Agenda

- Zynq SoC and MPSoC Architecture
- SDSoC Overview
- Real-life Success
- C/C++ to Optimized System
- Targeting Your Own Platform
- Next Steps
Agenda

- Zynq SoC and MPSoC Architecture
- SDSoc Overview
- Real-life Success
- C/C++ to Optimized System
- Targeting Your Own Platform
- Next Steps
Zynq-7000: The First All Programmable SoC

- Innovative ARM + FPGA architecture on a single die
- Reduce BOM cost by replacing multiple chips with a single Zynq
- Security through single chip solution and secure boot
- Remove off-chip communication bottleneck
- Architecture optimize for power
Zynq®-7000 Architecture

Processor Core Complex
- Dual ARM® Cortex®-A9 MPController
- NEON™ extensions
- Single / double precision floating point support
- Up to 1 GHz operation

High BW Memory
- L1 Cache – 32KB/32KB (per Core)
- L2 Cache – 512KB Unified
- On-Chip Memory of 256KB
- MultiPort Memory Controller (DDR3, DDR3L, DDR2, LPDDR2)

State-of-the-art 7 Series Programmable Logic
- 28K-440K logic cells
- 430K-6.6M equivalent ASIC gates
- Hard PCIe Core Gen2x8

Up to 100 Gb/s of BW between PS-PL
- One 64-bit ACP Port
- Four 64-bit HP Ports
- Four 32-bit GP Ports
Success Story 1: Automotive ADAS Platform

- Integrated into a single monolithic device
  - Sensing domain
  - Environmental characterization
  - Decision-making features
Success Story 2: 1080p60 HD Medical Endoscope

- Integrated into a single monolithic device
  - Camera unit control
  - 1080p60 image processing
  - Hardware acceleration of complex video analytics

![Diagram showing existing and Xilinx Zynq All Programmable SoC solutions for HD medical endoscope](image-url)
Industry Trends and Customer Challenges

Packet Processing & Transport
- > 400G OTN
- Video Over IP
- Software Defined Networks
- Network Function Virtualization

Wireless
- LTE Advanced
- Cloud-RAN
- Early 5G
- Heterogeneous Wireless Networks

Video and Vision
- 8K/4K Resolution
- Immersive Display
- Augmented Reality
- Video Analytics

Cloud and Data Center
- Acceleration
- Big Data
- Software Defined Data Center
- Public and Private Cloud

Industrial IoT
- Machine to Machine
- Sensory Fusion
- Industry 4.0
- Cyber-Physical
- Embedded Vision
- Cyber-Physical

Performance & Power Scalability
System Integration & Intelligence
Security, Safety & Reliability
Introducing UltraScale+™ MPSoC

- 2-5X System Performance/Watt
- Most System Integration & Intelligence
- Highest Levels of Security and Safety
Zynq® UltraScale+ MPSoC System Features

- **ARM Cortex A53**
  - Application Processors
  - 64-bit Quad-Core

- **ARM Cortex R5**
  - Real-Time Processors
  - 32-bit Dual-Core

- **Graphics Processor**
  - ARM Mali-400MP2

- **UltraScale+™ FPGA Logic**
  - UltraRAM, PCIe Gen4, 100G Ethernet, AMS

- **Video CODECs**
  - H.265 HEVC
  - 8Kx4K (15fps)
  - 4Kx2K (60fps)

- **Power Management**
  - Power Gating & Power Islands

- **Security**
  - Information Assurance, Trust, Anti-Tamper, TrustZone
  - Key & Vault Management

- **Safety & Reliability**
  - System Isolation & Error Mitigation, Lockstep

- **Processing System**
  - Application Processing
  - Real-Time Processing
  - Graphics Processing
  - Power Management
  - Memory
  - Safety & Security

- **Programmable Logic**
  - UltraRAM
  - AMS
  - Video Codec
  - Integrated Blocks (PCIe, Ethernet)
  - Transceivers

- **General & High-Speed Connectivity**
Zynq® UltraScale+™ MPSoC Applications

1 Terabyte OTN Switching
800G MAC-Interlaken Bridge
800G Data Center Interconnect
Mobile Backhaul
  1 GHz eBand Modem & Packet Processing
Mobile Backhaul
  112 MHz PtP MWR Modem & Packet Processing
Test & Measurement Instrumentation
24-Channel Radar
  (Beamformer + Pulse Compressor + Doppler Filter)

8x8 100 MHz LTE Remote Radio Head
Dual-Channel Battery-Powered Public Safety & Military Mobile Radios
Camera-Based Automotive Driver’s Assist Systems (ADAS)
4K Broadcast Cameras
Solid State Drives (SSDs) for Data Center
Video Conferencing
High-Performance Scalable Programmable Logic Controllers (PLCs)
Agenda

- Zynq SoC and MPSoc Architecture
- SDSoC Overview
- Real-life Success
- C/C++ to Optimized System
- Targeting Your Own Platform
- Next Steps
Scaling Productivity with Technology Advancement

- **Ease of Development**
  - CPU
  - GPU
  - ARM SoCs & DSPs
  - Zynq SoC & MPSoC

- **Performance / Watt & ‘Any to Any’ Connectivity**
  - Zynq SoC & MPSoC
  - VIVADO HLS
  - VIVADO Environment
  - SDSoc Environment
Typical Zynq Development Flow

APP()
{
funcA();
funcB();
funcC();
}

HW-SW partition?

funcA

funcB, funcC

HW-SW Connectivity?

