ETH Vorlesung Systembau / Lecture System Construction

>252-0286-00L - Dr. Felix Friedrich, Paul Reed
>
>Case Study: Custom-designed Single-Processor System
>Paul Reed (paulreed@paddedcell.com)
>

- First Lecture: The RISC Architecture
- [Second Lecture: Project Oberon on RISC]

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

Introduction
- RISC single-processor personal computer designed from scratch
- Hardware on field-programmable gate array (FPGA)
- (this lecture) Motivation and goals; RISC CPU
- (next lecture) Graphical workstation OS and compiler (Project Oberon)

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

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

Discussion: Why Build from Scratch? (1)
- reduce complexity: no "baggage"
- clear design: easy to see where to extend or fix
- increase control, reduce the number of dependencies
- more choices of implementation
- design based solely on problem domain and experience

Discussion: Why Build from Scratch? (2)
- eliminate surprises: deliver on time and on budget
- highly flexible solution
- more of what the customer asked for
- source of competitive advantage
- accept less of what you don't like

Discussion: Why not Build from Scratch?
- duplication of effort: "re-inventing the wheel"
- more fundamental knowledge required
- may be more actual work (the first time)
- restricted component choices
- not for the short-term

Introduction to Configurable Hardware
- evolution of programmable logic (PALs/GALs, CPLDs)
- look-up tables (LUTs), registers and interconnect
- current field programmable gate array (FPGA) technology
- loadable configuration, not "set in stone" like VLSI / ASIC
- applications from telecommunications to automotive and industrial
- even banking: high-frequency trading; Bitcoin mining
- now big (and fast) enough for entire system-on-chip

Introduction to HDLs
- hardware description language to define circuits formally
- used for both simulation and synthesis
- commercial examples: Verilog, VHDL
- developed at ETH: Lola, Active Cells
- VERY different from conventional programming languages

Hardware Flashing-LED Test
- [handout TestLEDs-Verilog.pdf: "TestLEDs.v"]
- hardware-only solution as a simple example of Verilog
- define module inputs and outputs, registers, and wires
- combinational (wiring up): "assign"
- register-transfer: "always @()"
- constraints file (Xilinx .ucf) for pin assignment

Introduction to Niklaus Wirth's RISC Processor
- originally a 32-bit virtual machine target for "Compiler Construction"
- RISC vs. CISC; registers vs. stack machine
- Harvard vs. Von Neumann memory architecture
- hardware floating-point option
- now defined in Verilog and implemented on FPGA

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

The RISC Instruction Set
- arithmetic and logic instructions (reg/reg and reg/immediate)
- load and store register to/from data memory (word and byte)
- conditional branch (-and-link)
- that's it! :)

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. ROM
- initialisation of stack and global base
- main loop - output to LED port
- nested delay loops

Sw/Hw Co-Design: Example 1, Pulse-Width Modulation
- demonstration / introduction to class 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
- so, why not do it in hardware: 1-line change to Verilog!
- assign leds = (cnt1[3:0] == 0) ? Lreg : 0;

Sw/Hw Co-Design: Example 2, (Kostenlos!) Light Detector
- photoelectric effect on voltage decay via parasitic capacitance
- alter Verilog to optionally 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, then 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

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]

ETH Vorlesung Systembau / Lecture System Construction
252-0286-00L - Dr. Felix Friedrich, Paul Reed