Contents

Acknowledgement

We would like to express our gratitude and thank our guide Prof. B Rajeshwari for her guidance and support throughout the project. We have learnt a great deal and picked up many skills working on this project and are confident that this knowledge will help us in our future undertakings.

Secondly, we would like to thank our parents and friends for their constant support and encouragement which helped in successful completion of this project within the given time frame. We would specifically like to thank Mr. Balaji Ravikumar for his guidance in critical phases of this project.

Lastly, we would like to thank PES University, for imparting knowledge and providing ample number of opportunities to learn and grow without which we would not have the capability to undertake a project of this proportion.

Abstract

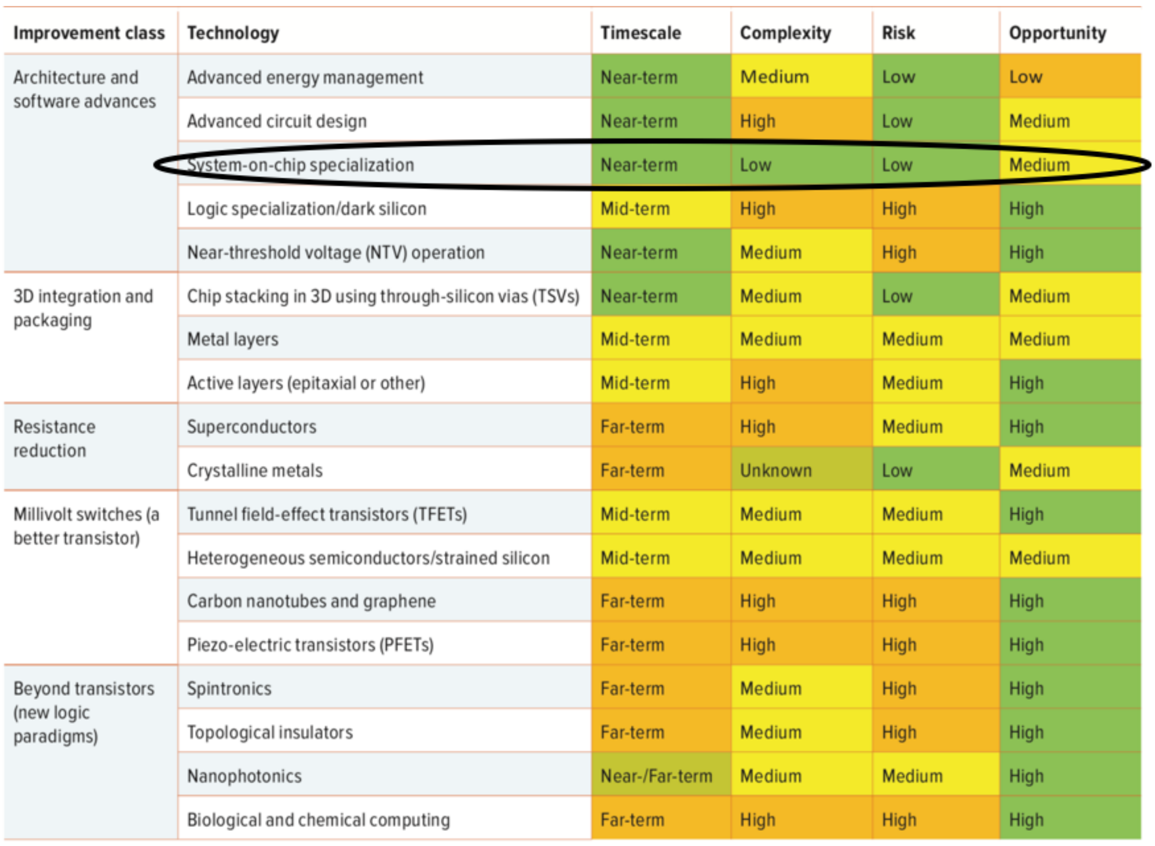
A System-on-Chip (SoC) involves integration of processors, memory and a variety of IP cores in a single design. The availability of a processor subsystem which can be implemented on an FPGA opens the door to a myriad of applications in Digital Signal Processing, Communication, and Embedded Systems. In this project, we demonstrate the design and implementation of an open source OpenRISC-based SoC for image processing. The designed system is complete with software drivers and is extremely scalable, allowing incorporation of new IP cores, accelerators and coprocessors into the design based on the application.

1. Literature Survey

- **Paper 1.1**: M.Bakiri, S. Titri, N. Izeboudjen, F. Abid, F. Louizand D. Lazib, "Embedded system with Linux Kernel based on OpenRISC 1200-V3"
  - **Motivation**: Design of an OpenRISC-based system using open source IP cores.
  - **Learning**: This paper serves as a guide/reference for complete end-to-end design of a system.
  -
- **Paper 1.2**: L.Akcay, M. Tukeland B. Ors, "Design and implementation of an OpenRISC system-on-chip with an encryption peripheral"
  - **Motivation**: Design of an OpenRISC-based SoC for encryption.
  - **Learning:** Overview of the design process and identification of possible applications.
  -
- **Paper 1.3**: A. López-Parrado and J. C. Valderrama-Cuervo, "OpenRISC-based System-on-Chip for digital signal processing"
  - **Motivation**: Design of an OpenRISC-based SoC for DSP applications.
  - **Learning:** Hardware design for better performance of DSP functions such as IIR filter, FIR filter, FFT, etc,.
  -
- **Paper 1.4**: C. He, A. Papakonstantinouand D. Chen, "A novel SoC architecture on FPGA for ultra fast face detection"
  - **Motivation**: Design of an SoC for image processing applications.
  - **Learning:** Hardware design for accelerated face detection.

2. Background

Photolithography systems and manufacturing technologies are on pace to reach atomic scale by the mid-2020s, necessitating alternatives to continue realizing faster, more predictable, and cheaper computing performance. Trends in VLSI have shown that the emerging technologies listed in table 1.1 are the way forward to extend digital electronics and keep pace with Moore’s law in the next decade.



Improvement class	Technology	Timescale	Complexity	Risk	Opportunity
Architecture and software advances	Advanced energy management	Near-term	Medium	Low	Low
	Advanced circuit design	Near-term	High	Low	Medium
	System-on-chip specialization	Near-term	Low	Low	Medium
	Logic specialization/dark silicon	Mid-term	High	High	High
	Near-threshold voltage (NTV) operation	Near-term	Medium	High	High
3D integration and packaging	Chip stacking in 3D using through-silicon vias (TSVs)	Near-term	Medium	Low	Medium
	Metal layers	Mid-term	Medium	Medium	Medium
	Active layers (epitaxial or other)	Mid-term	High	Medium	High
Resistance reduction	Superconductors	Far-term	High	Medium	High
	Crystalline metals	Far-term	Unknown	Low	Medium
Millivolt switches (a better transistor)	Tunnel field-effect transistors (TFETs)	Mid-term	Medium	Medium	High
	Heterogeneous semiconductors/strained silicon	Mid-term	Medium	Medium	Medium
	Carbon nanotubes and graphene	Far-term	High	High	High
	Piezo-electric transistors (PFETs)	Far-term	High	High	High
Beyond transistors (new logic paradigms)	Spintronics	Far-term	Medium	High	High
	Topological insulators	Far-term	Medium	High	High
	Nanophotonics	Near-/Far-term	Medium	Medium	High
	Biological and chemical computing	Far-term	High	High	High

Table 2.1: Summary of technology options for extending digital electronics.

The core precept of System-on-Chip technology is that chip cost is dominated by component design and verification costs. Also, SoCs are highly power efficient compared to traditional general-purpose computers. Usually, 90% of power consumption is in data and address bus cabling. Since all the components are integrated and internally connected on the same chip, the power consumption is drastically reduced. Further, System-on-Chips provide greater design security at hardware and firmware levels.

Therefore, tailoring chips to include only the circuit components of value to the application is more energy and economically efficient than designing a chip that serves a broad application range. This tailoring strategy is common practice for cell-phones, digital cameras, and other embedded systems. Thus, in this era, where Internet of Things and mobile computers are ubiquitous, designing tailor-made efficient smaller standalone systems with processing, memory and communication capabilities is extremely crucial.

3. Introduction

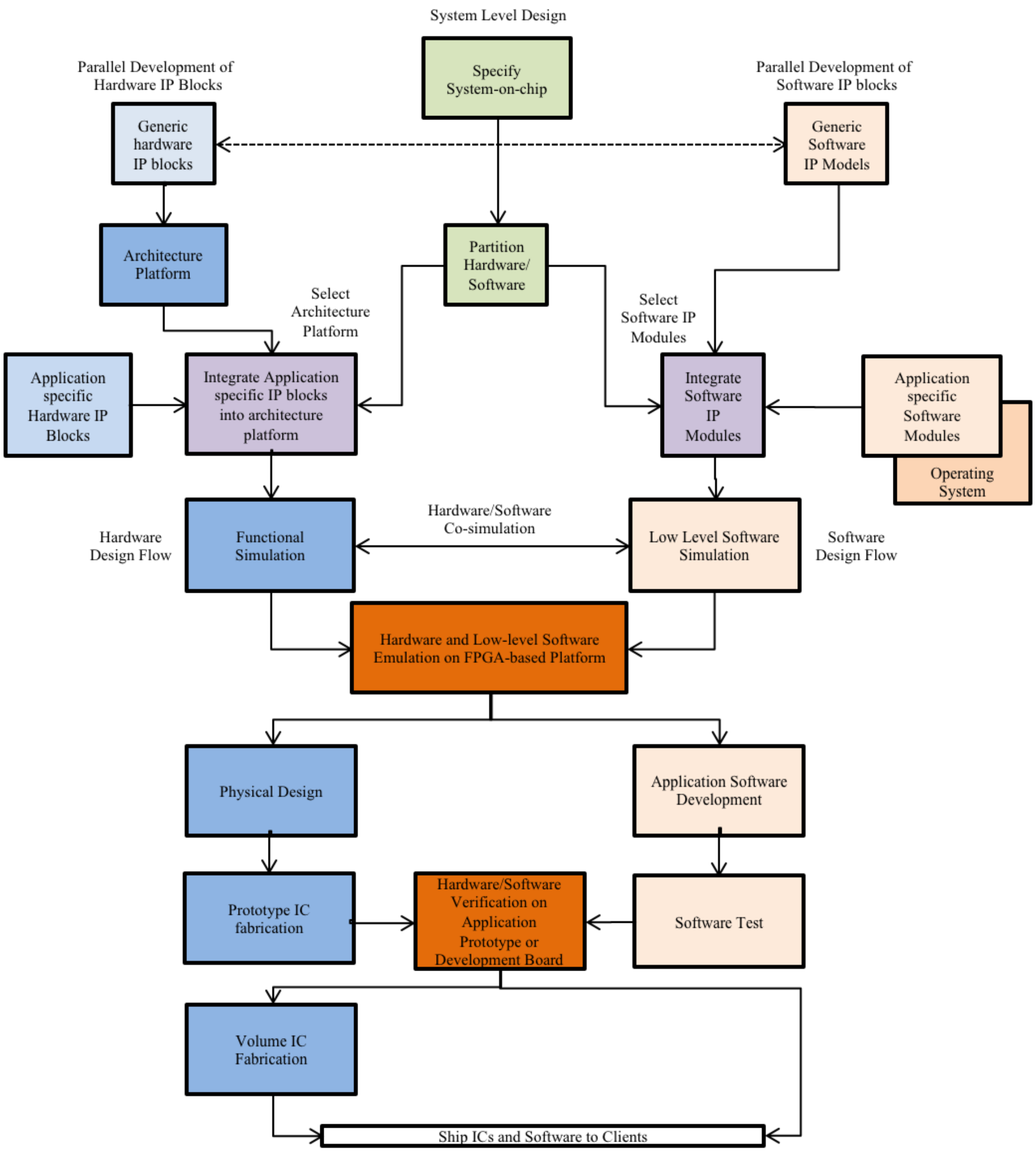
3.1 Overview

Design and implementation of a System-on-Chip needs complete end-to-end development and integration of both hardware and software subsystems. A fundamental decision in SoC design is to determine which features of the system are to be implemented in hardware and in software. The benefits and drawbacks of hardware and software implementations are summarized in table 2.1.

