RISC-V: The Free and Open RISC Instruction Set Architecture

ARJAN BINK | NOVEMBER 1 2019

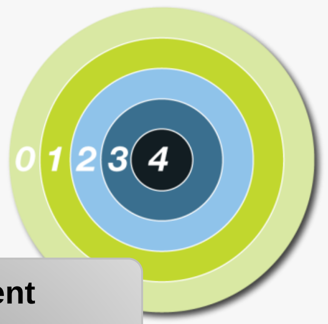
Content

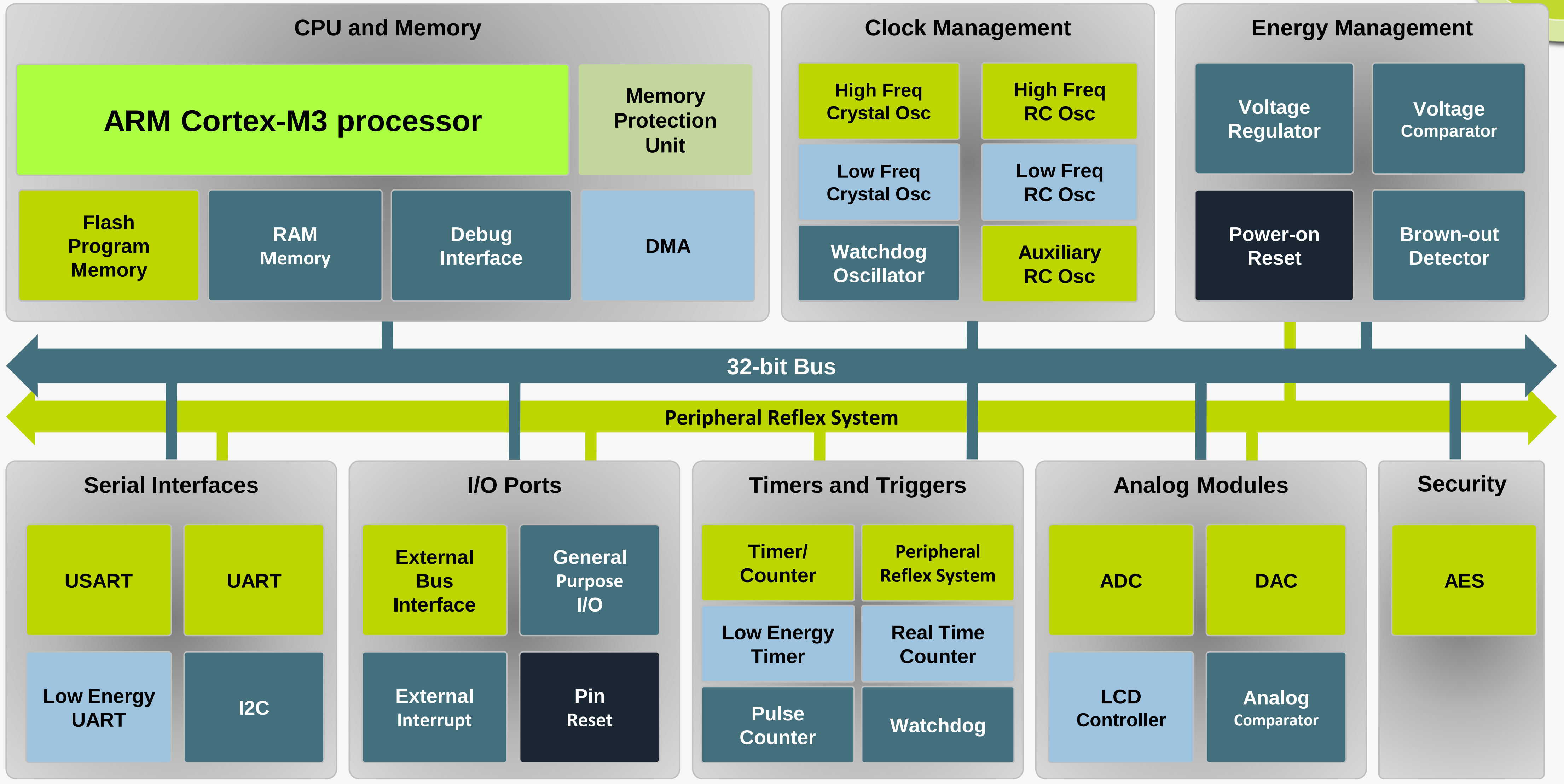
- Silicon Labs
  - Oslo office (former Energy Micro)
- Introduction to the RISC-V Instruction Set Architecture (ISA)
  - Base ISA
  - Standard extensions
- RISC-V cores
  - Commercial & Open Source
  - OpenHW
  - PULP’s RI5CY CPU (ETH Zurich)
    - Custom extensions
- Summer internship @ Silabs Norway
  - RISC-V Compliance
  - RI5CY bus interfaces

Company background

- Global mixed-signal semiconductor company
  - Founded in 1996; public since 2000 (NASDAQ: SLAB)
  - ~1,500 employees and 16 R&D locations worldwide
- Track record of innovation & differentiation
  - 20 year pioneer in mixed-signal and RF technologies
  - Fabless model. 7B+ devices shipped & 1,700+ patents

EFM32 Gecko microcontroller (MCU) – Energy Micro - 2010





Global Semiconductor Alliance

2015, 2016, 2017 & 2018 MOST RESPECTED SEMICONDUCTOR COMPANY

20 Years of Connectivity

A track record of multiple industry first, transforming or disrupting large markets

Founded	HQ	R&D	Size	Nasdaq
1996	Austin, Texas, USA	16 global locations	1500 people	SLAB

PC Modem
Breakthrough soft modem

Crystal Oscillators
Ultra-low jitter XOs and VCXOs

Aero Transceivers
Dominated GSM cellular market

FM/AM Tuners
1st CMOS tuner for CE & automotive

TV Tuners
Market leader since 2011

ZigBee SoCs
Mesh networking market leader

Thread Protocol
Founding member of Thread Group

Pearl/Jade Gecko
Next-gen EFM32 MCUs with HW cryptography

1996 → 2016

RF Synthesizers
1st RF CMOS device in any phone

High-performance Clock Generator
Disruptive *DSPLL*®
architecture

8-bit MCUs
High-performance 8051 core and analog peripherals

EZRadio® Transceiver
Sub-GHz short-range wireless

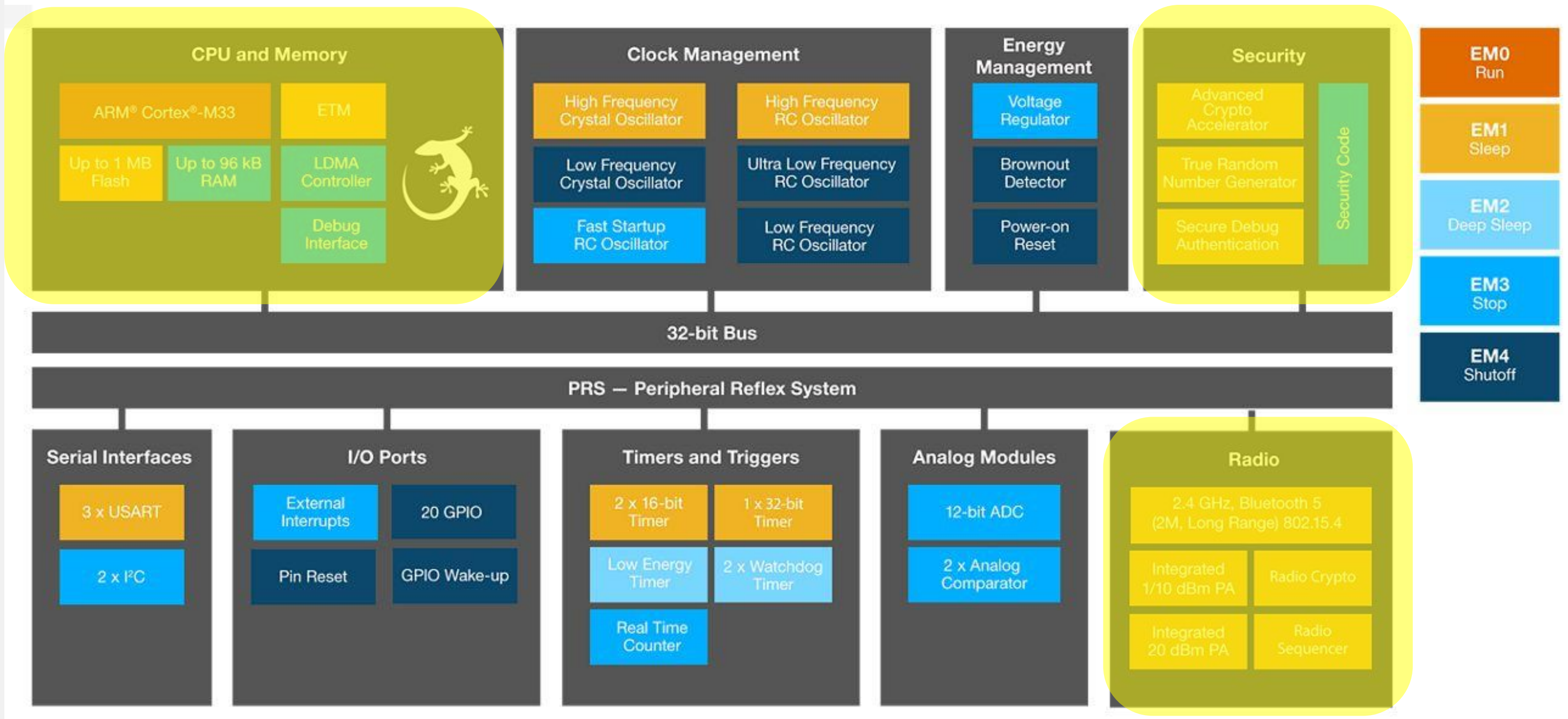
EFM32 MCUs
Energy-friendly 32-bit Gecko technology

Simplicity Studio
1st integrated MCU & RF development environment

Wireless Modules
Bluetooth, Wi-Fi and ZigBee modules and protocol stacks

Wireless Gecko
Multiprotocol, multiband SoC portfolio

EFR32MG21 Series 2 Multiprotocol Wireless SoC - 2019



- Multiple systems with multiple cores
  - Host
    - ARM Cortex-M33
    - Memories (Flash + RAMs)
    - Analog + digital peripherals
  - Secure Element
    - Hardware Cryptographic Acceleration
      - AES (128/192/256), SHA-1, SHA-2 (SHA-224/SHA256), ECC (256-bit), ECDSA (256-bit) and ECDH (p192, p256), HMAC, J-PAKE
    - True Random Number Generator (TRNG)
    - Secure Boot
    - Secure Debug / Debug Access Control
    - Unique ID
  - Radio
    - Zigbee, Thread and Bluetooth mesh

