Carnegie Mellon University
Software Engineering Institute

Co-design for Edge AI

JULY 19, 2023

Dr. John Wohlbier
Advanced Computing Lab
CMU-SEI AI Division

© 2023 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for

Copyright 2023 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

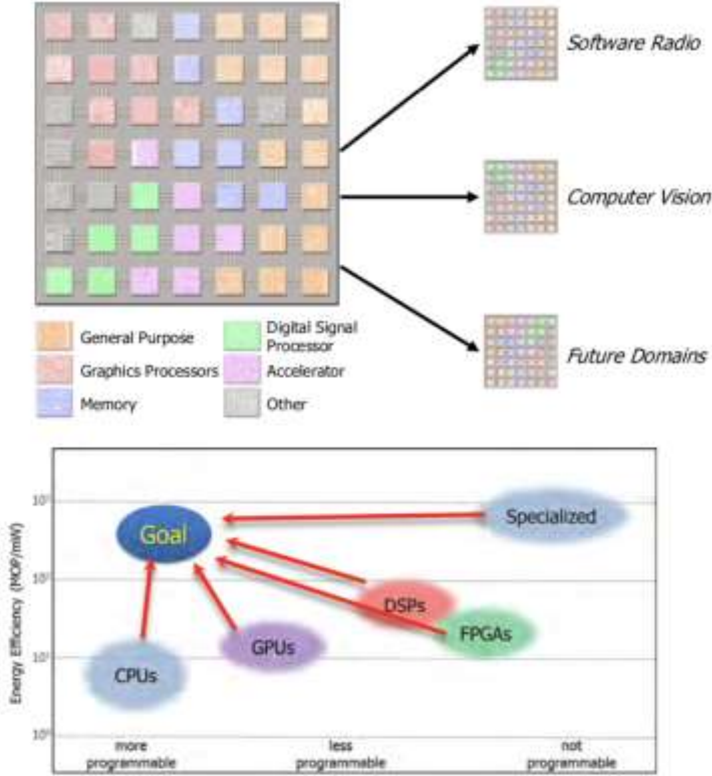
Carnegie Mellon® is registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM23-0730

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

Hardware Software Co-design of Domain Specific Systems on Chip

Basic problem: AI Engineering for rapidly developing heterogeneous systems on a single programmable device



