System on Chip Design

Jil Zerndt HS 2025
Lecture Content

Introduction

Course Information

Course: System on Chip Design (SCD)
Institution: Zürcher Hochschule für Angewandte Wissenschaften (ZHAW)
Instructors:
- Tobias Welti (welo@zhaw.ch, +41 58 934 67 30)
- Dominique Cachin (cacd@zhaw.ch, +41 79 45559 01)

Course Materials and Schedule
- **Platform:** Moodle - https://moodle.zhaw.ch/course/view.php?id=25948
- **Script:** Available on Moodle (scd_script.pdf)
- **Lab Instructions:** https://github.zhaw.ch/pages/hpmm/scd-labs/index.html

Assessment and Grading

Grading Components
- **Electronic Quiz:** 15% (November 11, 2025, Moodle test)
- **Lab Exercises:** 15% (6 labs during semester, graded by lecturer)
- **Written Exam:** 70% (January 2026, Moodle test)

Lab Grading System
Seven labs (four lessons each) contribute to the lab grade.
Credits per Lab:
- Not done: 0 points
- Required tasks done with small errors: 1 point
- Required tasks done without errors: 2 points

Lab Grade Formula:

$$\text{Lab Grade} = \frac{\text{Sum of Points}}{12} \times 5 + 1$$

Exam Guidelines:
- Open book: lecture and lab notes, personal notes, books allowed
- No generative AI such as ChatGPT
- Calculators allowed

Course Objectives

Target Audience
This course is designed for engineers who want to:
- Design high-performance digital circuits with SoC-FPGAs, beyond writing VHDL code
- Gain in-depth background knowledge of SoC and FPGA (for software engineers)
- Design systems with Linux on SoC-FPGA
- Obtain introduction and basic knowledge of Integrated Circuit design
- Design general high-speed digital systems with complex peripherals (DDRAM)

Learning Goals
By the end of this course, students will be able to:
- Work with FPGA block memory
- Configure a FPGA-SoC with ARM hardcore processor
- Configure the I/O and computer peripherals (DRAM) of a FPGA
- Port Yocto Linux to SoC-FPGA
- Configure and analyse timing to drive synthesis
- Configure clock generators in FPGA and route clocks on PCB
- Check signal integrity of clock and data lines on PCB
- Explain differences between different signaling standards
- Connect high-speed FPGA peripherals with differential signals
- Realize a project with video and audio output (Pacman game)

Lab Setup

Laboratory Environment
Location: Lab TE 519
Equipment:
- Lab PCs with required software setup running on Linux
- DE1-SoC Development Board with Intel Cyclone V SoC FPGA
- Hardware only available in lab (not distributed to students)

Work Organization:
- Students work in teams of two
- Lab instructions available on GitHub

Introduction to System on Chip Design

Fundamentals of Sequential Digital Circuits

General Representation of Sequential Circuits

Sequential circuits consist of two main components:
- **D-Flip-Flops:** Storage elements that hold state
- **Combinatorial Logic:** Logic circuits that compute next state and outputs
- **Common Clock:** Synchronizes all flip-flops

FPGA Architecture

SRAM-based FPGAs

SRAM-based FPGAs use **Lookup Tables (LUTs)** to generate logic functions.
Key Concept: A LUT is essentially a small RAM (e.g., 4x1 RAM) that can implement any Boolean function of its inputs.

Logic Cells (ALM)

ALM: Adaptive Logic Module (Altera/Intel terminology)
Different FPGA manufacturers use different names:
- **Altera (Intel):** Adaptive Logic Module (ALM)
- **Xilinx:** Configurable Logic Block (CLB) or Slice

Each logic cell typically contains:
- Lookup tables (LUTs)
- Flip-flops
- Multiplexers
- Carry logic

Classic FPGA Architecture

A traditional FPGA consists of:
- **FPGA Fabric:** Array of configurable logic cells
- **Interconnection Network:** Programmable routing between logic cells
- **Logic Cells:** Containing lookup tables and flip-flops

Additional FPGA Resources

Modern FPGA fabric may also contain:
- **SRAM Blocks:** Embedded memory blocks for data storage
- **DSP Blocks:** Dedicated hardware for digital signal processing
- **Hard IP Blocks:** Pre-designed functional blocks (e.g., PCIe, memory controllers)

FPGA Applications

Home Electronics: FPGAs are used in consumer devices for video processing, display controllers, and smart home automation.

Medical Diagnostics: FPGAs provide real-time signal processing for medical imaging equipment and diagnostic devices.

Industrial Automation: FPGAs enable precise control and monitoring in industrial systems.

Studio Equipment: Professional audio and video equipment uses FPGAs for real-time processing and effects.

Products where FPGAs are NOT ideal:
- Battery-operated mobile devices (high power consumption)
- Very high-volume products like mobile phones (cost per unit too high)
- Trivial control applications (washing machine, bike computer - overkill)
- Automotive applications (reliability and certification requirements)
- Some aerospace applications (radiation hardness requirements)

Evolution from FPGA to SoC

System on Chip (SoC) FPGA

Modern SoC FPGAs combine:
- **Hard Processor System (HPS):** Fixed CPU subsystem
- **FPGA Fabric:** Programmable logic
- **Bridges:** Connections between processor and FPGA

This integration enables software and hardware co-design on a single chip.

SoC FPGA Components

CPU Portion (Hard Processor System):
- Cortex-A9 CPU Subsystem
- Flash Controllers
- SDRAM Controller Subsystem
- On-Chip Memories
- Support Peripherals
- PLLs (Phase-Locked Loops)
- Debug Interface
- Peripherals Control Block

FPGA Portion:
- User I/O
- FPGA Fabric (LUTs, RAMs, Multipliers & Routing)
- HSSI Transceivers (High-Speed Serial Interface)
- PLLs
- Hard PCIe
- Hard Memory Controllers

Processor-FPGA Bridges: Enable communication between CPU and FPGA fabric

Topics Covered in SCD Course

Course Topics Overview

The following topics will be discussed throughout the course:
1. Bringing Linux to SoC
2. FPGA System Builder (Platform Designer)
3. Bootloader for Linux
4. RAM and FPGA on-chip Memory
5. DRAM (Dynamic Random Access Memory)
6. JTAG (Joint Test Action Group)
7. Timing Analysis and Synthesis
8. Clock Distribution and PLL
9. High-Speed I/O Interfaces
10. FPGA Configuration

Bringing Linux to SoC-FPGA

Linux on SoC-FPGA

Running Linux on SoC-FPGA enables:

- High-level software development using familiar tools
- Access to vast ecosystem of Linux software
- Network connectivity and modern protocols
- File systems and standard I/O
- Dynamic hardware reconfiguration from software

FPGA System Builder

Platform Designer

Platform Designer (formerly known as Qsys) is Intel's system integration tool for SoC FPGAs.

Functions:

- Graphical system design interface
- Integration of IP cores
- Automatic interconnect generation
- System-level simulation
- Export to Quartus for compilation

Bootloader for Linux

Boot Process

The bootloader is responsible for:

- Initializing hardware (CPU, memory, peripherals)
- Loading the Linux kernel into memory
- Passing boot parameters to the kernel
- Transferring control to the operating system

Common bootloaders: U-Boot, GRUB

Memory Systems

RAM and FPGA On-Chip Memory

On-Chip Memory Types:

- **Block RAM (BRAM):** Embedded memory blocks in FPGA fabric
- **Distributed RAM:** Memory implemented using LUTs
- **On-Chip RAM in HPS:** Fast memory integrated in processor system

Characteristics:

- Very fast access (single cycle or few cycles)
- Limited size (kilobytes to few megabytes)
- No refresh needed (SRAM-based)
- Higher cost per bit than external memory

Dynamic RAM (DRAM)

Inventor: Robert H. Dennard (invented at IBM in 1967)

Key Characteristics:

- **Storage Mechanism:** Stores data as charge in capacitors
- **Refresh Required:** Must be periodically refreshed (capacitors leak)
- **High Density:** Much higher storage density than SRAM
- **Lower Cost:** Cheaper per bit than SRAM
- **Slower Access:** Requires complex timing protocols

Common Types:

- DDR3, DDR4, DDR5: Double Data Rate SDRAM
- LPDDR: Low Power DDR for mobile devices

JTAG (Joint Test Action Group)

JTAG Interface

JTAG is a standard for testing and debugging electronic systems.

Primary Uses:

- Boundary scan testing of PCBs
- Programming FPGAs and flash memory
- Debugging processors (via debug interface)
- In-system programming

Signal Lines:

- **TDI:** Test Data In
- **TDO:** Test Data Out
- **TCK:** Test Clock
- **TMS:** Test Mode Select
- **TRST:** Test Reset (optional)

JTAG State Machine

The JTAG interface operates through a state machine with the following main states:

- **Test-Logic-Reset:** Initial state
- **Run-Test/Idle:** Idle state
- **Select-DR-Scan:** Select data register scan
- **Select-IR-Scan:** Select instruction register scan
- **Capture-DR/IR:** Capture data or instruction
- **Shift-DR/IR:** Shift data or instruction
- **Update-DR/IR:** Update data or instruction register

Timing Analysis and Synthesis

Static Timing Analysis

Static timing analysis verifies that a digital circuit meets all timing constraints without requiring simulation of the circuit's operation.

Key Concepts:

- **Setup Time (t_{su}):** Minimum time data must be stable before clock edge
- **Hold Time (t_h):** Minimum time data must remain stable after clock edge
- **Clock-to-Q Delay (t_{co}):** Time for data to appear at flip-flop output after clock
- **Propagation Delay:** Time for signals to travel through logic and routing

Timing Paths and Slack

Data Arrival Time is composed of:

1. Clock network delay from clock node to launch D-FF: 3.00 ns
2. Clock-to-Q delay from launch D-FF: 2.00 ns
3. Logic delay (cells, I/O pins, routing): 5.00 ns

Total Data Arrival Time: 10.00 ns

Data Required Time is determined by:

1. Clock period: 40.00 ns
2. Clock network delay to latch D-FF: 2.00 ns
3. Setup time of latch D-FF: -0.50 ns

Total Required Time: 41.50 ns

Setup Slack = Required Time - Arrival Time = 41.50 ns - 10.00 ns = 31.50 ns

Positive slack indicates timing is met. Negative slack indicates timing violation.

Clock Distribution and PLL

Phase-Locked Loop (PLL)

A **PLL** is a control system that generates an output signal whose phase is related to the phase of an input signal.

Key Functions:

- **Frequency Multiplication:** Generate higher frequency clocks
- **Frequency Division:** Generate lower frequency clocks
- **Phase Alignment:** Align clock phases
- **Jitter Reduction:** Clean up noisy clock signals
- **Clock Deskew:** Compensate for clock distribution delays

Clock Distribution Network

FPGAs have dedicated clock distribution networks:

- **Global Clock Networks (GCLK):** Low-skew distribution across entire device
- **Regional Clock Networks:** Lower skew within regions
- **Dual-Purpose Clock Networks (DPCLK):** Flexible routing
- **Clock Control Blocks (CDPCLK):** Gating and multiplexing

Proper clock distribution is critical for achieving timing closure in high-speed designs.

High-Speed I/O Interfaces

Differential Signaling

High-speed interfaces use **differential signaling** where information is transmitted using two complementary signals.

Advantages:

- Better noise immunity (common-mode noise rejection)
- Higher data rates possible
- Lower EMI emissions
- Better signal integrity over long distances

Common Standards:

- LVDS (Low-Voltage Differential Signaling)
- TMDS (Transition-Minimized Differential Signaling)
- SerDes (Serializer/Deserializer)

FPGA Configuration

FPGA Configuration Process

FPGAs are volatile devices that must be configured at power-up.

Configuration Sources:

- **Configuration Flash:** Non-volatile memory storing bitstream
- **JTAG:** Programming via debugger (Byte Blaster)
- **Serial Configuration:** Via SPI or similar interface
- **Parallel Configuration:** Faster configuration method

Configuration Modes:

- **Active Serial (AS):** FPGA controls configuration process
- **Passive Serial (PS):** External controller drives configuration
- **JTAG Mode:** For development and debugging

Configuration Pins:

- **nCE:** Active-low chip enable
- **nCONFIG:** Configuration control
- **CONF_DONE:** Configuration complete signal
- **DATA:** Configuration data lines

Typical pull-up/pull-down resistors: 10kΩ

FPGA Block Memory

Introduction to FPGA Block Memory

FPGA Block Memory

FPGA block memory consists of small blocks of random access memory integrated into the FPGA fabric.

Key Characteristics:

- Used instead of thousands of flip-flops for data storage
- Multiple applications in digital design
- Dual-port capability for simultaneous access
- Configurable data widths and addressing modes

FPGA fabric contains not only logic cells but also dedicated SRAM blocks and DSP blocks for efficient implementation of memory and arithmetic operations.

Learning Objectives

Course Learning Goals

After this lecture and lab exercise, students will be able to:

- Understand the architecture of FPGA block memory
- Explain the difference between registered and unregistered output from a memory block
- Understand the consequences of either configuration choice
- Sketch and interpret timing diagrams of memory accesses
- List at least four applications of FPGA block memory

Applications of FPGA Block Memory

Common Applications:

- **Look-Up Tables (LUT):** e.g., Synthesizer sine wave generation
- **ROM/RAM/Cache:** In SoC processors or softcore processors
- **Configuration/Status Register Bank:**
  - Port A: Processor access
  - Port B: Logic access
- **Cache for Flash Memory:** Speed up access to slow external flash
- **FIFOs:**
  - Receive/transmit buffers
  - Clock crossing bridges

Direct Digital Synthesis (DDS)

LUT in Direct Digital Synthesis

In DDS, a lookup table stores pre-calculated values of a sine wave period.

Process:

1. Phase accumulator generates address (phase index)
2. LUT outputs corresponding sine value
3. Low-pass filter smooths the output
4. Result: Clean sinusoidal waveform

Advantages:

- Fast generation of complex waveforms
- Precise frequency control
- Low resource usage

FIFO Buffers

FIFO (First-In-First-Out) Buffer

A FIFO is a data structure where the first element added is the first one to be removed.

Key Operations:

- **Enqueue:** Add element to back of queue
- **Dequeue:** Remove element from front of queue

Use Cases:

- Buffering data between different clock domains
- Rate matching between fast and slow interfaces
- Packet buffering in communication systems

Clock Crossing Bridge

Using Block Memory for Clock Domain Crossing

Application: Decoupling clock frequencies in audio systems

Example Configuration:

- Write Clock (WR_CLK): 12.288 MHz (I2S audio clock)
- Read Clock (RD_CLK): 50 MHz (system clock)
- FIFO depth: 256 words
- Serial audio data input via I2S protocol

Benefits:

- Safely transfers data between clock domains
- Prevents metastability issues
- Allows for rate buffering
- Provides elastic buffer for timing variations

Cache for Flash Memory

Memory Hierarchy

Memory systems follow a general principle: **larger = slower = cheaper**

Hierarchy (fast to slow):

1. Registers (fastest, smallest, most expensive per bit)
2. Cache (FPGA Block RAM)
3. Main Memory (external DRAM)
4. Flash Memory (non-volatile, slowest)

Block RAM can serve as cache to speed up access to slow external flash memory.

Character-Based Graphics Controller

Video Controller with Character Memory

System Overview:

- **Character Set:** 64 unique characters
- **Screen Buffer:** 3072 character positions
- Uses block RAM to store character and color data

Architecture Components:

- **Character RAM:** Stores which character at each position
- **Color RAM:** Stores color attributes for each position
- **Graphic RAM:** Stores pixel patterns for characters
- **Video Counter:** Generates timing signals
- **Color Multiplexer:** Selects output colors

SRAM Cell Architecture

Volatile CMOS Memory Cell

SRAM (Static Random Access Memory) uses a cross-coupled inverter pair to store one bit.

