CS250 VLSI Systems Design
Lecture 11: Patterns for Communication Links, Rocket µArchitecture, Testing

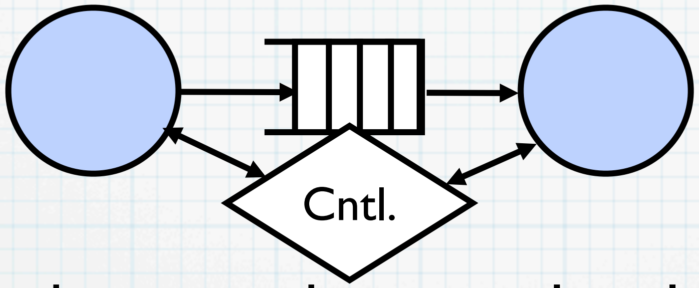
John Wawrzynek, Krste Asanovic,
with
John Lazzaro
and
Brian Zimmer (TA)

UC Berkeley
Fall 2011

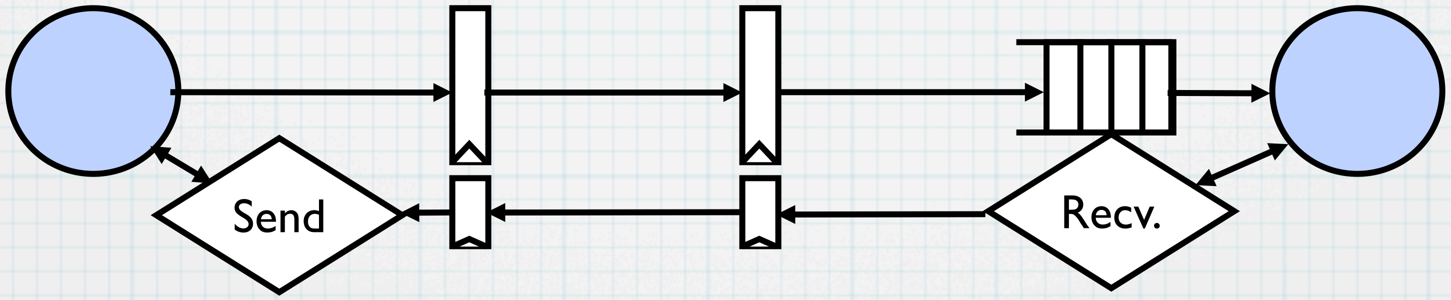
Interconnect Design Patterns

Implementing Communication Queues

- Queue can be implemented as centralized FIFO with single control FSM if both ends are close to each other and directly connected:

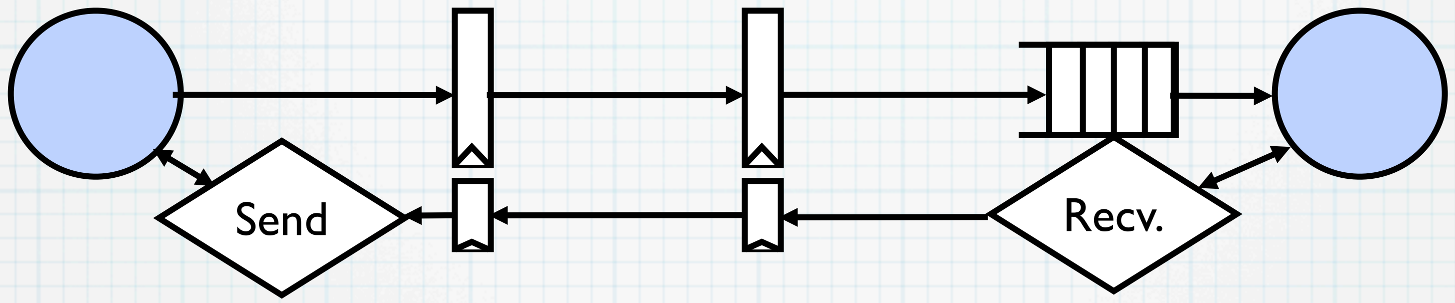


- In large designs, there may be several cycles of communication latency from one end to other. This introduces delay both in forward data propagation and in reverse flow control



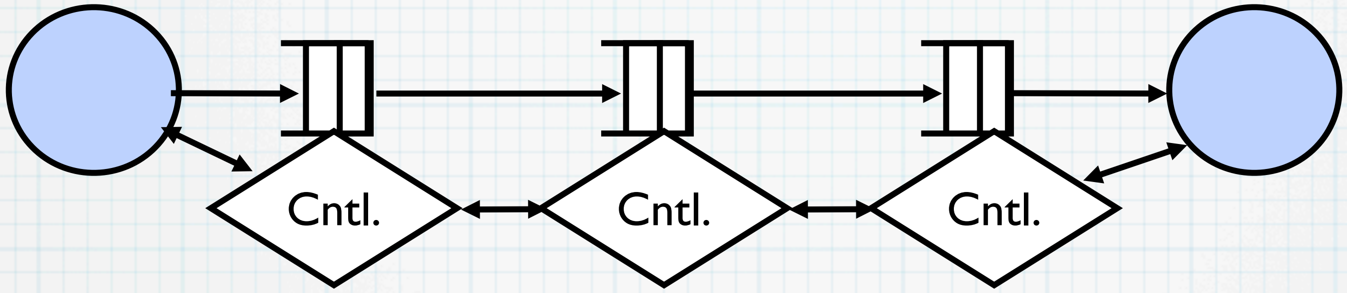
- Control split into send and receive portions. A *credit-based flow control* scheme is often used to tell sender how many units of data it can send before overflowing receiver's buffer.

End-End Credit-Based Flow Control



- For one-way latency of N cycles, need 2*N buffers at receiver to ensure full bandwidth
  - Will take at least 2N cycles before sender can be informed that first unit sent was consumed (or not) by receiver
- If receive buffer fills up and stalls communication, will take N cycles before first credit flows back to sender to restart flow, then N cycles for value to arrive from sender
  - meanwhile, receiver can work from 2*N buffered values

