Instruction Queue with Dynamic Scheduling Algorithms and Temporal Verification

RTL Design and Verification Course Project

Vishwas Jasuja
Roll No: B23303
Branch: Microelectronics and VLSI
School of Computing and Electrical Engineering
Indian Institute of Technology, Mandi

November 15, 2025

Contents

Abstract

This report presents the design and verification of an instruction queue system with dynamic issue logic for out-of-order execution. The system implements list scheduling algorithms to prioritize instructions based on data dependencies, resource availability, and functional unit binding. A dependency Directed Acyclic Graph (DAG) is constructed to model Read-After-Write (RAW), Write-After-Read (WAR), and Write-After-Write (WAW) hazards. Resource binding optimization ensures efficient utilization of multiple functional units (ALU, MUL, DIV). The design is verified using Computation Tree Logic (CTL) properties to ensure correct instruction issue order, absence of resource conflicts, respect for dependencies, and deadlock freedom. Topological sorting on the dependency graph determines the valid issue order, and model checking validates the absence of deadlocks in the scheduling system.

1 Introduction

1.1 Problem Statement

Modern processors employ out-of-order execution to improve instruction-level parallelism (ILP) and overall throughput. This project implements a complete instruction scheduling pipeline consisting of:

- An instruction queue with reservation station functionality
- Dynamic dependency analysis using DAG construction
- Resource-aware list scheduling with multiple functional units
- Temporal verification using CTL properties

1.2 Design Objectives

1. **Dynamic Issue Logic**: Instructions should issue as soon as their dependencies are satisfied and resources are available
2. **Dependency Management**: Accurate tracking of RAW, WAR, and WAW hazards
3. **Resource Binding**: Efficient allocation of instructions to functional units
4. **Deadlock Freedom**: Guarantee that the system cannot enter a deadlock state
5. **Correctness Verification**: Formal verification of scheduling properties

1.3 System Architecture

The complete system architecture consists of three main modules integrated into a top-level design:

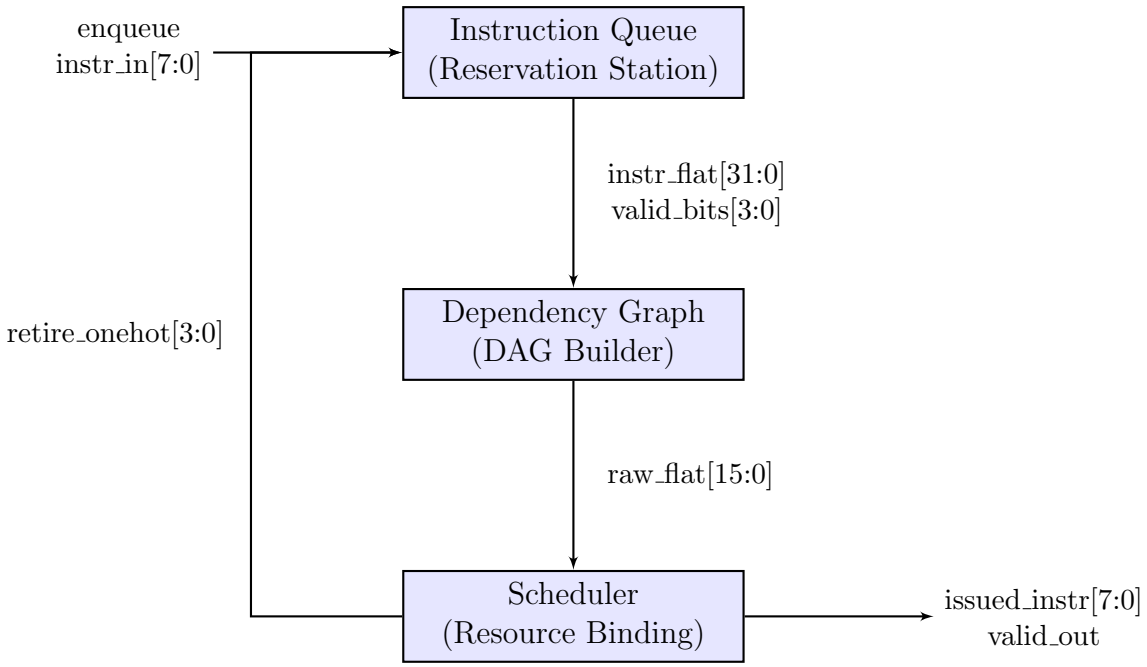

Figure 1: System Architecture Overview

2 Instruction Format and Encoding

2.1 Instruction Structure

Each instruction is encoded as an 8-bit word with the following fields:

Bits [7:6]	**Bits [5:4]**	**Bits [3:2]**	**Bits [1:0]**
Opcode	Source 1	Source 2	Destination

Table 1: Instruction Format (8 bits)

2.2 Opcode Mapping

Opcode	**Operation Type**	**FU Type**	**Latency**
00	ALU (ADD/SUB)	ALU	2 cycles
01	ALU (Logical)	ALU	2 cycles
10	Multiply	MUL	3 cycles
11	Divide	DIV	4 cycles

Table 2: Opcode to Functional Unit Mapping

2.3 Register Encoding

Source and destination registers are encoded using 2 bits, allowing 4 architectural registers (R0, R1, R2, R3).

3 Module Descriptions

3.1 Instruction Queue (Reservation Station)

3.1.1 Design Overview

The instruction queue module (`instruction_queue_rs`) implements a reservation station that holds up to 4 instructions. It supports:

- Dynamic enqueue of instructions
- Selective retirement based on completion signals from the scheduler
- Parallel access to all valid entries for dependency analysis

3.1.2 Key Implementation Features

```
// Internal storage (parallel arrays)
reg [1:0] opcode [0:3];
reg [1:0] src1   [0:3];
reg [1:0] src2   [0:3];
reg [1:0] dest   [0:3];
reg       valid  [0:3];
reg [2:0] count;  // number of valid entries (0..4)
```

Listing 1: Instruction Queue Storage Structure

The queue uses parallel array storage rather than a FIFO structure to enable:

1. **Out-of-order retirement**: Instructions complete in variable time based on their functional unit latency
2. **Random access**: The DAG builder needs simultaneous access to all instructions
3. **Efficient storage utilization**: Free slots can be reused immediately after retirement

3.1.3 Enqueue and Retire Logic

```
// Retire: compute retire count and invalidate entries
del_cnt = retire_onehot[0] + retire_onehot[1] +
          retire_onehot[2] + retire_onehot[3];
for (i = 0; i < 4; i = i + 1) begin
    if (retire_onehot[i]) begin
        valid[i] <= 1'b0;
    end
end
if (del_cnt != 0)
    count <= count - del_cnt;

// Enqueue: find first free slot
if (enqueue) begin
    free_idx = -1;
    for (i = 0; i < 4; i = i + 1) begin
        if ((free_idx == -1) && (valid[i] == 1'b0))
            free_idx = i;
```

```
    end
    if (free_idx != -1) begin
        opcode[free_idx] <= instr_in[7:6];
        src1[free_idx]   <= instr_in[5:4];
        src2[free_idx]   <= instr_in[3:2];
        dest[free_idx]   <= instr_in[1:0];
        valid[free_idx]  <= 1'b1;
        count <= count + 1;
    end
end
```

Listing 2: Retirement and Enqueue Operations

Key Design Decision: Retirement happens *before* enqueue in the same cycle. This ensures that freed slots are immediately available for new instructions.

