved by hardware
- Know what mechanisms can be used to communicate the hardware modules with the software program

Contents

Contents

System-on-Chip

- System-on-Chip (SoC) means literally to have a complete system inside a silicon chip
- In this context, it means to have a microprocessor that runs software + configurable and/or custom hardware modules
- Microprocessor + FPGA *inside the same chip*
- The best of both worlds! (µP + FPGA)
- Don’t need off-chip communication to communicate HW with SW

HW/SW partitioning

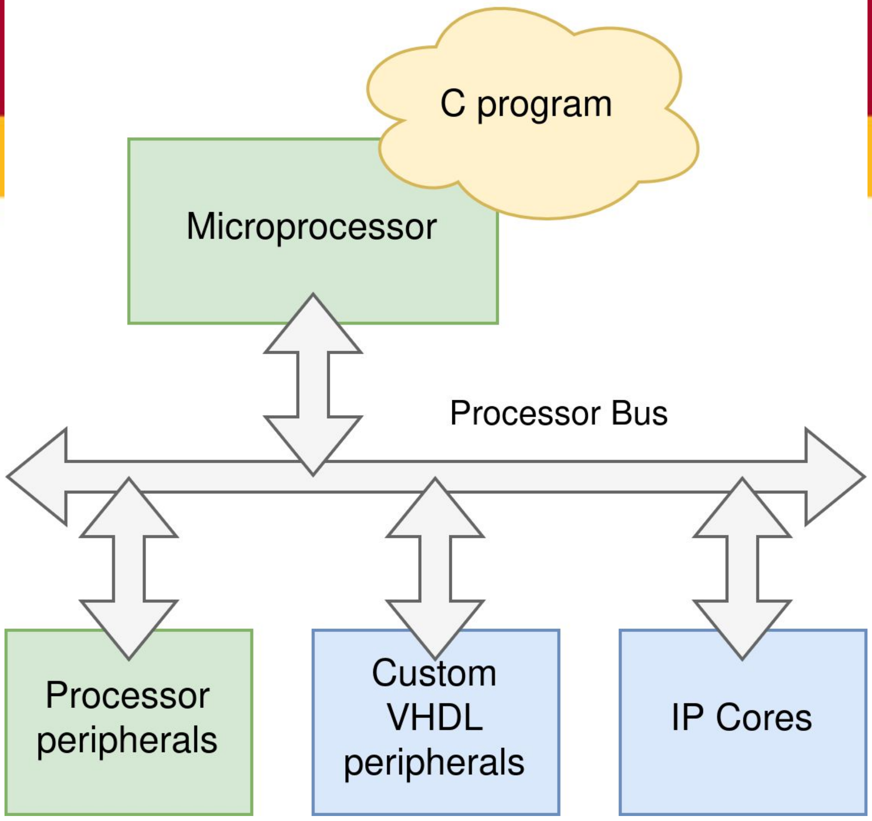
Microprocessors today are pretty powerful so:

- SW
  - When you don’t need that much performance
  - Tasks that are sequential in nature
  - Communications such as USB, TCP/IP, etc
  - Reuse software libraries
- HW
  - When you need a lot of performance
  - Tasks that are concurrent/parallel in nature
  - When you need **exact** timing
  - Reuse hardware IP cores

HW/SW partitioning

- Everything you can do in SW, you do in SW
- Things that have to run really fast, or for which you need exact time synchronization, you do in HW
- In case of doubt, it is typically faster to prototype a C code, profile it and only write a custom VHDL peripheral if needed
- It doesn't hurt to know a bit about software!
    - So you can design software-friendly hardware