Distributed Flow Control



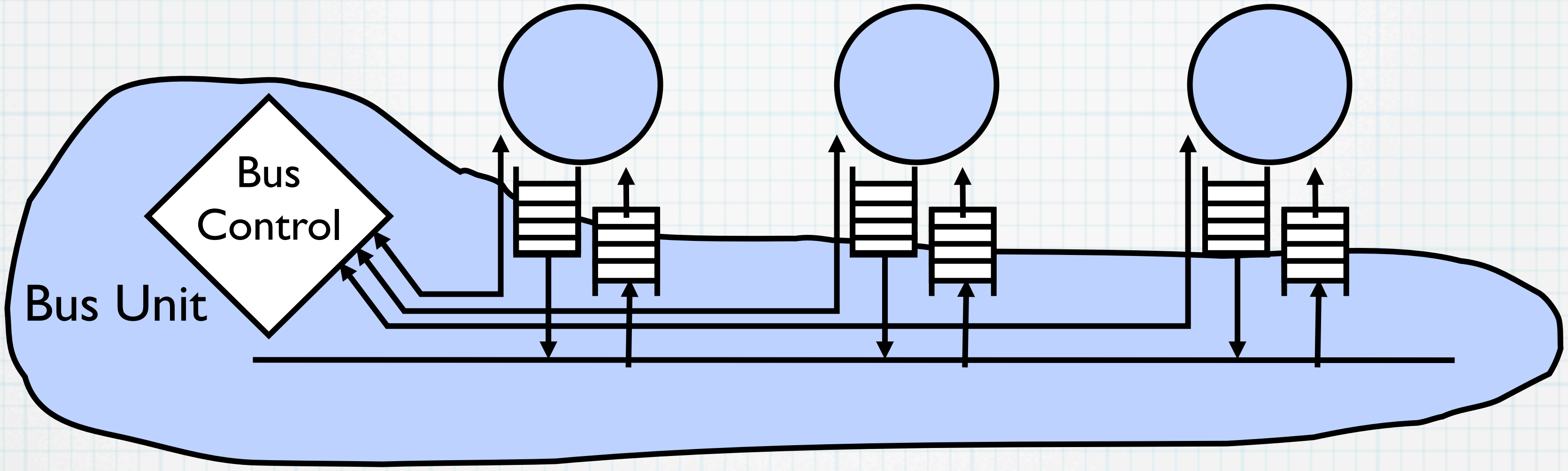
- An alternative to end-end control is distributed flow control (chain of FIFOs)
- Requires less storage, as communication flops reused as buffers, but needs more distributed control circuitry
  – Lots of small buffers also less efficient than single larger buffer
- Sometimes not possible to insert logic into communication path
  – e.g., wave-pipelined multi-cycle wiring path, or photonic link

Network Patterns

Connects multiple units using shared resources

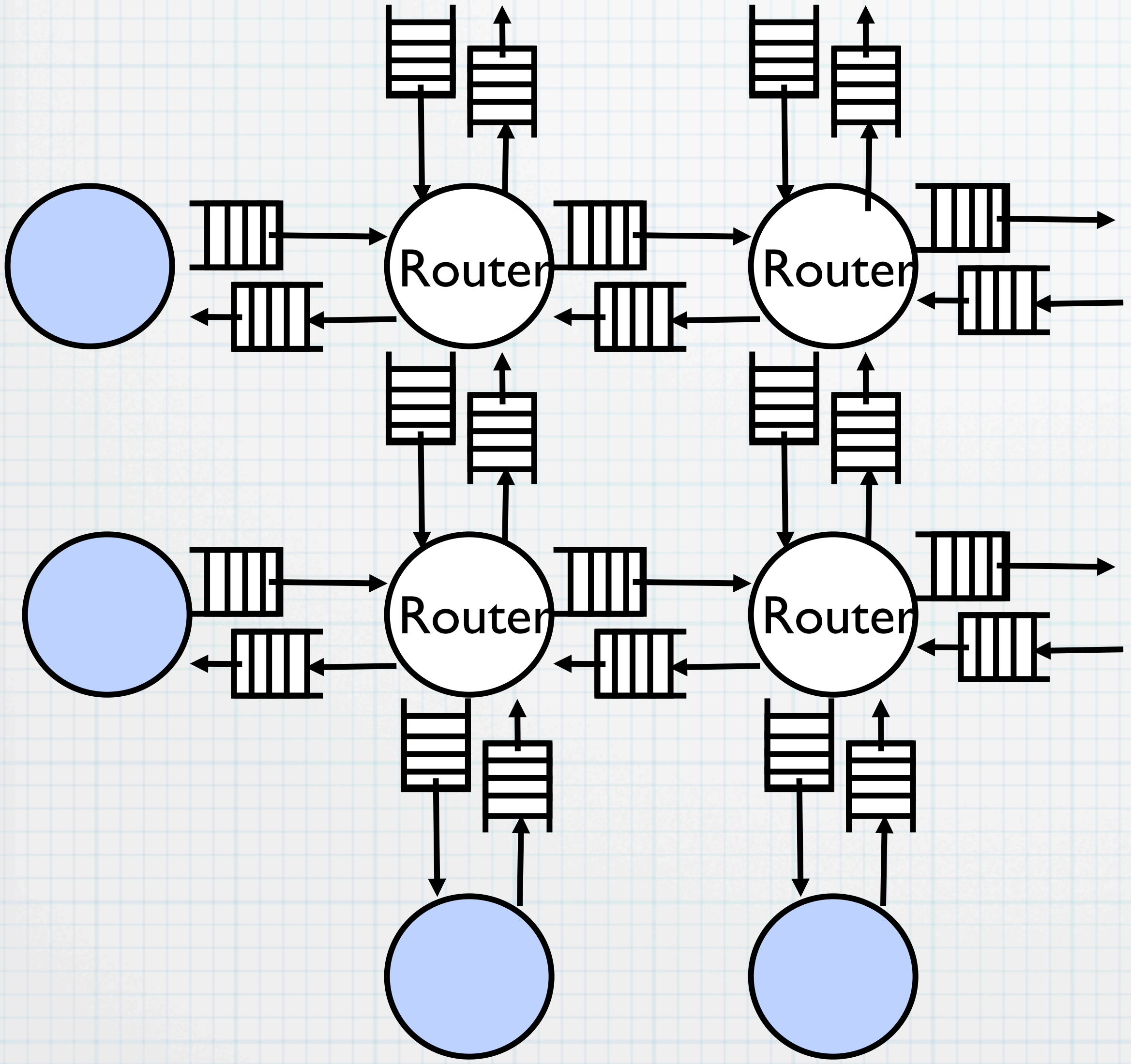
- Bus
  - Low-cost, ordered
- Crossbar
  - High-performance
- Multi-stage network
  - Trade cost/performance

Buses



- Buses were popular board-level option for implementing communication as they saved pins and wires
- Less attractive on-chip as wires are plentiful and buses are slow and cumbersome with central control
- Often used on-chip when shrinking existing legacy system design onto single chip
- Newer designs moving to either dedicated point-point unit communications or an on-chip network

On-Chip Network



- On-chip network multiplexes long range wires to reduce cost
- Routers use distributed flow control to transmit packets
- Units usually need end-end credit flow control in addition because intermediate buffering in network is shared by all units