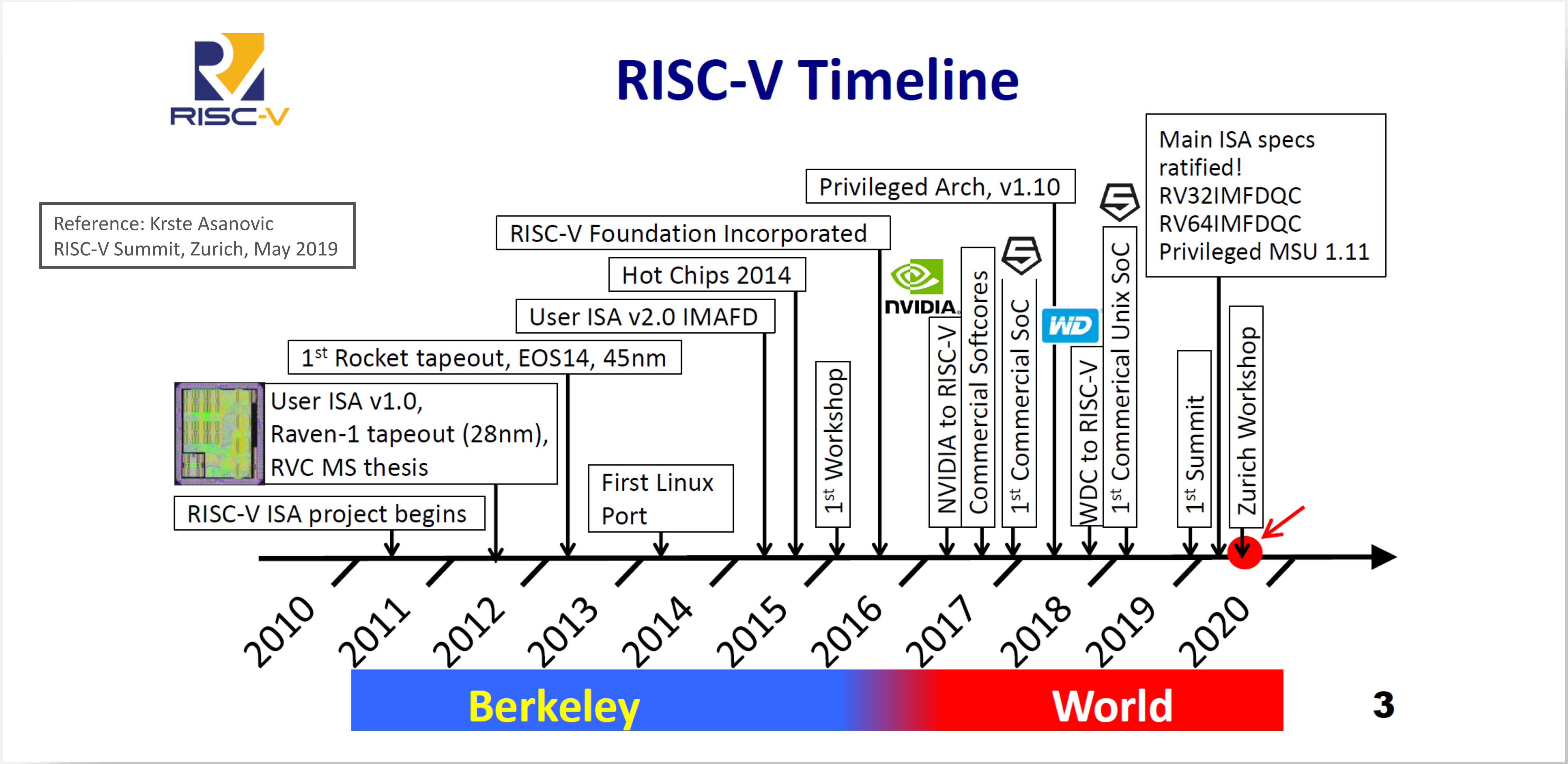
Introduction to the RISC-V Instruction Set Architecture

RISC-V and RISC-V foundation

- RISC-V is an Instruction Set Architecture (ISA)
  - Free and open ISA
    - Partitioned into unprivileged spec, privileged spec, external debug spec
      - https://riscv.org/specifications/
      - https://riscv.org/specifications/privileged-isa/
      - https://riscv.org/specifications/debug-specification/
    - Defines 32, 64 and 128 bit ISA
    - No implementation, just the ISA
  - Born in academia and research in 2010
    - Computer Science Division @ University of California, Berkeley

- RISC-V foundation
  - Governs the RISC-V open standard
  - Founded in 2015
  - More than 275 members
  - Non-profit corporation controlled by its members
    - Directs the future development
    - Drives the adoption of the RISC-V ISA
  - Board of Directors with representatives from
    - Bluespec, Inc.
    - Google
    - Microsemi
    - NVIDIA
    - NXP
    - University of California, Berkeley
    - Western Digital

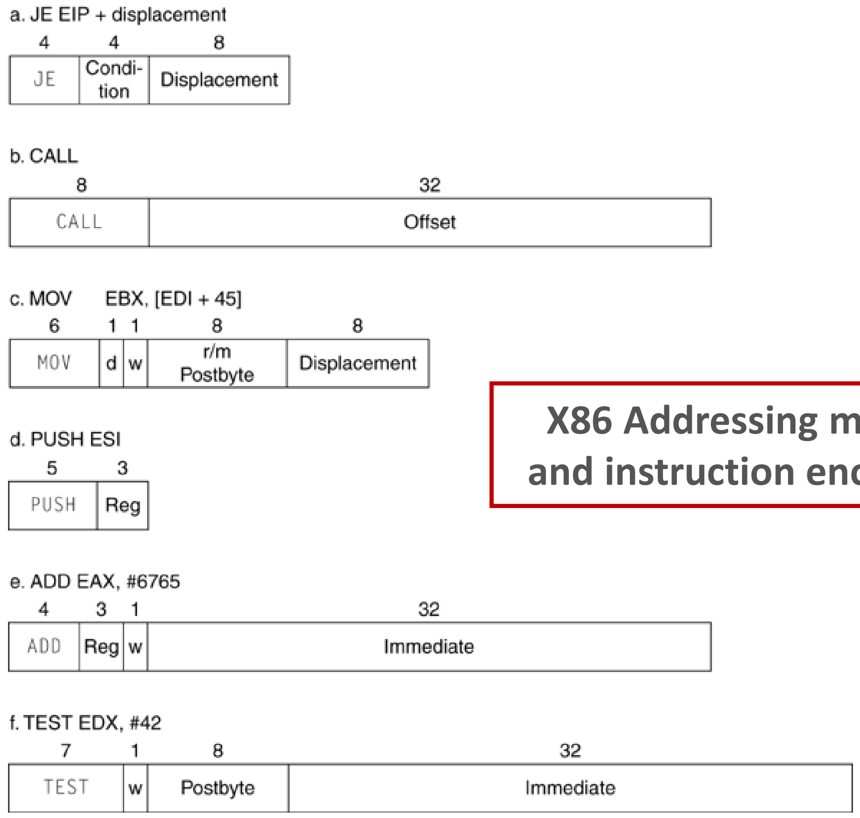
RISC-V History



What is Instruction Set Architecture (ISA)?

- A contract between the SW and HW
  - SW in this case is the compiler/assembler/operating system
  - HW in this case is the micro-architecture
    - Number of pipeline stages, branch prediction
    - Speculative, out-of-order execution, Super-scaler (static, dynamic)
- Micro-architecture PPA are influenced by the choice of ISA
  - Simplicity of instruction decode
  - Pipeline stall scenarios
- Early days ISA's were often focused on code-size
  - Variable length instructions
  - Arithmetic operands from register and memory
  - Often were called Complex Instruction Sets
- These HW/SW contracts led to complex HW design
  - Performance (no. of cycles to complete a program) was poor
    - Difficult instruction decoding
    - Too many pipeline stalls

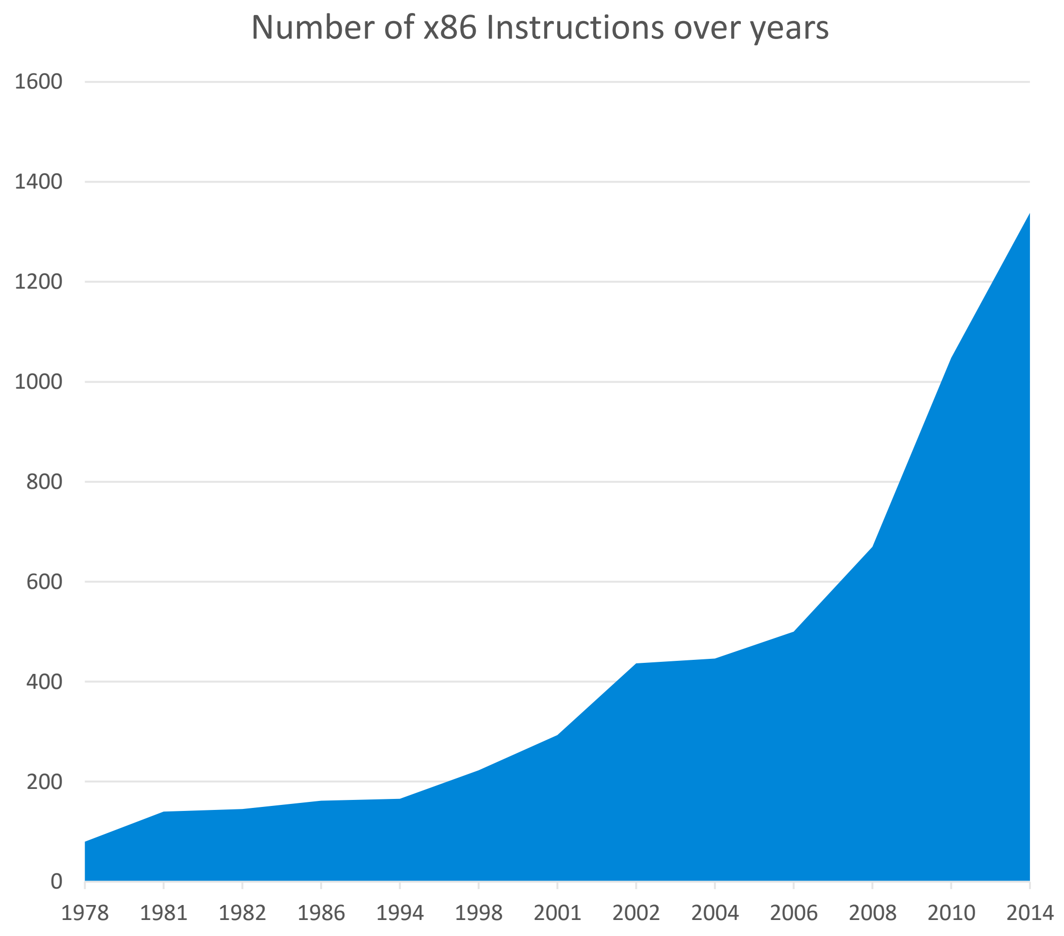
Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate



X86 Addressing modes and instruction encoding

Why a new instruction set?