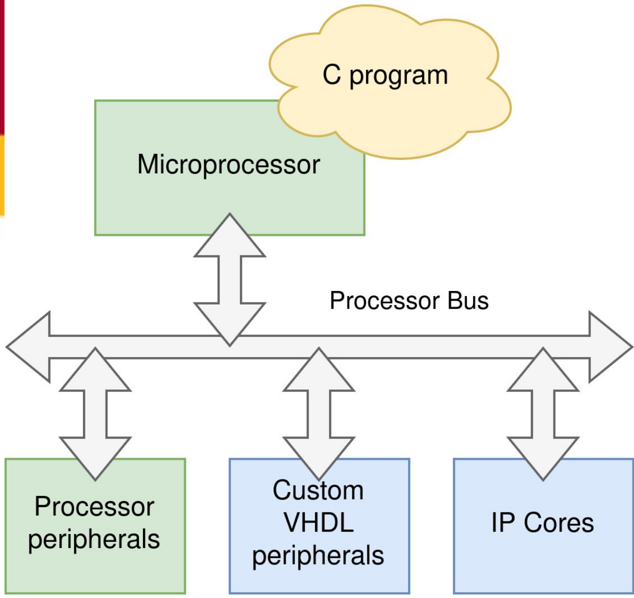
Typical SoC architecture



- Microprocessor
  - Know your architecture!
- Bus/Buses
  - Connects microprocessor to peripherals
- Peripherals
  - Some come with the microprocessor
  - Some are custom-made
    - We write those in VHDL or Verilog
  - We can also reuse IP cores

