Customizing RISC-V core using open source tools

Alex Badicioiu, NXP

Agenda

- Chisel HDL
  - Introduction
  - Development environment components and setup
  - Hello World project
- Extending RISC-V core with new instruction
  - riscv-sodor designs
  - New instruction implementation, integration & simulation

Chisel HDL

- Introduction
  - Scala extension to generate hardware description
  - Chisel/Scala statements are mixed - when Scala program runs hardware description output is generated as Chisel statements are executed
  - This hardware description output is called FIRRTL – Flexible Intermediate Form RTL
  - FIRRTL is converted in low level Verilog by a FIRRTL compiler
  - This approach provides powerful design generators and comprehensive testing possibilities

Chisel HDL

- Local development environment setup example
  - Ubuntu 14.0.4
  - Build essential
    - apt-get install build-essential autoconf flex bison
  - Java(OpenJDK1.7)
    - apt-get install openjdk-7-jdk
  - SBT/Scala - https://piccolo.link/sbt-0.13.16.tgz, Scala version 2.11.11
  - Verilator (3.886)
    - git clone http://git.veripool.org/git/verilator
    - git checkout verilator_3.886
    - autoconf && ./configure && make
  - FIRRTL – firrtl, firrtl-interpreter
    - https://github.com/freechipsproject/firrtl.git -3dd921f38f298c7c4aa338e14ac43bc77c652e8c
    - https://github.com/freechipsproject/firrtl-interpreter.git - cea56365cf1dce8dd13fa1379a4f5ed35347ee0b
  - Chisel3
    - https://github.com/freechipsproject/chisel3.git - 8168a8eea6c3465966081c5acd0347e09791361c
  - chisel-testers
    - https://github.com/freechipsproject/chisel-testers.git - f1f2645690de370063af01f86c5fe6e49a462f3b

Chisel HDL

- Build & install Chisel components
  - sbt compile && sbt publishLocal – compiles and registers the component to a local repository (Apache Ivy)
- SBT basic project layout
  - SBT - “make”program for Scala language (build.sbt is the Makefile)
    - start with an working example and customize it if needed
    - src/main/scala – usual location for Scala sources
    - scalaVersion – must match installed Java version
    - libraryDependencies – required libraries
  - usual commands – sbt compile, sbt run , sbt publishLocal

Chisel HDL

- Hello World
  - Hello.scala - Circuit with an 8-bit constant output value
  - build.sbt – dependencies are local chisel3, chisel-iotester installations
  - Generate Verilog or run a test using FIRRTL/Verilog backends

Extending RISC-V

- RISC-V Sodor - https://github.com/ucb-bar/riscv-sodor
  - Educational microarchitectures from UC Berkeley
  - Generated Verilog is fed into Verilator translator to generate C++ sources for simulators
  - Test engine based on riscv-test (https://github.com/riscv/riscv-tests) and riscv-fesvr (front end server) (https://github.com/riscv/riscv-fesvr) projects