Datamover
PS-PL interfaces
SW drivers

funcA

Processing System (PS)

Programming Logic (PL)

funcB, funcC

Explore optimal architecture
Before SDSoC: Connectivity Exploration

Need to modify multiple levels of design entry
Before SDSoC: HW-SW Partition Exploration

Involving multiple discipline to explore architecture
After SDSoC:

- C/C++
- C
- IPI project
- HLS Verilog, VHDL
- HW-SW partition spec
After SDSoC:

- Auto-generate SW drivers and HW connectivity
After SDSoC:

- Auto-generate SW drivers and HW connectivity
- Select hardware accelerator functions in GUI or via command line and C-callable libraries

C/C++

```c
func1(); <-SW
cfunc2(); <-HW
func3(); <-HW
```
After SDSoC:

- Auto-generate SW drivers and HW connectivity
- Select hardware accelerator functions in GUI or via command line and C-callable libraries
- HW function calls in application C/C++ code defines the hardware / software partition

C/C++

```c
func1(); <-SW
func2(); <-HW
func3(); <-HW
```
After SDSoC: Automatic System Generation

C/C++ to running system in hours, days
Base Platform

- **Platform**: base HW system, OS, bootloaders, file system, libraries
  - Processing system, memory interfaces, custom I/O, other subsystems, …
Complete End-to-End Flow

SDSoC

C/C++ Application

Generated

Application
Driver

AXI Bus
Connectivity

Interface IPs

IP
IP
IP
IP

Application
Driver

Application
Driver

Interface IPs

© Copyright 2015 Xilinx
A Complete Software Development Environment

Proven Xilinx tools in the backend

C/C++ Application

Estimator  Compiler  Debugger  Profiler
SDSoC’s Software Programming Experience

» User selects C/C++ functions to accelerate in programmable logic (PL)
» C/C++ system compiler and linker (CLI)
» Easy to use Eclipse IDE
» Optimized HLS libraries
» Support for target Linux, bare metal, FreeRTOS
SDSoC’s System Level Profiling

- **Rapid system estimation**
  - Pre-RTL synthesis, place & route
  - Fast design feedback on accelerator network

- **Automated performance measurement**
  - Runtime measurement of cache, memory, and bus utilization
  - Combined HW-SW event trace of accelerator network
Full System Optimizing Compiler

- **Full system from C/C++**
  - Optimizing programmable logic fabric cross-compiler (HLS)
  - Automatic data motion network inference
  - Application-specific system optimized for latency and throughput/
  - User override via C/C++ pragmas
## Available SDSoC Platforms

### Video Platforms (Externally Provided)

<table>
<thead>
<tr>
<th>Board Name &amp; Description</th>
<th>I/O enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZC702 + HDMI IO FMC</td>
<td>HDMI in, HDMI out, PS DDR</td>
</tr>
<tr>
<td>ZC706 + HDMI IO FMC</td>
<td>HDMI in, HDMI out, PS DDR</td>
</tr>
<tr>
<td>Smart Vision Development Kit (SVDK)</td>
<td>Camera in, GigEV out, PS DDR</td>
</tr>
<tr>
<td>Atlas-IZ7e + Captiva Carrier Card</td>
<td>GigEV in, HDMI out, PS DDR</td>
</tr>
<tr>
<td>MIAMI</td>
<td>PS DDR</td>
</tr>
<tr>
<td>Zing2 + HDMI IO FMC</td>
<td>HDMI IN, HDMI OUT, GPIO, PS, DDR3</td>
</tr>
<tr>
<td>Snowleo SVC</td>
<td>CMOS IN, HDMI OUT, GPIO, PS, DDR3</td>
</tr>
<tr>
<td>EMC2-Z7015</td>
<td>PS DDR</td>
</tr>
<tr>
<td>BORA</td>
<td>LVDS Video Out, PS DDR</td>
</tr>
<tr>
<td>BORA Xpress</td>
<td>LVDS Video Out, PS DDR</td>
</tr>
<tr>
<td>ZYBO</td>
<td>HDMI in, VGA out, buttons, switches, LEDs</td>
</tr>
</tbody>
</table>

### Radio Platforms (Externally Provided)

<table>
<thead>
<tr>
<th>Board Name &amp; Description</th>
<th>I/O enabled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atlas-II-Z7x + Mosaic carrier card</td>
<td>ADC, DAC, PS DDR</td>
</tr>
<tr>
<td>ZC706 + AD9361 SDR Systems Development Kit</td>
<td>ADC, DAC, PS DDR</td>
</tr>
</tbody>
</table>
Example 1: Matrix Multiply + Add

```c
main()
{
malloc(A, B, C);
mmult(A, B, D);
madd(C, D, E);
printf(E);
}

madd(inA, inB, out){
    HLS C/C++
}

mmult(inA, inB, out){
    HLS C/C++
}
```

SDSoC Environment

Generated

PS

Application

Driver

AXI Bus

A, B

mmult

D

madd

C

E

datamovers

© Copyright 2015 Xilinx
Example 2: FIR Filter using C-callable HDL IP

```c
main()
{
malloc(A,B,C);
fir_config(A);
fir_run(B,C);
printf(C);
}
```

```c
fir_config(float *coef);
fir_run(float *in, float *out);
```