Implementation \ Trade-off	**Benefits**	**Drawbacks**
Hardware	Fast, Low Power Consumption	Inflexible, Unadaptable, Complex to Build and Test
Software	Flexible, Adaptive, Simple to Build and Test	Slow, High Power Consumption

Table 3.1: Hardware and Software Implementation Trade-offs

Given that hardware and software have complementary features, SoC designs aim to combine the individual benefits of the two. Therefore, an efficient design methodology aims to implement the performance critical parts of the application in hardware, and the rest in software. Further, a good design flow (shown in Fig 2.1) involves parallel development of both the hardware and software subsystems to effectively reduce time-to-market. The hardware IPs are put together using CAD tools and the software modules are integrated using a software development environment.



System Level Design
Specify System-on-chip
Parallel Development of Hardware IP Blocks
Parallel Development of Software IP blocks
Generic hardware IP blocks
Generic Software IP Models
Architecture Platform
Partition Hardware/ Software
Select Architecture Platform
Select Software IP Modules
Application specific Hardware IP Blocks
Integrate Application specific IP blocks into architecture platform
Integrate Software IP Modules
Application specific Software Modules
Operating System
Hardware Design Flow
Functional Simulation
Hardware/Software Co-simulation
Low Level Software Simulation
Software Design Flow
Hardware and Low-level Software Emulation on FPGA-based Platform
Physical Design
Application Software Development
Prototype IC fabrication
Hardware/Software Verification on Application Prototype or Development Board
Software Test
Volume IC Fabrication
Ship ICs and Software to Clients

Fig 3.1: System-on-Chip Design Flow

3.2 Hardware Subsystem

At the system level, an SoC is a collection of hardware components that are intricately interconnected to perform the specified functions for end users. Therefore, an SoC design is a "product creation process" which starts by identifying the needs of the end-user and ends at delivering a product with high reliability and functionality. Most SoCs are developed from pre-qualified hardware IP cores (IP reuse) and/or by extending known platforms (Architecture reuse) to overcome design complexities and improve time-to-market. Once the overall architecture of the SoC has been defined, new hardware elements are designed and integrated into the same design to facilitate better performance of the application.

3.3 Software Subsystem

Once hardware IP blocks are finalized, software needs to be configured for the selected set of peripherals, i.e., device drivers have to be written so that application programs can communicate with the peripherals. High level application programs have to be compiled for the ISA of the processor with the help of cross-compilers and software toolchains.

4. OpenRISC Instruction Set Architecture

4.1 Basic Overview

An instruction set architecture (ISA) is an abstract model of a computer that serves as a boundary between software and hardware. The realization of an ISA is called an implementation. An ISA permits multiple implementations that may vary in performance, size, power consumption and cost. Since, ISA is the interface between hardware and software, software written for an ISA can run on different implementations of the same ISA. This has enabled binary compatibility between different generations of computers. For these reasons, the ISA is one of the most important abstractions in computing today.

4.2 The OpenRISC ISA

The OpenRISC 1000 architecture allows for a spectrum of system implementations at a variety of price/performance points for a wide range of applications. It is a 32/64-bit load and store RISC architecture designed with emphasis on performance, simplicity, low power requirements, and scalability. It targets medium and high performance embedded computer environments. The OpenRISC ISA has the following features:

- Linear 32-bit or 64-bit address space (currently no 64-bit implementations) Uniform-length instructions.
- The architecture defines an 8-bit byte, 16-bit halfword, a 32-bit word, and a 64-bit double word. It also defines IEEE-754 compliant 32-bit single precision float and 64-bit double precision float storage units. 64-bit vectors of bytes, 64-bit vectors of halfwords, 64-bit vectors of single words, and 64-bit vectors of single precision floats are also defined.
- Two simple addressing modes: Register indirect with displacement and PC relative.
- The OpenRISC architecture implements Most Significant Byte (MSB) ordering i.e., big endian byte ordering by default. But implementations can support Least Significant Byte (LSB) ordering as well.
- The OpenRISC register set includes,

   a. Thirty-two or sixteen 32/64-bit general-purpose registers.
   b. All other registers are special-purpose registers defined accessible through l.mtspr/l.mfspr instructions.

- Reserved opcodes for custom instructions

The OpenRISC 1000 instruction set is listed in Table 4.1

Sl. No.	Instruction	Description
1	l.add $rD, $rA, $rB	Basic AND operation
2	l.addi $rD, $rA, $lo16	AND immediate
3	l.bal ${disp-26}	Branch and link – pc relative iaddr
4	l.bf ${disp-26}	Branch if condition bit is set – pc relative iaddr
5	l.bnf ${disp-26}	Branch if condition bit not set – pc relative iaddr
6	l.brk ${uimm-16}	Break (exception)
7	l.div $rD, $rA, $rB	Divide (signed)
8	l.divu $rD, $rA, $rB	Divide (unsigned)
9	l.j ${abs-26}	Jump (absolute iaddr)
10	l.jal ${abs-26}	Jump and link (absolute iaddr)
11	l.jalr $rA	Jump register and link (absolute iaddr)
12	l.jr $rA	Jump register (absolute iaddr)
13	l.lbs $rD, ${simm-16} ($rA)	Load byte (sign extend)
14	l.lbz $rD, ${simm-16}($rA)	Load byte (zero extend)
15	l.lhs $rD, ${simm-16}($rA)	Load halfword (sign extend)

16	l.lhz $rD, ${simm-16}($rA)	Load halfword (zero extend)
17	l.lw $rD, ${simm-16}($rA)	Load word
18	l.mfsr $rD, $rA	Move from special purpose register
19	l.movhi $rD, $hi16	Move immediate high
20	l.mtsr $rA, $rB	Move to special purpose register
21	l.mul $rD, $rA, $rB	Multiplication operation
22	l.muli $rD, $rA, $lo16	Multiplication immediate signed
23	l.nop	No operation
24	l.or $rD, $rA, $rB	OR
25	l.ori $rD, $rA, $lo16	OR immediate
26	l.rfe $rA	Return from exception
27	l.ror $rD, $rA, $rB	Rotate right
28	l.rori $rD, $rA, ${uimm-5}	Rotate right immediate
29	l.sb ${ui16nc}($rA), $rB	Store byte
30	l.sfeq $rA, $rB	Compare for equality
31	l.sfeqi $rA, ${simm-16}	Compare Immediate for equality
32	l.sfges $rA, $rB	Compare signed (greater than or equal to)
33	l.sfgesi $rA, ${simm-16}	Compare Signed immediate (greater than or equal to)
34	l.sfgeu $rA, $rB	Compare Unsigned (greater than or equal to)
35	l.sfgeui $rA, ${uimm-16}	Compare unsigned immediate (greater than or equal to)
36	l.sfgts $rA, $rB	Compare signed (greater than)

37	l.sfgtsi $rA, ${simm-16}	Compare signed immediate (greater than)
38	l.sfgtu $rA, $rB	Compare unsigned (greater than)
39	l.sfgtui $rA, ${uimm-16}	Compare unsigned immediate (greater than)
40	l.sfles $rA, $rB	Compare signed (lesser than or equal to)
41	l.sflesi $rA, ${simm-16}	Compare signed Immediate (lesser than or equal to)
42	l.sfleu $rA, $rB	Compare unsigned (lesser than or equal to)
43	l.sfleui $rA, ${uimm-16}	Compare unsigned immediate (less than or equal to)
44	l.sflts $rA, $rB	Compare signed (lesser than)
45	l.sfltsi $rA, ${simm-16}	Compare signed Immediate (lesser than)
46	l.sfltu $rA, $rB	Compare unsigned (lesser than)
47	l.sfltui $rA, ${uimm-16}	Compare unsigned immediate (lesser than)
48	l.sfne $rA, $rB	Compare (unequality)
49	l.sfnei $rA, ${simm-16}	Compare immediate (unequality)
50	l.sh ${ui16nc}($rA), $rB	Store half word
51	l.sll $rD, $rA, $rB	Logical left shift
52	l.slli $rD, $rA, ${uimm-5}	Logical left shift immediate
53	l.sra $rD, $rA, $rB	Right shift
54	l.srai $rD, $rA, ${uimm-5}	Right shift immediate
55	l.srl $rD, $rA, $rB	Logical right shift
56	l.srli $rD, $rA, ${uimm-5}	Logical right shift immediate
57	l.sub $rD, $rA, $rB	Subtract
58	l.subi $rD, $rA, $lo16	Subtract immediate

59	l.sw ${ui16nc}($rA), $rB	Store single word
60	l.sys ${uimm-16}	System call instruction
61	l.xor $rD, $rA, $rB	XOR
62	l.xori $rD, $rA, $lo16	XOR immediate

Table 4.1: OpenRISC 1000 basic Instruction Set

The format of an instruction is illustrated with an example,

Instruction: l.add rD, rA, rB

31 26	25 . . . 21	20 . . . 16	15 . . . 11	10	9 8	7 . . 4	3 . . 0
opcode 0x38	D	A	B	reserved	opcode 0x0	reserved	opcode 0x0
6 bits	5 bits	5 bits	5 bits	1 bit	2 bits	4 bits	4 bits

5. System-on-Chip Design

5.1 Overview

The SoC hardware was designed by sourcing and integrating the following subsystems.

