Shakti I Class: Introduction

Nitya Ranganathan

Design Team: Rahul Bodduna, Shalendar Kumar, Arjun Menon, Sujay Pandit, Vipul Vaidya, Nitya Ranganathan

What is the I-Class processor?

- I-Class is a superscalar out-of-order (OoO) processor with potential applications in general purpose computing and high-end embedded markets
- A gentle introduction to version 1.0 of the core, not covering SoC
  - High-level design of version 1.0 with extra details on few blocks
  - Interesting design trade-offs
  - Current and future work
- **Note:** This is work in progress
  - Implementation in BSV, Verification and Performance Analysis ongoing

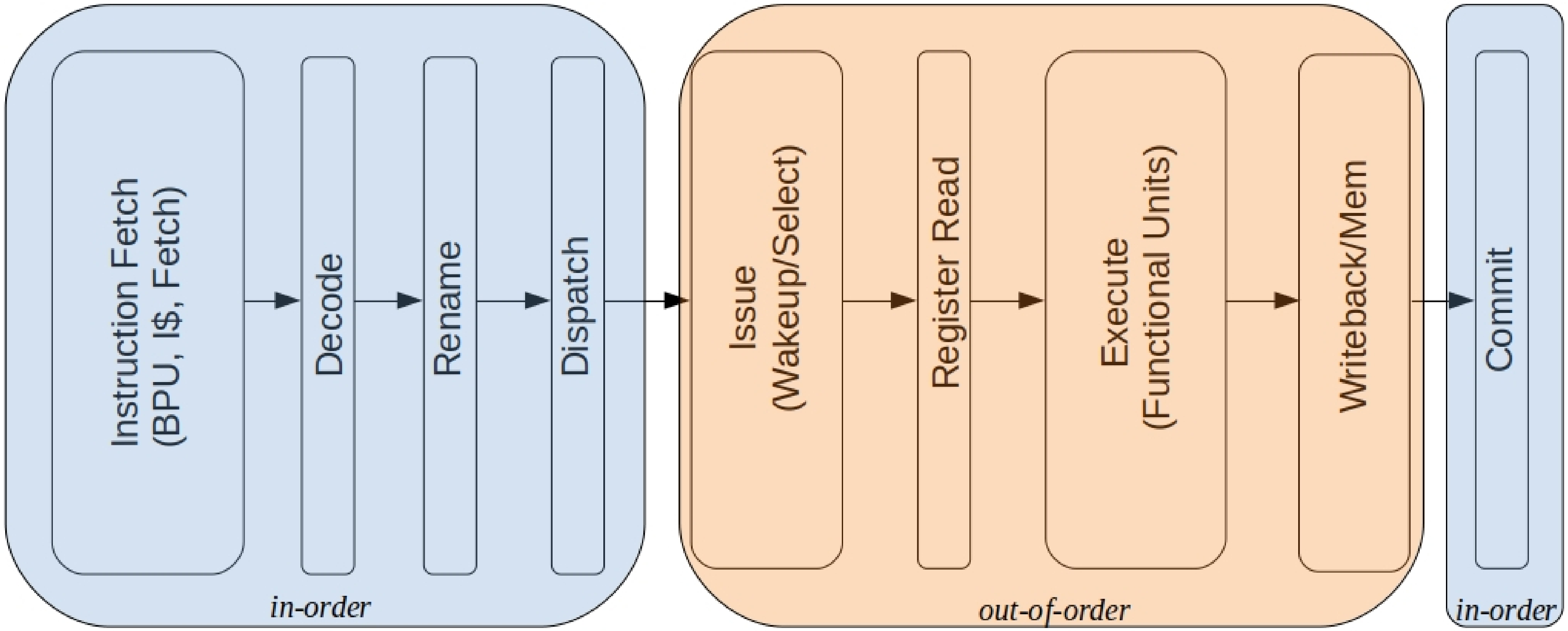
Designing an Out-of-Order processor

- Multi-wide out-of-order design is difficult to implement and verify even in large corporations
- We are a small team!
- How to choose from 1000's of proposed features for OoO performance/power/area?
- Employ a combination of techniques
  - Lessons from academia and industry
  - "Intuition" about OoO design tradeoffs
  - Feature refinement based on performance modelling, bottleneck analysis etc.
  - Simple first-cut OoO design, enhancements in next version
  - Cut development time with Bluespec; instantiate some components from libraries
- Balancing performance and power is critical
  - High performance designs typically come at the cost of power or area
  - A new performance feature is beneficial only if it significantly improves execution time without severely impacting power/area