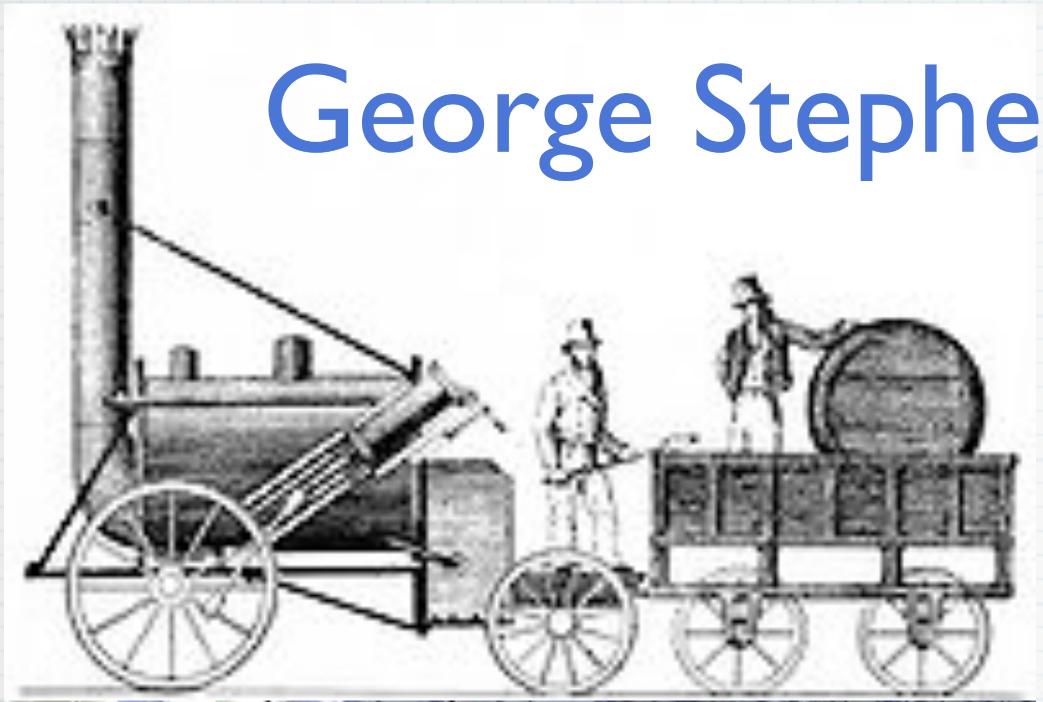
Rocket µArchitecture

UCB Rocket: An In-Order RISC-V Decoupled µArchitecture

- A family of µarchitectures supporting hardware floating-point, demand-paged virtual memory
- In-order single or dual-issue, decoupled floating-point unit, precise traps
- 32-bit or 64-bit implementations
- From 5-stage to ~9-stage pipelines
- Designed to be close to commercial cores

- Single-issue version will be made available in time for class projects

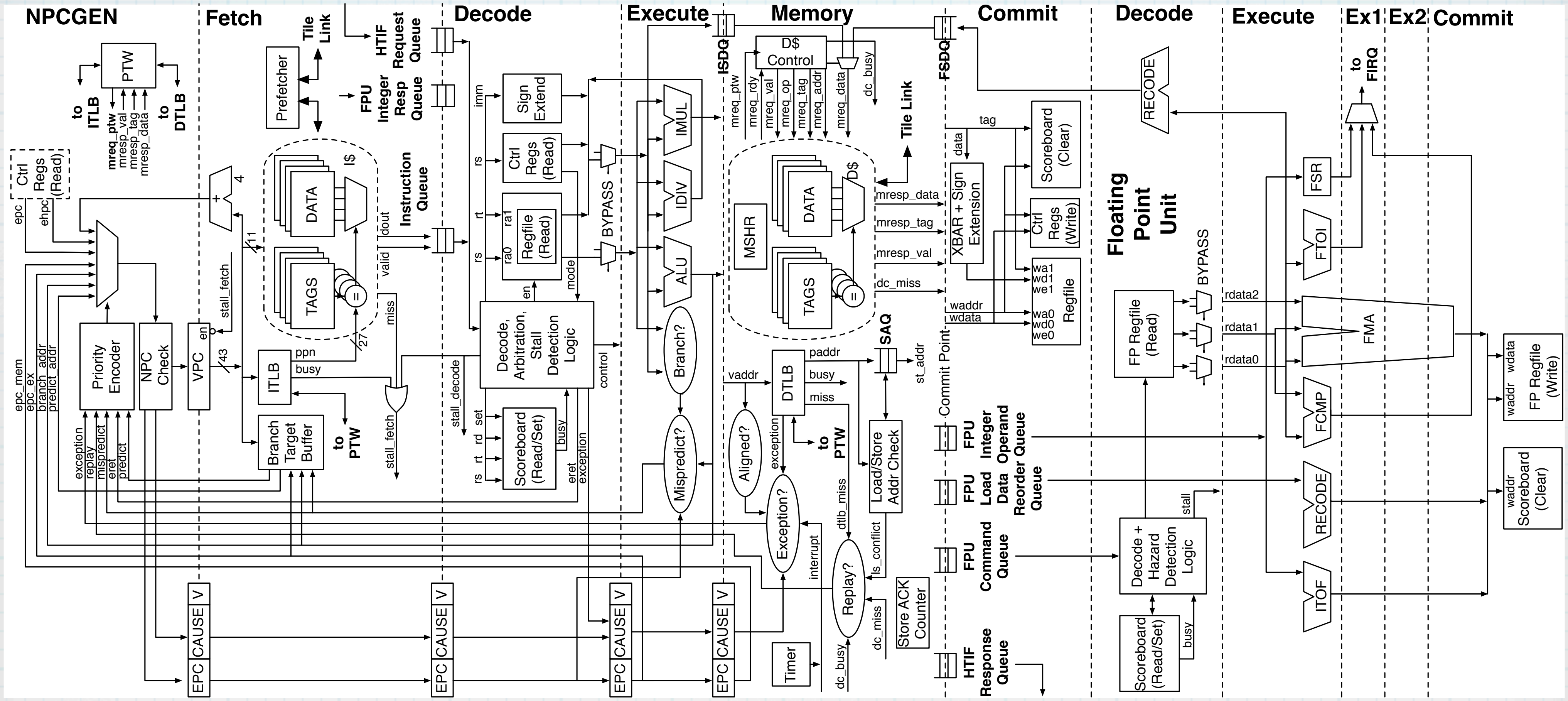
George Stephenson's Rocket



[AllyJane, LensFlare]

"The Rocket was the most advanced steam engine of its day. It was built for the Rainhill Trials held by the Liverpool & Manchester Railway in 1829 to choose the best and most competent design. It set the standard for a hundred and fifty years of steam locomotive power. Though the Rocket was not the first steam locomotive, its claim to fame is that it was the first to bring together several innovations to produce the most advanced locomotive of its day, and the template for most steam locomotives since." [Wikipedia]

A Simple Core?



