

Practical System-on-Chip

OSHUG #17: 29 March 2012

Julius Baxter, Jeremy Bennett, opencores.org

- Introduction
 - Systems On A Chip, IP cores, HDL, Implementation technologies (FPGA)
- OpenCores & OpenRISC Project
- Using ORPSoC
- Compiling software for OpenRISC bare metal

System On A Chip

System on a Chip (SoC)

- Integrate many functions onto a single silicon die
- Result of increased IC consolidation
- Enabled by improvements in VLSI process
- Modern high-end SoCs integrate:
 - μ /DSP/graphics processors
 - Memory controllers
 - DRAM, flash
 - Communications
 - USB, ethernet, i2c
 - Bespoke processing, I/O

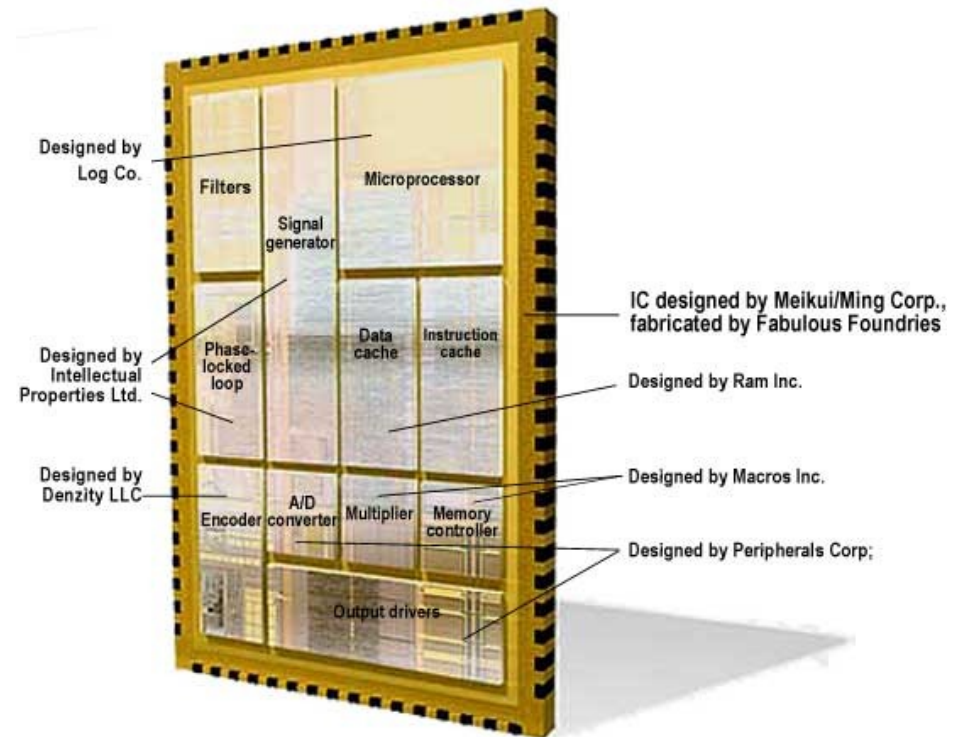
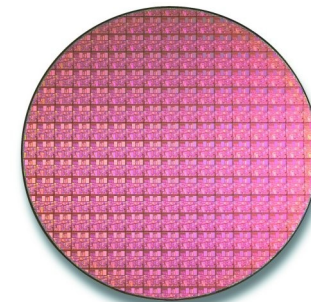
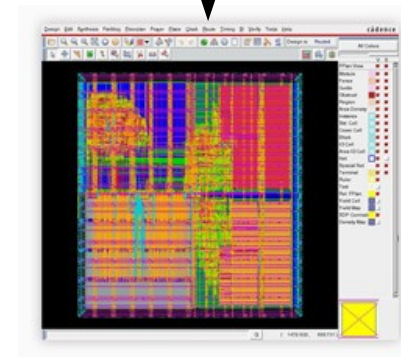
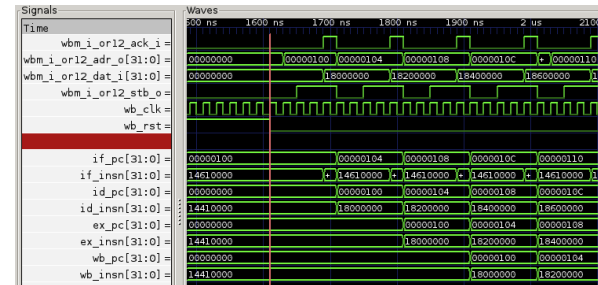


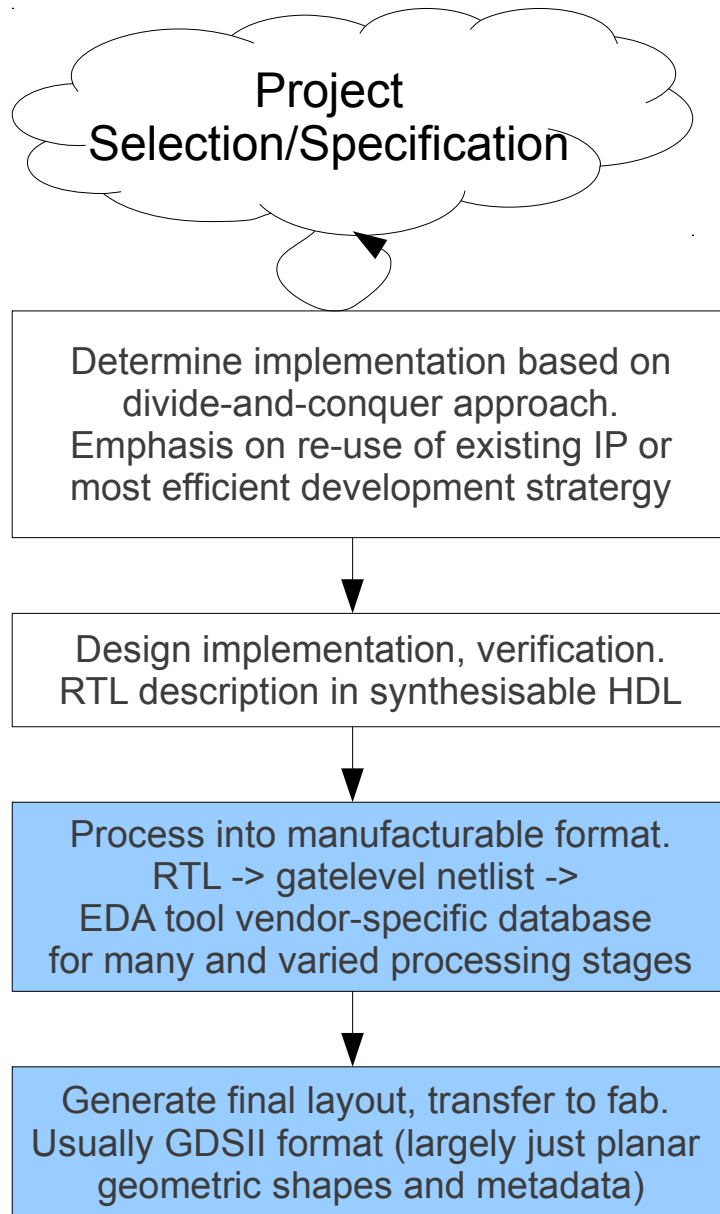
Image source: <http://spectrum.ieee.org/semiconductors/design/crossroads-for-mixedsignal-chips>

Chip Implementation Process

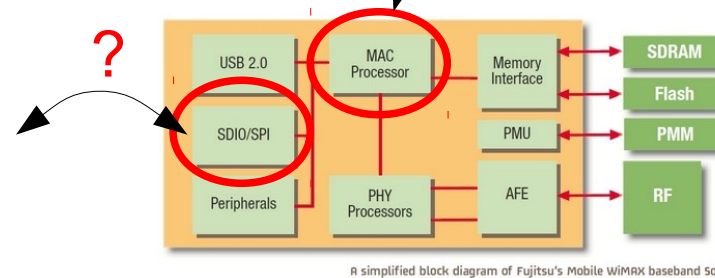
- ASIC: **A**pplication-**S**pecific **I**ntegrated **C**ircuit
- Modern process for purely digital ASICs (no major analog circuitry on chip) relatively straight forward
- Most chip design houses are *fabless* – they do not own and operate own manufacturing facility



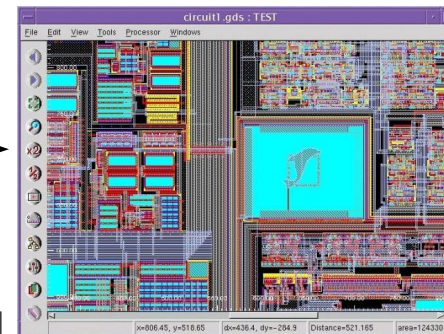
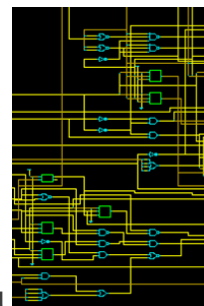
Fabless ASIC Development Process



Develop or buy IP?



