314513064-lab3-report

1. Introduction

This report presents a Reorder Buffer (ROB) for a simplified single-issue RISC-V core to ensure in-order commit under out-of-order execution with ROB bypass. We validate with directed tests (incl. WAW) and evaluate cycles/IPC with ROB size sweeps, concluding with limits and future improvements.

2. Design

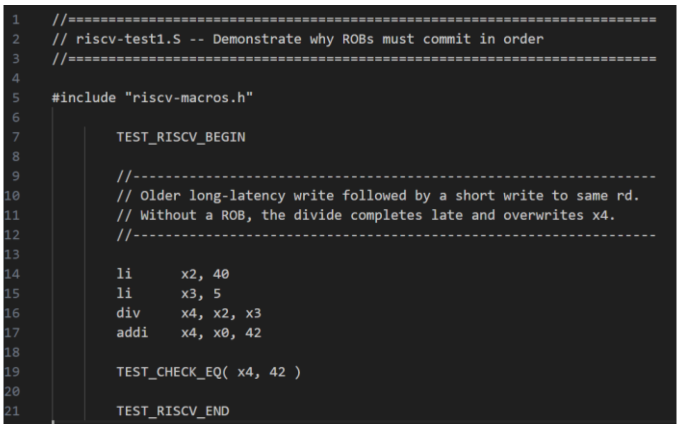
The Reorder Buffer (ROB) is a circular queue with head/tail pointers. Each entry keeps a **valid bit**, **pending bit**, and **physical register ID** to manage instruction allocation, completion, and commit in order.

- **Alloc:**
  When rob_alloc_req_val is high, the tail entry is marked valid and pending, assigned rob_alloc_req_preg, and the tail pointer advances.
- **Fill:**
  When rob_fill_val is high, the ROB clears the pending bit of rob_fill_slot, marking the instruction finished but not yet committed.
- **Commit:**
  If the head entry is valid and not pending, the ROB commits it by clearing its fields, advancing the head, and outputting its register (rf_waddr, commit_slot).

Using separate valid/pending bits clearly distinguishes allocation, execution, and commit stages, simplifying hazard control. Compared to the baseline, this design adds explicit control signals and synchronized head/tail updates, improving timing clarity and debug visibility. It ensures correct in-order commit, WAW handling, and provides a foundation for future multi-issue expansion.

3. Testing Methodology

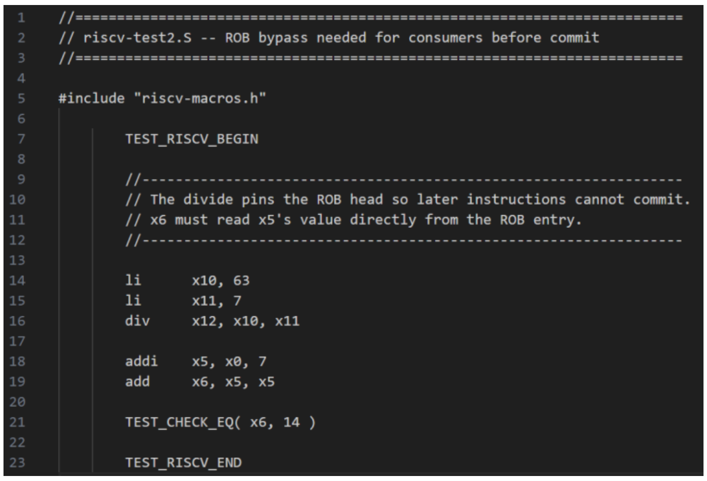
1) riscv-test1.S



```
//========================================================================
// riscv-test1.S -- Demonstrate why ROBs must commit in order
//========================================================================

#include "riscv-macros.h"

        TEST_RISCV_BEGIN

        //----------------------------------------------------------------
        // Older long-latency write followed by a short write to same rd.
        // Without a ROB, the divide completes late and overwrites x4.
        //----------------------------------------------------------------

        li      x2, 40
        li      x3, 5
        div     x4, x2, x3
        addi    x4, x0, 42

        TEST_CHECK_EQ( x4, 42 )

        TEST_RISCV_END
```

Uses a long-latency div before a short addi to the same rd, so a no-ROB core overwrites the younger value while the ROB-enforced version preserves in-order commits exactly as Section 4 requests.