1. The Debug System
2. General Purpose I/O Core
3. UART 16550 Core
4. MOR1kx Processor Core
5. OR1k Bootloader Core
6. SDRAM Controller Core
7. WishBone Interconnect Core
8. I2C Controller Core
9. SPI Controller Core
10. WishBone RAM Core

5.2 The Debug System

5.2.1 Overview of a Debug System

Devices communicate with the external world through a set of I/O pins. However, these pins provide limited visibility into the working of the device. JTAG uses boundary-scan technology, which creates a virtual access capability that circumvents normal inputs and provides direct control improving visibility at the outputs. To provide boundary scan capability, additional circuitry along with a dedicated scan path has to be added. The overhead of this additional logic is minimal and well worth the price to have efficient testing at the board level. The boundary-scan control signals, collectively referred to as the Test Access Port (TAP), define a serial protocol for scan-based devices. The five pins/signals are:

1. TCK/Clock, synchronizes the internal state machine operations.
2. TMS/Mode Select, is sampled at the rising edge of TCK to determine the next state.

3. TDI/Data In, is sampled at the rising edge of TCK and shifted into the device's test or programming logic when the internal state machine is in the correct state.
4. TDO/Data Out, represents the data shifted out of the device's test or programming logic and is valid on the falling edge of TCK when the internal state machine is in the correct state.
5. TRST/Reset (optional), when driven low resets the internal state machine.

The TCK, TMS, and TRST input pins drive the TAP controller FSM which manages the exchange of data and instructions. The controller advances to the next state based on the value of the TMS signal at each rising edge of TCK. Extensions can be developed around JTAG to implement software debug functions. Having extra pins on a device provides additional system integration capabilities for benchmarking, profiling, and system level breakpoints. With proper support built into a target CPU, the interface can be used to download code, execute it, and examine register and memory values. These functions cover functionality of a typical low-level debugger.

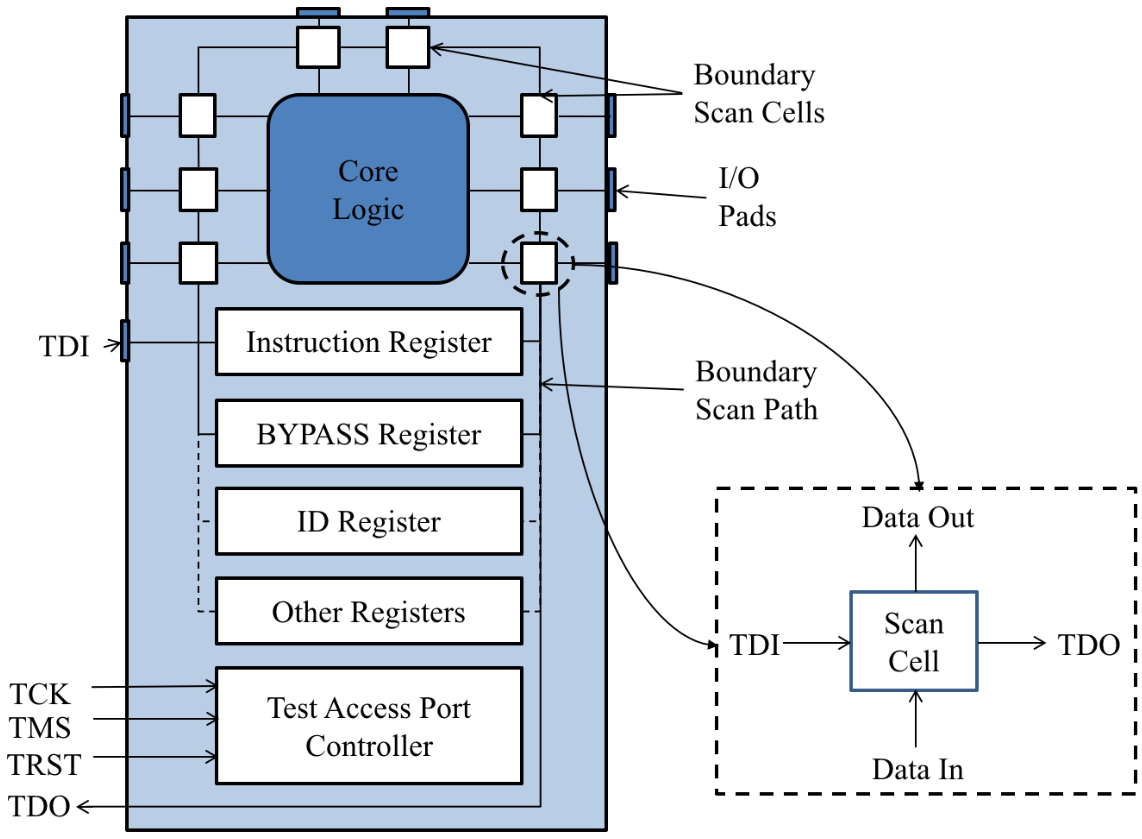


Fig 5.2.1: JTAG block diagram

5.2.2 Advanced Debug System

The Advanced Debug System is a suite of IP cores and software programs designed to allow end-users to download code to a target CPU and perform source-level debugging. Currently, systems using OpenRISC processors and WishBone interconnect are supported by this suite.

The designed SoC includes the following components to complete the debugger hardware (as shown in Fig 5.2.2):

1. "adv_dbg_if" core, interfaced directly to MOR1kx processor and WishBone bus.

   The hardware debug module ("adv_dbg_if" core) controls transactions to the CPU and the WishBone bus. It also provides the clock domain synchronization between the CPU, WishBone bus, and JTAG TAP. This module also includes the JTAG Serial Port (JSP), if enabled. The debug module decodes the protocol sent via JTAG by the 'bridge' software program. This protocol includes CPU stall and reset commands, CPU register reads and writes, WishBone data reads and writes, and bidirectional JSP data transfers. It is the primary hardware component of the debug system.

2. "altera_virtual_jtag" core, acting as a connection between the "adv_dbg_if" core and the external pins of the target chip.

   The Altera Virtual JTAG core provides direct access to JTAG control signals routed to the FPGA core logic. It gives fine granularity of control over the JTAG resource and opens up the JTAG resource as a general-purpose serial communication interface. It allows end-users to connect the debug hardware through the FPGA's TAP (the same TAP used to download a bitstream to the FPGA). This means that separate, dedicated pins for a debug system TAP are not required and hence, effectively eliminating the requirement of a second cable. This TAP may be used only when the system is implemented in an Altera FPGA.

3. "jtag_tap" core, represents a standard JTAG TAP described by IEEE 1149.1 standard.

   The standard JTAG core is accessed by four (or five) external pins, and includes an Instruction Register and several Data Registers.

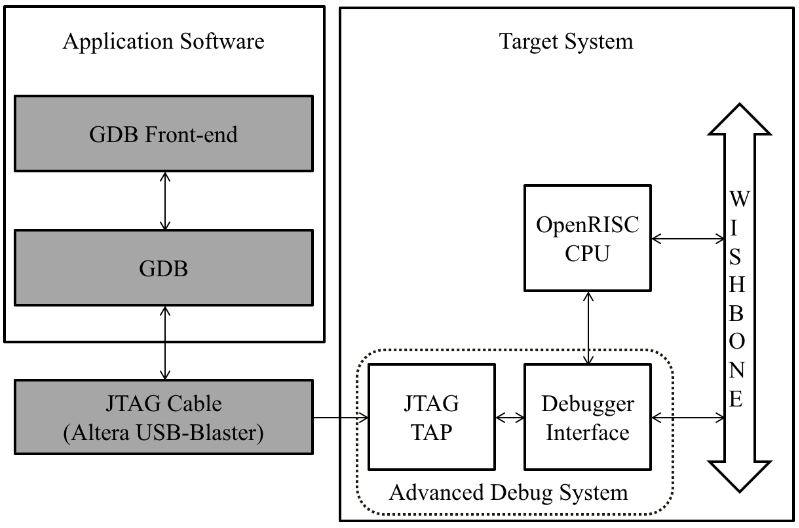


Fig 5.2.2: Debugging system block diagram

5.3 General Purpose I/O Core

The GPIO IP core is a user-programmable general-purpose I/O controller. The main features of the core are:

1. The number of I/O signals is customizable and can range from 1 to 32. For more I/Os several GPIO cores can be used in parallel.
2. All general-purpose I/O signals can be bi-directional.
3. The core has WishBone SoC Interconnection compliant interface.

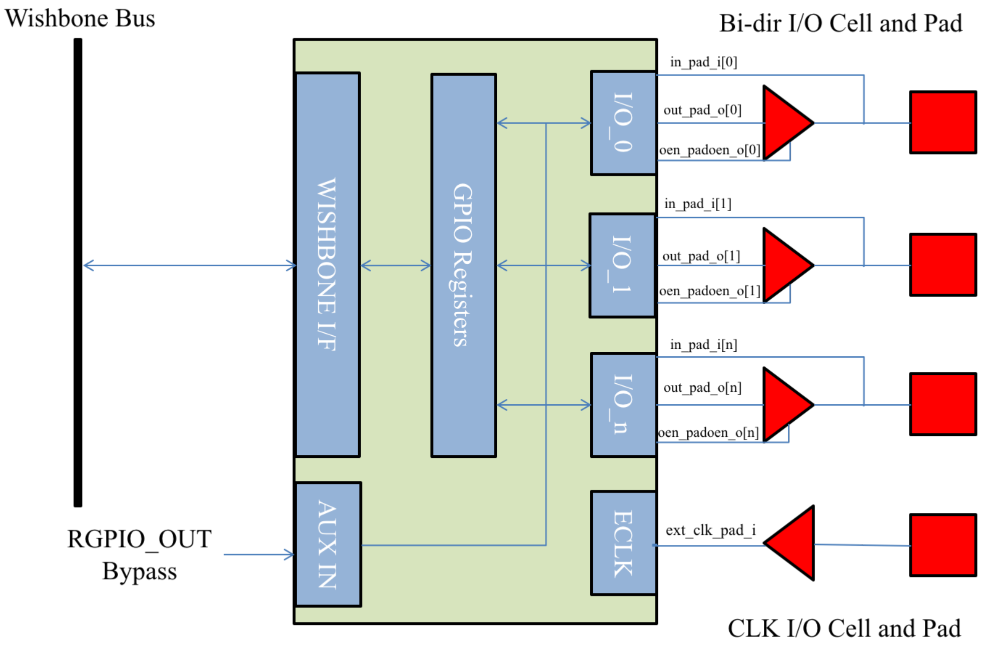


Fig 5.3.1: GPIO Core Architecture

Figure 5.3.1 shows the architecture of the GPIO IP core. It consists of four main building blocks:

1. WishBone host interface: This connects the core to the host system. The implementation only supports a 32-bit bus width.
2. GPIO registers: The core has several software accessible registers. Most registers have the same width as the number of signals and can be from 1 – 32 bits. The host programs the type and operation of each GPIO signal through these registers.
3. Auxiliary inputs: The auxiliary inputs can bypass RGPIO_OUT outputs based on programming of RPGIO_AUX register. Auxiliary inputs are used to multiplex other on-chip peripherals on GPIO pins.
4. Interface to external I/O cells and pads: This connects the core to external I/O ring cells and pads. The ECLK signal can be used to register inputs based on external clock reference.