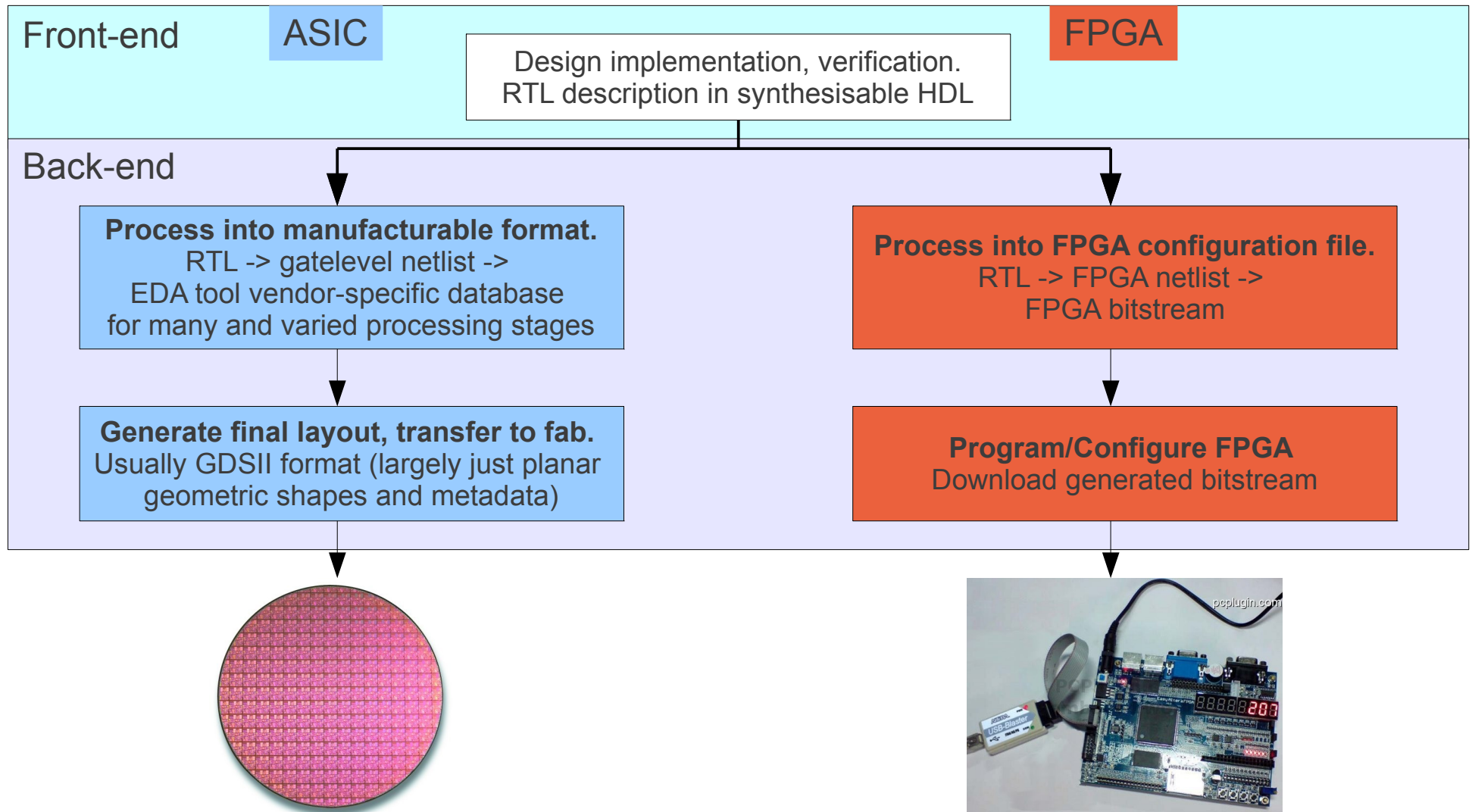
Source: http://www.ipod.org.uk/reality/reality_universe_computer.asp



Source: <http://freecode.com/projects/socgds>



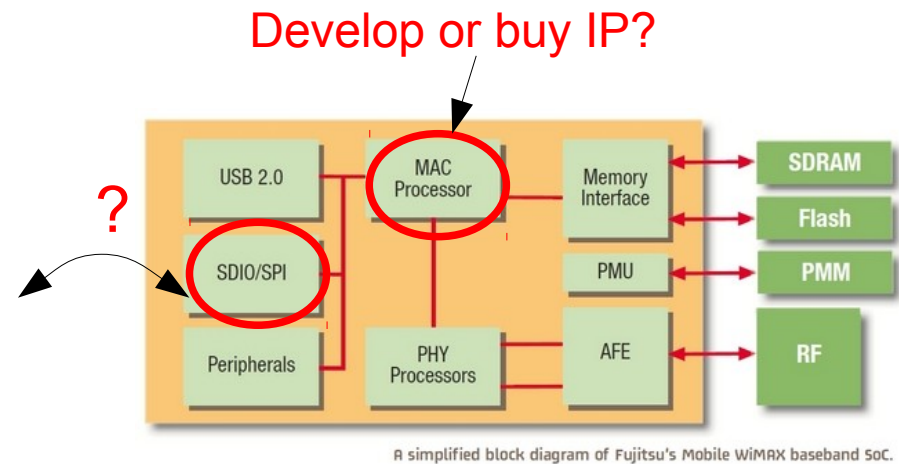
FPGA Development Process, In Comparison



IP Cores in SoCs

System Design with IP cores

- Decide on function and rough breakdown of how to achieve that
- Select units of semiconductor intellectual property (*IP cores*) for the job at hand



IP cores are units intended to provide a specific functionality and:

- usually provide a control mechanism via a standardised protocol (typically over a memory-mapped bus)
- are delivered in an electronic format (normally files with hardware description language, or HDL, code) which can be built and tested with the rest of the system
- are ultimately synthesised (combined) into the overall chip design so become part of the whole chip

- IP core design houses aim to design IP for maximum *reuse* (an industry mantra) this means:
 - Very configurable (heavy use of parameterised options)
 - Try to be application-generic
- Along with reuse and configurability are requirements for:
 - a demonstrably verified and 'proven' design (bugs causing a re-fab can cost \$10k->\$1M+)
 - Maximum area and power efficiency

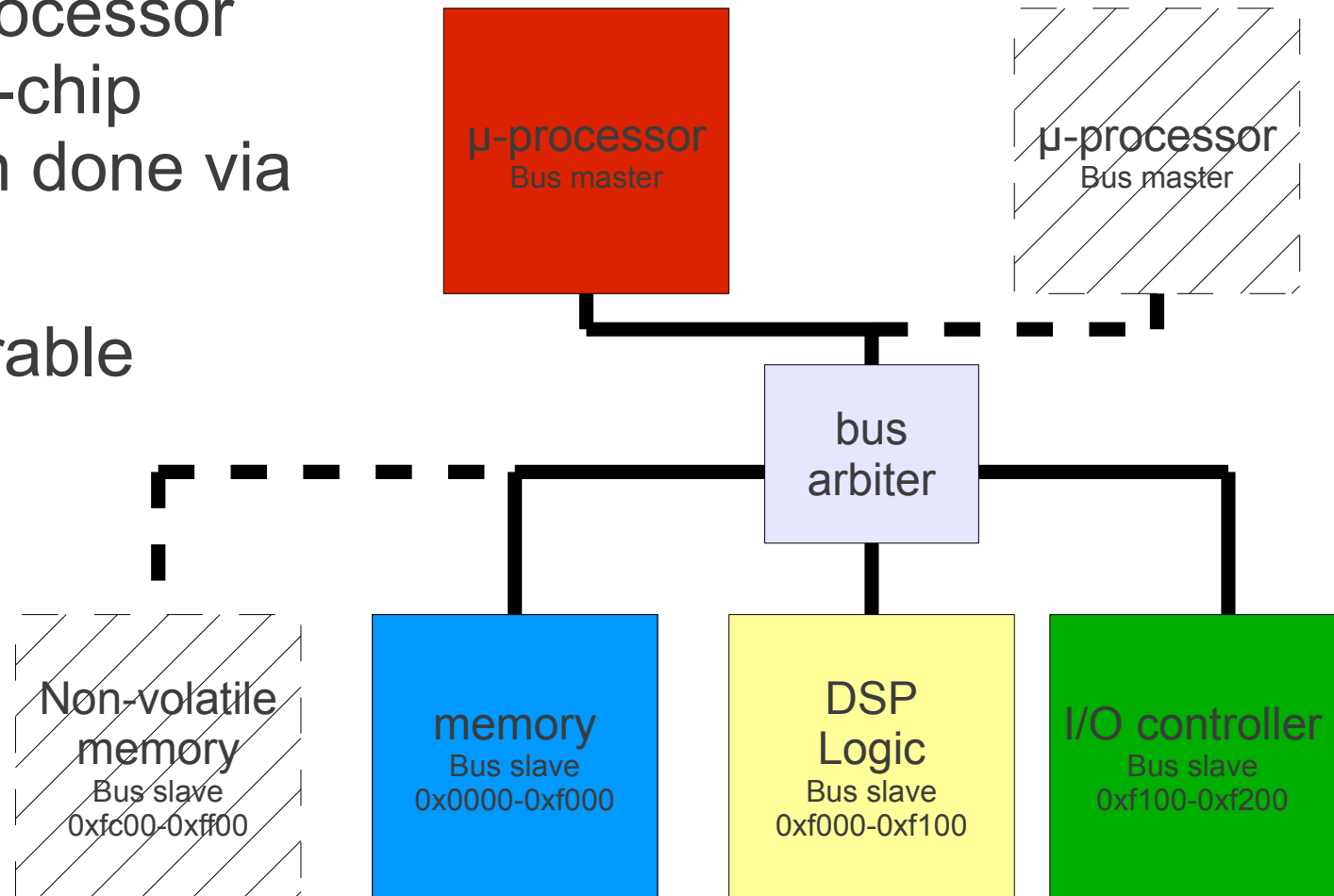
- Typically μ -processor based, with on-chip communication done via internal **bus**

- Highly configurable

- Many on-chip bus standards:

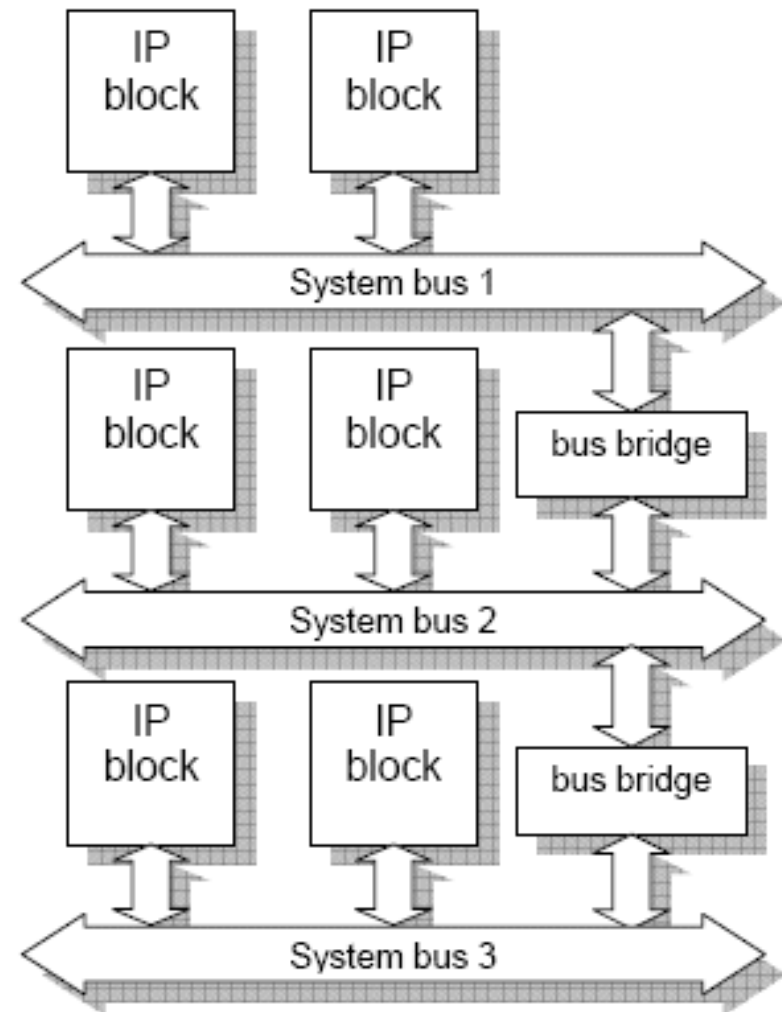
- Wishbone
- ARM AMBA
- OCP
- IBM CoreConnect