Rocket Pipeline Structure

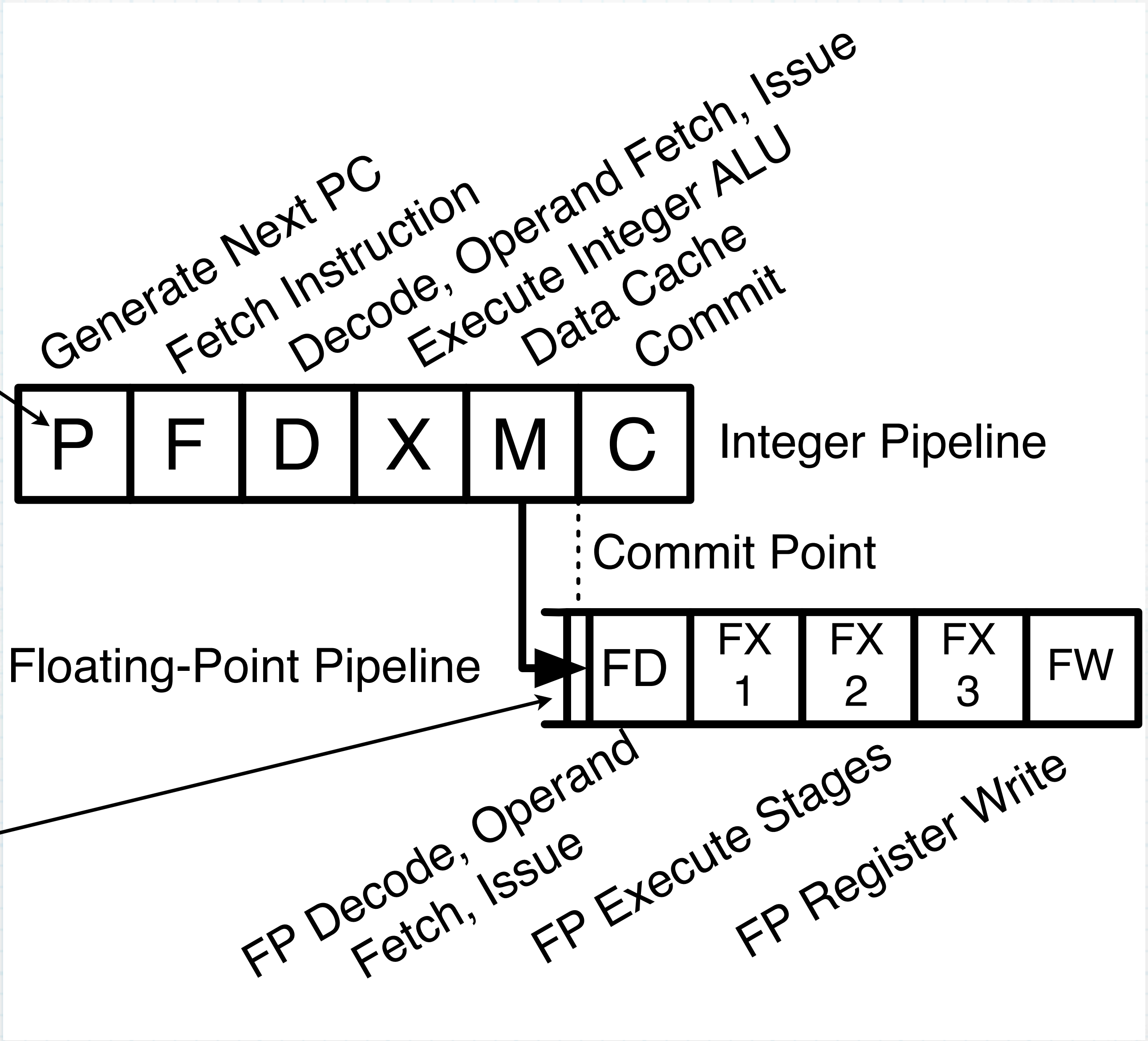
Four major phases of execution

- Instruction fetch
  - Get instruction bits from I-cache
- Decode, including operand fetch and issue
  - Read register file, determine interlocks and bypass control
- Execution
  - Perform instruction
- Commit
  - If no traps or interrupts, write architectural state

Each phase can contain multiple pipeline stages, but approx. one stage each in initial design.