Port	Width	Direction	Description
in_pad_i	1-32	Inputs	GPIO inputs
out_pad_o	1-32	Outputs	GPIO outputs
oen_padoen_o	1-32	Outputs	GPIO output enables (for three-state drivers)
ext_clk_pad_i	1	Input	Alternative GPIO inputs' latch clock

Table 5.3.1: External Interface

5.4 UART 16550 Core

The UART (Universal Asynchronous Receiver/Transmitter) core provides serial communication capabilities (data format and transmission speeds are configurable), which allow communication with modem or other external devices, like another computer using a serial cable and RS232 protocol. Each UART contains a shift register, to convert between serial and parallel data. Serial transmission of digital information (bits) through a single wire is less costly than parallel transmission through multiple wires. The transmitting UART receives data from a controlling device like a CPU and transmits it in serial to the receiving UART, which then converts the serial data back into parallel data for the receiving device. One or more UART peripherals are commonly integrated in microcontroller chips. The core architecture is shown in Fig 5.4.1.

The features of the UART core are:

- WishBone interface in 32-bit or 8-bit data bus modes (selectable)
- FIFO only operation
- Register level and functional compatibility with NS16550A.
- Debug Interface in 32-bit data bus mode

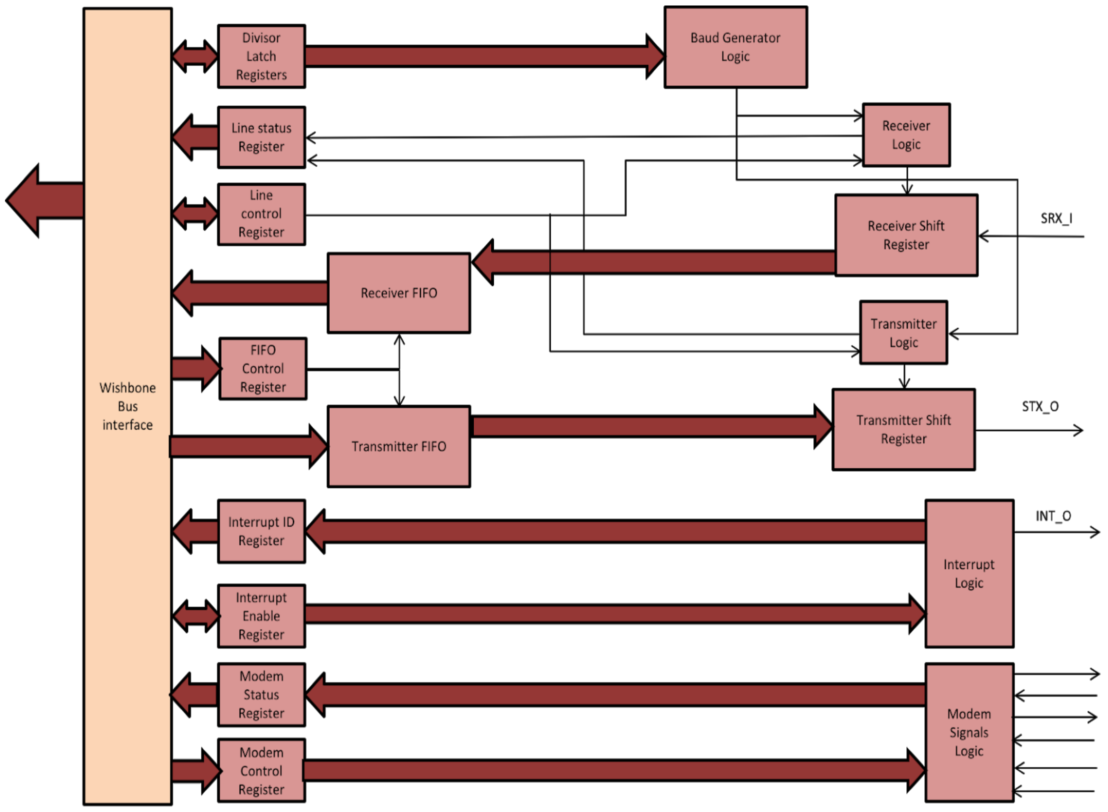


Fig 5.4.1: UART 16550 Core Architecture

5.5 MOR1kx Processor Core

This IP core is an implementation of the OpenRISC 1000 instruction set architecture. The mor1kx CPU can be configured and customized extensively to meet user requirements. The blocks within this core (shown in figure 5.5) have been developed to provide maximum reusability and customizability.

There are currently 3 pipeline implementations available for the mor1kx core. They are as follows:

1. Cappuccino - 6 stage, timers, delay slot, debug unit, PIC, tightly coupled cache, MMU
2. Espresso - 2 stage, delay slot, debug unit, timers, PIC
3. Pronto espresso - 2 stage, no delay slot, debug unit, timers, PIC

In the design, the Cappuccino variant of the core was used. The features of this implementation are:

- A 6-stage pipeline. (Address, Fetch, Decode, Execute, Control/Memory and Writeback)
- Caches supported
- MMUs supported

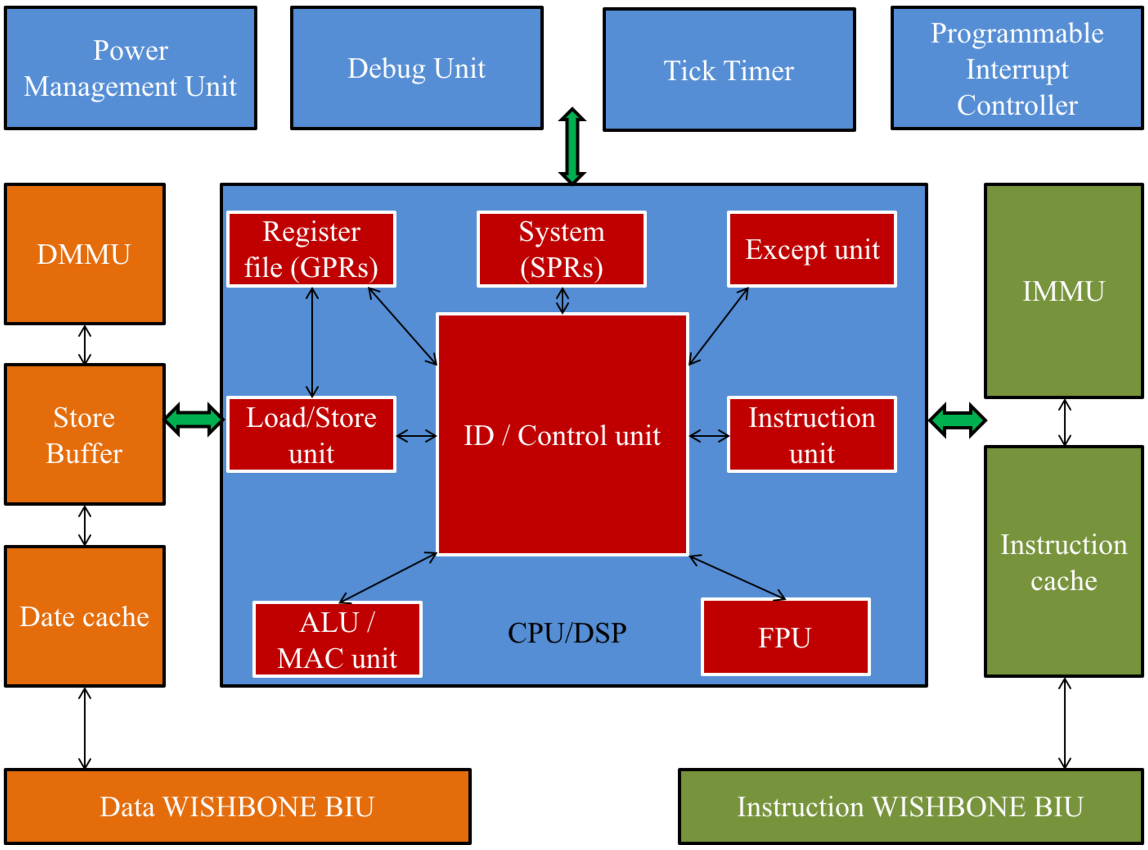


Fig 5.5: CPU Core

5.6 OR1k Bootloader Core

When a CPU starts, it has certain preset values in its registers and is usually unaware of the on-board memory. Thus, it expects to find program code at a specific address, this address usually points to ROM or Flash, this is the beginning of bootloader code. The first task of bootloader is to map the RAM to predefined addresses. After RAM is mapped, the Stack Pointer is setup. This is the minimal setup required for the bootloader to start it's work. Since it is the first software to run after powerup or reset, it is highly processor and board specific. The bootloader performs the necessary initializations to prepare the system for the Operating system.

5.7 SDRAM Controller Core

Dynamic memories are more complicated to drive than static ones as there are rows, columns, banks and refresh cycles to take care of. But SDRAMs provide high-speed and low cost per bit. Therefore, we need a way to access an SDRAM with ease. Hence, memory controllers are created which act as translation layers: on one side, they provide to the user an easy to use memory interface, and then do the dirty work to drive the real SDRAM signals.