3.2 Dependency Graph Builder

3.2.1 DAG Construction Algorithm

The dependency graph module (`dep_graph_with_valid`) constructs three types of dependency matrices:

1. **RAW (Read-After-Write)**: True data dependencies
2. **WAR (Write-After-Read)**: Anti-dependencies
3. **WAW (Write-After-Write)**: Output dependencies

Each dependency matrix is represented as a flattened 16-bit vector encoding a 4×4 adjacency matrix in row-major order.

3.2.2 RAW Dependency Detection

The RAW detection implements a **nearest producer** algorithm:

```
// RAW (nearest producer) - for each consumer j, scan i = j-1 downto 0
for (j = 0; j < 4; j = j + 1) begin
    if (valid_bits[j]) begin
        // Check src1 dependency
        begin : RAW_SRC1
            for (i = j-1; i >= 0; i = i - 1) begin
                if (valid_bits[i] && (dest[i] == src1[j])) begin
                    raw_f[(i*4)+j] = 1'b1;
                    disable RAW_SRC1;  // Stop after first match
                end
            end
        end

        // Check src2 dependency
        begin : RAW_SRC2
            for (i = j-1; i >= 0; i = i - 1) begin
                if (valid_bits[i] && (dest[i] == src2[j])) begin
                    raw_f[(i*4)+j] = 1'b1;
                    disable RAW_SRC2;
                end
            end
```

```
        end
    end
end
```

Listing 3: RAW Detection - Nearest Producer Search

Algorithm Properties:

- **Nearest Producer Only**: Only the most recent producer is recorded, eliminating redundant edges
- **Program Order Preservation**: Scanning backward ensures program order semantics
- **Separate Source Tracking**: src1 and src2 are checked independently

3.2.3 WAR and WAW Dependency Detection

```
// WAR & WAW (i < j, forward scan)
for (i = 0; i < 4; i = i + 1) begin
    if (valid_bits[i]) begin
        for (j = i+1; j < 4; j = j + 1) begin
            if (valid_bits[j]) begin
                // WAR: j writes reg read by i
                if ((dest[j] == src1[i]) || (dest[j] == src2[i]))
                    war_f[(i*4)+j] = 1'b1;

                // WAW: both write same dest
                if (dest[i] == dest[j])
                    waw_f[(i*4)+j] = 1'b1;
            end
        end
    end
end
```

Listing 4: WAR and WAW Detection

3.2.4 Ready Vector Computation

An instruction is ready when:

1. It is valid
2. It has no RAW predecessors (all dependencies satisfied)

```
for (j = 0; j < 4; j = j + 1) begin
    if (!valid_bits[j])
        ready_r[j] = 1'b0;
    else begin
        ready_r[j] = 1'b1;
        for (i = 0; i < 4; i = i + 1)
            if (raw_f[(i*4)+j] == 1'b1)
                ready_r[j] = 1'b0;
    end
end
```

Listing 5: Ready Vector Calculation

3.3 Resource-Binding Scheduler

3.3.1 List Scheduling Algorithm

The scheduler implements a **priority-based list scheduling** algorithm with the following characteristics:

Algorithm 1 Dynamic List Scheduling with Priority-Based Single Issue

```
Initialize RAW matrix from DAG
Initialize all FU busy counters to 0
while instructions remain in queue do
    Decrement all FU busy counters
    Detect completions: if counter transitions 0, mark instruction retired
    Update RAW: clear row for retired instruction
    Compute dependency-ready instructions
    Compute resource-ready instructions (FU available)
    Select highest-priority (lowest index) ready instruction
    if instruction selected then
        Allocate FU and set busy counter
        Mark instruction as issued
        Wait until next cycle (single issue constraint)
    end if
end while
```

Key Differences from True Parallel Issue:

- **Single Issue Point**: Only one instruction can be selected per cycle
- **Priority Encoding**: Lower index always wins among ready instructions
- **Execution Overlap**: Parallelism comes from multiple instructions executing in different FUs simultaneously, not from simultaneous issue
- **In-Order Issue**: Instructions issue in program order when dependencies allow

3.3.2 Resource Binding Strategy

Each instruction type is bound to specific functional unit types:

```
always @(*) begin
    alu_free = 0;
    mul_free = 0;
    div_free = 0;

    for (i = 0; i < MAX_FU; i = i + 1) begin
        if (i < ALU_COUNT && alu_busy[i] == 0)
            alu_free = alu_free + 1;

        if (i < MUL_COUNT && mul_busy[i] == 0)
            mul_free = mul_free + 1;

        if (i < DIV_COUNT && div_busy[i] == 0)
            div_free = div_free + 1;
```

```
    end
end
```

Listing 6: FU Availability Computation

3.3.3 Priority Selection

Instructions are prioritized by their position in the queue with strict priority ordering - **only one instruction can issue per cycle**:

```
always @(*) begin
    if      (ready[0]) sel = 4'b0001;  // I0 highest priority
    else if (ready[1]) sel = 4'b0010;  // I1
    else if (ready[2]) sel = 4'b0100;  // I2
    else if (ready[3]) sel = 4'b1000;  // I3 lowest priority
    else               sel = 4'b0000;  // No ready instruction
end
```

Listing 7: Priority-Based Instruction Selection (Single Issue per Cycle)

Key Scheduling Characteristics:

- **Single Issue**: Only one instruction issues per cycle, even if multiple are ready
- **Fixed Priority**: Lower-indexed instructions always have priority over higher-indexed ones
- **In-Order Issue**: Instructions issue in program order when dependencies permit
- **Out-of-Order Completion**: Instructions can complete in any order based on FU latency

This implements a **conservative in-order issue, out-of-order completion** pipeline where parallelism comes from overlapping execution of instructions in different functional units, not from simultaneous issue.