- Proprietary ISA's have a slew of problems
  - Intel ISA has too much backward compatibility baggage
  - ARMv7 ISA (2005) also has imperfections
    - Some corrected on ARMv8 (64bit, 2015)
- Proprietary ISA's suffocate micro-architecture innovations
  - One needs an expensive ARM architecture license to do its own micro-architecture
  - Also, there is no path to add new/specialized instructions to ISA or low-latency interrupts
- New and open-source ISA's solve both of these problems
  - Clean Slate → It can rectify the previous ISA mistakes
  - Community designed → Leads to a better ISA



Hallmarks of a good ISA

- Modular not incremental
  - Intel is the prime example of incremental → 80 instructions in 1978 to 1338 in 2014
  - The RISC-V Base ISA is frozen and will never change → runs a full stack software
    - All RISC-V processors run the Base ISA
  - RISC-V supports optional ISA extensions (modules) or specialized applications
    - SIMD, Vector, Bit Manipulation, Privileged, Fast Interrupt, even Security!
- Suitable for both low-cost and high-performance processors
  - RISC-V supports both 32 GP registers and 16 GP registers architecture (embedded)
    - Reduced number of gates → Lower cost, active power and leakage
  - It also supports 16-bit opcodes (compressed)
    - Reduced Code size → reduce Flash capacity → reduced cost
- Agnostic to processor micro-architecture
  - Pipeline length, branch prediction, speculative out-of-order, …
  - No delayed branching or delay loading
    - MIPS, 5-stage pipeline
  - No conditional execution
    - ARM, OOO unfriendly (register renaming issues)
- Should consider the processor implementation complexity and performance into account
  - Load/Store approach
    - load/store instructions are separate from ALU instructions
  - Fixed instruction length
  - Fixed position for register addresses
    - Register access start as soon as the instruction is fetched

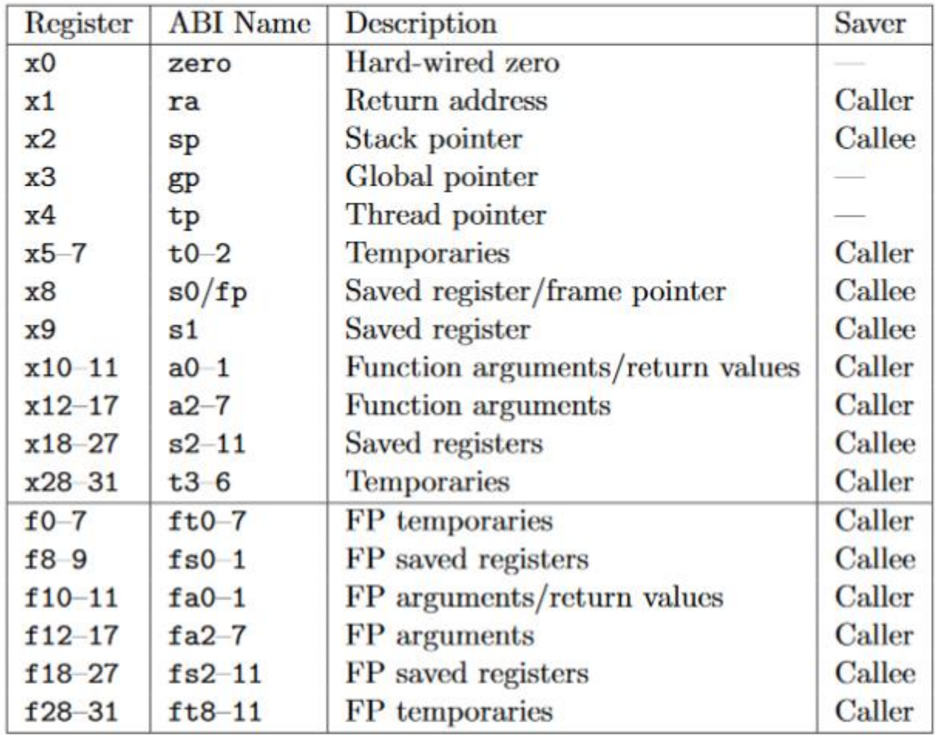
Lessons learned from the ARM ISA mistakes

- ARMv7 (2005) ISA problems:
- No hard-wired zero register
  - Reduces the number of instructions significantly
  - Examples: nop → addi x0, x0, 0
- PC as a general purpose register
  - Complicates hardware branch prediction
- 16 GP registers instead of 32
  - Causing issues for compiler/linker (too many push and pops onto stack)
- HW trouble-maker instructions
  - Load multiple, Conditional execution, e.g.
    - LDMIAEQ SP!, {R4-R7, PC}
- 32bit ISA and Thumb2 (16/32bit) separate ISA's
  - Need to branch to odd byte addresses to switch mode

- ARMv8 (2015) has made progress:
  - It does have a zero registers (not true for ARMv8-M)
  - It now carries 32 GP registers (16 for ARMv8-M)
  - PC is not a GP any more (not true for ARMv8-M)
  - No multiple load/store (not true for ARMv8-M)
  - No conditional execution (not true for ARMv8-M)
- But, it still has issues
  - No fixed location source/destination in instructions
  - Conditional move instructions
  - Complex addressing modes
  - 64bit ARM cores can not switch to Thumb-2 ISA
    - Thumb-2 ISA only works with 32bit addresses
  - Still not modular
    - 1000 instructions in 2015, 3 more expansions since then

RISC-V Base ISA

- Four base ISAs
  - RV32I (integer registers are 32-bit, 32-bit address space)
  - RV32E (reduced version of RV32I with only 16 integer GP registers)
  - RV64I (integer registers are 64-bit, flat 64-bit address space)
  - RV128I (integer registers are 128-bit, flat 128-bit address space)
  - (RV64E not defined, but considered)
- 32 general purpose (GP) registers
  - X0 always reads as 0
  - Only 16 for RV32E
- (32 floating point registers (optional, not part of Base))
- https://riscv.org/specifications/



Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5–7	t0–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

RISC-V ISA

- Types of instructions
  - R-Type: Register-register operations (*add rd, rs1, rs2*)

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

  - I-Type: Short immediate/loads (*addi rd, rs1, imm; lw rd, imm(rs1)*)
  - S-Type: Stores (*sw rs1, imm(rs2)*)
  - B/SB-Type: Conditional Branches (*beq rs1, rs2, imm*)

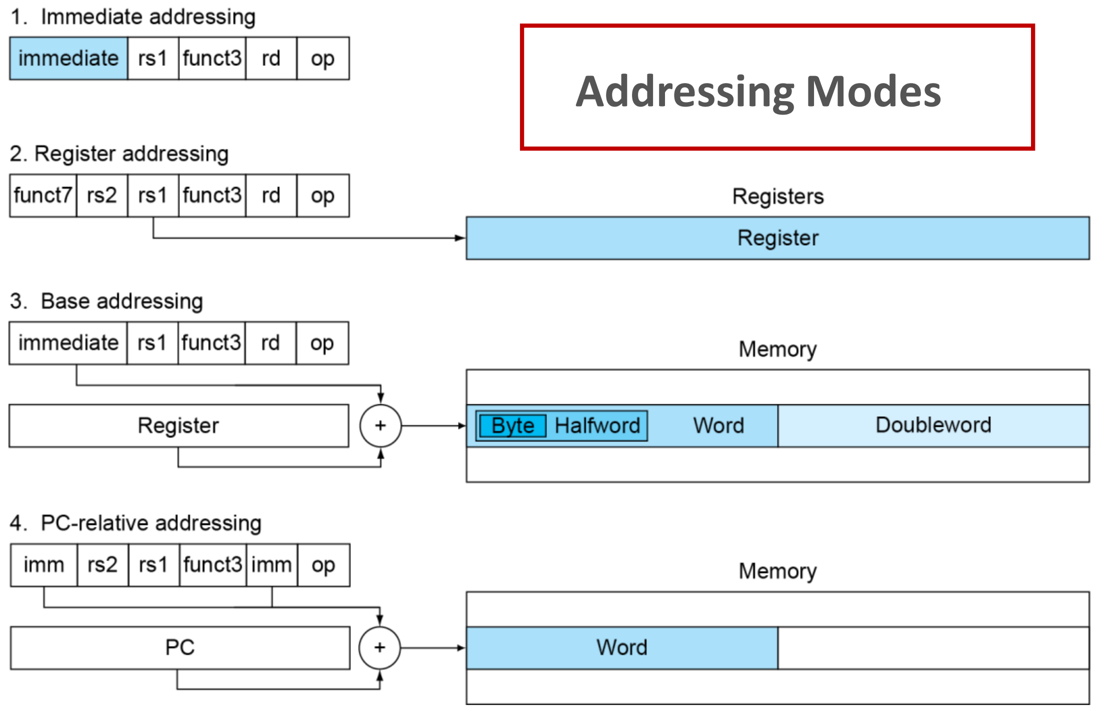
31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12|10:5]		src2	src1	BEQ/BNE	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BLT[U]	offset[11|4:1]		BRANCH
offset[12|10:5]		src2	src1	BGE[U]	offset[11|4:1]		BRANCH

  - U/UJ-Type: Long immediate (*lui rd, imm; auipc rd, imm; jal rd, imm*)

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

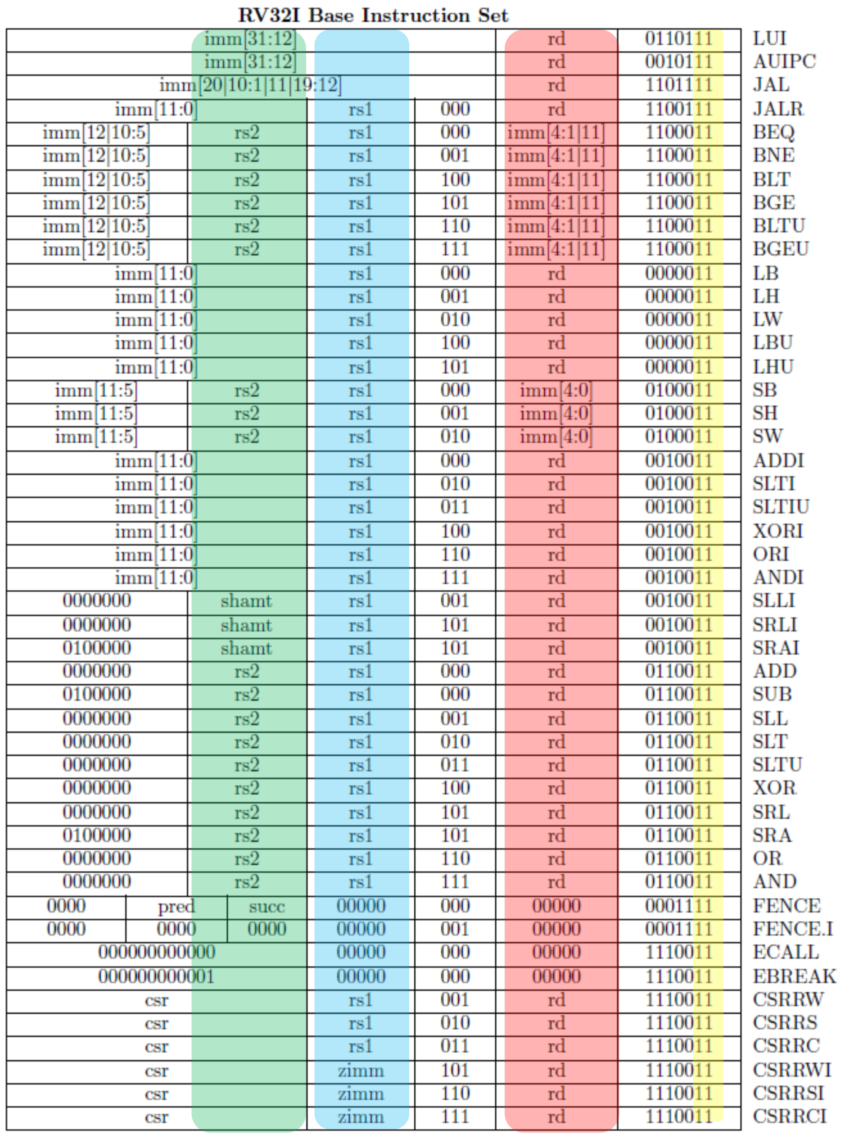
Name	Field						Comments
(Field Size)	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Addressing Modes



RISC-V ISA (cont.)