Key Performance Enablers for OoO processors

- Instruction supply
  - Accurate branch predictor
  - Low I-Cache miss rate
  - Early wrong path detection
  - Fast recovery from misspeculations
- Data supply
  - Low load-to-use latency
  - Low D-cache miss rate and miss penalty
  - Good store commit bandwidth
- Pipelining and data path
  - Optimal pipelining for high frequency while balancing branch misprediction penalty
  - Split issue queues to implement larger instruction windows
  - Operand bypass for back-to-back execution of dependent instructions
  - Pipelined functional units with low latencies
- **Summary:** Keep the processor busy, reduce wasteful execution and spend very little time waiting for data from memory!

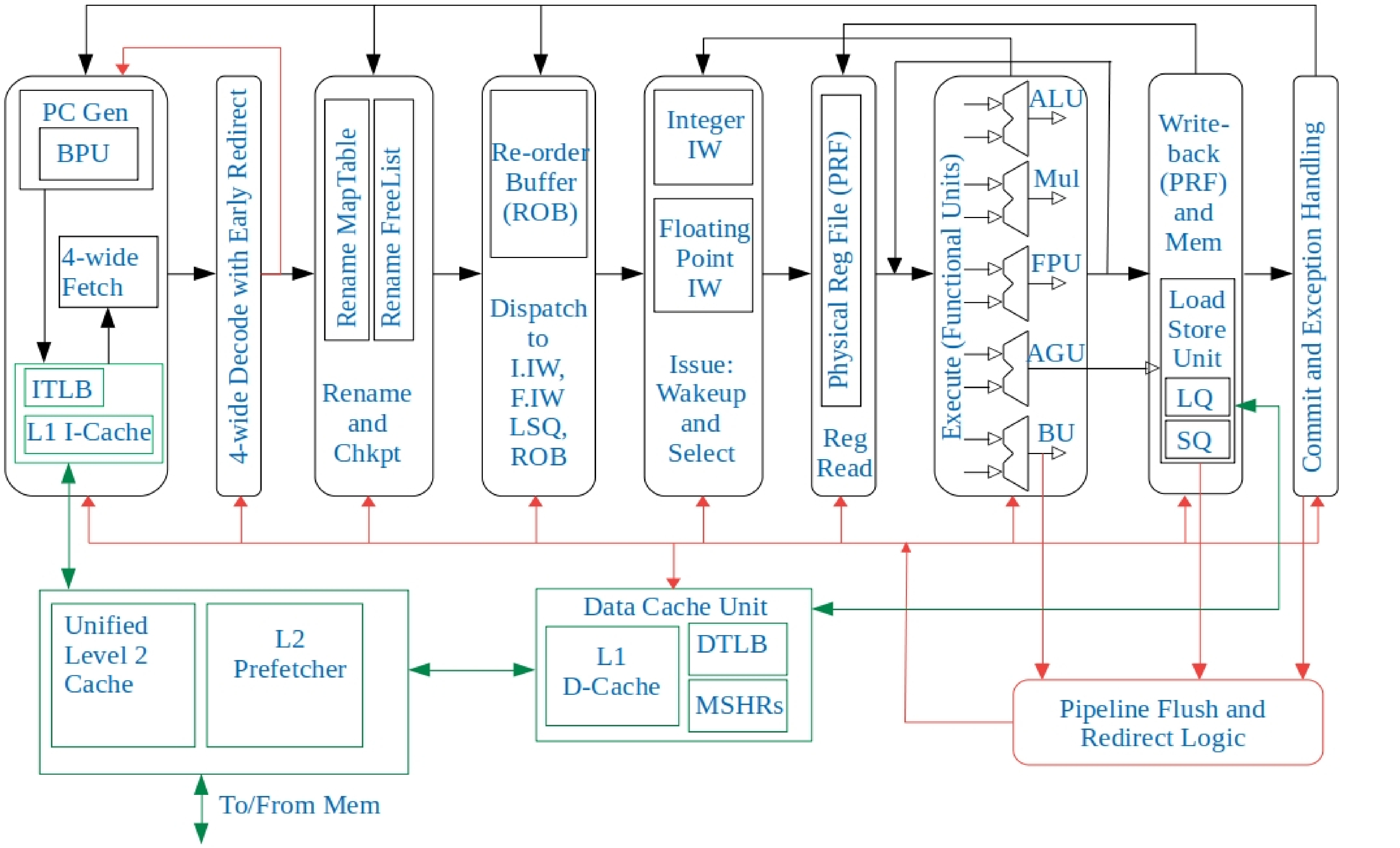
Basic I-Class Pipeline (version 1.0)



