ECE 385 – Digital Systems Laboratory

Lecture 13 – More Experiment 7, IPs and SoCs

Zuofu Cheng, Deming Chen

Spring 2017

[Link to Course Website]

ECE ILLINOIS

ILLINOIS

Experiment 7 Goals

- Create a working NIOS II/e based SoC which performs addition from switches into LEDs
- Behavior is similar to Lab 4, but using software (written in C) instead of hardware
- Program must execute from SDRAM and use PIO modules wired to LEDs
- May use provided pin-mapping file (DE2-115.QSF)
  - Simplifies pin-mapping for the (many) SDRAM signals
  - May need to reconcile names with HDL and QSYS
- Need to add an I/O constraint for SDRAM

Experiment 7 Demo Points

- The green LED blinks on the FPGA (1.5 points).
- The accumulator clears to 0 by pressing 'Reset' (0.5 point).
- The accumulator increments by the value on the switches by pressing 'Accumulate' (1 point).
- The accumulator overflows to 0 at 255 (0.5 point).
- The input and output constraints are fully-specified (valid Timequest Timing Analyzer analysis after compilation) (0.5 point).
- Correctly answer one TA-designated embedded question (1 point).

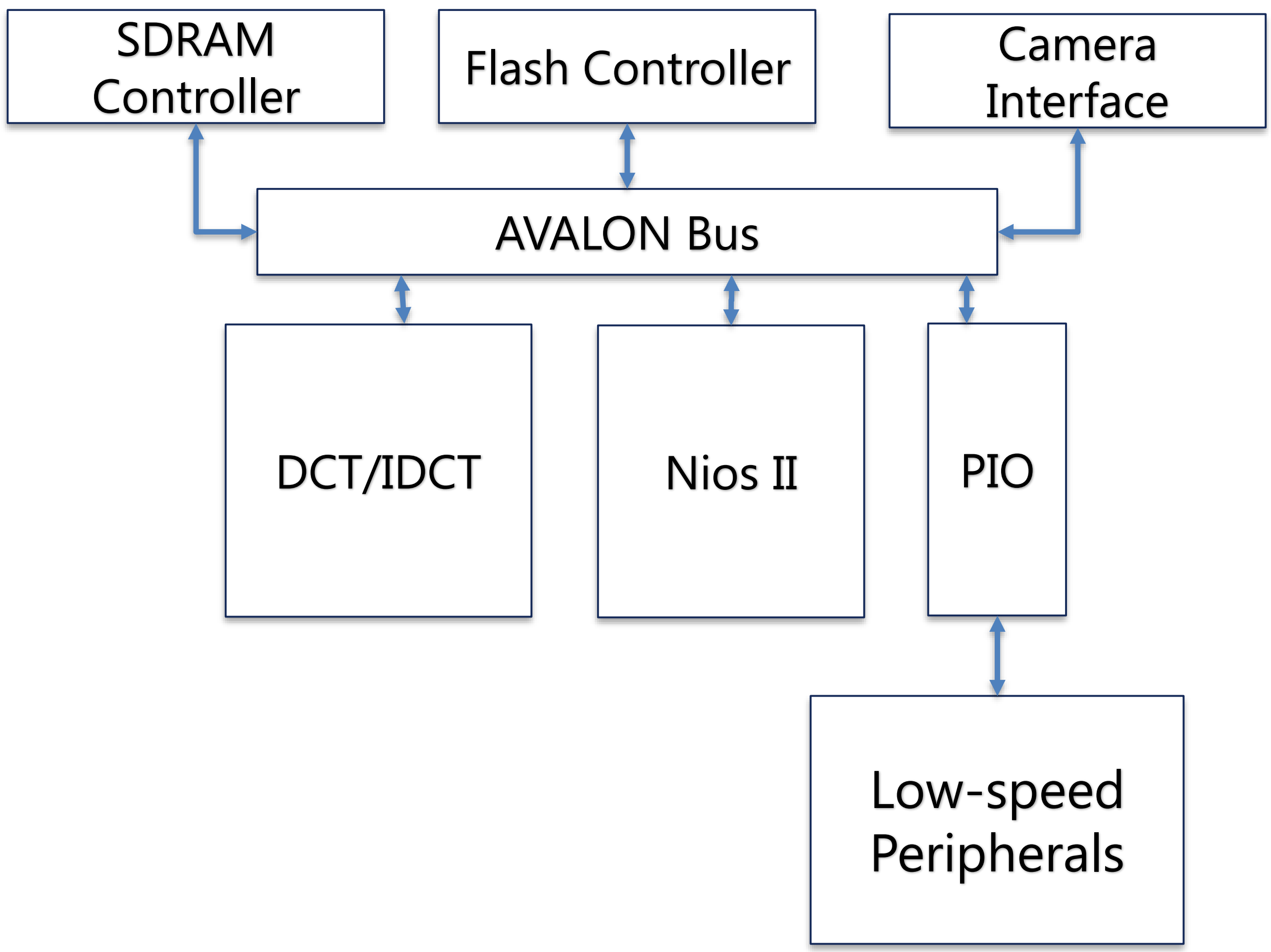
Experiment 7 Hints

- Make sure you import main.c correctly, no program entry point = no .ELF file (binary, like .EXE in windows)
- You may use debugger if you have a working build setup, switch Eclipse into “debug” view
- To verify SDRAM settings, make sure your ports out of the SoC map properly into the top level
- Make sure you have IO constraints in .SDC (in addition to clock = 50 MHz constraint)
- You need 3 inputs, system reset, clear, sum
  - Do not rely on system reset to clear
  - System reset may wipe RAM contents

Motivations for System-on-Chip (SoC)