- Procedure Call and Return
  - PC-relative Short jump (+-4kiB or +-1MiB) call
    - *jal x1, 100*
      - *x1 = PC +4 (return address); go to PC + 100 (procedure address)*
  - PC-relative long-jump call
    - *auipc x5, 0x12345*
      - x5 = PC + 0x12345000
    - *jalr x1, 100(x5)*
      - x1 = PC+4 (*return address*); go to x5 + 100 (*procedure address*)
  - Absolute address long-jump calls
    - *lui x5, 0x12345*
      - x5 = 0x12345000
    - *jalr x1, 100(x5)*
      - *x1 = PC +4; go to x5 + 100*
  - PC-relative return
    - *jalr x0, 0(x1)*
      - x0 = PC+4 (no change); go to x1 + 0 (return address)



RV32I Base Instruction Set

imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20|10:1|11|19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI
0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND
0000 pred	succ	00000	000	00000	0001111	FENCE
0000 0000	0000	00000	001	00000	0001111	FENCE.I
000000000000		00000	000	00000	1110011	ECALL
000000000001		00000	000	00000	1110011	EBREAK
csr		rs1	001	rd	1110011	CSRRW
csr		rs1	010	rd	1110011	CSRRS
csr		rs1	011	rd	1110011	CSRRC
csr		zimm	101	rd	1110011	CSRRWI
csr		zimm	110	rd	1110011	CSRRSI
csr		zimm	111	rd	1110011	CSRRCI

RISC-V (optional) Standard Extensions

Extension	Description
M*	Integer Multiplication and Division
A	Atomic Instructions
Zicsr	Control and Status Register Instructions
F	Single-Precision Floating-Point
D	Double-Precision Floating-Point
Q	Quad-Precision Floating-Point
L	Decimal Floating-Point
C	Compressed Instructions
Zifencei	Instruction-Fetch Fence

Extension	Description
B	Bit Manipulation
J	Dynamically Translated Languages
T	Transactional Memory
P	Packed-SIMD Instructions (DSP)
V	Vector Operations
N	User-Level Interrupts
Zam	Misaligned Atomics
Ztso	Total Store Ordering
Zfinx	Floating-point using integer register file

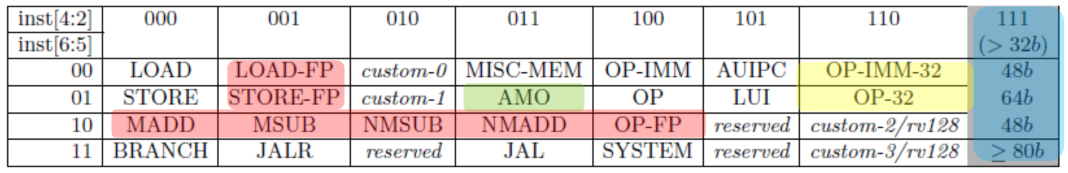
- RISC-V implementations mention supported extensions, e.g. RV32IMC (RV32I base + Multiply/divide + Compressed)
- RV32G is abbreviation for RV32IMAFDZicsr_Zifencei
- RV64G is abbreviation for RV64IMAFDZicsr_Zifencei
- Smallest core would support only RV32E (but RV32EC would typically be cheaper at system level)

* Extensions in **bold** are ratified (status in June 2019)

Standard and non-standard extensions

- RISC-V provides basis for specialized ISA extensions
  - Base ISA always needs to be supported
    - If so, it is a ‘RISC-V’
  - Plenty of room for (custom) extensions
    - E.g. RV32I ISA uses < 1/8 of 32-bit encoding space
    - Portions of encoding space are guaranteed to be left open by standard extensions
- Standard extension
  - Generally useful and designed to not conflict with any other standard extension
  - E.g. MAFDQLCBTPV
- Non-standard extension
  - May be highly specialized and may conflict with other standard or non-standard extensions
  - Might eventually be promoted to standard extension

- 30-bit encoding spaces
  - 3 available when not using compressed ISA
- 25-bit encoding spaces - major opcodes
  - Reserved: Aimed at future standard extensions
  - Custom-[0|1]: Will be avoided by future standard extensions
  - Custom-[2|3]/rv128: Reserved for future use by RV128



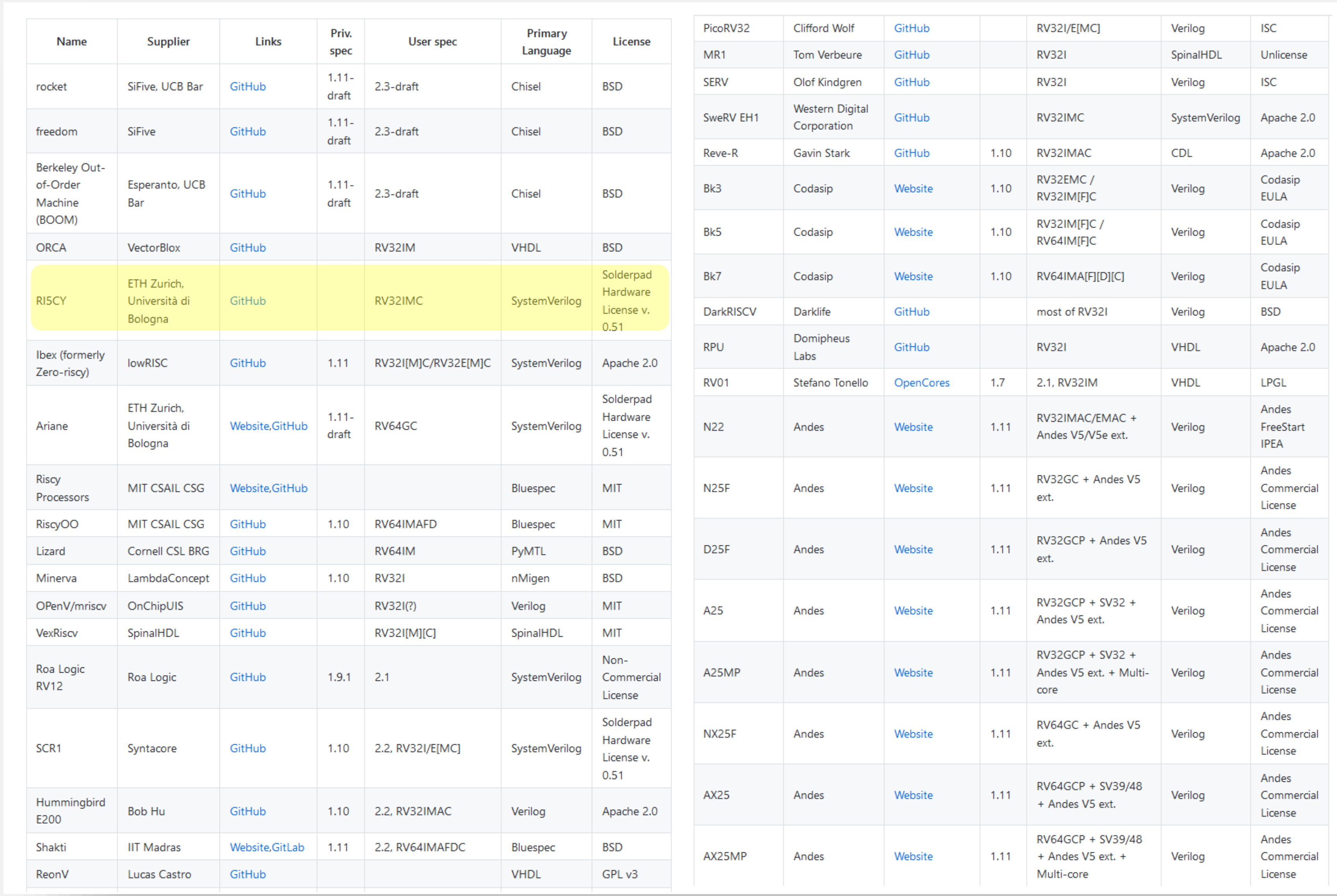
inst[4:2] / inst[6:5]	000	001	010	011	100	101	110	111 (> 32b)
00	LOAD	LOAD-FP	*custom-0*	MISC-MEM	OP-IMM	AUIPC	OP-IMM-32	48b
01	STORE	STORE-FP	*custom-1*	AMO	OP	LUI	OP-32	64b
10	MADD	MSUB	NMSUB	NMADD	OP-FP	*reserved*	*custom-2/rv128*	48b
11	BRANCH	JALR	*reserved*	JAL	SYSTEM	*reserved*	*custom-3/rv128*	≥ 80b

🟦	Available when not using >32-bit encoding
🟨	Reserved for RV64 (available in RV32)
🟥	Available when not using F (floating point extension)
🟩	Available when not using A (atomics extension)

- 22-bit encoding spaces - minor opcodes (funct3)
  - Several major opcodes have non-used minor opcodes

RISC-V Cores

RISC-V cores (https://github.com/riscv/riscv-cores-list)



