Ayush Tibrewal & Anton Melnychuk CPSC 420

Lab 2: Pipelined PARCv2 Processor

Abstract

In this lab, we extend the pipelined PARCv2 processor from the previous lab into a two-wide superscalar processor. The project is divided into two main parts: first, implementing dual instruction fetch with single-issue logic, and second, enabling full dual-issue execution with appropriate control, steering, and scoreboard logic to manage pipeline constraints and data hazards. The updated processor fetches two instructions per cycle and issues them to corresponding functional units A and B through a combined decode-issue stage. Unit A remains fully functional as in the previous implementation, while unit B supports only simple ALU instructions. In the absence of hazards or dependencies, which we handle since the register file will not, the processor can execute two instructions per cycle in the best case. For some tests and benchmarks, the speedups were present but not significant. For the masked filter and vvadd benchmarks, the improvement was a nearly 25% speedup, showing significant IPC improvement over single-issue pv2long implementations (Table 1). These results underscore how increased parallelism and careful hazard management contribute to throughput gains, while also illustrating tradeoffs with complexity and limitations due to dependency stalls and structural hazards in more complex programs.

Design

Part 1

This section involved implementing basic steering logic for the PARCv2 processor capable of fetching two instructions per cycle, while keeping single-issue design to the pipeline A. We introduced a steering mechanism, where the processor alternates each cycle between issuing the first and second fetched instructions—unless a stall is required due to operand dependencies. Compared to the previously implemented bypassed processor, this design required duplicating much of the stall and bypass logic to distinguish between the first and second instruction in flight. Additionally, logic had to be added to correctly select which set of bypass signals to apply on each cycle.

Scoreboarding

```
// Scoreboard Register Layout (6 bits total)
//   Bit 6     : In progress (memory/muldiv not done yet)
//   Bit 5     : Pipeline A/B (default: 0)
//   Bits 4:0  : Pipeline stage bits:
//               Bit 0 - Stage 0: X0 (Instruction capture/initialization)
//               Bit 1 - Stage 1: First pipeline update
//               Bit 2 - Stage 2: Second pipeline update
//               Bit 3 - Stage 3: Third pipeline update (with additional checks for mul/div)
//               Bit 4 - Stage 4: Write Back
reg [6:0] scoreboard [31:0];
```

To support simple bypassing logic while handling data hazards, we implemented a scoreboard to track register availability throughout the pipeline. The scoreboard is a 32-entry array (one per register), with each 7-bit entry encoding the pipeline stage occupancy and source pipeline. Bit 6 indicates if a value is in progress (in the case of memory loads and muldiv operations), bit 5 indicates the issuing pipeline (0 for A, 1 for B), while bits 0–4 correspond to the 5 pipeline stages (X0–W). On each cycle, the scoreboard is updated if no stall is present in a stage, effectively shifting dependency information forward through the pipeline. When a register is written by an instruction in X0 and rfA_wen_Dhl is asserted, the corresponding scoreboard entry is initialized to indicate that it is in use. In Part 1, we allow bypass logic for each source operand (rs, rt) checks the scoreboard to find which pipeline stage has the most recent value.

Part 2

For the second part, all of the steering logic and scoreboarding had to be extended to work for both pipelines A and B. This meant that if instructions did not have data dependencies, they are now allowed to pass through the pipelines in parallel. However, since pipeline B can only perform basic ALU instructions, we had to improve the steering logic to detect hazards and correctly steer instructions based on whether or not they used the ALU. Relatedly, the scoreboarding logic had to be duplicated to allow for bypassing from both pipeline A and pipeline B, using bit 5 as mentioned above.

The actual logic did not differ significantly from the given directions. We thought of the new logic in two parts, steering and issue. Firstly, we implemented the logic for which instruction needed to be steered into which pipeline, based on what had already been issued as well as whether the instructions use the ALU or not. The issue part of the logic determines when

an instruction in the A pipeline can be issued to X0, and similarly for B. This took a significant portion of our time as there were several different cases to consider. Notably, the D stage has to stall until both instructions have been issued or it is squashed from a jump/branch.

Testing Methodology

We used the testing suite provided, along with custom targeted tests for RAW, WAW, and back-to-back non-ALU instructions. The primary debugging tool was just the close inspection of waveform outputs in GTKWave to analyze unexpected behaviors and confirm signal-level correctness. The RAW test focused on memory ordering, ensuring proper enforcement of load-store constraints—specifically, that a store instruction does not bypass a prior load to the same address. For WAW hazards, we verified that register write-backs preserve the correct value from the final write in the instruction sequence. To access pipeline efficiency under ideal conditions, we developed a custom pv2-dualissue test. The results demonstrate a best-case scenario where nearly all instructions can be issued in parallel, resulting in a significant performance gain—from 257 to 134 cycles, and from 0.984436 to 1.888060 IPC—effectively nearly doubling throughput. This shows that the primary bottleneck for other tests in improving performance comes from a lack of concurrent parallelizable instructions.

Evaluations

	Bin Search		Complex Mult		Masked Filter		Vvadd	
	Cycles	IPC	Cycles	IPC	Cycles	IPC	Cycles	IPC
Stall	3382	0.378179	16718	0.111497	16174	0.278162	478	0.951883
Byp	1664	0.768630	15262	0.122133	13437	0.334822	473	0.961945
Long	1664	0.768630	2512	0.742038	5658	0.795157	473	0.961945
Dual	1664	0.768630	2512	0.742038	5658	0.795157	473	0.961945
Ssc	1551	0.823340	2357	0.790412	4780	0.941423	377	1.204244

Table 1: Performance Comparison Across Pipeline Strategies for Various Benchmarks

Part 1 did not upgrade the pipeline, since we keep it single issue, although 2 instructions are passed, 1 cycle is needed for each of them to be dispatched.