- A growing number of network-on-chip (NoC) interconnects, too



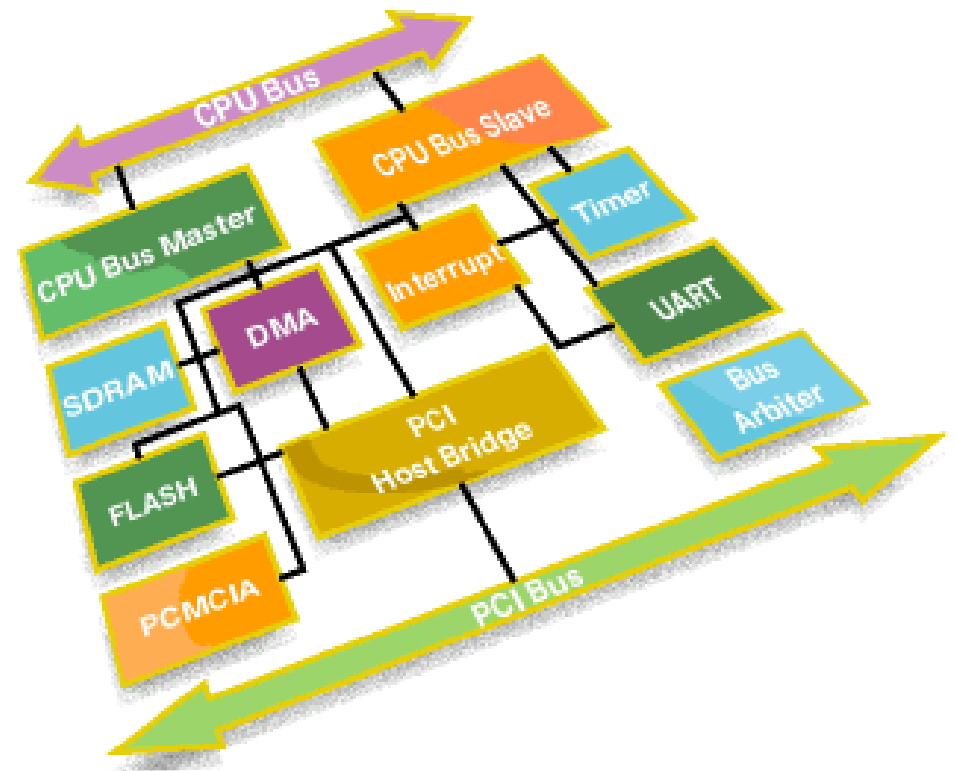
SoC Interconnect

- The SoC interconnect is what ties the IP blocks together
- Usually a memory-mapped bus providing access to control registers and memories



SoC Development Summary

- A large part of SoC design work is combination and verification (testing) of a set of IP
 - Confirm it's implementable
 - Confirm it works in your particular configuration
 - Confirm it has sufficient capacity for the intended application



Source: <http://www.eurekatech.com/>

Describing The Hardware

Describing the design

- IP blocks for implementation in modern digital VLSI process are usually designed using a *hardware description language*

- VHDL
- Verilog
- SystemVerilog increasingly

VHDL

```
library IEEE;
use IEEE.STD_Logic_1164, all;

entity LATCH_IF_ELSEIF is
  port (En1, En2, En3, A1, A2, A3: in std_logic;
        Y: out std_logic);
end entity LATCH_IF_ELSEIF;

architecture RTL of LATCH_IF_ELSEIF is
begin
  process (En1, En2, En3, A1, A2, A3)
  begin
    if (En1 = '1') then
      Y <= A1;
    elseif (En2 = '1') then
      Y <= A2;
    elseif (En3 = '1') then
      Y <= A3;
    end if;
  end process;
end architecture RTL;
```

Verilog

```
module LATCH_IF_ELSEIF (En1, En2, En3, A1, A2, A3, Y);
  input En1, En2, En3, A1, A2, A3;
  output Y;

  reg Y;

  always @(En1 or En2 or En3 or A1 or A2 or A3)
    if (En1 == 1)
      Y = A1;
    else if (En2 == 1)
      Y = A2;
    else if (En3 == 1)
      Y = A3;

end module
```

From Computer Desktop Encyclopedia
© 2004 The Computer Language Co. Inc.

Hardware Description Languages

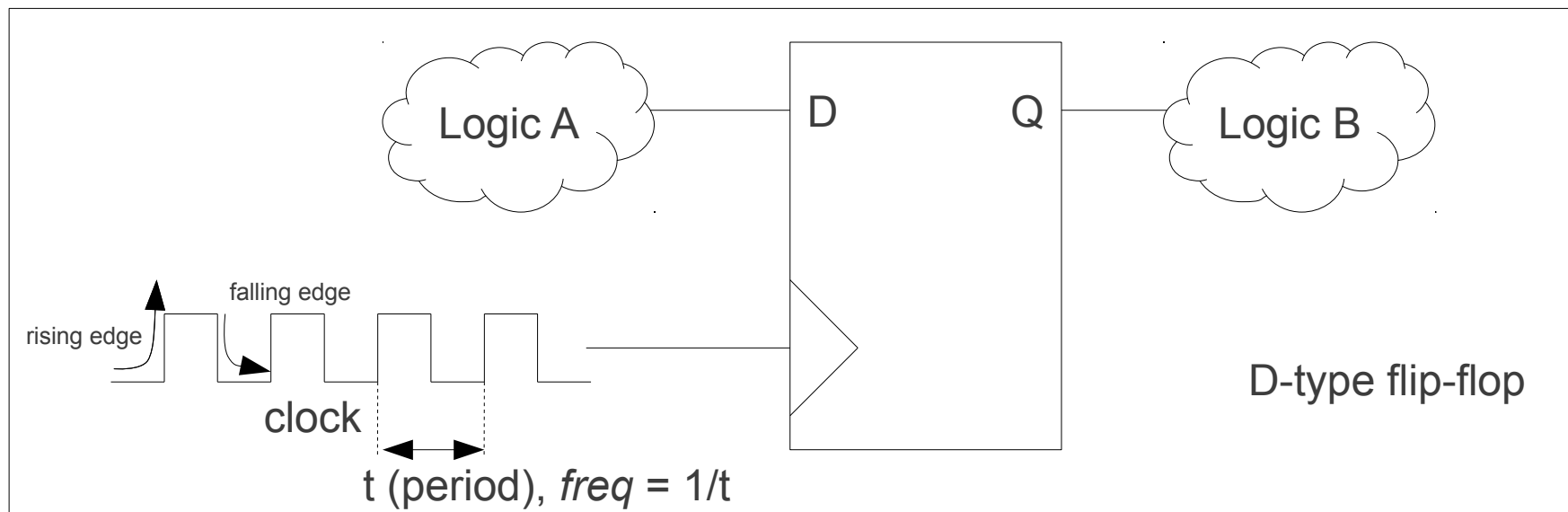
- Crucially allow the description of
 - **Synchronous** elements
 - **Combinatorial** logic
- **Synchronous** elements such as registers (flip-flops) store the state of the incoming signal based on the rising/falling of the clock
- **Combinatorial** elements comprise the logical functions between synchronous elements



A synchronous element in Verilog

- Always sampling on each **rising edge of the clock**, output to 'D' after a short time afterward, valid until next rising clock edge
- The simplest memory element, also considered a 1-cycle delay

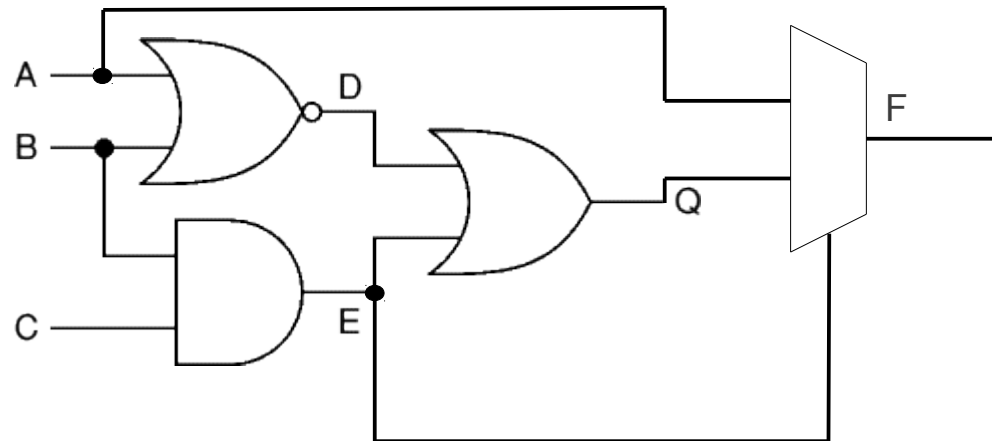
```
wire logic_a_output; /* assigned elsewhere */  
wire logic_b_input;  
reg q;  
  
always @(posedge clock)  
    q <= logic_a_output; /* the 'D' input */  
  
assign logic_b_input = q;
```