Components:

- **Two Cross-Coupled Inverters:** Form a bistable latch
- **Two Access Transistors:** Connect cell to bit lines
- **Word Line:** Activates access transistors for read/write
- **Bit Lines (true and complement):** Transfer data in/out

Characteristics:

- Volatile: Data lost when power removed
- Static: No refresh needed (unlike DRAM)
- Fast: Single-cycle access possible
- Large: 6 transistors per bit

Organization of Memory Cells

In a discrete SRAM chip, memory cells are organized in a matrix:

- **Word Lines:** Select rows of cells
- **Bit Lines:** Transfer data from/to selected cells
- **Address Decoder:** Activates appropriate word line
- **Sense Amplifiers:** Detect small voltage changes during read
- **Write Drivers:** Drive bit lines during write operations
- **RD/WR# Control:** Selects read or write operation

FPGA RAM Block Architecture

Cyclone V Memory Blocks

Cyclone V FPGAs contain two types of RAM blocks:

- **M10K Block:** 10 Kbit capacity
  - Larger memory blocks for bulk storage
  - Can be configured in various width/depth combinations
- **MLAB (Memory Logic Array Block):** 640 bit capacity
  - Smaller, distributed memory
  - Can be used as small FIFOs or shift registers

Key Feature: Address decoder and control logic are integrated, enabling fast clock speeds.

Multiport RAM Architecture

FPGA RAM blocks feature dual-port architecture:

Characteristics:

- **Two Sides:** Port A and Port B
- **Independent Clocks:** Each port can have different clock
- **Different Data Widths:** Port A and B can have different widths
- **Mixed-Port Data Flow:** Write on one port, read on other
- **Same-Port Data Flow:** Read and write on same port

Example Application:

- Port A: HPS (Hard Processor System) access
- Port B: FPGA logic access
- Result: Clock-independent shared memory between CPU and FPGA

Memory Timing

SRAM Read Cycle

SRAM Read Timing Parameters

Key Timing Parameters:

- t_{RC} **(Read Cycle Time):** Minimum time the address must be applied (typical: 10 ns)
- t_{AA} **(Address Access Time):** Time from address stable until data valid (typical: 10 ns)
- t_{OHA} **(Output Hold Time):** Time data remains valid after address change (typical: 2 ns)

Read Sequence:

1. Apply stable address
2. Wait for t_{AA}
3. Data becomes valid at output
4. Data remains valid for t_{OHA} after address changes

SRAM Write Cycle

SRAM Write Timing Parameters

Key Timing Parameters:

- t_{WC} **(Write Cycle):** Minimum time an address must be valid
- t_{SA} **(Address Setup Time):** Minimum time address must be applied before WE active (typical: 0 ns)
- t_{PWE} **(Write Pulse Width):** Minimal width of write pulse (typical: 8 ns)
- t_{HA} **(Address Hold):** Minimum time address may go unstable after rising edge of WE (typical: 0 ns)
- t_{SD} **(Data Setup Time):** Minimum time data must be stable before rising edge of WE (typical: 7 ns)
- t_{HD} **(Data Hold Time):** Minimum time data must remain stable after rising edge of WE (typical: 0 ns)

Write Sequence:

1. Apply address (t_{SA} before WE)
2. Apply data (t_{SD} before WE rising edge)
3. Assert WE (write enable) for at least t_{PWE}
4. Hold address for t_{HA} after WE
5. Hold data for t_{HD} after WE

FPGA Memory Read and Write Operations

Write Operation

FPGA Memory Write Timing

Writing to FPGA block memory is synchronous (clock-driven).

Signals:

- **CLK:** Clock signal
- **WREN:** Write enable (active high)
- **ADDRESS:** Memory address
- **DATA_IN:** Data to be written

Write Process:

1. On rising edge of CLK:
   - If WREN $= 1$: DATA_IN is written to ADDRESS
   - If WREN $= 0$: No write occurs
2. Data is stored in memory cell
3. Write completes in one clock cycle

Sequential Writing: Writing to consecutive memory addresses:

- Clock cycle 1: Write to address a0
- Clock cycle 2: Write to address a1
- Clock cycle 3: Write to address a2
- Each write occurs on the rising edge of CLK when WREN is high

Read Operation - Asynchronous Mode

Asynchronous Read Mode

In asynchronous (unregistered) read mode, data appears at the output after internal memory access delay.

Characteristics:

- **No output register:** Data comes directly from memory array
- **Faster initial access:** Data available within same clock cycle
- **Combinatorial path:** Creates longer critical path
- **Lower f_{max}:** Reduces maximum achievable clock frequency

Read Enable: Optional signal (rden) can gate the output

Read Operation - Synchronous Mode

Synchronous Read Mode

In synchronous (registered) read mode, data is latched in an output register.

Characteristics:

- **Output register:** Data latched on clock edge
- **One cycle latency:** Data available one clock cycle after address
- **Registered path:** Breaks combinatorial path
- **Higher f_{max}:** Enables higher clock frequencies

Timing:

- Clock cycle n: Address applied
- Clock cycle $n+1$: Data available at output

Comparison of Read Modes

Asynchronous vs Synchronous Read

Path Delay Analysis:

Asynchronous Mode:

$$\text{Path delay} = t_{clk\to q} + t_{mem} + t_{rout} + t_{su} = 0.5 + 2.5 + 1.5 + 0.5 = 5.0 \text{ ns}$$

$$f_{max} = 200 \text{ MHz}$$

Synchronous Mode:

- Memory to register: $t_{mem} = 2.5$ ns
- Register to logic cell: Standard registered path
- $f_{max} = 286$ MHz (from memory)
- $f_{max} = 400$ MHz (M10K block capability)

Synchronous mode enables higher clock frequencies by breaking the combinatorial path, at the cost of one cycle latency.

Trade-offs:

Asynchronous Mode (no output FF):

+ Data available after one clock cycle

- Combinatorial path is longer (slower clock possible)

Synchronous Mode (with output FF):

+ Higher maximum clock frequency

+ Better timing closure

- One additional cycle latency

Performance Specifications

Speed of Embedded Memory in Cyclone V

Performance by Configuration:

Memory Mode	ALUTs	Memory	Speed	Unit
Asynchronous Read (No Output Register)				
Single port	0	1	450/380/330	MHz
Simple dual-port	0	1	450/380/330	MHz
ROM	0	1	450/380/330	MHz
Synchronous Read (With Output Register)				
Single port	0	1	315/275/240	MHz
Simple dual-port	0	1	315/275/240	MHz
Simple dual-port (old data)	0	1	275/240/180	MHz
True dual-port	0	1	315/275/240	MHz
ROM	0	1	315/275/240	MHz
Clock Timing Constraints				
Min pulse width (high)	-	-	1450/1550/1650	ps
Min pulse width (low)	-	-	1000/1200/1350	ps

Notes:

- Speed grades shown as: C6 / C7 / C8 (fastest to slowest)
- To achieve maximum performance, use global clock routing from on-chip PLL
- Set PLL to 50% output duty cycle
- No f_{max} degradation when using CRC error detection

Memory Block Port Configuration

Dual-Port Memory Interface

FPGA memory blocks provide two independent ports (A and B) with the following signals:

Port A Signals:

- **DATA_IN_A:** Input data
- **ADRESSEN_A:** Address bus
- **WREN_A:** Write enable
- **RDEN_A:** Read enable (optional)
- **BYTEEN_A:** Byte enable for partial writes
- **DATA_OUT_A:** Output data

Port B Signals:

- **DATA_IN_B:** Input data
- **ADRESSEN_B:** Address bus
- **WREN_B:** Write enable
- **RDEN_B:** Read enable (optional)
- **BYTEEN_B:** Byte enable for partial writes
- **DATA_OUT_B:** Output data

Both ports can operate independently with different clocks, addresses, and data widths.

Memory Resources Summary

Memory Resources in Cyclone V

Cyclone V Device Family Resources:

Resource	5CEA2	5CEA4	5CEA5	5CEA9
Logic Elements	25K	49K	85K	150K
M10K Blocks	125	200	300	500
Memory (Kbit)	1,250	2,000	3,000	5,000
Variable-precision DSP 18×19 Multiplier	36	84	144	234
Hard Floating-point	Yes	Yes	Yes	Yes
Fractional Synthesis	2	4	4	6
12.5 Gbps Transceiver	-	-	4	6

Key Points:

- Larger devices have proportionally more memory blocks
- M10K blocks provide flexible memory implementation
- DSP blocks complement memory for signal processing
- High-speed transceivers available in larger devices

FPGA Memory Configurations and Modes

Learning Objectives

Course Goals

Students will be able to:

- List and identify several modes of operation of block memories in FPGA
- Explain how to use block memories for clock domain crossing
- Calculate signal widths for memories with different data bus widths
- Understand how a block memory can be defined in VHDL code
- Configure a block memory using the toolchain

Recap: FPGA Memory Basics

Key Topics from Previous Lecture:

- Applications of FPGA memory
- Architecture of SRAM cells
- Memory cells organization
- Timing diagrams of read/write accesses
- Registered vs unregistered output modes

Memory Block Configurations

4K-Bit Memory Block Configurations

Hard macros (silicon) of RAM blocks support various word sizes from 1 to 32 bits.

Key Characteristics:

- Toolchain combines RAM blocks to build desired word size and memory capacity
- Even larger than 32 bits possible through block combination
- Maximum performance: 250 MHz
- Total RAM bits per M4K block (including parity): 4,608 bits

Supported Configurations:

- 4K $\times$ 1 bit
- 2K $\times$ 2 bits
- 1K $\times$ 4 bits
- 512 $\times$ 8 bits
- 512 $\times$ 9 bits (with parity)
- 256 $\times$ 16 bits
- 256 $\times$ 18 bits (with parity)
- 128 $\times$ 32 bits (not available in true dual-port mode)
- 128 $\times$ 36 bits (not available in true dual-port mode, with parity)

Memory Operation Modes

ROM Mode

Read-Only Memory (ROM) in FPGA

ROM blocks provide non-volatile storage initialized at configuration time.

Characteristics:

- Memory contents loaded from external HEX file to configuration file
- Contents programmed during FPGA configuration
- Memory cannot be modified at runtime
- Block type: AUTO (toolchain selects optimal implementation)

Interface:

- **address:** Read address input
- **read_data:** Data output
- **clock:** Synchronous clock (for registered output)

Intel HEX Format

Intel HEX is a file format for representing binary data in ASCII text format.

Record Structure:

:LLAAAATTDD...DDCC

Where:

- **:** Start code (1 character)
- **LL:** Byte count (2 hex digits) - number of data bytes
- **AAAA:** Address (4 hex digits) - 16-bit address
- **TT:** Record type (2 hex digits):
  - 00 = Data record
  - 01 = End of file record
  - 02 = Extended segment address
  - 04 = Extended linear address
- **DD...DD:** Data bytes (2n hex digits)
- **CC:** Checksum (2 hex digits)

Example Records:

```
:040000100FFFF0000C   (Address 0x0000: 0xFF, 0xFF, 0x00, 0x00)
:0400020000FFFF002F   (Address 0x0002: 0x00, 0xFF, 0xFF, 0x00)
```

Single Port RAM Mode

Single Port RAM

Single port RAM provides one port for both read and write operations.

Interface Signals:

- **write_data:** Input data bus
- **read_data:** Output data bus
- **write_enable:** Write control signal
- **address:** Memory address
- **clock:** Synchronous clock

Features:

- Memory contents may be preset using a HEX file
- Contents initialized during configuration
- Random access during operation
- Read and write through same port (not simultaneous)

Example Configuration:

- Data width: 8 bits
- Address width: 5 bits
- Capacity: $2^5 = 32$ words

Simple Dual Port Mode

Simple Dual Port RAM

Simple dual port mode provides separate ports for reading and writing.

Port Characteristics:

- **Port A:** Write-only
- **Port B:** Read-only

Interface Signals:

- **Write_data:** Input data for writing
- **Write_address:** Address for write operations
- **Write_enable:** Write control signal
- **Read_data:** Output data from reading
- **Read_address:** Address for read operations
- **clock:** Single clock for both ports

Important Limitations:

- Port B does not know if there is any data written
- Port A does not know if the data has been read
- No handshaking mechanism between ports

Configuration Options:

- Simple Dual Port
- Synchronous inputs/outputs
- One clock or dual clock operation

True Dual Port Mode

True Dual Port RAM

True dual port mode allows both ports to read and write independently.

Port Capabilities:

- **Both Port A and Port B:** Can read AND write
- Independent operation on both ports
- Simultaneous access possible

Interface Signals (per port):

- **address:** Memory address
- **write_data:** Input data for writing
- **write_enable:** Write control signal
- **read_data:** Output data from reading
- **clock:** Synchronous clock (can be different for each port)

Configuration:

- True Dual Port mode
- Synchronous inputs/outputs
- One clock or dual clock operation
- Capacity: e.g., 32 words RAM

Potential Issues in True Dual Port Mode:

What could go wrong?

- **Write Conflict:** Both ports writing to same address simultaneously
- **Read-During-Write:** Port B reading from address Port A is writing to
- **No Port Coordination:** No port knows what the other is doing
- **No Data Validation:** No checking mechanisms to ensure data being read has been written

Careful design and handshaking protocols are required to avoid data corruption in true dual port configurations.

Triple Port Mode

Triple Port Memory

Triple port memory provides one write port and two independent read ports.

Port Configuration:

- **Write Port:** Single write address and data input
- **Read Port A:** Independent read address and data output
- **Read Port B:** Independent read address and data output

Implementation: The synthesis tool implements triple port memory using two dual-port RAM blocks:

- Both blocks receive same write data
- Each read port connects to separate RAM block
- Ensures data consistency across read ports

Synthesis Result:

- `altdpram:altdpram_component1` - First dual-port RAM
- `altdpram:altdpram_component2` - Second dual-port RAM
- Shared write interface
- Independent read interfaces

Different Bus Widths

Utilizing Different Bus Widths

FPGA memory blocks support different data widths on read and write ports.
Key Principle: Both sides must cover the full memory space:

$$\text{Number of words} \times \text{Word size} = \text{Capacity}$$

Relationship:
- **Larger word size** $\Rightarrow$ Smaller address width
- **Smaller word size** $\Rightarrow$ Larger address width

Example Configuration:
- **Write Port:** 8-bit data width
- **Read Port:** 32-bit data width
- Ratio: 1:4 (read width is 4$\times$ write width)
- Read address is 2 bits narrower than write address

Use Case: Useful when FPGA logic needs word size larger than bus width of processor.

Exercise: Different Word Widths

Given: Simple dual-port memory with different input and output word widths.
Task: Calculate the width of both read and write address signals.
Specifications:
- **Capacity:** 16,384 bits = 16 Kbit
- **Write data width:** 16 bits
- **Read data width:** 256 bits

Solution:
Write Side:

$$\text{Number of words} = \frac{16,384 \text{ bits}}{16 \text{ bits/word}} = 1,024 \text{ words}$$

$$\text{Address bits} = \log_2(1,024) = 10 \text{ bits}$$

Read Side:

$$\text{Number of words} = \frac{16,384 \text{ bits}}{256 \text{ bits/word}} = 64 \text{ words}$$

$$\text{Address bits} = \log_2(64) = 6 \text{ bits}$$

Answer:
- Write address: `address[9..0]` (10 bits)
- Read address: `rdaddress[5..0]` (6 bits)

Clock Domain Crossing

Separated Clock Domains

Memory blocks can operate with different clock domains on input and output.
Configuration Options:
Option 1: Input/Output Clock Separation
- Inputs controlled by `inclock`
- Outputs controlled by `outclock`
- Data written in one domain, consumed in another
- Both clocks can be related or independent

Option 2: Write/Read Clock Separation
- Writing controlled by `wrclock`
- Reading controlled by `rdclock`
- Completely independent writer and reader
- Asynchronous operation between domains

Clock domain crossing requires careful design to avoid metastability and data corruption.