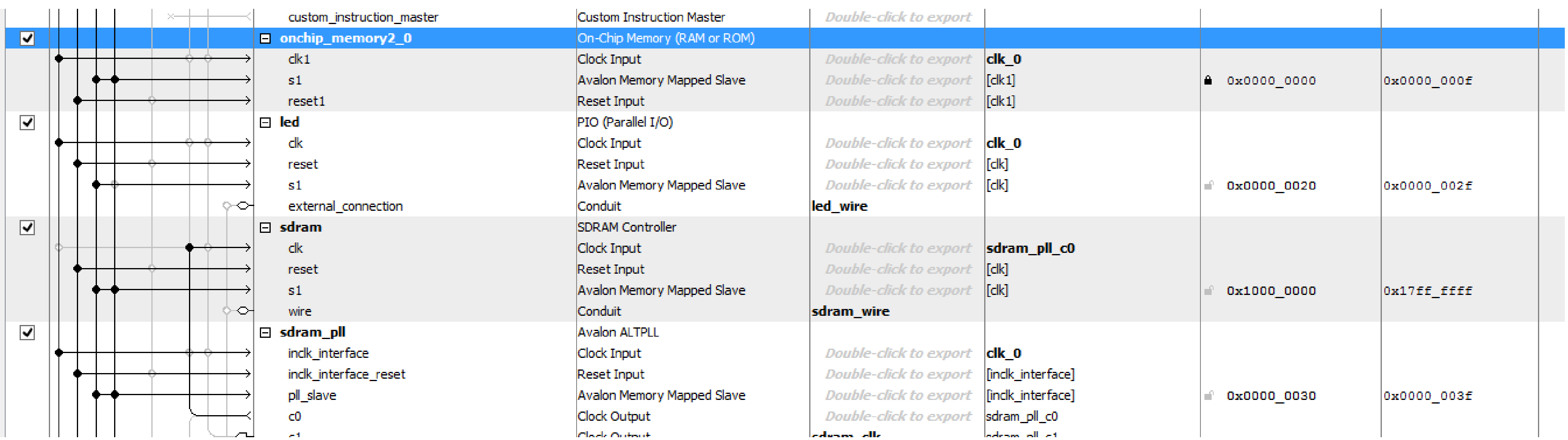
- So far, we've been designing in hardware (SystemVerilog)
- Good for tasks which require high performance (DSP, video processing, graphics)
- However – all systems need lots of low performance tasks (getting data in and out of system, formatting data, debugging, user interface)
- Want to use software for lower performance tasks

Typical SoC System (Video Encoding)



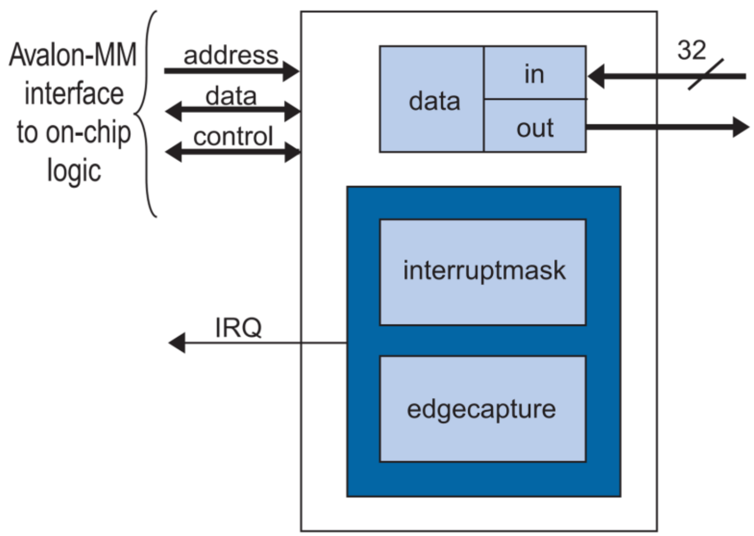
AVALON Memory Mapped Bus

- AVALON MM bus is a 32-bit memory mapped interface
- Each device on the bus is assigned a block of addresses
  - Device could be RAM, ROM, peripheral, etc.
  - These assignments are reconfigurable in Qsys
- This is how the NIOS II interfaces to memory and I/O



PIO (Parallel I/O) Module

- We'll use the PIO module as a bridge from AVALON to FPGA logic
- PIO modules may be input (to software), output (to FPGA fabric), or bidirectional
  - Note that restrictions about not having internal tristate buffers still apply
- Control registers memory mapped to addresses assigned by QSYS



PIO Register Map

- Base address (offset 0) is assigned via Qsys
- Additional addresses are offset * 4 addresses above base

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	`data`	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	`direction` (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	`interruptmask` (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	`edgecapture` (1) , (2)		R/W	Edge detection for each input port.				
4	`outset`		W	Specifies which bit of the output port to set.				
5	`outclear`		W	Specifies which output bit to clear.				

Multiple Addresses

- What if we want to address multiple registers?
- For example, we need to change direction register
- Could manually add offset to pointer every time, but this is very confusing
- Ideas?

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	direction (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture (1), (2)		R/W	Edge detection for each input port.				
4	outset		W	Specifies which bit of the output port to set.				
5	outclear		W	Specifies which output bit to clear.				

Multiple Addresses

- What if we want to address multiple registers?
- For example, we need to change direction register
- Could manually add offset to pointer every time, but this is very confusing
- Ideas?

Offset	Register Name		R/W	(n-1)	...	2	1	0
0	data	read access	R	Data value currently on PIO inputs				
		write access	W	New value to drive on PIO outputs				
1	direction (1)		R/W	Individual direction control for each I/O port. A value of 0 sets the direction to input; 1 sets the direction to output.				
2	interruptmask (1)		R/W	IRQ enable/disable for each input port. Setting a bit to 1 enables interrupts for the corresponding port.				
3	edgecapture (1), (2)		R/W	Edge detection for each input port.				
4	outset		W	Specifies which bit of the output port to set.				
5	outclear		W	Specifies which output bit to clear.				

Multiple Addresses

- Instead lets use a `struct`
- Remember, how are `structs` allocated in C?

```
typedef struct
{
 unsigned int volatile data;
 unsigned int volatile direction;
 unsigned int volatile interrupt_mask;
 ...
} NIOS_PIO_t
```

- **MUST** fill in all fields (why?)
- Then declare

```
NIOS_PIO_t* LED_PIO = (NIOS_PIO_t*) 0xABC;
```

Multiple Addresses (cont)

- Then how do we use?
- `LED_PIO-> data |= 0x8;`
- Remember, `LED_PIO->data` is just shorthand for `(*LED_PIO).data` (if you use “longhand” version, remember that “.” operator happens before * operator, so you need parenthesis
- What if we have fields we don’t want to use?

– typedef struct

```
{
 unsigned int volatile data;
 const char; //skips 1 byte
 unsigned int volatile interrupt_mask;
 ...
} NIOS_PIO_t
```