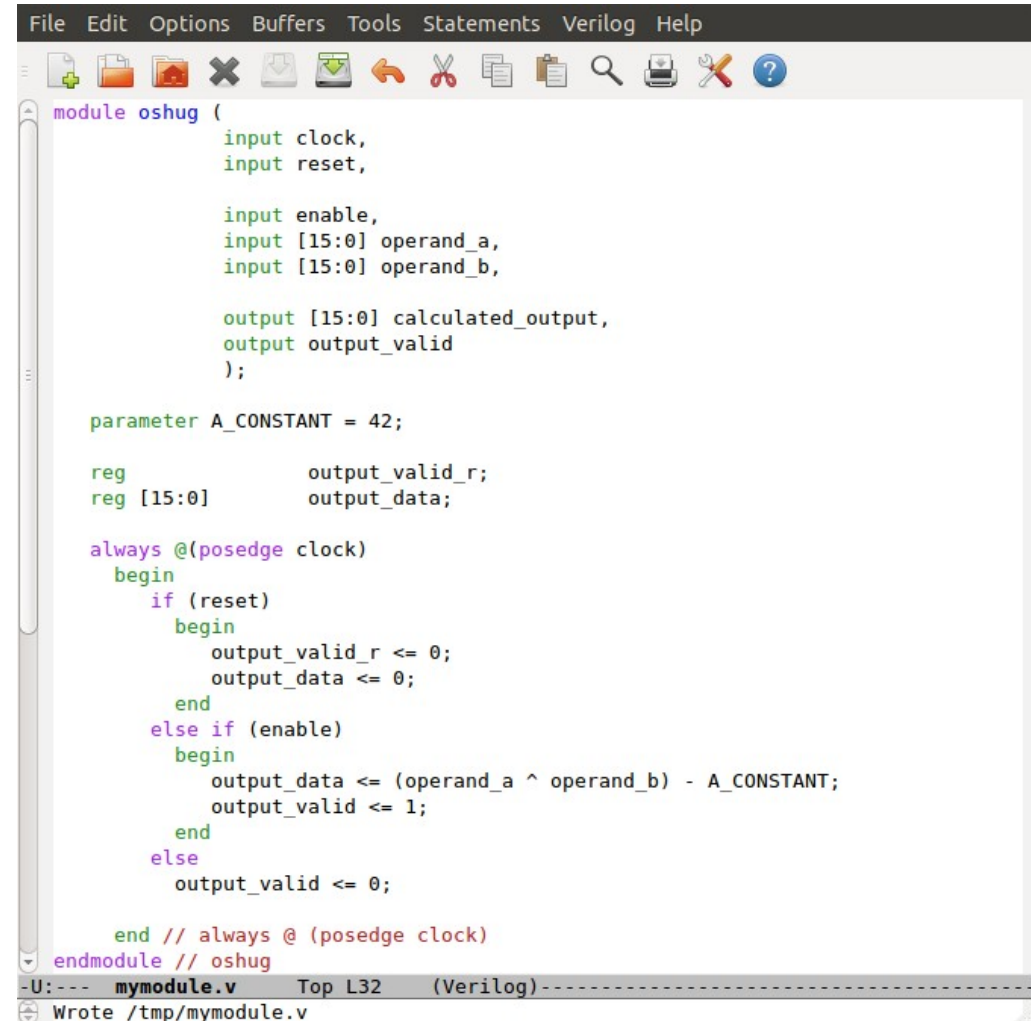
Combinatorial logic in Verilog

- Essentially any logic, arithmetic, conditional operator

```
assign d = !(a | b);  
assign e = b & c;  
assign q = d | e;  
Assign f = e ? a : q;
```



- Provides ways of describing the propagation of time and signal dependencies (sensitivity)
- Block-based structure



```
File Edit Options Buffers Tools Statements Verilog Help
module oshug (
    input clock,
    input reset,

    input enable,
    input [15:0] operand_a,
    input [15:0] operand_b,

    output [15:0] calculated_output,
    output output_valid
);

parameter A_CONSTANT = 42;

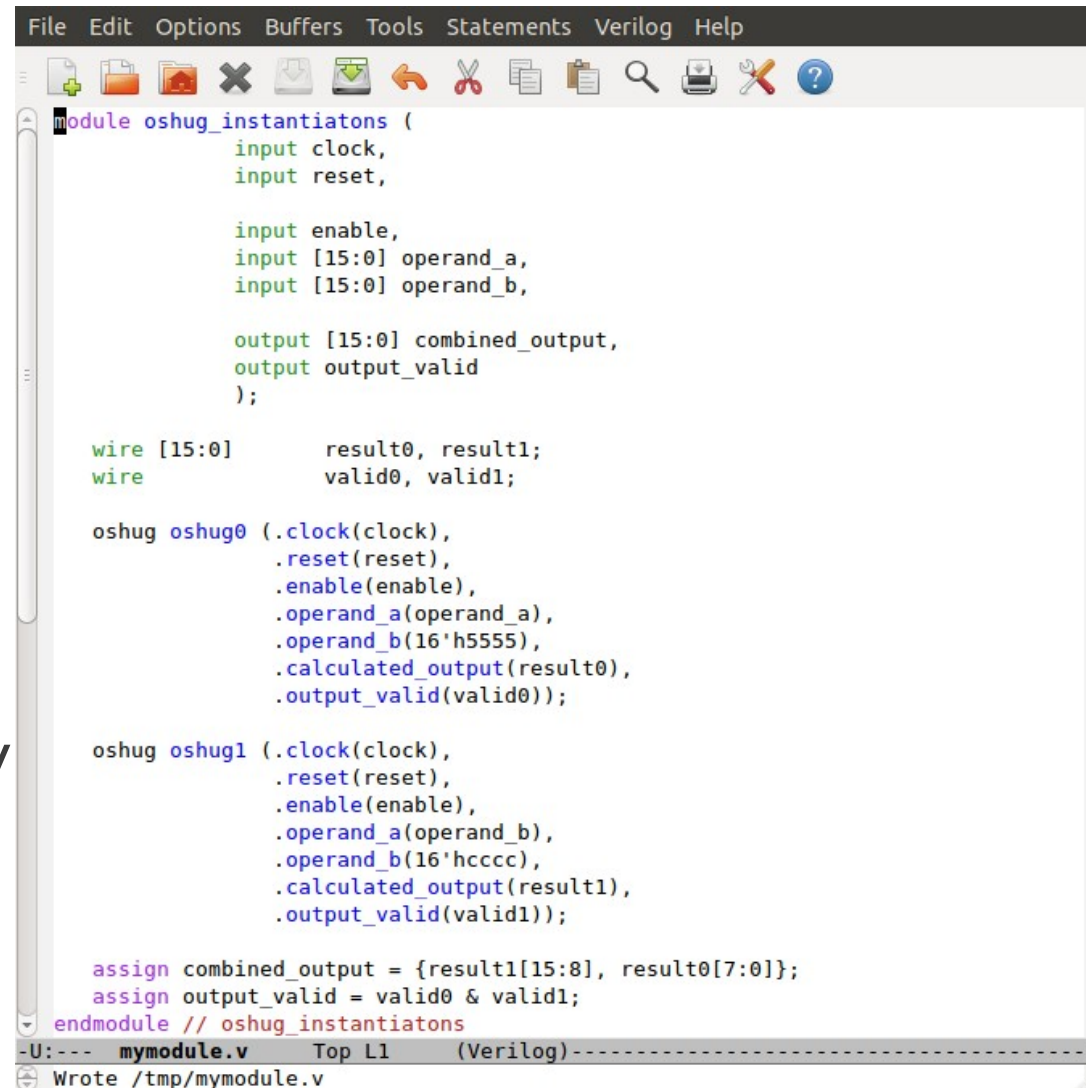
reg          output_valid_r;
reg [15:0]    output_data;

always @(posedge clock)
begin
    if (reset)
    begin
        output_valid_r <= 0;
        output_data <= 0;
    end
    else if (enable)
    begin
        output_data <= (operand_a ^ operand_b) - A_CONSTANT;
        output_valid <= 1;
    end
    else
        output_valid <= 0;

end // always @ (posedge clock)
endmodule // oshug
-U:--- mymodule.v Top L32 (Verilog)-----
Wrote /tmp/mymodule.v
```

Verilog Module Instantiations

- As design for re-use is emphasised, modular design is important.
- Abstraction and organisation is achieved by organising smaller, repeatable parts of a design into modules, much like functions in any other language



```
File Edit Options Buffers Tools Statements Verilog Help
module oshug_instantiations (
    input clock,
    input reset,

    input enable,
    input [15:0] operand_a,
    input [15:0] operand_b,

    output [15:0] combined_output,
    output output_valid
);

wire [15:0] result0, result1;
wire valid0, valid1;

oshug oshug0 (.clock(clock),
              .reset(reset),
              .enable(enable),
              .operand_a(operand_a),
              .operand_b(16'h5555),
              .calculated_output(result0),
              .output_valid(valid0));

oshug oshug1 (.clock(clock),
              .reset(reset),
              .enable(enable),
              .operand_a(operand_b),
              .operand_b(16'hcccc),
              .calculated_output(result1),
              .output_valid(valid1));

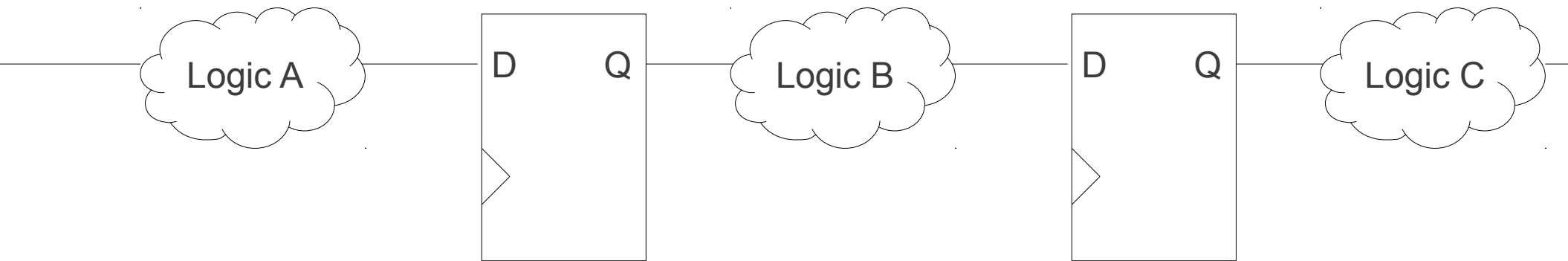
assign combined_output = {result1[15:8], result0[7:0]};
assign output_valid = valid0 & valid1;
endmodule // oshug_instantiations
-U:--- mymodule.v Top L1 (Verilog)-----
Wrote /tmp/mymodule.v
```

Levels of Abstraction

- Verilog could be used to describe a design at a low or high level of abstraction
- An example of quite low level Verilog is a *gatelevel netlist* which describes each and every atomic cell and their interconnections in a design
- Higher level design is achieved through the use of Verilog's arithmetic operators which can infer rather complex logic (multipliers, adders) or case statements on wide busses which can also infer large amounts of logic (multiplexors)

Register Transfer Level

- Most common level at which design is described is the *register transfer level* (RTL)
- Synthesisable Verilog is commonly referred to as RTL
- RTL? A description of the values signals should take on when the clock, or other signals, change their values
- Synchronous, or clock-based behaviour, results in flip-flops/registers being used to implement the design
- Combinatorial logic, or descriptions of logical functions, are implemented in fundamental logic components in hardware (not, and, or, xor etc.)