rv32_3stage microarch

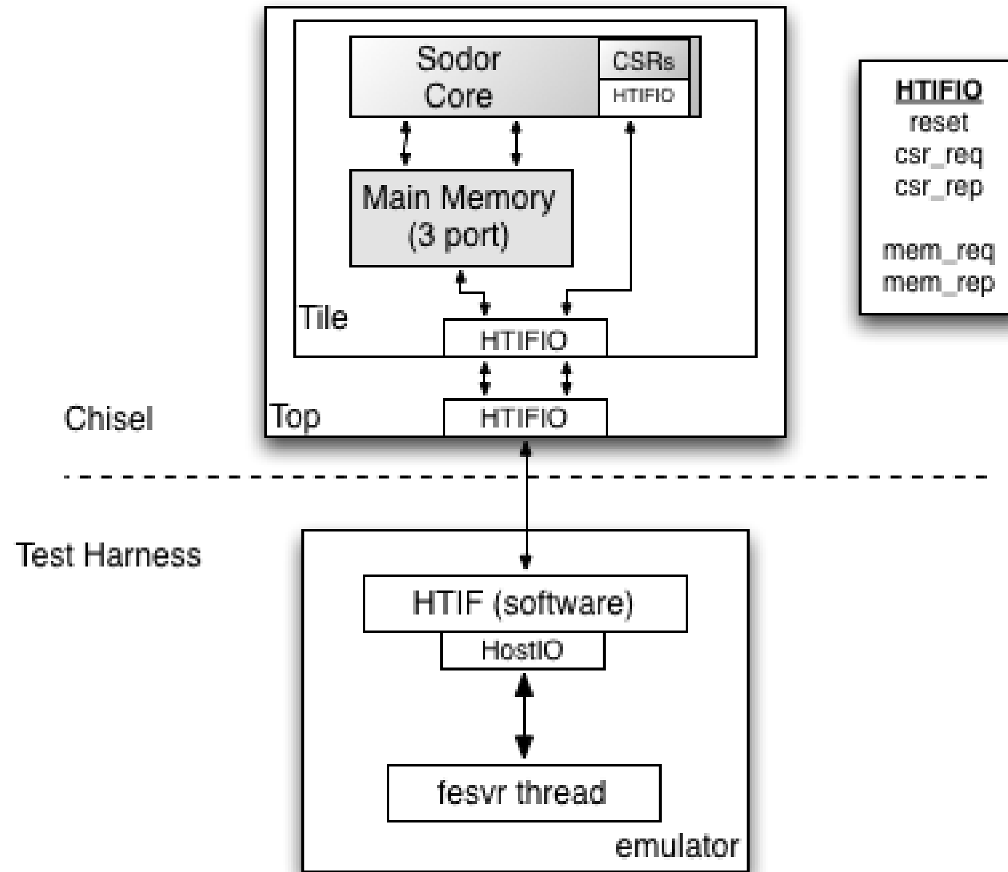
RV32IS *Z-Scale*

br or jmp
jalr
evec
Branch & Jump
TargGen
rs2
rs1
Branch
CondGen
br_eq?
br_lt?
br_ltu?
wb addr
ir[11:7]
pc+4
pc
PC
+4
PC
(BE rdy)
ir[31:12]
JType Sign
Extend
take br,
stall
IR
Memory
resp
data
(BE rdy)
BType Sext
SType Sext
ir[31:12]
UType
ir[31:20]
IType Sext
bypasses
Op2Sel
PC
adder
ALU
ALU
PC
-4
wb_sel
rf_wen
wa
en
wd
Reg
File
ir[24:20]
ir[19:15]
addr
Reg
File
data
imm_u
imm_z
Op1Sel
AluFun
addr
rdata
Memory
wdata
rs2
Decoder
ctrl_valid
Control
Signals
csr_fcn
csr addr
op
wdata
pc
CSRs
rdata
ir[31:20]
addr
xcpt/eret
wb
hazard
stall front-end
(IF/Exe)
val
true
!take_br
val
(BE rdy)
exception
xcpt
exception
OR
eret
hazard / XCPT
ctrl signals
ctrl
bubble
CSRs hold control/
status registers and
other uarch-
counters
Front-end (FE)
Back-end (BE)
NPC
Stage
Fetch Stage
Execute Stage
Writeback Stage
by Christopher Celio

3-stage RV32I core

- rv32_3stage module hierarchy
  - Top – top.scala – contains Scala main function
    - SodorTile – tile.scala
      - Core – core.scala
        - Frontend – frontend.scala (IF)
        - DatPath - dpath.scala (EX & WB)
          - ALU – alu.scala
          - CSRFile – csr.scala
        - CtlPath – cpath.scala (ID)
      - Memory – memory.scala
      - Debug – debug.scala
    - DTM (Debug Transport Module) – debug.scala

Simulator & test env



Simulator & test env

```
VTop dut; // design under test, aka, your chisel code

//Instantiated DTM – loads the program
  dtm = new dtm_t(to_dtm);

 // reset for a few cycles to support pipelined reset
  for (int i = 0; i < 10; i++) {
   dut.reset = 1;
   dut.clock = 0;
   dut.eval();
   dut.clock = 1;
   dut.eval();
   dut.reset = 0;
 }
```

```
//running loop
while (!dtm->done() && !dut.io_success &&
!Verilated::gotFinish()) {
      dut.clock = 0;
      dut.eval();

      dut.clock = 1;
      dut.eval();

      trace_count++;
      if (max_cycles != 0 && trace_count ==   max_cycles)
      {
         failure = "timeout";
         break;
      }
}
```

New RISC-V instruction

- Extending Sodor rv32_3stage
  - New instruction – *ipcsum rd, off(rs)*
  - Computes IPv4 checksum for IPv4 header at memory address *rs + off* and stores the result in *rd*
  - Checksum computation
    - 32bit sum of all 16bit words in the header skipping the checksum field (2*HLEN – 1 16bit words, HLEN=5..15 32bit words)
    - If 32bit sum > 16 bit fold it into halfwords and replace it with the sum of the halfwords
    - Flip each bit in the result to obtain the header checksum
  - Basically a multicycle load + ALU instruction

New RISC-V instruction

- Extending Sodor rv32_3stage – *ipcsum*
  - New module activated when the instruction is in EX stage
  - Generate address to read IPv4 header words from data memory
  - Combinational logic to compute the partial checksum
  - Halt and resume the pipeline
  - Internal registers to store the state (current address, partial checksum, word count, etc)