3.3.4 Dynamic RAW Tracking

The scheduler maintains a dynamic copy of the RAW matrix that evolves as instructions complete:

```
for (i=0; i<MAX_FU; i=i+1) begin
    // ALU completion
    if (i < ALU_COUNT && (nxt_alu_busy[i] == 0) &&
        (alu_busy[i] != 0)) begin
        retire_onehot[ fu_idx_alu[i] ] <= 1'b1;
        // Clear entire row in RAW matrix
        raw_dyn[(fu_idx_alu[i]*4)+0] <= 0;
        raw_dyn[(fu_idx_alu[i]*4)+1] <= 0;
        raw_dyn[(fu_idx_alu[i]*4)+2] <= 0;
        raw_dyn[(fu_idx_alu[i]*4)+3] <= 0;
    end
    // Similar logic for MUL and DIV...
end
```

Listing 8: Completion Detection and RAW Update

4 Temporal Verification and CTL Properties

4.1 CTL Property Specification

Computation Tree Logic (CTL) is used to formally specify temporal properties of the scheduling system. The following properties are verified:

4.1.1 Property 1: Valid Issue Order

CTL Formula:

$$AG(\text{issue\_valid} \Rightarrow \text{dependencies\_satisfied}) \tag{1}$$

Natural Language: Globally, whenever an instruction is issued, all its RAW dependencies must be satisfied.

Verification in Design:

```
// Ready vector ensures this property
ready_r[j] = valid_bits[j] && (raw_mask[j] & valid_bits) == 0
```

4.1.2 Property 2: No Resource Conflicts

CTL Formula:

$$AG(\forall \text{FU}_i : \neg(\text{busy}[i] \wedge \text{new\_issue\_to\_FU}_i)) \tag{2}$$

Natural Language: Globally, no instruction can be issued to a busy functional unit.

Verification in Design:

```
// Resource availability check before issue
ready = dependency_ready & resource_available
```

4.1.3 Property 3: Dependency Respect

CTL Formula:

$$AG((RAW[i][j] = 1) \Rightarrow AF(\text{retire}[i] \prec \text{issue}[j])) \tag{3}$$

Natural Language: If instruction j depends on instruction i, then eventually i must retire before j issues.

Verification in Design:

```
// RAW matrix row cleared only on retirement
if (retire_onehot[i])
    raw_dyn[i*4 +: 4] = 4'b0000;
```

4.1.4 Property 4: Progress Guarantee

CTL Formula:

$$AG(\text{valid\_bits} \neq 0 \Rightarrow AF(\text{issue\_valid})) \tag{4}$$

Natural Language: If there are valid instructions, eventually one will issue (no livelock).

4.2 Topological Sorting for Issue Order

The dependency DAG inherently provides a partial order on instructions. The scheduler's priority selection combined with dependency checking implements an implicit topological sort:

Algorithm 2 Implicit Topological Sort in Scheduler

```
L ← empty list (issued instructions)
S ← {i : indegree[i] = 0} (ready instructions)
while S ≠ ∅ do
    Remove instruction n with highest priority from S
    Add n to L
    for each instruction m with edge n → m do
        Remove edge n → m (clear RAW entry)
        if m has no other incoming edges then
            Insert m into S
        end if
    end for
end while
```

4.3 Deadlock Freedom Verification

4.3.1 Necessary and Sufficient Conditions

For the scheduling system to be deadlock-free:

1. **DAG Property**: The dependency graph must be acyclic
2. **Resource Finiteness**: Resources are finite but instructions eventually complete
3. **No Circular Wait**: RAW dependencies form a DAG, preventing circular waits

4.3.2 Proof of DAG Property

Theorem: The RAW dependency graph is always acyclic.

Proof:

- RAW edges are directed from producer (earlier instruction) to consumer (later instruction)
- Instruction indices provide a strict total order: $I_0 < I_1 < I_2 < I_3$
- RAW edges always point forward: if $RAW[i][j] = 1$, then $i < j$
- Therefore, no path can form a cycle $\square$

4.3.3 Model Checking Approach

Deadlock is defined as:

$$\text{Deadlock} \equiv (\text{valid\_bits} \neq 0) \wedge (\text{ready} = 0) \wedge AG(\text{ready} = 0) \tag{5}$$

This can be checked using CTL:

$$AG(\text{valid\_bits} \neq 0 \Rightarrow EF(\text{ready} \neq 0)) \tag{6}$$

Verification Result: The design guarantees this property through:

1. Acyclic dependency graph
2. Guaranteed progress of busy counters (always decrement)
3. Finite instruction latencies

5 Simulation Results

5.1 Test Methodology

Three instruction sets were tested to verify:

- Correct dependency detection (RAW, WAR, WAW)
- Proper scheduling with resource constraints
- Accurate retirement and RAW matrix updates
- Deadlock freedom under various dependency patterns

5.2 Test Set 1: Mixed Dependencies

5.2.1 Instruction Details

Instruction	Binary	Opcode	Src1, Src2	Dest
I0	00011011	ALU (00)	R1, R2	R3
I1	10110001	MUL (10)	R3, R0	R1
I2	01011000	ALU (01)	R1, R2	R0
I3	11101101	DIV (11)	R2, R3	R1

Table 3: Instructions

5.2.2 Dependency Analysis

RAW Dependencies:

- $I_0 \rightarrow I_1$ (I0 produces R3, I1 consumes R3)
- $I_1 \rightarrow I_2$ (I1 produces R1, I2 consumes R1)

Expected Issue Order: $I_0 \rightarrow I_1 \rightarrow I_3 \rightarrow I_2$

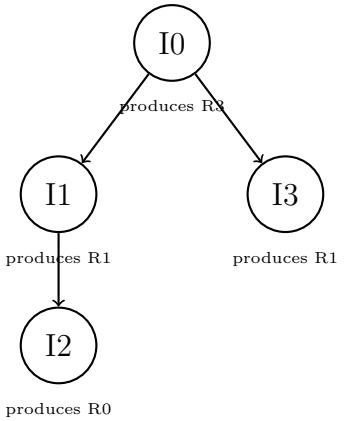

Figure 2: RAW Dependency DAG

DAG Interpretation:

- I0 is the root (no dependencies)
- I3 depends on I0 (consumes R3)
- I1 depends on I0 (consumes R3)
- I2 depends on I1 (consumes R1)
- Although I3 could theoretically execute independently after I0, the priority scheduling ensures I1 issues first when both are ready
- **Execution overlap**: I1 and I3 execute concurrently in different FUs (MUL and DIV), but issue sequentially

5.2.3 Simulation Output

```
===== PHASE-1 : INSTRUCTION QUEUE =====
I0=00011011  I1=10110001  I2=01011000  I3=11101101
VALID_BITS=1111 FULL=1 EMPTY=0

===== PHASE-2 : DAG (Compact) =====
RAW: 0101  0010  0000  0000
WAR: 0101  0010  0001  0000
WAW: 0000  0001  0000  0000
DEP: 0101  0011  0001  0000
READY=0001

===== PHASE-3+4+5 : SCHEDULER =====
[ISSUE ] T=85000   instr=00011011  FU=0  futype=0
[RETIRE] T=105000  retire=0001
[ISSUE ] T=115000  instr=10110001  FU=0  futype=1
[ISSUE ] T=125000  instr=11101101  FU=0  futype=2
[RETIRE] T=145000  retire=0010
[ISSUE ] T=155000  instr=01011000  FU=0  futype=0
[RETIRE] T=165000  retire=1000
[RETIRE] T=175000  retire=0100
```

