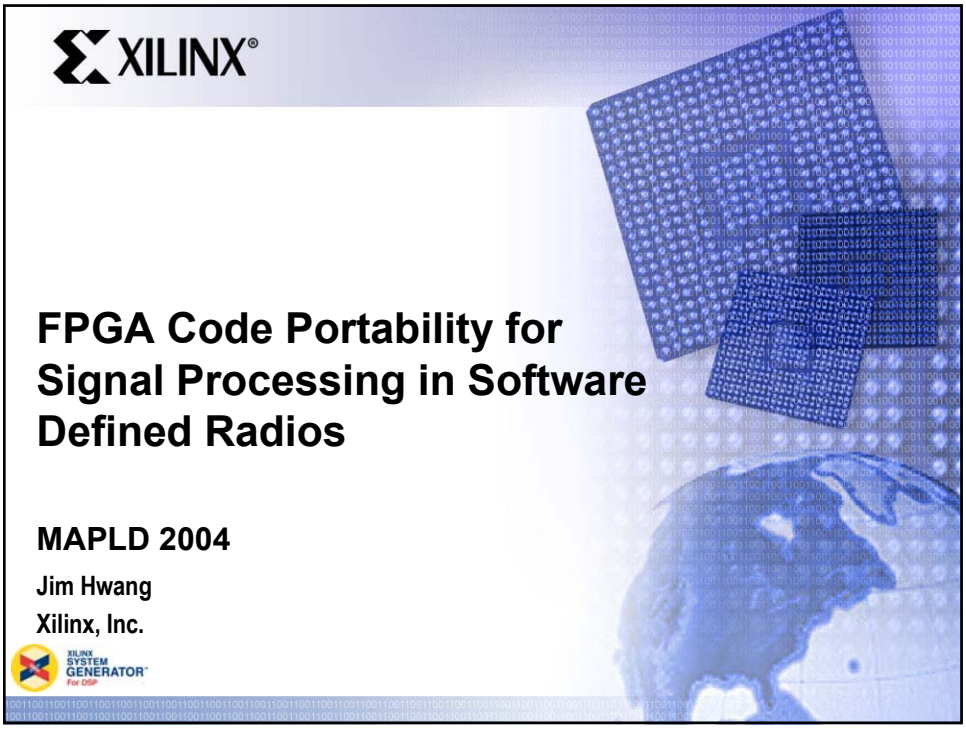


The Problem

- Source code often outlives the hardware generation for which it was designed
  - JTRS program lifetime 10 years
  - New FPGA architecture every 12-18 months
- FPGA architecture used for initial design may not be the FPGA architecture for deployment
- High-performance FPGA implementations exploit device-specific hardware resources
  - Block, distributed, and shift-register based RAM
  - Dedicated multipliers or DSP blocks
- Modern FPGAs encompass both hardware and embedded software
  - Inherent challenges of heterogeneous source code
- FPGA bitstreams are unique to vendor, family, part
  - cf. Intel x86 compatibility

XILINX®

Code Portability

- Bitsteam level portability
  - Bitstream for device V(n) performs the same function in device V(n+1) without modification
  - Bitstream for device V(n) performs the same function in device S(n) without modification
  - Requires vendor support
  - Not practical for existing FPGA architectures
- Source level portability
  - Source code for device V(n) performs the same function in device V(n+1) after recompilation
  - Source code for device V(n) performs the same function in device S(n) after recompilation
  - Achievable through coding style and methodology

Steps Towards Portability

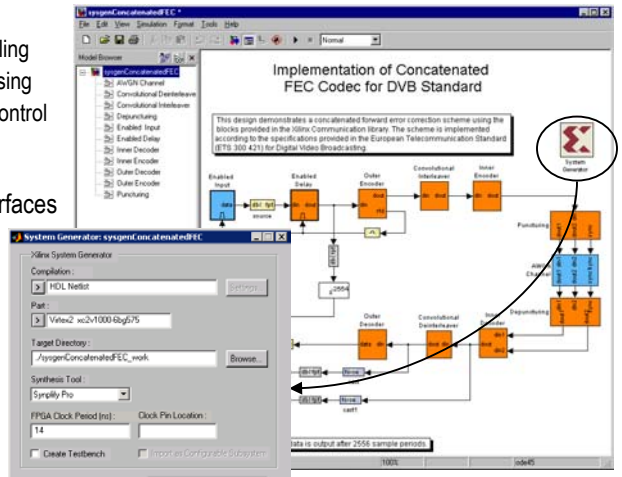
- Identify and develop guidelines and methodologies to support source level portability
- Adopt design styles to exploit available technologies
  - Distinguish subsystems where opportunities exist today
    - Signal processing subsystems provide greater opportunity for portability than I/O interfaces
  - Use coding styles that minimize architecture specific constructs
  - Use tools that provide efficient mappings from behavioral descriptions
- Compromise when expedient
  - Encapsulate areas where portability is infeasible
  - cf. assembly programming for general purpose processors

New Programming Models

- Platform-based viewpoint and methodologies
- Industry standard frameworks for code portability (“single-source”)
  - Modeling and arithmetic abstraction
    - MATLAB, Simulink
  - Register transfer level as intermediate form
    - HDL source code only when necessary
  - Firmware
    - ANSI C
- Code generation technology for code portability
  - Vendor specific IP libraries for highest performance FPGA
  - RTL descriptions for retargeting to alternative HW platforms
  - Bit-accurate and cycle accurate models for GPP
  - Embedded software targets
    - Relaxed bit and cycle accuracy