Name	Supplier	Links	Priv. spec	User spec	Primary Language	License
rocket	SiFive, UCB Bar	GitHub	1.11-draft	2.3-draft	Chisel	BSD
freedom	SiFive	GitHub	1.11-draft	2.3-draft	Chisel	BSD
Berkeley Out-of-Order Machine (BOOM)	Esperanto, UCB Bar	GitHub	1.11-draft	2.3-draft	Chisel	BSD
ORCA	VectorBlox	GitHub		RV32IM	VHDL	BSD
RI5CY	ETH Zurich, Università di Bologna	GitHub		RV32IMC	SystemVerilog	Solderpad Hardware License v. 0.51
Ibex (formerly Zero-riscy)	lowRISC	GitHub	1.11	RV32I[M]C/RV32E[M]C	SystemVerilog	Apache 2.0
Ariane	ETH Zurich, Università di Bologna	Website,GitHub	1.11-draft	RV64GC	SystemVerilog	Solderpad Hardware License v. 0.51
Riscy Processors	MIT CSAIL CSG	Website,GitHub			Bluespec	MIT
RiscyOO	MIT CSAIL CSG	GitHub	1.10	RV64IMAFD	Bluespec	MIT
Lizard	Cornell CSL BRG	GitHub		RV64IM	PyMTL	BSD
Minerva	LambdaConcept	GitHub	1.10	RV32I	nMigen	BSD
OPenV/mriscv	OnChipUIS	GitHub		RV32I(?)	Verilog	MIT
VexRiscv	SpinalHDL	GitHub		RV32I[M][C]	SpinalHDL	MIT
Roa Logic RV12	Roa Logic	GitHub	1.9.1	2.1	SystemVerilog	Non-Commercial License
SCR1	Syntacore	GitHub	1.10	2.2, RV32I/E[MC]	SystemVerilog	Solderpad Hardware License v. 0.51
Hummingbird E200	Bob Hu	GitHub	1.10	2.2, RV32IMAC	Verilog	Apache 2.0
Shakti	IIT Madras	Website,GitLab	1.11	2.2, RV64IMAFDC	Bluespec	BSD
ReonV	Lucas Castro	GitHub			VHDL	GPL v3
PicoRV32	Clifford Wolf	GitHub		RV32I/E[MC]	Verilog	ISC
MR1	Tom Verbeure	GitHub		RV32I	SpinalHDL	Unlicense
SERV	Olof Kindgren	GitHub		RV32I	Verilog	ISC
SweRV EH1	Western Digital Corporation	GitHub		RV32IMC	SystemVerilog	Apache 2.0
Reve-R	Gavin Stark	GitHub	1.10	RV32IMAC	CDL	Apache 2.0
Bk3	Codasip	Website	1.10	RV32EMC / RV32IM[F]C	Verilog	Codasip EULA
Bk5	Codasip	Website	1.10	RV32IM[F]C / RV64IM[F]C	Verilog	Codasip EULA
Bk7	Codasip	Website	1.10	RV64IMA[F][D][C]	Verilog	Codasip EULA
DarkRISCV	Darklife	GitHub		most of RV32I	Verilog	BSD
RPU	Domipheus Labs	GitHub		RV32I	VHDL	Apache 2.0
RV01	Stefano Tonello	OpenCores	1.7	2.1, RV32IM	VHDL	LPGL
N22	Andes	Website	1.11	RV32IMAC/EMAC + Andes V5/V5e ext.	Verilog	Andes FreeStart IPEA
N25F	Andes	Website	1.11	RV32GC + Andes V5 ext.	Verilog	Andes Commercial License
D25F	Andes	Website	1.11	RV32GCP + Andes V5 ext.	Verilog	Andes Commercial License
A25	Andes	Website	1.11	RV32GCP + SV32 + Andes V5 ext.	Verilog	Andes Commercial License
A25MP	Andes	Website	1.11	RV32GCP + SV32 + Andes V5 ext. + Multi-core	Verilog	Andes Commercial License
NX25F	Andes	Website	1.11	RV64GC + Andes V5 ext.	Verilog	Andes Commercial License
AX25	Andes	Website	1.11	RV64GCP + SV39/48 + Andes V5 ext.	Verilog	Andes Commercial License
AX25MP	Andes	Website	1.11	RV64GCP + SV39/48 + Andes V5 ext. + Multi-core	Verilog	Andes Commercial License

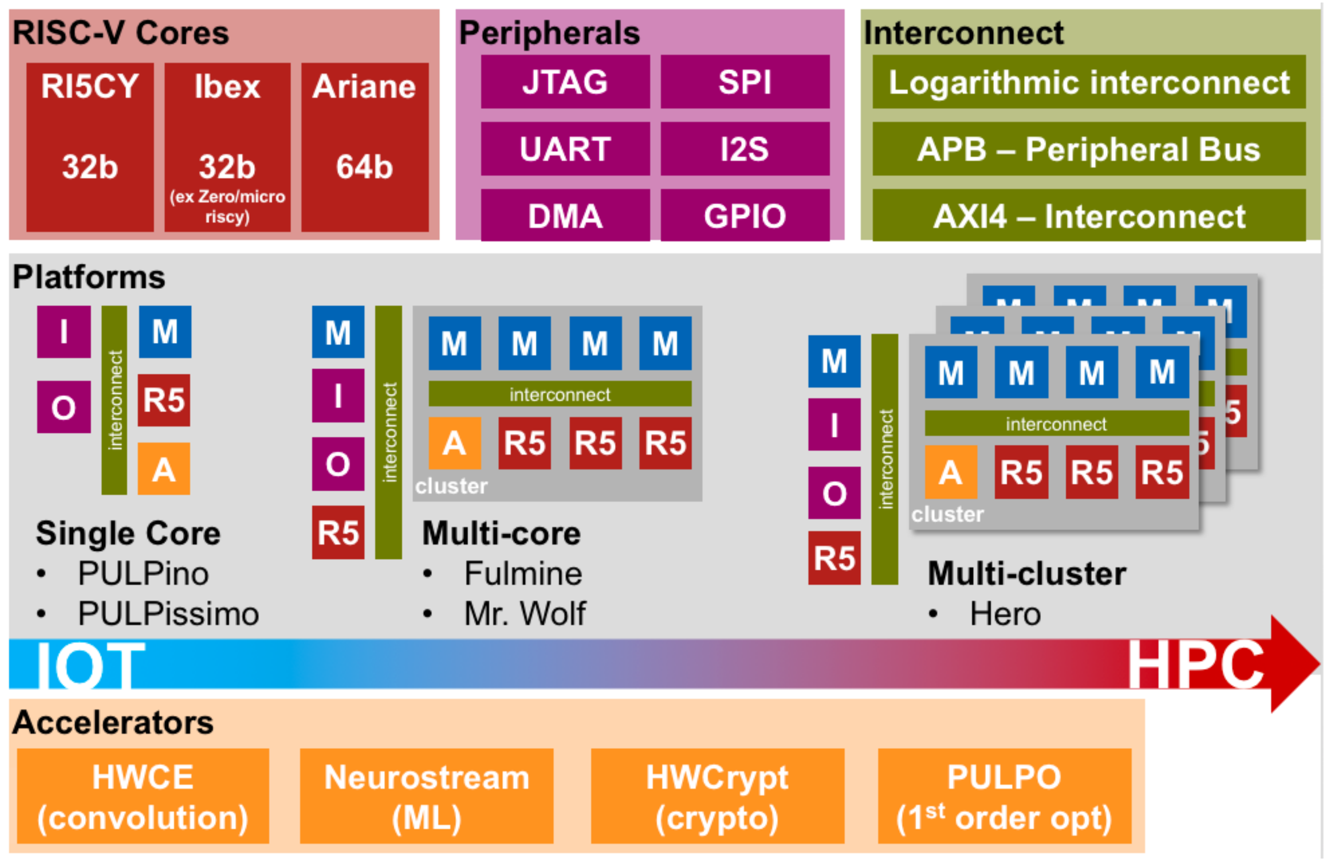
- Many RISC-V cores
  - Various ISA extensions
  - Various microarchitectures
  - Various languages
  - Various licenses
- Also SoCs
  - SiFive
    - FE310, U540
  - NXP
    - Vega
  - GreenWaves Technologies
    - GAP8
  - Microchip
    - PolarFire SoC

OpenHW (www.openhwgroup.org) and CORE-V

- CORE-V family of open-source cores
  - Hosted by OpenHW not-for-profit organization
    - Focus on high quality open-source HW development
    - In line with industry best practices
  - Based on RISC-V cores developed by ETH Zurich
- RI5CY
  - https://github.com/pulp-platform/riscv
  - 32-bit 4-stage in-order RISC-V CPU core
  - RV32IM[F]CXpulp
  - M-mode, U-mode
- Ariane
  - https://github.com/pulp-platform/ariane
  - 64-bit 6-stage single-issue in-order RISC-V CPU core
  - RV64IMAFDCN (+X)
  - M-mode, S-mode, U-mode (Linux-capable)

PULP 32-bit RISC-V cores

PULP and RISC-V*



- PULP (Parallel Ultra Low Power) is a project whose goal is to design an open-source energy efficient programmable platform for the Internet Of Things (IoT) applications
  - Integrated System laboratory, ETH, Zurich, Switzerland
  - Energy Efficient Embedded Systems Laboratory, University Of Bologna, Bologna, Italy
- Complete systems based on
  - Single-core micro-controllers (PULPissimo, PULPino)
  - Multi-core IoT Processors (OpenPULP)
  - Multi-cluster heterogeneous accelerators (Hero)
- Efficient implementations of RISC-V cores
  - 32 bit 4-stage core RI5CY
  - 64 bit 6-stage Ariane
  - 32-bit 2-stage Ibex (formerly Zero-riscy)

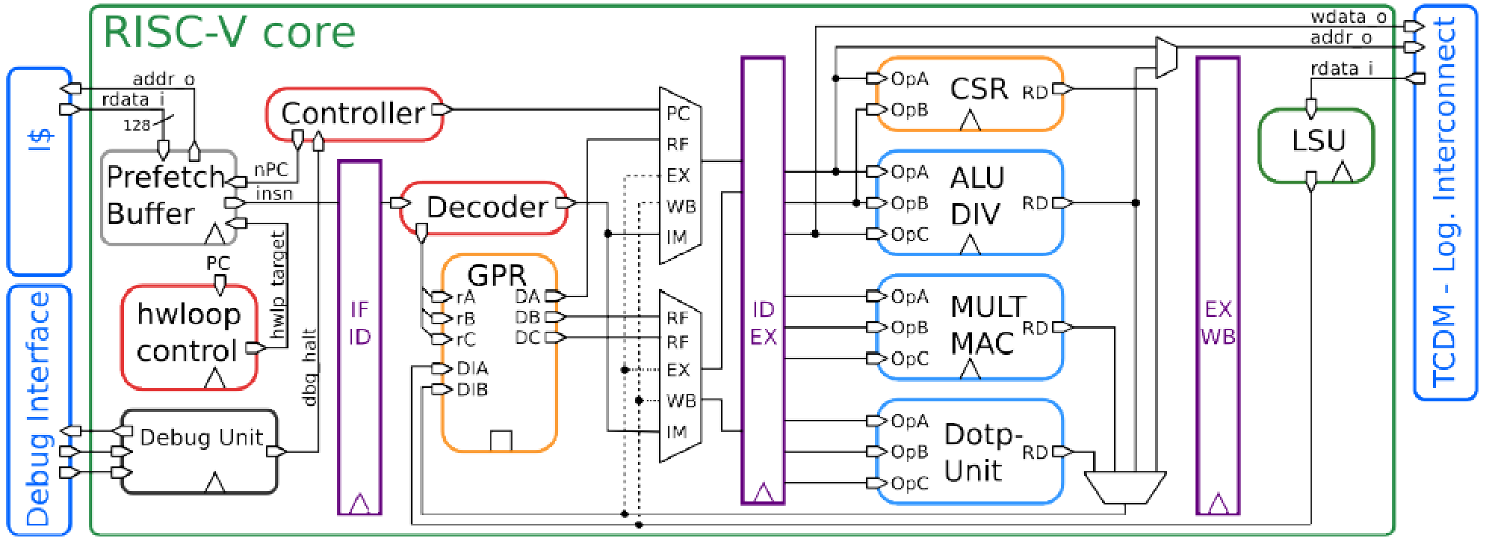
* Based on material from Integrated System laboratory, ETH, Zurich, Switzerland