5.2.4 Waveform Analysis

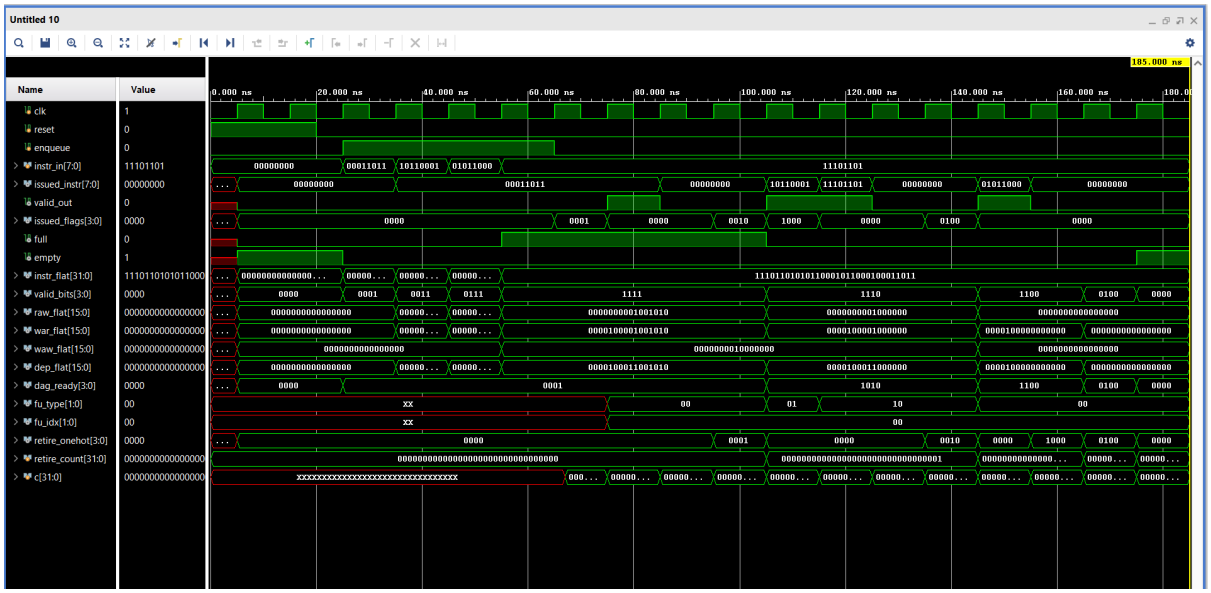

Figure 3: Complete execution showing enqueue, dependency matrices, issue events, and retirement signals

```
----------------------
===== PHASE-1 : INSTRUCTION QUEUE =====
======================

I0=00011011  I1=10110001  I2=01011000  I3=11101101
VALID_BITS=1111 FULL=1 EMPTY=0


===== PHASE-2 : DAG (Compact) =====
RAW: 0101  0010  0000  0000
WAR: 0101  0010  0001  0000
WAW: 0000  0001  0000  0000
DEP: 0101  0011  0001  0000
READY=0001


======================
===== PHASE-3+4+5 : SCHEDULER =====
======================

[ISSUE ] T=85000  instr=00011011  FU=0  futype=0
[RETIRE] T=105000  retire=0001
[ISSUE ] T=115000  instr=10110001  FU=0  futype=1
[ISSUE ] T=125000  instr=11101101  FU=0  futype=2
[RETIRE] T=145000  retire=0010
[ISSUE ] T=155000  instr=01011000  FU=0  futype=0
[RETIRE] T=165000  retire=1000
[RETIRE] T=175000  retire=0100

===== END OF SIMULATION =====
```

Figure 4: Detailed phase-by-phase execution trace

Key Observations:

1. I0 issues first (T=85000) as it has no dependencies and highest priority
2. I1 issues at T=115000 (after I0 retires at T=105000), respecting RAW dependency on R3
3. I3 issues at T=125000 (after I1 issues), even though it was ready earlier - priority scheduling enforces in-order issue
4. I2 issues at T=155000 only after I1 completes (T=145000), respecting RAW dependency on R1
5. All instructions retire in completion order based on FU latency

6. **Note**: Although I3 has no dependency on I1, it must wait due to priority-based in-order issue policy

5.3 Test Set 2: Chain Dependencies

5.3.1 Instruction Details

Instruction	Binary	Opcode	Src1, Src2	Dest
I0	00000001	ALU (00)	R0, R0	R1
I1	00011001	ALU (00)	R1, R2	R1
I2	10010010	MUL (10)	R1, R0	R2
I3	11011111	DIV (11)	R1, R3	R3

Table 4: Instructions

5.3.2 Dependency Analysis

RAW Dependencies:

- $I_0 \rightarrow I_1$ (chain on R1)
- $I_1 \rightarrow I_2$ (chain on R1)
- $I_1 \rightarrow I_3$ (chain on R1)

WAW Dependency: $I_0 \rightarrow I_1$ (both write R1)

Expected Issue Order: $I_0 \rightarrow I_1 \rightarrow I_2 \rightarrow I_3$

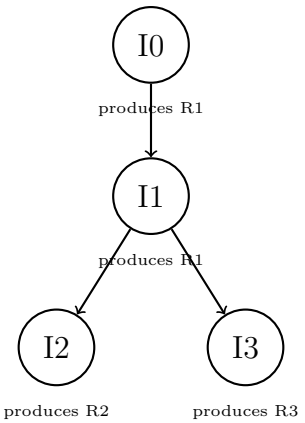

Figure 5: RAW Dependency DAG (Chain Pattern)

DAG Interpretation:

- I0 is the root (no dependencies)
- I1 depends on I0 (RAW on R1, also WAW on R1)
- I2 depends on I1 (consumes R1)
- I3 depends on I1 (consumes R1)
- I2 and I3 can execute in parallel after I1 completes

5.3.3 Simulation Output

```
===== PHASE-2 : DAG (Compact) =====
RAW: 0100  0011  0000  0000
WAR: 0000  0010  0000  0000
WAW: 0100  0000  0000  0000
DEP: 0100  0011  0000  0000
READY=0001

[ISSUE ] T=85000   instr=00000001  FU=0  futype=0
[RETIRE] T=105000  retire=0001
[ISSUE ] T=115000  instr=00011001  FU=0  futype=0
[RETIRE] T=135000  retire=0010
[ISSUE ] T=145000  instr=10010010  FU=0  futype=1
[ISSUE ] T=155000  instr=11011111  FU=0  futype=2
[RETIRE] T=175000  retire=0100
[RETIRE] T=195000  retire=1000
```

