

VLIW Processor Implementation

Introduction

Very Long Instruction Word (VLIW) processors are a type of architecture that executes multiple operations in a single instruction word. Unlike superscalar processors, which dynamically schedule instructions at runtime, VLIW processors rely on the compiler to statically schedule instructions into long instruction words. This approach reduces hardware complexity but places a greater burden on the compiler to detect and handle hazards.

This report discusses the implementation of a VLIW processor simulator that handles structural and data hazards (RAW and WAW) by inserting NOPs (No Operations). Control hazards are not addressed in this implementation. The simulator processes an assembly program, schedules instructions, and demonstrates the execution timeline.

VLIW Processor Overview

A VLIW processor executes multiple operations in parallel by packing them into a single instruction word. Each operation is assigned to a specific functional unit (e.g., integer adder, floating-point multiplier). The compiler is responsible for ensuring that there are no dependencies or resource conflicts between operations in the same instruction word.

Functional Units

The processor has the following functional units:

- **Integer Adder (IADD/ISUB):** Handles integer addition and subtraction (6 clock cycles).
- **Integer Multiplier (IMUL):** Handles integer multiplication (12 clock cycles).
- **Integer Divider (IDIV):** Handles integer division (24 clock cycles).
- **Floating-Point Adder (FADD/FSUB):** Handles floating-point addition and subtraction (18 clock cycles).

- **Floating-Point Multiplier (FMUL):** Handles floating-point multiplication (30 clock cycles).
- **Floating-Point Divider (FDIV):** Handles floating-point division (60 clock cycles).
- **Logical Units:** Handle logical operations (AND, OR, XOR, NOT) in 1 clock cycle.
- **Memory Unit (LD/ST):** Handles load and store operations in 1 clock cycle.
- **Instruction Fetch (IF):** Fetches instructions from memory (1 clock cycle).
- **Instruction Decode (ID):** Decodes instructions (1 clock cycle).
- **Write Back (WB):** Writes results back to registers (1 clock cycle).

Hazards

Hazards in a processor occur when instructions cannot execute in the expected order due to dependencies or resource conflicts. The three types of hazards are:

- **Structural Hazards:** Occur when multiple instructions require the same functional unit simultaneously.
- **Data Hazards:** Occur when an instruction depends on the result of a previous instruction that has not yet completed.
- **Control Hazards:** Occur due to branch instructions, which can change the program counter and disrupt the instruction pipeline.

This implementation handles structural and data hazards (RAW and WAW) by inserting NOPs. Control hazards are not addressed. Additionally, WAR (Write-After-Read) hazards are not explicitly handled in this implementation due to the following assumption:

- **Assumption for WAR Hazards:**
 - A lock is placed on source registers ('src') during the Instruction Decode (ID) stage and is released immediately after the ID stage. This means that once an instruction reads a register during the ID stage, the register is no longer locked, and subsequent instructions can write to it without causing a WAR hazard.
 - For destination registers ('dst'), a lock is placed during the ID stage and is only released after the Write Back (WB) stage. This ensures that no other instruction can write to the same destination register until the current instruction completes its write operation, preventing WAW hazards.

Instruction Set

The processor supports the following instructions, each mapped to specific functional units and pipeline stages. The table below provides a detailed breakdown:

Instruction	Description	Functional Units and Pipeline Stages
IADD Rdst, Rsrc1, Rsrc2	Integer Addition	IF (1 cycle), ID (1 cycle), ADD (6 cycles), WB (1 cycle)
ISUB Rdst, Rsrc1, Rsrc2	Integer Subtraction	IF (1 cycle), ID (1 cycle), ADD (6 cycles), WB (1 cycle)
IMUL Rdst, Rsrc1, Rsrc2	Integer Multiplication	IF (1 cycle), ID (1 cycle), MUL (12 cycles), WB (1 cycle)
IDIV Rdst, Rsrc1, Rsrc2	Integer Division	IF (1 cycle), ID (1 cycle), DIV (24 cycles), WB (1 cycle)
FADD Rdst, Rsrc1, Rsrc2	Floating-Point Addition	IF (1 cycle), ID (1 cycle), FADD (18 cycles), WB (1 cycle)
FSUB Rdst, Rsrc1, Rsrc2	Floating-Point Subtraction	IF (1 cycle), ID (1 cycle), FADD (18 cycles), WB (1 cycle)
FMUL Rdst, Rsrc1, Rsrc2	Floating-Point Multiplication	IF (1 cycle), ID (1 cycle), FMUL (30 cycles), WB (1 cycle)
FDIV Rdst, Rsrc1, Rsrc2	Floating-Point Division	IF (1 cycle), ID (1 cycle), FDIV (60 cycles), WB (1 cycle)
AND Rdst, Rsrc1, Rsrc2	Logical AND	IF (1 cycle), ID (1 cycle), AND (1 cycle), WB (1 cycle)
OR Rdst, Rsrc1, Rsrc2	Logical OR	IF (1 cycle), ID (1 cycle), OR (1 cycle), WB (1 cycle)
XOR Rdst, Rsrc1, Rsrc2	Logical XOR	IF (1 cycle), ID (1 cycle), XOR (1 cycle), WB (1 cycle)
NOT Rdst, Rsrc1	Logical NOT	IF (1 cycle), ID (1 cycle), NOT (1 cycle), WB (1 cycle)
LD Rdst, [Mem]	Load from Memory	IF (1 cycle), ID (1 cycle), MEM (1 cycle), WB (1 cycle)
ST Rsrc, [Mem]	Store to Memory	IF (1 cycle), ID (1 cycle), MEM (1 cycle), WB (1 cycle)
NOP	No Operation	IF (1 cycle)

Code Explanation

The simulator is implemented in Python and consists of the following components:

- **Pipeline Stages:** Each instruction type has a predefined set of pipeline stages (e.g., IF, ID, ADD, WB) with associated latencies.
- **Hazard Handling:** The simulator detects structural and data hazards and inserts NOPs to resolve them.

- **Instruction Parsing:** The assembly code is parsed into a list of instructions, each with its operation, source/destination registers, and pipeline stages.
- **Execution Timeline:** The simulator generates a timeline showing the execution of each instruction across clock cycles.

Key Functions

- `parse_assembly`: Parses the assembly code into a list of instructions.
- `insert_nops_for_structural_hazards`: Inserts NOPs to resolve structural hazards.
- `handle_data_dependencies`: Inserts NOPs to resolve RAW and WAW hazards.
- `display_clock_cycle_execution`: Displays the execution timeline for each instruction.
- `display_functional_units`: Displays the usage of functional units across clock cycles. It's used to generate instruction packets, which are groups of instructions that can be executed in parallel by the VLIW processor. These packets are sent to the VLIW processor for execution.

Input and Output

Input Assembly Code

The input assembly program performs basic arithmetic operations on registers and memory locations. The program is as follows:

```
LD R1, [1000]
LD R2, [2000]
IADD R3, R1, R2
ISUB R4, R1, R2
IMUL R5, R1, R2
IDIV R6, R1, R2
ST R3, [3000]
ST R4, [4000]
ST R5, [5000]
ST R6, [6000]
```

This program loads values from memory into registers, performs addition, subtraction, multiplication, and division, and stores the results back into memory.

Output Explanation

The simulator outputs the updated assembly code with NOPs inserted to handle hazards, the total clock cycles, and the execution timeline.

Updated Code with NOPs

The updated code includes NOPs to resolve structural and data hazards:

```
LD R1, [1000]
LD R2, [2000]
NOP
IADD R3, R1, R2
NOP
NOP
NOP
NOP
NOP
ISUB R4, R1, R2
IMUL R5, R1, R2
IDIV R6, R1, R2
NOP
NOP
NOP
ST R3, [3000]
ST R4, [4000]
NOP
NOP
NOP
NOP
NOP
NOP
ST R5, [5000]
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
ST R6, [6000]
```

Total Clock Cycles

The program takes **40 clock cycles** to execute.

Execution Timeline

The execution timeline shows the pipeline stages for each instruction across clock cycles. For example:

- LD R1, [1000] executes in cycles 1-4.
- IADD R3, R1, R2 executes in cycles 4-12.
- IDIV R6, R1, R2 executes in cycles 12-37.

Functional Units Usage

The functional units timeline shows how each functional unit is utilized across clock cycles. For example:

- The ADD unit is used by IADD and ISUB.
- The MUL unit is used by IMUL.
- The DIV unit is used by IDIV.
- The MEM unit is used by LD and ST.

Conclusion

This report demonstrated the implementation of a VLIW processor simulator that handles structural and data hazards by inserting NOPs. The simulator processes an assembly program, resolves hazards, and generates an execution timeline. The total clock cycles and functional unit usage are also displayed. Future work could include handling control hazards and optimizing the scheduling algorithm.