Teaching basic computer architecture, assembly language programming, and operating system design using RISC-V

Liliana Andrade, Mounir Benabdenbi,
Olivier Muller, Frédéric Rousseau,
Frédéric Pétrot

🏠 tima.imag.fr/sls/people/petrot

✉ frederic.petrot@univ-grenoble-alpes.fr

The RISC-V Week
Paris • October 1 to 3

Outline

1 Preamble
2 Introduction
3 Overview of the curriculum
4 Class specifics
5 Take away

Preamble

Today's view on computers and processors

What computer a typical first year Computer Science (CS) students owns?

- You name it! But all are x86-64 based!

What processors do typical first year CS students know?

- Intel Core Ix, sometimes named x86 (or x86-64 for the connoisseurs)
- AMD Ryzen, but wait, this is also an x86!
- Atom, that's in my tablet, x86, uh, ...

Other devices

they know phone brands, but not the processors in them

1991 1992 1993 ... 2015 2016 2017

Intel did it right!

Preamble

Intel did it right, but ...

- "[x86] mov is Turing-complete", Stephen Dolan, 2013.
  Actual code generator by Christopher Domas
  `https://github.com/xoreaxeaxeax/movfuscator`

  But wait, why bother with instructions?
- "[x86] Page-faults are Turing-complete", Julian Bangert and Sergey Bratus, 2015.
  Actual code generator `https://github.com/jbangert/trapcc`

Although x86 is the mainstream desktop computer architecture, it may be worth using something else as a pedagogical vehicule!

Outline

1 Preamble
2 Introduction
3 Overview of the curriculum
4 Class specifics
5 Take away

Introduction

Teaching computer engineering using RISC-V

Target students

Bologna style	L3/M1/M2 in Computer Science or Electrical Engineering
	In reality engineering schools in Grenoble
	Ensimag, computer science
	Phelma, electronic and micro-electronic engineering
	Polytech Grenoble, electronics engineering and industrial IT

Goal

Unifying a set of classes using various processors under the RISC-V umbrella

Outline

1 Preamble

2 Introduction

3 Overview of the curriculum

4 Class specifics

5 Take away

Overview of the curriculum

Basic computer architecture
(Computer Science, 3rd year, ≈250 students, Electronics Engineering, 5th year, ≈20 students)

- digital circuit design for computer scientists
- ISA interpretation using a finite state machine + data-path

Assembly language programming
(Computer Science, 3rd year, ≈250 students)

- basic instruction usage
- function calling conventions and C ABI

Grenoble INP Ensimag

Overview of the curriculum

**Computer architecture
(Computer Science, 4th year, ≈40 students)**

- 5 pipeline stages processor
- multiprocessor and atomic operations

**Operating system implementation
(Computer Science, 4th year, ≈75 students)**

- boot, interrupt, kernel threads
- virtual memory, processes

Overview of the curriculum

System level design in SystemC (Computer Science, 5th year, ≈20 students)

- hardware modeling in SystemC, transaction level modeling
- modeling in SystemC with native or cross-compiled software

HW/SW system integration (Electrical Engineering, 5th year, ≈40 students)

- performance analysis of a cache-based multiprocessing system
- HW/SW integration on FPGA

Overview of the curriculum

	L3/Bachelor	M1/Master	M2/Master
Ensimag	Basic computer architecture	Operating system implementation	System level design in SystemC
	Assembly language programming	Computer architecture	
Polytech			Assembly language programming
			Basic computer architecture
Phelma			HW/SW system integration

Outline

1 Preamble

2 Introduction

3 Overview of the curriculum

4 Class specifics

5 Take away

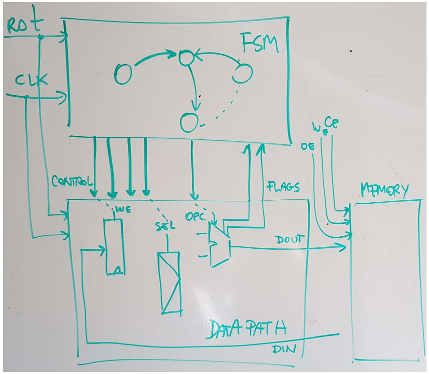
Basic computer architecture

Class and practical objectives

Write VHDL code interpreting instructions to understand why a computer needs to be powered, how it might execute a program, and why it is not indefinitely fast

What do we provide the students?