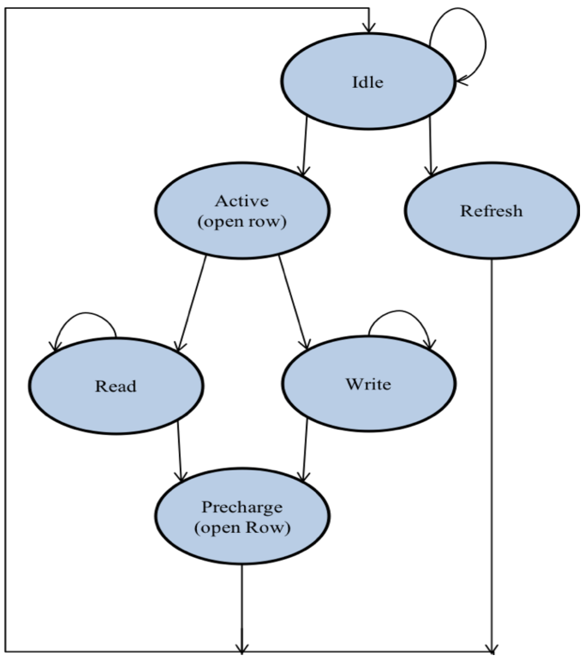


Fig 5.7.1: FSM of SDRAM controller

The SDRAM Controller core has the following features:

- WishBone compliant
- Operates at 100Mhz, CAS 3, 32MB, 16-bit data
- On reset will go into INIT sequence
- After INIT the controller sits in IDLE waiting for REFRESH, READ or WRITE
- REFRESH operations are spaced evenly 8192 times every 32ms
- READ is always single read with auto precharge
- WRITE is always single write with auto precharge

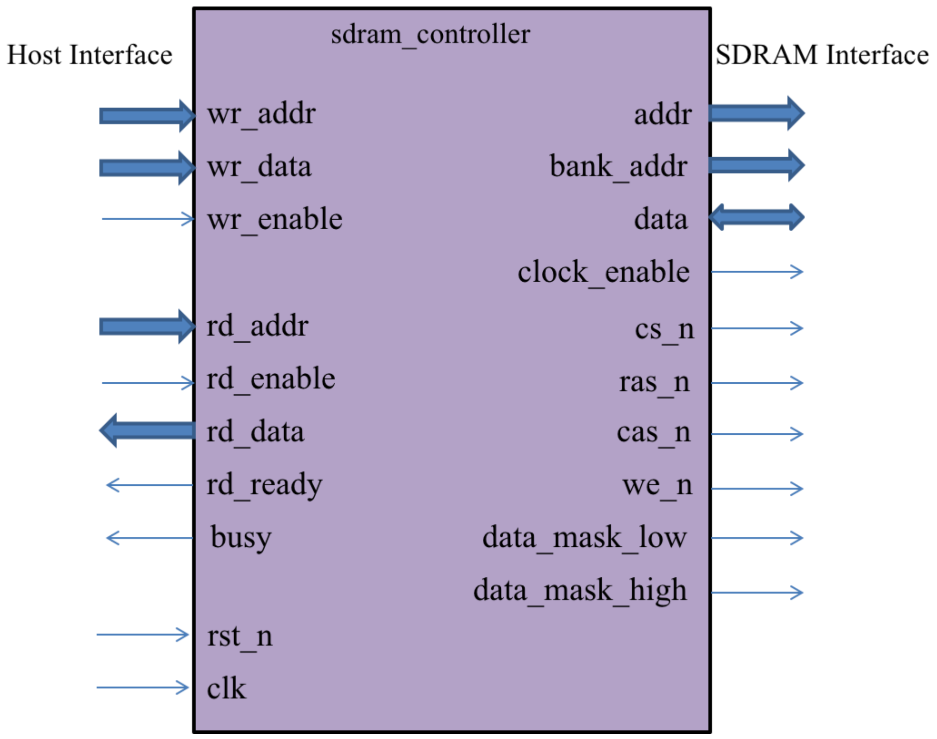


Figure 5.7.2: SDRAM Controller Core

The signals can be seen in the figure 5.7.2 and most signals are self-explanatory. However, a few are described below:

1. wr_addr and rd_addr, are equivalent to the concatenation of {bank, row, column}.
2. rd_enable, should be set to high once an address is presented on the addr bus and has to be read data.

3. wr_enable, should be set to high once addr and data is offloaded onto the bus.
4. busy, will go high when read/write command is acknowledged and will go low when write/read operation is complete.
5. rd_ready, will go high when data rd_data is available on the data bus.

5.8 WishBone Interconnect Core

The WishBone Interconnection Architecture is a flexible design methodology intended to be used with portable semiconductor IP cores. Its purpose is to foster design reuse by alleviating System-on-Chip integration problems. This is accomplished by creating a common interface between IP cores. By adopting a standard interconnection scheme, the cores can be integrated more quickly and easily by the end user. The WishBone master/slave signals are listed and explained in Table 5.8.1.

Wishbone Signal	**Description**	**Signal Type**
CLK_I ()	Coordinates all activities for the internal logic. All signals are with respect to the rising edge of CLK_I.	Common
DAT_I ()	Data input array is used to pass binary data.	Common
DAT_O ()	Data output array is used to pass binary data.	Common
RST_I ()	Forces Wishbone interfaces to restart and all state machines to their initial state.	Common
TGD_I ()	Holds information associated with DAT_I () and verified by STB_I (). Simplify task of defining new signals.	Common
TGD_O ()	Holds information associated with DAT_O () and verified by STB_O (). Simplify task of defining new signals.	Common
ACK_I ()	If asserted, indicates the normal termination of a bus signal.	Master

ADR_O ()	Used to pass binary address. The higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size and granularity.	Master
CYC_O ()	Indicates if a valid bus signal is in process. The signal is asserted for the duration of all bus cycles.	Master
STALL_I ()	Indicates if the current slave is not able to accept transfer in the transaction queue.	Master
ERR_I ()	Indicates if there is an abnormal cycle termination.	Master
LOCK_O ()	Indicates if the current bus cycle is uninterruptible.	Master
RTY_I ()	Indicates that the interface is not ready to accept data and the transaction must be retried.	Master
SEL_O ()	Indicates where valid data is expected on the DAT_I () signal.	Master
STB_O ()	Indicates a valid data transfer cycle.	Master
TGA_O ()	Contains information on the ADR_O () lines and is verified by the STB_O () line.	Master
TGC_O ()	Contains information about the bus cycles and is verified by CYC_O line.	Master
WE_O ()	Indicates if it is READ or a WRITE cycle.	Master
ACK_O ()	Indicates termination of a bus cycle.	Slave
ADR_I ()	Used to pass binary address. The higher array boundary is specific to the address width of the core, and the lower array boundary is determined by the data port size and granularity.	Slave
CYC_I ()	Indicates if a valid bus signal is in process. The signal is asserted for the duration of all bus cycles.	Slave

STALL_O ()	Indicates if the current slave is not able to accept transfer in the transaction queue.	Slave
ERR_O ()	Indicates if there is an abnormal cycle termination.	Slave
LOCK_I ()	Indicates if the current bus cycle is uninterruptible	Slave
RTY_O ()	Indicates that the interface is not ready to accept data and the transaction must be retried	Slave
SEL_I ()	Indicates where valid data is expected on the DAT_O () signal	Slave
STB_I ()	Indicates a valid data transfer cycle	Slave
TGA_I ()	Contains information on the ADR_I () lines and is verified by the STB_I () line	Slave
TGC_I ()	Contains information about the bus cycles and is verified by CYC_I () line	Slave
WE_I ()	Indicates if it is READ or a WRITE cycle	Slave

Table 5.8.1: WishBone Interconnect signals

The important Wishbone operations and their protocols are listed below:

1. Reset Operation: All hardware interfaces are set to their initial states through the RST_O signal. This is also connected to the RST_I signal on all master/slave interfaces.
2. Transfer Cycle Initiation:Master interfaces initiate a transfer cycle by asserting the CYC_O signal. If this signal is negated, all master devices are invalid. Slave devices respond only on the assertion of the CYC_I signal.
3. Handshaking Protocol: There are two types of handshaking protocols:
    a. Standard wishbone protocol: The master asserts the STB_O signal when it is ready to transfer data. This signal will remain asserted until the slave

responds with an ACK_I signal inferring that the transaction was successful.

b. Pipelined wishbone protocol:The master does not wait for an ACK signal from the slave but a STALL signal present in the master indicates if the slave is ready to accept another request.

5.9 I2C Controller Core

I2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange between devices. It is primarily used as a board level communications protocol. The IP Core provides an interface between a Wishbone Master and an I2C bus. It is an easy path to add I2C capabilities to any Wishbone compatible system. This core is provided so that future on-board peripherals can be easily interfaced to the designed SoC.

5.10 SPI Controller Core

SPI (Serial Peripheral Interface) is a serial, synchronous, full duplex communication protocol. It is widely used as a board-level interface between different devices such as microcontrollers, DACs, ADCs and others. This core is provided so that future on-board peripherals can be easily interfaced to the designed SoC.

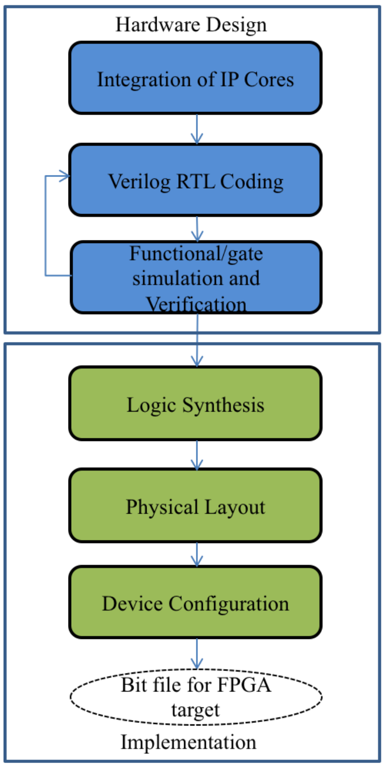
5.11 WishBone RAM Core

This is a generic memory IP that is intended to map against on-chip RAM or registers. It is a behavioral description and can be used to simulate the working of RAM in early developmental stages of an SoC.

6. Tool-flow

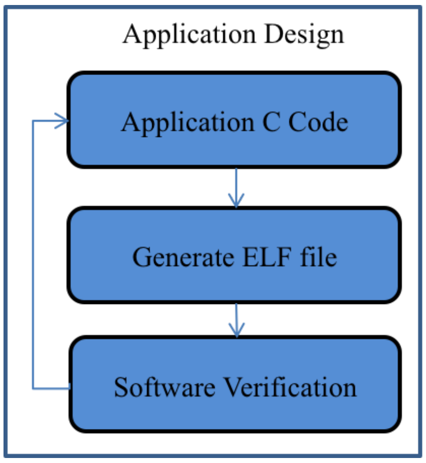
6.1 Overview

Tool-flows are unique to every project and establishing the right flow is arguably the most important step of the project. The hardware tool-flow (figure 6.1) and software tool-flow (figure 6.2) used in our project is shown below.



Tool-flow	Tools
IP Integration	FuseSoC
Verilog Design	Sublime Text, Atom, Gedit
Simulation / Verification	Modelsim, Icarus Verilog
Waveform Viewer	GTKWave
Synthesis	Altera Quartus II
Physical Design and Implementation	Altera Quartus II

Fig 6.1.1: Hardware design tool-flow and design overview



Tool-flow	Tools
Application C Code	Sublime Text, Atom, Gedit
ELF generation	Or1k toolchains
Software Verification	Or1ksim Architectural simulator

Fig 6.1.2: Application development tool-flow and design overview

6.2 Hardware Design tool-flow

6.2.1 IP integration

The IP cores were integrated using FuseSoC which is an open source package manager for HDL code. FuseSoC makes it easier to reuse existing cores; create compile-time or run-time configurations; run regression tests against multiple simulators and port designs to new targets. The directory structure and HDL design structure of the IP cores, along with simulator information are described in Core API (CAPI) files which is used by FuseSoC to aggregate and integrate the specified IP cores. Then, a configuration file is written to specify the following information about IP cores: the Master-Slave relationships, memory-mapped location on the interconnect and their bus size. The interconnect generator uses this information to generate the WishBone Interconnect stitching together all the specified IP cores.