Embedded Programming Hints

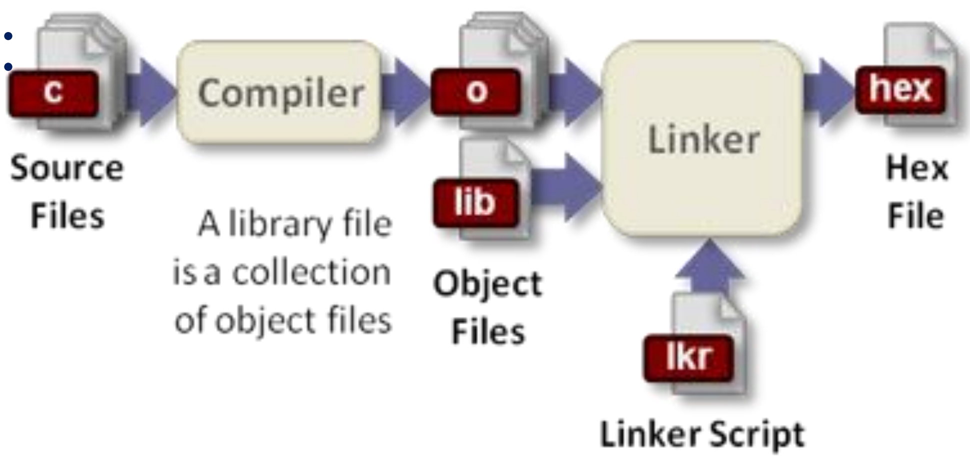
- Note that we're careful to use "`unsigned int`", "`unsigned char`", etc..
- It is good practice to use types of *known* size when we are dealing with system variables (e.g. memory mapped registers)
- We can use common types like "`int`" or "`char`" for "normal" code, e.g. `int num_jobs = 10; float pi = 3.14f`
- It's common to define the types `uint_32t, int_16t`, etc…because compilers use different lengths for different types (check compiler documentation – we're using **GCC for the NIOS II**)
- Can look for a file called "types.h", or just write yourself (easy to do with `typedef`)

Embedded Programming Hints

- Note we have infinite loop in main function, unlike what you are use to.
- Typically we have `int main () {…return 0;}`
- You probably just learned to write that as a way to "get programs to work", but what is the more general case?
- `main` can have parameters (`argc`, `argv`)
- Also different return values other than 0
- Note: return from main -> return control to OS
- What is OS in NIOS II (the way we are running it)?

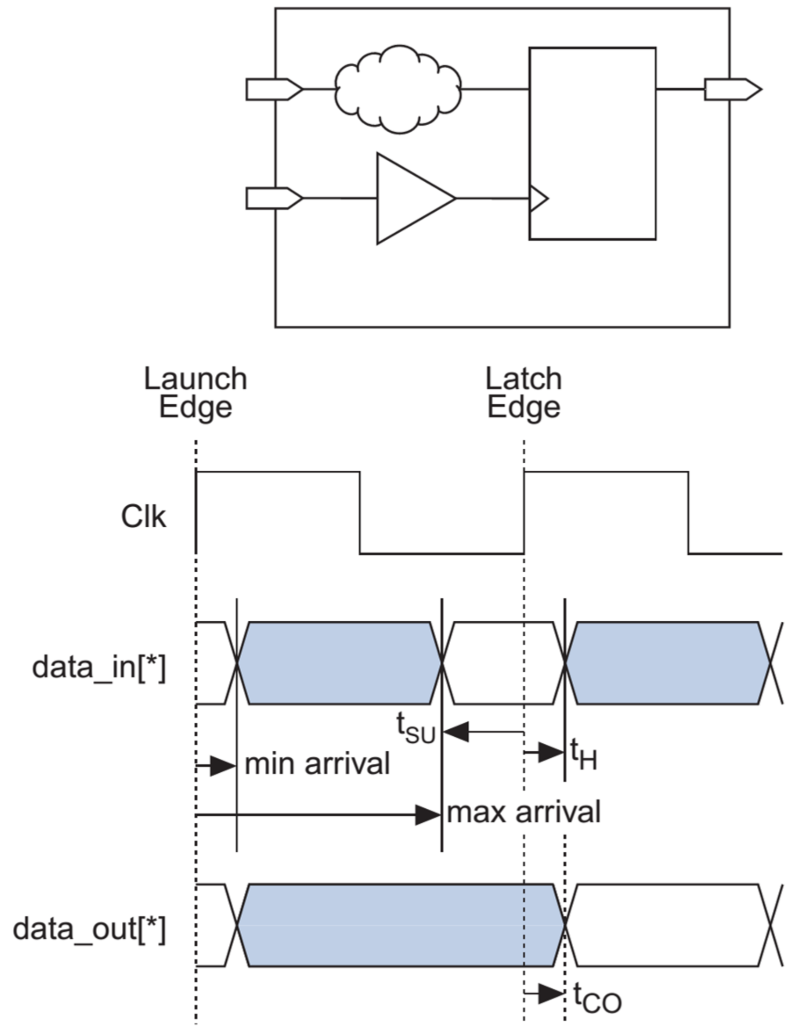
Board Support Package (BSP)

- Peripherals are not the only thing which is configurable in SoC, memory is too!
- This creates a problem without OS…what is this problem?
- BSP contains (among other things) our **linker script**
- What is a linker script? What is a linker?
- Remember :



I/O Constraints

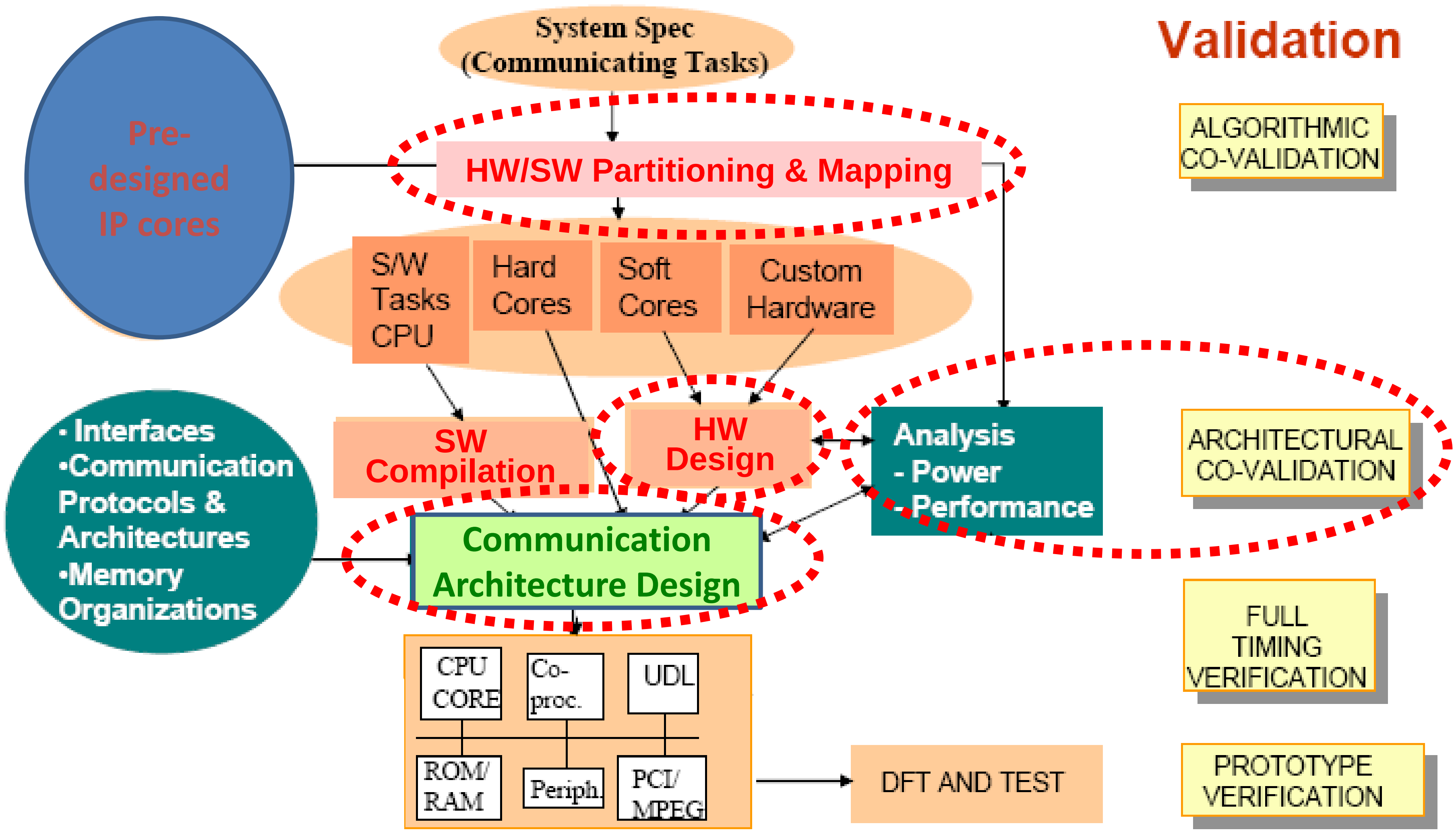
- External signals require setup and hold time constraints.
- This is either done using the GUI (TimeQuest Timing Analyzer) or via manual editing of the .SDC file – an example is provided



