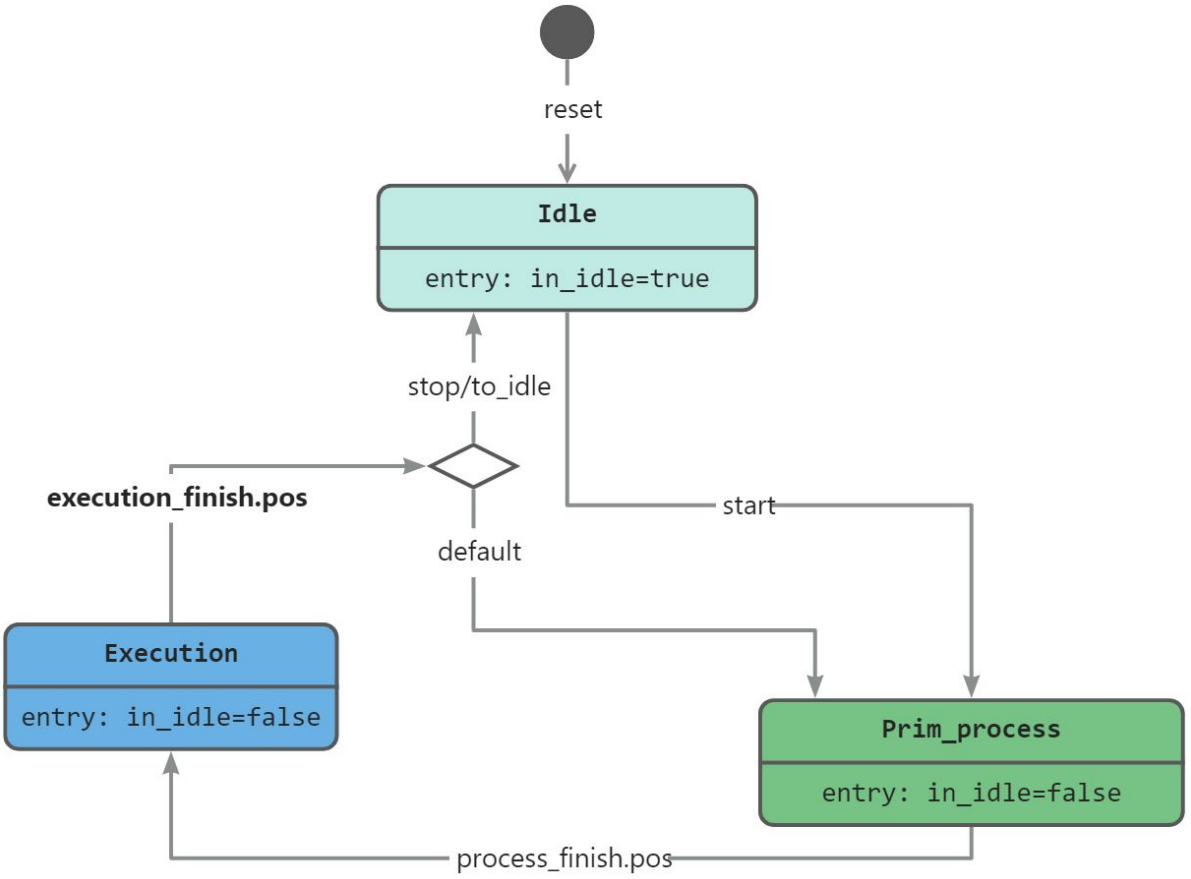
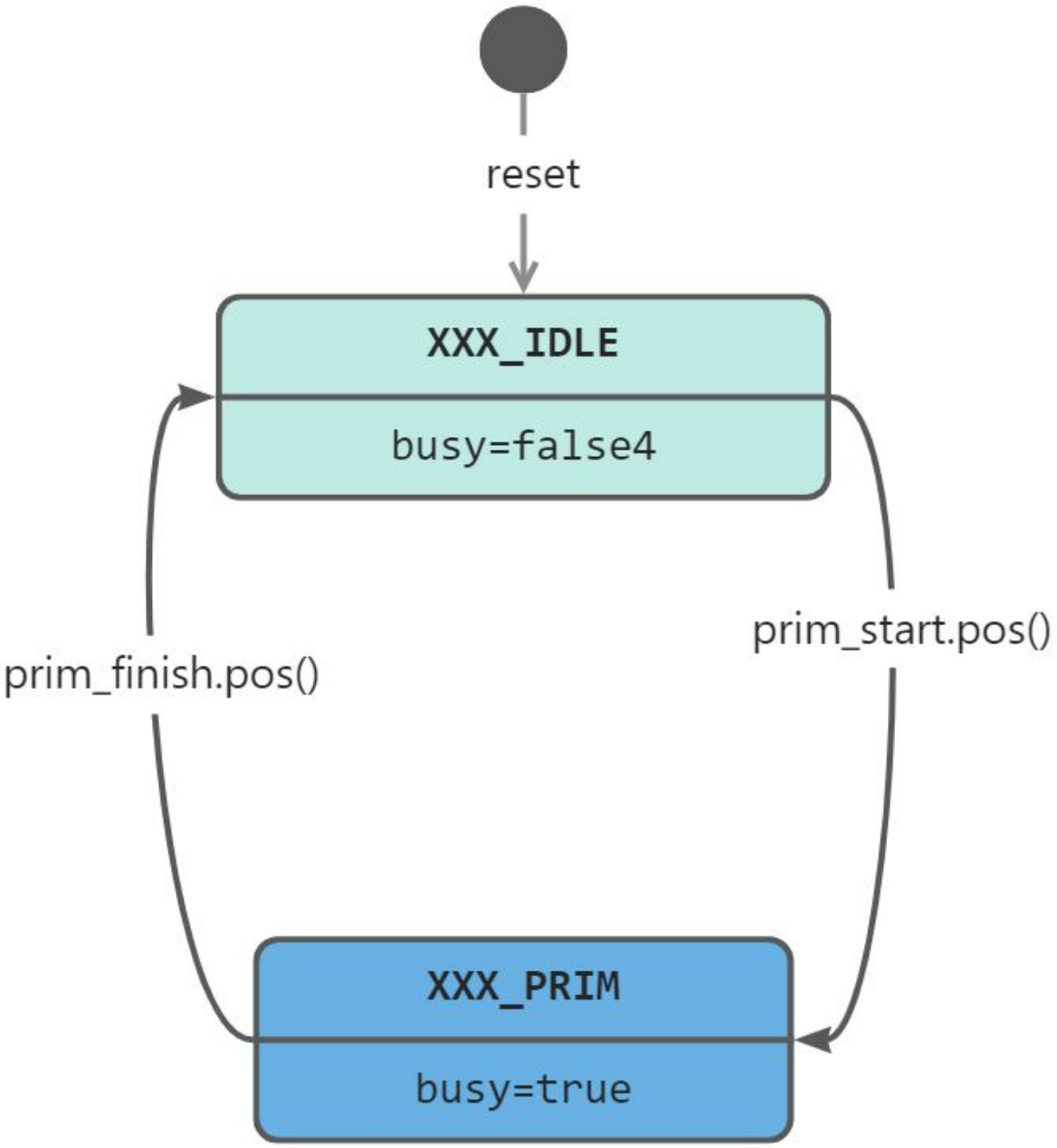
功能核状态机