6.2.2 Verilog Design

The selected IP cores must be instantiated and connected in a top module. Certain IP cores require tweaks in their HDL and/or change in their architecture depending on the application. Some cores require wrapper modules to interface them to the interconnect. These are designed and developed based on the application.

6.2.3 Simulation/Verification

The individual IP cores must be verified before integration. This was done through Modelsim. The integrated SoC must be extensively verified for functionality before implementation. This was achieved through FuseSoC by specifying and invoking the correct simulator (Icarus) and waveform viewer (GTKWave).

6.2.4 Synthesis and Implementation

Synthesis is a process in which algorithms are employed to perform logic minimization and translate the HDL design into a functionally equivalent netlist based on design and constraint files. The generated netlist is then mapped onto FPGA devices such as LUTs, block memories, registers, etc. Once this is done, the mapped devices are interconnected in the routing step. If the design is successfully implemented, a bit file which can downloaded on the target FPGA is generated. The figure below shows the design flow in Altera Quartus II:

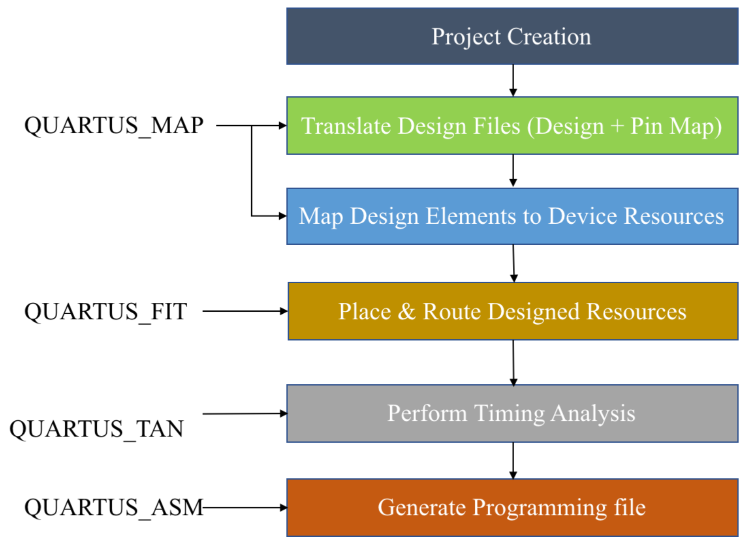

Project Creation
QUARTUS_MAP
Translate Design Files (Design + Pin Map)
Map Design Elements to Device Resources
QUARTUS_FIT
Place & Route Designed Resources
QUARTUS_TAN
Perform Timing Analysis
QUARTUS_ASM
Generate Programming file

Fig 6.2.1: Altera Quartus II FPGA design flow

6.3 Software Design tool-flow

Applications can be developed using C programming language. The developed application is cross-compiled for the OpenRISC architecture using or1k GNU toolchains which generates ELF/binary file. The compiled application can be verified on the OpenRISC ISA simulator, or1ksim. After suitable testing and debugging, the code can be flashed onto the target system.

7. Platform

The designed SoC was successfully implemented on the DE0-Nano board (Fig 6.1) which has the following specifications:

1. Altera Cyclone® IV EP4CE22F17C6N FPGA
   - 22,320 Logic elements (LEs)
   - 594 Embedded memory (Kbits)
   - 66 Embedded 18 x 18 multipliers
   - 4 General-purpose PLLs
   - 153 FPGA I/O pins
2. Configuration Status and Set-Up Elements
   - On-board USB-Blaster circuit for programming
   - FPGA Serial Configuration Device (EPCS)
3. Expansion Header
   - Two 40-pin Headers (GPIOs) provides 72 3.3V I/O pins
   - Two 5V power pins, two 3.3V power pins and four ground pins
   - One 26-pin header provides 16 3.3V digital I/O pins and 8 analog input pins to connect to analog sensors, etc
4. Memory Devices
   - 32MB SDRAM
   - 2Kb I2C EEPROM
5. General User Input/Output
   - 8 green LEDs
   - 2 debounced push-buttons
   - 4 dip switches
6. G-Sensor
   - ADI ADXL345, 3-axis accelerometer with high resolution (13-bit)
7. 7. A/D Converter
   - NS ADC128S022, 8-Channel, 12-bit A/D Converter
   - 50 ksps to 200 ksps
8. Clock System
   - On-board 50MHz clock oscillator
9. Power Supply

- USB Type mini-AB port (5V)
- Two DC 5V pins of the GPIO headers (5V)
- 2-pin external power header (3.6-5.7V)

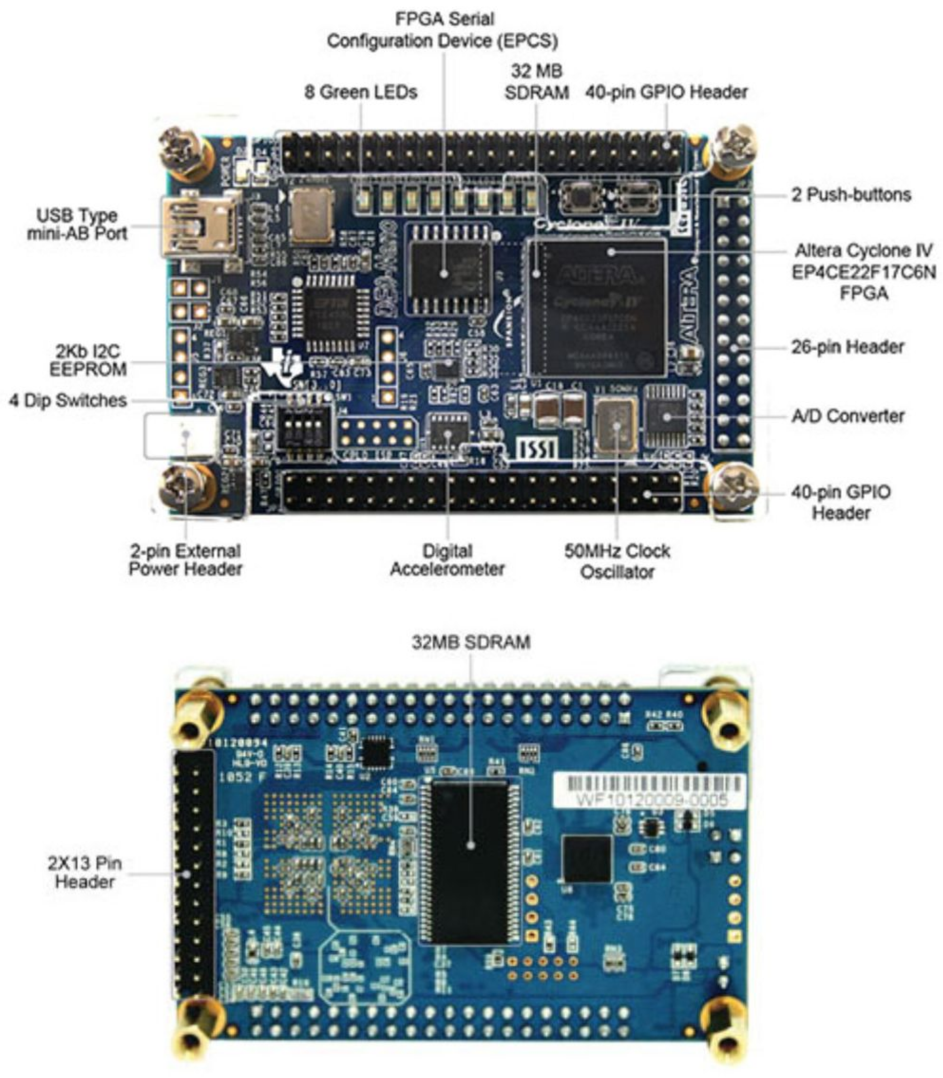


Fig 6.1: DE0-Nano Board Layout

8. Application: Image Processing

8.1 Overview

Image processing is a method to perform some operations on an image, in order to get an enhanced image or to extract some useful information from it. It is a type of signal processing in which input is an image and output may be image or characteristics/features associated with that image. Digital image processing is the use of computer algorithms to perform image processing on digital images. The three general phases that all types of data have to undergo in digital image processing are: pre-processing, enhancement, and display/information extraction.

8.2 Basic image processing functions

Some basic image processing functions were implemented in C, cross-compiled and executed on the SoC. The source and processed images are shown in the figure 7.2.