Clock and I/O Constraints

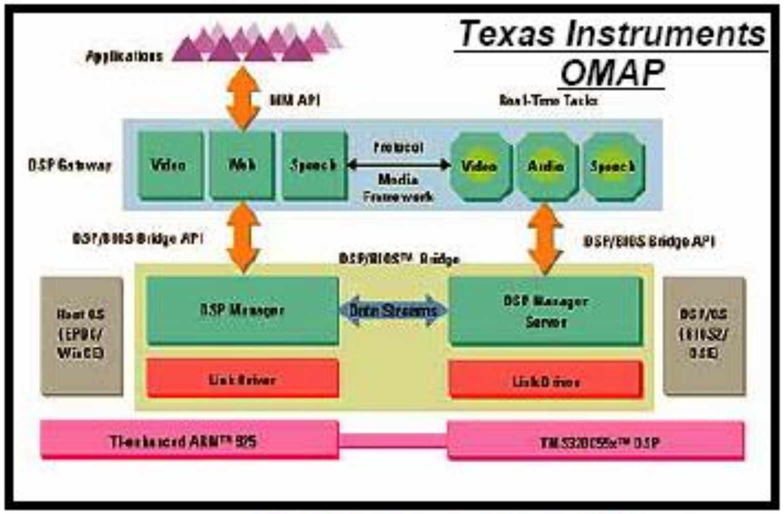
- Inputs require minimum setup time and minimum hold time, this tells the FPGA how much time it has on either side of the clock edge to latch in correct data
- In general, for synchronous I/O, check Altera TimeQuest Cookbook
- In Lab 7, need to add “false path” for switches (they are asynchronous (note, asynchronous signals should have synchronizer)
- The SDRAM has already been constrained for you (input: 2-3ns, output 2ns)

