Generating and Simulating Custom SoCs with Vector Units in CHIPYARD

Jerry Zhao
jzh@berkeley.edu
UC Berkeley

Tutorial Outline

Basic Usage

1. Generate RTL for a simple SoC
2. Explore Chipyard compilation flow and outputs
3. Compile RISC-V baremetal software
4. Simulate SoC running bare-metal software

Chipyard Directory Structure

```
chipyard-morning/
  generators/     ←  Our library of Chisel generators
    chipyard/
    saturn/
  sims/           ←  Utilities for simulating SoCs
    verilator/
    firesim/
  fpga/           ←  Utilities for FPGA prototyping
  software/       ←  Utilities for building RISC-V software
  vlsi/           ←  HAMMER VLSI Flow
  toolchains/     ←  RISC-V Toolchain
```

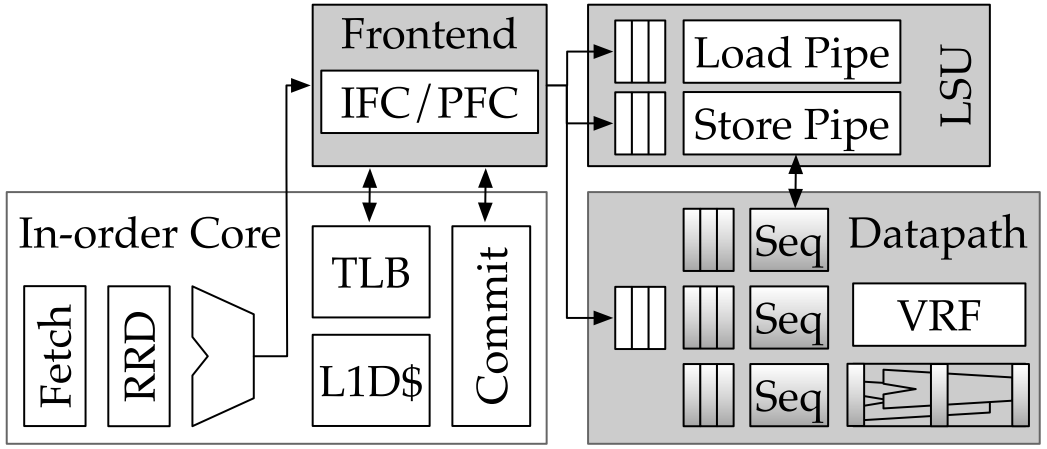
```
> ls $MCYDIR
> tmux new –s buildrtl

(if detached, run)
> tmux a -t buildrtl
> cd $MCYDIR/generators/chipyard
> cd src/main/scala/config
> less SaturnConfigs.scala
```

