Quiz 3 Review

Axel Feldmann

(slides are from previous semesters)

Quiz 3 logistics

- Time: 1pm on Wednesday, December 11
  - In-class quiz
- Usual rules (no calculators, closed book)

Topics

- Microcoded and VLIW processors
- Vector processors and GPUs
- Transactional memory
- Accelerators
- Security

Microcoded processors

- Introduces a layer of interpretation
  – Each ISA instruction is executed as a sequence of simpler microinstructions

- Pros:
  – Enables simpler hardware
  – Enables more flexible ISA
- Cons:
  – Sacrifices performance

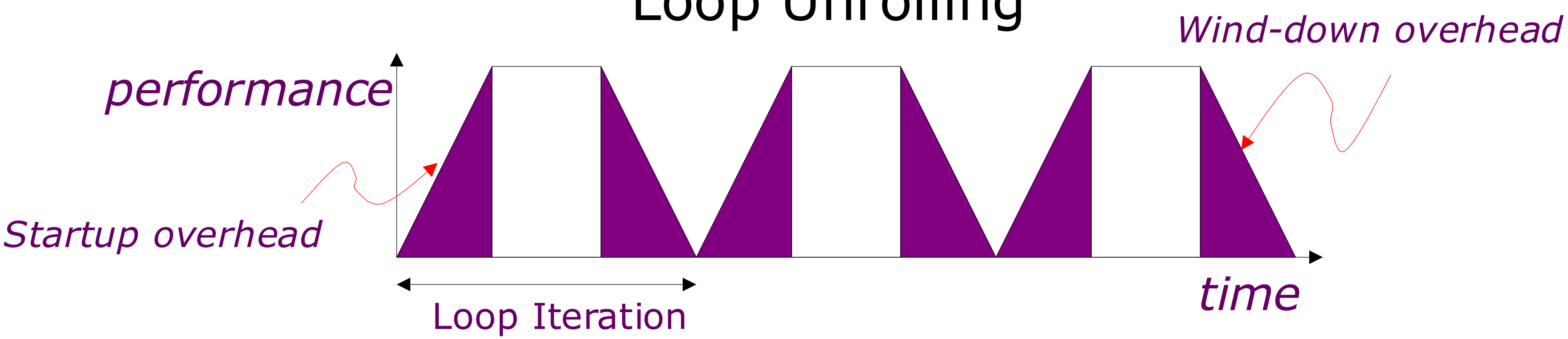
VLIW: Very Long Instruction Word

- The compiler:
  - Guarantees intra-instruction parallelism
  - Schedules (reorders) to maximize parallel execution
- The architecture:
  - Allows operation parallelism within an instruction
    - No cross-operation RAW check
  - Provides deterministic latency for all operations

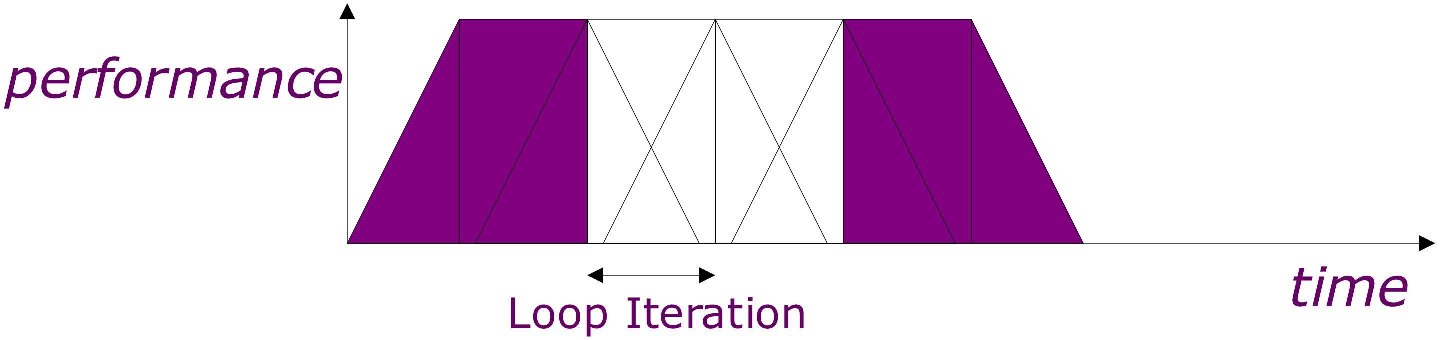
- Enables simple hardware but leaves hard tasks to software