RI5CY - Overview

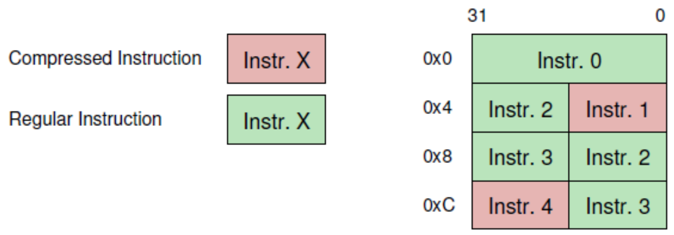
- RISCY
  - https://github.com/pulp-platform/riscv
  - 32-bit 4-stage in-order RISC-V CPU core
  - RV32IM[F]C[Zfinx]Xpulp
  - ~70 Kgates + 30 Kgates for FPU
  - 3.19 Coremark/MHz
  - M-mode, U-mode
  - Debug Spec 0.13 compliant
  - Implemented in SystemVerilog
- Xpulp custom extensions
  - Automatic increment load/store
  - Zero-overhead hardware-loop
  - Bit-manipulation
  - Enhanced signal processing
  - Packed-SIMD

- RI5CY in industry
  - Product announced by GreenWaves Technology:
    - GAP8 - IoT Application Processor
    - TSCM55 PULP-based chip
    - https://greenwaves-technologies.com/en/gap8-product/
  - Google is evaluating RISCY on Pixel Visual Core
    - VALTRIX Systems for verification effort
    - http://valtrix.in/programming/running-sting-on-pulpino
    - https://www.youtube.com/watch?v=m7aAUlHoV2E&feature=youtu.be
  - NXP RV32M1-VEGA
    - https://github.com/open-isa-org/open-isa.org/tree/master/Reference%20Manual%20and%20Data%20Sheet

RI5CY - Main configuration*



- Pipeline
  - Fetch
    - 4 word prefetch buffer
    - Align and decompress instructions



  - Decode
  - Execute / load / store
  - Writeback

- Main parameters
  - FPU: Floating Point Unit (single precision) (based on IEEE 754-2008)
    - Zfinx: Re-use X registers instead of separate 32 register FP register file
  - PULP_SECURE
    - PMP configuration: Physical Memory Protection
    - User mode

* Based on material from Integrated System laboratory, ETH, Zurich, Switzerland

RI5CY – Pipeline effects*

- No delay slot in RISC-V
  - Jumps done in the ID stage (loose one cycle)
    - Next instruction already fetching and probably ready in IF stage
- Combined branches (no set flag instruction)
  - Branch decision computed with branching instructions
  - Branch decision computed in EX stage
  - Taken branches loose two cycles
    - Branch only available late in pipeline
  - Not taken branches do not loose cycles

- Jumps

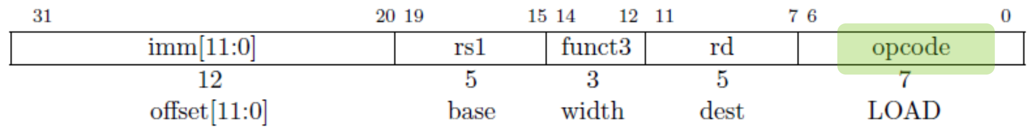
PC	IF	ID	EX
A to Imem	-	-	-
A+4 to Imem	X from Imem[A]	-	-
J to Imem	Y from Imem[A+4]	Jump to J	-
J+4 to Imem	K from Imem[J]	Bubble	add

- Branches

PC	IF	ID	EX
A to Imem	-	-	-
A+4 to Imem	X from Imem[A]	-	-
A+8 to Imem	Y from Imem[A+4]	X is Branch	-
B to Imem	Bubble	Bubble	Jump to B

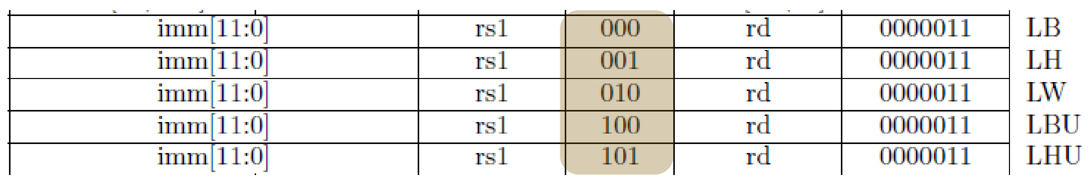
* Based on material from Integrated System laboratory, ETH, Zurich, Switzerland

RI5CY extensions - Post-increment load and reg-reg mode load*



31 ... 20	19 ... 15	14 ... 12	11 ... 7	6 ... 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

🟩 Add opcode: “0001011” LOAD WITH POST INCREMENT



imm[11:0]	rs1	000	rd	0000011	LB
imm[11:0]	rs1	001	rd	0000011	LH
imm[11:0]	rs1	010	rd	0000011	LW
imm[11:0]	rs1	100	rd	0000011	LBU
imm[11:0]	rs1	101	rd	0000011	LHU

🟫 Add funct3: “111” REG-REG

- Post-increment load (similar for store)
  - Greenfield Extension
  - New major opcode
    - 0001011 LOAD with POST INCREMENT
    - p.lw rd, imm12(rs1!)
    - Load value in rs1, store the value from memory in rd, and increment rs1 by imm12
    - I-Type with funct3 to selected byte, half-word, word

- Reg-reg mode load (similar for store)
  - Brownfield Extension
  - Reuse LOAD opcode, encode in free func3 encoding
  - p.lw rd, rs2(rs1)
  - Load value in rs2+rs1, store the value from memory in rd
  - funct3 to selected if standard load or R-type load
  - If R-type, use funct7 to select to load byte, half-word, word

* Based on material from Integrated System laboratory, ETH, Zurich, Switzerland

RI5CY extensions - Bit manipulation*

- RISC-V bit manipulation (B) not ratified yet
- PULP team developed their own extension
  - Contributing to Bit Manipulation task group
  - Will possibly align with B extension
- Bit Manipulation instructions
  - Extract N bits starting from M from a word and extend (or not) with sign
  - Insert N bits starting from M in a word
  - Clear N bits starting from M in a word
  - Set N bits starting from M in a word
  - Find first bit set
  - Find last bit set
  - Count numbers of 1
  - Rotate

- Original RISC-V

```
mv   x5, 0
mv   x7, 0
mv   x4, 32
Lstart:
  andi x6, x8, 1
  add x7, x7, x6
  addi  x4, x4, -1
  srli x8, x8, 1
bne  x4, x5, Lstart
```

Bit manipulation extension

```
p.cnt x7, x8
```

* Based on material from Integrated System laboratory, ETH, Zurich, Switzerland

RI5CY extensions – Hardware loops*

- Hardware Loops / Zero Overhead Loops
  - Remove branch overheads in for loops
  - Smaller loops benefit more (up to factor 2)
- Loop needs to be set up beforehand and is defined by 3 CSRs
  - Start address:
    - lp.starti L, imm12
    - START_REG[L] = PC + 2*imm12
  - End address
    - lp.endi L, imm12
    - END_REG[L] = PC + 2*imm12
  - Counter
    - lp.count{-,i}, L, {rs1,imm12}
    - COUNT_REG[L] = rs1/imm12
  - Short-cut
    - lp.setup{-,i}, L, {rs1,immc}, imm12
    - START_REG[L] = PC + 4, END_REG[L] = PC + 2*imm12, COUNT_REG[L] = {rs1,immc}
- Two sets registers implemented to support nested loops
  - L=0 or 1

- Original RISC-V

```
mv   x5, 0
mv   x4, 100
Lstart:
  addi  x4, x4, -1
  nop
bne  x4, x5, Lstart
```

Hardware Loop extension

```
lp.setupi 100, Lend
nop
Lend: nop
```

* Based on material from Integrated System laboratory, ETH, Zurich, Switzerland

RI5CY Extensions - Putting it All Together*

```
for (i = 0; i < 100; i++)
    d[i] = a[i] + b[i];
```

Baseline

```
mv    x5, 0
mv    x4, 100
Lstart:
  lb     x2, 0(x10)
  lb     x3, 0(x11)
  addi   x10,x10, 1
  addi   x11,x11, 1
  add    x2, x3, x2
  sb     x2, 0(x12)
  addi   x4, x4, -1
  addi   x12,x12, 1
bne      x4, x5, Lstart
```

11 cycles/output

Auto-incr load/store

```
mv    x5, 0
mv    x4, 100
Lstart:
  lb    x2, 1(x10!)
  lb    x3, 1(x11!)
  addi  x4, x4, -1
  add   x2, x3, x2
  sb    x2, 1(x12!)
bne     x4, x5, Lstart
```

8 cycles/output

HW Loop

```
lp.setupi 100, Lend
  lb    x2, 1(x10!)
  lb    x3, 1(x11!)
  add   x2, x3, x2
Lend:   sb x2, 1(x12!)
```

5 cycles/output

Packed-SIMD

```
lp.setupi 25, Lend
  lw   x2, 4(x10!)
  lw   x3, 4(x11!)
  pv.add.b x2, x3, x2
Lend: sw x2, 4(x12!)
```

1,25 cycles/output

* Based on material from Integrated System laboratory, ETH, Zurich, Switzerland

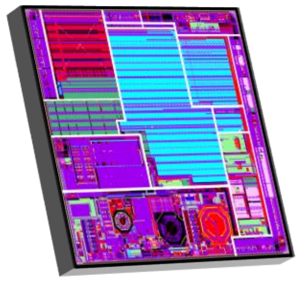
RISC-V summer internship – Compliance & Bus interfaces