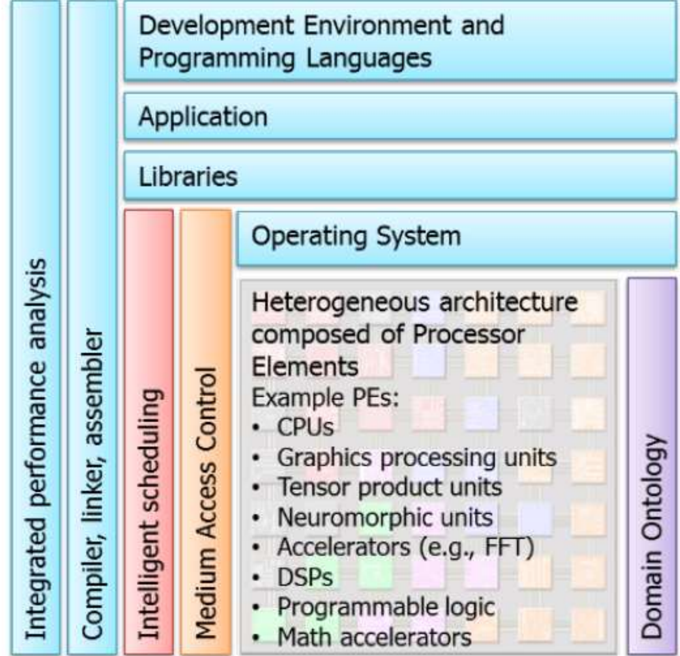


Figure 4. Vertical integration to enable decoupled application development

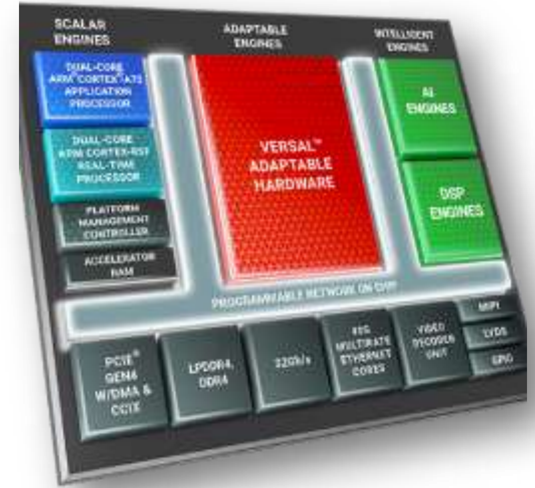
Images: DARPA Domain Specific System on Chip

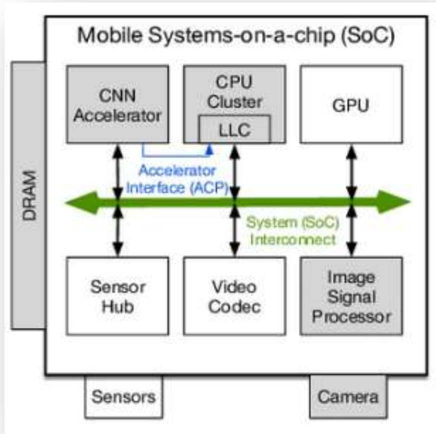
[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

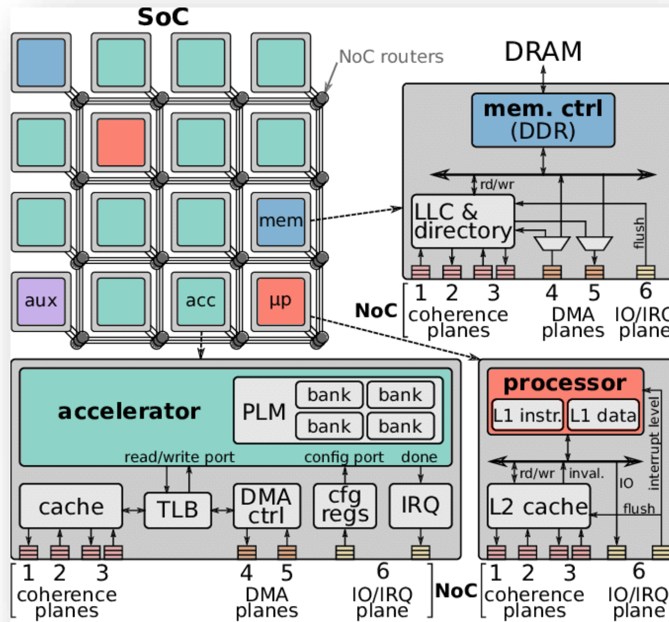
© 2023 Carnegie Mellon University

SoC co-design flow to produce Hardware Description Language (HDL)

- Implementation in Field Programmable Gate Arrays
- Application Specific Integrated Circuit Design

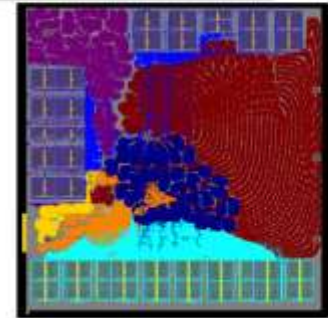






Gemmini

- Supports TFLite quantization spec
- Can readout intermediate matrices
- Can access memories using scan
- Has (a very primitive) ECC
- Speedup:
  - 1000x (ideal application)
  - 50-100x (TFLite application)
- VLSI Metrics @TTTT, 25C, 0.85V
  - Clock frequency: 600MHz
  - Leakage: 0.45mW
  - Dynamic power: 1.4W

