# Copyright 2019-2021 T-Head Semiconductor Co., Ltd.
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
# http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
# See the License for the specific language governing permissions and
# limitations under the License.

This file will give some basic introduction of force-riscv about the following 2 topics:

1. The basic components force-riscv provides
2. A basic program, generated by force-riscv, exeution flow from user's perspective

## 1.1 basic components

Picture1 gives a block digram about the basic component of force-riscv.

- Config: force-riscv front-end and backend have some basic configuration files and APIs to help users to make force-riscv compatiable with your implementation and easy to get configuration info from "templates":
    - configs/riscv_xxx.config: which provides architecture info (thread/core limits, instruction, operand, paging, simulator related .etc.). There are front end APIs which help to get architecture info
    - fpix/configs/riscv_xxx.config: which provides fpix_riscv configruations. Fpix_riscv is a wrapper of handcar (spike integrated), which is part of force-riscv workflow to make QA validation
    - py/riscv/memory_xxx.py: which provides the expected memory region for cachable and non-cacheable areas. Architecture and implemetation defined memeory traits can be defined here too
    - py/riscv/PcConfig.py: which provides some address regions to make reset, boot, init .etc.

- Arch: force framework is expected to be easily expended to other cpu arhitectures with the following features:
    - Carefully designed xml based architecture description system for instruction/registers/operands/paging. For riscv, riscv/arch_data provides basic

information for riscv arhictecture info, which will be loaded by force framework on run-time.
    - Fine grained variables/knobs to control different architecture or program framework random options

- Memory: force provides a sparse memory model to manage data and instruction, with symbol and memory PMA traits enabled
- Virtual Memory: force-riscv provides a well designed virtual memory framework to support VA/PA allocation, paging management

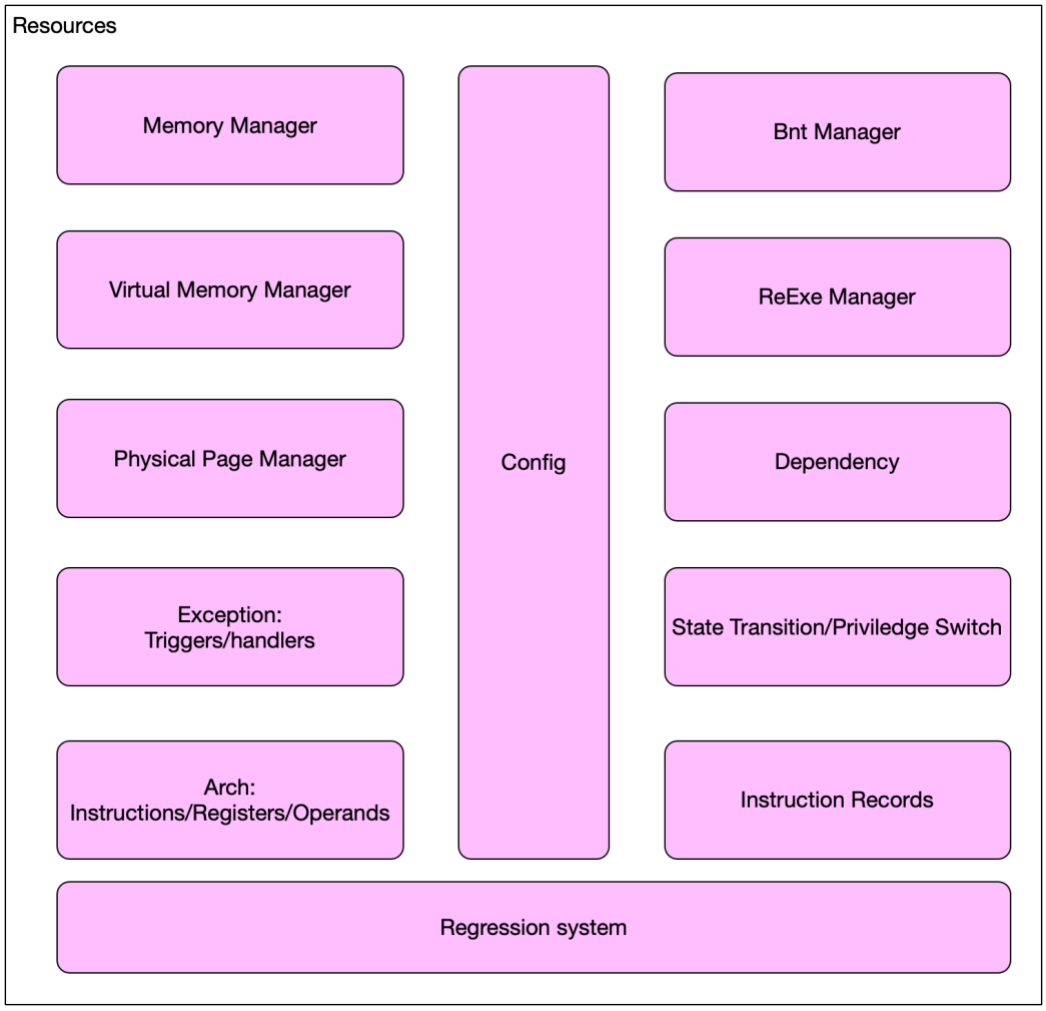
    - Provides memory mapper, address space for different regime type
    - Provides memory context, control block for paging map and context switch

- State Transition/Priviledge Switch: force-riscv provides baisc sequences to make cpu state transition and priviledge switch

    - To cover kernel context switch features, priviledge switch/virtual memory context switch is an inevitable feature
    - To make users focus on the scenario itself, state transition scheme provides easy-to-use toolset to change to specific cpu state

- Dependency: currently force-riscv provides basic algorithm for gpr dependency features
- ReExe/Bnt: for exception handling, exclusive features, branch related features, force-riscv provides basic scheme like Bnt and ReExec.