HDL IP

SDSoC Environment

Generated

PS

Application

Driver

AXI Bus

FIR

datamovers

© Copyright 2015 Xilinx
Example 3: 1080p60 Motion Detection

```c
main()
get_camera(A);
sobel(A, B);
diff(B, C);
...display(out);
```

ZC702 + HDMI FMC Platform

SDSoC Environment

Image processing on the video I/Os via DDR3 memory
Example 4: DDS using direct I/O connection

```c
main()
{
  DDS(freq, out);
  txDAC(out);
}
```

**ADI SDR platform**

**SDSoC Environment**

**Direct I/O connection to the platform DAC**
Agenda

- Zynq SoC and MPSoC Architecture
- SDSoc Overview
- Real-life Success
- C/C++ to Optimized System
- Targeting Your Own Platform
- Next Steps
Object Recognition

```c
main()
{
    get_camera(A);
    auANPRTop (A,B);
    display(out);
}
```

ZC706 + HDMI FMC Platform

SDSoC Environment

Uses HW Optimized OpenCV Libraries
Hardware Optimized OpenCV Libraries

<table>
<thead>
<tr>
<th>Computations</th>
<th>Input processing</th>
<th>Filter</th>
<th>Other</th>
<th>Other</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute difference</td>
<td>Channel combine</td>
<td>Box</td>
<td>Canny edge detection</td>
<td>Histogram of Gradients (HoG)</td>
</tr>
<tr>
<td>Accumulate</td>
<td>Channel extract</td>
<td>Gaussian</td>
<td>Scale/Resize</td>
<td>ORB</td>
</tr>
<tr>
<td>Accumulate squared</td>
<td>Color convert</td>
<td>Median</td>
<td>Warp Affine</td>
<td>SVM (binary)</td>
</tr>
<tr>
<td>Accumulate weighted</td>
<td>Convert bit depth</td>
<td>Sobel</td>
<td>Warp Perspective</td>
<td>OTSU Thresholding</td>
</tr>
<tr>
<td>Arithmetic addition</td>
<td>Table lookup</td>
<td>Custom convolution</td>
<td>Image pyramid</td>
<td>Mean Shift Tracking (MST)</td>
</tr>
<tr>
<td>Arithmetic subtraction</td>
<td>Histogram</td>
<td></td>
<td>Fast corner</td>
<td>LK Optical Flow</td>
</tr>
<tr>
<td>Bitwise: AND, OR, XOR, NOT</td>
<td>Gradient Phase</td>
<td>Dilate</td>
<td>Harris corner</td>
<td></td>
</tr>
<tr>
<td>Pixel-wise multiplication</td>
<td>Min/Max Location</td>
<td>Erode</td>
<td>Remap</td>
<td></td>
</tr>
<tr>
<td>Integral image</td>
<td>Mean &amp; Standard Deviation</td>
<td></td>
<td>Bilateral</td>
<td>Equalize Histogram</td>
</tr>
<tr>
<td>Gradient Magnitude</td>
<td>Thresholding</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Face Detection and Tracking

```c
main()
getCamera(A);
while (FaceFeatures){
    face_detect(A,B);
}
DrawFacial(A,C);
Display;
```

Uses Optimized HDL IP as a C function in SDSoC
Automatically Generated Vivado Design from C/C++ Application
Agenda

- Zynq SoC and MPSoC Architecture
- SDSoC Overview
- Real-life Success
- C/C++ to Optimized System
- Targeting Your Own Platform
- Next Steps
C/C++ to Optimized System

Recommended design flow

1. Cross-compile C/C++ code for ARM CPU
2. Identify hotspots with TCF Profiler
3. Select HW Functions
4. Estimate Performance
5. Optimize accelerator code
6. Optimize data transfers and system parallelism
7. Analyze Performance with event trace
8. Build HW & SW and run on hardware
Example: Matrix Multiplication

- Common compute primitive for many applications, suitable for hardware acceleration
  - $O(n^3)$ time for schoolbook algorithm, $O(n^{2+\gamma})$ for more sophisticated sequential algorithms
  - Can trade space for time, but must inspect $O(n^2)$ elements in DDR

- Problem size: 32 x 32 matrices of float
- Algorithm: schoolbook implementation
Matrix Multiplication

Output element $C_{ij} = A_{i*} \cdot B_{*j}$ (dot product)

\[
\begin{bmatrix}
11 & 12 & 13 \\
14 & 15 & 16 \\
17 & 18 & 19 \\
\end{bmatrix} \times 
\begin{bmatrix}
21 & 22 & 23 \\
24 & 25 & 26 \\
27 & 28 & 29 \\
\end{bmatrix} = 
\begin{bmatrix}
870 & 906 & 942 \\
1086 & 1131 & 1176 \\
1302 & 1356 & 1410 \\
\end{bmatrix}
\]

- $r_{00} = 11 \times 21 + 12 \times 24 + 13 \times 27 = 870$
- $r_{01} = 11 \times 22 + 12 \times 25 + 13 \times 28 = 906$

Etc…
C/C++ to Optimized System

Recommended design flow
– Profiling with TCF Profiler

1. Cross-compile C/C++ code for ARM CPU
2. Identify hotspots with TCF Profiler
3. Select HW Functions
4. Estimate Performance
5. Optimize accelerator code
6. Optimize data transfers and system parallelism
7. Analyze Performance with event trace
8. Build HW & SW and run on hardware
C/C++ to Optimized System