Implementation

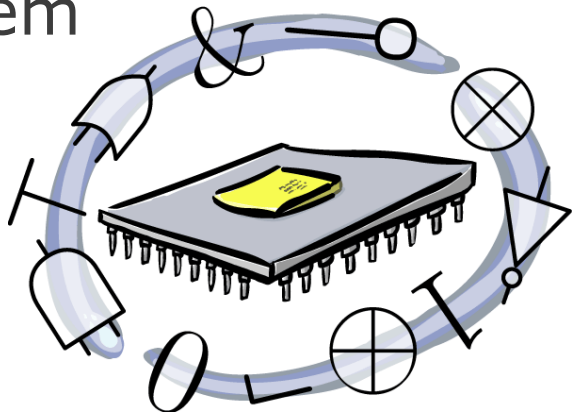
Getting the design onto the silicon

- Code in a synthesisable subset of the language supported by the synthesis tool in use
- You are ultimately intending to implement your logic with a particular *technology*
- *Technology* in this case is used to refer to the ASIC process or FPGA which will have the design 'put on it' – each provide a library of cells which can be used to implement the design



FPGA Technology

- **F**ield **P**rogrammable **G**ate **A**rray
- An array of multi-purpose logic which can be configured to create (within reason) arbitrary logical functions and interconnections between them
- American FPGA vendor Xilinx uses thousands of lookup tables (LUTs) which have signals routed through to emulate the logic functions the designer describes in the HDL



- FPGAs contain arrays of the bread-and-butter of digital logic (combinatorial and sequential logic elements in the LUTs) as well as:
 - Routing interconnect
 - RAMs
 - I/O hardware
 - Clock generation
- They are **reconfigurable** so can be applied for a wide variety of designs
- Have area, power, operating frequency **disadvantages** when compared to ASIC
- Often used for ASIC prototyping

- Are now big enough to implement systems-on-chip (consisting of processor, memory, I/O, accelerators) capable of running Linux distros
- How do we get from Verilog to an FPGA configuration file?



- Each ASIC process, FPGA generation and family provide implementation components in the way of gates and macro cells (RAMs, adders, multipliers)
- The synthesis tool must be aware of this and optimise and map the design described in the HDL to the targeted technology
- The synthesised design is described in the synthesisable subset of the HDL

SoC design with HDL source

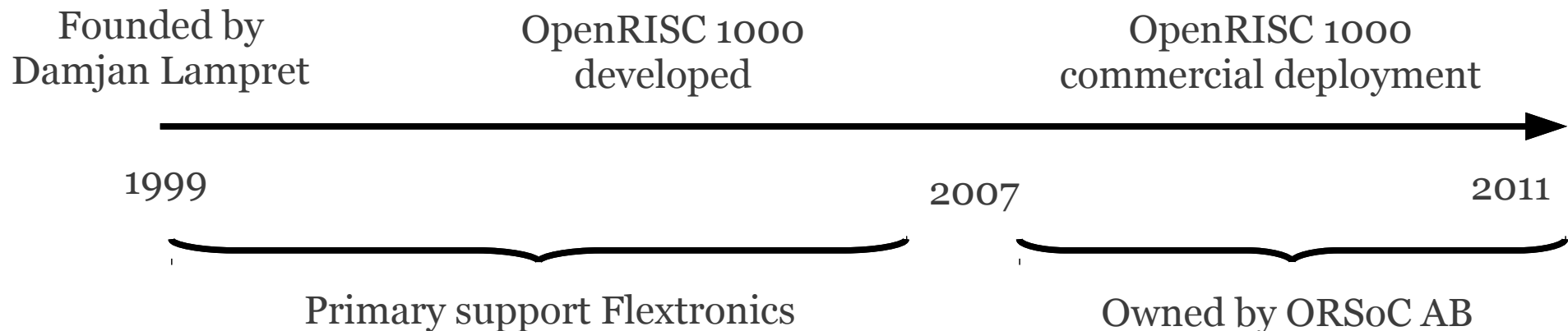
- The technology an IP block will be used on is determined by SoC designers
- IP designers don't know this and must design accordingly
- IP blocks have a lot of 'tunability' for synthesis (removing unwanted or unnecessary or unimplementable features.)
 - Verilog has a preprocessor, so features can be selected with a set of defines, equivalent to C #defines
 - Verilog parameters are similar and are a 'compile-time' (synthesis-time) option

Open Source Digital Design

OpenCores and OpenRISC

Overview of OpenCores

- ♦ 147,001 registered users reported as of 28 March 2012
- ♦ 919 projects as of 28 March 2012

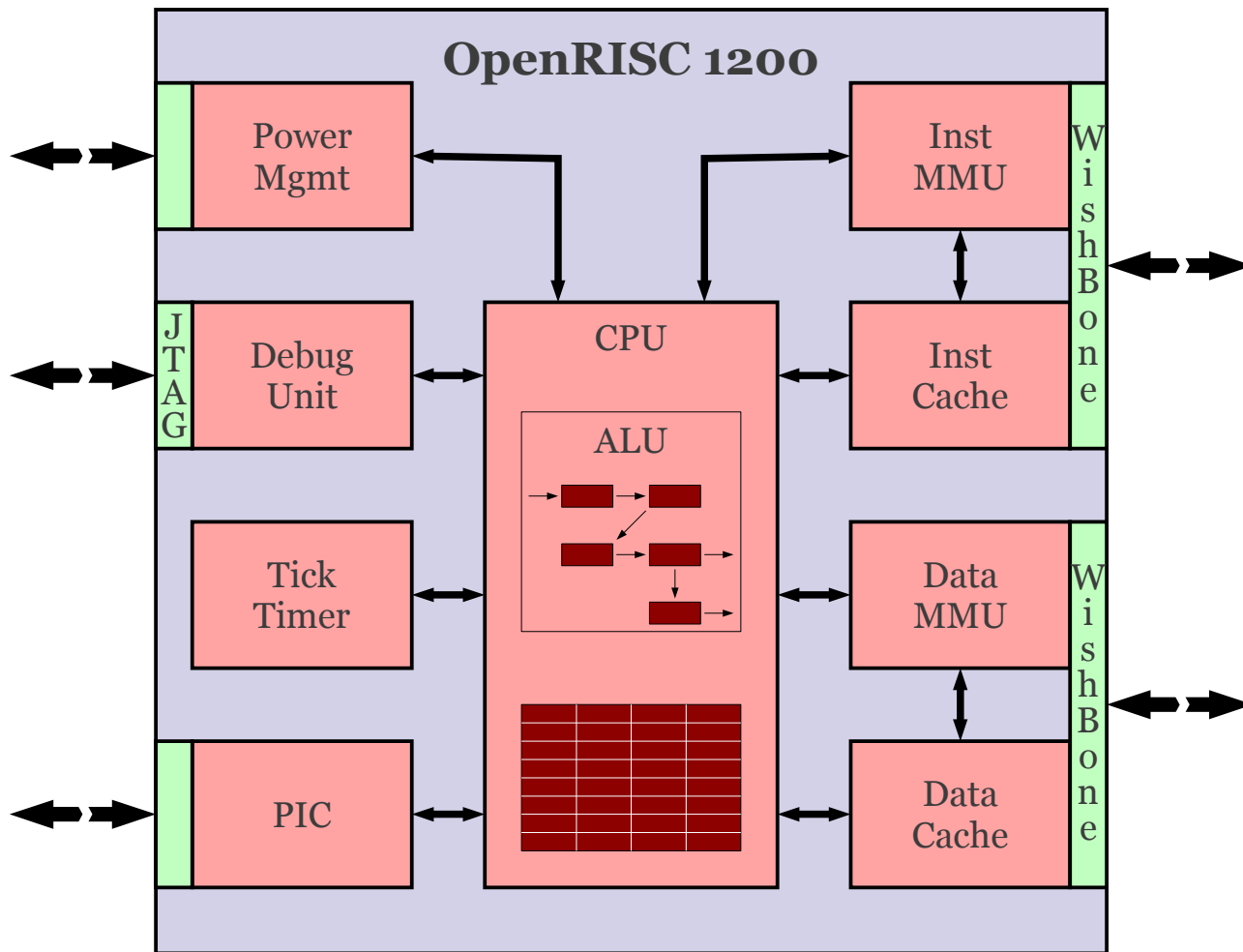


Web: www.opencores.org and
www.openrisc.net
IRC: freenode.net, channel [#opencores](https://freenode.net)

The OpenRISC 1000 Project

- Objective to develop a *family* of open source RISC designs
 - 32 and 64-bit architectures
 - floating point support
 - vector operation support
- Key features
 - fully free and open source
 - linear address space
 - register-to-register ALU operations
 - two addressing modes
 - delayed branches
 - Harvard or Stanford memory MMU/cache architecture
 - fast context switch
- Looks rather like MIPS or DLX