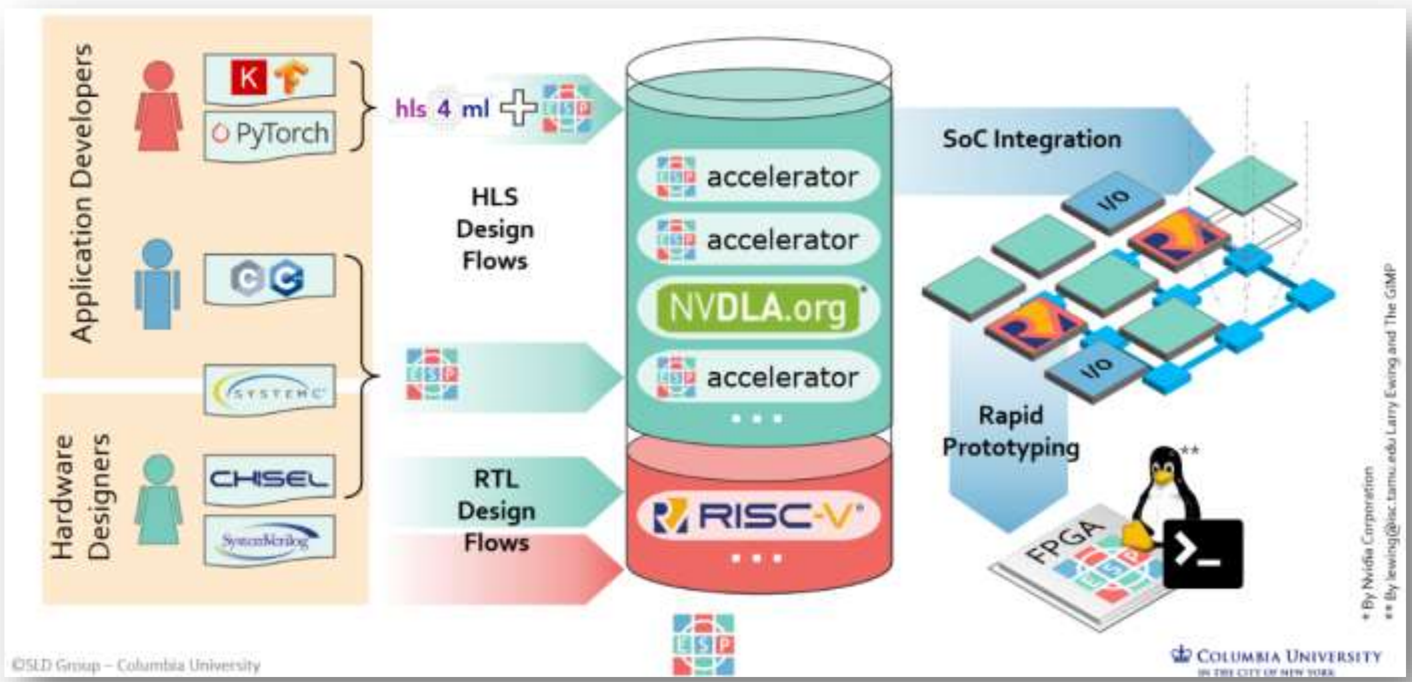


[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

© 2023 Carnegie Mellon University

Embedded Scalable Platforms – Columbia University

- Open source SoC design flow
- Accelerator synthesis
  - Python frameworks
  - System-C
  - Chisel
  - HDL's
- CPU cores
  - 64 bit RISC-V
  - 32 bit RISC-V
  - 32 bit Sparc
- Software
  - Bare metal
  - Linux on CPU's