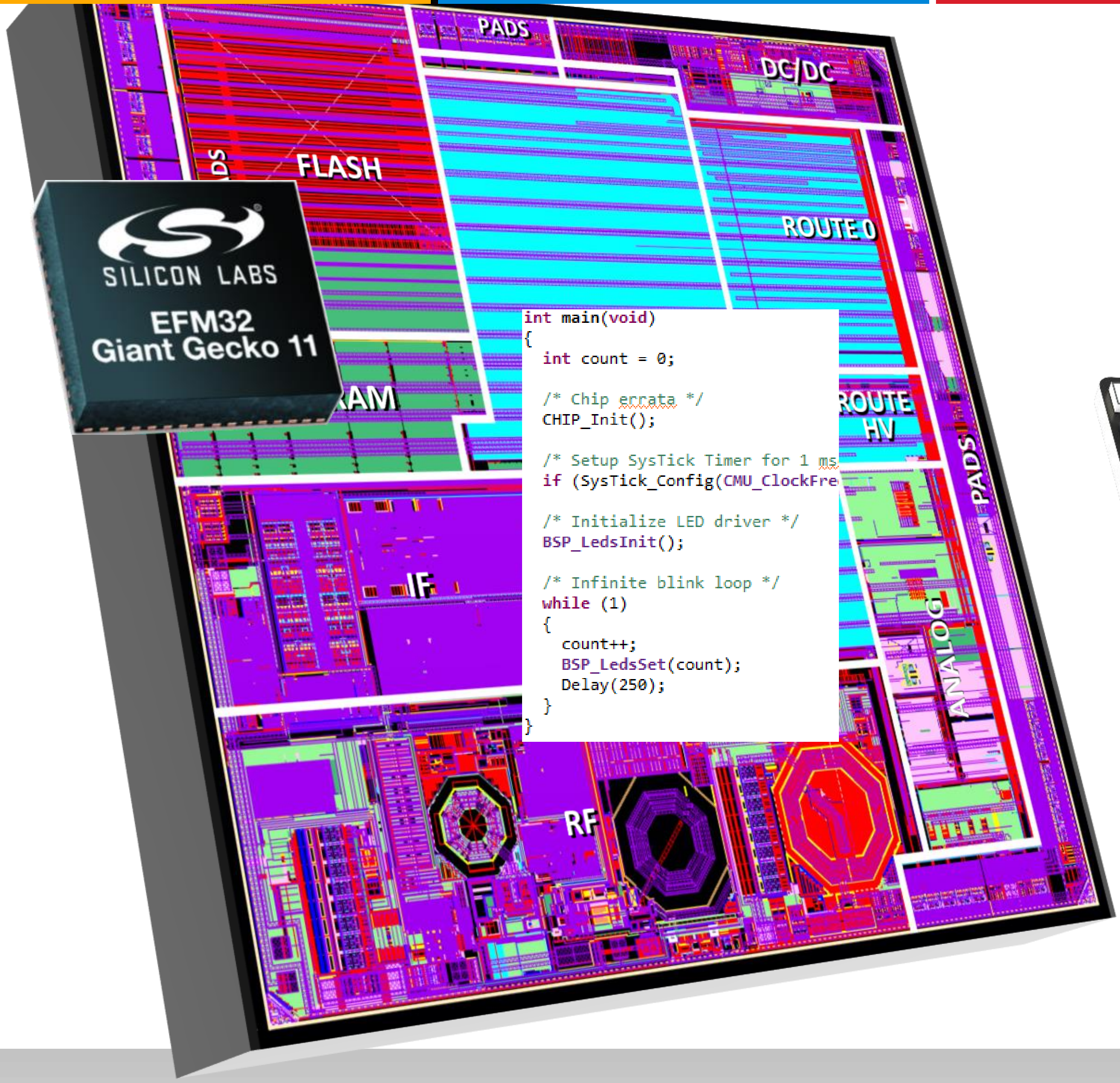
ANALOG AND DIGITAL DESIGN

VALIDATION

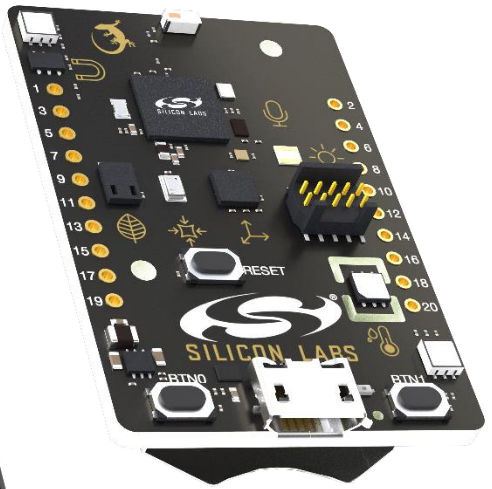
SOFTWARE

HARDWARE TOOLS





```
int main(void)
{
  int count = 0;

  /* Chip errata */
  CHIP_Init();

  /* Setup SysTick Timer for 1 ms
  if (SysTick_Config(CMU_ClockFre

  /* Initialize LED driver */
  BSP_LedsInit();

  /* Infinite blink loop */
  while (1)
  {
    count++;
    BSP_LedsSet(count);
    Delay(250);
  }
}
```



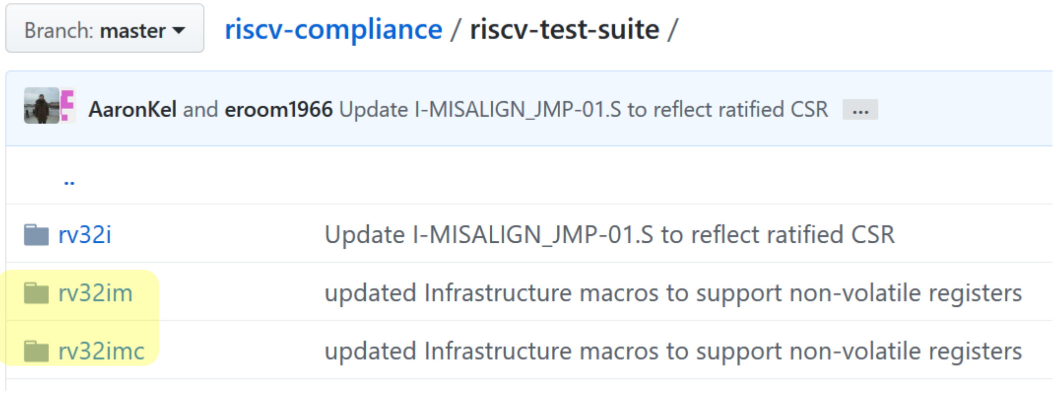
RISC-V summer internship

- Summer internship in Silabs Norway
  - Analog
  - Digital
  - Validation (test and characterization)
  - Software
  - Hardware Tools
- 6-8 weeks
- RI5CY related tasks
  - Addition of signature checks to RISC-V Compliance test cases
  - Addition of the C and M tests from RISC-V Compliance test suite
  - RTL design related to interfacing between RI5CY's proprietary bus interface and AMBA (AHB/AXI)
  - Formal verification of RI5CY's bus interface protocol

RISC-V Compliance Suite (https://github.com/riscv/riscv-compliance)

- Goal
  - Check whether a processor meets the open RISC-V standards or not
  - Assurance that the specification has been interpreted correctly
- No substitute for design verification
  - Check all important aspects of the specification,
  - but no details as for example
    - all possible values of instruction operands
    - all combinations of possible registers
    - bypasses, interlocks, etc.
- Detailed compliance test
  - RISC-V assembler code that is executed on the processor
    - Assembler code is (partially) self checking
    - Provides results in a defined memory area (the signature)
  - Signature to be checked against the reference signature from a RISC-V golden model

RISC-V Compliance Suite – ADDI Example

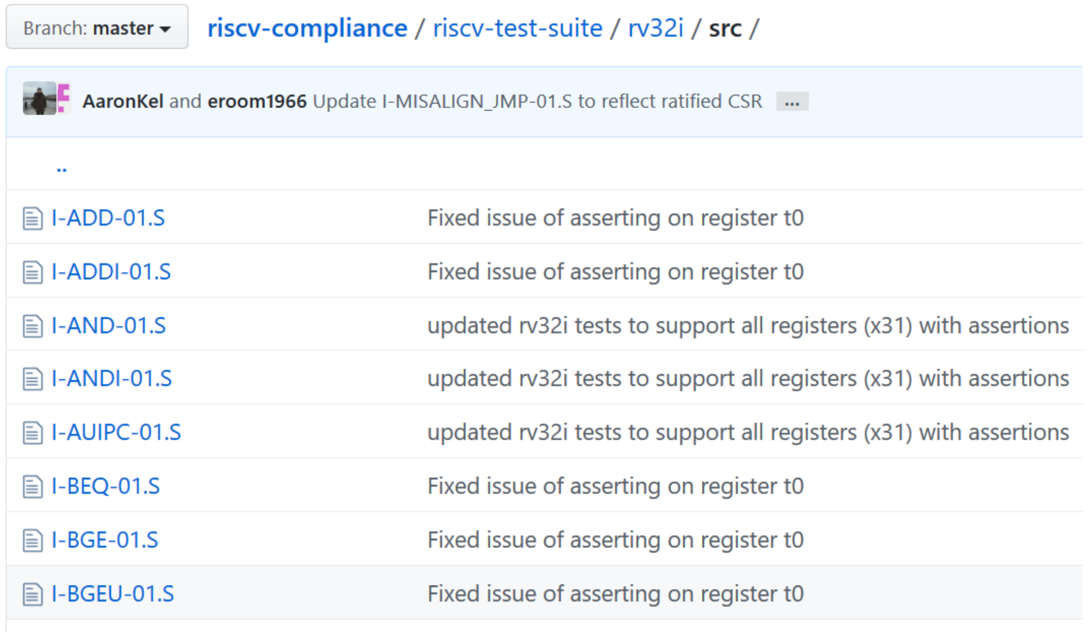


Branch: master ▾ riscv-compliance / riscv-test-suite /

AaronKel and eroom1966 Update I-MISALIGN_JMP-01.S to reflect ratified CSR …

..

rv32i	Update I-MISALIGN_JMP-01.S to reflect ratified CSR
rv32im	updated Infrastructure macros to support non-volatile registers
rv32imc	updated Infrastructure macros to support non-volatile registers



Branch: master ▾ riscv-compliance / riscv-test-suite / rv32i / src /

AaronKel and eroom1966 Update I-MISALIGN_JMP-01.S to reflect ratified CSR …

..

I-ADD-01.S	Fixed issue of asserting on register t0
I-ADDI-01.S	Fixed issue of asserting on register t0
I-AND-01.S	updated rv32i tests to support all registers (x31) with assertions
I-ANDI-01.S	updated rv32i tests to support all registers (x31) with assertions
I-AUIPC-01.S	updated rv32i tests to support all registers (x31) with assertions
I-BEQ-01.S	Fixed issue of asserting on register t0
I-BGE-01.S	Fixed issue of asserting on register t0
I-BGEU-01.S	Fixed issue of asserting on register t0