Recommended design flow
- Performance estimation

1. Cross-compile C/C++ code for ARM CPU
2. Identify hotspots with TCF Profiler
3. Select HW Functions
4. Estimate Performance
5. Optimize accelerator code
6. Optimize data transfers and system parallelism
7. Analyze Performance with event trace
8. Build HW & SW and run on hardware
C/C++ to Optimized System

Recommended design flow

- Optimize accelerator microarchitecture using Vivado HLS

- Cross-compile C/C++ code for ARM CPU

- Identify hotspots with TCF Profiler

- Select HW Functions

- Optimize accelerator code

- Optimize data transfers and system parallelism

- Estimate Performance

- Analyze Performance with event trace

- Build HW & SW and run on hardware
very brief

A SDSoC programmer’s introduction to Vivado HLS
HLS as Cross Compiler

SDSoC employs Vivado HLS as programmable logic cross-compiler

- Hardware function source code shared between SDSoC and VHLS
  - Requires data type consistency between VHLS and arm-gcc
- SDSoC automatically creates VHLS projects for synthesized IP blocks
- User can optionally launch HLS GUI from SDSoC
  - Optimize accelerator code
  - Simulate hardware function
Microarchitecture Optimizations

The most important HLS compiler directives are familiar to performance-oriented software programmers

<table>
<thead>
<tr>
<th>Directives and Configurations</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PIPELINE</td>
<td>Reduces the initiation interval by allowing the concurrent execution of operations within a loop or function.</td>
</tr>
<tr>
<td>DATAFLOW</td>
<td>Enables functions and loops to execute concurrently. Avoid at the top-level hardware function.</td>
</tr>
<tr>
<td>INLINE</td>
<td>Inline a function to function hierarchy, enable logic optimization across function boundaries and reduce function call overhead.</td>
</tr>
<tr>
<td>UNROLL</td>
<td>Unroll for-loops to create multiple independent operations rather than a single collection of operations.</td>
</tr>
<tr>
<td>ARRAY_PARTITION</td>
<td>Partition array into smaller arrays or individual registers to increase concurrent access to data and remove block RAM bottlenecks.</td>
</tr>
</tbody>
</table>

Use hardware buffers to improve communication bandwidth between accelerator and external memory

– Copy loops at the function boundary when multiple accesses required and to burst data into local buffers
Loop Unrolling and Pipelining

**Default Behavior**

- The same hardware is used for each iteration of the loop:
  - Small area
  - Long latency
  - Low throughput

- Different hardware is used for each iteration of the loop:
  - Higher area
  - Short latency
  - Better throughput

**Different iterations are executed concurrently:**
- Higher area
- Short latency
- Best throughput

```
... loop: for (i<3; i>=0; i--) {
    if (i==0) {
        acc+=x*c[i];
        shift_reg[i]=x;
    } else {
        shift_reg[i]=shift_reg[i-1];
        acc+=shift_reg[i]*c[i];
    }
} ... 
```
**Loop and Function Pipelining**

**Without Pipelining**
- Initiation Interval = 3 cycles
- Latency = 3 cycles
- Loop Latency = 6 cycles

**With Pipelining**
- Initiation Interval = 1 cycle
- Latency = 3 cycles
- Loop Latency = 4 cycles

**Pipelined loops**
- Combined with array partitioning to achieve $II=1$

```c
void foo(...) {
    op_Read;
    op_Compute;
    op_Write;
}
```

```c
for (index_b = 0; index_b < B_NCOLS; index_b++) {
    #pragma HLS PIPELINE II=1
    float result = 0;
    for (index_d = 0; index_d < A_NCOLS; index_d++) {
        float product_term = in_A[index_a][index_d] * in_B[index_d][index_b];
        result += product_term;
    }
    out_C[result_a * B_NCOLS + index_b] = result;
}
```
Array Partitioning

Partition into multiple memories to increase concurrent access

void mmult_kernel(float in_A[A_NROWS][A_NCOLS], float in_B[A_NCOLS][B_NCOLS], float out_C[A_NROWS*B_NCOLS])
{
    #pragma HLS INLINE self
    #pragma HLS array_partition variable=in_A block factor=16 dim=2
    #pragma HLS array_partition variable=in_B block factor=16 dim=1
    // snip
}
Example: Matrix Multiplication

- Microarchitecture optimizations
  1. Pipeline the dot-product loop with \( \text{II}=1 \) to unroll the inner loop
  2. Add pipelined copy loops to local dual-port BRAMs partitioned for parallel access
C/C++ to Optimized System

Recommended design flow
- Use SDSoC pragmas and memory allocation to influence data mover selection

- Cross-compile C/C++ code for ARM CPU
- Identify hotspots with TCF Profiler
- Select HW Functions
- Estimate Performance
- Optimize data transfers and system parallelism
- Optimize accelerator code
- Analyze Performance with event trace
- Build HW & SW and run on hardware
System optimizations

Data mover inference based on program properties
- Transfer size
- Memory properties: physical contiguity
- Accelerator memory access patterns

Platform interface connectivity based on program properties
- Transfer size
- Memory properties: cacheability