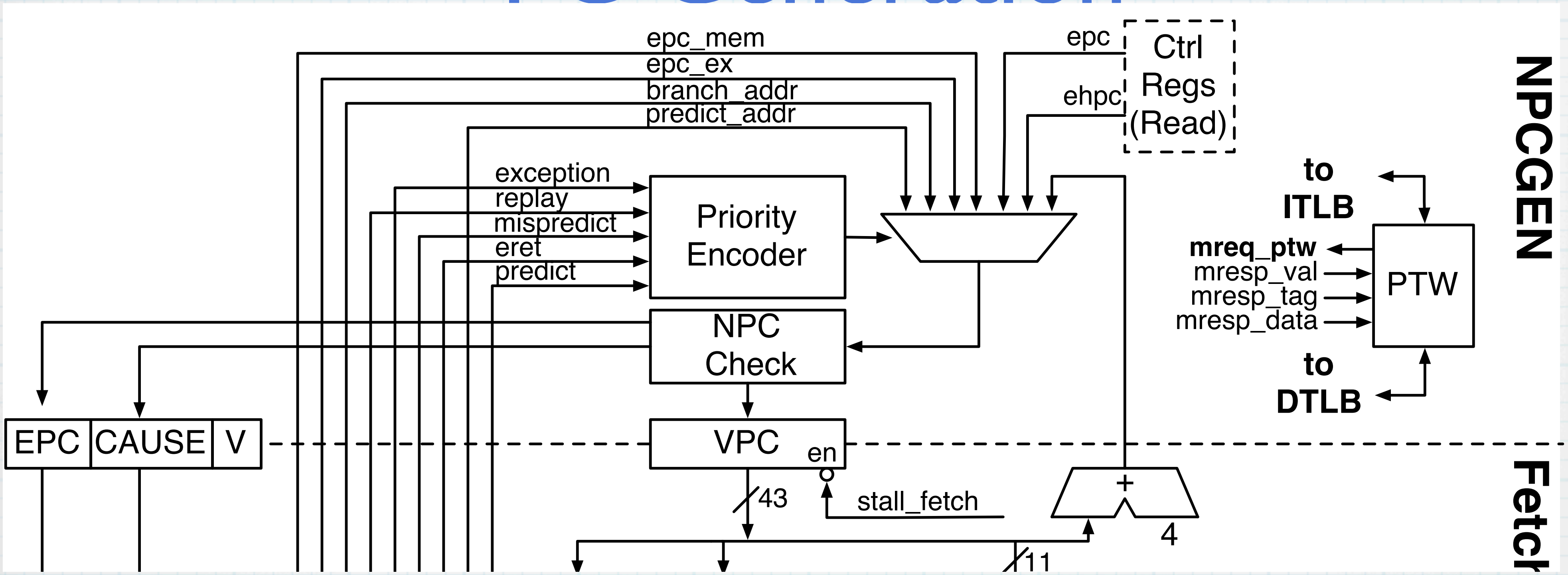
Rocket 5-Stage Pipeline Structure

P is a pseudo-stage, as contents spread over many stages



FPU decoupling queue placed at commit point in pipeline

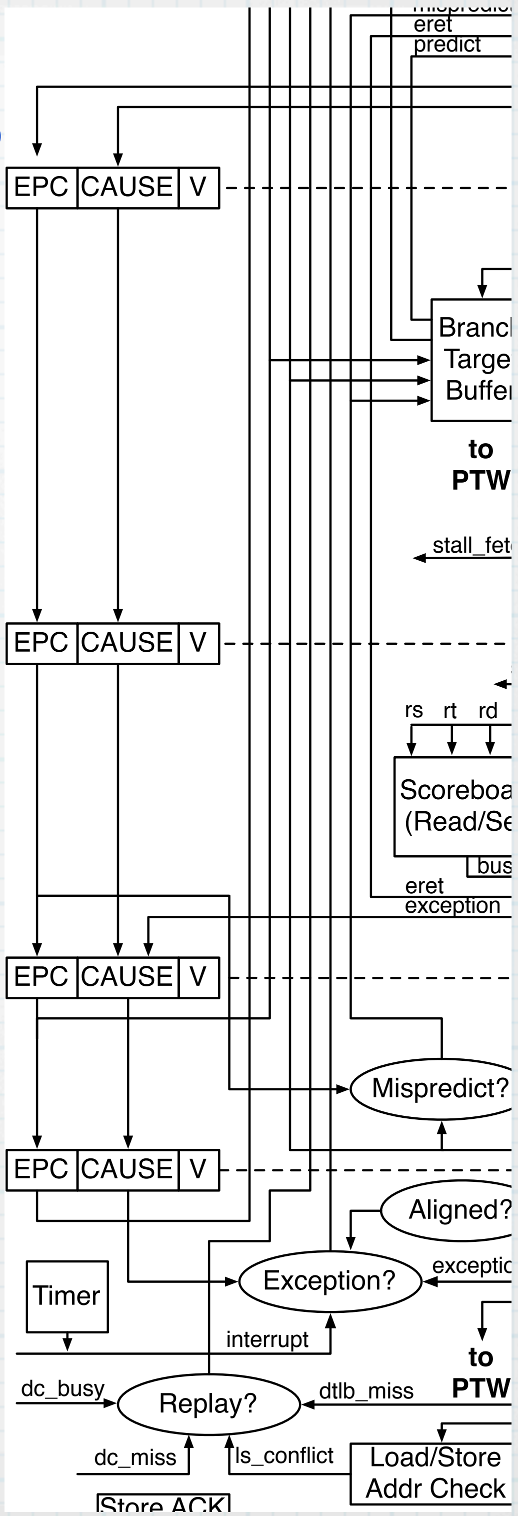
PC Generation



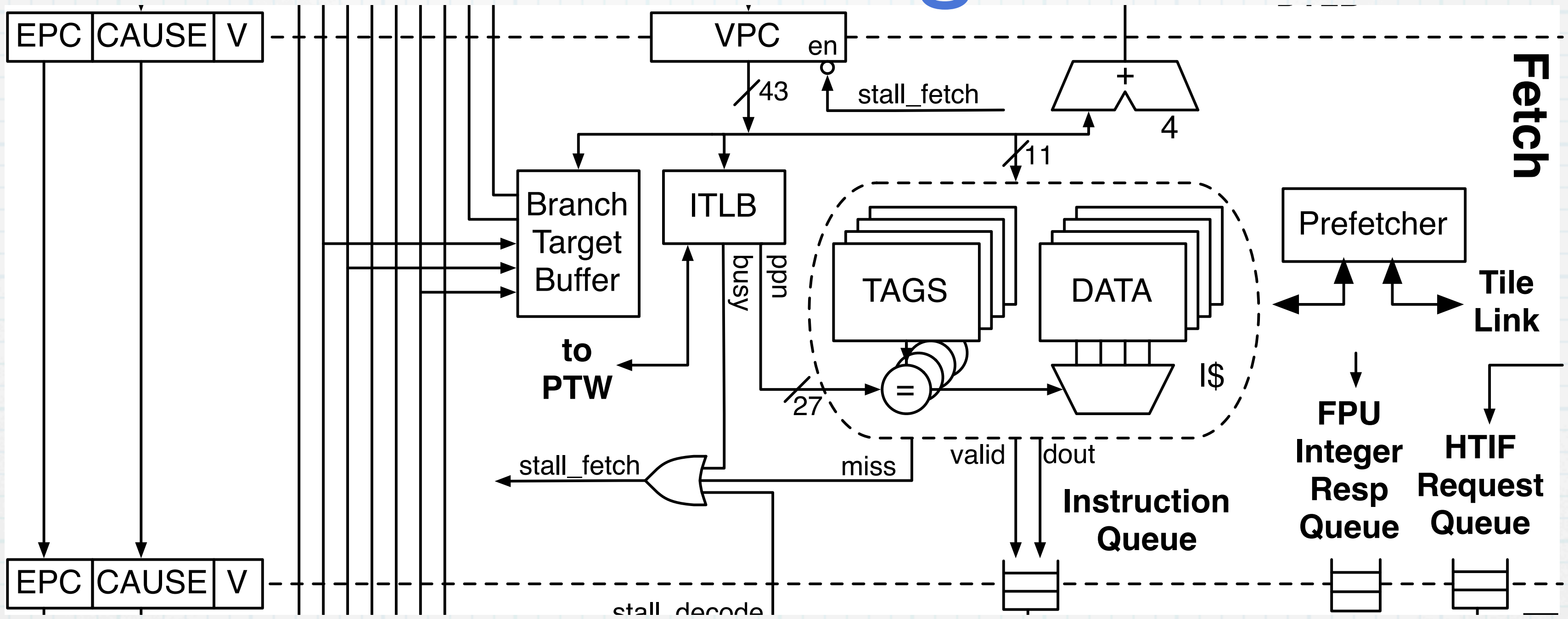
- Next PC can come from number of sources
  - PC+4 if sequential fetch (predicted not-taken)
  - Predicted branch address (if predicted taken)
  - Resolved branch address (if mispredicted)
  - Replay PC (if pipeline flush/replay from either X or M stage)
  - Trap handler address (on trap/interrupt)
  - Restore PC (at end of trap/interrupt handler)

Implementing Precise Traps

- Handle traps in program order at end of memory stage (the commit point)
- Synchronous trap can be generated in any stage, held in Error PC & Cause shifted down pipeline
- EPC always holds PC of instruction in that stage
- Asynchronous interrupts handled in memory stage
- Trap/interrupt flushes pipe and resets PC to handler address
- RISC-V floating-point ISA designed to have no traps (only exception flags), so commit point before FPU decode