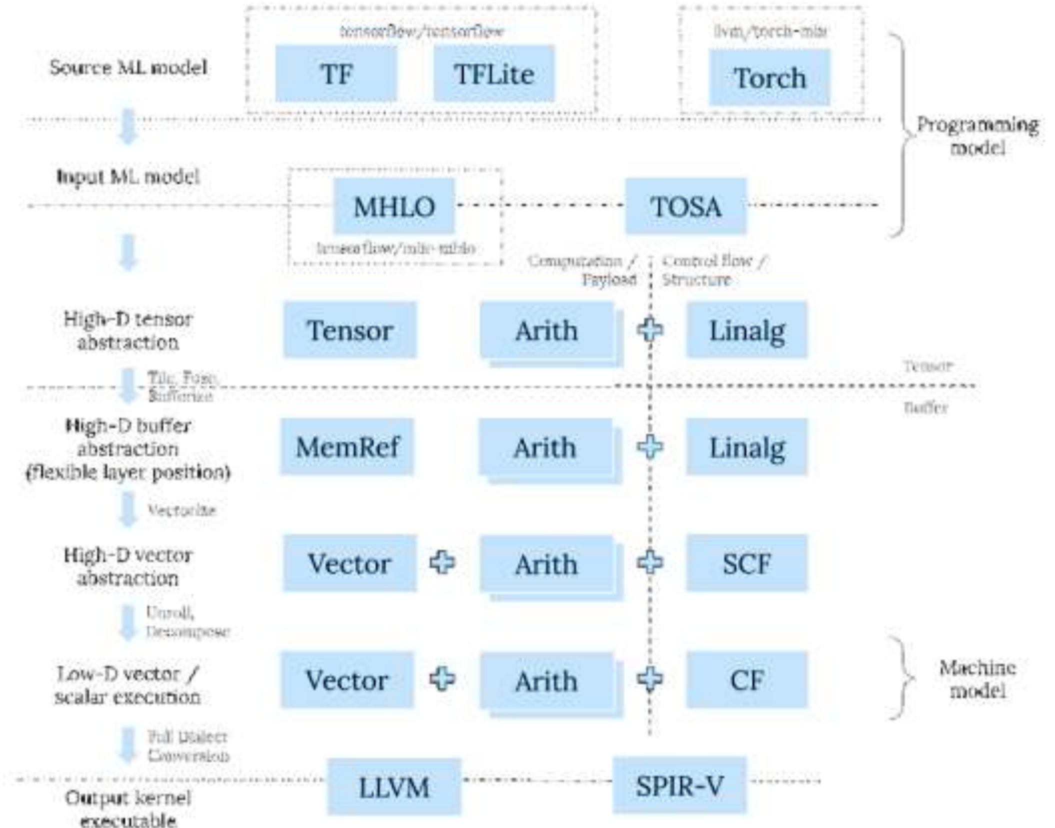


[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

© 2023 Carnegie Mellon University

Accelerator flow with MLIR based soda-opt

- MLIR – multi-level intermediate representation
  - Compiler infrastructure enabling multiple layers of abstraction reduction
  - Tensors => buffers => vectors
- Multiple synthesis backends
  - Bambu
  - CIRCT
- PyTorch => HDL