IpCsum module

- Extending Sodor rv32_3stage – IpCsum interface

```
class IpCsum(implicit conf: SodorConfiguration) extends Module
{
   val io = IO(new Bundle {
      val in   = Input(UInt(conf.xprlen.W))   //input from dmem
      val cs   = Input(Bool())                 //circuit select
      val out = Output(UInt(conf.xprlen.W))   //checksum output
      val done = Output(Bool())               //checksum is done
      val iph = Input(UInt(conf.xprlen.W))    //header start address
      val addr = Output(UInt(conf.xprlen.W)) //address for dmem
   })
```

IpCsum module

- Extending Sodor rv32_3stage – IpCsum implementation
  - Typical FSM
  - State is advanced at each clock cycle if the input from memory is available
  - Output activation state is reached when all the words in the IP header were used for checksum computation
  - Reset to initial state in the WB clock cycle

IpCsum integration

- Extending Sodor rv32_3stage – integration
  - Allocate opcode for new instruction
  - Add an ALU operation code and output signal for *ipcsum* instruction
  - Generate control signals
  - Datapath connection logic
    - Input from memory
    - Module activation and pipeline stall / resume logic
    - Write-back

IpCsum integration

- Opcode, Control path & ALU

```
//allocate opcode & generate control signals
LW      -> List(Y, BR_N  , N, OP1_RS1, OP2_IMI , ALU_ADD , WB_MEM, REN_1, N, MEN_1, M_XRD, MT_W,
CSR.N, M_N),
+ CUSTOM0_RD_RS1 -> List(Y, BR_N  , N, OP1_RS1, OP2_IMI , ALU_IPCSUM ,  WB_MEM, REN_1, N, MEN_1,
M_XRD  , MT_W,  CSR.N, M_N),

//add ALU function code and output signal
object ALU //ALU function codes
{
+ val ALU_IPCSUM = 3.U
}
class ALUIO(implicit conf: SodorConfiguration) extends Bundle {
+ val ipcsum = Output(Bool())
  val out_xpr_length =
-    Mux(io.fn === ALU_ADD || io.fn === ALU_SUB, sum
+    Mux(io.fn === ALU_ADD || io.fn === ALU_SUB ||  io.fn === ALU_IPCSUM, sum
+   io.ipcsum := Mux(io.fn === ALU_IPCSUM, true.B, false.B)
}
```

IpCsum integration

- Datapath

```
class DatPath(implicit conf: SodorConfiguration) extends Module {
+ val ipcsum_a = Module(new IpCsum()) //instantiate module

+ ipcsum_a.io.cs := alu.io.ipcsum && io.dmem.resp.valid //enable operation when executing ipcsum and memory response is valid

+ ipcsum_a.io.in := io.dmem.resp.bits.data //connect memory data output to IpCsum input

+ wb_hazard_stall := wb_hazard_stall_dpath || alu.io.ipcsum &&
!ipcsum_a.io.done //stall as long as executing ipcsum and the operation is not finished (or'ed with original stall logic renamed as _dpath)
}
```

IpCsum integration

- Datapath

```
class DatPath(implicit conf: SodorConfiguration) extends Module {
+ io.dmem.req.bits.addr  := Mux(alu.io.ipcsum && io.dmem.resp.valid,
ipcsum_a.io.addr, exe_alu_out) //data memory address is provided by
IpCsum when executing ipcsum

+ ipcsum_a.io.iph := Mux(alu.io.ipcsum , io.alu.out, 0.asUInt(32.W))
//provide IP header start address from ALU - io.alu.out

wb_wbdata := MuxCase(wb_reg_alu, Array(
+                   (Reg(next=ipcsum_a.io.done) === true.B) ->
                     ipcsum_a.io.out, //write back the checksum in the
                     next cycle after done is asserted
}
```

Test & simulation

- Manual instruction coding

```
/* move ipheader address into x1*/
la x1, ipheader
/* custom opcode for ipcsum - CUSTOM0_RD_RS1 */
off=4                  rs=x1     rd=x2  opcode
000000000100           00001 110 00010  0001011
*/
.word 0x0040e10b
TEST_CASE(1, x2, 0xb861, nop)
```

```
.global ipheader;
ipheader:  .word 0xdeadbeef;
           .word 0x45000073;
           .word 0x00004000;
           .word 0x4011b861;
           .word 0xc0a80001;
           .word 0xc0a800c7;
```

Test & simulation

Cyc= 46 Op1=[0x80002000] Op2=[0x00000004] W[W, 1= 0x80002000] [_,0x0040e10b] 38 **H** PC=(0x80000108,**0x80000104**,0x80000100) [13, 11, 19], WB: DASM(f0408093)