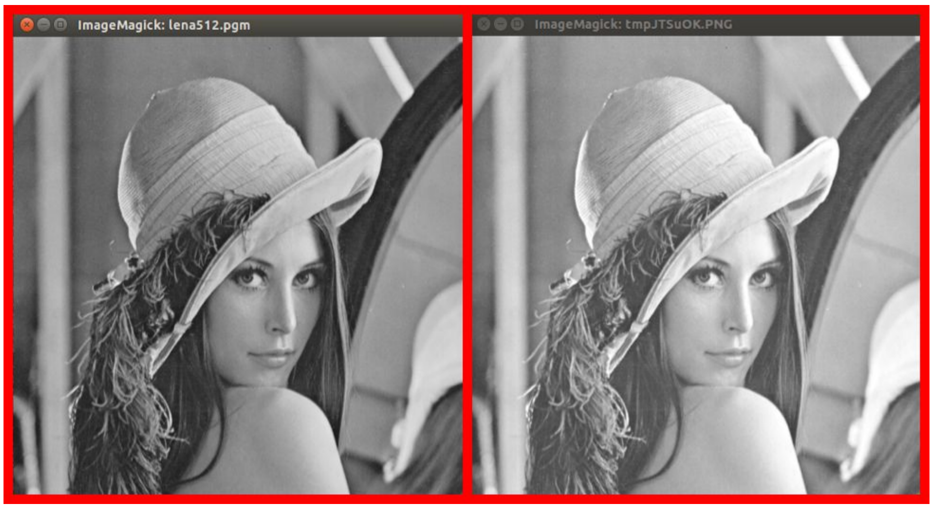


Fig 8.2.1: Adjusting brightness

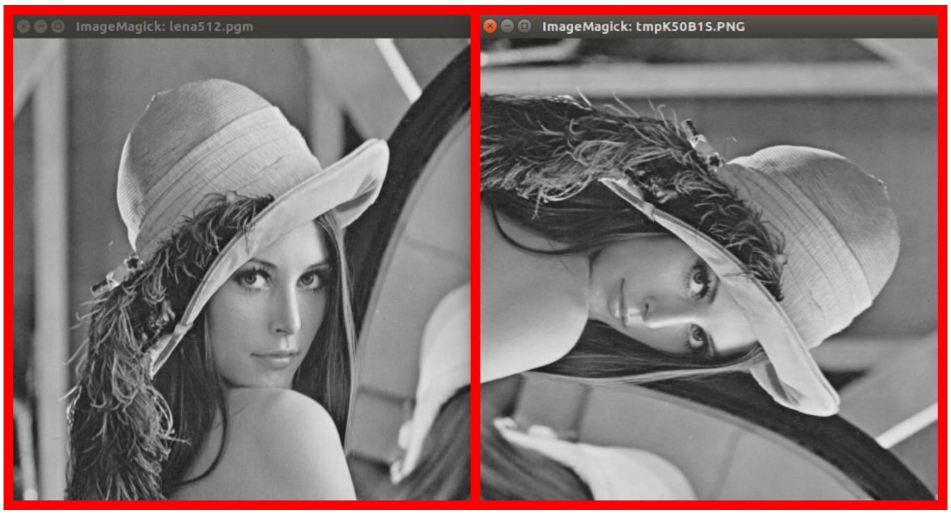


Fig 8.2.2: Image rotation

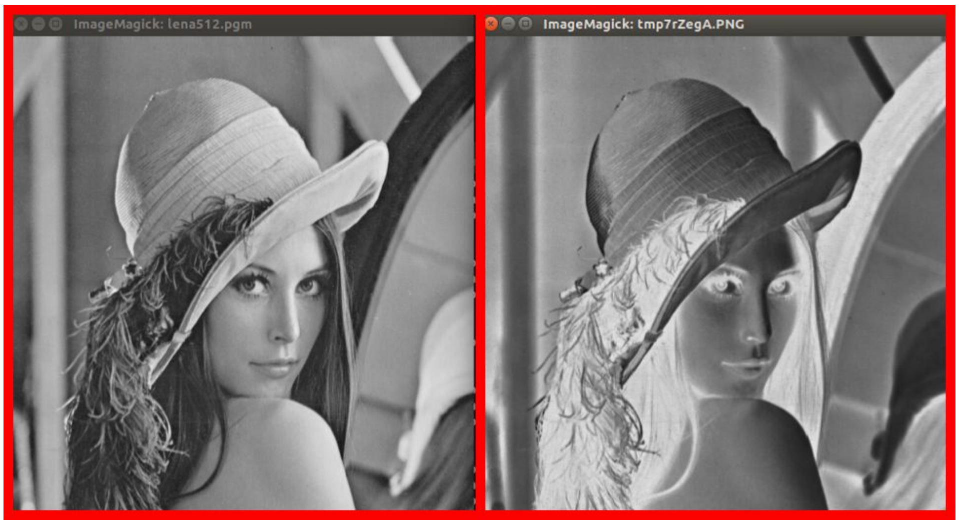


Fig 8.2.3: Inverting color

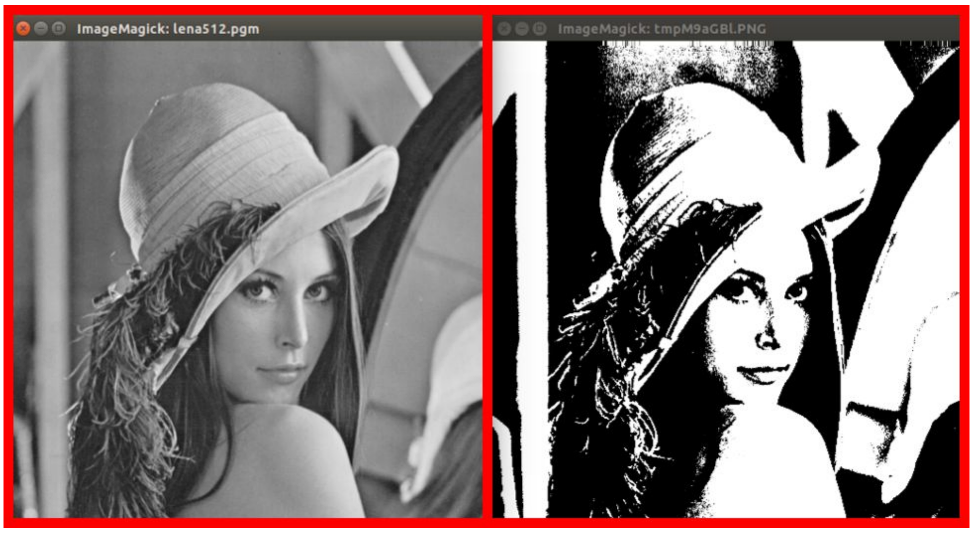


Fig 8.2.4: Contrast

7.3 Edge Detection (Canny Algorithm)

The Canny edge detection algorithm can be split into the following 5 steps:

1. Removal of noise from the image: To perfectly detect the edges of an image, noise removal is a very important step. To completely ensure that maximum noise is removed, the given image is convoluted with a dynamic gaussian filter based on the size of the image. This step is essential to prevent false detection of edges. Once the Image has been convoluted with the gaussian filter, the next step is to find the intensity gradient of the image.
2. Intensity gradient: This step is necessary to find the edges of the image. The canny algorithm uses 4 filters namely for the horizontal, vertical and the diagonal edges of an image. This is performed by finding the derivatives in that direction.
3. Non- maximum Suppression: Technique used to thin the edges and leave out the weaker edges. It helps to suppress the weaker edges and identify the correct edge to detect.
4. Double threshold: after suppression technique, the image is left with only the edges that contribute to the detection algorithm but there maybe a few edges contributing to noise and color variation. Double threshold is a technique to maximum eliminate these edges by filtering out the edges with weak gradient values.
5. Hysteresis: This step ensures that the last of all the weak edges that have not been filtered out after all these steps are finally removed. The edges are tracked repetitively until all the edges with a low gradient value are eliminated and thus leaving only the strong and necessary edges to contribute to the edge detection algorithm.

This algorithm was implemented in C, cross-compiled and executed on the SoC. The source and processed images are shown in the figure 7.3.1.

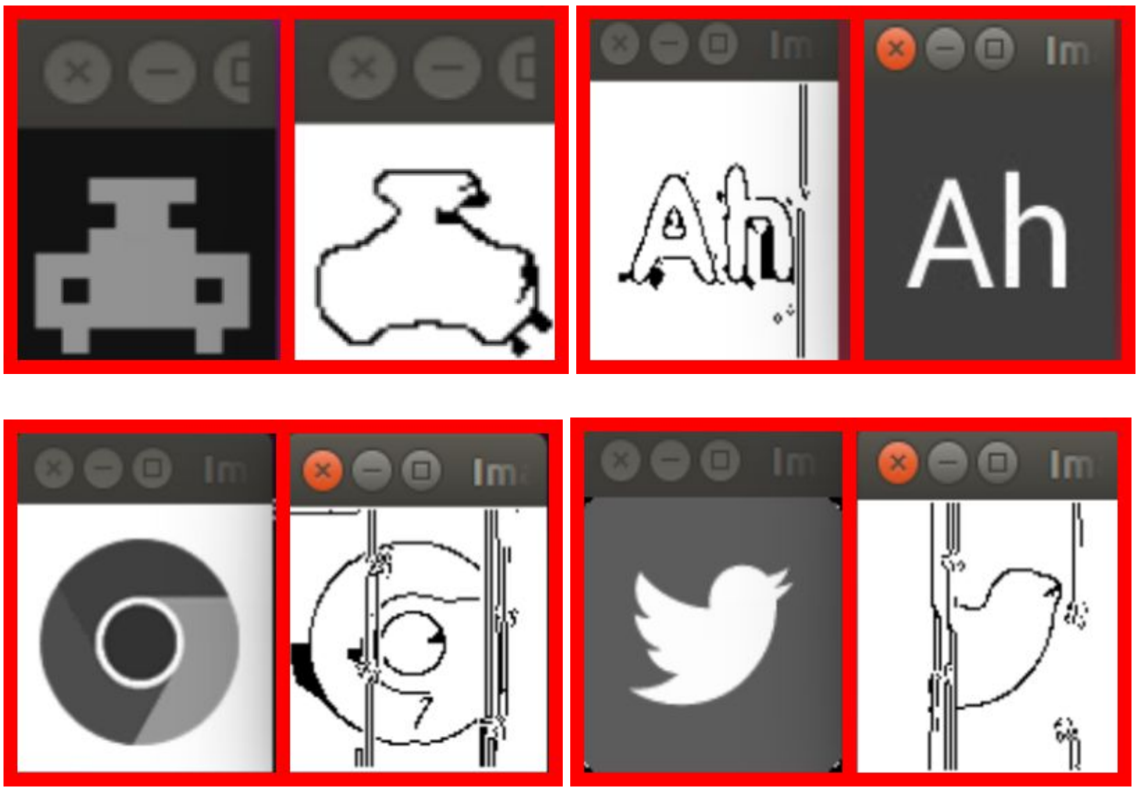


Fig 8.3.1: Edge Detection

9. Experiments

9.1 Integrating an FPU into the system

A standalone FPU IP core (figure 8.1.1) was successfully integrated into the system by designing a WishBone complaint controller as shown in figure 8.1.2. This core served as a placeholder and could be easily swapped with an accelerator/coprocessor depending on the application. This further shows the scalability of the system.