Strengthening the knowledge: SoC Design Flow

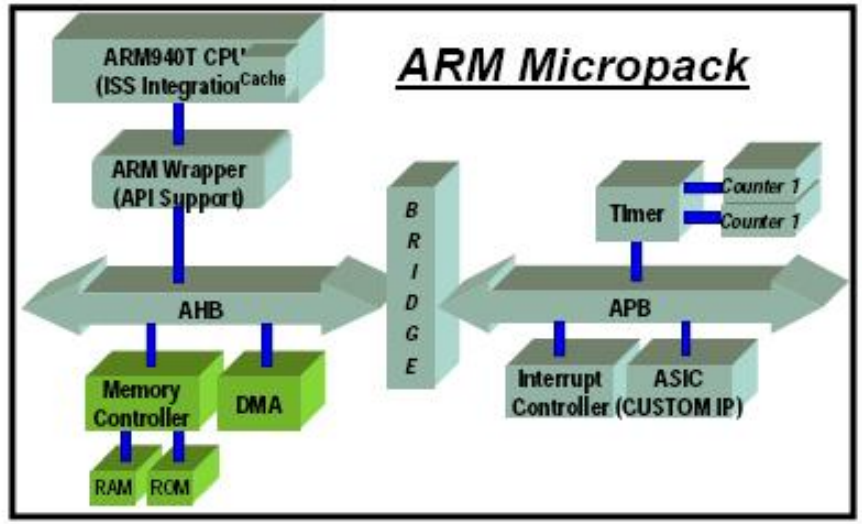


Credit: Prof. Sujit Dey, UC San Diego

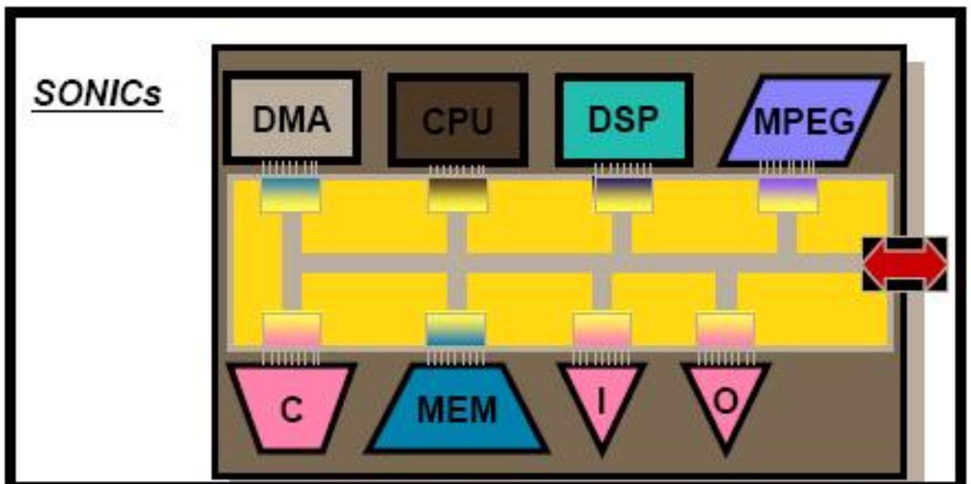
Platform Alternatives



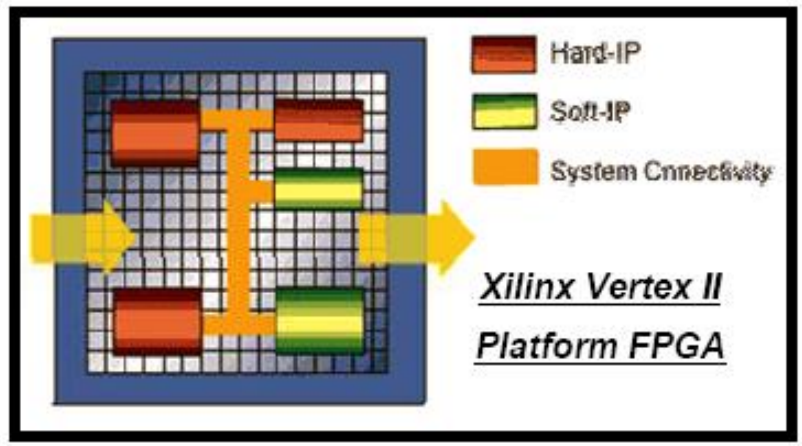
Full Application



Processor-Centric



Communications-Centric



Highly-Programmable

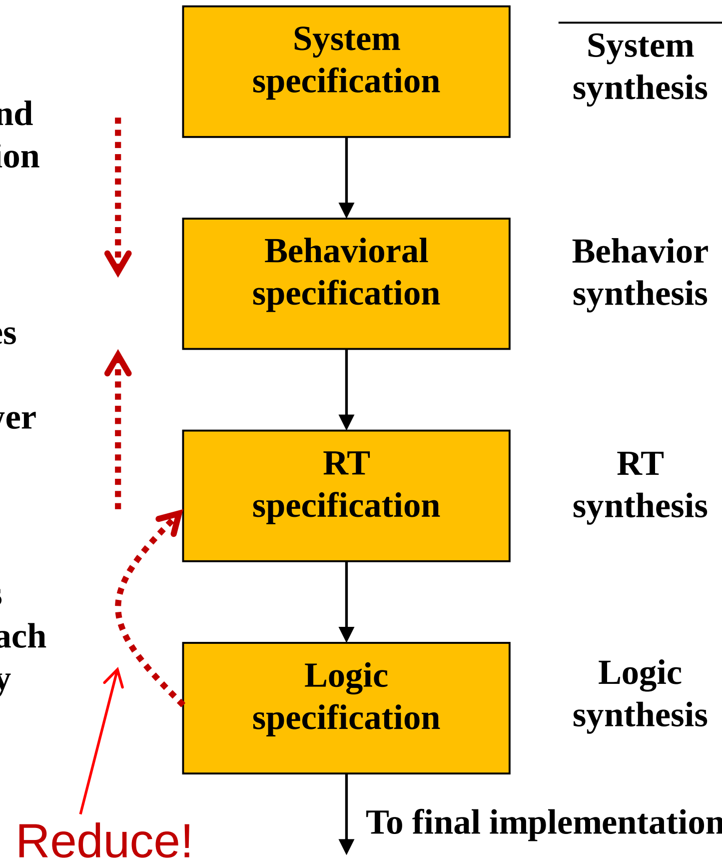
SoC Design Abstractions

- The manner in which we convert our concept of desired system functionality into an implementation

Compilation/Synthesis: **Automates exploration and insertion of implementation details for lower level.**

Libraries/IP: **Incorporates pre-designed implementation from lower abstraction level into higher level.**

Test/Verification: **Ensures correct functionality at each level, thus reducing costly iterations between levels.**



	Compilation/ Synthesis	Libraries/ IP	Test/ Verification
System specification	System synthesis	Hw/Sw/ OS	Model simulat./ checkers
Behavioral specification	Behavior synthesis	Cores	Hw-Sw cosimulators
RT specification	RT synthesis	RT components	HDL simulators
Logic specification	Logic synthesis	Gates/ Cells	Gate simulators

Intellectual Property (IP)

- Building block components (roughly equivalent terms)
  - Macros, cores, IPs, virtual components (VCs)
- Examples
  - Microprocessor core, A/D converter, Digital filter, Audio compression algorithm
- Three types of IP blocks
  - Hard (least flexible)
  - Firm
  - Soft (most flexible)

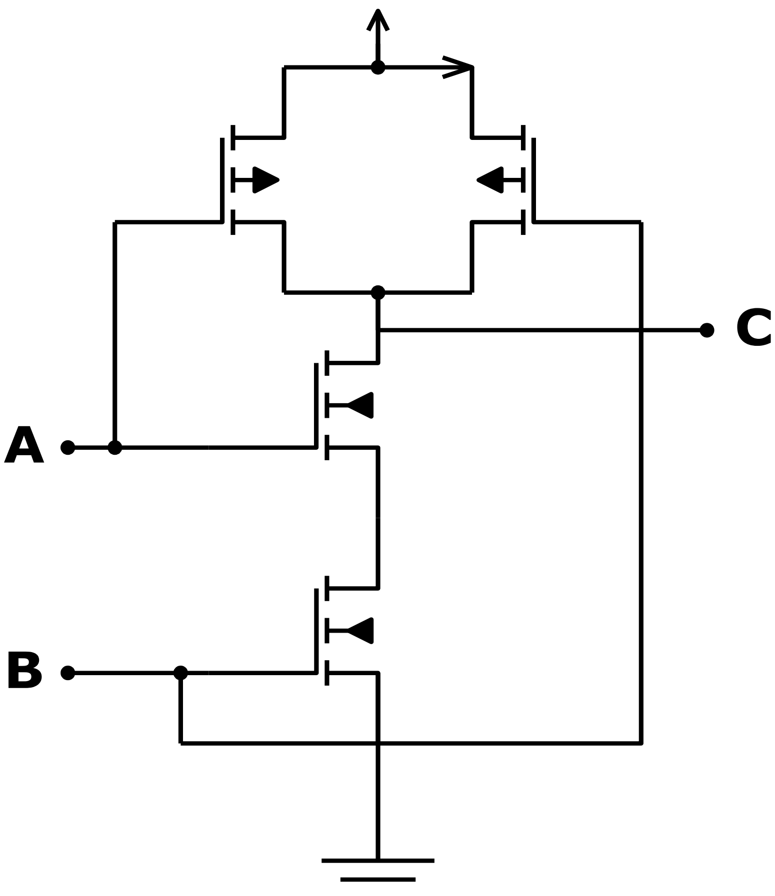
IPs to Deal with the SoC Complexity Problem

- Heavy IP reuse
  - Share costs and risks of developing IP modules
  - Avoids duplication of efforts
- Automation of IP integration
  - Improves time-to-market by reducing time-consuming and error prone manual design
- Verification
  - Helps designers save testbench development time and reach functional coverage goals faster