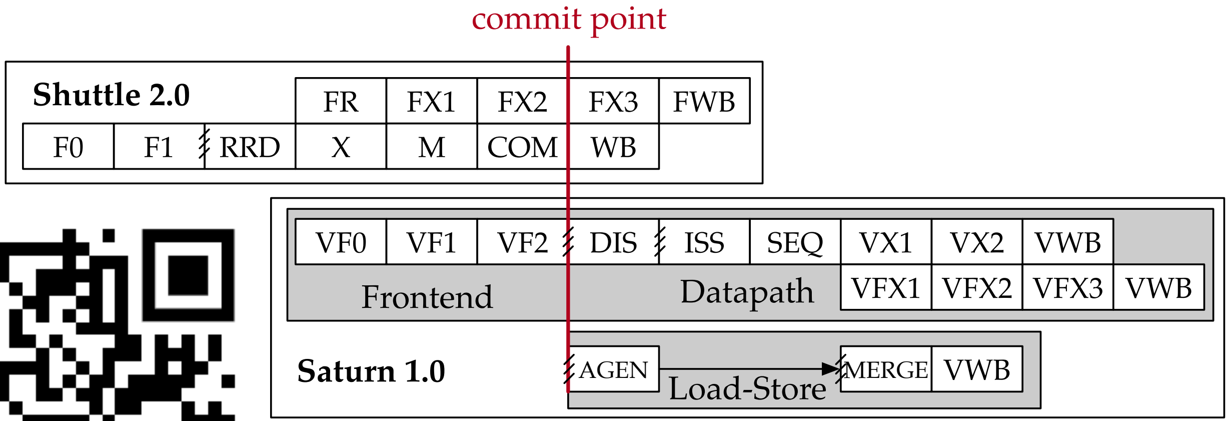
Designing a SoC Config

- Look at the many variants of Saturn configs
- A Config is composed of Config “fragments”
  - “fragments” set/override/adjust/clear keys in the Config object
  - Generators query this “Config” object at runtime to figure out what to do
- See for yourself:
  - These configs build on top of a base config (AbstractConfig)
  - Each Config adds a core (Rocket or Shuttle)
  - Each Config adds a variants of Saturn
  - Some Configs modify the system interconnect width from the default 64b

Background: Saturn Vector Unit

- Compact vector unit with short vector lengths and wide SIMD datapath
- Targets DSP-core like deployments
- Vector chaining with out-of-order multi-issue
- Support for all RVV 1.0 instructions
- Executes unmodified Linux vector binaries
- Highly parameterized:
  - Core (Rocket or Shuttle)
  - VLEN - vector length
  - DLEN - datapath width
- github.com/ucb-bar/saturn-vectors





Tutorial Configs

	GENV256D128ShuttleConfig	GENV512D256ShuttleConfig	REFV256D128RocketConfig
VLEN (Vector length)	256b	512b	256b
DLEN (Datapath width)	128b	256b	128b
Host Core	Dual-issue Shuttle	Dual-issue Shuttle	Single-issue Rocket

- Tutorial will use the above three configs
- We've pre-cached these builds for you, so you don't have to wait ~5 minutes for Verilator
- Try the other vector units later

```
> cd $MCYDIR/sims/verilator
> make CONFIG=GENV256D128ShuttleConfig

...

> ls generated-src

> cd generated-src/ … GEN256D128ShuttleConfig/
> ls
```

Looking at what is generated

- **`<CONFIG>.dts`**
  - Device tree string - describes to software what's on the SoC
- **`<CONFIG>.fir`**
  - FIRRTL intermediate representation
- **`<CONFIG>.top.f`**
  - Filelist of all source Verilog files for the design
  - Tool-consumable
- **`gen-collateral/`**
  - Directory containing output verilog files, harness files, etc.

```
> cd gen-collateral/
> ls *.cc
> ls *.sv
> less ChipTop.sv
> less TestHarness.sv
```

ChipTop and TestHarness

ChipTop.sv

- Contains definition of the ChipTop
- ChipTop defines a single die and its top-level IO
- This would get passed to VLSI tools as the target module

TestHarness.sv

- Contains definition of the TestHarness
- TestHarness instantiates a ChipTop
- Also instantiates simulation models of I/O devices
  - Ex: model of off-chip DRAM

Compiling Software

Three common approaches

- **bare-metal**
  - No virtual memory
  - No system calls
  - Fast … minimal overhead before your `main` starts
- **proxy-kernel**
  - Compile RISC-V application as you would for Linux
  - PK “proxies” syscalls to the x86 host
  - Virtual memory
  - Slow, and only supports a subset of OS capabilities
- **linux**
  - User binary run as a normal program under OS
  - Run on FPGA prototypes or FireSim

```
> cd $MCYDIR/tests
> spike hello.riscv
```

Running tests on RTL simulators

- Chipyard uses make targets to invoke RTL simulators
  - **`run-binary`** - Runs a binary
  - **`run-binary-debug`** - Runs a binary with waveforms
- Make variables specify simulation options
  - **`CONFIG`** - What config to build
  - **`BINARY`** - Run this binary
  - **`LOADMEM`** - Sets fast-memory initialization
  - **`timeout_cycles`** - Terminate simulation after this many cycles
- Chipyard SoCs match Spike's bringup behavior
  - If it runs in Spike it (probably) runs in an equivalent architecture Chipyard SoC

```
> cd $MCYDIR/sims/verilator
> make \
    CONFIG=GENV256D128ShuttleConfig \
    BINARY=$MCYDIR/tests/hello.riscv \
    LOADMEM=1 \
    run-binary
```