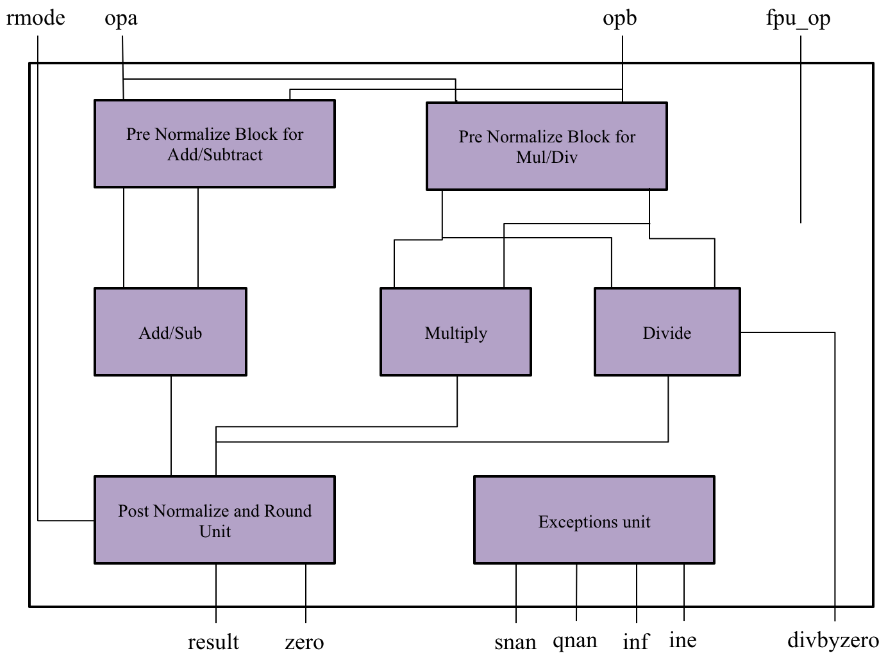

rmode
opa
opb
fpu_op
Pre Normalize Block for Add/Subtract
Pre Normalize Block for Mul/Div
Add/Sub
Multiply
Divide
Post Normalize and Round Unit
Exceptions unit
result
zero
snan
qnan
inf
ine
divbyzero

Fig 9.1.1: Architecture of the FPU Core

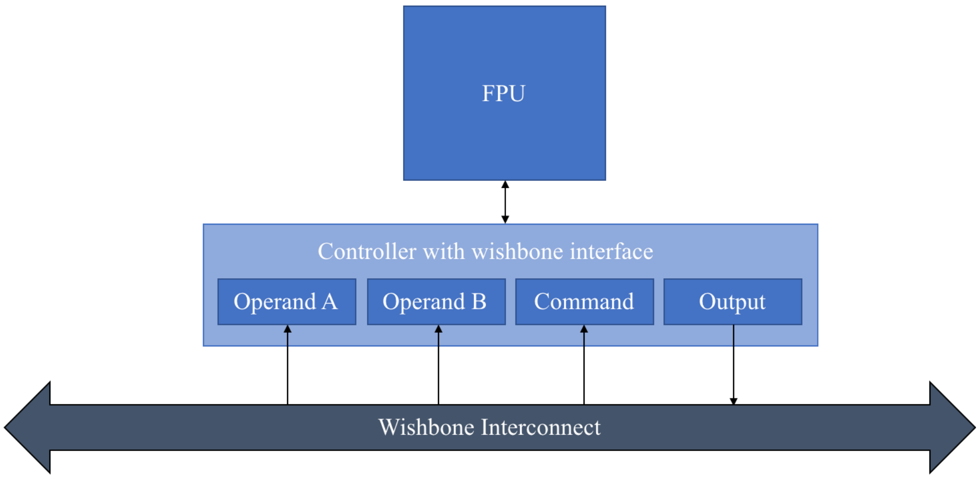
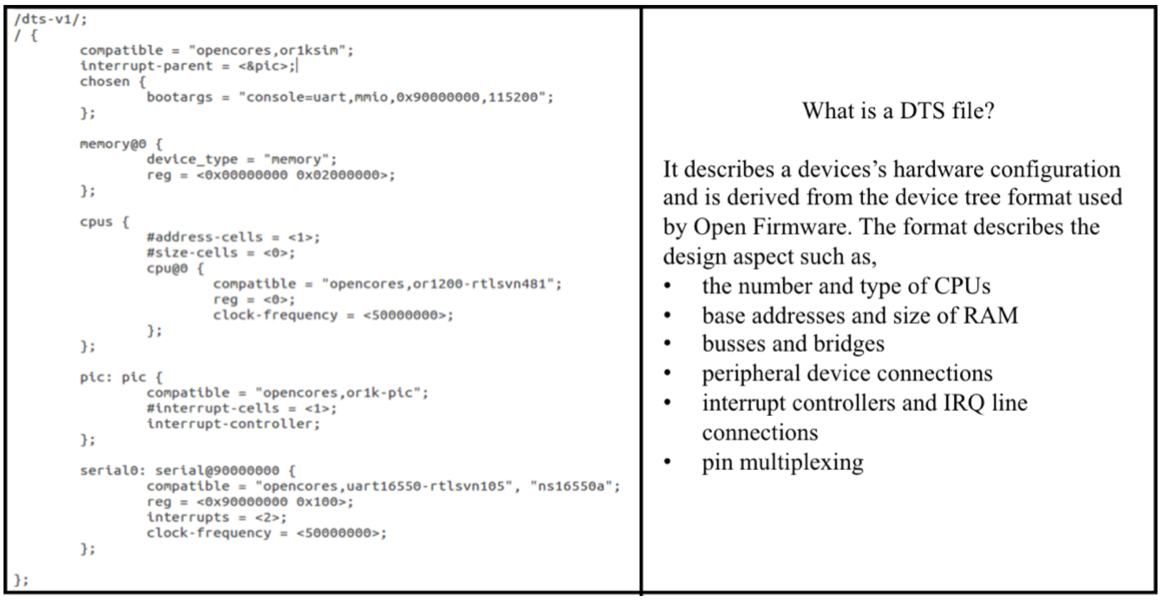


Fig 9.1.2: Integration of an FPU Core

9.2 Porting Linux onto the designed system

The process to compile Linux Kernel involves three important steps: The first step is to port Linux Kernel for OR1200 Architecture and define the cross compiler toolchain “or1k-linux-(gcc, ld, gdb, sim)” used for compilation. The second step is to write the Device Tree description (.dts file) as shown in Fig 8.2.1 for the designed hardware system specifying all the IP cores (UART, Processor, Memory, GPIO, etc.,). The third step is to list the required OS features in a configuration file or use menuconfig or default configuration file instead. Once these steps are completed, compile Linux and generate the binary image. After building the GNU toolchain and Linux kernel image, verify if the compiled kernel has been correctly ported and is booting on the designed system.



```
/dts-v1/;
/ {
        compatible = "opencores,or1ksim";
        interrupt-parent = <&pic>;
        chosen {
                bootargs = "console=uart,mmio,0x90000000,115200";
        };

        memory@0 {
                device_type = "memory";
                reg = <0x00000000 0x02000000>;
        };

        cpus {
                #address-cells = <1>;
                #size-cells = <0>;
                cpu@0 {
                        compatible = "opencores,or1200-rtlsvn481";
                        reg = <0>;
                        clock-frequency = <50000000>;
                };
        };

        pic: pic {
                compatible = "opencores,or1k-pic";
                #interrupt-cells = <1>;
                interrupt-controller;
        };

        serial0: serial@90000000 {
                compatible = "opencores,uart16550-rtlsvn105", "ns16550a";
                reg = <0x90000000 0x100>;
                interrupts = <2>;
                clock-frequency = <50000000>;
        };

};
```

What is a DTS file?

It describes a devices’s hardware configuration and is derived from the device tree format used by Open Firmware. The format describes the design aspect such as,

- the number and type of CPUs
- base addresses and size of RAM
- busses and bridges
- peripheral device connections
- interrupt controllers and IRQ line connections
- pin multiplexing

Fig 9.2.1: DTS File

10. Results

The project has demonstrated an easy approach to design and implement SoCs using open source IP cores. The designed system is extremely scalable and easy to implement. Further, image processing was carried out on the designed hardware and it was determined that the SoC has very good performance/price compared to general-purpose computers.

11. Conclusion

Advances in IoT and mobile computing devices has resulted in billions of devices endowed with processing, memory and communication capabilities. These processing nodes will be, in effect, simple Systems on Chips (SoCs) . These standalone devices have to be inexpensive and be able to operate under stringent performance, power and area constraints. Therefore, design and implementation of efficient SoCs is the way forward to get better performance per dollar and make computing truly ubiquitous.

12. Future Work

There is is wide scope for future work. Some are listed below:

1. Architectural tweaks to enhance performance of the system.
2. Boot an operating system on the SoC.
3. Design accelerators and/or coprocessors to speed up Image Processing.
4. Design/Integrate an SD-Card Controller IP Core into the system to facilitate storage of both source and processed images, hence, providing real-time image processing capability.
5. Build/Develop other applications using the designed system as a backbone.

13. References

[1] Moore, Gordon E. “Cramming more components onto integrated circuits,” Electronics, McGraw Hill, Inc. Vol. 38, No.8 (April 19, 1965).

[2] W. M. Holt, “Moore’s law: A Path Going Forward”, IEEE Proceedings of International Solid-State Circuits Conference, February 2016.

[3] J. M. Shalf and R. Leland, "Computing beyond Moore's Law," in Computer, vol. 48, no. 12, pp. 14-23, Dec. 2015.

[4] A. Waterman, Y. Lee, David A. Patterson, KrsteAsanovi The RISC-V Instruction Set Manual,Vol. I: Base User-Level ISATech. Rep. UCB/EECS-2011-62, EECS Department, UCB, May 2011

[5] OpenRISC 1000 Architecture Manual, Architecture Version 1.0, Document Revision 0, December 5, 2012.

[6] A. López-Parrado and J. C. Valderrama-Cuervo, "OpenRISC-based System-on-Chip for digital signal processing," 2014 XIX Symposium on Image, Signal Processing and Artificial Vision, Armenia, 2014, pp. 1-5.

[7] L. Akçay, M. Tükeland S. B. Örs, "Implementation of an OpenRISC based SoCand Linux Kernel installation on FPGA," 2016 24th Signal Processing and Communication Application Conference (SIU), Zonguldak, 2016, pp. 1969-1972.

[8] F. Abidand N. Izeboudjen, "ASIC implementation of an OpenRISC based SoCfor VoIP application," 2015 6th International Conference on Information and Communication Systems (ICICS), Amman, 2015, pp. 64-67.

[9] L. Akcay, M. Tukeland B. Ors, "Design and implementation of an OpenRISC system-on-chip with an encryption peripheral," 2017 European Conference on Circuit Theory and Design (ECCTD), Catania, 2017, pp. 1-4.

[10] M. Bakiri, S. Titri, N. Izeboudjen, F. Abid, F. Louizand D. Lazib, "Embedded system with Linux Kernel based on OpenRISC 1200-V3“, 2012 6th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT), Sousse, 2012, pp. 177-182.

[11] L. Bara, O. Boncaloand M. Marcu, "Hardware support for performance measurements and energy estimation of OpenRISC processor," 2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics, Timisoara, 2015, pp. 399-404.

[12] A. Jacoby, D. Llamocca, R. Jordan and G. A. Vera, "Proteus: An open source dynamically reconfigurable system-on-chip with applications to digital signal processing," 2014 International Caribbean Conference on Devices, Circuits and Systems (ICCDCS), Playa del Carmen, 2014, pp. 1-6.

[13] F. Abidand N. Izeboudjen, "Technology-independent approach for FPGA and ASIC implementations," 2015 4th International Conference on Electrical Engineering (ICEE),Boumerdes, 2015, pp. 1-4.

[14] C. He, A. Papakonstantinouand D. Chen, "A novel SoC architecture on FPGA for ultra fast face detection," 2009 IEEE International Conference on Computer Design, Lake Tahoe, CA, 2009, pp. 412-418.

[15] A. López-Parradoand J. Velasco-Medina, "SoC-FPGA implementation of the sparse fast fourier transform algorithm," 2017 IEEE 60th International Midwest Symposium on Circuits and Systems (MWSCAS), Boston, MA, 2017, pp. 120-123.