Contents

Buses

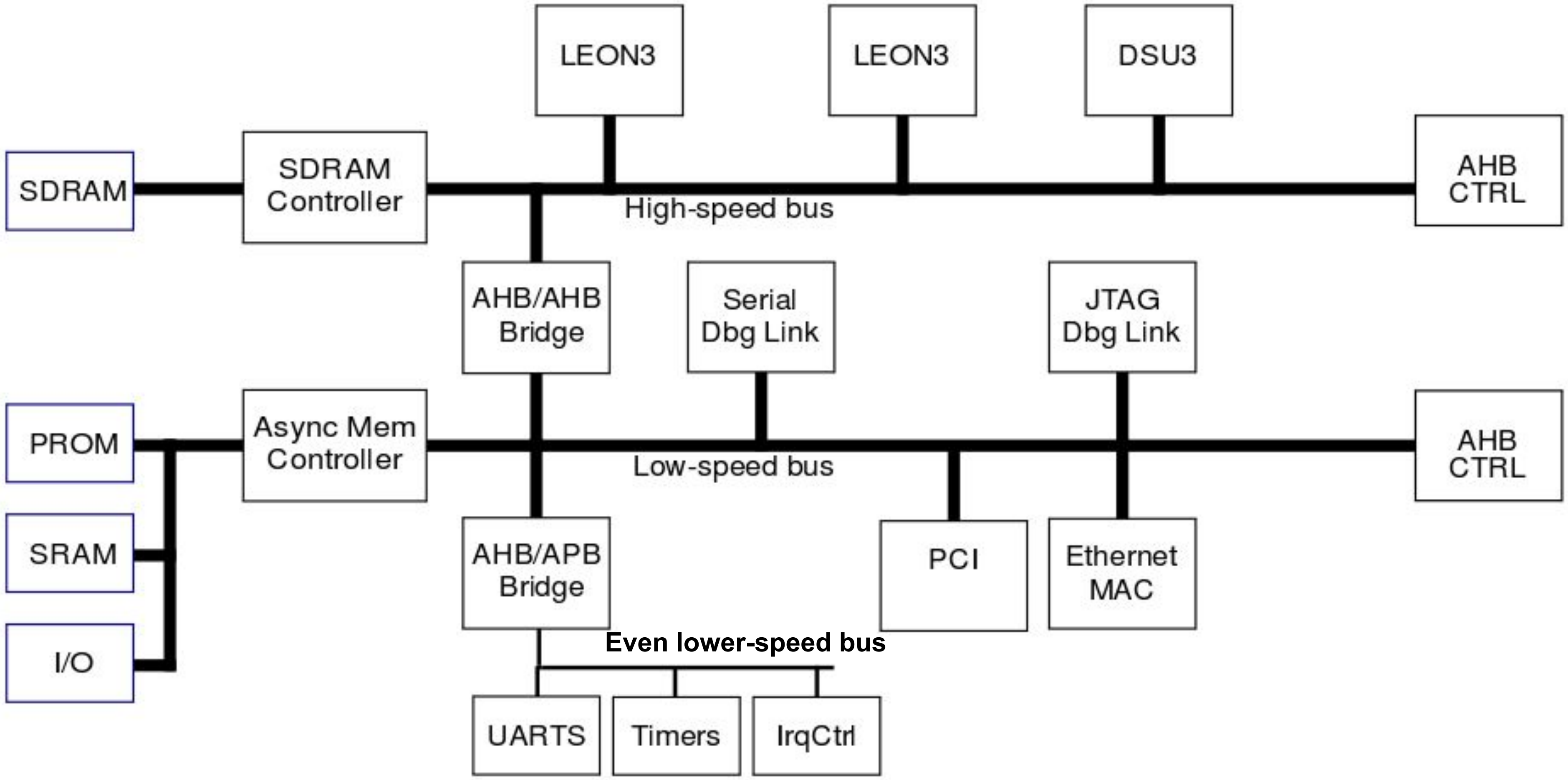


- The microprocessor is connected to a bus to which the peripherals are also connected
- Some registers of the peripherals are accessible through the memory map of the microprocessor
- This means that writing from the C code to a specific address will write a value into a specific VHDL register
- And reading from the C code from a specific address will read a value from a specific VHDL register and make it available to the software

Bus hierarchies

- Sometimes a peripheral in a bus is a **bridge** to another bus
- For when you need to ‘speak’ a different language
    - e.g., my microprocessor only supports AXI buy my peripheral is Wishbone
- For when you want to separate fast peripherals from slow peripherals
    - Put the slower peripherals in a less performance, but easier to use, bus

Bus hierarchy: Leon3 multi-core example



Contents

AXI (Advanced eXtensible Interface)

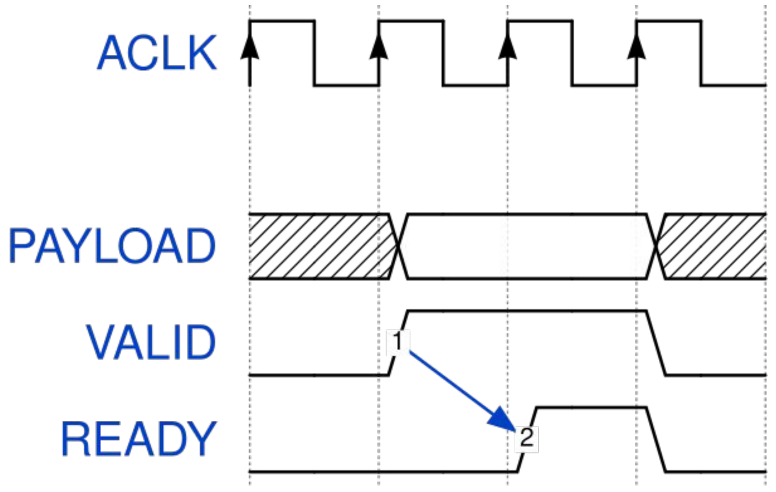
- AXI is a interconnection specification
- Higher bandwidth with respect to older bus standards
- Is part of AMBA (Advanced Microprocessor Bus Architecture)
    - Like the older AHB (Advanced High-performance Bus) and APB (Advanced Peripheral Bus)
- Supported by AMD Xilinx Programmable SoCs
- Highly popular nowadays

The AXI bus

- Comes in three flavors:
  - **AXI4** (more powerful and complex)
  - **AXI4-Lite** (reduced version, not as powerful but easy to develop peripherals that support it)
  - **AXI4-Stream** (used when data should be transmitted in streams instead of copied to memory positions)

Valid/Ready signals

- A wants to write data to B
- A puts the data and raises `valid`
- A leaves `valid` at ‘1’ until B raises `ready`
- The moment `valid` and `ready` are both ‘1’ is the moment where B is actually reading A’s data
    - This is called one ***beat***