- VHDL of an FSM squeleton and a data-path with a few missing parts
- ordered list of instructions (encoding+behavior) to implement
- test environment to check their implementation



Basic computer architecture

What do we expect from the students?

- Proper decoding and execution of instructions with FSM+data-path
- Add missing parts in the data-path
  - condition computations for branches
  - control and status registers
  - interruptions
- Mapping on Xilinx FPGA board with 100 MHz minimal frequency target
- Own unitary test for each instruction
  - shall not depend upon future implemented instructions
  - are pretty easy develop as the micro-architecture is naïve

Basic computer architecture

MIPS I neat, not so hype anymore

- all instructions are 32-bit
- 38 instructions including `eret`
- 3 instruction formats
- 5 immediate formats, zero or sign extended, 16-bit or 26-bit
- reg/reg instructions update `rd`, reg/imm instructions update `rt`
  ⇒ `sllv` and `srav` instead of `slli/srai`
- delay slot, hidden from the students
- branch target computed using pc + 4

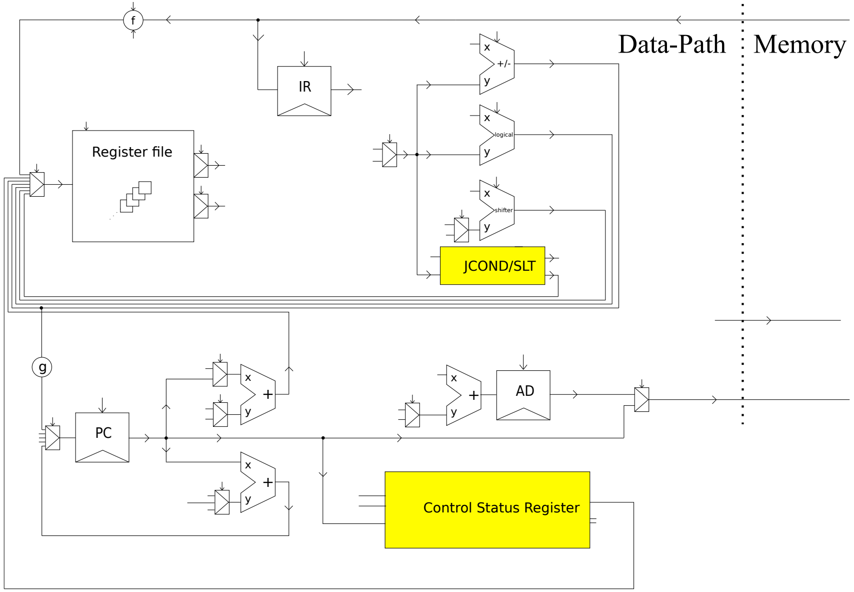
⇒ both former specificities assume pipeline implementations

RISC-V rv32i stylish and trendy

- all instructions are 32-bit
- 32 integer instruction including `mret`
- 6 instruction formats
- 6 immediate formats, sign extended, weirdly built, 12-bit or 20-bit
- all instructions update rd
- no delay slot
- branch target computed using pc as is

⇒ no specific implementation strategy inferred

Basic computer architecture



Basic computer architecture

Conclusion

Complexity is alike, but RISC-V has a few interesting features:

- all instructions update rd
- no delay slot
- branch computed using pc directly
- strange ways to compute the immediats
  example: branch offset $\Leftarrow (IR_{31}^{20} \parallel IR_7 \parallel IR_{30\ldots25} \parallel IR_{11\ldots8} \parallel 0)$

And is much more attractive to our students
google "RISC-V processor" : "Environ 9.050.000 résultats (0,60 secondes)"
google "mips processor" : "Environ 3.380.000 résultats (0,62 secondes)"

Small demo

Sneaking into some code
FPGA demo

Introduction to asm programming (3rd year, ≈250 students)

Objective of the class

- writing simple programs in asm
- understand variable classes: data, heap, stack
- systematically translate 'C' statements in asm
- ABI oriented towards function calling conventions

What did we do in the past?