请参考`core_control_unit.h`和`core_control_unit.cpp`中`ctrlStatusTrans()`函数



执行单元状态机

请参考任意模块的stateTrans函数



详细运行流程（代码Debug方式）

请根据下面的流程描述在代码的对应位置打断点，然后调试运行，方便理解。推荐将`single_core_test_relu.cpp`作为入口运行，注意代码中的英文注释。

1. `sc_main()`中`sc_start()`前：模块构建和连线阶段
2. `sc_start()`后进入`core_tb::test_cores()`函数
    1. `writeData()`和`writePrims`中数据和原语写入功能核
    2. `before_start()`运行前测试
    3. `start.write(true)`拉高`start`信号，功能核/芯片开始运行
3. `core_control_unit.cpp`中的`ctrlStatusTrans()`函数，状态由IDLE转换为PRIM_PROCESS
4. `core_control_unit.cpp`中的`run()`函数受状态变化触发，运行PRIM_PROCESS相应的操作（读取原语），最后拉高`process_finish`

5. `core_control_unit.cpp` 中的 `ctrlStatusTrans()` 函数受 `process_finish` 触发，状态由PRIM_PROCESS转换为EXECUTION
6. `core_control_unit.cpp` 中的 `run()` 函数受状态变化触发，运行EXECUTION相应的操作（读取原语），根据原语将不同执行单元的start信号拉高并写原语和执行参数，然后等待对应执行单元busy信号的下降沿 `wait(xxxxx_busy.negedge_event())`
7. 对应执行单元的 `core_xxx_unit.cpp` 中的 `xxxStateTrans()` 函数由start信号触发，状态由XXX_IDLE转换为XXX_PRIM
8. 对应执行单元的 `core_xxx_unit.cpp` 中的 `xxxPrimExecution()` 函数由状态转移触发，读取原语后执行
9. 原语和执行单元的具体执行过程参照Interface between Primitives and Units
10. 运行完成后 `xxxPrimExecution()` 函数将 `xxx_prim_finish` 信号拉高，触发 `xxxPrimExecution()` 函数执行状态转移，并将busy信号拉低
11. `core_control_unit.cpp` 中的 `run()` 函数等到busy信号后将 `execution_finish` 信号拉高
12. `core_control_unit.cpp` 中的 `ctrlStatusTrans()` 函数受到 `execution_finish` 信号触发：如果刚执行完成的原语是stop，则状态变为IDLE，功能核 `in_idle=true`；否则转移到PRIM_PROCESS，继续执行下一条原语。
13. 功能核进入IDLE后，`in_idle` 信号变高，`test.cpp` 中的 `core_tb::test_cores()` 函数从 `wait(core_in_idle.posedge_event())` 继续执行