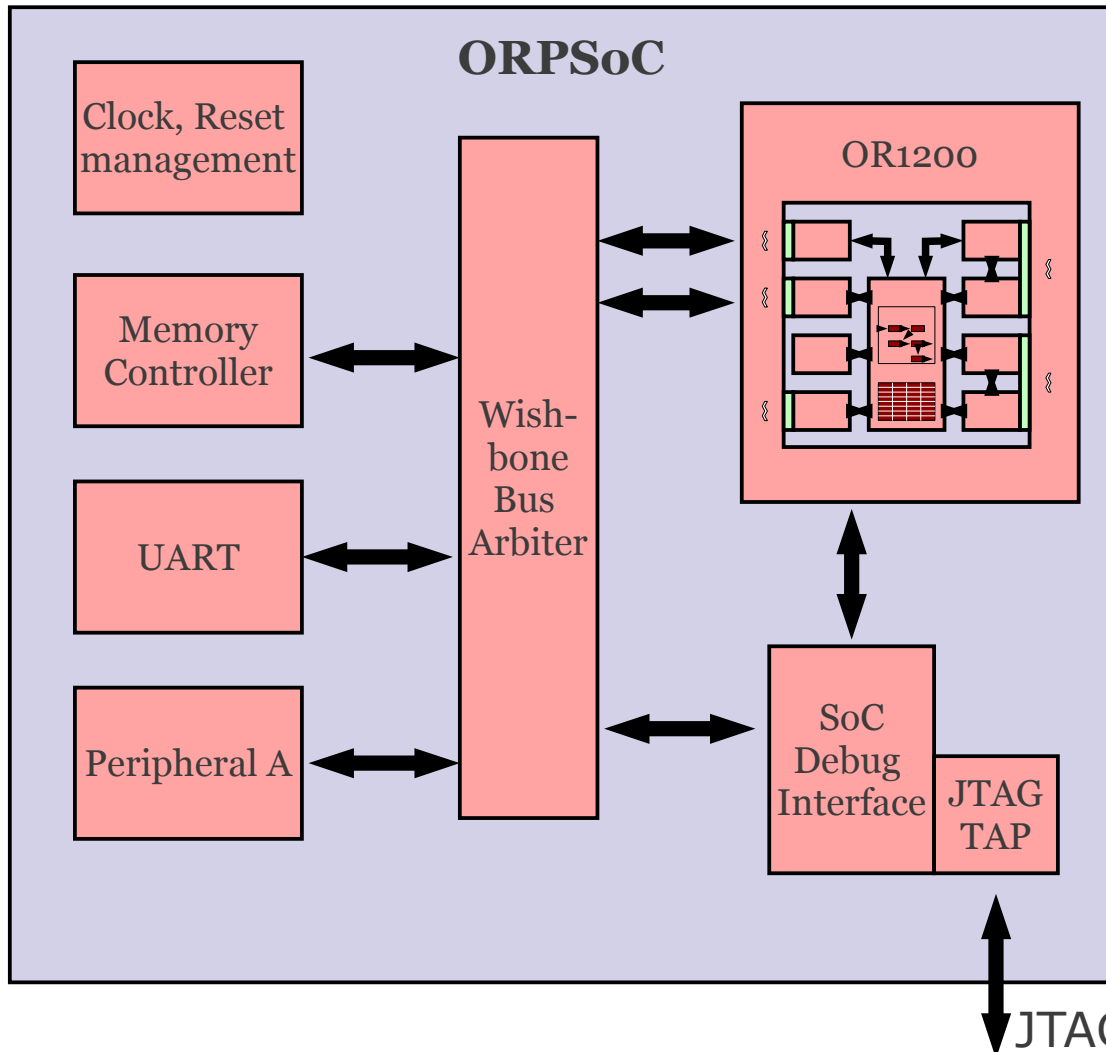
The OpenRISC 1200



- 32-bit Harvard RISC architecture
 - MIPS/DLX like instruction set
 - first in OpenRISC 1000 family
 - originally developed 1999-2001
- Open source under the
 - GNU Lesser General Public License
 - allows reuse as a component
- Configurable design
 - caches and MMUs optional
 - core instruction set
- Source code Verilog 2001
 - approx 32k lines of code
- **Full GNU tool chain and Linux port**
 - various RTOS ported as well



ORPSoC: OpenRISC Reference Platform System-on-Chip



- Combined reference implementation and board adaptations
- Reference implementation – minimal SoC for processor testing, development
 - compilable into cycle-accurate model
- Boards ports target multiple technologies
- Lowers barrier to entry for OpenRISC-based SoC design
 - Push-button compile flow
 - Largely utilises open-source EDA tools

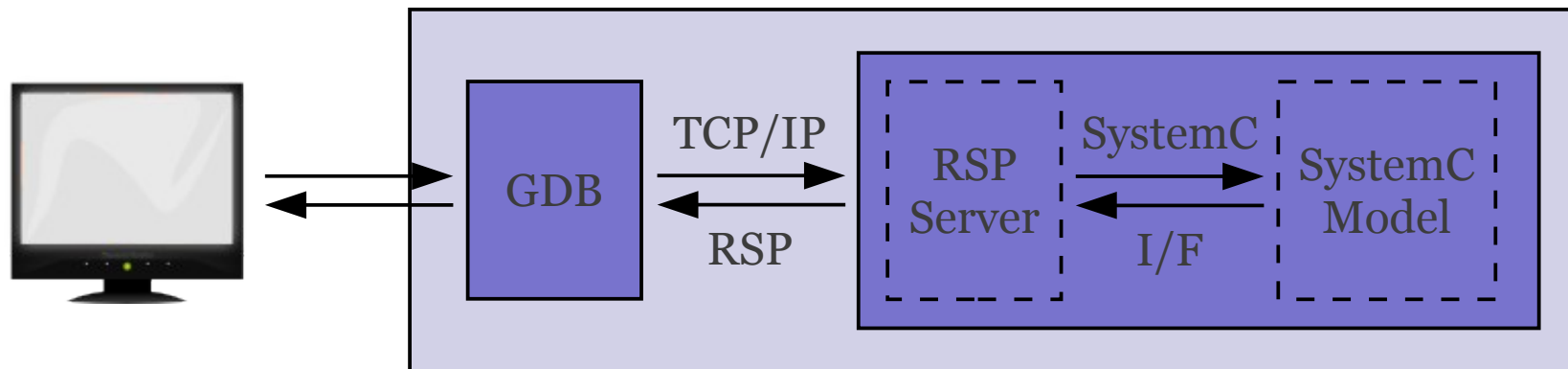
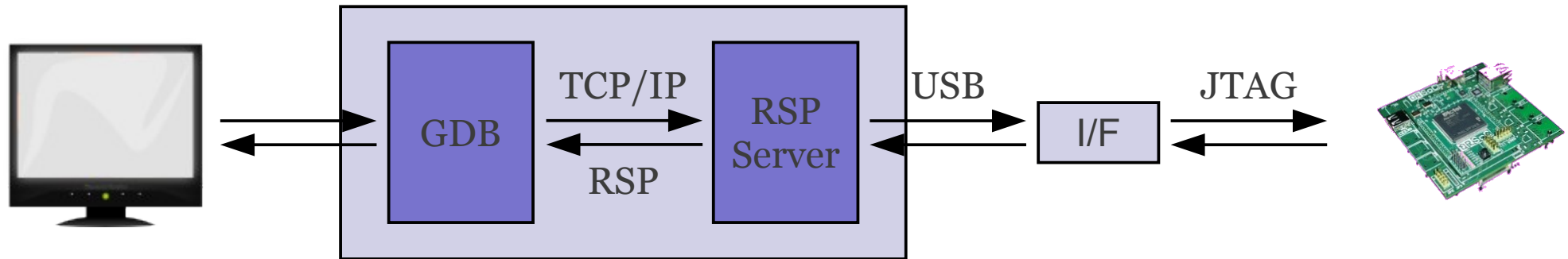
- Objective is to use an open source EDA tool chain
 - back end tools for FPGA all proprietary
 - free (as in beer) versions available
 - front end tools now have open source alternatives
- OpenRISC 1000 simulation models
 - Or1ksim: golden reference ISS
 - C/SystemC interpreting ISS, 2-5 MIPS
 - Verilator cycle accurate model from the Verilog RTL
 - 130kHz in C++ or SystemC
 - Icarus Verilog event driven simulation
 - 1.4kHz, 50x slower than commercial alternatives
- All OpenRISC 1000 simulation models suitable for SW use
 - all support GDB debug interface

The OpenRISC 1000 Tool Chain

- A standard GNU tool chain
 - binutils 2.20.1
 - gcc 4.5.1
 - gdb 7.3 (for BCS use only!)
 - C and C++ language support
- Library support
 - static libraries only
 - newlib 1.18.0 for bare metal (**or32-elf-***)
 - uClibc 0.9.32 for Linux applications (**or32-linux-***)
- Testing
 - regression tested using Or1ksim (both tool chains)
 - **or32-linux-*** regression tested on hardware
 - **or32-elf-*** regression tested on a Verilator model

- Boards with BSP implementations
 - Or1ksim
 - Xilinx ML501, Terasic Altera DE-2, DE0-nano, ...
- RTOS support
 - FreeRTOS, RTEMS and eCos all ported
- Linux support
 - adopted into Linux 3.1 kernel mainline
 - some limitations (kernel debug, ptrace)
 - BusyBox as application environment
- Debug interfaces
 - JTAG for bare metal
 - *gdbserver* over Ethernet for Linux applications

Software Development Remote Connection to GDB



```
(gdb) target remote :51000
```


Building the Tool Chain

- Download the source:

```
svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/or1ksim
svn co http://opencores.org/ocsvn/openrisc/openrisc/trunk/gnu-src
cd gnu-src; git clone git://git.openrisc.net/jonas/uClibc; cd ..
cd gnu-src; git clone git://git.openrisc.net/jonas/linux; cd ..
```

- Build and install Or1ksim

```
cd or1ksim; mkdir bd; cd bd
../configure --target=or32-elf32 --prefix=/opt/or1ksim-new
make; make install; make pdf; cd ../..
```

- Build and install the tool chains into /opt/or32-new

```
cd gnu-src
./bld-all.sh --force --prefix /opt/or32-new \
              --or1ksim-dir /opt/or1ksim-new \
              --uclibc-dir uClibc --linux-dir linux
export PATH=$PATH:/opt/or1ksim-new:/opt/or32-new/bin
```

- You can then use the tools to build BusyBox and Linux
 - see http://opencores.org/or1k/OR1K:Community_Portal

- Stefan Wallentowitz at TUM
 - multicore version of OpenRISC 1200
 - student working on LLVM
- Pete Gavin
 - bringing the GNU tool chain up to date
- Ruben Diez
 - automated nightly builds
 - common test platform for models and HW