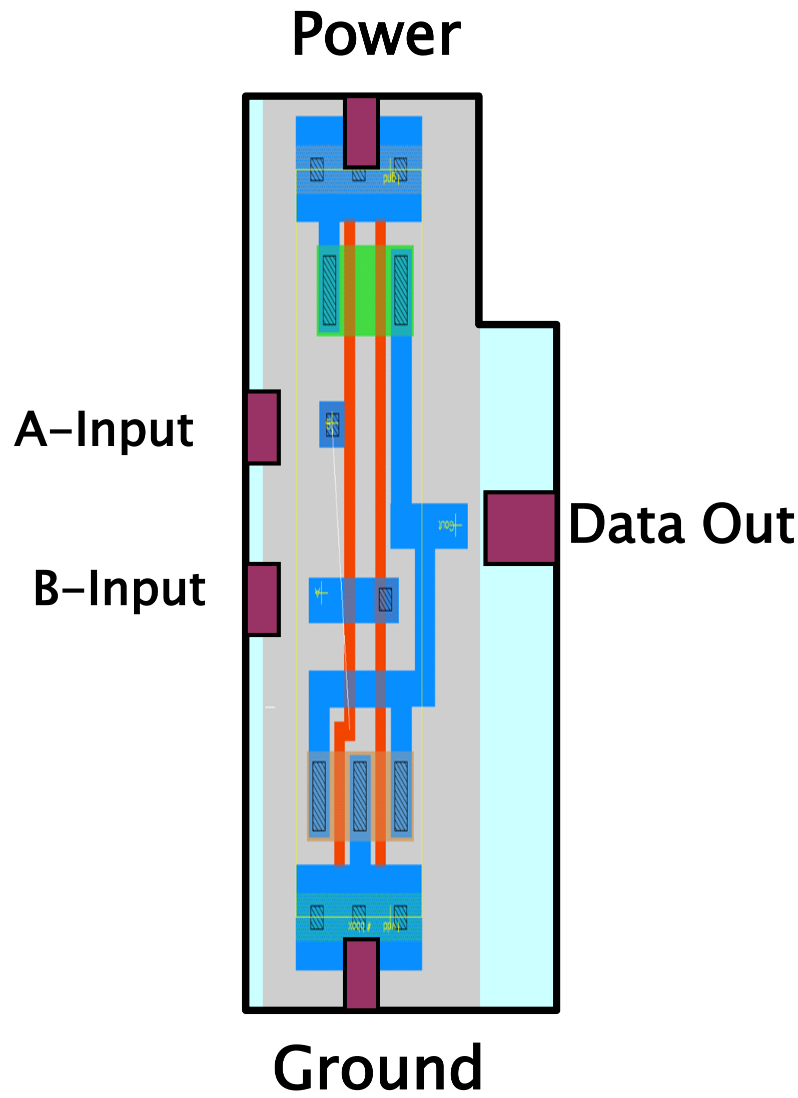
Hard IP

- Delivered in physical form (e.g., GDSII file)
- Fully
  - Designed
  - Placed and routed
  - Characterized for timing, power, etc.
- Tied to a manufacturing process
  - Actual physical layout
  - Fixed shape
- Complete characterization
  - Guaranteed performance
  - Known area, power, speed, etc.
- No flexibility

Fixed Schematics and Layout



Hard Macros



Schematic of a NAND gate

Layout of a NOR gate

Hard IP Examples and Constraints

- A microprocessor core
  - PowerPC, ARM
- AMS (analog/mixed-signal) blocks
  - ADC, DAC, filter
- A phase-locked loop (PLL)
- A memory block design

- Features
  - Deeply process dependent
  - Stricter performance requirements
  - Electrical constraints, such as capacitance, resistance, and inductance ranges
  - Geometric constraints, such as symmetry, dimension, pin location, etc.
  - Need to provide interface for functional and timing verification

Soft IP

- Delivered as synthesizable RTL HDL code (e.g., VHDL or Verilog) – can be SystemC/C/C++ code now.
- Performance is synthesis and process dependent
- Synthesizable Verilog/VHDL/SystemC/C/C++
- Synthesis scripts, timing constraints
- Scripts for testing issues
  - Scan insertion, ATPG (automatic test pattern generation), etc.

