ECE 251: Computer Architecture

Week 12: Exceptions, Interrupts, and Architecture Synthesis

Prof Rob Marano

Spring 2026

1. Chapter 4 Retrospective: Architecture Evolution

- **Single-Cycle**: $CPI = 1.0$, bottlenecked incredibly by the absolute longest instruction format ($T_c \approx 800 - 1000$ps).
- **Multi-Cycle**: FSM driven, faster clock boundaries ($T_c \approx 200$ps), but inflates cycle footprint ($CPI \approx 3 - 5$) operating sequentially.
- **Pipelining**: Overlaps instructions concurrently across D-Flip-Flops to marry the rapid T_c of the Multicycle with the optimal $CPI = 1.0$ limit natively.

2. The Cost of Pipelining: Complexity and Faults

- Pipelining generates extreme throughput at the cost of structural convolution.
- **Data Dependencies**: Mandates massive Forwarding loops to bypass Register caching limits.
- **Control Penalties**: Requires hardware flushing of fetching phases on mispredicted jumps natively.
- Overlapping matrices make stopping instruction tracking incredibly complex when localized sequential constraints fail unexpectedly.

3. Overview: Handling the Unexpected

- Modern processors must systematically handle logic faults or external demands natively without permanently locking up computation.
- **Exceptions**: Synchronous electrical faults generated internally by the datapath executing instructions (e.g., Undefined Opcodes, Math Overflow).
- **Interrupts**: Asynchronous external electrical signals generated outside the datapath execution flow (e.g., Network Packets, I/O Hardware).
- Both conditions force the datapath to immediately halt instruction overlap, backup tracking logic, and jump execution explicitly to an OS handler vector.

4. The Trapping Mechanisms

- **1. Exception Program Counter (EPC)**: A dedicated 32-bit hardware register tasked strictly with capturing the Program Counter memory pointer (PC) of the exact instruction generating the localized exception fault.
- **2. Cause Register**: Records a strict numerical identity classifying exactly *why* the CPU faulted natively, letting the OS run analytical branching routines.
- **3. Vector Handler Routing**: The hardware forces the multiplexer riding the PC tracking pin to dynamically rewire directly to the OS boundary at `0x8000 0180`.

5. SystemVerilog Physics: EPC and Routing

```
// The EPC Latch inside the datapath
always_ff @(posedge clk) begin
  if (Exception_Flag)
    EPC <= PC_Plus4_E - 32'd4; // Capture the failing instruction
end

// The NextPC Multiplexer expansion
always_comb begin
  if (Exception_Flag) NextPC = 32'h8000_0180; // Trap to OS
  else if (PCSrcE)    NextPC = PCBranchE;     // Standard branch
  else                NextPC = PCPlus4F;      // Sequential logic
end
```

6. Pipeline Considerations: Flushing Exceptions

- The biggest challenge is mapping faults correctly across multiple overlapping instructions.
- If an add in the Execute (EX) stage mathematically overflows, the two innocent instructions trailing directly behind it inside the Fetch (IF) and Decode (ID) stages must die instantly.
- The logic strictly utilizes the exact same `StallF` and `FlushD/FlushE` pins engineered for Branch hazards, rewriting the trailing sequence overlapping buffers into inert `NOP` structural states.

7. Complex Exception Theory: Restartable Precise Exceptions

- **Precise Exceptions**: The MIPS pipeline explicitly guarantees that the state of the processor is saved precisely at the faulting instruction natively. All preceding instructions must cleanly commit, and instruction following it are entirely purged.
- **Restartable Datapaths**: Because we physically track execution limits backwards via the `EPC` (P_C - 4), the OS possesses the native mathematical capacity to transparently re-inject the failed program directly into the fetching queues exactly where it stopped.
- **Imprecise Exceptions**: Heavily clustered Out-Of-Order CPU architectures mathematically drop pinpoint recovery limitations, isolating faults to a general "vicinity" and drastically ballooning OS handler complexity constraints natively.

8. Complex Exception Theory: Multiple Fault Constraints

- What natively happens if **two isolated instructions** concurrently crash across uniquely decoupled pipeline stages during the exact same T_c clock window? (e.g., A memory fault physically intersecting an arithmetic overflow).
- **Hardware Prioritization**: The digital architecture intrinsically favors the *oldest* structural instruction actively running (residing mathematically furthest down the pipeline tracks, such as EX/MEM bounds).
- Younger instructions trailing backwards across the IF/ID registers simply haven't formally completed their theoretical lifetimes yet, and thus structurally yield their active interrupts until re-execution.

9. Global Sandbox: pipelined_cpu_exceptions

- All the digital concepts actively synthesized are tracked continuously inside our live pipelined_cpu_exceptions hardware namespace.
- The entire Patterson & Hennessy theoretical framework natively compiles into executable boolean topologies:

```
datapath.sv    # Maps NextPC -> 0x8000_0180 OS boundary
hazard.sv      # Maps asynchronous Exception_Flag -> flush loops
mem.sv         # Bootstraps .org boundary arrays cleanly
controller.sv # Natively tracks Exception bounds
test_prog.asm # Compiles structural exception MIPS sequences
```

10. Debugging Resolutions: 4-Bit ALU Decoupling

- Incorporating complex math vectors natively clashed with native MIPS bounds.
- By natively widening `alucontrol` variables from structural 3-Bit `[2:0]` to deep 4-Bit `[3:0]` limits, we decoupled math logic fundamentally:
- Combinational overlap failures mathematically disappeared because Arithmetic Logic (‘4’b0111‘) permanently diverged dynamically away from Sequential structural footprints actively isolating `MULT` (‘4’b1001‘) and `DIV` (‘4’b1000‘) execution bounds!