ORPSoC

OpenRISC Reference Platform System-on-Chip

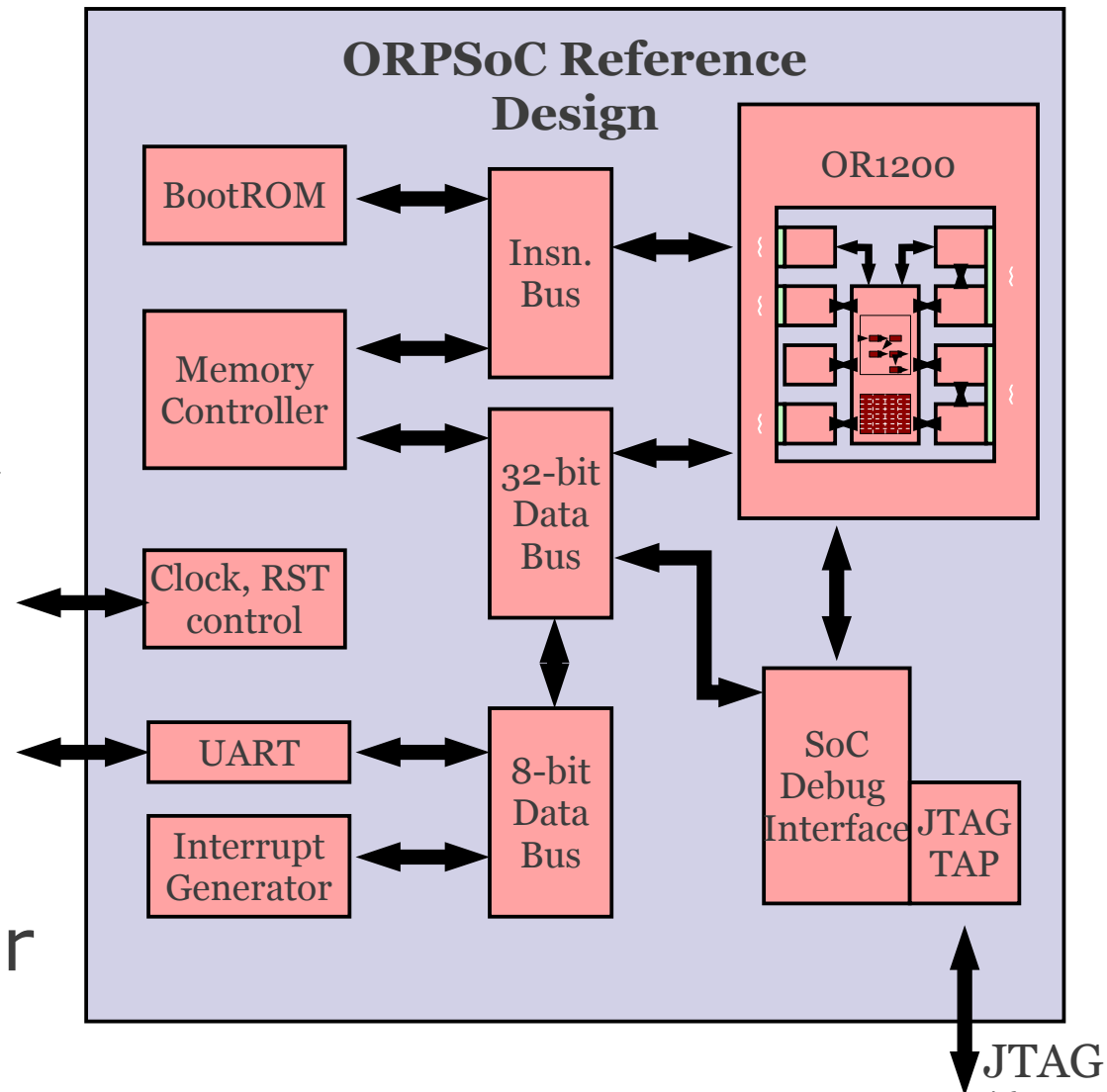
Two Sides Of ORPSoC

- “Reference build”
 - Processor testing platform
 - Not technology targeted
 - Can build fast cycle-accurate model
- “Board builds”
 - Targeted at particular FPGA boards
 - Live in own sub-project
 - Intended to provide “push-button” synthesis flows

Intended Audience And Uses Of ORPSoC

- Provides framework for users to experience digital design and, hopefully, get commodity FPGA development boards running an open source system
 - FPGA majors provide non-free (libre and beer) SoC implementations :(
- Potential uses are numerous but use cases for bespoke processing or I/O are common
 - Develop an IP for SHA256/DSP/motor control and instantiate multiple on FPGA along with OR1200 running Linux, connected via ethernet

- Simplest useful system for processor verification
 - “On-chip” memory
 - Debug interface
 - Can master bus
 - UART
 - Interrupt generator



Using ORPSoC

Running the reference design

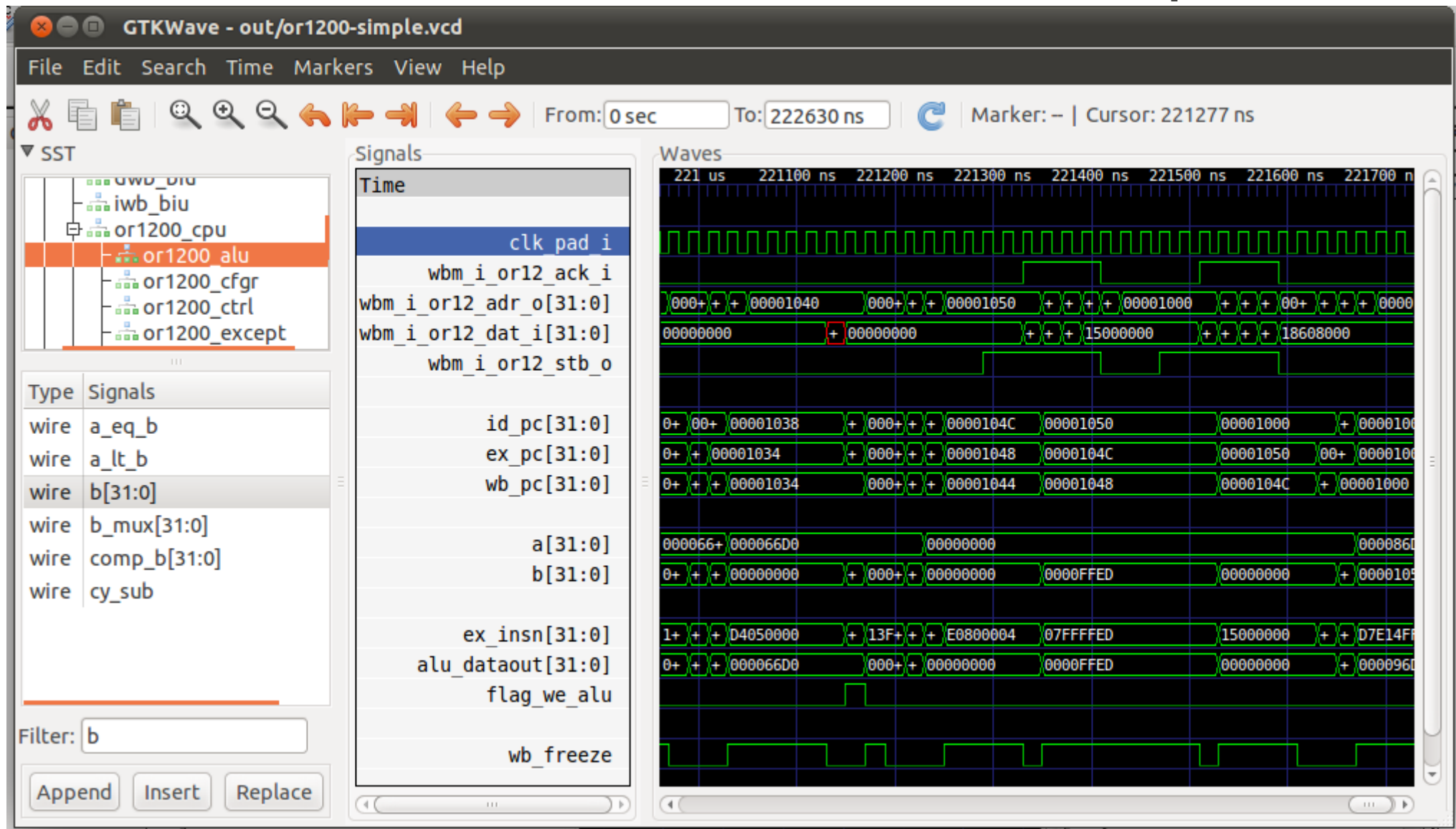
- Can execute a basic test, running the CPU through bootup, into a main() loop and immediately exiting

```
orpsovcv2$ cd sim/run  
run$ make rtl-test TEST=or1200-simple VCD=1
```

- The VCD=1 will create a dump of the internal signals which can be viewed in a waveform viewer such as GTKWave



GTKWave viewing or1200-simple.vcd



- Running a test will cause software to be built. This includes:
 - A simple C library containing basically rand() and printf()
 - Low-level CPU functions for features like interrupts and timers
 - The boot code (OR1K assembly, crt0.S)
 - Some application code
- It is all compiled and converted into appropriate format for loading into sim.

ORPSoC simulation directories

- All sims launched from sim/run but output generated in sim/out
- Some intermediate files generated in sim/run
- VCD in sim/out
- Memory image of program we executed was in sim/run/sram.vmem (symlink to it)

sim\$ tree -L 2

```
.
├── bin
│   ├── definesgen.inc
│   ├── Makefile
│   └── refdesign-or1ksim.cfg
├── out
│   ├── or1200-simple-executed.log
│   ├── or1200-simple-general.log
│   ├── or1200-simple-lookup.log
│   ├── or1200-simple-sprs.log
│   ├── or1200-simple.vcd
│   └── vvp.log
├── run
│   ├── icarus.scr
│   ├── Makefile
│   └── sram.vmem →
│       ../../sw/tests/or1200/sim/or1200-
│       simple.vmem
│   ├── test-defines.v
│   └── vlogsim.elf
```

- A sub-project of ORPSoC intended to be built and run on a specific FPGA system
- Contained under boards/ directory, then sorted by FPGA vendor

```
boards$ tree -L 2
```

```
.
├── actel
│   ├── backend
│   └── ordb1a3pe1500
├── README
├── xilinx
│   ├── atlys
│   ├── backend
│   ├── ml501
│   └── s3adsp1800
```

- Inherent modularity of SoC designs makes it relatively straight forward to add or remove features
- Removing features is usually as simple as commenting out a ``define`

```
...  
`define JTAG_DEBUG  
// `define RAM_WB  
// `define XILINX_SSRAM  
`define CFI_FLASH  
`define XILINX_DDR2  
`define UART0  
`define GPIO0  
// `define SPI0  
`define I2C0  
`define I2C1  
`define ETH0  
`define ETH0_PHY_RST  
...
```