- used an x86 subset
  guaranteed headache, 13 different ABI running in the wild!
- moved to MIPS ISA, excluding unaligned word accesses, and MIPS o32 ABI

Introduction to asm programming (3rd year, ≈250 students)

What do we provide the students?

- Cross-development environment
- QEMU mimicking system of the “Basic Computer Architecture” class
- C source code of the exercises

What do we expect from the students?

- assembly code written as literally as possible
  e.g. as generated by `gcc -O0`
- agile usage of the cross-dev environment
  particularly `gdb` using remote connection on `QEMU`
- ability to call and be called from C functions
- understand low-level interrupt service routines

Introduction to asm programming (3rd year, ≈250 students)

MIPS I ABI in a nutshell

- all instructions but stores have their results in right-hand-side
- registers have hardware and software names
  `$0, ..., $31` vs `$zero, $at, $v0, ..., $a0, ..., $t0, ..., $s0, ...`
- register `$zero` writable but always reads as zero
- 4 first arguments in `$a0, ..., $a3` and return value in `$v0`
- all jumps followed by a `nop` to avoid explaining the delay slot
- macros make use of the implicit register `$1` (`$at`)

Introduction to asm programming (3rd year, ≈250 students)

- macro accept weirdly written asm lines
  `addi $t0, 0xf00d` ⇒ `addi $t0, $t0, 0xf00d`
  `lw` and `sw` accept constants and symbols as arguments
  `lb $t0, 0xdeadbeef` ⇒ `li $t0, 0xdeadbeef / lb $t0, 0($t0)`
  `sw $t0, variable` ⇒ `la $at, variable / sw $t0, 0($at)`
  But who knows if that is what the student meant to write ???
- sign extended or zero extended 16-bit immediates
  visual instruction type and constant binary decoding easy for the teacher
  `addi $v0,$v0,0xffffdead` ⇒ "operand out of range" although sign extended
  `ori $v0,$v0,0xdead` ⇒ no error although zero extended
- interrupt/exception/traps all jump at same address
  ⇒ everything done in software
- access to cause, status and timer related registers using two simple instructions `mfc0/mtc0`

Introduction to asm programming (3rd year, ≈250 students)

RISC-V psABI quite similar in spirit to MIPS

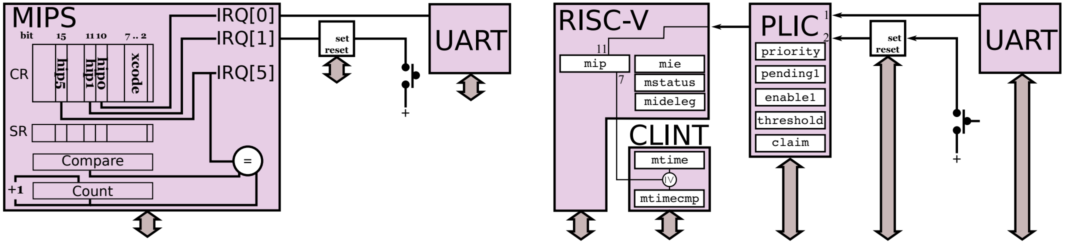
- all instructions but stores have their results in right-hand-side
- registers have hardware and software names
  `x0, ..., x31` vs `zero, ra, ..., a0, ..., t0, ..., s0, ...`
- register `zero` writable but always reads as zero
- 20 and 12 bit immediates always sign extend
  `addi x31, x31, 0xbad` ⇒ "invalid operand error"
  `0x00000bad` ≠ `0xffffbad`
- even stranger:

```
li x31, 0xdeadbeef => lui  x31,      0xdeadc
                      addi x31, x31, 0xffffeef
```

- `sw` and `lw` accept a symbol as argument, but not a constant

Introduction to asm programming (3rd year, ≈250 students)

- 8 first arguments in `a0, ..., a7` and return value in `a0`
- no delay slot to hide
- access to cause, status and timer related registers using `csrrw` instructions
- only machine mode interrupt/exception/traps presented, configuration without vectors ⇒ everything done in software
- CLINT and PLIC make things more complex than MIPS to handle simple timer interrupt



Introduction to asm programming (3rd year, ≈250 students)