Clock Crossing Bridge Using FIFO

Application: Audio data buffering with clock domain crossing
Configuration:
- **Write Clock (WR_CLK):** 12.288 MHz (I2S audio clock)
- **Read Clock (RD_CLK):** 50 MHz (system clock)
- **FIFO Depth:** 256 words
- **Write Enable:** Controlled by I2S protocol
- **Data Path:** Serial audio data input via I2S

Operation:
- Input on fast clock, but sparse writing events (every 256 cycles)
- Output on slow clock, consuming data every cycle
- Control logic provided by FIFO soft IP block or custom developer logic
- Some delay introduced by FIFO (typically a few clock cycles)

Benefits:
- Safe data transfer between clock domains
- Rate buffering and flow control
- Prevents data loss due to clock frequency mismatch

Byte Enable Feature

Byte Enable

Byte enable allows writing individual bytes of wider data inputs.
Signal: `byteena[3..0]` for 32-bit data
Functionality:
- Each bit in `byteena` controls one byte of data
- `byteena[0] = '1'` enables writing of data[7..0]
- `byteena[1] = '1'` enables writing of data[15..8]
- `byteena[2] = '1'` enables writing of data[23..16]
- `byteena[3] = '1'` enables writing of data[31..24]

Use Cases:
- Partial word updates
- Byte-addressable memory emulation
- Efficient memory usage in mixed-width systems

Alternative: Multiple write operations with address manipulation

VHDL Implementation

Why Use VHDL for RAM Blocks?

Advantages of describing RAM blocks in VHDL:

- **Platform Independence:** Code portable across FPGA families
- **Easier Customization:** Modify parameters without GUI
- **Reusability:** Parameterized designs for multiple uses
- **Scriptable:** Automate generation and configuration
- **Simulation:** Easier to simulate and verify behavior

VHDL Array Type

Arrays in VHDL allow defining memory structures.

Syntax:

```vhdl
type <type_name> is array (<low> to <high>) of <data_type>;
```

Example:

```vhdl
type 64b_ram is array(0 to 7) of std_logic_vector(7 downto 0);
```

This defines an array of 8 elements, each 8 bits wide, creating a 64-bit RAM structure.

Memory Layout:

- Index 0: 00000000
- Index 1: 00010001
- Index 2: 00100010
- Index 3: 00110011
- Index 4: 01000100
- Index 5: 01010101
- Index 6: 01100110
- Index 7: 01110111

Generic RAM in VHDL

Complete VHDL code for a parameterized RAM block:

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.all;

entity ram_generic is
  generic (
    RAM_WIDTH : natural := 10;  -- Port width
    RAM_DEPTH : natural := 12); -- Address width
  port (
    clock   : in  std_logic;
    address : in  std_logic_vector(RAM_DEPTH-1 downto 0);
    wren    : in  std_logic;
    data    : in  std_logic_vector(RAM_WIDTH-1 downto 0);
    q       : out std_logic_vector(RAM_WIDTH-1 downto 0));
end ram_generic;

architecture rtl of ram_generic is
  -- Define memory type: array of words
  type mem is array(0 to (2**RAM_DEPTH)-1) of
    std_logic_vector(RAM_WIDTH-1 downto 0);
  signal ram_block : mem;
begin
  process(all)
  begin
    if rising_edge(clock) then
      if (wren = '1') then
        -- Write operation
        ram_block(to_integer(unsigned(address))) <= data;
      end if;
      -- Read operation (synchronous)
      q <= ram_block(to_integer(unsigned(address)));
    end if;
  end process;
end rtl;
```

Key Features:

- Generic parameters for width and depth
- Synchronous read and write
- Type conversion: `to_integer(unsigned(address))`
- Infers block RAM in synthesis

Synthesis Result in Cyclone IV:

The VHDL code synthesizes to:

- **SYNC_RAM block:** Inferred synchronous RAM
- **Output registers:** `q[0]~reg[9..0]`
- **Inputs:** clock, data[9..0], address[11..0], wren

The synthesizer recognizes the RAM pattern and maps it to dedicated hardware blocks.

IP Catalog Configuration

Configuring Memory Using IP Catalog

Step 1: Open IP Catalog
- In Quartus, navigate to Tools → IP Catalog
- Select device family: Cyclone V (E/GX/GT/SX/SE/ST)

Step 2: Select Memory Type
- Navigate to: Library → Basic Functions → On Chip Memory
- Choose from available options:
  - FIFO
  - RAM Initializer
  - RAM: 1-PORT
  - RAM: 2-PORT
  - ROM: 1-PORT
  - ROM: 2-PORT
  - Shift register (RAM-based)

Step 3: Configure Memory Parameters See detailed configuration options below.

Data Bus Width and Depth Configuration

Configuring Width and Depth

Configuration Parameters:

Width Configuration:
- Question: "How wide should the 'q' output bus be?"
- Example: 16 bits
- Arbitrary values allowed

Depth Configuration:
- Question: "How many 16-bit words of memory?"
- Example: 65,536 words
- Results in: 96 bits + 128 M10K + 3 registers

Memory Block Type:
- **Auto:** Let toolchain decide optimal implementation
- **MLAB:** Force use of Memory LAB (small distributed RAM)
- **M10K:** Force use of M10K blocks

Maximum Block Depth:
- Set to Auto for optimal packing
- Can specify manual limit if needed

Clocking Method:
- **Single clock:** One clock for all operations
- **Dual clock:** Separate input and output clocks

Port and Enable Configuration

Configuring Ports and Enables

Registered Ports:
- ☐ 'data' and 'wren' input ports
- ☐ 'address' input port
- ☐ 'q' output port (recommended for high performance)

Clock Enable:
- ☐ Create one clock enable signal for each clock
- All registered ports controlled by enable signal

Byte Enable:
- ☐ Create byte enable for port A
- Specify byte width (typically 8 bits)

Asynchronous Clear:
- ☐ Create 'aclr' asynchronous clear for registered ports

Read Enable:
- ☐ Create 'rden' read enable signal (optional)

Registering the 'q' output improves timing performance but adds one cycle latency.

Read-During-Write Configuration

Read-During-Write Behavior

Defines output behavior when reading from a location being written to.

Options:
- **New Data:** Output shows newly written data
- **Old Data:** Output shows data before write
- **Don't Care:** Output is undefined (X in simulation)

Byte Enable Interaction:
- ☐ Get X's for write-masked bytes instead of old data
- Applies when byte enable is used
- Helps identify unintended read-during-write conditions

Read-during-write behavior is crucial for avoiding data hazards in memory access patterns.

Memory Initialization

Initializing Memory Contents

Question: "Do you want to specify the initial content of the memory?"

Options:

Option 1: Blank Memory
- ○ No, leave it blank
- Memory contains undefined values at power-up

Option 2: Simulation-Only Init
- ○ Yes, but leave it blank in simulation
- Initialize for hardware, but not in simulation

Option 3: File-Based Init
- ○ Yes, use this file for memory content data
- File formats:
  - Hexadecimal (Intel-format) File [.hex]
  - Memory Initialization File [.mif]
- Specify filename and path
- Select which port's dimensions to match (Port A or Port B)

Advanced Feature:
- ☐ Use Memory Content Editor
- Capture and update content independently of system clock
- Requires Instance ID assignment

Synthesis and Simulation Support

Simulation Libraries:

To properly simulate the generated design, the following simulation model files are needed:
- **altera_mf:** Altera megafunction simulation library

Timing and Resource Estimation:
- ☐ Generate netlist for timing and resource estimation
- Enables better design optimization in third-party synthesis tools
- Check with tool vendor for support information
- Note: Netlist generation can be time-intensive
- Process time depends on design size and system speed

Not all third-party synthesis tools support netlist-based estimation. Verify compatibility before enabling this feature.

VHDL Testbench and Simulation

Learning Objectives

Course Goals

Students will be able to:

- Name at least four advantages of simulation versus hardware testing
- Describe the operation of a testbench
- Implement a procedure-based testbench for their designs

Testbench Architecture

Testbench Components

A testbench is a VHDL construct that provides automated testing for a design.

Key Components:

- **Device Under Test (DUT):** The design being tested
- **Stimulus Generator:** Creates input signals for DUT
- **Expected Results:** Known correct outputs
- **Comparison Logic:** Compares actual vs expected outputs
- **Infrastructure:** Clock generation, file I/O, reporting

Important Characteristic:

- Testbench entity has **no ports** (no external I/O)
- All signals are internal to simulation

Testbench Entity Structure

```
-- Testbench file: block_top_tb.vhd

ENTITY block_top_tb IS
  -- Entity has no ports since no I/Os
END block_top_tb;