Performance bottlenecks to avoid
- Pointer arithmetic is usually ill-suited for hardware
  - Instead, burst chunks of data into FIFOs or BRAM for accelerator access
- Transferring data through cache when CPU doesn’t touch it
- Transferring cacheable memory through HP ports
Example: Matrix Multiplication

- Microarchitecture optimizations
  1. Pipeline the dot-product loop with II=1 to unroll the inner loop
  2. Add pipelined copy loops to local dual-port BRAMs partitioned for parallel access

- System optimizations
  1. Sequential access pragma
  2. Allocate buffers in physically contiguous memory for most efficient DMA (axidma_simple)
C/C++ to Optimized System

Recommended design flow
- Use event tracing to analyze performance of accelerators and data motion network

1. Cross-compile C/C++ code for ARM CPU
2. Identify hotspots with TCF Profiler
3. Select HW Functions
4. Estimate Performance
5. Optimize accelerator code
6. Optimize data transfers and system parallelism
7. Analyze Performance with event trace
8. Build HW & SW and run on hardware
Automatic software and hardware instrumentation for performance monitoring

Provide visibility into “higher level events” during program execution, with finer granularity than overall run time
- Accelerator tasks
- Data transfers between accelerators and between accelerators and PS

Assist in system debugging, showing “what happened when”

Provide application-specific trace points
- e.g., depending on accelerators

Minimize impact on execution time and PL area
Trace Example

Matrix Multiplication

main function

```c
int main(int argc, char* argv[]) {
    float *A, *B, *C;
    init(A, B, C);
    mmult(A, B, C);
    check(C);
}
```

mmult function

```c
void mmult(float *A, float *B, float *C) {
    for (int a=0; a<A_NROWS; a++)
        for (int b=0; b<B_NCOLS; b++) {
            float result = 0;
            for (int c=0; c<A_NCOLS; c++)
                result += A[a][c]*B[c][b];
            C[a][b] = result;
        }
}
```

Original Code
Trace Example

Matrix Multiplication

**main function**

```c
int main(int argc, char* argv[]) {
    float *A, *B, *C;

    init(A, B, C);
    mmult(A, B, C);

    check(C);
}
```

**mmult function**

```c
void mmult(float *A, float *B, float *C) {
    for (int a=0; a<A_NROWS; a++)
        for (int b=0; b<B_NCOLS; b++) {
            float result = 0;
            for (int c=0; c<A_NCOLS; c++)
                result += A[a][c]*B[c][b];
            C[a][b] = result;
        }
}
```

**Original Code**

```c
int main(int argc, char* argv[]) {
    float *A, *B, *C;

    init(A, B, C);
    mmult(A, B, C);

    check(C);
}
```

**Stub Code**

```c
int main(int argc, char* argv[]) {
    float *A, *B, *C;

    init(A, B, C);
    _p0_mmult_0(A, B, C);

    check(C);
}
```

```c
void _p0_mmult_0(float *A, float *B, float *C) {
    cf_send_i(&req0, cmd);
    cf_wait(req0);
    cf_send_i(&req1, A);
    cf_send_i(&req2, B);
    cf_wait(req1);
    cf_wait(req2);
```
Trace Example

Matrix Multiplication

**main function**

```c
int main(int argc, char* argv[]) {
    float *A, *B, *C;

    init(A, B, C);
    mmult(A, B, C);

    check(C);
}
```

**mmult function**

```c
void mmult(float *A, float *B, float *C){
    for (int a=0; a<A_NROWS; a++)
        for (int b=0; b<B_NCOLS; b++) {
            float result = 0;
            for (int c=0; c<A_NCOLS; c++)
                result += A[a][c]*B[c][b];
            C[a][b] = result;
        }
}
```

**Trace Code**

```c
int main(int argc, char* argv[]) {
    float *A, *B, *C;

    init(A, B, C);
    _p0_mmult_0(A, B, C);

    check(C);
}
```

```c
void _p0_mmult_0(float *A, float *B, float *C) {
    sds_trace(EVENT_START);
    cf_send_i(&req0, cmd);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(req0);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_send_i(&req1, A);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_send_i(&req2, B);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(req1);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
    cf_wait(req2);
    sds_trace(EVENT_STOP);
    sds_trace(EVENT_START);
}
```

No change

Added IPs
- 4 AXI Stream monitors
- 1 Accelerator monitor
- Trace output infrastructure

© Copyright 2015 Xilinx
Example: Matrix Multiply-Add

- Add matrix addition operator to demonstrate how to construct hardware pipelines to increase concurrent computation

- SDSoC compiler will create hardware ‘direct connections’ between accelerators and between platform and accelerators
  - Program dataflow analysis to ensure correct behavior
  - Software synchronization automatically instrumented by the compiler
bool mmultadd(float *A, float *B, float *C, float *Ds, float *D) {
    float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];
    for (int i = 0; i < NUM_TESTS; i++) {
        mmultadd_init(A, B, C, Ds, D);
        mmult(A, B, tmp1);
        madd(tmp1, C, D);
        mmult_golden(A, B, tmp2);
        madd_golden(tmp2, C, Ds);
        if (!mmult_result_check(D, Ds))
            return false;
    }
    return true;
}
How SDSoC Compiler Maps Programs to SW