Source: Wikipedia

The AXI bus protocol

AXI4

- Data bus width can be up to 1024 bytes
  - Typical values are 32 and 64
- Defines multiple channels:
  - Read Address channel (AR)
  - Read Data channel (R)
  - Write Address channel (AW)
  - Write Data channel (W)
  - Write Response channel (B)
- Supports bursts, which are composed of beats
- Size of beats can be configured
  - No need to always use the full bus width
- **Good for:** sending big blocks of data to different memory locations

AXI4-Lite

- Simpler subset (removes some AXI4 signals)
- Data bus width can only be 32 or 64 bits
- Same 5 channels
- Bursts of only 1 beat
- Beats always of full data bus width
- **Good for:** reading/writing on individual registers

AXI4-Stream

- Even simpler subset
- Bus width can be *any* integer number of bytes
- For high-speed streaming
- Unidirectional data flow
  - If you need to send data back, you could use for example a second AXI4-Stream
- A single channel:
  - An AXI Stream is like a Write Data Channel, but:
    - No bursts: data is organized into packets, frames and data streams
    - No limit on data length (can be continuous)
- **Good for:** streaming data

Contents

Planning a SoC architecture

First steps

- List functionalities
- All that can be made in software, should be made in software
  - Reuse software libraries whenever possible
- The rest should be done in hardware
  - If available, use IP cores when possible
  - Develop the most performance-critical designs in a traditional HDL such as VHDL or Verilog
- When possible, avoid back-and-forth communication between HW and SW
  - It takes time!

Hunting for performance

- Performance optimizations may happen in the software side, too
  - Not the most popular topic among engineers that do electronics
  - But sometimes could be the answer to speed-up a bit of already-existing software
  - Run the program using a profiler
    - Identify how many times each function is run and how much time is spent on each function
  - If using a multi-core processor, reserve one or more cores for your application
    - Avoid an OS creating tasks that slow your application down
  - The C `asm()` keyword allows to directly embed assembly code
    - Normally not needed, but nice to know

Contents

Custom peripherals

- FPGA vendors provide ‘wizards’ that generate the HDL that ‘speaks’ the bus protocol
- In AMD Xilinx Vivado, use the wizard to generate a peripheral scaffold that includes the circuitry to communicate through AXI, then add your custom VHDL code
- Before adding complex functionalities, test first basic register read/write capabilities!

Contents

Other interesting IP cores

- DMA (Direct Memory Access)
  - It moves data so the microprocessor doesn’t have to do so, freeing up computing power
- Interrupt controller
  - Your custom peripherals can generate interrupts for the microprocessor
- Application-specific:
  - Cryptography/security, signal processing, networking, …

System-on-Chip

Contents

Conclusions

- SoC design is a matter of planning/partitioning…
- … and (sometimes) designing a bus hierarchy …
- … and connecting the appropriate buses …
- … and reusing software code and IP cores …
- … and of course actually writing your actual, application-specific C / VHDL code!
- Planning is strongly dependent on previous experience
- But nevertheless, we have simple rules of thumb to begin planning
- Take it easy and build complex systems incrementally, block by block
- It opens many job opportunities in the FPGA/SoC world!

Contents

Bibliography

- embedded.com, “Software-friendly hardware”.
- Wikipedia, “Advanced eXtensible Interface”.
- ARM, “AMBA AXI and ACE Protocol Specification”.
- ARM, “AMBA AXI-Stream Protocol Specification”.
- AMD Xilinx, “Vivado Design Suite User Guide: Creating and Packaging Custom IP (UG1118)”

Learning results

- Of a number of tasks, determine which are more suitable to be implemented in software and which are more suitable to be implemented in hardware
- Be able to interpret a bus hierarchy seen in a SoC schematic
- Understand what mechanisms do we have available to communicate hardware with software, and which are more suitable for different kind of applications