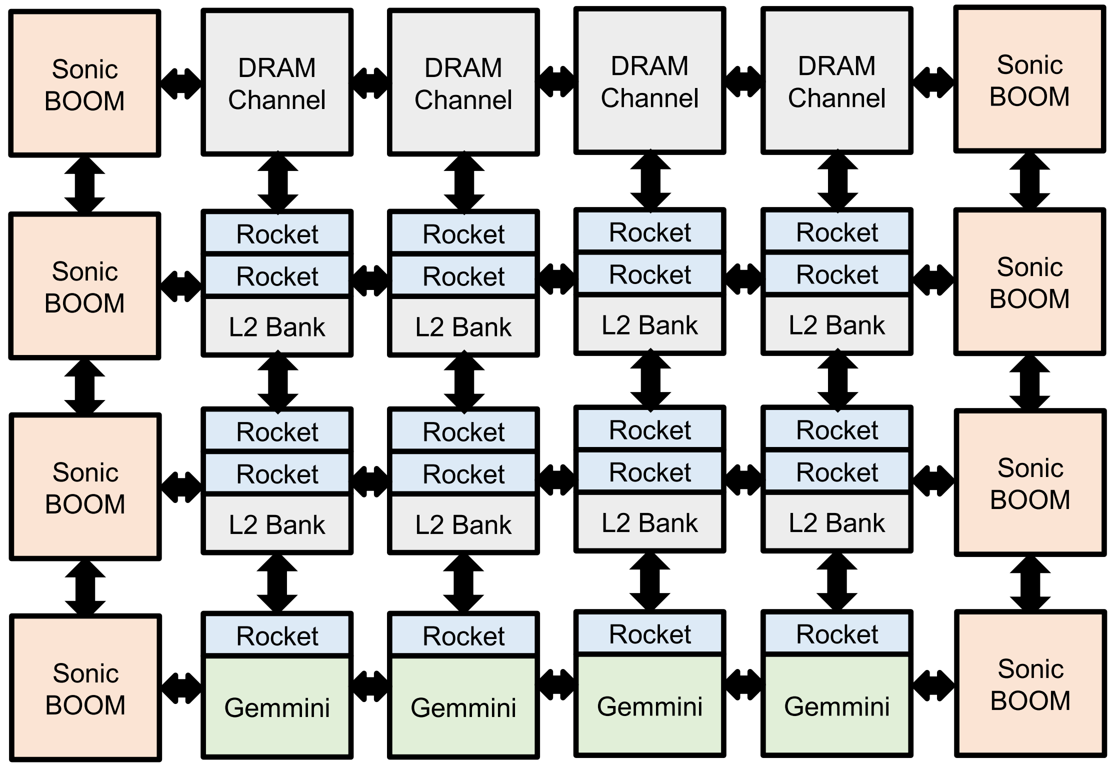
```
> cd $MCYDIR/generators/saturn/benchmarks
> make
> spike --isa=rv64gcv vec-sgemm-v3.riscv
> cp *.riscv $MCYDIR/sims/verilator

> cd $MCYDIR/sims/verilator
> make \
    CONFIG=GENV256D128ShuttleConfig \
    LOADMEM=1 \
    BINARY=vec-sgemm-v3.riscv \
    run-binary
```

Highly Configurable

Not limited to just configuring cores/accelerators/peripherals

Examples:

- Customize interface to off-chip memory (Serdes/QSPI/DDR)
- API for integrating PLLs, setting up clock muxes/dividers
- Custom interconnect topology/routing
- Clock-domain construction + CDC
- Coherent/incoherent memory architectures



```
> cd $MCYDIR/generators/chipyard
> cd src/main/scala/config
> ls
```

Many Example Configs

Cores

- BoomConfigs.scala
- RocketConfigs.scala
- ShuttleConfigs.scala
- IbexConfigs.scala
- CVA6Configs.scala
- SodorConfigs.scala
- VexiiRiscvConfigs.scala

Vector units

- SaturnConfigs.scala
- AraConfigs.scala

Tapeouts + VLSI

- ChipConfigs.scala
- ChipletConfigs.scala
- ClockingConfigs.scala

Multi-core

- HeteroConfigs.scala
- NoCConfigs.scala
- MemorySystemConfigs.scala

Accelerators+Peripherals

- MMIOAcceleratorConfigs.scala
- PeripheralDeviceConfigs.scala
- RoCCAcceleratorConfigs.scala

Great place to start to experiment with SoC design