Structure of generated software

```c
bool mmultadd_test(float *A, float *B, float *C, float *D, float *D)
{
    std::cout << "Testing mmult .." << std::endl;
    float tmp1[A_NROWS * A_NCOLS], tmp2[A_NROWS * A_NCOLS];
    for (int i = 0; i < NUM_TESTS; i++) {
        mmultadd_init(A, B, C, D, D);
        p0_mmult(A, B, tmp1);
        p0_madd(tmp1, C, D);
        mmult_golden(A, B, tmp2);
        madd_golden(tmp2, C, D);
    }
    return 0;
}
```

```c
void p0_mmult_0(float in_A[1024], float in_B[1024], float out_C[1024])
{
    switch_to_next_partition(0);
    int start_seq[3];
    start_seq[0] = 0x00000000;
    start_seq[1] = 0x00010000;
    start_seq[2] = 0x00020000;
    cf_request_handle_t p0_swinst_mmult_0_cmd;
    cf_send_i(&p0_swinst_mmult_0.cmd_mmult,
               start_seq, 3*sizeof(int), &p0_swinst_mmult_0.cmd);
    cf_wait(p0_swinst_mmult_0.cmd);
    return 0;
}
```

```c
void p0_madd_0(float A[1024], float B[1024], float C[1024])
{
    switch_to_next_partition(0);
    int start_seq[3];
    start_seq[0] = 0x00000000;
    start_seq[1] = 0x00010000;
    start_seq[2] = 0x00020000;
    cf_request_handle_t p0_swinst_madd_0_cmd;
    cf_send_i(&p0_swinst_madd_0.cmd_madd, start_seq, 3*sizeof(int), &p0_swinst_madd_0.cmd);
    cf_wait(p0_swinst_madd_0.cmd);
    return 0;
}
```

Control transfer

Control Synchronization

Data transfers
Summary

➢ System performance achieved through accelerator and system level optimizations
  – SDSSoC compiler creates function pipelines with direct connections in hardware

➢ Increase concurrency within accelerators using HLS directives
  – Pipeline and dataflow loops, function calls, and operations
  – Copy data samples into local BRAM to improve burst read/write and partition to increase compute / memory bandwidth within accelerator
  – UG902: HLS User Guide for more details

➢ Data mover and system connectivity inference from user program
  – Data mover selection based on buffer allocation, transfer payload
  – System connections and driver efficiency based on program memory properties, e.g., cacheability
  – UG1027: SDSSoC User Guide for more details
Agenda

» Zynq SoC and MPSoC Architecture
» SDSoC Overview
» Real-life Success
» C/C++ to Optimized System
» Targeting Your Own Platform
» Next Steps
We make a clear distinction between **platforms** and **Software-Defined systems on chip**

A **platform** is a base system designed for reuse
- Processing system, I/O subsystems, memory interfaces,…
- OS, device drivers, boot loaders, file system, libraries,…
- Built using standard SoC HW & SW design methodologies and tools

A **software-defined SoC** extends a platform with application-specific hardware and software
- User specifies functions for programmable logic
- Compiler analyzes program and compiles into an application-specific SoC
- Hardware accelerator and data motion network
- **#pragmas** to assist and override compiler
zc702_trd Platform (Targeted Reference Design)
Creating a Platform

IP repository
Manual custom board def

Vivado
Build platform hardware

SDK/HSI

DTG/HSI

Vivado
Project files

FSBL
Device Tree
Platform SW libs
Bif file
Uboot.elf
uramdisk

Build Linux and Uboot

Device drivers

GIT repo

Build Ramdisk

Operating System
Platform
PS
Platform component

Page 69
Creating an SDSoC Platform

- IP repository
- Manual custom board def
- Device drivers
- GIT repo
- SDK/HSI
- DTG/HSI
- Vivado
  - Build platform hardware
  - Write and execute SDSoC TCL script
- Vivado Project files
- Platform HW XML
- Platform SW XML
- FSBL
- Device Tree
- Platform SW libs
- Bif file
- Uboot.elf
- uramdisk
- uramdisk

Add configs
- xilinx-apf
- CMA

Build Linux and Uboot

Build Ramdisk

Write by hand

Operating System

Platform component
SDSoC add-on
SDSoC Platform Hardware

Start from essentially any Vivado hardware system
- Zynq-7000® or Zynq-UltraSCALE+ MPSoC® processing system
- Memory interfaces, custom I/O, and other peripherals
- Set of AXI, AXI-S, clocks, resets, interrupt ports

Create TCL script
- Declare platform interfaces in a Vivado block diagram
- Generate platform hardware description XML file
SDSoC Platform Hardware APIs

```bash
set pfm [sdsoc::create_pfm zc702_hw.pfm]
sdsoc::pfm_name $pfm "xilinx.com" "xd" "zc702" "1.0"
sdsoc::pfm_description $pfm "Zynq ZC702 Board"
sdsoc::pfm_clock $pfm FCLK_CLK0 ps7 0 false proc_sys_reset_0
sdsoc::pfm_clock $pfm FCLK_CLK1 ps7 1 false proc_sys_reset_1
sdsoc::pfm_clock $pfm FCLK_CLK2 ps7 2 true  proc_sys_reset_2
sdsoc::pfm_clock $pfm FCLK_CLK3 ps7 3 false proc_sys_reset_3
sdsoc::pfm_axi_port $pfm M_AXI_GP0 ps7 M_AXI_GP
sdsoc::pfm_axi_port $pfm M_AXI_GP1 ps7 M_AXI_GP
sdsoc::pfm_axi_port $pfm S_AXI_ACP ps7 S_AXI_ACP
sdsoc::pfm_axi_port $pfm S_AXI_HP0 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP1 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP2 ps7 S_AXI_HP
sdsoc::pfm_axi_port $pfm S_AXI_HP3 ps7 S_AXI_HP
for {set i 0} {$i < 16} {incr i} {
    sdsoc::pfm_irq $pfm In$i xlconcat
}
sdsoc::generate_hw_pfm $pfm
```
SDSoC Platform Software