- 4-wide out-of-order core: fetch/dispatch/issue/commit 4 insts/cycle
- 12-stage pipeline for simple integer operations
- RV64IMAFDC (int, mul/div, atomic, single/double precision floating point, compressed)

- **Key features:** Multiple branch prediction, register renaming with checkpointing, separate issue windows for Int and FP, reorder buffer, operand bypass, pipelined functional units (except div/sqrt), memory dependence predictor, non-blocking cache

I-Class Pipeline (detailed)

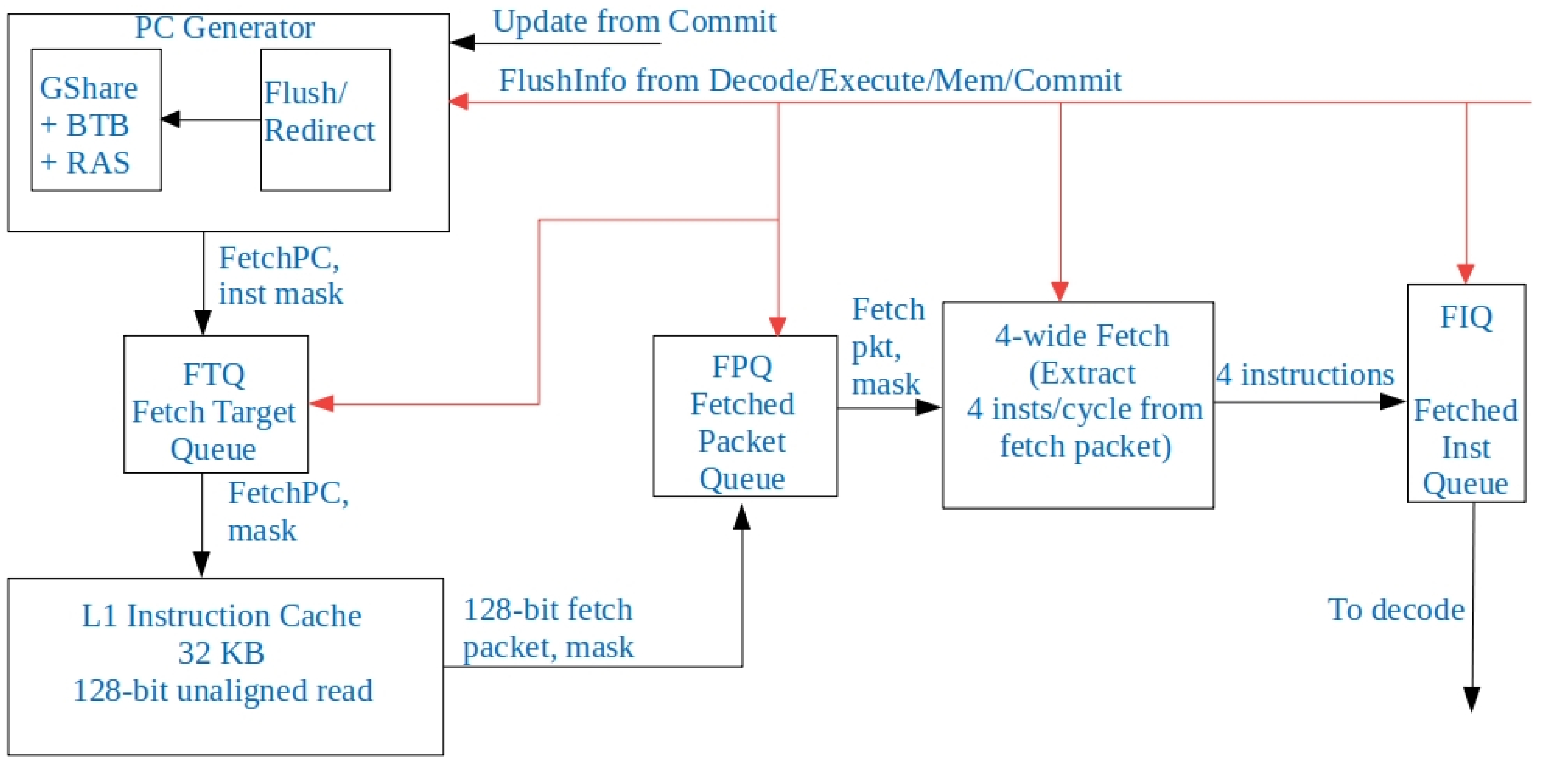


Note: *Memory accesses are in green, Flushes and Redirects are in red*

Latencies: Pipeline stages, functional units

Branch prediction, I-Cache Read, 4-wide Instruction Fetch	3 cycles
Instruction Decode	1 cycle
Renaming and Checkpointing	1 cycle
Dispatch (Allocate to IWs, ROB, LSQ)	1 cycle
Issue (Wakeup/Select)	2 cycles
Register Read (From Physical Register File)	1 cycle
Execute (ALU, AGU, BU, Mul, Div, FPU)	1 to 32 cycles for arith; Minimum 4 cycles for loads Single cycle int add/sub, shifts, logical Pipelined int multiply, FP add/sub/mul, fmac Non-pipelined int divide, FP divide/sqrt
Writeback (To PRF)	1 cycle
Commit	1 cycle