Stack layouts

MIPS I r3000

high	5th parameter	other ones above
addresses	room for `$a3`	needed only if callee
↓	room for `$a2`	wants them back after
↓	room for `$a1`	calling an other function
↓	room for `$a0`	`$sp` in caller (f)
↓	`$ra`	return address (g)
↓	registers to be	
↓	saved	
↓	local variables	
↓	parameter $n-1$	when calling
↓	. . .	an other
↓	parameter 5	function (h)
↓	room for `$a3`	
↓	room for `$a2`	
low	room for `$a1`	for the next call (h)
addresses	room for `$a0`	`$sp` in callee (g)

RISC-V rv32i

high addresses	f local variables	`sp` in calller (f)
↓	`ra`	return address
↓	other registers	
↓	to save	
↓	g local variables	
↓	parameter $n-1$	preparing call to
↓	...	next function h
↓	parameter 8	`sp` in callee (g)
low addresses	...	

A bit easier to explain to students

Introduction to asm programming (3rd year, ≈250 students)

Conclusion

Complexity is alike, but RISC-V rv32i has a few interesting features:

- (very) low integer instruction count
- integer calling conventions pretty simple for fixed number of parameters
- no delay slot
- an unusual pc-relative instruction `auipc`

Small demo

Small function example
QEMU demo

Operating System Implementation (4th year, ≈75 students)

Objective of the class

- understand what happens when a computer is turned on
- understand virtual memory to physical memory translation
- learn to implement:
  boot, kernel threads,
  page tables, frame allocation (overlays ⇒ no file system, no page faults)
  user processes, queues, shared memory, ...
- do all that on RISC-V 64 using QEMU sifive_u board

What did we do (and still do) before?

- originally developed for x86-32
- hiding quite a few stuff under the carpet
- still in use: only 2 groups over 8 did it on RISC-V last year

Operating System Implementation (4th year, ≈75 students)

What do we provide the students?

- skeleton of code, with set of (complex) Makefiles
- header files with helper functions
  inline asm stuff, Linux priority queues, ...
- ≈ 15 userland tests stressing the implementation

What do we expect from the students?

- handle timer interrupts
- build all OS functions first in kernel mode
  starting by process creation and context switch
- add (limited) support for virtual memory
- understand that there is one page table structure per process
- add system calls to wrap the kernel OS functions

Operating System Implementation (4th year, ≈75 students)

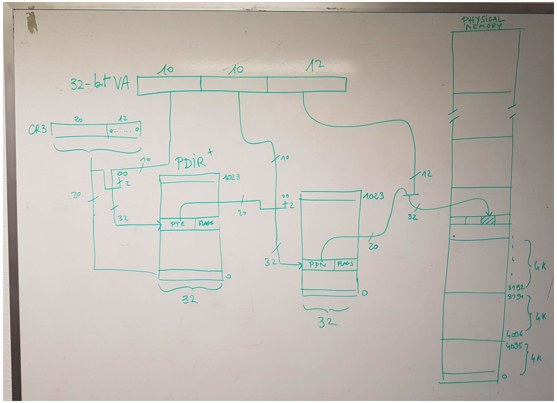
What changes when using RISC-V

- move to 64-bit
- boot phase can be fully written by the students
  no gory details of x86 legacy to skip over
  avoid hardwired stuffs that are hardly explainable: `idt`, `gdt`, `tss`, ...
  ⇒ still need to configure the `pmp` areas
- simpler interruption mechanism and implementation of system calls
  no implicit push of things on stack
  ⇒ interruption setup a bit complex: `m[ei]deleg`, shadow registers, ...

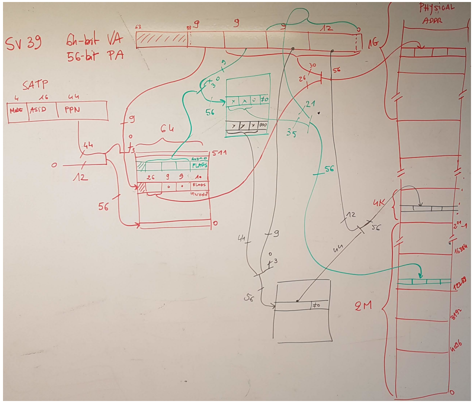
Operating System Implementation (4th year, ≈75 students)