- Operating systems
  - Linux, bare metal, FreeRTOS

- Boot loaders
  - FSBL, U-Boot

- Library files
  - Needed for cross-compiling and linking application code
  - Shared libraries must also reside in target rootfs

- Platform software description metadata file
  - Provides information needed to compile, link, generate SD cards, etc.
  - Written by hand by platform provider
SDSoC Platform Software Description

zc702

```xml
<xd:bootFiles
  xd:os="linux"
  xd:bif="boot/linux.bif"
  xd:readme="boot/generic.readme"
  xd:devicetree="boot/devicetree.dtb"
  xd:linuxImage="boot/uImage"
  xd:ramdisk="boot/uramdisk.image.gz"
/>

<xd:bootFiles
  xd:os="standalone"
  xd:bif="boot/standalone.bif"
  xd:readme="boot/generic.readme"
/>

<xd:libraryFiles
  xd:os="standalone"
  xd:libDir="arm-xilinx-eabi/lib"
  xd:ldscript="arm-xilinx-eabi/lscript.ld"
/>

<xd:libraryFiles
  xd:os="freertos"
  xd:osDepend="standalone"
  xd:includeDir="/arm-xilinx-eabi/include/freertos"
  xd:libDir="/arm-xilinx-eabi/lib/freertos"
  xd:libName="freertos"
  xd:ldscript="freertos/lscript.ld"
/>

<xd:hardware
  xd:system="prebuilt"
  xd:bitstream="hardware/prebuilt/bitstream.bit"
  xd:export="hardware/prebuilt/export"
  xd:swcf="hardware/prebuilt/swcf"
  xd:hwcf="hardware/prebuilt/hwcf"
/>
```
Testing Your SDSoC Platform

Platform checklist with design guidelines

<table>
<thead>
<tr>
<th>Description</th>
<th>Notes</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>PS configuration</td>
<td></td>
<td>- zc702_acp, zc706_mem</td>
</tr>
<tr>
<td>Non-PS AXI interfaces</td>
<td></td>
<td>- zc702_axis_io</td>
</tr>
<tr>
<td>Clocks</td>
<td></td>
<td>- zc702, zc705, zed, microzed, micromzed00, zybo</td>
</tr>
<tr>
<td>Resets</td>
<td></td>
<td>- zc702, zc706, zed, microzed, micromzed00, zybo</td>
</tr>
<tr>
<td>Directory structure</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

And basic datamover conformance tests

Basic platform tests to ensure that all SDSoC datamovers work on platform. In addition, provide a test for every external I/O function

<table>
<thead>
<tr>
<th>Description</th>
<th>Datamover</th>
<th>Sample command</th>
</tr>
</thead>
<tbody>
<tr>
<td>axi_dma_simple datamover test</td>
<td>axi_dma_simple</td>
<td>make axi_dma_simple [PLATFORM=&lt;your_platform&gt;]</td>
</tr>
<tr>
<td>axi_dma_2d datamover test</td>
<td>axi_dma_2d</td>
<td>make axi_dma_2d [PLATFORM=&lt;your_platform&gt;]</td>
</tr>
<tr>
<td>axi_fifo datamover test</td>
<td>axi_fifo</td>
<td>make axi_fifo [PLATFORM=&lt;your_platform&gt;]</td>
</tr>
<tr>
<td>zero_copy data mover test</td>
<td>zero_copy</td>
<td>make zero_copy [PLATFORM=&lt;your_platform&gt;]</td>
</tr>
<tr>
<td>axi acceleration adapter data mover test</td>
<td>axi_dma_2d</td>
<td>make axi_adapter [PLATFORM=&lt;your_platform&gt;]</td>
</tr>
</tbody>
</table>

Reference: ug1146 SDSoC Platforms and Libraries
Location: <dsoo_root>/docs

© Copyright 2015 Xilinx
Available SDSoc Platforms

- **Standard “memory-based I/O” platforms**
  - zc702, zc706, zed, zybo, microzed

- **Video & image processing oriented platforms**
  - zc702_trd, zc706_trd (separate download)
  - zc702_osd, zed_osd

- **Additional downloads from Xilinx and partners**
  - Zynq base targeted reference designs (zc702_trd, zc706_trd)

- **Teaching platform examples**
  - zc702_axis_io – direct I/O
  - zc702_led – software control of platform IPs (standalone, Linux)
  - zc702_acp – sharing an AXI interface between platform and sdscc
Summary

➤ Platform-based design increases productivity and encourages design reuse
   – Many applications can target a single platform
   – An application can target multiple platforms

➤ SDSoC platforms are simple extensions of standard hardware / software systems that enable design reuse
   – Hardware platform easily exported from Vivado
   – Software platform built using standard flows, simple metadata file
Agenda

Zynq SoC and MPSoC Architecture
SDSoC Overview
Real-life Success
C/C++ to Optimized System
Targeting Your Own Platform
Next Steps
Next Steps