A defines file in
boards/xilinx/ml501/rtl/verilog/include/orpsoc-defines.v

Configuring IP Cores

- IP blocks usually have own configuration information
- A Verilog `defines header is usually used to store config
 - Parameters on the instantiations are preferred as it allows multiple instances with differing configurations

```
...
// Do not implement Data cache
//`define OR1200_NO_DC

// Do not implement Insn cache
//`define OR1200_NO_IC

// Do not implement Data MMU
//`define OR1200_NO_DMMU

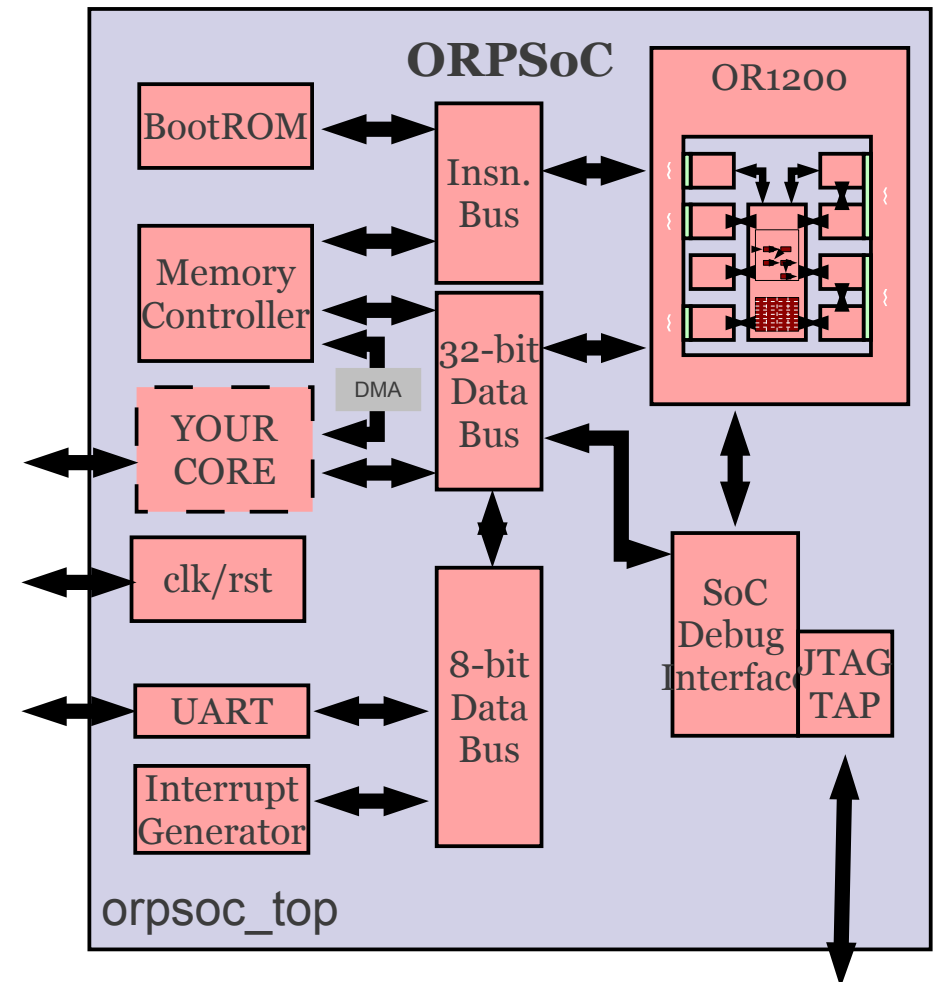
// Do not implement Insn MMU
//`define OR1200_NO_IMMU

// Size/type of insn/data cache if implemented
// (consider available FPGA memory resources)
//`define OR1200_IC_1W_16KB
`define OR1200_IC_1W_32KB
//`define OR1200_DC_1W_16KB
`define OR1200_DC_1W_32KB

// Implement optional l.div/l.divu instructions
// By default divide instructions are not implemented
// to save area.
`define OR1200_DIV_IMPLEMENTED
...
```

Adding new cores is a little more involved

- Instantiate the core in top-level
- Attach to bus
- Provide some software (driver/test)
- Attach I/O (if any)
 - Add constraints to back-end scripts



Adapting ORPSoC for new boards

- Be aware of FPGA technology (family, variant) and whether existing clocking and memory components can be used
- Ensure new pin mapping applied (which signals from ORPSoC go to which pins on the device/board)
- Check design fits on device (quick synthesis check)

Debug infrastructure on boards

- Two debug options
 - “Mohor” SoC debug IF
 - Advanced debug IF
- Both use JTAG physical layer
- Mohor adds own JTAG TAP and needs 4 extra pins
- `adv_debug_if` can use FPGA's TAP and save pins

- Be sure to determine debug solution!

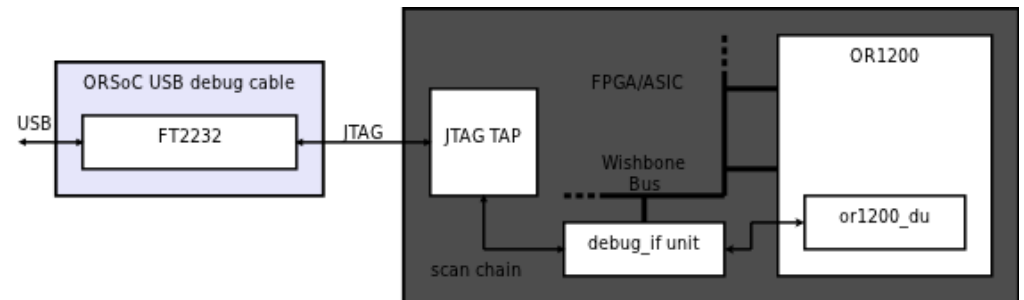
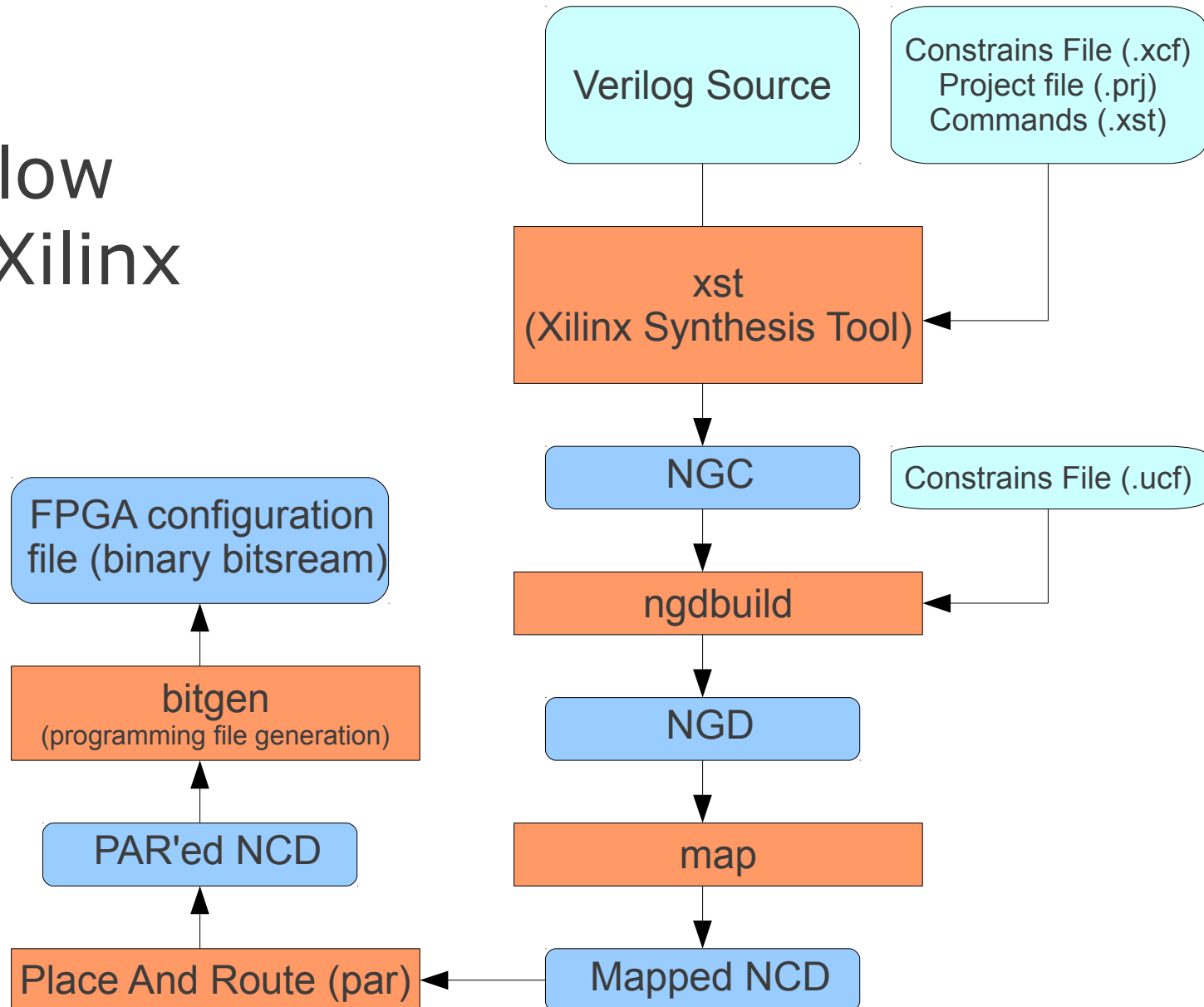


Diagram of Mohor Debug Interface Connecting to ORPSoC

ORPSoC synthesis flow

- Example flow based on Xilinx tools



Compiling Software For The Bare Metal

- GNU tool chain for both bare metal and Linux userspace programs
- Bare metal tool chain relies on newlib for its C library
- Newlib's libgloss handles low level interaction (supposed to implement syscall support.)
- OR1K libgloss is designed for bare metal usage

Adding new port to or32 libgloss

- A single object file must be compiled which contains some symbols defining, eg.
- Clock frequency of design
- UART address on bus

```
cat gnu-src/newlib-1.18.0/libgloss/or32/ml501.S
/*
 * Define symbols to be used during startup -
 * file is linked at compile time
 */
.global _board_mem_base
.global _board_mem_size
.global _board_clk_freq

_board_mem_base:      .long 0x0
_board_mem_size:      .long 0x800000

_board_clk_freq:      .long 66666666

/* Peripheral information - Set base to 0 if not present */
.global _board_uart_base
.global _board_uart_baud
.global _board_uart_IRQ

_board_uart_base:      .long 0x90000000
_board_uart_baud:      .long 115200
_board_uart_IRQ:      .long 2
```

Compiling with new board library

- Once the file is compiled with the correct values, it should be archived and placed alongside the rest of the newlib board support files:
 - The `-mboard=boardname` switch can be now passed to the compiler and software should initialise correctly for the board
- `$TOOLCHDIR/or32-elf/lib/boards/<boardname>/libboard.a`

Run “helloworld” in the simulator

- Create a basic helloworld C file:

```
#include <stdio.h>
int main(void)
{
    printf("Hello world!\n");
    return 0;
}
```

- Compile it:

```
or32-elf-gcc hello.c -o hello
```

- Run it in or1ksim

```
$ or32-elf-sim -m8M hello
```

```
Seeding random generator with value ...
Or1ksim 2012-03-23
Building automata... done
```

```
...
Section: .jcr, vaddr: 0x000089bc,...
Section: .data, vaddr: 0x000089c0, ...
Hello world!
exit(0)
```

```
@reset : cycles 0, insn #0
@exit  : cycles 3692, insn #2842
diff   : cycles 3692, insn #2842
```

- Note: GCC defaults to use the “or1ksim” board