And page tables, somehow more complex on RISC-V

x86 page tables



RISC-V SV39 page tables



Operating System Implementation (4th year, ≈75 students)

Conclusion

x86-32 vs RISC-V 64 fight not over!

x86	students like x86 as they know its name
x86	page table structure
x86	see only kernel and user mode
RISC-V	boot process from start address user mode
RISC-V	no weird hw tables to update here and there
RISC-V	no implicit stuff pushed on or popped from the stack

Overall different, allowing to go a bit more in depth on hw related matters

Small demo

Context creation and context switch

Computer architecture (4th year, ≈35 students)

Objective of the class

- detail how simple 3/4/5/6 stage pipeline processors work
  bypasses, interlocks
- explain cache coherency and memory consistency issues and solutions
  MSI, MESI protocols, plus atomic operation support

What did we do (and still do) before?

- CAAQA with 5-stage pipeline MIPS, like everyone does!

Computer architecture (4th year, ≈35 students)

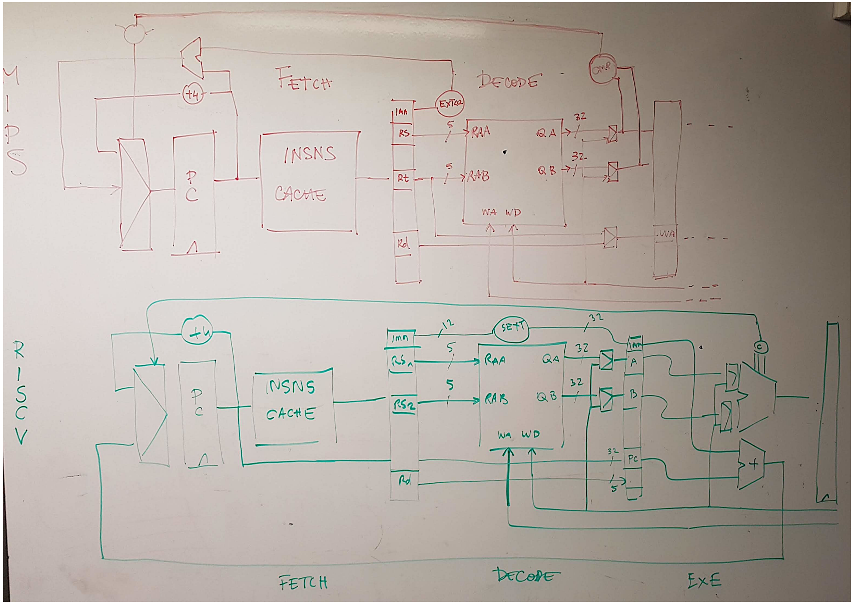
Main changes due to RISC-V in classical 5-stage pipeline

On branches

- target is `pc + offset` (was `pc + 4 + offset`)
- non-conditional branches have no delay slot
  $\Rightarrow$ kill (at least) 1 following instruction
- conditional branches have no delay slot either, have “complex” conditions and resolved late
  $\Rightarrow$ kill 2 following instructions if branch taken

$\Rightarrow$ Quite expensive, ... or add a branch predictor

Computer architecture (4th year, ≈35 students)



Computer architecture (4th year, ≈35 students)

Conclusion

Simple 5-stage pipeline implementation of RISC-V is:

- less efficient or harder
  branch instructions will increase the CPI
  or a branch predictor must be added
- a bit bigger, because pc must be propagated
- but the critical path should be shorter

Numerous atomic memory operations to present

No demo

Lucky you!

System level design in SystemC (5th year, ≈20 students)