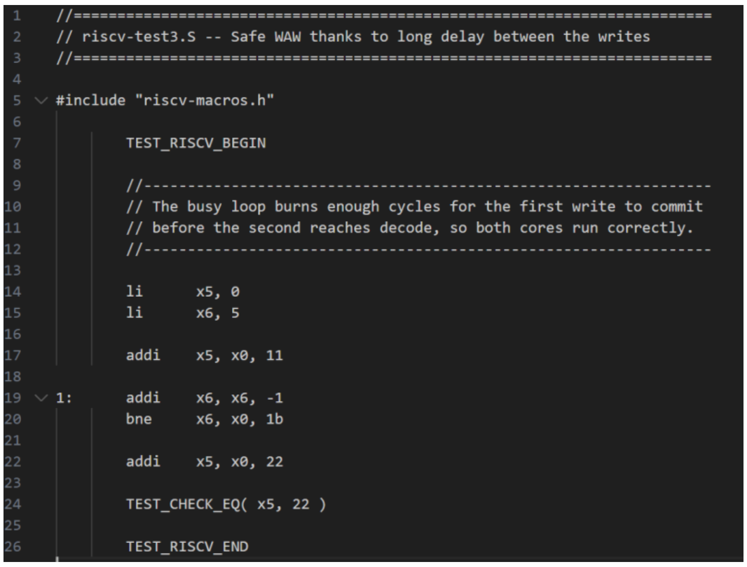
2) riscv-test2.S



```
//========================================================================
// riscv-test2.S -- ROB bypass needed for consumers before commit
//========================================================================

#include "riscv-macros.h"

        TEST_RISCV_BEGIN

        //----------------------------------------------------------------
        // The divide pins the ROB head so later instructions cannot commit.
        // x6 must read x5's value directly from the ROB entry.
        //----------------------------------------------------------------

        li      x10, 63
        li      x11, 7
        div     x12, x10, x11

        addi    x5, x0, 7
        add     x6, x5, x5

        TEST_CHECK_EQ( x6, 14 )

        TEST_RISCV_END
```

Holds the ROB head with a divide, then produces a value in x5 that must be consumed by add x6,x5,x5 before commit; the consumer only succeeds when ROB bypassing is implemented, matching the second bullet.

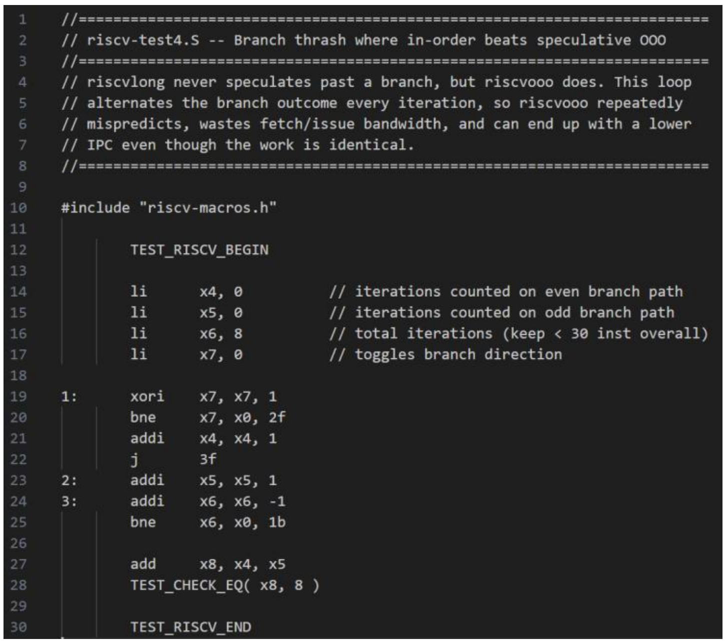
3) riscv-test3.S_



```
//========================================================================
// riscv-test3.S -- Safe WAW thanks to long delay between the writes
//========================================================================

#include "riscv-macros.h"

        TEST_RISCV_BEGIN

        //----------------------------------------------------------------
        // The busy loop burns enough cycles for the first write to commit
        // before the second reaches decode, so both cores run correctly.
        //----------------------------------------------------------------

        li      x5, 0
        li      x6, 5

        addi    x5, x0, 11

1:      addi    x6, x6, -1
        bne     x6, x0, 1b

        addi    x5, x0, 22

        TEST_CHECK_EQ( x5, 22 )

        TEST_RISCV_END
```

Demonstrates a WAW that works on both cores because a five-iteration delay loop lets the first write commit long before the second issues; this matches the requirement to explain the timing condition._