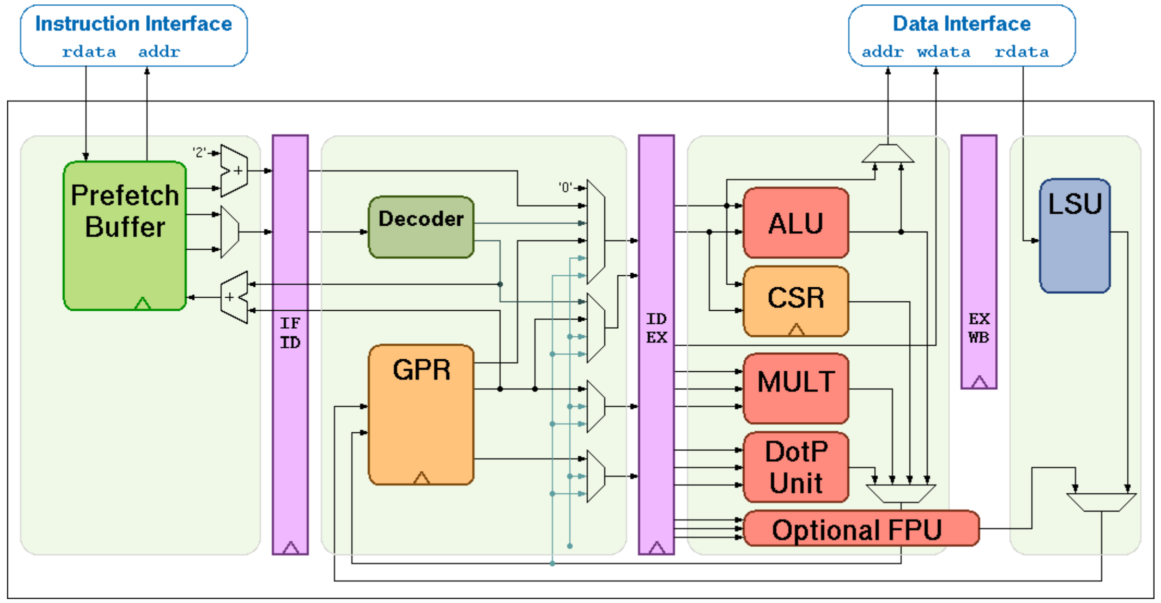
```
# Addresses for test data and results
la      x1, test_A3_data
la      x2, test_A3_res

# Load testdata
lw      x13, 0(x1)

# Test
addi    x14, x13, 1
addi    x15, x13, 0x7FF
addi    x16, x13, 0xFFFFFFFF
addi    x17, x13, 0
addi    x18, x13, 0xFFFFF800

# Store results
sw      x13, 0(x2)
sw      x14, 4(x2)
sw      x15, 8(x2)
sw      x16, 12(x2)
sw      x17, 16(x2)
sw      x18, 20(x2)

RVTEST_IO_ASSERT_GPR_EQ(x2, x13, 0xFFFFFFFF)
RVTEST_IO_ASSERT_GPR_EQ(x2, x14, 0x00000000)
RVTEST_IO_ASSERT_GPR_EQ(x2, x15, 0x000007FE)
RVTEST_IO_ASSERT_GPR_EQ(x2, x16, 0xFFFFFFFE)
RVTEST_IO_ASSERT_GPR_EQ(x2, x17, 0xFFFFFFFF)
RVTEST_IO_ASSERT_GPR_EQ(x2, x18, 0xFFFFF7FF)
```

Internship tasks (1/2)

- Addition of signature checks
- Addition of the C and M tests
  - RV32I
    - 55 focused tests
    - no coverage of FENCE, SCALL, SBREAK, pseudo and CSR instructions
  - RV32IM
    - 7 focused tests
  - RV32IMC
    - 24 focused tests

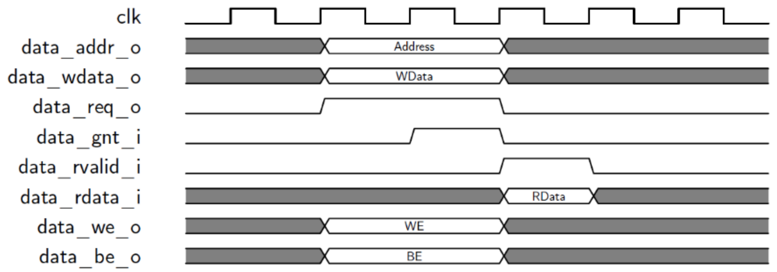
- Capabilities
  - C coding
  - RISC-V assembly coding
  - Makefile / compiler / assembler setup
  - Digital simulation
  - System Verilog testbench design

RI5CY bus interfaces

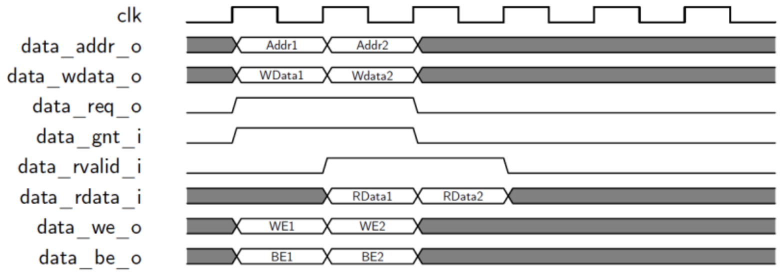


- Separate instruction / data interfaces
  - 32-bit wide
- Proprietary protocol (not AMBA compliant)
  - Provide address (write data, write enable, byte enable)
  - Set request high and wait for grant
  - Wait for end of data phase (rvalid with optional read data)

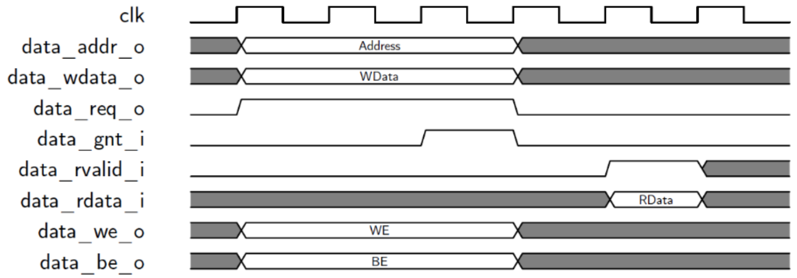
- Data interface examples



Basic transfer



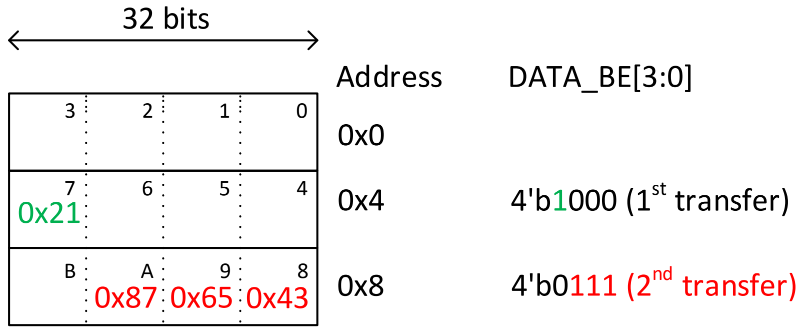
Back-to-back transfer



Slow response transfer

How about conversion to AMBA protocols? (1/2)

- Non-aligned transfers
  - Allowed in RISC-V
    - Load/store of word to non-word-aligned address
    - Load/store of half-word to non-halfword-aligned address
    - Trap allowed
  - RI5CY handles non-aligned transfers in hardware
    - `li x1, 0x87654321`
    - `sw x1, 7(x0)`



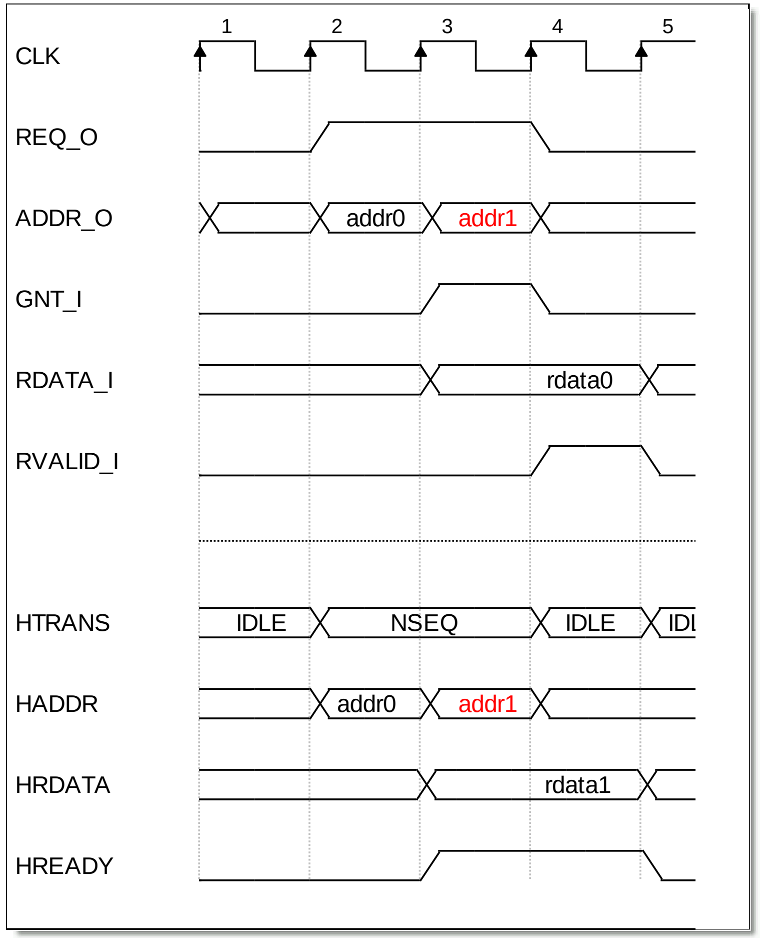
    - Some byte lane combinations do not have an AMBA AHB equivalent
      - 4’b1110, 4’b0111, 4’b0110

- Grant is not allowed before corresponding request
  - Request from master (RI5CY) to slave (e.g. memory, bus system)
  - Grant from slave to master
  - Cannot tie grant (or valid) high for simple slaves
  - Slave must look at request to generate a grant
    - Long combinational path
      - from master (via request) to slave,
      - and back from slave to master (via grant)
    - or extra cycle(s) latency when breaking such paths with flip-flops
- Combinational paths between RVALID_I and REQ_O

How about conversion to AMBA protocols? (2/2)

- Stability of addr_o, wdata_o, we_o, be_o
  - Should not change until request is granted
- RI5CY does not keep address phase info stable for non-granted request
  - On instruction interface
    - By design (https://github.com/pulp-platform/riscv/issues/128)
  - On data interface
    - Bug (https://github.com/pulp-platform/riscv/issues/124) (actually request is withdrawn)
  - Complicates conversion to AHB/AXI
- AMBA-AHB requirement
  - “When the HTRANS type changes to NSEQ the master must keep its address constant, until HREADY is HIGH”
- Such bus protocol properties can easily be checked formally
  - Write (System Verilog) assertions for RI5CY bus protocol
  - Run formal verification on RI5CY + bus protocol assertions

Illegal AHB transfer (for ‘ideal AHB conversion’)



Internship tasks (2/2)

- RTL design related to interfacing between proprietary bus interface and AMBA (AHB/AXI)
- Formal verification of RI5CY's bus interface protocol

- Capabilities
  - RTL design
  - Processor architecture
    - Load/store unit
    - Bus interfaces
  - Digital simulation
  - System Verilog assertions
  - Formal verification
  - Debug

- Privilege promotion/demotion for load/store
  - https://github.com/pulp-platform/riscv/issues/124
- Address channel signal stability during non-granted REQ_O
  - https://github.com/pulp-platform/riscv/issues/128
- Dependency between REQ_O and GNT_I
  - https://github.com/pulp-platform/riscv/issues/127
- Combinational paths between RVALID_I and REQ_O
  - https://github.com/pulp-platform/riscv/issues/126

- Application for 2020 Summer internships
  - https://tinyurl.com/y6nzfbsg

Thank you!

SILABS.COM