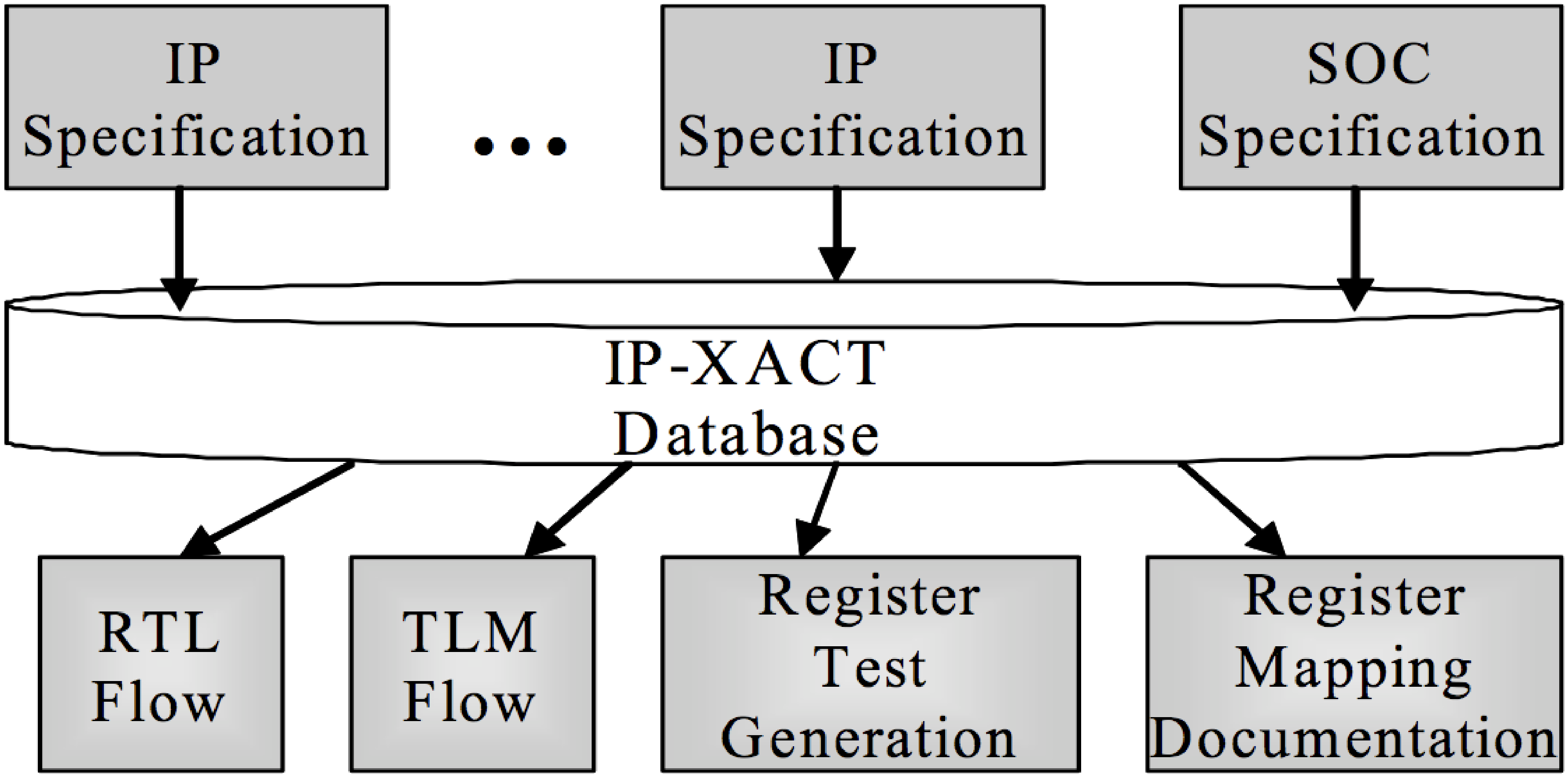
Firm IP Blocks

- Intermediate form between hard and soft IP
  - Some physical design info to supplement RTL
  - RTL or netlist or mixture of both
  - More (or less) detailed placement
  - Limited use beyond specified foundry

Understand IPs

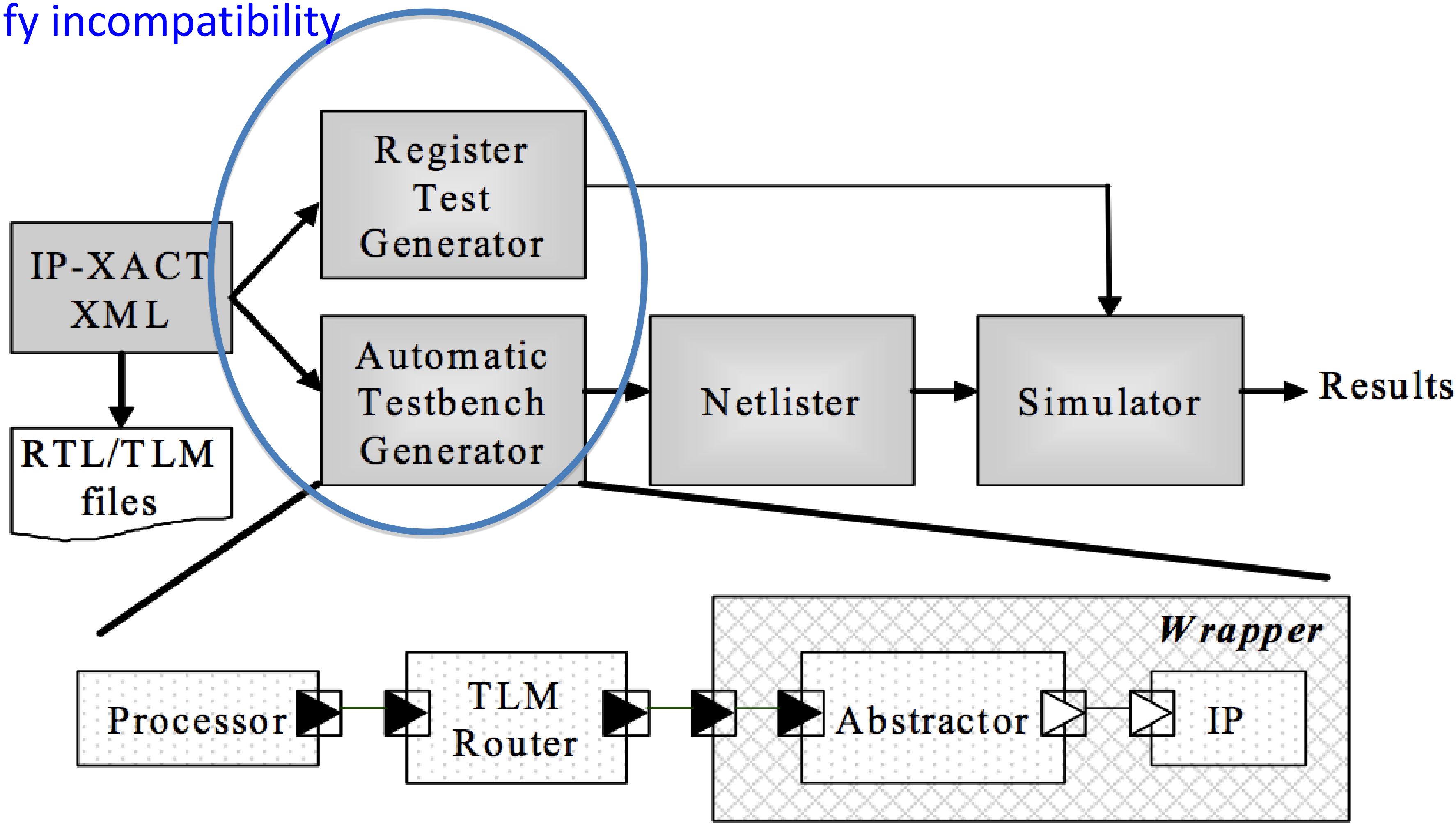
- The quality of IPs and support will be the key to the success of the IP business
- Need to pay much attention on software IP issues
- Need application and system design expertise
- Core-based design is effective on IP/core integration
- Need to develop a combining platform- and core-based design methodology/environment for system designs