11. Debugging Resolutions: Memory Depth & Aliasing

- Exceptions mechanically redirect hardware to absolute vector limit `0x8000 0180`.
- A generic Datapath limiting array depth to parameter $r = 6$ natively forces Index 96 (`0x180`) to roll back via modulus limits dynamically onto integer 32 (`0x080`).
- Expanding system boundaries natively to $r = 8$ establishes a valid OS memory isolation space stretching cleanly to $2^8 = 256$ deep memory frames, granting the OS handler dedicated non-aliasing boundaries natively.

12. Verification Matrix: The 7 Assembly Playbooks

- We fully assert the architecture logic leveraging extensive workload matrices simulating MIPS bounds:
  1. `prog1_simple`: Linear testing ensuring Forwarding pipelines.
  2. `prog2_leaf`: Validating Control branches flushing predictions cleanly.
  3. `prog3_nested`: Nested variables generating Load/Use stalls on stacks.
  4. `prog4_interrupts`: Structurally overriding datapath sequences internally dumping active limits securely into kernel space.

13. Retrospective: The von Neumann Architecture Journey

- **Assembly (ISA)**: Human logic bounds mapped natively to binary combinations.
- **Datapath Construction**: ALU calculation grids mapping mathematically mapped alongside immense sequential registry stores.
- **Pipelining Overlaps**: Expanding Single-Cycle bottlenecks geometrically overlapping hardware to maximize processor frequency constraints whilst preserving idealized $CPI = 1.0$.
- **SystemVerilog HDL Control**: SystemVerilog formally proves and describes these complex sequential networks to synthesize logic networks matching photolithography silicon fabrication models reliably.

14. Simulating the Processor (Part 1)

- To functionally test the SystemVerilog components, we must generate a binary executable payload for the CPU's memory arrays.
- **Step 1: Run the Automated Makefile** – The `Makefile` automatically assembles the MIPS instructions, synthesizes the SystemVerilog netlist, and dynamically executes the simulation natively:

```
$ make clean all ASM=test_prog
```

- The testbench internally hooks the binary payload bounds directly into active memory arrays cleanly.
- The testbench additionally generates a digital execution wave file locally: `tb_computer.vcd`.

15. Simulating the Processor (Part 2: Text Trace)

- **Step 2: Trace Cycle-by-Cycle Execution** – The simulation quietly diverts all console outputs directly into a tracked diagnostic file natively!

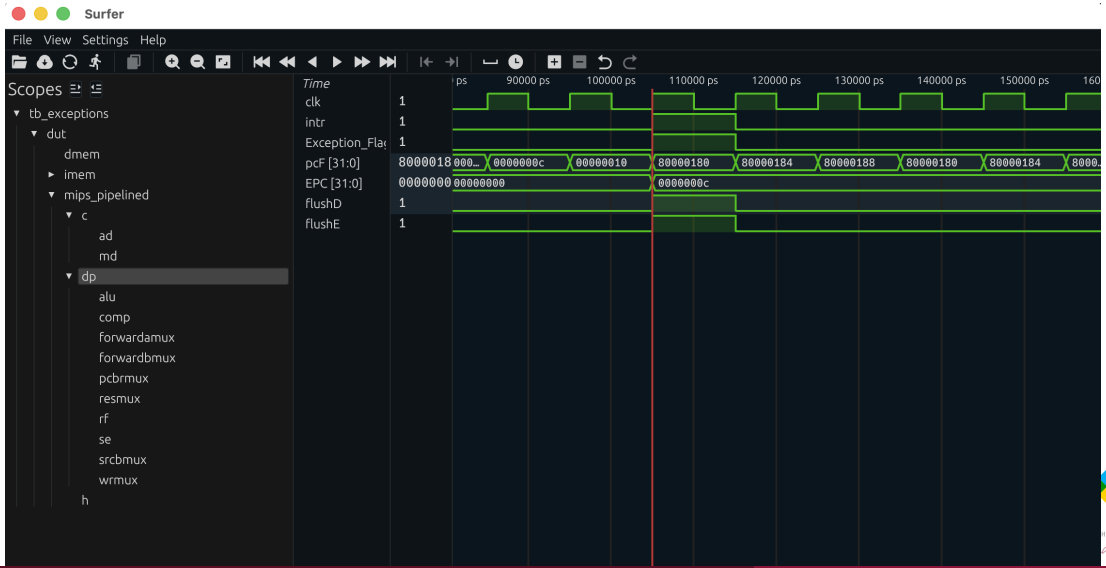
```
$ cat debug_output.txt
```

- The text output mathematically traces the exact hexadecimal state of every boundary stage (`[IF]`, `[ID]`, `[EX]`, `[MEM]`, `[WB]`) dynamically!
- We can visually read precisely when `FlushD/FlushE` sequence pins correctly isolate structural limits directly from the terminal without ever rendering digital logic wave models visually!

16. Reviewing Execution Waves with VS Code "Surfer"

- **Step 4: Inspect the Digital Waveform** – Visually inspect the pipeline delays natively utilizing the VS Code **Surfer** extension!
- Open the generated `tb_computer.vcd` waveform file directly inside Visual Studio Code.
- Track datapath variables across the timeline, heavily inspecting the `EPC`, `FlushD/E`, `StallF/D`, and dynamic ALU Forwarding boundaries.
- **Action Required:** Take meticulous screenshots of your custom Exception faulting waveforms structurally validating logic behaviors within Surfer!

17. Finding the Exception Event in the Waveform



18. Conclusion of the Processor Component

- You now possess the complete hardware logic sequence validating how generic silicon evaluates and processes procedural logic.
- Next steps: Branching beyond the processor logic specifically into Memory Hierarchies and spatial/temporal Caching optimizations spanning across RAM boundaries!