Discussion 5: Connecting to Rocket

CS250 Spring 2016
Christopher Yarp
Work up to lab 4:

- You’ve done a lot of work in labs 1-3
  - Constructed a SHA3 unit from a reference design
  - Implemented unit tests to validate datapath functionality
  - Implemented integration tests to validate the functionality of the unit
  - Added configurable pipelining to the datapath
  - Modified the memory controller to exploit multiple in-flight requests
  - Modified the design to use SRAMS
  - Validated your design through RTL, Post-Synthesis, and Post-PAR simulations
Lab 4

• The SHA3 unit was built and tested in isolation
• Now, it is time to connect it to a processor!
  • The instruction set of the processor is RISC-V
  • The processor implementation is called rocket
  • We will use RoCC to make the connection
Instruction Set Architectures

• Instruction Set Architecture (ISA)
  • Defines the user (programmer) facing instructions available from the processor
  • Often includes details about register files (if used), memory addressing, number representations ...
  • May be paired with or contain an explicit memory model
  • Basically, it is the contract that any processor must fulfill from the programmer’s perspective (the HW/SW bridge)

• Some popular ISAs
  • IA-32 (x86)
  • AMD64 (x86-64, x64, EM64T)
  • ARM/Thumb
  • PowerPC
  • RISCV

• The ISA is separate from the implementation
  • Many processors implement x86-64
    • Intel Core Series
    • AMD Athlon (newer versions)
  • Regardless of which chip you buy, it should run programs compiled for its ISA
Typical Software Development

- **C**
  - Code is written in a High Level Language (ex. C)

- **ASM**
  - Code is compiled into ISA instructions
  - Most ISAs have an assembly language

- **Machine**
  - Assembly is assembled into machine code (binary)
  - Can be executed by the processor
C to Assembly

C Program

#include <stdint.h>

int64_t add3Nums(int64_t a, int64_t b, int64_t c)
{
    int64_t d = a+b+C;
    return d;
}

Assembly

.include "hello.c"
.text
.align 2
.globl add3Nums
.type add3Nums,
@function add3Nums:
    add a0,a0,a1
    add a0,a0,a2
    ret
.size add3Nums, .-
.add3Nums
.ident "GCC: (GNU) 5.3.0"
RISC-V Instructions/Formats

Instruction Formats:

<table>
<thead>
<tr>
<th>Field</th>
<th>Bits</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>funct7</td>
<td>7</td>
<td>R-type</td>
</tr>
<tr>
<td>imm[11:0]</td>
<td>12</td>
<td>I-type</td>
</tr>
<tr>
<td>imm[12:10:5]</td>
<td>12</td>
<td>SB-type</td>
</tr>
<tr>
<td>imm[31:12]</td>
<td>12</td>
<td>U-type</td>
</tr>
</tbody>
</table>

Some Example Arithmetic Instructions (Not full List)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD rd,rs1,rs2</td>
<td>010011</td>
<td>0110011</td>
</tr>
<tr>
<td>SUB rd,rs1,rs2</td>
<td>010011</td>
<td>0110011</td>
</tr>
<tr>
<td>SLL rd,rs1,rs2</td>
<td>010011</td>
<td>0110011</td>
</tr>
<tr>
<td>SLT rd,rs1,rs2</td>
<td>010011</td>
<td>0110011</td>
</tr>
<tr>
<td>SLTU rd,rs1,rs2</td>
<td>010011</td>
<td>0110011</td>
</tr>
<tr>
<td>XOR rd,rs1,rs2</td>
<td>010011</td>
<td>0110011</td>
</tr>
</tbody>
</table>

Custom Instructions with RoCC

A full list for the base ISA can be found on page 50 of User Level ISA Specification v2.0 at http://riscv.org/specifications/
Using Custom Instructions in Programs

• Custom Instructions are understood by the RISC-V assembler
• They are denoted: custom0, custom1, custom2, custom3
• How do you call your accelerator from a C program?
• Simple Case: Inline Assembly
Inline Assembly

• Syntax

asm [volatile] (AssemblerTemplate
  : Output Operands
  [: InputOperands
    :Clobbers])

Volatile – means the assembly instruction has side effects and should not be removed by the compiler

Info from: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
Inline Assembly for Custom Instructions

Assembly Format:

```
custom0 rd rs1 rs2 functCode
```

Example from SHA3:

```
asm volatile ("custom0 0, %[msg_addr], %[hash_addr], 0"
    : : [msg_addr]"r"(&maddr),
        [hash_addr]"r"(&haddr));
```

This example had no rd

maddr and haddr are variables in the C program

The & takes the address of these variables

The "r" is a constraint that specifies that a register operand is allowed

Info from: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
Inline Assembly With Return

• Example

```c
asm volatile ("custom0 [%rd], [%rs1], [%rs2], 0"
: [rd]"=r"(rd)
: [rs1]"r"(rd), [rs2]"r"(rs2));
```

• When writing, and = or + is used at the start of the constraint
  • = when a variable is being overwritten
  • + when reading and writing

• = can be used when the write operand is also one of the inputs

Info from: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
Fencing

• When calling an assembly instruction, you may need to call fence first
  • \texttt{asm volatile("fence")}

• Memory transactions are not always complete when an assembly instruction is called

• Fence forces the processor to wait for memory operations to complete before proceeding
RISC-V Toolchain

- RISC-V provides a full software toolchain for you
  - gcc/g++
  - LLVM/clang
  - ISA simulator (spike)
    - Allows you to test programs written for an ISA before a chip is even available.
    - Relies on a model of what different instructions do
    - You will extend spike in lab4

- Since the servers used for the lab are x86_64 machines, you will be using a cross compiler
  - A compiler that produced a binary for a different processor than used by the development machine
Rocket Emulation

• A C++ emulator and RTL cycle accurate simulator can be compiled for rocket-chip

• You can use these emulators/simulators to run RISC-V binaries!

• The C++ emulator is typically much faster than the RTL simulator
Running Bare-Metal with the Proxy Kernel (pk)

- You will be running rocket **bare-metal**
  - This means without an operating system
- Several C functions rely on an operating system being present
  - To execute system calls
  - To manage page faults
  - And several other things
- The Proxy Kernel (pk) is a light weight piece of code that implements the essential features of an OS required for a simple C program to run