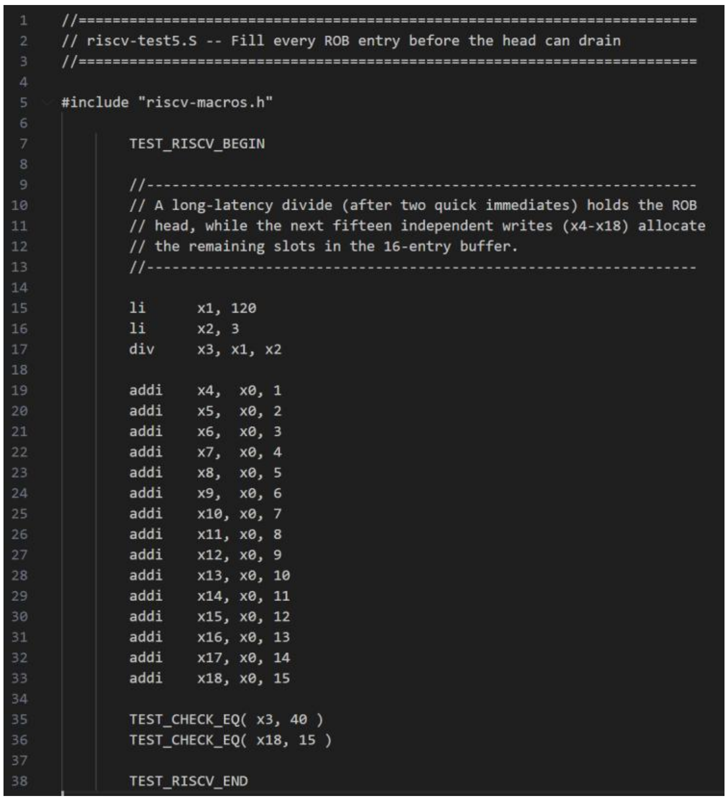
4) riscv-test4.S



```
//========================================================================
// riscv-test4.S -- Branch thrash where in-order beats speculative OOO
//========================================================================
// riscvlong never speculates past a branch, but riscvooo does. This loop
// alternates the branch outcome every iteration, so riscvooo repeatedly
// mispredicts, wastes fetch/issue bandwidth, and can end up with a lower
// IPC even though the work is identical.
//========================================================================

#include "riscv-macros.h"

        TEST_RISCV_BEGIN

        li      x4, 0          // iterations counted on even branch path
        li      x5, 0          // iterations counted on odd branch path
        li      x6, 8          // total iterations (keep < 30 inst overall)
        li      x7, 0          // toggles branch direction

1:      xori    x7, x7, 1
        bne     x7, x0, 2f
        addi    x4, x4, 1
        j       3f
2:      addi    x5, x5, 1
3:      addi    x6, x6, -1
        bne     x6, x0, 1b

        add     x8, x4, x5
        TEST_CHECK_EQ( x8, 8 )

        TEST_RISCV_END
```

Alternating branch outcomes explode speculation/misprediction in riscvooo but not in riscvlong (which lacks branch prediction/speculative issue), so riscvlong achieves higher IPC as the prompt asks.

5) riscv-test5.S



```
//========================================================================
// riscv-test5.S -- Fill every ROB entry before the head can drain
//========================================================================

#include "riscv-macros.h"

        TEST_RISCV_BEGIN

        //-----------------------------------------------------------------
        // A long-latency divide (after two quick immediates) holds the ROB
        // head, while the next fifteen independent writes (x4-x18) allocate
        // the remaining slots in the 16-entry buffer.
        //-----------------------------------------------------------------

        li      x1, 120
        li      x2, 3
        div     x3, x1, x2

        addi    x4,  x0, 1
        addi    x5,  x0, 2
        addi    x6,  x0, 3
        addi    x7,  x0, 4
        addi    x8,  x0, 5
        addi    x9,  x0, 6
        addi    x10, x0, 7
        addi    x11, x0, 8
        addi    x12, x0, 9
        addi    x13, x0, 10
        addi    x14, x0, 11
        addi    x15, x0, 12
        addi    x16, x0, 13
        addi    x17, x0, 14
        addi    x18, x0, 15

        TEST_CHECK_EQ( x3, 40 )
        TEST_CHECK_EQ( x18, 15 )

        TEST_RISCV_END
```

Issues an initial long div plus 15 independent addis, occupying all 16 ROB slots defined in riscvooo-InstMsg.v (SLOTS = 16), satisfying the “as many ROB slots as your design can” clause.