IP Integration flow of STMicroelectronics

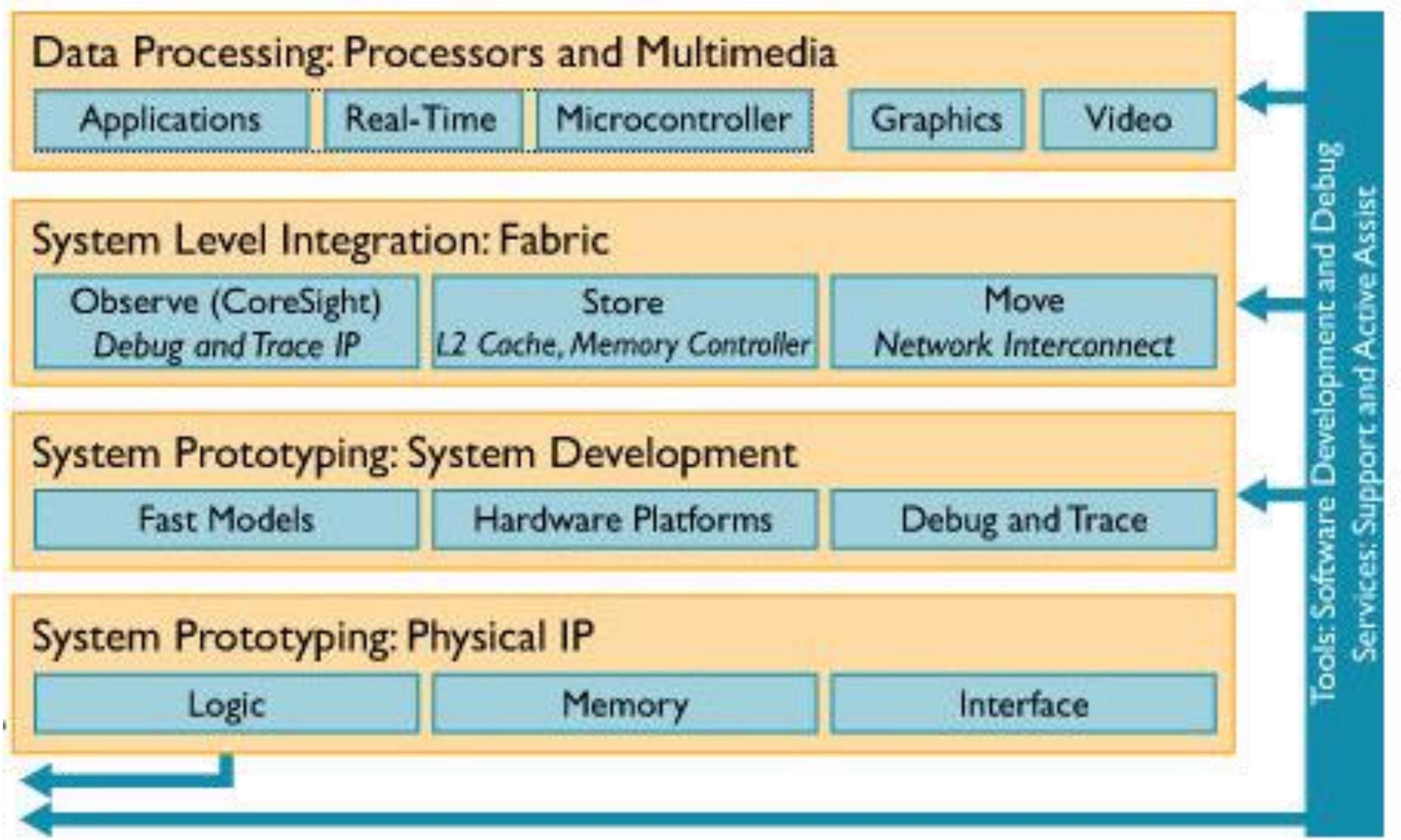


IP Quality Check Flow

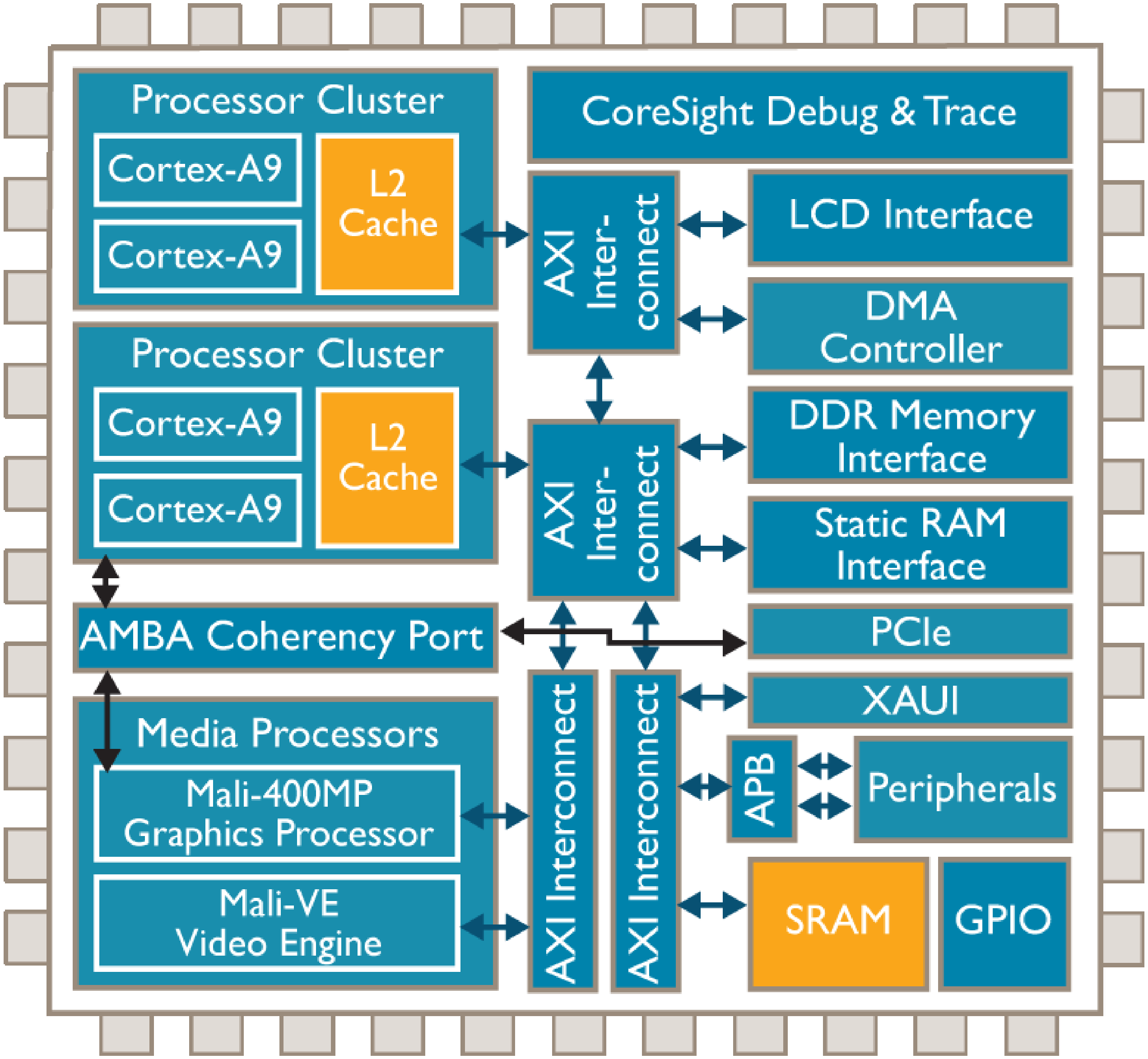
- Verify accuracy
- Verify incompatibility



Case Study: Portfolio of ARM IPs



Memory IP from ARM



System IP from ARM