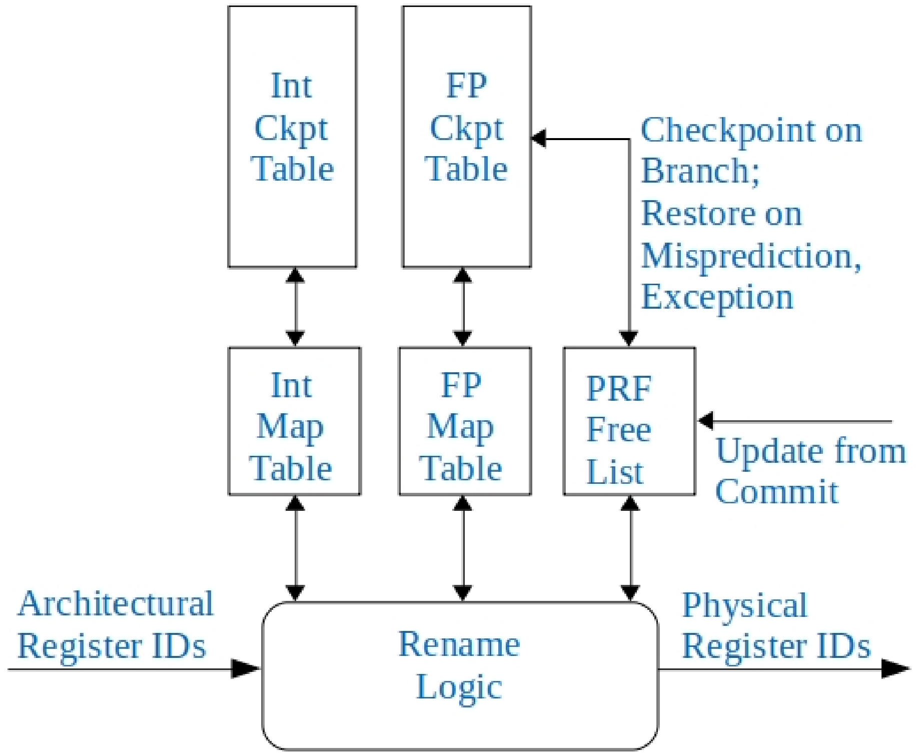
Instruction Fetch and Branch Prediction



- Fetch any combination of four 32-bit or 16-bit instructions; stop on predicted taken branch or end of cache line
- Compressed instuction support lowers I-Cache footprint but complicates branch prediction and instruction extraction from fetch packet!
- BPU: Gshare-style branch direction predictor, branch type predictor, BTB and RAS
- Several decoupling buffers between blocks

Decode and Register Renaming

- **Decode:** Simple decode for RISC-V
  - Few fixed formats, only two instruction widths
- Detect *definite* mispredictions based on decoded information like opcode, branch type
  - Flush Fetch, Decode stages; Send early redirect to BPU
- **Renaming** removes Write-After-Read (WAR) and Write-After-Write (WAW) dependences
  - Only true data dependences remain
- Rename Architectural Register File (ARF) identifiers to Physical Register File (PRF) identifiers
- Checkpoint register map tables, free lists regularly
  - Quickly recover processor state from mispredictions



```
Example:
ADDW R6, R6, R4     =>      ADDW P24, P15, P14
MUL R6, R6, R10     =>      MUL P35, P24, P19
```