- Hands-on training with one of our Authorized Training Providers
- Video Tutorials
- User Guides

To further enhance your productivity, consider:
- Libraries & Design Examples
- Boards, Kits & Modules
Self-Training Material

Video Tutorials

- Custom Platform Creation
- Estimation & Implementation
- Optimization & Debug

User Guides

- UG1028
  - Getting started
- UG1027
  - SDSoc flows, features & functions
- UG1146
  - Create a custom SDSoc platform and a C-callable RTL IP Library
Optimized libraries for faster programming

Available from Xilinx and ecosystem partners

### Hardware Optimized Libraries

<table>
<thead>
<tr>
<th>Library Suites</th>
<th>Latest SDSoc Version Supported</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>OpenCV</td>
<td>2015.4</td>
<td>Auviz</td>
</tr>
<tr>
<td>40+ hardware optimized OpenCV functions, including Gaussian, Median, Bilateral, Harris corner, Canny edge detection, HoG, ORB, SVM, LK Optical Flow, and many more</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HLS Built-in Libraries</td>
<td>2015.4</td>
<td>Xilinx</td>
</tr>
<tr>
<td>Many functions in OpenCV, linear algebra and signal processing, See XAPP1167</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Design Example Built-in to the Development Environment

<table>
<thead>
<tr>
<th>Design Example &amp; Descriptions</th>
<th>Latest SDSoc Version Supported</th>
<th>Board &amp; SoM Supported</th>
<th>Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>Matrix Multiply and Addition</td>
<td>2015.4</td>
<td>All</td>
<td>Xilinx</td>
</tr>
<tr>
<td>32x32 Floating point matrix multiply and matrix addition. Demonstrates AXI DMA inference as well as direct IP-IP streaming connections</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIR Filter</td>
<td>2015.4</td>
<td>All</td>
<td>Xilinx</td>
</tr>
<tr>
<td>Demonstrates a simple C-callable HDL IP using Xilinx FIR compiler</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>File I/O Video Processing</td>
<td>2015.4</td>
<td>All</td>
<td>Xilinx</td>
</tr>
<tr>
<td>Demonstrate a typical algorithm development using an input file and output file. Highly portable to any platforms</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Design Example Offered by Partners

<table>
<thead>
<tr>
<th>Design Example &amp; Descriptions</th>
<th>Latest SDSoc Version</th>
<th>Board &amp; SoM Supported</th>
<th>Provider</th>
</tr>
</thead>
</table>
Boards, Kits & Modules

➢ System-level solutions for multiple functions including video, radio & control

<table>
<thead>
<tr>
<th>Board Name</th>
<th>I/O Enabled</th>
<th>Latest SDSoc Version Supported</th>
<th>Design Examples</th>
<th>SDSoc Platform Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZC702</td>
<td>PS DDR</td>
<td>2015.4</td>
<td>Basic Suite*</td>
<td>Xilinx</td>
</tr>
<tr>
<td>ZC702</td>
<td>USB Webcam in, HDMI out, PS DDR</td>
<td>2015.4</td>
<td>Basic Suite*</td>
<td>Xilinx</td>
</tr>
<tr>
<td>ZC706</td>
<td>PS DDR</td>
<td>2015.4</td>
<td>Basic Suite*</td>
<td>Xilinx</td>
</tr>
<tr>
<td>ZC706</td>
<td>PL DDR, PS DDR</td>
<td>2015.4</td>
<td>Basic Suite*</td>
<td>Xilinx</td>
</tr>
<tr>
<td>ZedBoard</td>
<td>PS DDR</td>
<td>2015.4</td>
<td>Basic Suite*</td>
<td>Xilinx</td>
</tr>
<tr>
<td>ZedBoard</td>
<td>USB Webcam in, HDMI out, PS DDR</td>
<td>2015.4</td>
<td>Basic Suite*</td>
<td>Xilinx</td>
</tr>
<tr>
<td>MicroZed</td>
<td>PS DDR</td>
<td>2015.4</td>
<td>Basic Suite*</td>
<td>Xilinx</td>
</tr>
<tr>
<td>ZYBO</td>
<td>PS DDR</td>
<td>2015.4</td>
<td>Basic Suite*</td>
<td>Xilinx</td>
</tr>
</tbody>
</table>

Video Platforms (Externally Provided)

<table>
<thead>
<tr>
<th>Board Name</th>
<th>I/O Enabled</th>
<th>Latest SDSoc Version Supported</th>
<th>Design Examples</th>
<th>SDSoc Platform Provider</th>
</tr>
</thead>
<tbody>
<tr>
<td>ZC702 + HDMI IO FMC</td>
<td>HDMI in, HDMI out, PS DDR</td>
<td>2015.2.1</td>
<td>Sobel Filter, Basic Suite*</td>
<td>Xilinx</td>
</tr>
<tr>
<td>Atlas-I-Z7e + Captiva Carrier Card</td>
<td>GigEV in, HDMI out, PS DDR</td>
<td>2015.2.1</td>
<td>Canny Edge Detection, Basic Suite*</td>
<td>iVela</td>
</tr>
<tr>
<td>MIAMI</td>
<td>PS DDR</td>
<td>2015.2.1</td>
<td>Basic Suite*</td>
<td>TOPIC</td>
</tr>
</tbody>
</table>
SDSoC Development Environment

Thank You