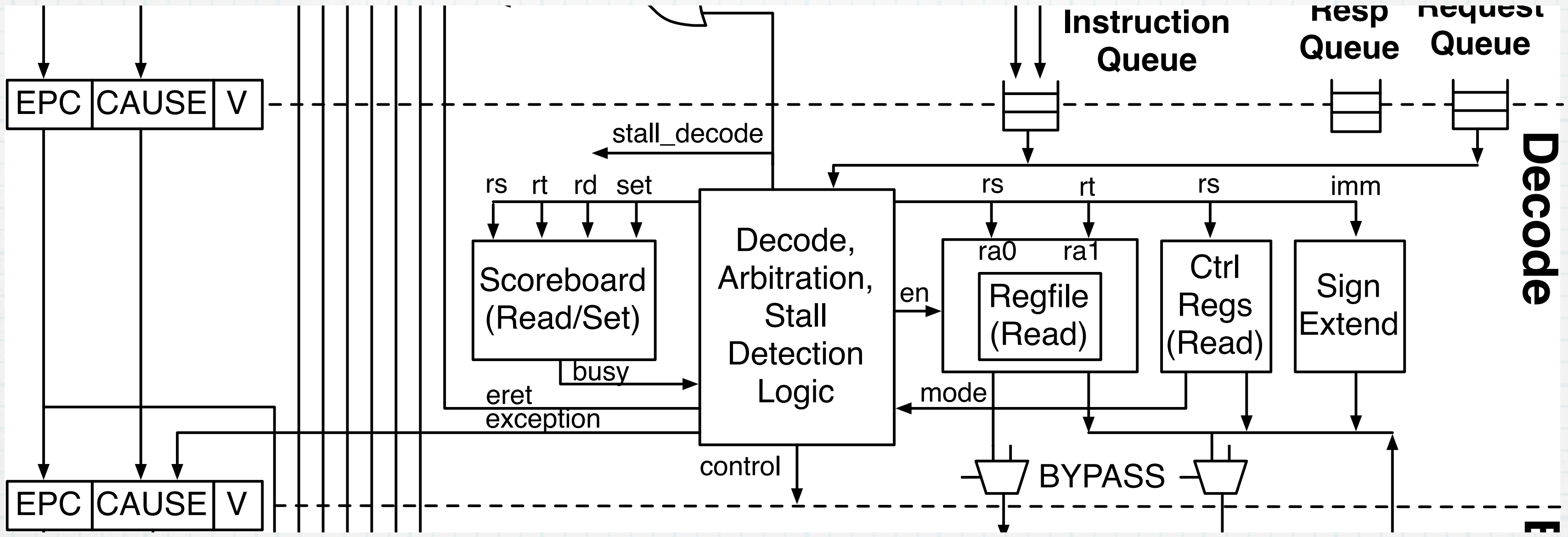


Fetch Stage



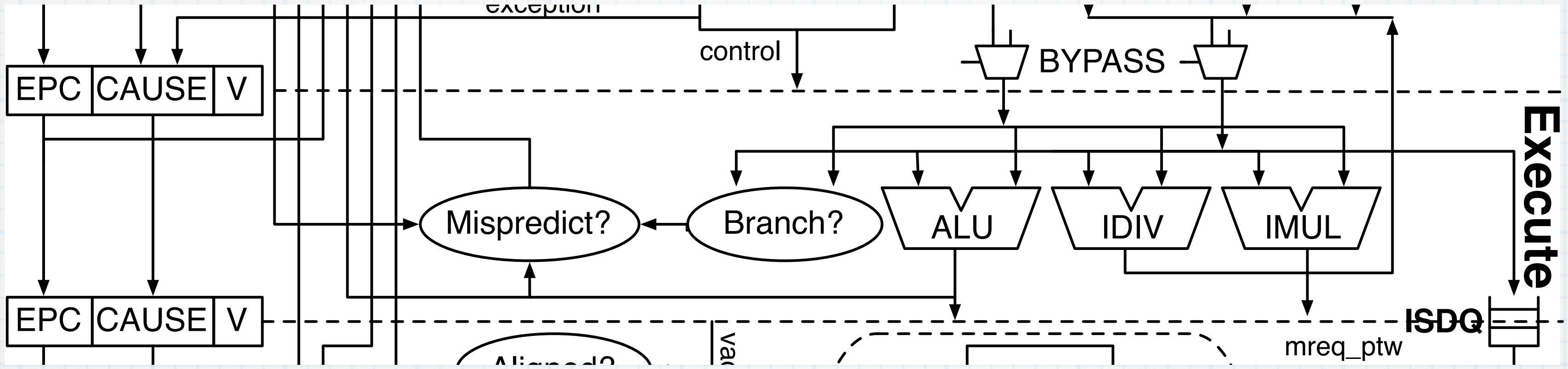
- Predict next PC from current PC using BTB - fed back to P stage
- Fetch instructions from cache into instruction queue
- Translate virtual address PC into physical address PC for I-cache physical tag check, check for illegal PC -> signal trap
- I-cache miss goes to memory system, I-stream prefetcher fetches sequential blocks ahead of miss

Decode Stage



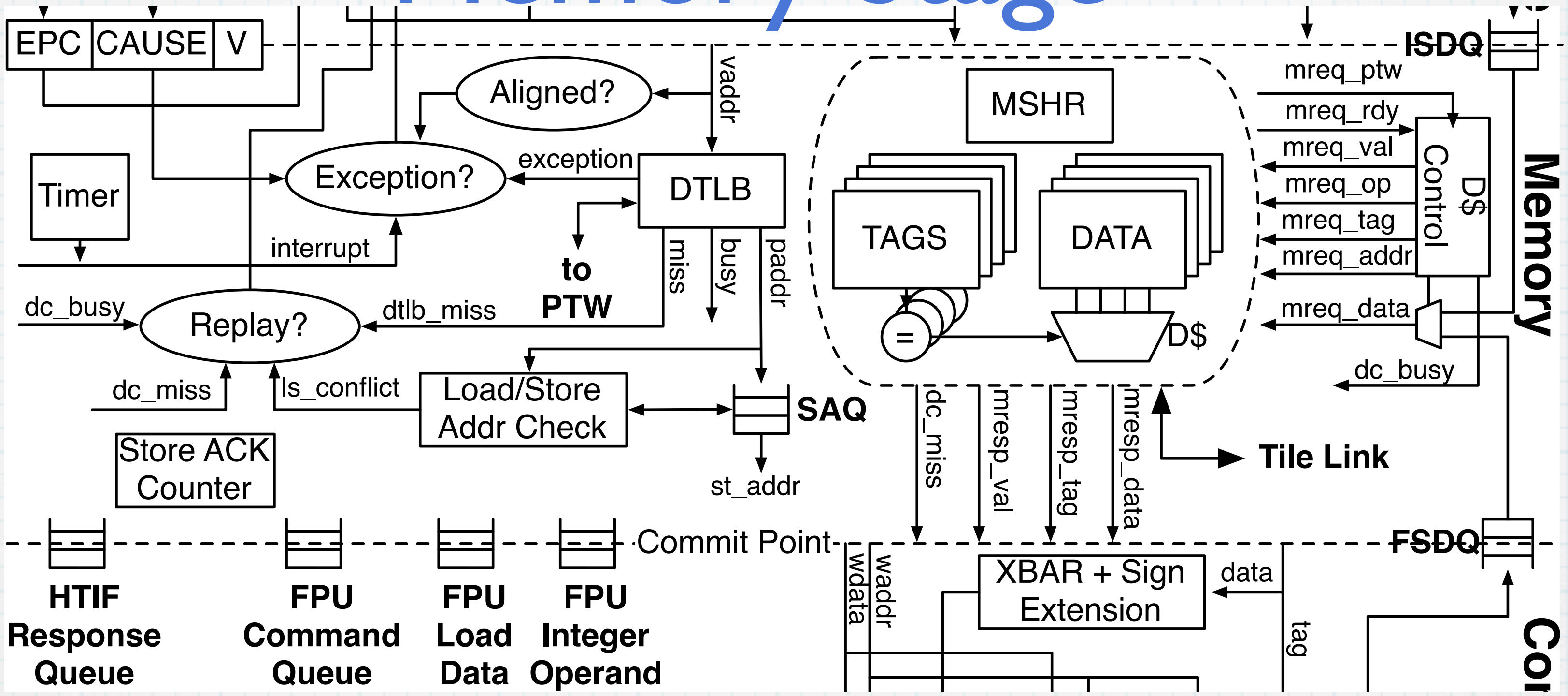
- Decode instructions from queue, check for illegal ops -> signal trap
- Fetch register operands and sign-extend immediate
- Check for unavailable source operands using scoreboard (busy bit per register), stall decode if not available
- Set busy bit for long latency operations
- Calculate bypass control and mux bypass operands into ALU inputs