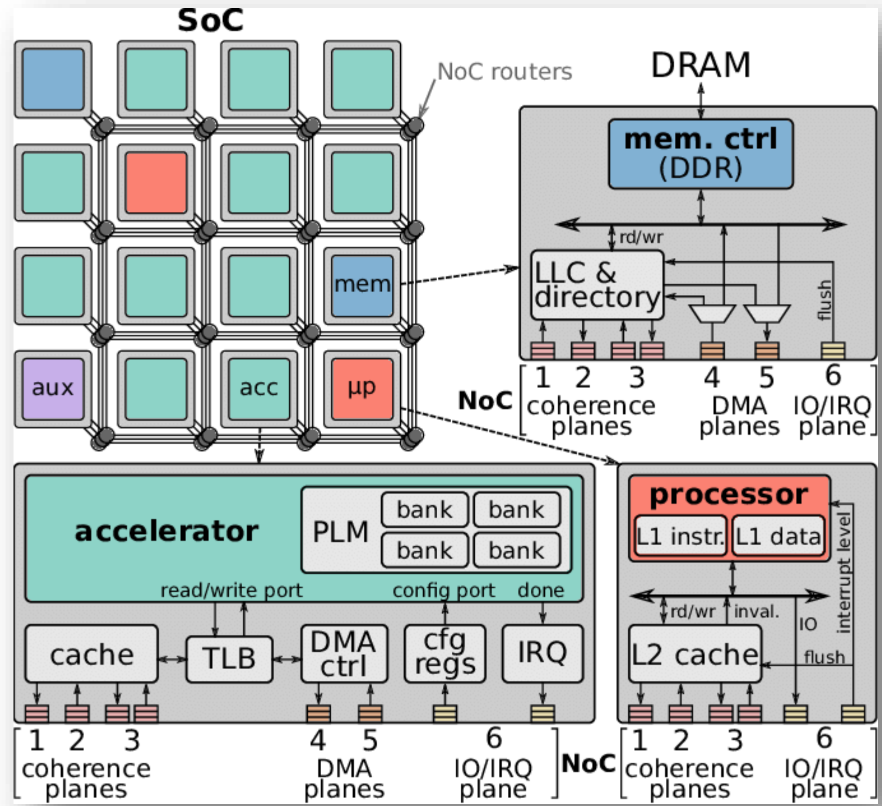
© 2023 Carnegie Mellon University

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

System on Chip Security Considerations

ADVANCED COMPUTING LAB

- SoCs
  - IoT
  - Cyber-physical systems
  - Embedded systems
- Hardware
  - 3rd party synthesis tools, open source and proprietary
  - 3rd party IPs
- Software
  - Bare metal
  - Linux



[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

© 2023 Carnegie Mellon University

32-bit RISC-V processor from scratch

ADVANCED COMPUTING LAB

- A complete 32-bit RISC-V (RV32I) processor written in Verilog
  - Tested with simulation and on an iCEstick FPGA
  - Basic integer operations – no floating point or multiplier
  - Based on the FROM BLINKER TO RISCV tutorial from Bruno Levy
- Try it out for yourself (links are to code.sei.cmu.edu)
  - Clone and build the docker environment
  - Clone the RISC-V repository
  - Follow the steps on branches step1/0 – step24/0
- Demo
  - Compute Mandelbrot set from assembly program in memory
  - Ray tracing cross compiled with gcc from C source

README.md

- 1/1 : Slow down counter to see LEDs change
- 1/2 : Knight rider blinky
- 2/0 : Add clock divider to slow blinky
- 2/1 : Knight rider, reset toggles direction
- 3/0 : Blinky reads patterns from ROM
- 3/1 : Two blinking modes selected by RESET
- 4/0 : Add instruction decoder
- 4/1 : Initialize memory with different RISC-V instruction
- 5/0 : Add register file and state machine
- 6/0 : Add ALU
- 7/0 : Use Verilog assembler for instructions
- 8/0 : Add jumps JAL, JALR
- 8/1 : Add instructions before the loop
- 9/0 : Handle branch instructions
- 9/1 : Knight rider blinky with two inner loops in opposite directions
- 10/0 : Add `LUI` and `AUIPC` instruction handling
- 11/0 : Add memory and processor as separate modules
- 12/0 : CPU optimizations. ~1340 LUTS to ~840 LUTs.
- 13/0 : Add subroutines without load/store instructions, following RISC-V ABI.
- 13/1 : Knight rider blinky with one subroutine l/r and one r/l
- 14/0 : subroutines using RISC-V ABI
- 14/1 : add multiplication subroutine and test in software and hardware
- 15/0 : implement load instructions
- 15/1 : try other load instructions, make a tinsel
- 16/0 : add store instructions. Loads take some time before blinky.
- 17/0 : memory mapped device. Use ./terminal.sh to see UART. ctrl-a-q to quit
- 18/0 : computing the mandelbrot set
- 19/0 : simulate with verilator. compare: make simulate to make run-verilate
- 20/0 : Using the gnu toolchain to compile programs. Read firmware.
- 21/0 : Use the gnu toolchain to compile C programs.
- 22/0 : More than 6kB of memory using SPI FLASH. NB: simulate and verilate ta program : runs the program ./terminal : view output THIS IS NOT WORKING, T
- 23/0 : running program from SPI flash, first steps
- 24/0 : running program from SPI flash, better linker script

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

© 2023 Carnegie Mellon University