!252-0286-00L
"The belief...
>that complex systems require armies of designers and programmers
>is wrong.
>
>"A system that is not understood in its entirety, or at
>least to a significant degree of detail by a single individual,
>should probably not be built."
>
>
- Niklaus Wirth (Feb. 1995), "A Plea for Lean Software", IEEE Computer

ETH Vorlesung Systembau / Lecture System Construction
>Case Study: Custom-designed Single-Processor System
>Paul Reed (paulreed@paddedcell.com)
>
>Overview
- RISC single-processor personal computer designed from scratch
- Hardware on field-programmable gate array (FPGA)
- (this lecture) Motivation and goals; FPGAs; RISC CPU
- (next lecture) Graphical workstation OS and compiler (Project Oberon)

Motivation
- "Project Oberon" (1992) by N. Wirth & J. Gutknecht, at ETH Zurich
- available commercial systems are far from perfect
- building a complete system from scratch is achievable and beneficial
- not just a "toy" system: complete and self-hosting
- personally: need good and reliable tools for commercial programming
- more recently: security - knowing what's inside (IoT, medical)

Case Study Goals
- weigh pros and cons of designing from scratch
- overview of using FPGAs to design custom hardware
- benefits of software/hardware co-design
- competence in building complete system from the ground up
- understanding of "how it really works" from hardware to application
- courage to apply "lean systems" approach wherever appropriate

Why Build from Scratch?
- clear design: easy to see where to extend or fix
- flexible and based solely on problem domain and experience
- reduce complexity: no "baggage", less of what you don't like
- increase control, reduce the number of dependencies
- more choices of implementation, more of what the customer asked for
- eliminate surprises: deliver on time and on budget
- source of competitive advantage
- opportunity to change the world :)

Why not Build from Scratch?
- duplication of effort: "re-inventing the wheel"
- more fundamental knowledge required
- may be more actual work (the first time)
- risky: tendency to underestimate
- restricted component choices
- not for the short-term
- no credit for only *trying* to change the world :(

Configurable Hardware
- evolution of programmable logic (PALs/GALs, CPLDs)
- look-up tables (LUTs), registers and interconnect
- special functions: PLLs, multipliers (DSP), I2C, DDR, video/SERDES
- loadable configuration (bitstream), not fixed like VLSI / ASIC
- applications from telecommunications to automotive and industrial
- even banking (high-frequency trading) and cryptocurrency mining
- flexible, but not the best for performance or for power
- now big (and fast) enough for entire system-on-chip
- ~US$50: Xilinx XC3S700AN (11K LUT) or Lattice ECP5-85 (85K LUT)

Hardware Description Languages
- used to describe digital circuits textually
- (hopefully) more precise and formal than schematic capture
- same source code used for both simulation and synthesis
- commercial examples: (System)Verilog, VHDL
- developed at ETH: Lola, Active Cells
- VERY different from software programming languages:
- concurrent, notion of time, resource limitations
- not a perfect description (e.g. timing, metastability)

FPGA Development Toolchains
- synthesis: create logical "netlist" of components and connections
- technology mapping: against a physical chip family
- placement: onto a particular target chip
- routing: of connections between the placed cells
- bitstream generation: encoding used to configure the chip
- timing closure: are all timing requirements met?
- simulation: at synthesis level or post-P&R
- power and electrical models (e.g. IBIS)
- integrated HDL-driven environment (usually proprietary $$$)
- (possibly) integrated software/hardware co-design

Hardware Flashing-LED Test (Demo)
- [handout TestLEDs-Verilog.pdf: "TestLEDs.v"]
- fully-hardware-only solution as a simple example of Verilog
- define module inputs and outputs, registers, and wires
- single-bit signals and multi-bit busses/registers

- combinational: "assign" (wiring up)
- register-transfer: "always @()" (state changes)
- constraints/preferences (e.g. Xilinx .ucf) for pin assignment
- low-level, pedantic

Introduction to Niklaus Wirth's RISC Processor
- originally a 32-bit virtual machine target for "Compiler Construction"
- follows successful reduced-instruction-set design philosophy
- registers instead of a stack machine
- Harvard or Von Neumann memory architecture
- hardware floating-point option
- defined in Verilog and implemented on Xilinx Spartan 3 FPGA
- integer RISC0: ~1900LUTs/200reg (inc. ~400LUT/140reg for mul/div)
- complete 30MHz RISC0 microcontroller, timer, RS232: 2K LUT, 300 regs

RISC Architecture Overview
- [handout RISC-Architecture.pdf: "The RISC Architecture"]
- program counter (PC) and instruction register (IR)
- instruction decode logic - "control unit"
- 16 general-purpose 32-bit registers - "register file"
- arithmetic and logic, barrel shifter; flags NZCV
- memory interface

The RISC Instruction Set
- 16 arithmetic and logic instructions (reg/reg and reg/immediate)
- load and store register to/from data memory (word and byte)
- conditional branch (-and-link), 8 conditions and their opposite
>
>That's all folks! :)
>
>
>
>
- example: MOV SP -64   ????_????_????_????_????_????_????_????
>(assume fields u and b are not used and set to 0; and SP is R14)