Discussion

We see that all benchmarks saw improved performance. In vvadd and masked filter, a brief look at the instructions showed that many instructions could be issued in parallel, which gave an 18% and 25% speedup, respectively. However, in bin search and complex mult, there was not a significant difference between the single issue and double issue processors. Ultimately, we feel that these speedups may not be worth the additional complexity introduced unless programs are rewritten to maximize their ability to use both pipelines concurrently.

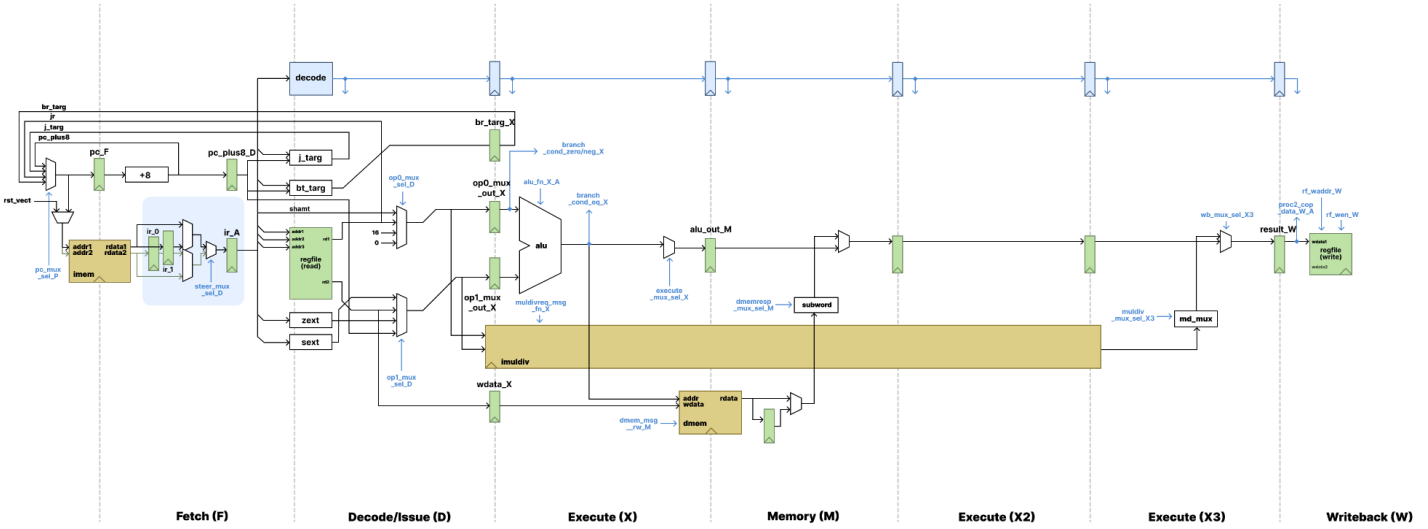

Figure 1: Datapath of 5-stage PARCv2 2-Wide Superscalar, In-Order Processor, including Scoreboard

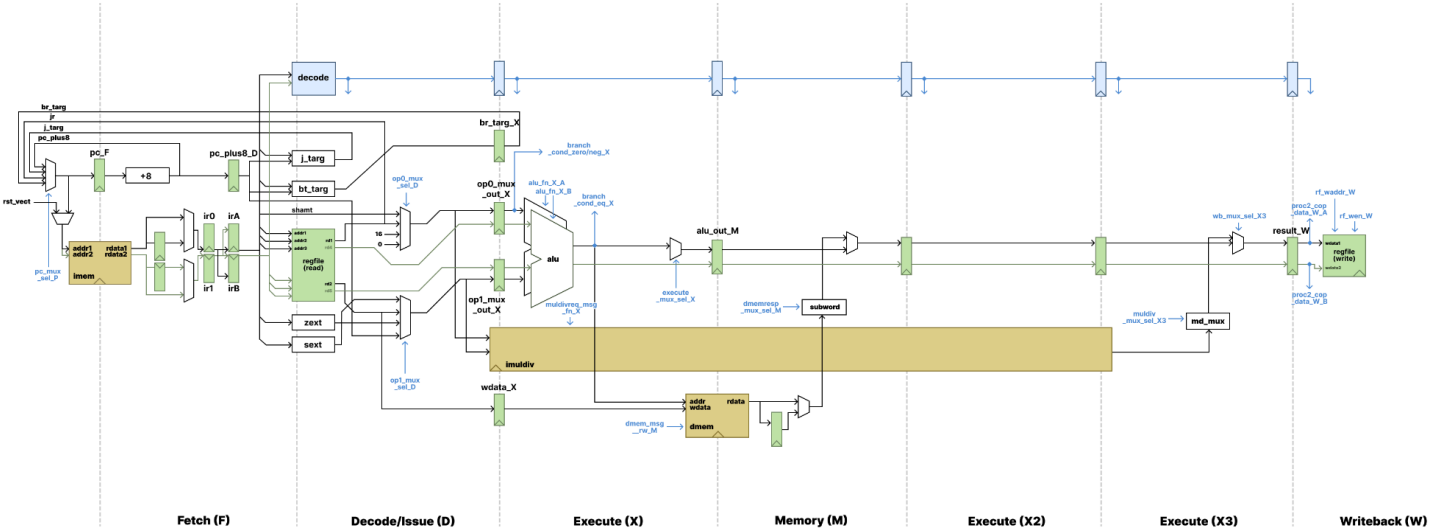

Figure 2: Datapath of 5-stage PARCv2 2-Wide Superscalar, OoO Processor, including Scoreboard