ARCHITECTURE behavior OF block_top_tb IS
  -- Component declarations
  -- Signal declarations
  -- Constants
BEGIN
  -- DUT instantiation
  -- Stimulus generation
  -- Response checking
END behavior;
```

The testbench has no ports because it is the top-level entity in simulation.

Purpose of Simulation

Why Simulate?

Simulation provides significant advantages over hardware-only testing:

Accessibility Benefits:

- Testing and debugging in SoC is difficult:
  - Pins buried under device
  - Internal signals not accessible from outside
  - Many parallel processes occurring simultaneously
- Simulation allows tracking of **every signal at every point in time**
- All internal states visible and traceable

Verification Benefits:

- **Automated verification:** Comparison of output with expected results
- **Regression testing:** Test every block automatically
- **Better test coverage:** Easily provoke edge cases and corner conditions

Development Efficiency:

- **Faster iterations:** DUT can be recompiled in seconds vs synthesized in minutes
- **Reduced risk:** ASIC fabrication takes months and costs \$100,000+ depending on process
- **Lower testing effort:** Simulation can reduce overall testing time
- **Automation:** Can be automated even for large designs

Questasim/ModelSim Tool Flow

VHDL Compilation and Simulation Flow

The simulation process involves two main phases:

Phase 1: Compilation

1. User VHDL files are written (design + testbench)
2. Questasim compiler processes all VHDL files
3. Compiled objects stored in libraries (typically `work`)
4. Standard libraries automatically included

Phase 2: Simulation

1. Simulator loads compiled libraries
2. Testbench procedures generate stimuli
3. DUT responds to stimuli
4. Results captured as waveforms and text output

Procedure-Based Testbench

Why Procedure-Based?

Advantages of using procedures:

- Simple writing and expansion of test cases
- Supports scripting approach
- Easy to maintain and modify
- Input format: `command [arg_0 arg_1 ... arg_n]`

Test Script Structure

Format: `command arg1 arg2 arg3 arg4`

Each argument is one byte (2 hex digits), MSB = arg1, LSB = arg4

Hard Processor System (HPS) in SoC FPGA

Learning Objectives

Course Goals

After this lecture and lab exercise, students will be able to:

- Explain why it is often required to execute software in a SoC
- Explain the difference between a softcore and a hard processor system
- Name advantages for both softcore and hard processor options
- Configure a HPS and/or a softcore processor
- Apply the principle of bus bridges to access components in different chip parts

FPGA Applications and Software Requirements

Typical FPGA Applications

FPGAs excel at tasks requiring deterministic, real-time processing:

Data Processing:

- Filtering and triggering
- Control loops with precise timing
- Real-time signal processing

Video Processing:

- Color space transformation
- Compression and decompression
- Filters and effects
- Real-time video manipulation

Key Advantage: Deterministic latency - guaranteed response times

Tasks Better Solved in Software

Many system functions are more efficiently implemented in software:

Complex Algorithms:

- Video processing filters requiring coefficient calculation over multiple frames
- Complex decision-making algorithms
- Machine learning inference

Networking and Communication:

- Web server with configuration controls
- Network stack for data transfer via internet
- Protocol implementations (TCP/IP, HTTP, etc.)

System Management:

- API for easier control of FPGA application
- User interface and configuration management
- File system and data logging

Combining FPGA hardware acceleration with CPU software flexibility provides optimal system performance.

Evolution from Discrete Systems to SoC

Classic FPGA & CPU Architecture

Traditional systems use separate FPGA and CPU chips.

System Before (Discrete Architecture):

- **Multiple Devices:** Separate FPGA and CPU chips
- **Multiple Memories:** Different memory for each device
- **Limited Bandwidth:** Chip-to-chip communication bottleneck
- **Complex PCB:** More layers, more routing
- **Higher Costs:** Multiple chips, complex board
- **Power Consumption:** Data transfer between chips costs energy
- **Speed:** Inter-chip communication is slow

Layout: One PCB contains discrete FPGA & CPU as separate components.

SoC FPGA Integrated Architecture

Modern SoC FPGAs integrate CPU and FPGA on a single die.

System After (SoC FPGA):

- **Single Device:** CPU and FPGA in one chip
- **Single Memory:** Shared memory system
- **High Bandwidth:** Internal connections between FPGA and CPU
- **Simpler PCB:** Fewer layers required
- **Lower Costs:** Single chip solution
- **Less Power:** No inter-chip communication overhead
- **Better Performance:** Fast internal data paths

SoC FPGA architecture provides significant advantages in bandwidth, power, cost, and PCB complexity.

Evolution Stages of SoC FPGA

Stage 1: Pure FPGA

Original FPGA architecture with basic programmable logic.

Components:

- **Programmable Logic:** LUTs and flip-flops
- **GPIO:** General-purpose I/O pins
- **On-Chip Memory:** SRAM blocks
- **Clock Generation:** PLLs for clock management
- **Multipliers:** Dedicated DSP blocks
- **Fast Serial I/O:** Gigabit transceivers (SERDES)

Capabilities:

- Hardware acceleration
- Parallel processing
- Real-time deterministic operation

Stage 2: FPGA with Softcore Processor

Add programmable CPU implemented in FPGA fabric.

Softcore Processor Characteristics:

- **Implementation:** IP core in FPGA fabric (e.g., NIOS V, RISC-V)
- **Software Execution:** Can run software inside FPGA
- **Clock Speed:** Relatively slow CPU clock (limited by fabric)
- **Memory:** Small on-chip memory or external memory via I/O pins
- **Resource Usage:** Consumes FPGA logic resources
- **Flexibility:** Configurable architecture (instruction set, peripherals)

Advantages:

- No external CPU required
- Customizable processor architecture
- Tight integration with FPGA logic

Disadvantages:

- Uses valuable FPGA resources
- Lower performance than hard CPU
- Limited memory bandwidth

Stage 3: SoC with Hard Processor Core

Integrate dedicated hardware CPU with FPGA fabric.

Hard Processor System (HPS) Characteristics:

- **Full Integrated Processor:** Dedicated silicon, not programmable fabric
- **Fast CPU Clock:** Up to 1.0 GHz or higher
- **Dedicated Peripherals:** Hardware I/O controllers
- **Dedicated Pins:** Separate pins for CPU peripherals
- **Memory Controller:** Dedicated SDRAM controller to external memory
- **High Bandwidth:** Fast data transfers between FPGA and CPU
- **No FPGA Resources:** Doesn't consume programmable logic

Advantages:

- High CPU performance
- Full operating system support
- Preserves all FPGA resources
- Professional peripherals (Ethernet, USB, PCIe)

System-Level Benefits of SoC FPGA

Performance Advantages

SoC FPGAs provide exceptional computational capabilities:

Processing Power:

- **1,600 GMACS:** Multiply-accumulate operations in FPGA
- **320 GFLOPS:** Floating-point operations in DSP multipliers
- **125 Gbps:** Interconnect bandwidth between CPU and FPGA

Memory Architecture:

- **Cache Coherent Access:** FPGA can access CPU caches
- **Shared Memory:** Unified memory space
- **DMA Support:** Direct memory access for high-speed transfers

CPU Integration:

- **Custom Instructions:** Hardware accelerators appear as CPU instructions
- **Tight Coupling:** Minimal latency between CPU and FPGA
- **Coherent Fabric:** Automatic cache management

The combination of high-speed interconnects and cache coherency enables unprecedented CPU-FPGA co-operation.

Processor-Level Benefits

The HPS provides full-featured processor capabilities:

CPU Features:

- **Fast Multicore:** Up to 1 GHz or higher clock speed
- **Full OS Support:** Can run Linux, FreeRTOS, etc.
- **MMU:** Memory Management Unit for virtual memory
- **Cache Hierarchy:** L1 and L2 caches for performance

Peripheral Expansion:

- Ability to expand CPU peripherals by placing IP blocks in FPGA
- Add controllers: SD, I^{2}C, CAN, UART, Ethernet, USB
- Custom interface protocols
- Flexible I/O configuration

Hardware Acceleration:

- **Custom Accelerators:** Complex calculations offloaded to FPGA
- **Parallel Execution:** CPU continues other tasks while FPGA computes
- **Direct Memory Access:** DMA for efficient data transfer
- **Cache Coherency:** Automatic synchronization between CPU and FPGA

SoC FPGA Block Architecture

SoC FPGA Main Components

A SoC FPGA consists of three main sections:

1. CPU Portion (Hard Processor System):

- Cortex-A9 CPU Subsystem (dual-core)
- Flash Controllers
- SDRAM Controller Subsystem
- On-Chip Memories
- Support Peripherals
- PLLs (Phase-Locked Loops)
- Debug Interface
- Peripheral Control Block

2. FPGA Portion:

- User I/O pins
- FPGA Fabric (LUTs, RAMs, Multipliers, Routing)
- HSSI Transceivers (High-Speed Serial Interface)
- PLLs
- Hard PCIe controller
- Hard Memory Controllers

3. Processor-FPGA Bridges:

- High-bandwidth connections
- Multiple bridge types for different use cases
- Enable communication between CPU and FPGA

SoC FPGA Architecture Details

Key Characteristics:

Three Main Parts:

- FPGA fabric
- CPU (Hard Processor System)
- Bridges connecting them

FPGA Independence:

- FPGA fabric is same as in FPGA-only devices
- FPGA may be used without CPU (some manufacturers)
- CPU can be powered down if not needed

CPU Options:

- ARM Cortex-A9, -A53, -A78 (dual- to octa-core)
- ARM Cortex-R52 (real-time)
- RISC-V processors also available

Interconnect Features:

- Multiple bridges provide high bandwidth
- Multiport SDRAM controller handles concurrent accesses
- DMA controller for efficient data movement

Detailed SoC FPGA Architecture

ARM Cortex-A9 CPU Subsystem

The HPS is built around ARM Cortex-A9 processor cores.

CPU Features:

- **Dual-Core:** Two ARM Cortex-A9 processors
- **L1 Cache:** Per-core Level 1 cache (instruction & data)
- **L2 Cache:** Shared Level 2 cache for both cores
- **FPU:** Floating-Point Unit per core
- **NEON:** SIMD (Single Instruction Multiple Data) instruction set

Pin Configuration:

- **Dedicated MPU Pins:** For CPU peripherals
- **Dedicated DDR Pins:** For external SDRAM
- **FPGA Pins:** Shared or dedicated for FPGA I/O

Interconnect:

- System bus: Pipelined network between components
- High-bandwidth internal connections
- Multiport DDR controller for concurrent access

HPS Peripherals and Controllers

The HPS includes comprehensive peripheral support:

Standard Microcontroller Peripherals:

- Timers (watchdog, general-purpose)
- SPI (Serial Peripheral Interface)
- I^{2}C (Inter-Integrated Circuit)
- UART (Universal Asynchronous Receiver-Transmitter)
- GPIO (General-Purpose I/O)

High-Speed Interfaces:

- **Gigabit Ethernet:** To external PHY
- **PCIe:** PCI Express controller
- **USB:** USB 2.0 OTG controller

Memory Interfaces:

- **Static Memory Controller:** For QSPI NAND/NOR Flash, SD-card
- **Multiport DDR Controller:** For external SDRAM

System Components:

- **DMA:** Direct Memory Access controller
- **Scratch RAM:** Used in boot process, sufficient for FreeRTOS
- **Boot ROM:** Contains boot code

FPGA-HPS Bridges

Altera FPGA-HPS Bridge Architecture

Three main communication paths connect FPGA and HPS:

1. General Purpose AXI Bridges:

- **H2F (HPS-to-FPGA):** CPU accesses FPGA components
- **F2H (FPGA-to-HPS):** FPGA accesses HPS memory/peripherals
- **Width:** Up to 128-bit wide
- **Modes:** Streaming and memory-mapped
- **Use:** High-bandwidth data transfers

2. Lightweight AXI Bridge:

- **Width:** 32-bit
- **Mode:** Memory-mapped access only
- **Use:** Control registers, configuration, status
- **Latency:** Lower latency than general purpose bridges

3. Multiport DDR Controller:

- **Width:** 256-bit
- **Function:** FPGA direct access to external SDRAM
- **Feature:** Cache coherent with CPU
- **Use:** Shared memory between CPU and FPGA

Different bridges are optimized for different use cases: lightweight for control, general purpose for data, DDR for shared memory.

Address Mapping Example

Memory Map:

Region Name	Base Address	Size
Peripherals	0xFC00_0000	62 MB
Lightweight FPGA slaves	0xFF20_0000	2 MB

Example: Control Register in FPGA

CPU View (Full Address):

0xFF22_0030

This address selects the lightweight bridge (starts with 0xFF20_xxxx).

FPGA View (Local Address):

0x0002_0030

The bridge translates the address:

- Remove base address: 0xFF20_0000
- Remaining offset: 0x0002_0030
- This is the address seen by FPGA component

Address translation by bridges allows FPGA components to use simple local addresses while appearing in CPU memory map.

SoC FPGA Configuration

Three Types of Components in SoC FPGA

SoC FPGAs contain different types of configurable components:

1. CPU Hard Macros (Silicon):

- CPU cores, SDRAM controller
- Processor-FPGA bridges
- Fixed hardware, not reconfigurable
- **Configuration Method:** Platform Designer

2. FPGA Hard Macros (Silicon):

- Gigabit transceivers (HSSI)
- PLLs (Phase-Locked Loops)
- PCIe controller
- I/O blocks, multipliers, memory blocks
- **Configuration Method:** Wizards (generate VHDL or config files)

3. Soft Macros (Programmable):

- I/O interfaces, PLLs
- Custom logic in FPGA fabric
- IP blocks from library
- **Configuration Method:**
  - Soft IP blocks in Platform Designer
  - Custom VHDL code

Platform Designer HPS Configuration

Platform Designer HPS Block

Platform Designer provides a graphical interface for configuring the HPS.

Available Interfaces:

- **memory:** Memory interface conduit
- **hps_io:** HPS I/O conduit
- **h2f_reset:** Reset output from HPS
- **h2f_axi_master:** HPS-to-FPGA AXI master
- **f2h_axi_slave:** FPGA-to-HPS AXI slave
- **h2f_lw_axi_master:** Lightweight HPS-to-FPGA AXI master
- **f2h_irq0/1:** Interrupt receivers (32 interrupts each)

Clock Inputs:

- Separate clocks for each AXI interface
- Allows different clock domains

Configuring HPS in Platform Designer

Configuration Tabs:

1. FPGA Interfaces:
- ☐ Enable MPU standby and event signals
- ☐ Enable general purpose signals
- ☐ Enable Debug APB interface
- ☐ Enable System Trace Macrocell hardware events
- ☐ Enable FPGA Cross Trigger interface
- ☐ Enable FPGA Trace Port Interface Unit
- ☐ Enable boot from FPGA signals
- ☐ Enable HLGPI Interface

2. AXI Bridges:
- **FPGA-to-HPS width:** 64-bit, 128-bit, or 256-bit
- **HPS-to-FPGA width:** 32-bit, 64-bit, or 128-bit
- **Lightweight width:** 32-bit (fixed)

3. Peripheral Pins:
- Select pin-set for each peripheral
- Enable or disable peripherals
- Configure pin multiplexing

4. SDRAM Configuration:
- DDR3/DDR2 parameters
- Memory timing configuration
- Port assignments

5. HPS Clocks:
- Clock source selection
- PLL configuration
- Clock dividers

Hardware Development Flow

Platform Designer Role

Platform Designer is Intel's system builder for SoC designs.

Configuration Scope:
- **CPU and Peripherals:** Select and configure HPS features
- **IP Blocks:** Add and configure FPGA IP components
- **Interconnects:** Automatic generation of bus connections

Output Products:
- **Hardware Description:** Memory map, device types
- **For Software:** Header files with addresses and registers
- **For FPGA:** VHDL files for interconnects and peripherals
- **System Configuration:** Stored project settings

IP Implementations:
- peripheral_1.vhdl
- peripheral_n.vhdl
- Interconnect fabric
- Address decoders

Complete Hardware Development Flow

Step 1: System Definition (Platform Designer)
- Configure HPS and peripherals
- Add and configure IP blocks
- Define interconnections
- Generate system

Step 2: Quartus Compilation
- **Synthesis:** Convert VHDL to gate-level netlist
- **Optimization:** Minimize logic usage
- **Mapping:** Map to FPGA resources
- **Placement:** Place components on chip
- **Routing:** Route connections between components
- **Retiming:** Optimize for timing
- **Fitting:** Final placement optimization

Step 3: Timing Verification
- Static timing analysis
- Check all timing constraints met
- Identify critical paths

Step 4: Bitstream Generation
- Generate configuration bitstream
- Include HPS parameters
- Program into device

Booting and Device Tree

Learning Objectives

Course Goals

After this lecture and lab exercise, students will be able to:

- Sketch the resources that need to be configured during boot
- Explain the boot sequence of an SoC
- Configure and build a bootloader program for their SoC
- Understand the basic concept of a device tree
- Modify the device tree

SoC FPGA Architecture Overview

SoC FPGA Boot Components

The Cyclone V SoC FPGA architecture includes:

Hard Processor System (HPS):

- Dual ARM Cortex-A9 cores with NEON/FPU
- L1 Cache (instruction and data)
- L2 Cache (shared)
- Dedicated MPU pins
- Dedicated DDR pins for external SDRAM

Interconnect:

- L3 Interconnect fabric
- SDRAM Controller Subsystem
- Bridges between HPS and FPGA

FPGA Fabric:

- Programmable logic
- FPGA pins for custom I/O
- Connection to HPS via bridges

Boot Sequence Overview

Boot Process

Booting a processor involves bringing it from reset to a running application through multiple steps.

Boot Stages:

Reset → Boot ROM → Preloader (SPL) → Bootloader (U-Boot) → Operating System → Application

Key Characteristics:

- Each step executed by hard or software
- Not all steps always required
- User software (everything after Boot ROM) can be modified
- Progressive initialization of system resources

Stage 1: Reset

Reset Stage

Reset brings the HPS to a defined default state.

When Reset Occurs:

- At power-on
- On user request (button or input signal on dedicated pin)
- Triggered by debugger
- Triggered by watchdog timer
- Triggered from FPGA fabric

Reset Process:

- Controlled by hardware reset controller block
- Controls sequence of resetting modules
- Resets clocks
- Resets CPU cores
- Reaches defined default state for critical resources

Stage 2: Boot ROM

Boot ROM Stage

The Boot ROM is the first code executed after reset.

Characteristics:

- 64 KB of code in on-chip ROM
- Located at reset exception address
- Cannot be modified (hard-coded in silicon)

Boot ROM Functions:

1. Determine boot source (read BSEL pins)
2. Initialize HPS core systems:
   - Set up watchdog timer
   - Configure clocking (PLL)
   - Initialize flash controller
3. Copy preloader program from selected boot source to on-chip RAM
4. Jump execution to preloader program

Boot Source Selection (BSEL)

BSEL is a group of input pins that tell Boot ROM which device to boot from.

BSEL Pin Configuration:

BSEL[2..0]	Flash Device
0x0	Reserved
0x1	FPGA (HPS-to-FPGA bridge)
0x2	1.8V NAND flash memory
0x3	3.3V NAND flash memory
0x4	1.8V SD/MMC flash memory
0x5	3.3V SD/MMC flash memory
0x6	1.8V quad SPI flash memory
0x7	3.3V quad SPI flash memory

Hardware Configuration:
- BSEL pins set by DIP switches or resistors on PCB
- Configuration must match physical boot device
- Determines which flash memory Boot ROM will access

SD Card Boot Structure

Boot from SD Card:
When BSEL is configured for SD card boot, the Boot ROM looks for the preloader on the SD card.

SD Card Partition Structure:
- **Master Boot Record (MBR):** 512 bytes
  - Stores partition table
  - Located at beginning of SD card
- **RAW Partition (Type A2):** 64 KB $\times$ 4 = 256 KB
  - Contains four copies of preloader image
  - Redundancy for reliability
  - No filesystem (raw binary data)

Boot Process:
1. Boot ROM reads MBR to find partition table
2. Locates RAW partition (type A2)
3. Copies preloader image to on-chip memory
4. Jumps to preloader entry point

The preloader partition has no filesystem - it contains raw executable code.

Stage 3: Preloader (SPL)

Preloader/SPL (Second Program Loader)

The preloader is user-defined code that initializes critical hardware.

Key Responsibilities:
- **Clock Reconfiguration:** Set up system clocks for optimal performance
- **SDRAM Initialization:** Initialize SDRAM interface and check RAM size
- **I/O Pin Configuration:** Configure I/O pins of HPS
- **Peripheral Initialization:** Initialize peripheral interfaces needed to load U-Boot
- **Load U-Boot:** Copy U-Boot bootloader to SDRAM
- **Jump to U-Boot:** Transfer execution to bootloader

Characteristics:
- Executes from on-chip RAM (fast, limited size)
- User-defined and customizable
- Must be small to fit in on-chip RAM
- Critical for system initialization

Stage 4: Bootloader (U-Boot)

U-Boot Bootloader

U-Boot is a versatile, feature-rich bootloader program.

Primary Functions:
- **Hardware Tests:** Verify system functionality
- **FPGA Configuration:** Load bitstream to FPGA fabric
- **Load Linux Kernel:** Copy kernel image to memory
- **Load Device Tree:** Provide hardware description to kernel

Open Source:
- Available at: https://docs.u-boot.org/en/latest/
- Highly configurable
- Active community support
- Board-specific configurations available

U-Boot Capabilities

U-Boot is the earliest point of developer interaction during boot.

Testing and Access:
- Access and test SDRAM
- Test FPGA fabric
- Test on-chip peripherals
- Access flash memory
- Access USB devices
- Access network interfaces

User Interface:
- Interactive shell for user commands
- Scripting support
- Environment variable configuration
- Boot menu system

System Configuration:
- Configure FPGA fabric
- Set boot parameters
- Load and execute operating system kernel

Stage 5: Operating System

Operating System Boot
The OS is loaded and initialized by the bootloader.
OS Choices:
- **RTOS:** Real-Time Operating System
- **Embedded Linux:** Lightweight Linux distribution
- **Desktop Linux:** Full-featured Linux distribution

Hardware Description Requirement: The OS needs a device tree to know what hardware is present:
- In the HPS (Hard Processor System)
- In FPGA fabric (custom IP blocks)
- On the PCB (external peripherals)

Without a device tree, Linux cannot discover and utilize the hardware automatically.

Stage 6: Application

User Application
The final stage is the user application running under the OS.
Characteristics:
- Started in its own process context by the OS
- OS provides resource management
- OS controls access to FPGA and peripherals
- OS provides device drivers
- Application runs in user space (protected mode)

Resource Access:
- Access to hardware through OS drivers
- Memory mapping for FPGA peripherals
- Standard Linux APIs available
- Multi-process support

Boot Sequence Options

Customization Possibilities
Everything marked Üser Software"can be modified by the developer:
Modifiable Components:
- Preloader (SPL)
- Bootloader (U-Boot)
- Operating System
- Applications

Developer Actions:
- Optimize performance
- Add capabilities
- Remove unnecessary features
- Contribute to open source projects

Alternative Boot Path:
An RTOS can be loaded directly by the Boot ROM by replacing the Preloader image.
Questions to Consider:
- Why would you do this?
- What are the advantages?
- What problems might arise?

Analysis:
- **Advantage:** Faster boot time (fewer stages)
- **Advantage:** Simpler system (less software)
- **Problem:** Limited initialization (Boot ROM constraints)
- **Problem:** Size constraints (must fit in space allocated for preloader)

Device Tree

Device Tree Concept
A device tree is a data structure that describes hardware to the operating system.
Why Needed:
- Embedded Linux cannot automatically discover hardware
- Hardware configuration varies between designs
- FPGA fabric contains custom IP blocks
- PCB layout differs between boards

Loading Process:
- Bootloader must load Device Tree Blob (DTB) to memory
- DTB loaded before kernel boot
- Kernel reads DTB to discover hardware

Information Provided:
- Device address ranges
- Type of device
- Compatible drivers
- Register maps
- Clock configurations
- Interrupt assignments
- Other device properties

Device Tree Functions

The device tree serves multiple purposes:

Memory Map Description:

- Describes complete system memory map
- Translates between memory maps of subregions
- Maps HPS memory space
- Maps FPGA peripheral space
- Maps external peripheral space

Peripheral Details:

- Peripheral base addresses
- Address ranges and sizes
- Interrupt connections
- Clock sources
- Driver compatibility information

Bridge Configuration:

- HPS-FPGA bridge specifications
- Data bus width
- Address translation
- Memory-mapped block types
- Associated drivers

Device Tree Exploration in Linux

The device tree can be explored in the running Linux file system:

File System Locations:

- `/sys/class` - Device classes
- `/sys/devices/platform/sopc@0` - Platform devices

Description Contents:

- Hierarchy of hardware components
- Peripheral addresses and clocks
- Compatible driver information
- Device status (enabled/disabled)
- HPS-FPGA bridge configuration
- Memory map details

Device Tree Structure

Device Tree Hierarchy

The device tree is organized as a hierarchical structure:

Root Node: /

- `#size-cells`: Number of cells for size values
- `#address-cells`: Number of cells for address values
- `compatible`: Device compatibility string

Major Child Nodes:

- `cpus{}`: CPU configuration
  - Bus frequency
  - CPU cores and their properties
- `memory{}`: Physical memory description
- `clocks{}`: Clock sources and configurations
- `sopc@0`: SoC peripheral controller
  - `bridge@0xc0000000`: HPS-FPGA bridge
  - Memory map definitions
  - `jtag_uart_0: serial@0x...`: JTAG UART peripheral
  - Additional peripheral nodes

Device Tree Nodes

Devices are described as nodes in the device tree.

Node Structure:

```
label: name@unit_address
```

Node Contents:

- **Properties:** Key-value pairs describing the device
- **Child Nodes:** Nested devices or sub-components

Example Node:

```
cpu@0x1 {
    device_type = "cpu";
    compatible = "arm,cortex-a9";
    reg = <0x00000001>;
    next-level-cache = <&hps_0_L2>;
    enable-method = "psci";
};
```

Node Components:

- `cpu@0x1`: Node name with unit address
- `device_type`: Type of device
- `compatible`: Compatible driver string
- `reg`: Register address
- `next-level-cache`: Reference to cache (phandle)

Node Structure Components

A typical peripheral node contains several key properties:

Example: JTAG UART Node

```
serial@0x100020000 {
    compatible = "altr,juart-16.0", "altr,juart-1.0";
    reg = <0x00000001 0x00020000 0x00000008>;
    interrupt-parent = <&hps_0_arm_gic_0>;
    interrupts = <0 42 4>;
    clocks = <&clk_0>;
};
```

Property Descriptions:

- **Label:** serial@0x100020000
- **Properties:** Device characteristics and configuration
- **Values:** Specific settings for this instance

Node Properties

Common Node Properties

Standard properties found in device tree nodes:

compatible

- Defines programming model
- Identifies compatible driver
- Can list multiple compatible strings (fallback options)
- Example: ältr,juart-16.0", ältr,juart-1.0"

reg

- Address and length of memory region
- Format: <base_address length>
- Can use multiple address cells for extended addressing
- Example: <0x00000001 0x00020000 0x00000008>
  - Address: 0x1'0002'0000 (35-bit ARM physical address)
  - Length: 0x08 bytes

interrupt-parent

- Phandle to interrupt controller
- Specifies which controller handles interrupts
- Example: <&hps_0_arm_gic_0>

interrupts

- Interrupt number(s) and configuration
- Format depends on interrupt controller
- Example: <0 42 4>

clocks

- Clock reference (phandle)
- Links to clock provider node
- Example: <&clk_0>

35-bit ARM Physical Address:
The ARM Cortex-A9 in Cyclone V uses a 35-bit physical address space.
Address Ranges:

- HPS internal: 0x0'0000'0000 - 0x0'FFFF'FFFF
- LW (Lightweight) Bridge: 0x1'0000'0000 - 0x1'FFFF'FFFF
- HPS-to-FPGA Bridge: Higher addresses

Everything on the Lightweight bridge uses addresses starting at 0x1'0000'0000.

Bus Properties

Bus Node Properties

Bus nodes define address spaces and translation between parent and child buses.

Example: HPS-FPGA Bridge

```
bridge@0xc0000000 {
    compatible = "altr,bridge-16.0", "simple-bus";
    reg = <0xc0000000 0x20000000>,  // HPS-FPGA bridge
          <0xff200000 0x00200000>;  // Lightweight bridge
    reg-names = "axi_h2f", "axi_h2f_lw";
    clocks = <&clk_0 &clk_0>;
    clock-names = "h2f_axi_clock", "h2f_lw_axi_clock";
    #address-cells = <2>;
    #size-cells = <1>;
    ranges = <0x00000000 0x00000000 0xc0000000 0x00010000>,
             <0x00000001 0x00021000 0xff221000 0x00000020>,
             ...
};
```

Key Properties:

- **reg:** Base address and range of bridge regions
- **#address-cells:** Number of 32-bit cells for base address (2 = 64-bit)
- **#size-cells:** Number of 32-bit cells for size (1 = 32-bit)
- **ranges:** Address translation table

Memory Ranges and Address Translation

The ranges property defines address translation between buses.

Format:

<child_address parent_address size>

Example:

```
ranges = <0x00000000 0x00000000 0xc0000000 0x00010000>,
         <0x00000001 0x00021000 0xff221000 0x00000020>,
         <0x00000001 0x00021020 0xff221020 0x00000020>,
         <0x00000001 0x00020000 0xff220000 0x00000008>;
```

Translation Components:

- **Child address:** Address on the child bus (from device perspective)
- **Parent address:** Address on parent bus (from CPU/Linux perspective)
- **Size:** Size of the mapped region

Usage:

- Linux programs use parent address
- Hardware uses child address
- Device tree provides translation

Example Device Reference:

```
jtag_uart: serial@0x100020000 {
    reg = <0x00000001 0x00020000 0x00000008>;
    ...
};
```

Child address 0x1'0002'0000 translates to parent address 0xff220000 for Linux access.

Building the Device Tree

Device Tree Build Process

Step 1: Extract System Information

The system description in `*.sopcinfo` contains the memory map in XML format.

```
# Convert .sopcinfo into human-readable header file
sopc-create-header-files pacman_soc.sopcinfo
# Generates: system.h
```

Step 2: Add Device Nodes

Add custom nodes to device tree template using vendor documentation.

Example: GPIO Node

```
gpio_altr: gpio@ff200000 {
    compatible = "altr,pio-1.0";
    reg = <0xff200000 0x10>;
    interrupts = <0 45 4>;
    altr,ngpio = <32>;
    altr,interrupt-type = <IRQ_TYPE_EDGE_RISING>;
    #gpio-cells = <2>;
    gpio-controller;
    #interrupt-cells = <2>;
    interrupt-controller;
};
```

Step 3: Compile Device Tree

Use device tree compiler (`dtc`) to compile source to binary blob.

Refer to vendor documentation for specific property requirements and values.

Device Tree Compilation Flow

The complete compilation flow:

Input Files:

1. **.sopcinfo:** Platform Designer system description
2. **system.h:** Generated header with memory map
3. ***.dtsi:** Device tree include file with custom nodes
4. ***.dts:** Main device tree source file

Tools:

1. `sopc-create-header-files`: Convert .sopcinfo to .h
2. `dtc` (Device Tree Compiler): Compile .dts to .dtb

Output:

- **.dtb:** Device Tree Blob (binary format)
- Loaded by bootloader
- Passed to Linux kernel

File Locations:

- Base template: `socfpga_cyclone5_de1soc.dtsi`
- Custom additions: Your `*.dtsi` file
- Final blob: `socfpga.dtb`

Booting the Kernel

U-Boot Configuration File

U-Boot uses a configuration file to locate kernel and device tree.

File: extlinux.conf

- Located in BOOT partition of SD card
- Text file with boot configuration

Configuration Parameters:

```
KERNEL: "zImage"          # Kernel image filename
FDT: "socfpga.dtb"        # Device Tree Blob filename
APPEND: root=/dev/mmcblk0p2  # Boot options (root filesystem)
```

Boot Process:

1. U-Boot reads `extlinux.conf`
2. Locates kernel image file
3. Locates device tree blob file
4. Loads both to memory
5. Passes control to kernel with device tree pointer

U-Boot Loader Output During Boot:

1. Read `extlinux.conf` configuration
2. Load kernel image into memory
3. Load Device Tree Blob into memory
4. Start kernel execution

The console output shows each step with memory addresses and file sizes, providing visibility into the boot process.

U-Boot Build Process

U-Boot Build Flow

Building U-Boot requires cross-compilation and configuration.

Prerequisites:

- Cross-compiler toolchain for ARM
- U-Boot source code
- Board-specific configuration
- Generated handoff files from Quartus

Build Flow:

1. Configure HPS in Platform Designer
2. Compile design in Quartus
3. Generate handoff folder
4. Build preloader (SPL)
5. Configure U-Boot for board
6. Compile U-Boot
7. Combine SPL and U-Boot into single image
8. Copy to SD card

Platform Designer Configuration

Building SoC System in Platform Designer

Step 1: Configure HPS Block

- Configure HPS block settings
- Set up clocks and PLLs
- Configure SDRAM controller
- Enable peripherals
- Configure FPGA-HPS bridges

Step 2: Add FPGA IP Blocks

- Add custom IP blocks to FPGA fabric
- Connect via bridges to HPS
- Configure IP block parameters

Step 3: Assign Addresses

- Assign base addresses to all peripherals
- Create memory map
- Verify no address conflicts
- Document address assignments

Example System Components:

- **hps_0:** Hard Processor System block
- **sysid_qsys_0:** System ID peripheral
- **jtag_uart_0:** JTAG UART for debug
- **pio_0:** Parallel I/O block
- Clock and reset connections

SOPCINFO File

.sopcinfo File

The `.sopcinfo` file is generated by Platform Designer and contains the complete system description.

Contents:

- Description of HPS, including all configuration settings
- Description of IP blocks and their parameters
- Interconnect topology (connections between blocks)
- Memory addresses and address maps
- Clock and reset connections
- Interrupt assignments

Format:

- XML-based format
- Machine-readable
- Contains embedded C macros for software

Example Content:

```
<kind>embeddedsw.CMacro.BIT_CLEARING_EDGE_REGISTER</kind>
<version>0</version>
<name>embeddedsw.CMacro.BIT_MODIFYING_OUTPUT_REGISTER</name>
<name>embeddedsw.CMacro.CAPTURE</name>
```

Usage:

- Input to `sopc-create-header-files`
- Generates `system.h` header file
- Used for building bootloader
- Used for building device tree

Quartus Handoff Folder

Handoff Folder

Quartus compiler creates a handoff folder containing configuration information required to build the preloader and bootloader.

Generated Files:

- `alt_types.h` - Type definitions
- `emif.xml` - EMIF configuration
- `hps.xml` - HPS configuration
- `pacman_soc_hps_0.hiof` - HPS I/O configuration
- `sdram_io.h` - SDRAM I/O settings
- `sequencer.c/.h` - Memory sequencer code
- `sequencer_auto*.c/.h` - Auto-generated sequencer files
- `sequencer_defines.h` - Sequencer definitions
- `system.h` - System memory map
- `tclrpt.c/.h` - TCL report files

Configuration Information:

- Pin configuration (pinmux)
- Enabled peripherals
- Clock configuration
- SDRAM timing and configuration

hps.xml Configuration File

The `hps.xml` file contains HPS peripheral and clock configuration.

Example Contents:

```
<name>EMAC</name> <choice>value</choice>
<name>USB</name> <choice>value</choice>
<name>SPI</name> <choice>value</choice>
<name>I2C</name> <choice>value</choice>
<name>SDMMC</name> <choice>value</choice>
<name>UART</name> <choice>value</choice>
<name>CAN</name> <choice>value</choice>
<name>PLL_CLK</name> <value>...</value>
<name>SDRAM_CLK</name> <value>...</value>
```

Configuration Categories:

- Peripheral enable/disable settings
- Clock frequency settings
- Pin multiplexing choices
- I/O standard selections

The handoff folder contains minimal but essential configuration information.

BSP Generator

Creating Bootloader BSP (Board Support Package)

Tool: bsp-create-settings
The BSP generator converts the handoff folder to header files for bootloader compilation.
Command:

```
cd $PROJECT/
mkdir -p software/bootloader

bsp-create-settings \
  --type spl \
  --bsp-dir software/bootloader \
  --preloader-settings-dir "hps_isw_handoff/pacman_soc_hps_0" \
  --settings software/bootloader/settings.bsp
```

Generated Files:
- Header files in generated/ directory
- Pin multiplexing configuration
- PLL configuration
- Reset manager configuration
- SDRAM initialization sequences

Output Directory:
- `software/bootloader/generated/`
- Contains all necessary headers
- Ready for preloader compilation

U-Boot Compilation

Cross-Compiling U-Boot

Prerequisites:
- ARM cross-compiler toolchain installed
- U-Boot source code (e.g., socfpga_v2020.04 branch)
- Modified source code for your board (if needed)

Step 1: Configure for Target Board

```
cd ~/de1soc/
# Select default configuration for Cyclone V
make socfpga_cyclone5_defconfig
# Output: configuration written to .config
```

Step 2: Compile

```
make
# Compiles all components:
# - scripts/basic/fixdep
# - scripts/kconfig/conf.o
# - SPL (preloader)
# - U-Boot proper
# Output: u-boot-with-spl.sfp
```

Build Output Files:
- `u-boot-with-spl.sfp` - Combined preloader + U-Boot image
- Ready to copy to SD card

Customization:
- Modify source code as needed for custom boards
- Adjust board-specific settings
- Add custom commands or features

Copying to SD Card

Writing Bootloader to SD Card

Important Note: The preloader partition has no filesystem - it contains raw binary data.
Command:

```
# Copy bootloader image to raw partition (type A2)
dd if=u-boot-with-spl.sfp of=/dev/sdx3 bs=1M
sync
```

Parameters:
- `if=`: Input file (bootloader image)
- `of=`: Output device (`/dev/sdx3` where x is SD card device letter)
- `bs=1M`: Block size of 1 megabyte
- `sync`: Ensure data is written to disk

Important:
- The bootloader image contains 4 copies of the preloader
- Correctly formatted for raw partition
- No filesystem needed
- Direct binary copy to partition

Always verify the correct device name (sdx) before running dd command to avoid overwriting wrong disk!

SDRAM - Synchronous Dynamic Random Access Memory

Introduction and Motivation

Learning Objectives

- List the pros and cons of SDRAM
- Describe the principle of operation of an SDRAM cell
- Calculate the number of banks, words, and bits of an SDRAM device
- Determine the refresh frequency of an SDRAM device
- Extract timing properties from an SDRAM datasheet
- Interpret the read and write accesses in a timing diagram and identify the corresponding timing parameters
- Explain the internal operation of a DDR SDRAM device

Memory Hierarchy and Storage Types

Computer Storage Hierarchy

Memory systems are organized hierarchically based on speed, cost, and capacity.

Hierarchy levels:

- **Fast and expensive:** Primary Memory (SRAM, DRAM)
- **Medium:** Secondary Memory (Disk)
- **Slow and cheap:** Tertiary Memory (Tape)

Trade-offs:

- High capacity $\Rightarrow$ Slower access time
- Fast access $\Rightarrow$ High cost per bit

Typical Memory System Configuration

System components and timing:

- **1st Level Cache:** Integrated in processor
- **2nd Level Cache:** SRAM, typical 5 ns access time
- **Main Memory:** DRAM, typical 50 ns access time (10x slower than cache)
- **Disk:** Typical 5 ms access time (100,000x slower than DRAM)
- **Tape:** Typical 5-50 s access time (1,000x slower than disk)

Example calculation:

For a 2 GHz processor (0.5 ns per clock cycle):

- Disk access time: 5 ms = 10,000,000 clock cycles
- Transfer rate: Typically 1-4 GByte/s between cache levels
- I/O Bus: Typically 100-200 MByte/s

The large speed differences motivate the use of caching and hierarchical memory structures.

Types of Semiconductor Memory

- **Volatile Memory:** Data lost when power is removed
  - SRAM - Static Random Access Memory
  - SDRAM - Synchronous Dynamic Random Access Memory (SDR, DDR)
- **Non-volatile Memory:** Data retained without power
  - Mask-programmed ROM
  - PROM - Programmable Read Only Memory (OTP)
  - EEPROM - Electrically Erasable PROM
  - Flash EEPROM (NOR, NAND)
  - nvSRAM, FRAM - Non-volatile RAM technologies

SRAM vs DRAM Comparison

Static RAM (SRAM)

Cell structure:

- 6 transistors per bit (flip-flop/latch)
- Large cell area

Characteristics:

- Low density, high cost
- Up to 512 Mb per device
- Almost no static power consumption
- Asynchronous interface (no clock)
- Simple bus connection
- All accesses take similar time (approximately 5 ns, 200 MHz)
- Suitable for distributed accesses

Dynamic RAM (DRAM)

Cell structure:

- 1 transistor + 1 capacitor per bit
- Small cell area
- Reduced complexity
- Reduced cost

Characteristics:

- High density, low cost
- Up to 64 Gb per device
- Leakage currents require periodic refresh
- Synchronous interface (clocked)
- Requires dedicated SDRAM controller
- Long latency for first access
- Fast access for burst data
- Large overhead for single byte access

DRAM Cell Operation

Storage mechanism:

Information is stored as electrical charge in a capacitor. The charge represents a binary value (0 or 1).

Key features:

- Information stored as charge in capacitor
- High integration density allows large memories at low cost
- Leakage current causes loss of charge
- Capacitor holds charge only for a few milliseconds
- Charge must be refreshed periodically $\Rightarrow$ *dynamic*
- Refresh logic usually located on SDRAM device

Cell components:

- Bit line: used for reading/writing data
- Word line: activates transistor for access
- FET (Field Effect Transistor): controls access to capacitor
- Capacitor (C): stores the charge

The reduced cell size (1T+1C vs 6T) allows DRAM to achieve much higher densities than SRAM, making it the preferred technology for main memory despite the need for refresh circuitry.

Historical Context

Inventor: Robert H. Dennard

- Born in 1932
- Invented DRAM at IBM in 1968
- Revolutionary impact on computer memory technology

---

1 Mbit DRAM (1985):

- Die size: 8.66 mm $\times$ 3.96 mm
- First megabit-scale DRAM

Memory Architecture and Organization

Basic Memory Architecture

A memory is organized as an $n \times m$ array:

- $n =$ number of words
- $m =$ number of data bits per word
- Bit cell stores a single bit ('0' or '1')

Components:

- k address lines where $n = 2^k$
- Address decoder selects one out of n word lines
- Word lines: one per memory word
- Data lines: m lines for word width
- Address range: 0 (lowest) to $n - 1$ (highest)

SDRAM Memory Devices

DRAM cells use less area on silicon, allowing more bits per device.

Capacity comparison:

- **SRAM:** Up to 512 Mb $\Rightarrow 2^{29}$ bits $\Rightarrow$ 26 address lines (byte addressing)
- **DRAM:** Up to 512 Gb $\Rightarrow 2^{39}$ bits $\Rightarrow$ 36 address lines
- **Largest devices:** Up to 512 Gb organized as 32G$\times$16b or 128G$\times$4b
- 128G words require 38 address lines

Problem: A very large address decoder with 38 address pins is impractical.

Solution: Address multiplexing

- Use wider rows with multiple words per row
- Give row address first, then column address
- Only half the number of address pins required

Address Multiplexing

Addresses are multiplexed to reduce the number of required pins.

Example: 1 Mbit DRAM

- Total: 1024 rows $\times$ 1024 columns
- Row address: 10 bits
- Column address: 10 bits
- Address select signal determines whether row or column address is present
- Only 10 address pins needed instead of 20

Process:

1. Present row address on address lines
2. Latch row address with RAS (Row Address Strobe)
3. Present column address on same address lines
4. Latch column address with CAS (Column Address Strobe)

DRAM Architecture

DRAM is organized in a rectangular structure for optimization.

Organization:

- **Rows:** Activated by row address
- **Columns:** Selected by column address
- **Controller logic:** Required for multiplexed addresses and refresh cycles
- **Amplifier per column:** One sense amplifier for each column

Example configuration:

- Multiple DRAM chips to build a memory system
- One bit per chip (bit-sliced organization)
- 4 bits per row $\Rightarrow$ 2 column address lines

Refresh Cycles and Amplifiers

The non-square array structure optimizes refresh and power.

Key relationships:

- A DRAM has as many refresh cycles as rows
- The number of refresh cycles determines the time between two refreshes
- A DRAM has as many amplifiers as columns
- The number of amplifiers essentially determines the power dissipation
- Optimization is achieved with a non-square array structure

Example: 1 Mbit DRAM

- 512 rows $\times$ 2048 columns
- 11-bit addressing (row and column)
- Fewer refresh cycles (512 vs 1024)
- More columns (higher power but acceptable)

DRAM Timing and Access Cycles

Single Inline Memory Module (SIMM)

Early memory module standard with asynchronous interface.

Characteristics:

- 30-Pin SIMM configuration
- Asynchronous operation (no clock)
- Word size and number of words determined by module configuration

Read Access Operation

Reading data from DRAM involves multiple steps with specific timing requirements.

Read sequence:

1. Address placed on address bus
2. RAS (Row Address Strobe) activated
3. Row address latched and row activated
4. CAS (Column Address Strobe) activated
5. Column address latched
6. Data amplified and output
7. Data available on data bus

Internal operations:

- Sense amplifiers detect small voltage differences
- Row activation precharges bit lines
- Column multiplexer selects specific word
- Output buffers drive data bus

Refresh Operation

DRAM requires periodic refresh to maintain data integrity.

Refresh process:

- One row refreshed per refresh cycle
- Addressing the row automatically refreshes it
- Occurs before potential read/write operations
- All rows must be refreshed within the retention time (typically 64 ms)
- Refresh can be initiated by external controller or internal logic

Refresh operations consume time and power, reducing effective memory bandwidth.

Asynchronous Read Cycle Timing

Key timing parameters for asynchronous DRAM.

Timing parameters:

- t_{RAC}: RAS Access Time (typical 60 ns) - time from RAS to valid data
- t_{CAC}: CAS Access Time (typical 30 ns) - time from CAS to valid data
- t_{RCD}: RAS to CAS Delay - minimum time between RAS and CAS

The CAS access time is shorter because the row is already activated.

Page Mode Operation

Page mode allows faster access to multiple words in the same row.

Operation:

- Row address given once with RAS
- Multiple column addresses given with CAS
- Each CAS cycle reads one word from the active row
- Significantly faster than full RAS/CAS cycles
- Useful for sequential or burst accesses

Advantages:

- Reduced latency for subsequent accesses
- Better bandwidth utilization
- Lower power consumption per access

Asynchronous Write Cycle Timing

Write operations follow a similar but distinct timing pattern.

Write sequence:

- RAS activated with row address
- CAS activated with column address
- WE (Write Enable) signal activated
- Data placed on data bus
- Data written to selected cell
- Data bus may go to high impedance after write

Refresh Cycle Timing

Dedicated refresh cycles maintain data integrity.

Refresh process:

- RAS activated with row address
- CAS activated simultaneously (CAS before RAS refresh)
- One row refreshed per cycle
- No data transfer occurs
- Internal refresh counter can increment automatically

Fast Page Mode DRAM

Fast Page Mode (FPM) improves on basic page mode operation.

Improvements:

- Faster CAS cycle times
- Pipelined access within a page
- Reduced setup and hold times
- Better suited for burst transfers

Datasheet Exercise: IS41LV16100D

Task: Extract timing parameters from a 16 Mb DRAM (1M $\times$ 16) datasheet.

Find the following parameters:

- t_{RCD} = RAS to CAS Delay Time
- t_{RAC} = RAS Access Time
- t_{CAC} = CAS Access Time
- t_{PC} = Page Cycle Time

Typical values from datasheet:

- These values depend on the speed grade of the device
- Faster speed grades have shorter timing parameters
- Trade-off between speed and cost

Always consult the actual datasheet for precise timing values for your specific device and speed grade.

Access Time vs Cycle Time

Important distinction for memory performance analysis.

Key timing parameters:

- t_{RAS}: RAS Active Time - how long RAS must remain active
- t_{RC}: Row Cycle Time - minimum time between successive row accesses
- t_{RP}: RAS Precharge Time - time needed to precharge before next RAS
- t_{RCD}: RAS to CAS Delay - minimum delay between RAS and CAS

Problem: Long access time for rows because rows, bit lines, and sense amplifiers need to be precharged.

Question: How could we improve the access speed?

The precharge time t_{RP} shown before the next RAS limits performance. This motivates the use of banking and interleaving.

Banking and Interleaving

Memory Banks and Interleaved Operation

Banks allow overlapping of operations to hide latency.

Concept:

- Split memory into multiple independent banks
- Read from one bank while precharging another
- Interleaved operation hides precharge time
- Bank address (BA) selects bank for current command

Benefits:

- Increased effective bandwidth
- Reduced perceived latency
- Better utilization of memory bus

Trade-off:

- Controller becomes more complex
- Requires careful scheduling of bank accesses
- Optimal when access pattern has good locality

DRAM Controller Design

Principle of a DRAM Controller

The DRAM controller bridges the CPU and DRAM devices.

Controller functions:

- Generate DRAM control signals (RAS, CAS, WE)
- Multiplex addresses (row and column)
- Manage refresh cycles
- Arbitrate between processor access and refresh
- Convert internal bus signals to DRAM protocol

Components:

- **Refresh Logic:** Generates periodic refresh requests
- **Refresh Counter:** Tracks which row to refresh next
- **Control Signal Generator:** Creates RAS, CAS, WE timing
- **Address Multiplexer:** Switches between row and column addresses
- **Access/Refresh Arbitration:** Prioritizes CPU and refresh requests

The CPU requires a corresponding controller to convert internal bus signals into DRAM-specific signals.

Commercial SDRAM controller IP cores (like Altera SDRAM Controller IP) provide ready-made solutions that handle all timing and control signal generation automatically.

Synchronous DRAM (SDRAM)

Synchronous Memory Interface

SDRAM adds a clock signal to synchronize all operations.

Key characteristics:

- All signals synchronized to a common clock
- Address and control signals sampled at clock rising edge
- Predictable timing - all operations take fixed number of clock cycles
- Burst access mode for sequential data

Read cycle timing:

- Long latency for first data item in a row
- Short access time for subsequent items in same row
- Row address given first, followed by column addresses

SDRAM Timing Parameters

Key timing values are specified in clock cycles.

Critical parameters:

- t_{RCD} = RAS to CAS Delay (in clock cycles)
- t_{CL} = CAS Latency (in clock cycles)
- t_{RP} = RAS Precharge Time (in clock cycles)

Typical values:

- Access time: 20 to 50 ns
- Clock cycle time: 10 ns at 100 MHz bus (PC100)
- Parameters expressed as number of clock cycles

Advantages of Synchronous Operation

Synchronous interface provides several benefits over asynchronous.

Benefits:

- **Simplified timing:** Address and control signals sampled at rising clock edge
- **Glitch immunity:** Glitches on address/control lines do not cause failures
- **Predictable delays:** RAS-CAS delays are fixed number of clock cycles
- **Fixed access times:** All operations take deterministic time
- **Fixed precharge:** RAS precharge time is fixed number of cycles

The DRAM controller must provide signals synchronous to a common clock, but this simplifies overall system design.

DDR SDRAM Technology

Double Data Rate (DDR) SDRAM

DDR transfers data on both clock edges, doubling effective bandwidth.

Key innovation:

- Data transferred on rising AND falling clock edges
- Doubles data rate without increasing clock frequency
- Same t_{RP}, t_{RCD}, t_{CL} as SDR SDRAM
- Requires differential data strobe signals (DQS)

Advantages:

- Double bandwidth compared to SDR
- Lower power per bit transferred
- Compatible with existing memory architecture

DDR SDRAM Designation