Picture1: force-riscv main components



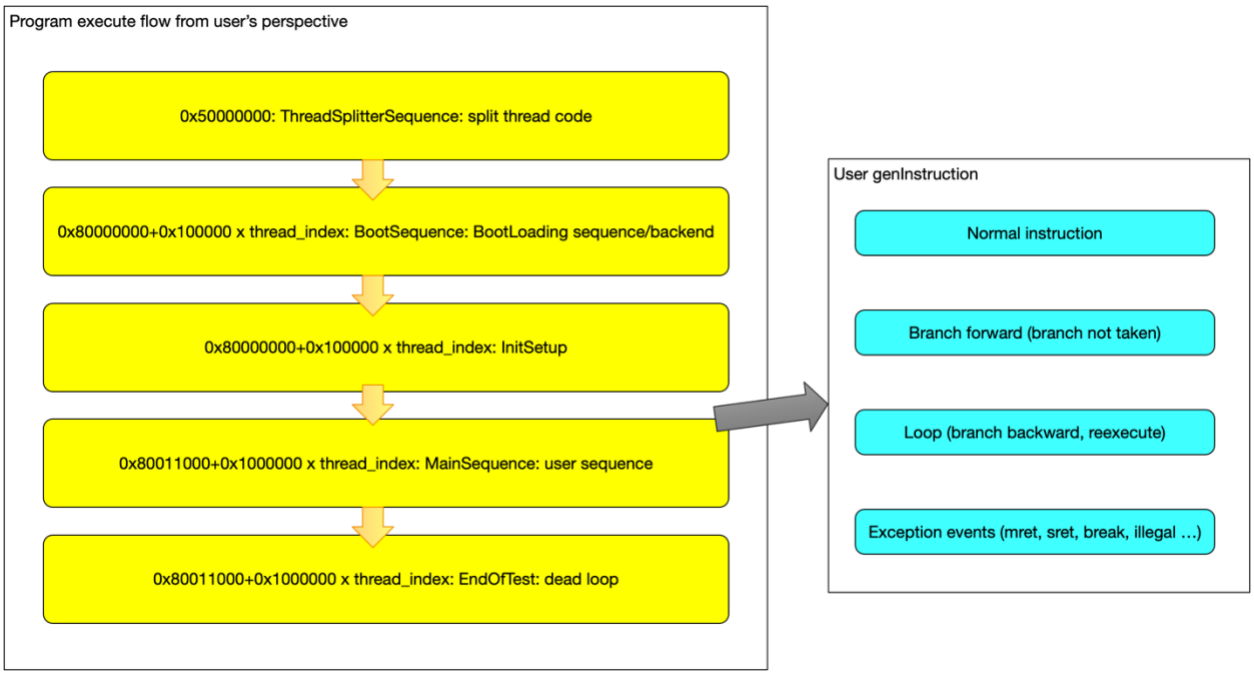

## 1.2 program execution flow from a user's perspective

As described in force-riscv/README.md, master_run.py can be used to regress all the templates integrated.
For each template config in _def_ctrl.py files, the following files will be generated by the forrest_run.py workflow:

- *.ELF, *.S files
- fpix_sim.log: fpix handcar execution log
- gen.log: friscv log file
- sim.log: force-riscv run-time handcar execution log

For a specific target (cpu simulator, cpu rtl testbench), *.ELF can be used to generate binary files, which is loaded into the target memory.
For HW DV users, an ISSCMP flow is excepcted to make lockstep-compare, which helps to check wrong execution flow.

Picture2 gives a brief view of how one force-riscv generated program executes on specific target (cpu core or simulator).
Picture2:




1. As the "reset_pc" in py/riscv/PcConfig.py defines, cpu will start from 0x50000000. ThreadSplitterSequence will try to split thread program pc to expected boot region for that hart
2. Cpu will run to bootloading sequence to make inevitable resouce initilization (gpr, csr .etc.), which is used in the subsequential programs
3. The architecture must to be initilized resource, InitSetup sequence in backend will make basic initilization flow, which will be executed in step2
4. After bootloading, cpu will run to user-defined template sequence
5. The template can invoke the front-end APIs to set variables/knobs, to allocate memory space, to generate instruction .etc.
6. After the template execution, cpu will run to EndOfTest sequence. The current scheme is a "branch-to-self" loop. User can try to change to tube shutdown scheme.