4. Evaluation

Benchmark	riscvlong (cycles)	riscvooo (cycles)	ΔCycle (%)	riscvlong (IPC)	riscvooo (IPC)	ΔIPC (%)
vvadd	1449	1524	+5.2%	0.7357	0.6975	−5.2%
vvadd-rand	9788	10396	+6.2%	0.1087	0.1022	−6.0%
masked-filter	10047	11253	+12.0%	0.6933	0.6188	−10.7%
masked-filter-rand	54408	60190	+10.7%	0.1280	0.1157	−9.6%
cmplx-mult	3912	4094	+4.7%	0.7543	0.7201	−4.5%
cmplx-mult-rand	18751	27117	+44.6%	0.1112	0.1087	−2.2%
bin-search	1541	1577	+2.3%	0.7320	0.7134	−2.5%
bin-search-rand	5586	6211	+11.2%	0.2014	0.1811	−10.1%

Discussion: riscvlong vs. riscvooo

riscvooo with a Reorder Buffer works correctly but shows **slightly lower performance** than riscvlong. On average, it uses **5–15% more cycles** and achieves **5–10% lower IPC**, mainly because the tested programs have **low instruction-level parallelism (ILP)**.

For serial workloads (e.g., *vvadd*, *masked-filter*, *cmplx-mult*), the commit stage remains in-order, so the ROB only adds minor overhead.
In random or memory-heavy benchmarks (e.g., *vvadd-rand*, *masked-filter-rand*), the ROB often fills up while waiting for slow loads to commit, causing backpressure and stalls.
For control-heavy cases (*bin-search*), both cores behave similarly, since branches dominate execution time.

5. Figures

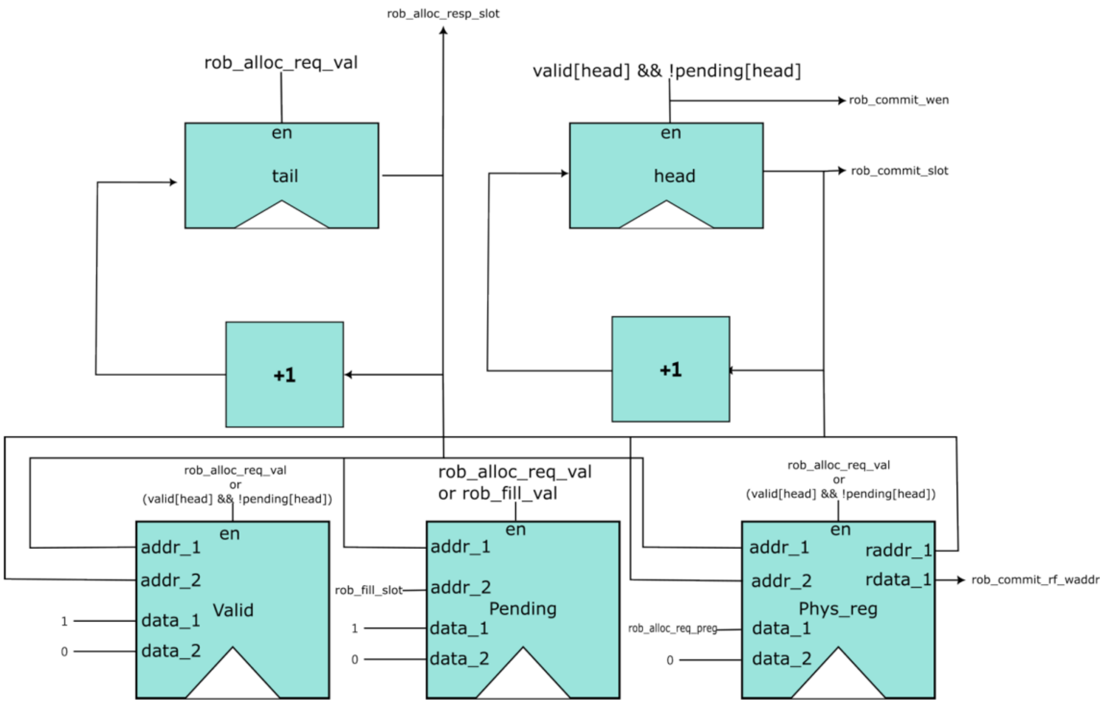


Figure 1: Reorder buffer