Objective of the class

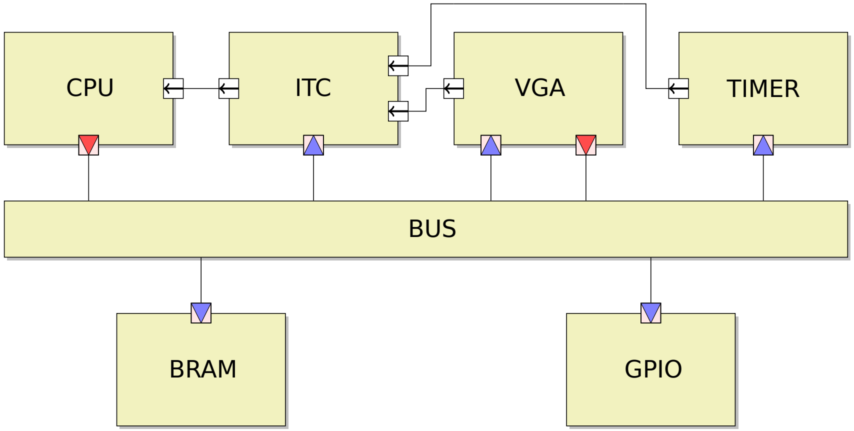
Class targeting an in-depth presentation of SystemC TLM concepts and modeling

- introduce the notion of virtual prototype
- show how to model and simulate a digital system in SystemC before its actual implementation
  - using "native" simulation, through dedicated hardware abstraction layer APIs
  - using an instruction accurate simulator
- build the same system on a (cheap) FPGA board and run the same software just use another implementation of the HAL

What did we do before?

- Originally developed around Xilinx' microblaze
  - SystemC model from the SoCLib library
  - microblaze RTL model from Xilinx

System level design in SystemC (5th year, ≈20 students)



System level design in SystemC (5th year, ≈20 students)

Using RISC-V

Interest

- very parameterizable architecture
- compressed instructions

But needs

- simple SystemC compatible ISS
  ⇒ developed a SoCLib rv32imafc model (machine mode only)
- synthesizable core on Xilinx FPGAs
  ⇒ have (a very very slow) one from Basic Architecture class, good enough

Note that keeping in pace with Vivado hurts, ...

Small demo

Native and cross-compiled top level
System execution

HW/SW System Integration with RISC-V (5th year, ≈40 students)

Objective of the class

Teaching the link between HW and SW in SoCs

- SW environment (assembly, linker, ISA simul., compil., debug)
- HW environment (SoC generation, emulation, ...)
- exceptions, interrupts and traps
- multi-tasking, multi-processing and memory coherence
- development and integration of a custom peripheral with its HAL
- HW and SW mapping $\Rightarrow$ C application on FPGA

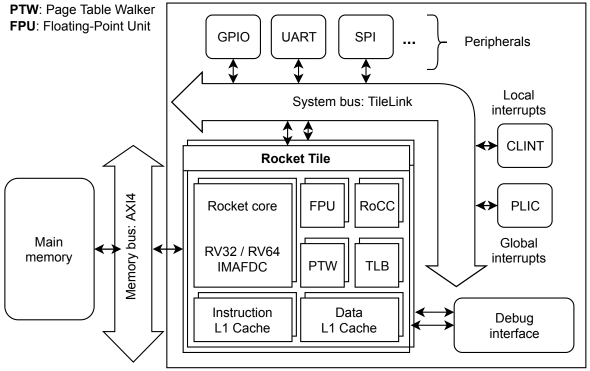
HW/SW System Integration with RISC-V (5th year, ≈40 students)

Needs

- tunable, fast to simulate HW/SW platform
- open source and synthesizable HW/SW SoC ⇒ FPGA evaluation
- platform widely accepted and well supported

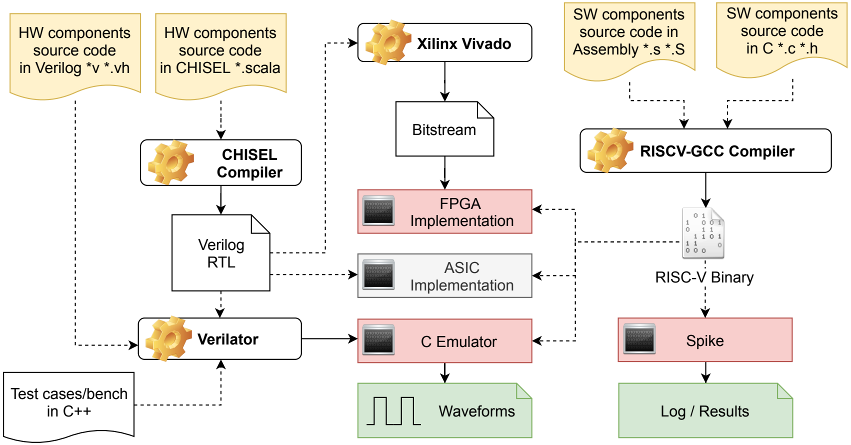
⇒ Berkeley's RISC-V based Rocket Chip SoC generator