Xilinx System Generator

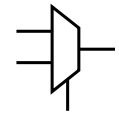
- Xilinx Block Set (Simulink)
  - Bit and cycle true modeling
  - Multi-rate signal processing
  - Arithmetic, memories, control logic, DSP, Comms, …
  - HDL Black Box
- Simulink and MATLAB interfaces
  - Type, rate propagation
  - m-code compilation
- Co-simulation interfaces
  - Hardware in the loop
  - HDL simulation
- Automatic code generation
  - RTL VHDL
  - IP Cores



Register Transfer Level

- Current prevailing abstraction for FPGA design
  - Intermediate form in the System Generator context
- Logic synthesis
  - Tools & algorithms for architecture specific mappings from general constructs
  - Limited behavioral optimization
    - Retiming can provide significant speed-up
- System Generator blocks with generic RTL targets
  - Low level library blocks
    - Delay, register, up/down sampler, etc.
  - Expression block
    - MATLAB expressions to synthesizable HDL
  - M-Code block
    - MATLAB .m code function to synthesizable HDL

M-Code Block Mapping



```
function [c] = multiplexer(a, b sel)
    zero = xfix({xlUnsigned, 2, 0},0);
    if (sel == zero), c = b;
    else, c = a, end
```

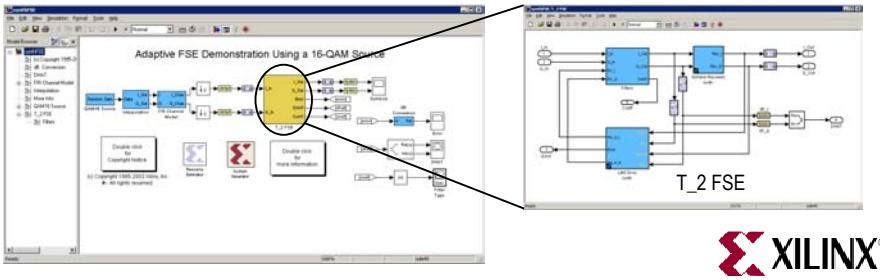

```
function [c] = adder(a, b)
    c = a + b;
```


```
function [c] = multiplier(a, b)
    c = a * b;
```

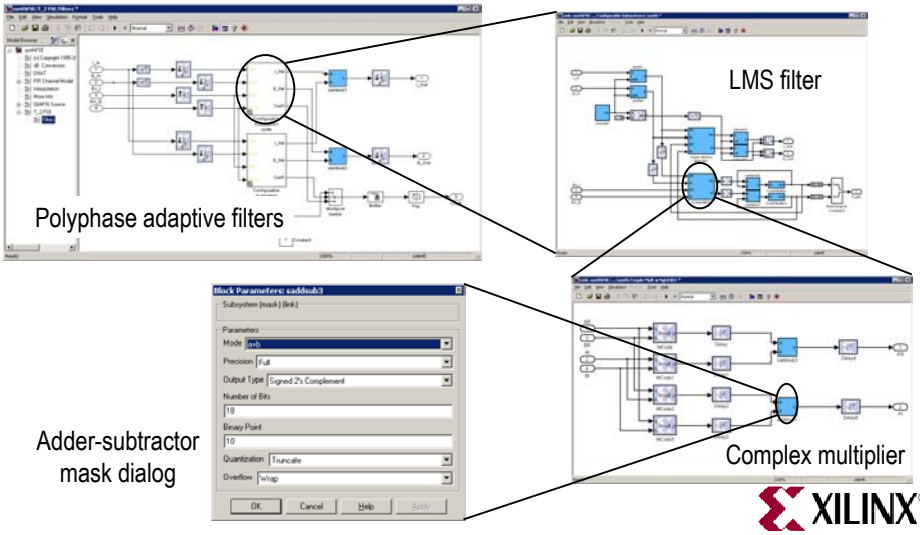
- MATLAB function interpreted as a streaming operator in Simulink
  - Arguments can be either a block parameter or a port
  - Supports branching constructs
    - Arbitrarily nested switch, if-then-else
  - Fixed point arithmetic operations
    - Addition, subtraction, multiplication, relational bitwise logic
- System Generator converts m-code into RTL VHDL

Case Study: FSE Equalizer

- Fractionally spaced adaptive equalizer
  - Original design targeted IP cores
  - Replace all non-synthesizable blocks by m-code blocks
  - Simulink source level portability across multiple FPGA families
    - Also portable to ASIC
  - Delay insertion to exploit retiming during logic synthesis



System Generator Model



Compilation Results

Part	MHz Cores	MHz Synth	LUTs Cores	LUTs Synth	DFFs Cores	DFFs Synth
xc2vp20 –7	88.9	104.1	1636	1595	1868	2195
xc3s1500 –5	79.3	85.5	1636	1531	1868	2193

- Notes
  - Synplify Pro 7.6, retiming & pipelining options
  - ISE 6.2i service pack 2, placer effort = router effort = high
  - Resource discrepencies largely due to SRL16 mapping
  - Performance discrepencies largely due to retiming
    - IP cores inhibit boundry optimizations

Conclusions

- Summary
  - Code portability achieved across FPGA families from Simulink using System Generator
    - RTL an effective intermediate form
  - No significant performance or area penalty for portability
- Future Work
  - Refine system level abstractions and code generation to further leverage downstream optimization
    - Synthesis tools continue to improve
    - Retiming only recently became effective
  - Further develop RTL constructs and support in context of high-level simulation
  - Extend code portability considerations to hardware abstraction layer definition and guidelines