Integer Execute Stage



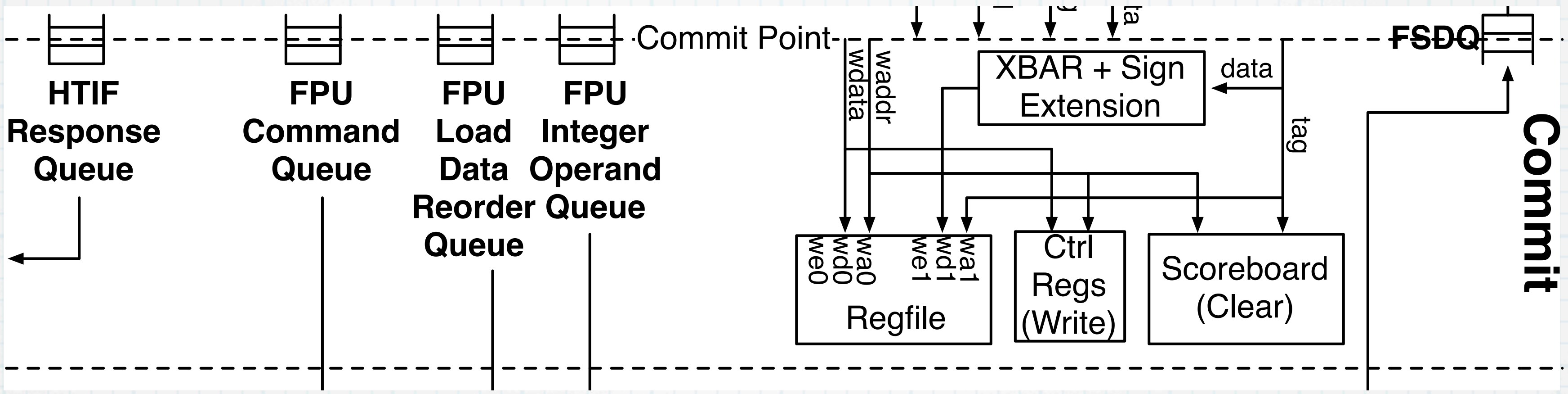
- Most integer instructions complete in one cycle and can be bypassed to next instruction
- Integer multiply takes few cycles overlapped with memory stage
- Integer divide takes many cycles - so sets busy bit on destination register
- Branches resolved in ALU - mispredict detected by comparing target address with EPC in following instruction (was correct path taken?)
- ALU calculates load+store addresses, integer store data placed in store data queue (ISDQ)