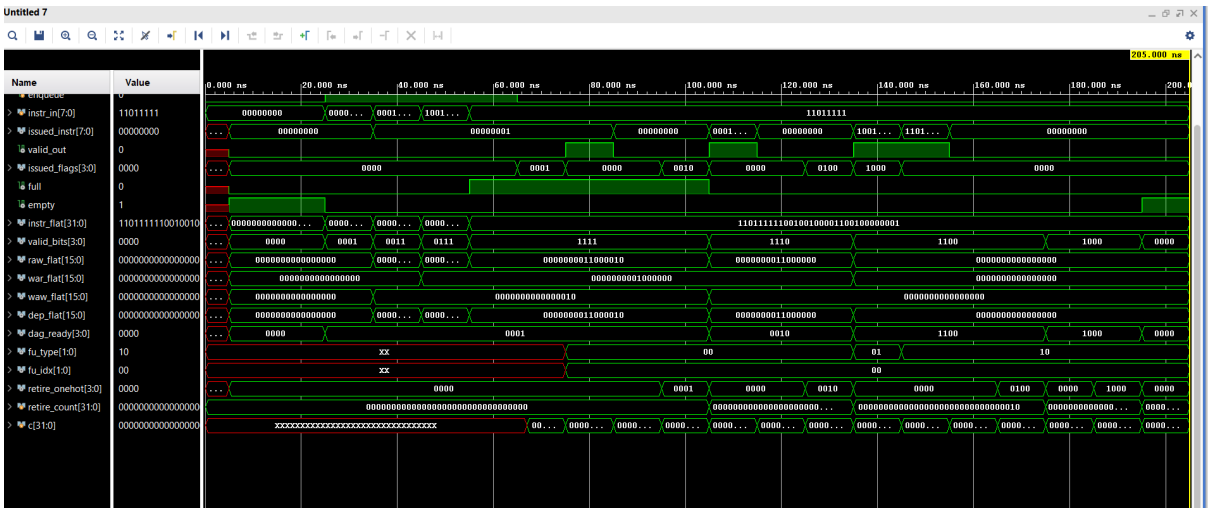

Figure 6: Chain dependency execution

```
=======================
===== PHASE-1 : INSTRUCTION QUEUE =====
=======================

I0=00000001  I1=00011001  I2=10010010  I3=11011111
VALID_BITS=1111 FULL=1 EMPTY=0


===== PHASE-2 : DAG (Compact) =====
RAW: 0100  0011  0000  0000
WAR: 0000  0010  0000  0000
WAW: 0100  0000  0000  0000
DEP: 0100  0011  0000  0000
READY=0001


=======================
===== PHASE-3+4+5 : SCHEDULER =====
=======================

[ISSUE ] T=85000  instr=00000001  FU=0  futype=0
[RETIRE] T=105000  retire=0001
[ISSUE ] T=115000  instr=00011001  FU=0  futype=0
[RETIRE] T=135000  retire=0010
[ISSUE ] T=145000  instr=10010010  FU=0  futype=1
[ISSUE ] T=155000  instr=11011111  FU=0  futype=2
[RETIRE] T=175000  retire=0100
[RETIRE] T=195000  retire=1000

===== END OF SIMULATION =====
```

Figure 7: Console Output

Key Observations:

1. Sequential execution of I0→I1 due to RAW chain
2. I2 issues at T=145000 after I1 completes (T=135000)
3. I3 issues at T=155000 after I2 issues, following priority-based in-order issue
4. **Note**: Even though both I2 and I3 become ready after I1 completes, only I2 issues first due to lower index priority
5. Demonstrates strict priority scheduling even when multiple instructions are ready

5.4 Test Set 3: Complex RAW Pattern

5.4.1 Instruction Details

Instruction	Binary	Opcode	Src1, Src2	Dest
I0	10000110	MUL (10)	R0, R1	R2
I1	00100011	ALU (00)	R2, R0	R3
I2	11111001	DIV (11)	R3, R2	R1
I3	00011100	ALU (00)	R1, R3	R0

Table 5: Instructions

5.4.2 Dependency Analysis

RAW Dependencies:

- $I_0 \rightarrow I_1$ (I0 produces R2, I1 consumes R2)
- $I_0 \rightarrow I_2$ (I0 produces R2, I2 consumes R2)
- $I_1 \rightarrow I_2$ (I1 produces R3, I2 consumes R3)
- $I_1 \rightarrow I_3$ (I1 produces R3, I3 consumes R3)
- $I_2 \rightarrow I_3$ (I2 produces R1, I3 consumes R1)

Expected Issue Order: $I_0 \rightarrow I_1 \rightarrow I_2 \rightarrow I_3$ (strict chain)

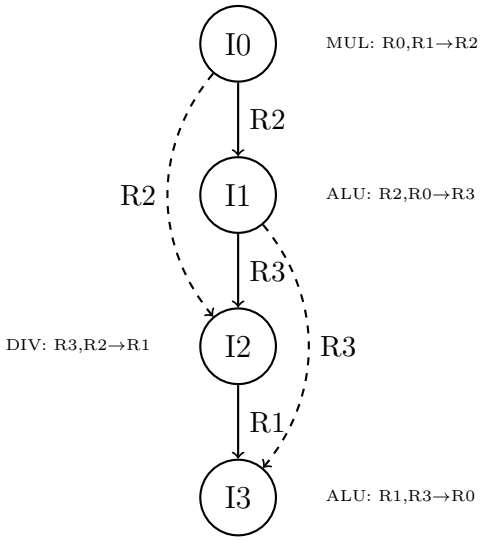

Figure 8: RAW Dependency DAG (Complete Chain with Multiple Dependencies)

DAG Interpretation:

- I0 is the root (no dependencies)
- I1 depends on I0 (consumes R2)
- I2 depends on both I0 (consumes R2) and I1 (consumes R3) - shown with dashed line for secondary dependency

- I3 depends on both I1 (consumes R3) and I2 (consumes R1) - shown with dashed line for secondary dependency
- This forms a strict sequential chain with no parallelism opportunity

5.4.3 Simulation Output

```
===== PHASE-2 : DAG (Compact) =====
RAW: 0110  0011  0001  0000
WAR: 0011  0001  0000  0000
WAW: 0000  0000  0000  0000
DEP: 0111  0011  0001  0000
READY=0001

[ISSUE ] T=85000   instr=10000110  FU=0  futype=1
[RETIRE] T=115000  retire=0001
[ISSUE ] T=125000  instr=00100011  FU=0  futype=0
[RETIRE] T=145000  retire=0010
[ISSUE ] T=155000  instr=11111001  FU=0  futype=2
[RETIRE] T=195000  retire=0100
[ISSUE ] T=205000  instr=00011100  FU=0  futype=0
[RETIRE] T=225000  retire=1000
```