IpCsum: wcnt 0 io.addr 0x80002008 io.in 0x45000073 ipcsum 0x00000000 io.out 0x00000000

Cyc= 47 Op1=[0x80002000] Op2=[0x00000004] W[_, 2= 0x45000073] [_,0x0040e10b] 39 **H** PC=(0x80000108,**0x80000104**,0x00000000) [13, 11, 51], WB: DASM(00004033)

IpCsum: wcnt 1 io.addr 0x8000200c io.in 0x00004000 ipcsum 0x00004573 io.out 0x00000000

Cyc= 48 Op1=[0x80002000] Op2=[0x00000004] W[_, 2= 0x00004000] [_,0x0040e10b] 39 **H** PC=(0x80000108,**0x80000104**,0x00000000) [13, 11, 51], WB: DASM(00004033)

IpCsum: wcnt 2 io.addr 0x80002010 io.in 0x4011b861 ipcsum 0x00008573 io.out 0x00000000

Cyc= 49 Op1=[0x80002000] Op2=[0x00000004] W[_, 2= 0x4011b861] [_,0x0040e10b] 39 **H** PC=(0x80000108,**0x80000104**,0x00000000) [13, 11, 51], WB: DASM(00004033)

IpCsum: wcnt 3 io.addr 0x80002014 io.in 0xc0a80001 ipcsum 0x0000c584 io.out 0x00000000

Cyc= 50 Op1=[0x80002000] Op2=[0x00000004] W[_, 2= 0xc0a80001] [_,0x0040e10b] 39 **H** PC=(0x80000108,**0x80000104**,0x00000000) [13, 11, 51], WB: DASM(00004033)

IpCsum: wcnt 4 io.addr 0x80002018 io.in 0xc0a800c7 ipcsum 0x0001862d io.out 0xb861

Cyc= 51 Op1=[0x80002000] Op2=[0x00000004] W[_, 2= 0xc0a800c7] [_,0x0040e10b] 39 **H** PC=(0x80000108,**0x80000104**,0x00000000) [13, 11, 51], WB: DASM(00004033)

Cyc= 52 Op1=[0x80002000] Op2=[0x00000004] W[_, 2= 0x00000000] [_,0x0040e10b] 39 PC=(0x80000108,**0x80000104**,0x00000000) [13, 11, 51], WB: DASM(00004033)

IpCsum: reset to initial state

Cyc= 53 Op1=[0x00000000] Op2=[0x00000000] W[**W, 2= 0x0000b861**] [_,0x00000013] 39 PC=(0x8000010c,0x80000108,**0x80000104**) [37, 19, 11], WB: DASM(0040e10b)

Test & simulation

- IP checksum C function

```
register unsigned int csum = 0;
 asm("nop");
 csum =  (unsigned short)(iphdr[0] >> 16) + (unsigned short)(iphdr[0]);
 csum += (unsigned short)(iphdr[1] >> 16) + (unsigned short)(iphdr[1]);
 csum += (unsigned short)(iphdr[2]);
 csum += (unsigned short)(iphdr[3] >> 16) + (unsigned short)(iphdr[3]);
 csum += (unsigned short)(iphdr[4] >> 16) + (unsigned short)(iphdr[4]);
 while(csum & 0xffff0000) {
     csum = ((unsigned short)(csum >> 16) + (unsigned short)(csum));
 }
 *result = ~csum;
```

Test & simulation

Cyc= 195 Op1=[0x800029b0] Op2=[0x00000000] W[W, 0= 0x00000000] [_,0x00052303] 162 PC=(0x80001050,0x8000104c,0x80001048) [03, 3, 19], WB: DASM(00000013)

Cyc= 196 Op1=[0x800029b0] Op2=[0x00000004] W[W, 6= 0x73000045] [_,0x00452883] 163 PC=(0x80001054,0x80001050,0x8000104c) [37, 3, 3], WB: DASM(00052303)

Cyc= 230 Op1=[0x8002294e] Op2=[0x00000000] W[W,15= 0xffff61b8] [_,0x00f59023] 196 PC=(0x800010d8,0x800010d4,0x800010d0) [13, 35, 19], WB: DASM(fff7c793)

Cyc= 231 Op1=[0x00000000] Op2=[0x00000000] W[_, 0= 0x8002294e] [_,0x00000013] 197 PC=(0x800010dc,0x800010d8,0x800010d4) [67, 19, 35], WB: DASM(00f59023)

Cyc= 232 Op1=[0x80002684] Op2=[0x00000000] W[W, 0= 0x00000000] [_,0x00008067] 198 R PC=(0x800010e0,0x800010dc,0x800010d8) [13,103, 19], WB: DASM(00000013)