Memory Stage

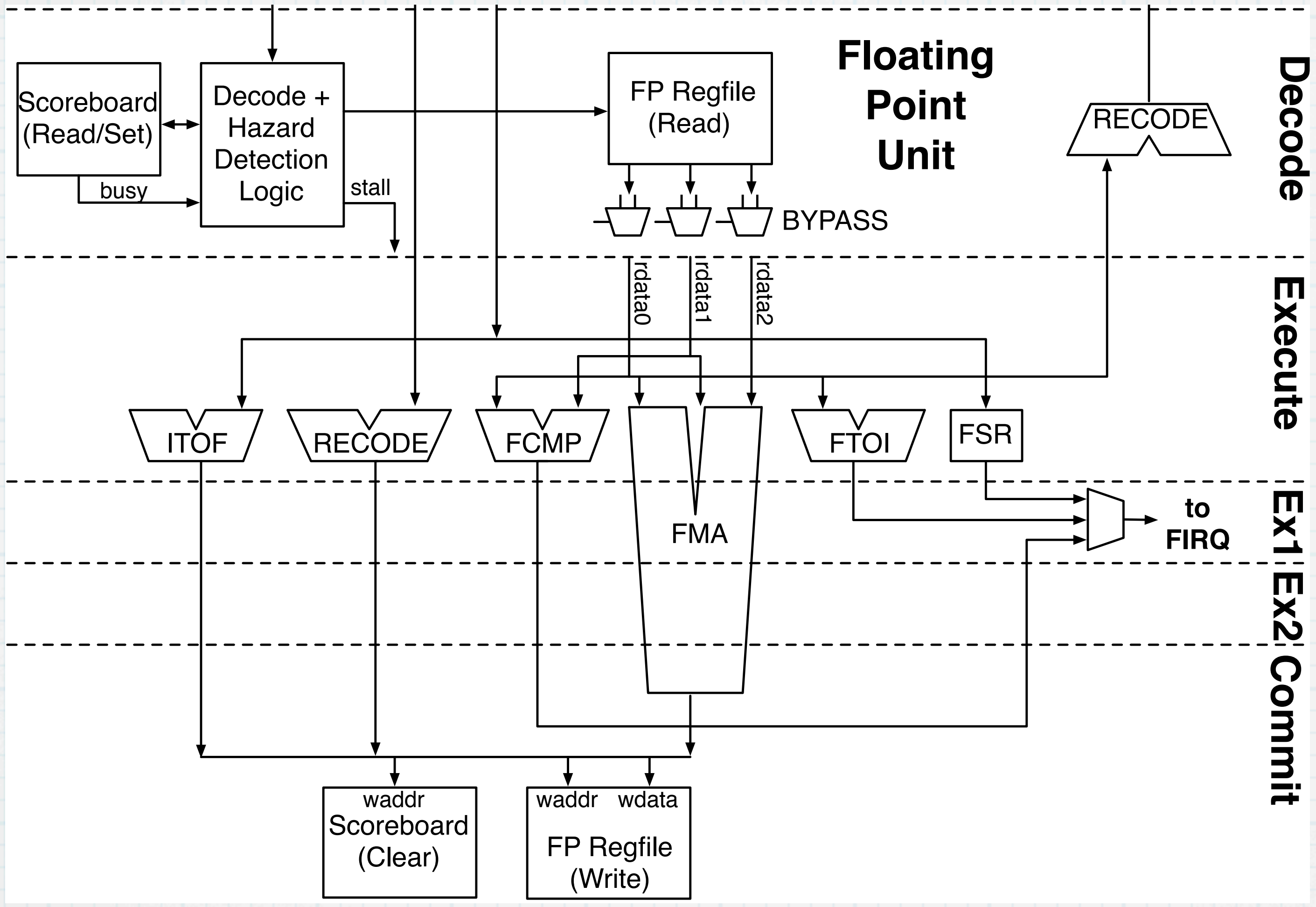


- Virtual load/store address translated and checked -> illegal address trap
- Store addresses are always queued in-order in SAQ to wait for data in ISDQ or FSDQ (from FPU). Stores go-ahead when address and data available.
- Loads can bypass stores if no conflict, but replayed if conflict with address in SAQ.
- Non-blocking cache supports multiple outstanding primary and secondary misses.
- Flushes pipe and injects handler PC if any traps or interrupts.
- End of memory stage is commit point - FPU operations enqueued if no traps.
- FPU load instruction enqueues command to read FPU load data queue
- FPU store instruction enqueues command to send FP register value to FSDQ

Commit Stage



- Architectural registers written with final values
- Busy bits on scoreboard cleared as results arrive
- Data cache finishes aligning and sign-extending small width values. Rocket only bypasses 32-bit and 64-bit values from end of memory stage, other sizes of load operands bypassed from end of commit stage.
- FPU begins decoding instructions from FPU queue.



- FPU built around a fused multiply-add unit (2008 revision of IEEE 754 FP standard) with full hardware support for all cases including subnormals.
- Regfile holds value in internal recoded format with extra bit to simplify handling of subnormals. Have to convert on load/store and move to/from integer.

Design Verification

Verification large part of NRE cost

- Too expensive to respin part
  - prototype cost in $Ms
  - lost time-to-market $10Ms
- 2-3X engineer time on verification versus design
- Only getting worse over time as chips get larger and more complex

Types of Checks

- Design verification: "Does RTL design implement the functional specification?"
- Tool/implementation checking: "Does design layout match RTL design?"
- Physical design checking: "Does design work across all process corners, obey all the electrical design rules (antenna rules, electromigration, ...), is power/clock/reset distribution OK, does design meet design-for-X rules (X=test, manufacturing,reliability,...)"
- Manufacturing testing: "Does a fabricated chip implement the design to specification?"

Design Verification Greatest Challenge

- Tool/implementation checking mostly automated using static formal verification checks (though finding and fixing error can be labor-intensive)
- Same for EDRC rules and other physical design checks
- Manufacturing tests can be automatically generated from RTL if scan chains used for all state elements (automatic test pattern generation - ATPG)

Source of Bugs in RTL Design

- Specification incorrect
  - Designers built an implementation faithful to the specification, but the specification was wrong.
- Specification misread
  - Designers built an implementation faithful to their reading of the specification, but they misunderstood specification.
- Incorrect RTL design
  - The RTL design does not do what designer wanted it to.
- Incorrect RTL coding
  - The RTL design was correct in designer's head, but the RTL code doesn't match that RTL design.

Avoiding Incorrect Specification

- Build an executable version of the specification, which should be simple functional model of intended design
  - For RISC-V cores, we have a C++ instruction set interpreter, requiring only a few lines of code for each instruction.
- Exercise executable specification inside system-level test harness with representative workload
  - For RISC-V, we have built a test harness that can run programs on simulator. Classic test for processors was booting Unix on functional model.
  - System-C common in industry for this level of modeling, where entire system modeled sufficiently to run whole software stack. FPGA emulation popular to accelerate model.

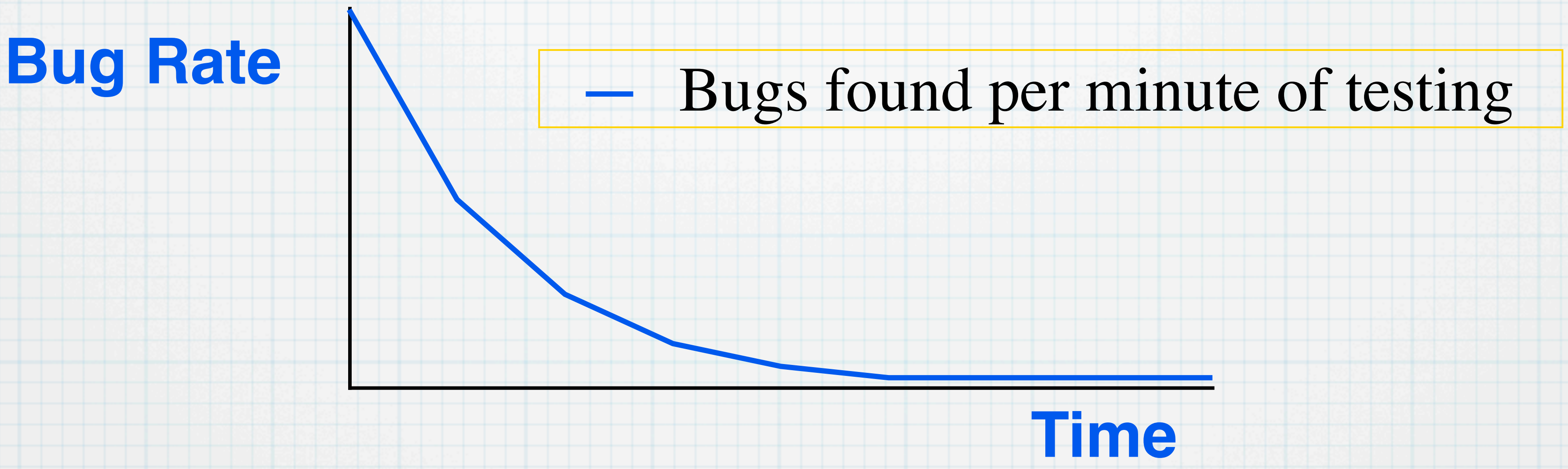
Avoiding Misread Specification

- Have executable specification as "golden model"
- Have different designers write executable specification and system test code to catch misread specification when building golden model
  - If errors found, don't just fix model, also rewrite specification to make it less ambiguous or more readable.
- Perform extensive directed and random testing to compare RTL design with golden model

Catching bad RTL design or coding

- Perform extensive directed and random testing to compare RTL design with golden model
- Modern processor design team will perform many billions of cycles of RTL simulation using 10,000s cores prior to tapeout

When are you done?



- But did you find all bugs, or reach limits of your test coverage?

Test Coverage

- Did every bit toggle?
- Was every value on every bus?
- Was every state machine transition taken?
- Could your tests observe this happening?

Unit Testing

- Divide and Conquer
- Tradeoff between cost of defining unit boundary and improved test visibility and coverage
- Typical granularity of test units in processor:
  - Floating-point functional units
  - Caches
  - Integer core
  - Whole processor