RISC0 Implementation on a Xilinx FPGA
- [handout RISC0-Verilog.pdf: "module RISC0Top..."]
- Harvard RISC0 core in Verilog
- on-FPGA ROM for program, on-FPGA RAM for data
- memory-mapped I/O ports
- port examples: timer, LEDs, switches/jumpers, RS232
- Verilog "top" module: outside-world interface
- choice of fast multiply where FPGA hardware is available
- user constraints file (UCF)

Software Flashing-LED Test
- [handout TestLEDs-Oberon.pdf: "MODULE* TestLEDs"]
- "MODULE*" signifies a standalone module e.g. for PROM
- machine code output by compiler (prom.mem)
- (unused) initialisation of stack
- variables based at address 0
- main loop - output to LED port at -60 (FFFFFFC4)
- nested delay loops
- unused termination code

Software/Hardware Co-Design
>Example 1, Pulse-Width Modulation (exercise 1d)
>
- OberonStation has waaay-bright LEDs! :)
- use mS timer (adr -64) to illuminate LEDs for 1/16th duration
- but, need to include such for every routine writing to LEDs (exercise)
- so, why not do it in hardware: 1-line change to Verilog!
- assign leds = (cnt1[3:0] == 0) ? Lreg : 0;

Building a Multiplexer from LUTs
- 8 LED outputs controlled by a single signal
- switches between LED port output and 0 (off)
- when cnt1[3:0] == 0 handled by one 4-input LUT
- each of 8 bits multiplexed: one 3-input LUT
- but one input is always 0, so really only 2 inputs
- could optimise? Depends on technology but probably not
- plenty of opportunity in general - "flatten hierarchy"
- leads to renaming, makes examination and tracing difficult

Software/Hardware Co-Design
>Example 2, (Kostenlos!) Light Detector (exercise 1e)
>
- photoelectric effect on voltage decay via parasitic capacitance
- alter Verilog for ability to read LED outputs as input (inout)
- assign leds = (cnt1[3:0] == 0) ? Lreg : ...
- ... (cnt1[3:0] == 1) & (cnt0[14:7] == 0) ? 8'hFF : 8'hzz;
- (display Lreg value for 1mS, fire all LEDs for 2uS @ 30MHz, then tri-state)
- fire LEDs, input falls through Vih/Vil, leds no longer read as high
- do the rest in software: sync with hardware, count until leds # 0FFH
- show delay as moving bar better than binary, because of fluctuation
- shine a bright light to test (don't touch or heat!)

Exercise 1: RISC on the OberonStation FPGA Board

Exercise 1a: Tools and Workflow
- [handout OberonStation.pdf: "OberonStation"]
- [handout XilinxSetupRISC0.pdf: "RISC0 Project Setup and Test Instructions"]
- [handout ORC-Compile.pdf: "ORC: The Oberon-07 Command-line Compiler"]
- install Xilinx ISE and Oberon cross-compiler ORC
- create RISC0 project, add Verilog source code (src directory)
- compile TestLEDs.Mod Oberon program, prom.mem to proj dir
- in ISE generate "programming file", ie hardware bitstream
- download to board using programming tool, e.g. iMPACT
- compile TestSwi.Mod example, update prom.mem and regenerate bitstream
- move "parked" jumper across J0, J1 to test "switches" J0-J3 (LEDs 0..2, 7)

Exercise 1b: Develop an Instruction Timer
- use TestLEDs.Mod as template, add variable t
- SYSTEM.GET(-64, t): 32-bit mS time at port -64
- get time in t at beginning, and into z at end, of outer loop
- run middle loop 100 iterations, inner loop 10000 iterations
- display (z - t) DIV 100 on LEDs at end of outer loop
- note mS, then compare after adding a (non-trivial) DIV in inner loop
- (optional) calculate exact cycle time for DIV instruction

Exercise 1c: Compare Hardware Implementations
- use 1b instr. timer to measure (non-trivial) multiply instead of DIV
- change hardware to use Multiply1.v employing MULT18X18
- (remove Multiplier.v, add Multiplier1.v, edit RISC0.v)
- measure performance of multiply again
- consider pros and cons of both designs

Exercise 1d (optional): Pulse-Width Modulation
- (for overview see lecture slide)
- first, implement PWM in software in TestLEDs, using mS timer
- (hint - you will need to move SYSTEM.PUT)
- revert software to non-PWM TestLEDs version, check brighter again
- add lecture slide PWM Verilog code, then test

Exercise 1e (optional): Kostenlos Light Detector
- (for overview see lecture slide)
- change [7:0] leds in Verilog module definition from output to inout
- allow reading leds: (iowadr == 1) ? {16'b0, leds, ~swi}
- change assign leds = display, fire and detect delay (lecture slide)
- modify outer loop of TestLEDs to start with sync to hardware
- (wait for timer MOD 16 = 0, then # 0, using temp variable n)
- use middle loop of TestLEDs to detect decay
- ie, x counts number of inner delay loops of (say) y := 50
- then x := x - 1; SYSTEM.GET(swiAdr, n) UNTIL n DIV 100H # 0FFH
- subtract ambient level - 7 (x-7 stored in z when z = 0 on reset) from x
- to display, SYSTEM.PUT(ledAdr, LSL(1, x)) for a moving bar
- (limit x to between 0 and 7 incl. to show full deflection either way)
- reduce/increase inner loop iterations to increase/decrease sensitivity

[end of first lecture and exercises]