DDR memory modules are designated by bandwidth and timing parameters.

Naming convention:

- PC xxxx a-b-c or DDR yyy a-b-c
- xxxx = Bandwidth in MB/s
- yyy = Transfer rate in MT/s (Mega Transfers per second)
- a = CAS Latency (t_{CL})
- b = RAS-CAS Delay (t_{RCD})
- c = RAS Precharge (t_{RP})

DDR Memory Designation

Example: PC 1600 2-3-3 or DDR200 2-3-3

Interpretation:

- 200 MHz transfer rate (DDR200)
- 1600 MB/s bandwidth (for 64-bit wide DIMM: 200 MT/s × 8 bytes)
- Access time: 50 ns for first access, then 8 bytes every 5 ns
- Timing: 2-3-3 means t_{CL}=2, t_{RCD}=3, t_{RP}=3 clock cycles

An additional fourth value may specify Row-Active-Time (t_{RAS}).

DDR Performance Comparison

Type	Bus Clock	Transfer Rate	Bandwidth	Timing
PC100	100 MHz	100 MT/s	800 MB/s	SDR
DDR-400	200 MHz	400 MT/s	3200 MB/s	CL2-2-2-5
DDR2-800	400 MHz	800 MT/s	6400 MB/s	CL4-4-4-10
DDR3-1600	800 MHz	1600 MT/s	12800 MB/s	CL9-9-9-21

Notes:

- Bandwidth values are for 64-bit wide DIMM modules
- Transfer rate is double the bus clock (DDR)
- Higher frequencies require longer latencies in clock cycles

Timing Parameter Details

Understanding what the timing numbers mean:

Parameter definitions:

- **CAS Latency (t_{CL}):** Number of clock cycles waiting time for addressing a non-directly following memory cell within the same row
- **RAS to CAS Delay (t_{RCD}):** Number of clock cycles waiting time between row addressing and column addressing
- **RAS Precharge Delay (t_{RP}):** Number of clock cycles waiting time between two consecutive row addressings
- **Row-Active-Time (t_{RAS}):** Number of clock cycles waiting time between two consecutive row addressings of the same row

DDR Timing Examples

Different DDR generations with same absolute timing:

- **DDR-400 CL2-2-2-5:** 200 MHz bus clock
  - t_{CL} = 10 ns, t_{RCD} = 10 ns, t_{RP} = 10 ns, t_{RAS} = 25 ns
- **DDR2-800 CL4-4-4-10:** 400 MHz bus clock
  - t_{CL} = 10 ns, t_{RCD} = 10 ns, t_{RP} = 10 ns, t_{RAS} = 25 ns
- **DDR3-1600 CL9-9-9-21:** 800 MHz bus clock
  - t_{CL} = 11.25 ns, t_{RCD} = 11.25 ns, t_{RP} = 11.25 ns, t_{RAS} = 26.25 ns

Observations:

- Higher clock frequencies require more clock cycles for same absolute time
- Bandwidth doubles with each generation despite similar absolute timings
- DDR3 shows slightly longer absolute latencies due to technology constraints

Actual Timing Examples

Different speed grades have different absolute timings:

- Fast timing: t_{CL} = 10.0 ns
- Medium timing: t_{CL} = 13.5 ns
- Slower timing: t_{CL} = 13.75 ns

Faster timing grades cost more but provide better performance. Always match memory speed to motherboard capabilities.

DDR2 SDRAM

DDR2 doubles the data rate again through internal prefetch.

Key features:

- 4-bit prefetch (vs 2-bit in DDR)
- Internal clock runs at half the I/O rate
- Example: 100 MHz internal, 200 MHz bus, 400 MBit/s transfer
- Lower operating voltage: 1.8V (vs 2.5V for DDR)
- Improved signal integrity with on-die termination

DDR3 SDRAM

DDR3 continues the evolution with even higher speeds.

Key features:

- 8-bit prefetch (vs 4-bit in DDR2)
- Even lower operating voltage: 1.5V (vs 1.8V for DDR2)
- Higher clock frequencies (400 MHz bus for DDR3-1600)
- Improved power efficiency
- Better signal integrity

Common speeds:

- DDR3-800 (PC3-6400): 100 MHz core, 400 MHz I/O, 800 MT/s
- DDR3-1066 (PC3-8500): 133 MHz core, 533 MHz I/O, 1066 MT/s
- DDR3-1333 (PC3-10600): 166 MHz core, 666 MHz I/O, 1333 MT/s
- DDR3-1600 (PC3-12800): 200 MHz core, 800 MHz I/O, 1600 MT/s

DDR3 on DE1-SoC Board

IS42R16320D SDRAM chip:

Determine the following:

- Number of words: ?
- Word size: ?
- Number of bits: ?
- Number of address lines: ?
- Number of banks: ?

Organization analysis:

- Number of rows: ?
- Number of columns: ?

Exercise: Consult the IS42R16320D datasheet to answer these questions.

SDRAM Block Diagram Analysis

Analyze the internal block diagram:

Determine from the diagram:

- Number of banks
- Row address width
- Column address width
- Word width
- Total number of words

Analysis approach:

1. Count the number of independent memory arrays (banks)
2. Determine address bus widths from decoder inputs
3. Identify data bus width
4. Calculate total capacity from organization

Non-volatile SRAM (nvSRAM)

Combines SRAM speed with non-volatile storage.

Key features:

- Speed comparable to SRAM
- Data retained without power
- Each SRAM cell has a Flash or other NV cell alongside
- NV contents updated regularly during operation
- Upon power loss, modified SRAM cells stored in NV cells
- Backup powered by capacitor during power-down

Applications:

- Critical data storage
- Fast access with data persistence
- Replacing battery-backed SRAM

PacMan Project

Introduction to the PacMan Game Architecture

Project Overview

The PacMan project demonstrates the integration of a processor (HPS) with programmable logic (FPGA) to create a classic arcade game. The project focuses on:

- Goal definition and system architecture
- Partitioning tasks between processor and FPGA
- Understanding VGA graphics principles
- Character-based graphic generation in FPGA

System Overview

System Architecture

The PacMan system consists of several interconnected components:

- Hard Processor System (HPS) running the game logic
- FPGA fabric handling graphics generation
- Video memory for frame buffer storage
- VGA controller for display output

Software and Programmable Logic Partitioning

Hardware-Software Partitioning

The system is divided into distinct hardware and software components:

FPGA Components:

- `pacman_top.vhd` - Top-level entity
- `graphic_controller.vhd` - VGA timing and display control
- Video Memory - Frame buffer storage
- Clock management (50 MHz input clock)

Software Components:

- Game logic running on HPS
- Character position management
- Collision detection
- Score tracking

VGA Graphics Principles

VGA Resolution and Timing

VGA (Video Graphics Array) is a display standard that defines how images are rendered on a screen through synchronized horizontal and vertical scanning.

Key Parameters:

- Horizontal pixels per line: H pixels/line
- Vertical lines per frame: V lines/frame
- Each pixel is composed of Red, Green, and Blue (RGB) components
- Scanning occurs left-to-right, top-to-bottom

CRT Display Technology

CRT (Cathode Ray Tube) Operation

Traditional CRT displays use electron beams to create images:

Components:

- **Cathode:** Separate electron beams for Red, Green, and Blue
- **Deflection Coil (Yoke):** Magnetically steers beam in left-to-right, top-to-bottom pattern (separate H and V coils)
- **Shadow Mask:** Ensures R beam only illuminates R pixels, etc.
- **Phosphor Screen:** Emits light when excited by electron beam
- **Anode:** Accelerates electrons toward the screen

Operation: The intensity of the electron beam determines the brightness of each pixel. The beam is continuously scanned across the screen in a raster pattern.

Although modern displays use LCD or LED technology, understanding CRT operation helps explain VGA timing requirements that are still used today.

VGA Scanning Pattern

Raster Scanning

VGA displays use a raster scanning method:

- **Horizontal Scan:** Beam moves left-to-right across 1024 pixels
- **Horizontal Return:** Beam returns to left edge (blanking period)
- **Vertical Scan:** After 768 lines, beam returns to top (vertical return)
- **Frame Rate:** Complete cycle repeats continuously (typically 60 Hz)

VGA Timing Specifications

Horizontal Video Timing for 1024 $\times$ 768 Resolution

Horizontal Timing Parameters

Parameter	Time	Pixel Clocks
Pixel Clock	15.39 ns	65 MHz
HSYNC Pulse Width (T_{pw})	2.092 μs	136 pixclk
Back Porch (T_{bp})	369.2 ns	24 pixclk
Visible Video (T_{disp})	15.754 μs	1024 pixclk
Front Porch (T_{fp})	2.462 μs	160 pixclk
Total Line Time	**20.66 μs**	**1344 pixclk**

Horizontal Sync Signal

The HSYNC signal controls horizontal beam positioning:

- **HSYNC Pulse:** Triggers horizontal retrace (136 clocks)
- **Back Porch:** Blanking after HSYNC before visible video (24 clocks)
- **Visible Video:** Active pixel data display (1024 clocks)
- **Front Porch:** Blanking before next HSYNC (160 clocks)

Vertical Video Timing for 1024 $\times$ 768 Resolution

Vertical Timing Parameters

Parameter	Time	Lines
One Line Period	20.66 μs	1 HSYNC
VSYNC Pulse Width	124.0 μs	6 lines
Back Porch	599.4 μs	29 lines
Visible Video	15.86 ms	768 lines
Front Porch	61.97 μs	3 lines
Total Frame Time	**16.66 ms**	**806 lines**

Vertical Sync Signal

The VSYNC signal controls vertical beam positioning:

- **VSYNC Pulse:** Triggers vertical retrace (6 lines)
- **Back Porch:** Blanking after VSYNC (29 lines)
- **Visible Video:** Active lines containing pixel data (768 lines)
- **Front Porch:** Blanking before next VSYNC (3 lines)
- **Frame Rate:** $\frac{1}{16.66\text{ ms}} \approx 60$ Hz

Character-Based Graphics in FPGA

Character-Based Display System

The PacMan project uses character-based graphics for efficient FPGA implementation:

Advantages:

- Reduced memory requirements
- Simplified graphics controller logic
- Easy character animation
- Efficient for tile-based games

System Components:

- Character ROM - stores character patterns
- Video Memory - stores which character appears at each position
- Display Controller - reads memory and generates video signal

The PacMan project demonstrates practical application of VGA timing principles, character-based graphics, and hardware-software partitioning in an FPGA-based system. The modular architecture allows for efficient implementation and easy modification of game elements.

General Purpose Input/Output (GPIO)

Learning Objectives

Course Goals

After this lecture and lab exercise, students will be able to:

- Sketch the logic circuit of GPIO pins
- Explain the function of a configurable GPIO pin
- Configure GPIO pins in an SoC design

Agenda

This lecture covers three main topics:

- Applications and different modes of GPIO pins
- GPIO Internals: schematic and capabilities
- GPIO in a Platform Design

Typical Applications for GPIO

GPIO Applications

GPIO (General Purpose Input/Output) pins are versatile digital interfaces used for:

- LED control and status indicators
- Button and switch inputs
- Sensor interfacing
- Simple protocol communication
- External device control
- Interrupt generation

GPIO Operating Modes

GPIO Modes

- **Output Mode:** Pin drives signal to external circuit
- **Input Mode:** Pin reads signal from external circuit
- **Bidirectional (Tristate):** Pin can switch between input and output modes

GPIO Internal Architecture

GPIO Hardware Components

The GPIO pin structure includes several configurable elements:

Core Components:

- **Input Buffer:** Conditions incoming signals for the FPGA core
- **Output Buffer:** Drives signals to external pins with programmable strength
- **Input Register:** Synchronizes and stores input data
- **Output Register:** Holds output data to be driven
- **Output Enable (OE) Register:** Controls direction (input/output)

Programmable Features:

- Programmable driver strength
- Programmable slew rate control
- On-chip termination
- Internal pull-up resistor
- Bus-hold circuit
- Delay control
- Open-drain output option

Output Enable Logic

The Output Enable (OE) signal controls pin direction:

- `OE = 1`: Output mode - pin drives signal
- `OE = 0`: Input mode - pin is high-impedance (tristate)

All GPIO features are configurable through the Quartus II/Prime software during FPGA compilation. These settings optimize signal integrity and power consumption for specific applications.

GPIO Hardware Features

Key Hardware Capabilities

- **Input and Output Buffers:** Provide electrical interface with triggers and drivers
- **Registers:** Synchronize data transfer (Input, Output, and Output Enable registers)
- **Driver Strength Control:** Adjustable output current capability
- **Slew Rate Control:** Controls signal transition speed to reduce EMI
- **On-chip Termination:** Provides impedance matching without external components
- **Pull-Up Resistor:** Internal weak pull-up for floating inputs
- **Bus-Hold Circuit:** Maintains last logic level when input is floating

FPGA vs HPS GPIO Architecture

I/O Pin Assignment in SoC Devices

Pins on an SoC device are dedicated to either FPGA fabric or Hard Processor System (HPS):

Physical Organization:

- **FPGA I/O:** Located around FPGA fabric perimeter
- **HPS Column I/O:** Dedicated vertical columns for HPS
- **HPS Row I/O:** Dedicated horizontal rows for HPS

Boot Requirements:

- CPU requires certain pins during boot process → fixed assignment
- Some peripheral functions need specific circuitry (e.g., USB only on specific pins)
- HPS peripherals are directly attached to system bus

FPGA I/O Characteristics

- Attached to FPGA fabric
- Controlled after fabric configuration
- Used as GPIO or peripheral pins from FPGA
- Can be controlled by HPS via bridges

HPS I/O Characteristics

- Available before fabric configuration
- Used for HPS peripherals (Ethernet, USB, SD, SPI, CAN)
- Used for GPIO from HPS
- Can be used from FPGA as "Loan I/O"

HPS Pin Function Selection

The Platform Designer configuration tool allows selection of HPS pin functions:

- **FPGA Interfaces:** Bridges and connections to fabric
- **Peripheral Pins:** Ethernet, USB, SD, UART, I2C, etc.
- **HPS Clocks:** System and peripheral clock inputs
- **SDRAM:** DDR memory interface pins

Note: Dedicated pin locations limit configuration options. Some functions can only be assigned to specific pins due to internal circuitry requirements.

GPIO in Platform Design

Configuring GPIO in Platform Designer

Step 1: Add GPIO IP Core

1. Open Platform Designer in Quartus
2. Add "PIO (Parallel I/O)ÏP core from library
3. Configure data width (1-32 bits)
4. Set direction (input, output, or bidirectional)

Step 2: Configure Parameters

- Set data width to match application requirements
- Choose input/output/bidirectional mode
- Enable/disable interrupt generation
- Set edge capture capabilities if needed
- Configure reset value for output pins

Step 3: Connect to System

1. Connect to system clock and reset
2. Connect Avalon-MM slave interface to system interconnect
3. Export external connections to top-level
4. Assign base address for register access

Step 4: Generate and Integrate

1. Generate HDL from Platform Designer
2. Instantiate system in top-level VHDL/Verilog
3. Assign external pins in Pin Planner
4. Compile design

GPIO Register Interface

GPIO cores typically provide these registers:

- **Data Register:** Read input values or write output values
- **Direction Register:** Set pin direction (bidirectional mode only)
- **Interrupt Mask:** Enable/disable interrupts per pin
- **Edge Capture:** Record edge events on input pins

GPIO Software Access

Task: Control an LED and read a button using GPIO

C Code Example:

```c
// Define GPIO base addresses
#define LED_BASE 0xFF200000
#define BUTTON_BASE 0xFF200010

// LED control
volatile unsigned int *led_ptr = (unsigned int *)LED_BASE;
*led_ptr = 0x01;  // Turn on LED 0

// Button reading
volatile unsigned int *button_ptr = (unsigned int *)BUTTON_BASE;
unsigned int button_state = *button_ptr & 0x01;  // Read button 0
```

Explanation:

- Use volatile pointers to prevent compiler optimization
- Access GPIO registers through memory-mapped addresses
- Bitwise operations allow individual pin control
- Base addresses defined in Platform Designer

GPIO provides a simple but powerful interface for connecting external devices to an SoC. Proper configuration in Platform Designer and Pin Planner ensures reliable operation and optimal signal integrity.

JTAG - Joint Test Action Group

Overview of IC Testing

Agenda

This lecture covers:

- Overview of IC testing approaches
- Functional vs Structural testing
- Introduction to JTAG standard
- Core Scan methodology
- Boundary Scan technique
- Boundary Scan Register architecture
- TAP (Test Access Port) Controller

General Test Procedure

Standard Test Methodology

Testing integrated circuits follows three fundamental steps:

1. Set hardware to a defined state
2. Execute one or multiple clock cycles
3. Read out the resulting state and verify

Critical Questions:

- How do we set the hardware to a defined state?
- How can we read out the current state without modifying it?

Functional Testing Challenges

Simple Logic IC Testing

Characteristics:

- Limited number of flip-flops
- Limited number of states
- All states can be tested
- Functional test is feasible

Complex IC Testing

Challenges:

- Many test cases needed
- Time-consuming execution
- Time-consuming development
- Test coverage problematic
- Debugging support lacking

As IC complexity increased, functional test development time grew from 3-6 months (1977-1980) to 12-24 months (1987-1990). This trend necessitated alternative testing approaches.

Shift Register Fundamentals

Shift Register Operation

A shift register chains flip-flops together, allowing serial data input and parallel state observation:

- Data shifts through on each clock edge
- Enables serial loading of test patterns
- Allows serial readout of circuit state
- Forms basis for scan chain testing

JTAG Standard Overview

JTAG - Joint Test Action Group

Standardization:

- Standardized in 1990
- IEEE 1149.1-1990 standard
- Purpose: Testing of parts and components
- Enables boundary scan testing
- Provides debug access

Key Features:

- Serial test interface (4-5 pins)
- Standardized test access port
- Boundary scan capability
- Device identification
- Program/debug support

Core Scan Methodology

Core Scan Technique

Core scan converts internal flip-flops into a shift register for testing:

Normal Mode:

- Flip-flops operate in functional mode
- Data flows through combinational logic

Test Mode:

- Flip-flops form scan chain
- Test patterns shifted in serially
- Results shifted out serially
- Provides observability and controllability

Boundary Scan

Boundary Scan Architecture

Boundary scan places test cells between core logic and I/O pins:

Capabilities:

- Test interconnections between ICs
- Program internal devices (FPGA configuration)
- Debug embedded systems
- Observe/control I/O pins

Boundary Scan Register:

- Shift register around IC perimeter
- One cell per I/O pin
- Can capture, shift, and update data
- Provides pin-level access

TAP Controller

Test Access Port (TAP)

TAP Signals:

- **TCK:** Test Clock - synchronizes test operations
- **TMS:** Test Mode Select - controls TAP state machine
- **TDI:** Test Data In - serial data input
- **TDO:** Test Data Out - serial data output
- **TRST:** Test Reset (optional) - asynchronous reset

JTAG Testing Procedure

Step 1: Access JTAG Interface

- Connect to TCK, TMS, TDI, TDO pins
- Initialize TAP controller

Step 2: Select Test Operation

1. Navigate TAP state machine using TMS
2. Select instruction register or data register
3. Load instruction (e.g., EXTEST, SAMPLE, BYPASS)

Step 3: Execute Test

1. Shift test data through TDI
2. Capture internal/boundary scan data
3. Shift results out through TDO
4. Verify expected vs actual results

JTAG Applications

Common Uses:

- **Manufacturing Test:** Verify PCB assembly and interconnects
- **FPGA Configuration:** Program configuration memory
- **Firmware Programming:** Flash memory programming
- **Debug Access:** Access processor debug features
- **Boundary Scan:** Test IC-to-IC connections

JTAG has become the standard interface for device programming, debugging, and testing in modern embedded systems. Its serial nature minimizes pin count while providing comprehensive test access.

Timing Analysis Part 1

Introduction to Timing Analysis

Learning Objectives

After this lecture, students will be able to:

- Understand timing constraints in digital systems
- Analyze setup and hold time requirements
- Calculate maximum clock frequency
- Identify timing violations
- Apply timing constraints in FPGA designs

Flip-Flop Timing Parameters

Critical Timing Parameters

Every flip-flop has characteristic timing requirements:

Setup Time (t_{su}):

- Minimum time data must be stable BEFORE clock edge
- Ensures proper data capture
- Typically 50-200 ps for modern FPGAs

Hold Time (t_h):

- Minimum time data must remain stable AFTER clock edge
- Prevents metastability
- Typically 0-100 ps for modern FPGAs

Clock-to-Q Delay (t_{co}):

- Time from clock edge to output change
- Represents flip-flop propagation delay
- Typically 100-400 ps for modern FPGAs

$$T_{clk} \geq t_{co} + t_{logic} + t_{routing} + t_{su}$$

where:

- T_{clk} = Clock period
- t_{co} = Clock-to-output delay
- t_{logic} = Combinational logic delay
- $t_{routing}$ = Routing delay
- t_{su} = Setup time

Setup Time Analysis

Setup Time Constraint

The setup time constraint ensures data arrives at the destination flip-flop with sufficient margin before the clock edge.

Setup Equation:

$$t_{arrival} \leq T_{clk} - t_{su}$$

where $t_{arrival}$ is the time when data reaches the flip-flop input.

Setup Time Calculation

Given:

- Clock frequency: 100 MHz ($T_{clk} = 10$ ns)
- $t_{co} = 0.5$ ns
- $t_{logic} = 4$ ns
- $t_{routing} = 2$ ns
- $t_{su} = 0.3$ ns

Solution:

Data arrival time:

$$t_{arrival} = t_{co} + t_{logic} + t_{routing} = 0.5 + 4 + 2 = 6.5 \text{ ns}$$

Required time (setup constraint):

$$t_{required} = T_{clk} - t_{su} = 10 - 0.3 = 9.7 \text{ ns}$$

Setup slack:

$$\text{Slack} = t_{required} - t_{arrival} = 9.7 - 6.5 = 3.2 \text{ ns}$$

Result: Setup constraint is MET with 3.2 ns positive slack.

Hold Time Analysis

Hold Time Constraint

The hold time constraint ensures data remains stable at the flip-flop input for sufficient time after the clock edge.

Hold Equation:

$$t_{arrival} \geq t_h$$

Critical: Hold violations cannot be fixed by changing clock frequency!

Hold time violations typically occur when:

- Clock skew is too large
- Fast data path between flip-flops
- Insufficient routing delay
- Clock and data paths are unbalanced

Clock Skew

Clock Skew

Clock skew is the difference in arrival time of the clock signal at different flip-flops.

Types:

- **Positive Skew:** Clock arrives later at destination FF
- **Negative Skew:** Clock arrives earlier at destination FF

Effects:

- Positive skew: Helps setup, hurts hold
- Negative skew: Hurts setup, helps hold

$$T_{clk,min} = t_{co} + t_{logic} + t_{routing} + t_{su} - \text{skew}$$

Maximum Clock Frequency

Determining Maximum Clock Frequency

Step 1: Identify Critical Path

- Find path with longest delay
- Sum all delays: $t_{co} + t_{logic} + t_{routing}$

Step 2: Apply Setup Constraint

$$T_{clk,min} = t_{critical\_path} + t_{su}$$

Step 3: Calculate Maximum Frequency

$$f_{max} = \frac{1}{T_{clk,min}}$$

Step 4: Verify Hold Constraints

- Check shortest path
- Ensure $t_{co,min} + t_{logic,min} + t_{routing,min} \geq t_h$

Key Timing Concepts

Concept	**Impact**
Setup Time	Limits maximum frequency
Hold Time	Independent of frequency
Clock Skew	Affects both setup and hold
Critical Path	Determines f_{max}
Slack	Timing margin (pos/neg)

Modern FPGA tools perform static timing analysis (STA) to automatically identify timing violations and report slack for all paths. Always verify timing reports before deploying designs.

Timing Analysis Part 2

Advanced Timing Concepts

Advanced Topics

This lecture extends timing analysis with:

- Multi-cycle paths
- False paths
- Clock domain crossing
- Timing exceptions
- Advanced clock constraints
- Timing optimization techniques

Multi-Cycle Paths

Multi-Cycle Path

A multi-cycle path is a data path that intentionally takes more than one clock cycle to complete.

Characteristics:

- Relaxes timing constraints
- Allows slower logic operations
- Requires explicit constraint specification
- Must ensure data stability

Constraint:

$$T_{path} \leq N \cdot T_{clk} - t_{su}$$

where N is the number of clock cycles.

Multi-Cycle Path Application

Scenario: A multiplication operation takes 15 ns to complete

Given:

- Clock period: $T_{clk} = 10$ ns (100 MHz)
- Multiplication delay: 15 ns
- Single-cycle would require: $f_{max} = 1/15$ ns $= 66.7$ MHz

---

Solution: Define as 2-cycle path

- Available time: $2 \times 10 = 20$ ns
- Path delay: 15 ns
- Slack: $20 - 15 = 5$ ns (PASS)
- Maintains 100 MHz clock frequency

Constraint: Must tell timing analyzer about multi-cycle requirement

False Paths

False Path

A false path is a timing path that can never be sensitized in actual circuit operation.

Examples:

- Asynchronous clock domain crossings (handled separately)
- Test/debug paths not used in normal operation
- Mutually exclusive paths
- Configuration paths

Purpose:

- Reduce unnecessary optimization effort
- Improve tool runtime
- Avoid over-constraining design

Clock Domain Crossing (CDC)

Clock Domain Crossing Challenges

When signals cross between different clock domains, special care is required:

Problems:

- Metastability risk
- Timing analysis complexity
- Data corruption possibility
- Setup/hold violations likely

Solutions:

- Synchronizer circuits (2-FF synchronizer)
- Handshake protocols
- Asynchronous FIFOs
- Gray code counters

Two-Flip-Flop Synchronizer

The standard solution for single-bit CDC:

Structure:

- Two flip-flops in series
- Both clocked by destination domain clock
- First FF may go metastable
- Second FF resolves metastability

MTBF (Mean Time Between Failures):

$$\text{MTBF} = \frac{e^{t_{res}/(.\tau)}}{f_{clk} \cdot f_{data} \cdot T_0}$$

where:

- t_{res} = Resolution time available
- $.\tau$ = FF time constant
- T_0 = Window of vulnerability

Implementing Safe CDC

For Single-Bit Signals:

1. Use 2-FF synchronizer in destination domain
2. Constrain as false path
3. Verify MTBF is acceptable
4. Add timing constraints to prevent optimization

For Multi-Bit Signals:

1. Use Gray code if counter/state
2. Use handshake protocol with ready/acknowledge
3. Use asynchronous FIFO for data streams
4. Never synchronize multi-bit buses directly!

For Control Signals:

1. Pulse stretching in source domain
2. Edge detection in destination domain
3. Handshake confirmation

Timing Constraints in Quartus

Synopsys Design Constraints (SDC)

Modern FPGA tools use SDC format for timing constraints:

Common Constraints:

- `create_clock` - Define clock sources
- `create_generated_clock` - Define derived clocks
- `set_input_delay` - Input timing relative to clock
- `set_output_delay` - Output timing relative to clock
- `set_false_path` - Disable timing analysis
- `set_multicycle_path` - Multi-cycle paths
- `set_max_delay` - Maximum delay constraint
- `set_min_delay` - Minimum delay constraint

SDC Constraint Examples

```
# Define primary clocks
create_clock -name clk_50 -period 20.000 [get_ports clk_50]
create_clock -name clk_100 -period 10.000 [get_ports clk_100]

# Define generated clock (PLL output)
create_generated_clock -name clk_pll \
    -source [get_pins pll_inst|inclk[0]] \
    -multiply_by 2 \
    [get_pins pll_inst|clk[0]]

# Input delays (relative to external clock)
set_input_delay -clock clk_50 -max 5.0 [get_ports data_in*]
set_input_delay -clock clk_50 -min 2.0 [get_ports data_in*]

# Output delays
set_output_delay -clock clk_50 -max 8.0 [get_ports data_out*]
set_output_delay -clock clk_50 -min 1.0 [get_ports data_out*]

# False path for asynchronous reset
set_false_path -from [get_ports rst_n]

# Multi-cycle path (2 cycles for multiply operation)
set_multicycle_path -from [get_registers mult_stage1*] \
    -to [get_registers mult_result*] -setup 2
set_multicycle_path -from [get_registers mult_stage1*] \
    -to [get_registers mult_result*] -hold 1

# CDC constraints
set_false_path -from [get_clocks clk_50] \
    -to [get_clocks clk_100]
```

Timing Closure Strategy

Iterative Process:

1. Apply timing constraints
2. Run compilation
3. Analyze timing reports
4. Identify violations
5. Optimize critical paths:
   - Pipeline long paths
   - Reduce logic levels
   - Use faster resources
   - Add registers
   - Optimize placement
6. Re-compile and verify
7. Repeat until timing met

Proper timing constraints are essential for reliable FPGA designs. Always verify that all clocks are constrained and timing reports show positive slack on all paths. Unconstrained paths may work in one compilation but fail in another.

PLL - Phase Locked Loop

Introduction to Clock Generation

Learning Objectives

After this lecture, students will be able to:

- Understand PLL operation principles
- Configure PLLs in Intel/Altera FPGAs
- Generate multiple clock domains
- Implement clock domain crossing safely
- Apply phase shifting techniques
- Calculate PLL parameters

Clock Requirements in Digital Systems

Need for Clock Management

Modern digital systems require sophisticated clock generation:

Requirements:

- Multiple clock frequencies
- Phase-aligned clocks
- Low jitter and skew
- Programmable frequency division/multiplication
- Clock domain isolation
- Dynamic frequency scaling

Challenges:

- Input clocks may be non-standard frequencies
- Different IP cores need different clock rates
- External interfaces require specific frequencies
- Power management needs variable frequencies

PLL Fundamentals

Phase Locked Loop (PLL)

A PLL is an electronic control system that generates an output signal with a phase related to its input signal.

Key Components:

- **Phase Detector (PD):** Compares input and feedback phases
- **Loop Filter:** Smooths phase detector output
- **Voltage Controlled Oscillator (VCO):** Generates output clock
- **Feedback Divider:** Divides VCO output for comparison

Operation:

1. Phase detector compares input clock to feedback
2. Error signal adjusts VCO frequency
3. System locks when output phase matches input
4. Frequency multiplication/division achieved through dividers

$$f_{out} = f_{in} \times \frac{M}{N \times C}$$

where:

- f_{in} = Input clock frequency
- M = Feedback multiplier (multiply counter)
- N = Input divider (pre-scale counter)
- C = Output divider (post-scale counter)

PLL Architecture in Intel FPGAs

Intel FPGA PLL Features

Intel/Altera FPGAs provide sophisticated PLL IP cores:

Capabilities:

- Multiple output clocks (typically 5-9 outputs)
- Independent frequency for each output
- Phase shift adjustment (0-360°)
- Duty cycle control
- Clock switchover for redundancy
- Loss-of-lock detection
- Dynamic reconfiguration

PLL Types:

- **Integer PLL:** Integer frequency ratios only
- **Fractional PLL:** Non-integer ratios (higher flexibility)
- **ATX PLL:** High-performance transceiver clocking

VCO Operating Range

The VCO has a specific frequency range:

- $f_{VCO,min}$: Minimum VCO frequency (typically 600 MHz)
- $f_{VCO,max}$: Maximum VCO frequency (typically 1200-1600 MHz)
- Must satisfy: $f_{VCO,min} \leq f_{VCO} \leq f_{VCO,max}$

$$f_{VCO} = f_{in} \times \frac{M}{N}$$

PLL Configuration in Platform Designer

Configuring a PLL

Step 1: Add PLL IP Core

1. Open Platform Designer / IP Catalog
2. Select ÏOPLL Intel FPGA IPör ÄLTPLL"
3. Configure number of clocks needed

Step 2: Set Input Clock

- Specify input clock frequency
- Set input clock duty cycle (typically 50%)

Step 3: Configure Output Clocks For each output clock:

1. Set desired frequency
2. Adjust phase shift if needed
3. Set duty cycle
4. Enable/disable output

Step 4: Verify Parameters

- Check VCO frequency is in valid range
- Verify all outputs meet requirements
- Review jitter specifications

Step 5: Generate and Integrate

1. Generate PLL instance
2. Instantiate in top-level design
3. Connect input clock
4. Route output clocks to consumers
5. Use generated `locked` signal

PLL Configuration Example

Requirements:

- Input clock: 50 MHz
- Output 1: 100 MHz (main system clock)
- Output 2: 25 MHz (slow peripheral clock)
- Output 3: 100 MHz with 90° phase shift

Solution:

Calculate VCO frequency:

$$f_{VCO} = 50 \text{ MHz} \times \frac{M}{N}$$

Choose $M = 24$, $N = 1$:

$$f_{VCO} = 50 \times 24 = 1200 \text{ MHz}\checkmark$$

(in valid range)

Output dividers:

- Output 1: $C_1 = 12 \rightarrow 1200/12 = 100$ MHz
- Output 2: $C_2 = 48 \rightarrow 1200/48 = 25$ MHz
- Output 3: $C_3 = 12$, Phase $= 90°$ $\rightarrow$ 100 MHz shifted

Using PLL Outputs

Locked Signal

The PLL provides a `locked` output signal:

Purpose:

- Indicates PLL has achieved lock
- Clocks are stable and valid
- Used to release system reset

Typical Usage:

```vhdl
-- Reset logic using PLL locked signal
process(clk, pll_locked)
begin
    if pll_locked = '0' then
        system_reset <= '1';
    elsif rising_edge(clk) then
        -- Release reset after PLL locks
        system_reset <= '0';
    end if;
end process;
```

Clock Network Distribution

Global Clock Networks

FPGAs provide dedicated low-skew clock distribution networks:

Features:

- Dedicated routing resources
- Minimal skew across device
- Low jitter
- Balanced tree structure
- Direct connection from PLL outputs

Best Practices:

- Use global clock buffers (GCLK)
- Minimize clock domain crossings
- Route clocks on dedicated networks
- Avoid using clocks as data
- Use clock enable instead of gated clocks

PLL Design Guidelines

Guideline	Reason
Keep VCO in optimal range	Minimize jitter
Use integer ratios when possible	Better phase noise
Minimize output dividers	Reduce jitter accumulation
Use locked signal for reset	Ensure clock stability
Add synchronizers for CDC	Prevent metastability

PLLs are essential for clock management in complex FPGA designs. Proper PLL configuration and usage of the locked signal ensure reliable system operation. Always verify that generated clock frequencies meet timing requirements for all clock domains.