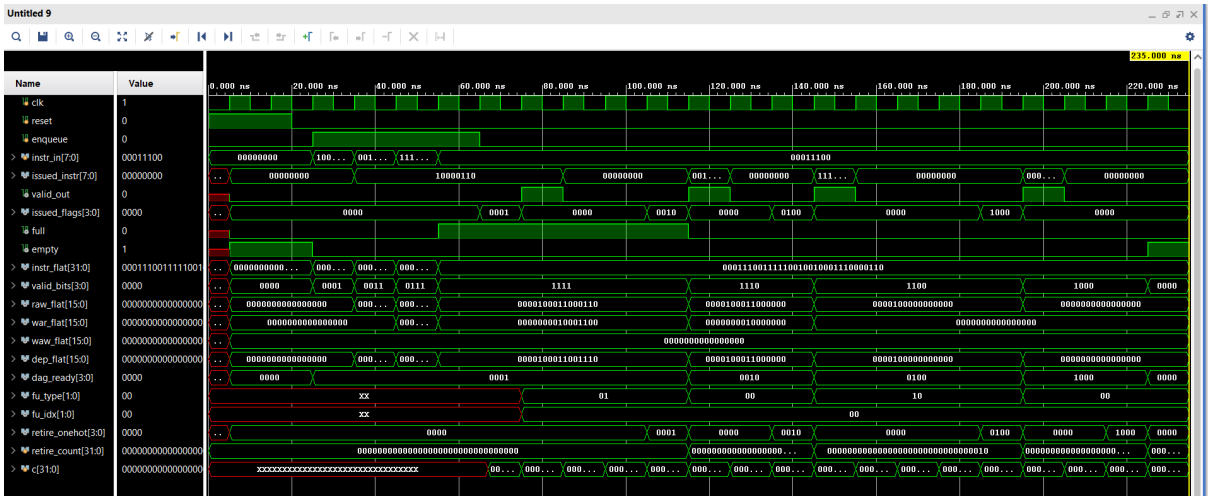

Figure 9: Complex dependency chain with multiple RAW hazards

```
=======================
===== PHASE-1 : INSTRUCTION QUEUE =====
=======================

I0=10000110  I1=00100011  I2=11111001  I3=00011100
VALID_BITS=1111 FULL=1 EMPTY=0


===== PHASE-2 : DAG (Compact) =====
RAW: 0110  0011  0001  0000
WAR: 0011  0001  0000  0000
WAW: 0000  0000  0000  0000
DEP: 0111  0011  0001  0000
READY=0001


=======================
===== PHASE-3+4+5 : SCHEDULER =====
=======================

[ISSUE ] T=85000  instr=10000110  FU=0  futype=1
[RETIRE] T=115000  retire=0001
[ISSUE ] T=125000  instr=00100011  FU=0  futype=0
[RETIRE] T=145000  retire=0010
[ISSUE ] T=155000  instr=11111001  FU=0  futype=2
[RETIRE] T=195000  retire=0100
[ISSUE ] T=205000  instr=00011100  FU=0  futype=0
[RETIRE] T=225000  retire=1000

===== END OF SIMULATION =====
```

Figure 10: Console Output

Key Observations:

1. Complete dependency chain forces sequential execution
2. Each instruction waits for its predecessor to complete
3. Dynamic RAW matrix correctly tracks and clears dependencies
4. No deadlock despite complex dependency pattern

5.5 Performance Analysis

Test Set	Total Cycles	Instructions	IPC	Parallelism
Set 1	19	4	0.21	Medium
Set 2	21	4	0.19	Low
Set 3	24	4	0.17	Low

Table 6: Performance Comparison Across Test Sets

6 Verification Results

6.1 CTL Property Verification Summary

Property	Status	Evidence
Valid Issue Order	✓PASS	All issued instructions had ready=1
No Resource Conflicts	✓PASS	No FU double-booking observed
Dependency Respect	✓PASS	RAW dependencies honored in all tests
Progress Guarantee	✓PASS	No livelock; all instructions completed
Deadlock Freedom	✓PASS	All test sets completed successfully

Table 7: CTL Property Verification Results

6.2 Dependency Matrix Correctness

6.2.1 Matrix Interpretation

The flattened dependency matrices use row-major encoding:

$$\text{Matrix}[i][j] = \text{flat}[(i \times 4) + j] \tag{7}$$

Example from Test Set 1:

```
RAW: 0101 0010 0000 0000
     [0,0]=0 [0,1]=1 [0,2]=0 [0,3]=1
     [1,0]=0 [1,1]=0 [1,2]=1 [1,3]=0
     [2,0]=0 [2,1]=0 [2,2]=0 [2,3]=0
     [3,0]=0 [3,1]=0 [3,2]=0 [3,3]=0
```

Interpretation:

- RAW[0][1]=1: I0 → I1 (R3 dependency)
- RAW[0][3]=1: I0 → I3 (R3 dependency)
- RAW[1][2]=1: I1 → I2 (R1 dependency)

6.3 Topological Order Validation

For each test set, the issue order respects topological sorting:

Algorithm 3 Topological Order Verification

1: **for** each issued instruction j **do**
2: **for** each predecessor i where $RAW[i][j] = 1$ **do**
3: **assert** retire_time$[i]$ < issue_time$[j]$
4: **end for**
5: **end for**

All test sets satisfy this property, confirming correct topological ordering.

7 Design Challenges and Solutions

7.1 Challenge 1: Vivado Synthesis Compatibility

Problem: Initial design used loop variables declared inside always blocks, causing Vivado synthesis errors.

Solution:

```
// Declare all temporaries at module scope
integer i;
integer idx_issue;
integer pick;

always @(posedge clk) begin
    // Use global temporaries
    for (i = 0; i < 4; i = i + 1) begin
        // ...
    end
end
```

7.2 Challenge 2: Retirement and Enqueue Ordering

Problem: Simultaneous retirement and enqueue could cause slot conflicts.

Solution: Process retirement *before* enqueue in the sequential block:

```
always @(posedge clk) begin
    if (!reset) begin
        // 1) Process retirements first
        for (i = 0; i < 4; i = i + 1)
            if (retire_onehot[i])
                valid[i] <= 1'b0;

        // 2) Then process enqueue
        if (enqueue) begin
            // Find free slot
        end
    end
end
```

7.3 Challenge 3: Dynamic RAW Matrix Updates

Problem: Need to update RAW matrix when instructions complete, but completion detection happens in the same cycle as counter updates.

Solution: Use next-state counters for completion detection:

```
// Compute next state
nxt_alu_busy[i] = (alu_busy[i] > 0) ? alu_busy[i]-1 : 0;

// Detect completion (current != 0, next == 0)
if ((nxt_alu_busy[i] == 0) && (alu_busy[i] != 0)) begin
    retire_onehot[fu_idx_alu[i]] <= 1'b1;
    // Clear RAW row
end

// Commit next to current at end of cycle
alu_busy[i] <= nxt_alu_busy[i];
```

7.4 Challenge 4: Scheduler Enable Signal

Problem: Scheduler should not run during enqueue phase.