Dispatch and Issue

- **Dispatch** checks for structural hazards in issue windows, re-order buffer and load/store queues
  - Dispatch to Issue windows; Allocate ROB, LSQ entries
  - Dispatch detects csr instructions, fences and atomics
- **Issue** consists of Wakeup (set sources ready) and Select (pick for execute)
  - **Wakeup** instructions from issue windows based on result tags broadcast from functional units
  - Out-of-order wakeup when source registers are available from PRF/bypass network
  - **Select** up to 4 instructions every cycle based on certain constraints: functional unit and register write port availability
  - Selected instructions are immediately removed from the issue windows
  - Wakeup/Select one of the most timing critical loops
- **Re-order Buffer (ROB)** stores instruction metadata for all instructions in flight
  - 80-entry ROB => maximum 80 instructions in flight
  - Split Instruction Window/ROB design to reduce complexity of tag broadcast
  - Simple ROB (only instruction metadata) is required to preserve sequential semantics

Load/Store Queues and Memory Disambiguation

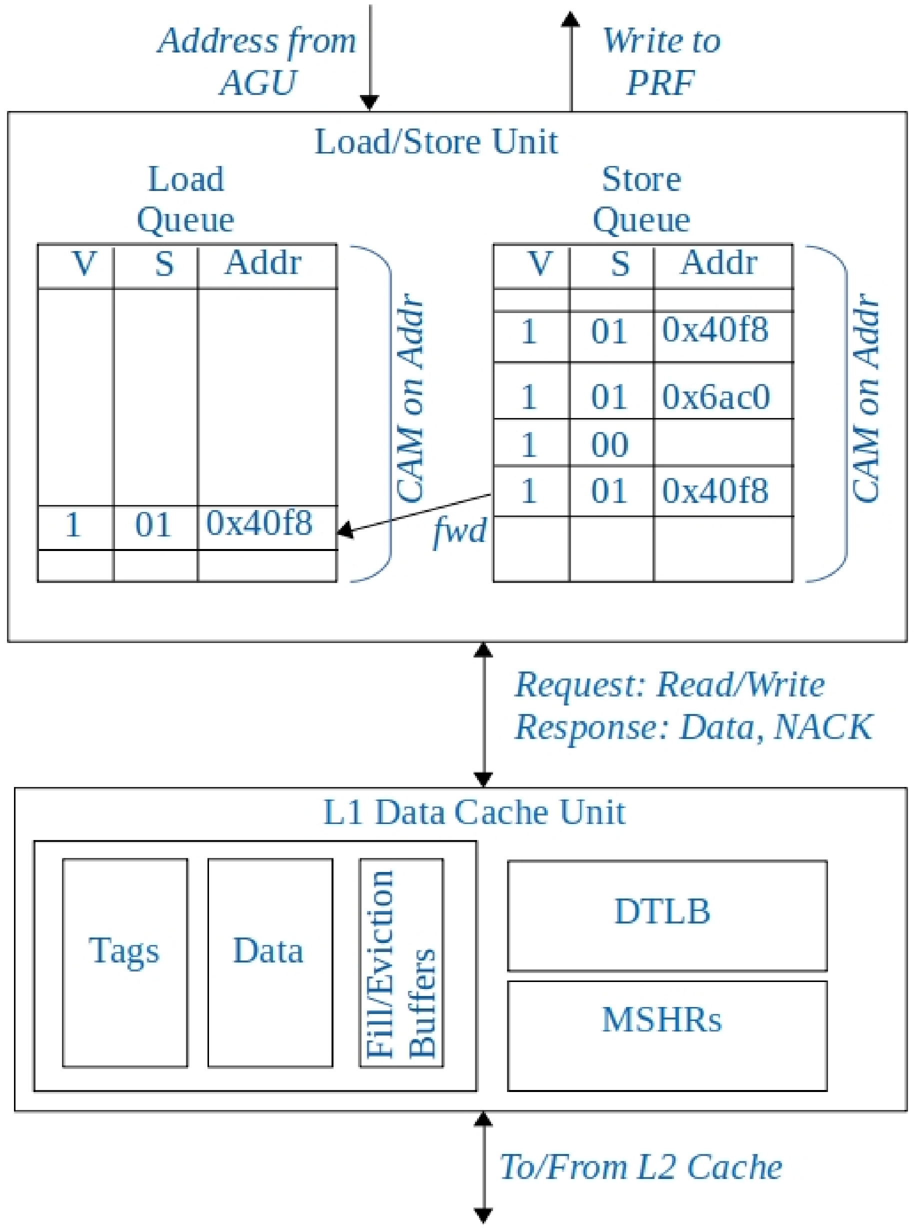
- Unlike arithmetic instructions, Loads and Stores cannot execute as soon as their operands are available! An example of ***load after store*** ordering issue:

Memory operations	Store addr resolves earlier and matches: **Forward** to Load	Store addr resolves earlier and different: **Issue** Load	Non-speculative load's addr resolves earlier: **Wait** for store	Speculative load's addr resolves earlier: **Issue** Load. **Flush** if mismatch detected by store
sw p15, 48[p3]	0x10001024	0x10001664	Addr not ready	Addr not ready
lw p17, 12[p4]	0x10001024	0x10001024	0x10001024	0x10001024

- Our solution for memory disambiguation:
  - Use load queue (LQ) and store queue (SQ) and check for address matches by CAM'ing
  - Allocate LSQ entries at dispatch but send inst info to LSQ only after address generation
  - Loads can either get their value from earlier stores in the SQ or from the D-Cache
  - Only loads marked "speculative" by dependence predictor can bypass older stores
  - Stores forward data to waiting loads in the LQ
  - Detect misspeculation and trigger pipeline flush if load received wrong data
  - Stores always issue to memory in-order at commit

LSU, L1 D-Cache and MMU

- LSU is the only core block that interacts with L1 D-Cache
- 32KB VIPT writeback data cache with 2-cycle access time
- Non-blocking cache supporting multiple outstanding misses with Miss Status Handling Registers (MSHRs)
- Accepts read/write requests from Load-Store Unit
- Responds with requested data or NACK (on MSHR full)
- Full support for fences and all atomic instructions
- Fully associative TLB for address translation
- Hardware page table walk



Register File, Writeback, Commit

Physical Register File (PRF)

- Single large physical register file includes both integer and FP registers, no separate arch file
- PRFs hold both speculative and non-speculative values
- Instructions read operands from PRF after selection
- Currently, PRF has 9 read ports and 4 write ports!
- Splitting the PRF reduces complexity but lowers performance on int-heavy or fp-heavy programs

Writeback/Mem

- Write to PRF (out-of-order) as soon as instructions complete execution
- Send destination register tags for wakeup in IWs
- Write load and store addresses to LSQs

On Commit

- Stores write to caches only at commit
- No regular writes to PRF
- Exception detection and recovery (sequential semantics)
- Updates to free list, branch predictor, checkpoint state, ROB, LSQ

Collaborate/Work with us!

We are currently working on:

- Implementing atomics
- Memory dependence prediction
- Instruction Window/Scheduler optimizations
- Implementation of some functional units
- Performance analysis/projections
- Optimizations to meet first-cut target frequency: 1 Ghz on 22nm

Starting soon:

- Better branch prediction
- Op-fusion, Loop buffer in decode
- Low complexity issue windows, speculative wakeup, split PRF
- Prefetchers – Instruction and Data
- Unified L2 cache with coherence
- Multithreading

Backup Slides

I-Class: Major Blocks and Structures

- Branch Prediction (Gshare predictor, BTB, RAS, BLB)
- Instruction TLB and I-Cache
- Instruction Fetch
- Decode and early pipeline re-direct
- Register Renaming and Checkpointing (Map tables, Free Lists, Backups)
- Dispatch and Allocate
- Instruction Windows (I.IW, F.IW)
- Reorder Buffer (ROB)
- Functional Units (Integer ALU, Int Multiply, Int Divide and Floating Point units)
- Load/Store Queues (LQ, SQ), Dependence Predictor and Memory Disambiguation
- Physical Register File (Register Read and Writeback)
- Data TLB and L1 D-Cache
- Unified L2 Cache
- Instruction Commit

I-Class: Functional Units (version 1.0)

- Integer 1-cycle ops (ALU/AGU/BU)
  - Simple arithmetic, add, sub, shifts, logical, address generation, branch unit etc.
- Integer multiply - pipelined
- Integer divide – non-pipelined
- Floating point conversion
  - SP/DP conversion
  - Int/Float conversion
- FP Add/Sub, FP Mul, FMAC - pipelined
- FP Div/Sqrt - non-pipelined