TLB: Translation Lookaside Buffer
RoCC: Rocket Custom Coprocessor
PTW: Page Table Walker
FPU: Floating-Point Unit
CLINT: Core Local Interruptor
PLIC: Platform-Level Interrupt Controller



HW/SW System Integration with RISC-V (5th year, ≈40 students)

Design Flow



HW/SW System Integration with RISC-V (5th year, ≈40 students)

Show impact of array placement in memory on L1 collision

L1 data-cache: 8 kB, 256 blocks, 32-byte each
x, y arrays of 4096 elements of 32-bit
address of x is `0x8101_0000`, address of y is `0x8101_4000`

```
loop: for (i = 0; i < 4096; i++)
          s += x[i] + y[i];
```

No data reuse
Only line "prefetch" effect

- paper analysis of execution to evaluate dcache miss rate
- execution with spike to gather statistics
- propose a better placement of y to avoid collision
  hint: make sure x and y never share a line

Small demo

Cache statistics with `spike`, with the bad choice and a good choice

HW/SW System Integration with RISC-V (5th year, ≈40 students)

Real-time multi-tasking

Students

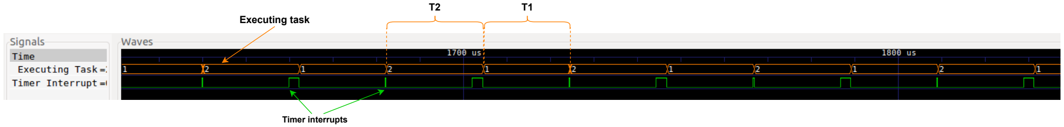
- analyze and complete assembly code
  - memory tasks allocation, init stacks, ...
  - save/restore contexts, switch tasks, scheduling, ...
  - time allocation: give #ticks per task
- code two simple tasks in C, *e.g.* increment in turn a shared variable
- simulate and debug using Spike simulator: functional validation

HW/SW System Integration with RISC-V (5th year, ≈40 students)

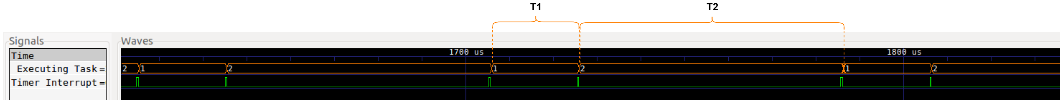
Real-time multi-tasking

- simulate the HW using the cycle-accurate bit-accurate C++ Emulator

T1 = T2 = 100 ticks



T1 = 100 ticks, T2 = 300 ticks



Outline

1 Preamble

2 Introduction

3 Overview of the curriculum

4 Class specifics

5 Take away

Take away

Momentum around RISC-V brings in some freshness

Opportunity to renew classes

Not a revolution: basics are basics, but:

- benefit from the hype
- make student aware that x86-64 doesn't rule them all
- escape complex CISC or complex RISC ISA

Benefit for teachers

- clean and orthogonal ISA (at least for rv32i)
- simple integer calling conventions
- stable cross-development tools, including simulators
- actual implementations on FPGA (even too many perhaps!) and ASIC

$\Rightarrow$ useful replacement to a mix of x86/MIPS/whatever all along the curriculum

Take away

Fear

Are we taking a reckless risk(-v)?

Thanks

Students/Engineers/PhD that have contributed to the classes

Noureddine Ait-Said	(IRT)
Mathieu Barbe	(TIMA)
Amaury Butaux	(TIMA)
Marius Leblanc	(TIMA)
Loïc Jovanovic	(TIMA)
Robin Stieglitz	(IDEX)
Arthur Vianes	(IDEX)
Benoît Wallon	(TIMA)

Sponsors

IDEX Université Grenoble Alpes *Université de l'innovation*

NANOELEC.

tima CNRS - Grenoble INP - UGA

Grenoble INP Ensimag