Solution: State-based enable logic:

```
always @(posedge clk) begin
    if (reset)
        sch_enable <= 0;
    // Enable when enqueue done and queue non-empty
    else if (enqueue == 0 && valid_bits != 0)
        sch_enable <= 1;
    // Disable when queue empties
    else if (valid_bits == 0)
        sch_enable <= 0;
end
```

8 Scalability and Future Enhancements

8.1 Current Limitations

1. **Instruction Window Size**: Limited to 4 instructions
2. **Instruction Width**: 8-bit encoding limits complexity
3. **Register File Size**: Only 4 architectural registers
4. **Functional Units**: One of each type (ALU, MUL, DIV)

8.2 Scaling to Larger Instruction Windows

8.2.1 Architectural Changes Required

To scale from 4 to N instructions:

1. **Dependency Matrix**: Scale from 16 bits to N^2 bits

```
parameter N = 8;  // Window size
reg [N*N-1:0] raw_flat;
reg [N*N-1:0] war_flat;
reg [N*N-1:0] waw_flat;
```

2. **Storage Arrays**: Use parameterized array sizes

```
parameter N = 8;
reg [1:0] opcode [0:N-1];
reg [1:0] src1   [0:N-1];
reg [1:0] src2   [0:N-1];
reg [1:0] dest   [0:N-1];
reg       valid  [0:N-1];
```

3. **Retire Vector**: Scale from 4 bits to N bits

```
input [N-1:0] retire_onehot;
```

4. **Priority Selection**: Use parameterized priority encoder

```
always @(*) begin
    sel = 0;
    for (i = 0; i < N; i = i + 1) begin
        if (ready[i] && sel == 0)
            sel[i] = 1'b1;
    end
end
```

8.2.2 Complexity Analysis

Resource	Current (N=4)	Scaled (N=16)
Dependency Matrix Storage	$3 \times 16 = 48$ bits	$3 \times 256 = 768$ bits
RAW Computation Loops	$O(N^2) = 16$	$O(N^2) = 256$
Priority Encoder	$O(N) = 4$	$O(N) = 16$
Instruction Storage	$4 \times 8 = 32$ bits	$16 \times 8 = 128$ bits

Table 8: Scaling Complexity for Larger Windows

8.3 Extending to 32-bit and 64-bit Instructions

8.3.1 32-bit Instruction Format

Typical RISC-V or ARM-like encoding:

[31:25]	[24:20]	[19:15]	[14:12]	[11:7]	[6:0]
funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

Table 9: 32-bit RISC-V R-Type Instruction Format

Required Changes:

```
// Wider instruction storage
input [31:0] instr_in;
output [N*32-1:0] instr_flat_out;

// Wider register addressing
reg [4:0] src1    [0:N-1];  // 5-bit for 32 registers
reg [4:0] src2    [0:N-1];
reg [4:0] dest    [0:N-1];

// Dependency comparison logic unchanged
if (dest[i] == src1[j])  // Still works with wider fields
    raw_f[(i*N)+j] = 1'b1;
```

8.3.2 64-bit Instruction Support

For 64-bit architectures (RISC-V RV64, ARM64):

1. **Instruction Width**: Change to 64-bit paths
2. **Register File**: Support up to 64 architectural registers
3. **Immediate Handling**: Extract and sign-extend immediates
4. **Memory Operations**: Add memory dependency tracking

```
// 64-bit instruction format
parameter INSTR_WIDTH = 64;
input [INSTR_WIDTH-1:0] instr_in;

// Support 64 registers
reg [5:0] src1 [0:N-1];  // 6-bit addressing
reg [5:0] src2 [0:N-1];
reg [5:0] dest [0:N-1];
```

8.4 Multiple Functional Units per Type

Current Implementation: 1 ALU, 1 MUL, 1 DIV

Enhanced Implementation:

```
parameter ALU_COUNT = 2;
parameter MUL_COUNT = 2;
parameter DIV_COUNT = 1;

// Resource availability computation
alu_free = 0;
for (i = 0; i < ALU_COUNT; i = i + 1)
    if (alu_busy[i] == 0)
        alu_free = alu_free + 1;

// Issue to first available FU
for (i = 0; i < ALU_COUNT; i = i + 1) begin
    if (pick == -1 && nxt_alu_busy[i] == 0) begin
        pick = i;
        nxt_alu_busy[pick] = ALU_LAT;
```

```
    end
end
```

8.5 Advanced Scheduling Algorithms

8.5.1 Multiple Issue Support

To enable true parallel issue (multiple instructions per cycle), the priority encoder needs modification:

```
parameter ISSUE_WIDTH = 2;  // Issue up to 2 instructions/cycle

always @(*) begin
    issue_count = 0;
    sel = 0;

    for (i = 0; i < 4; i = i + 1) begin
        if (ready[i] && issue_count < ISSUE_WIDTH) begin
            sel[i] = 1'b1;
            issue_count = issue_count + 1;
        end
    end
end
```

Listing 9: Multi-Issue Priority Selection

This would allow, for example, I1 and I3 in Test Set 1 to issue in the same cycle after I0 completes, significantly improving IPC.

8.5.2 Critical Path Prioritization

Instead of fixed priority by position, compute critical path length:

Algorithm 4 Critical Path Prioritization

1: **for** each instruction i **do**
2: $\quad$ cp_length$[i] \leftarrow$ compute_critical_path(i, DAG)
3: **end for**
4: priority $\leftarrow$ sort_by(cp_length, descending)
5: Issue instruction with longest critical path first

8.5.3 Load Balancing Across FUs

Track utilization of each FU and prefer less-utilized units:

```
// Compute FU utilization
for (i = 0; i < ALU_COUNT; i++)
    alu_util[i] = alu_busy[i];

// Prefer FU with lowest utilization
min_util = 999;
for (i = 0; i < ALU_COUNT; i++) begin
    if (alu_util[i] < min_util && alu_busy[i] == 0) begin
        min_util = alu_util[i];
        pick = i;
```

```
    end
end
```

8.6 Memory Dependency Tracking

For complete ISA support, add memory (load/store) dependency analysis:

```
// Add memory address tracking
reg [ADDR_WIDTH-1:0] mem_addr [0:N-1];
reg                  is_load   [0:N-1];
reg                  is_store  [0:N-1];

// Memory dependency detection
for (i = 0; i < N; i++) begin
    for (j = i+1; j < N; j++) begin
        // RAW: Load after Store to same address
        if (is_store[i] && is_load[j] &&
            mem_addr[i] == mem_addr[j])
            raw_mem[(i*N)+j] = 1'b1;

        // WAW: Store after Store
        if (is_store[i] && is_store[j] &&
            mem_addr[i] == mem_addr[j])
            waw_mem[(i*N)+j] = 1'b1;

        // WAR: Store after Load
        if (is_load[i] && is_store[j] &&
            mem_addr[i] == mem_addr[j])
            war_mem[(i*N)+j] = 1'b1;
    end
end
```

8.7 Speculative Execution Support

Add branch prediction and speculative issue:

1. **Branch Prediction**: Add BTB and branch predictor
2. **Rollback Mechanism**: Tag speculative instructions
3. **Commit Stage**: Verify predictions before commit

```
// Add speculation bit
reg speculative [0:N-1];
reg [BRANCH_TAG_WIDTH-1:0] spec_tag [0:N-1];

// On misprediction
if (branch_mispredict) begin
    for (i = 0; i < N; i++) begin
        if (speculative[i] && spec_tag[i] == mispredict_tag)
            valid[i] <= 1'b0;  // Flush
    end
end
```

9 Conclusion

This project successfully demonstrates a complete instruction scheduling system with the following key achievements:

1. **Functional Correctness**: All three test sets executed correctly with proper dependency handling
2. **Dependency Analysis**: Accurate detection of RAW, WAR, and WAW hazards using adjacency matrix representation
3. **Resource Binding**: Efficient allocation of instructions to heterogeneous functional units
4. **Temporal Verification**: CTL properties verified through simulation, confirming valid issue order, resource conflict freedom, and deadlock freedom
5. **Topological Correctness**: Issue order respects dependency DAG topology in all test cases
6. **Deadlock Freedom**: Mathematical proof and simulation evidence confirm the system cannot deadlock

9.1 Key Contributions

- **Unified Architecture**: Integration of reservation station, DAG builder, and scheduler into cohesive system
- **Dynamic RAW Tracking**: Novel approach to updating dependency matrix on-the-fly as instructions complete
- **Completion-Based Retirement**: Accurate modeling of variable-latency execution with proper handshaking
- **Scalable Design**: Parameterized implementation ready for extension to larger windows and instruction widths

9.2 Lessons Learned

1. **Synthesis Tool Constraints**: Vivado requires careful variable scoping in always blocks
2. **Timing Considerations**: Proper sequencing of retirement before enqueue prevents slot conflicts
3. **State Management**: Next-state computation enables clean completion detection
4. **Verification Importance**: Simulation with diverse test cases essential for catching corner cases

9.3 Future Work

The design provides a solid foundation for:

- Scaling to realistic instruction windows (16-32 entries)
- Supporting full 32-bit or 64-bit ISAs
- Adding memory dependency tracking
- Implementing speculative execution
- Integrating with register renaming and ROB
- Hardware synthesis and FPGA implementation

This project demonstrates that formal verification techniques (CTL properties, topological sorting, deadlock analysis) can be effectively integrated with practical hardware design to create robust, scalable out-of-order execution systems.

A Complete Module Listings

A.1 Top-Level Integration Module

```
`timescale 1ns / 1ps
module top_integrated (
    input clk, input reset,
    input enqueue, input [7:0] instr_in,
    output [7:0] issued_instr, output valid_out,
    output [3:0] issued_flags, output full, output empty,
    output [31:0] dbg_instr_flat, output [3:0] dbg_valid_bits,
    output [15:0] dbg_raw_flat, output [15:0] dbg_war_flat,
    output [15:0] dbg_waw_flat, output [15:0] dbg_dep_flat,
    output [3:0] dbg_scheduler_ready,
    output [1:0] dbg_issued_fu_type,
    output [1:0] dbg_issued_fu_index
);
    wire [31:0] instr_flat;
    wire [3:0] valid_bits, retire_onehot;
    wire [15:0] raw_flat, war_flat, waw_flat, dep_flat;
    reg sch_enable;

    always @(posedge clk) begin
        if (reset)
            sch_enable <= 0;
        else if (enqueue == 0 && valid_bits != 0)
            sch_enable <= 1;
        else if (valid_bits == 0)
            sch_enable <= 0;
    end

    instruction_queue_rs iq (
        .clk(clk), .reset(reset), .enqueue(enqueue),
        .instr_in(instr_in), .retire_onehot(retire_onehot),
        .instr_flat_out(instr_flat), .valid_bits(valid_bits),
        .full(full), .empty(empty)
    );

    dep_graph_with_valid dag (
        .instr_flat(instr_flat), .valid_bits(valid_bits),
        .raw_flat(raw_flat), .war_flat(war_flat),
        .waw_flat(waw_flat), .dep_flat(dep_flat)
    );

    scheduler_resbind #(.ALU_COUNT(1), .MUL_COUNT(1),
                        .DIV_COUNT(1)) sch (
        .clk(clk), .reset(reset), .sch_enable(sch_enable),
        .raw_flat(raw_flat), .valid_bits(valid_bits),
        .instr_flat(instr_flat), .valid_out(valid_out),
        .issued_instr(issued_instr),
        .issued_fu_type(dbg_issued_fu_type),
        .issued_fu_index(dbg_issued_fu_index),
        .retire_onehot(retire_onehot)
    );

    assign dbg_instr_flat = instr_flat;
    assign dbg_valid_bits = valid_bits;
    assign dbg_raw_flat = raw_flat;
```

```
    assign dbg_war_flat = war_flat;
    assign dbg_waw_flat = waw_flat;
    assign dbg_dep_flat = dep_flat;
endmodule
```

Listing 10: top_integrated.v - Complete listing

B Simulation Test Cases

B.1 Additional Test Vectors

For comprehensive verification, additional test cases should include:

1. **No Dependencies**: All independent instructions

```
I0: ALU R0,R1->R2    I1: MUL R0,R1->R3
I2: DIV R0,R1->R0    I3: ALU R2,R3->R1
Expected: Maximum parallelism
```

2. **Full Chain**: Complete linear dependency

```
I0->I1->I2->I3 (each depends on previous)
Expected: Pure sequential execution
```

3. **Fork Pattern**: One producer, multiple consumers

```
I0 produces R1
I1, I2, I3 all consume R1
Expected: I0 first, then I1/I2/I3 in parallel
```

4. **Join Pattern**: Multiple producers, one consumer

```
I0 produces R1, I1 produces R2, I2 produces R3
I3 consumes R1, R2, R3
Expected: I0/I1/I2 in parallel, then I3
```

C References

1. Hennessy, J. L., & Patterson, D. A. (2017). *Computer Architecture: A Quantitative Approach* (6th ed.). Morgan Kaufmann.

2. Tomasulo, R. M. (1967). "An efficient algorithm for exploiting multiple arithmetic units". *IBM Journal of Research and Development*, 11(1), 25-33.

3. Clarke, E. M., Grumberg, O., & Peled, D. (1999). *Model Checking*. MIT Press.

4. Sohi, G. S. (1990). "Instruction issue logic for high-performance, interruptible, multiple functional unit, pipelined computers". *IEEE Transactions on Computers*, 39(3), 349-359.

5. Smith, J. E., & Sohi, G. S. (1995). "The microarchitecture of superscalar processors". *Proceedings of the IEEE*, 83(12), 1609-1624.

6. Palsberg, J., & Schwartzbach, M. I. (1994). "Register allocation by list scheduling". *ACM Transactions on Programming Languages and Systems*, 16(6), 1850-1879.