Software pipelining vs. Unrolling

Loop Unrolling

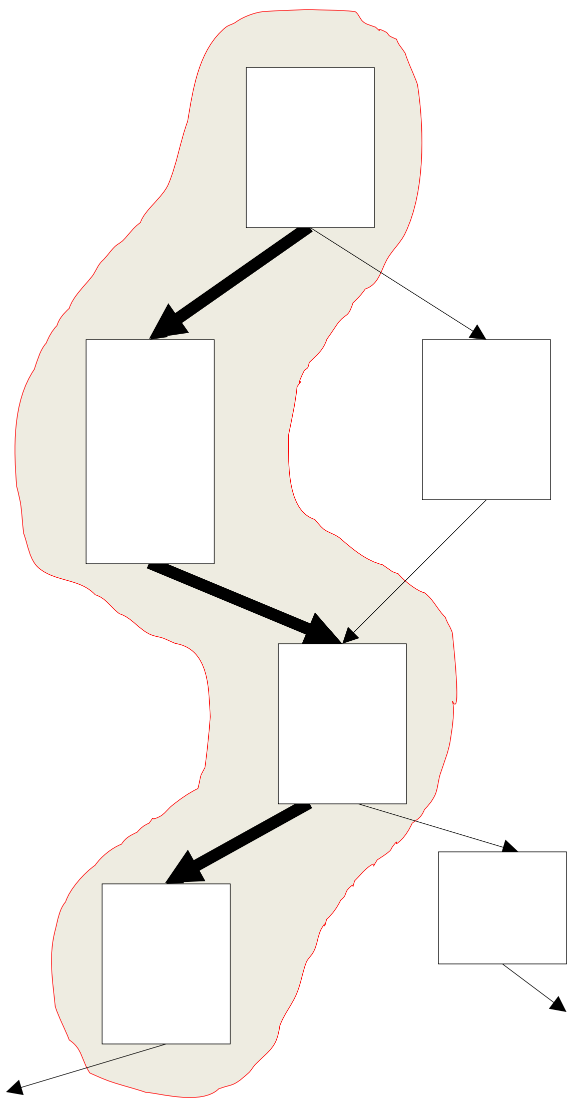


Software Pipelining



Software pipelining pays startup/wind-down costs only once per loop, not once per iteration

Trace scheduling



- Pros
  - Can hoist instructions that come after the branch so that we use VLIW instructions more efficiently
- Cons
  - Compensation path can be expensive

VLIW issues

- Limited by static information
  - Unpredictable branches
    - Possible solution: predicated execution
  - Unpredictable memory operations
    - Possible solution: Memory Latency Register (MLR)

- Code size explosion
  - Wasted slots
  - Replicated code

- Portability
- Compiler complexity

Vector processing

- Supercomputers in 70s – 80s
- Multimedia/SIMD extensions in current ISAs

- Single-Instruction Multiple-Data (SIMD)

- Typical hardware implications
  – Simpler instruction fetch due to fewer instructions
  – Banked register files/memory due to simple access patterns

Vector processing

- Vector chaining
- Vector stripmining
- Vector scatter/gatter
- Masked vector instructions

Example: Masks

Problem: Want to vectorize loops with conditional code:

```
for (i = 0; i < N; i++)
    if (A[i] > 0) then
        A[i] = B[i];
```

Solution: Add vector *mask* (or *flag*) registers

- vector version of predicate registers, 1 bit per element

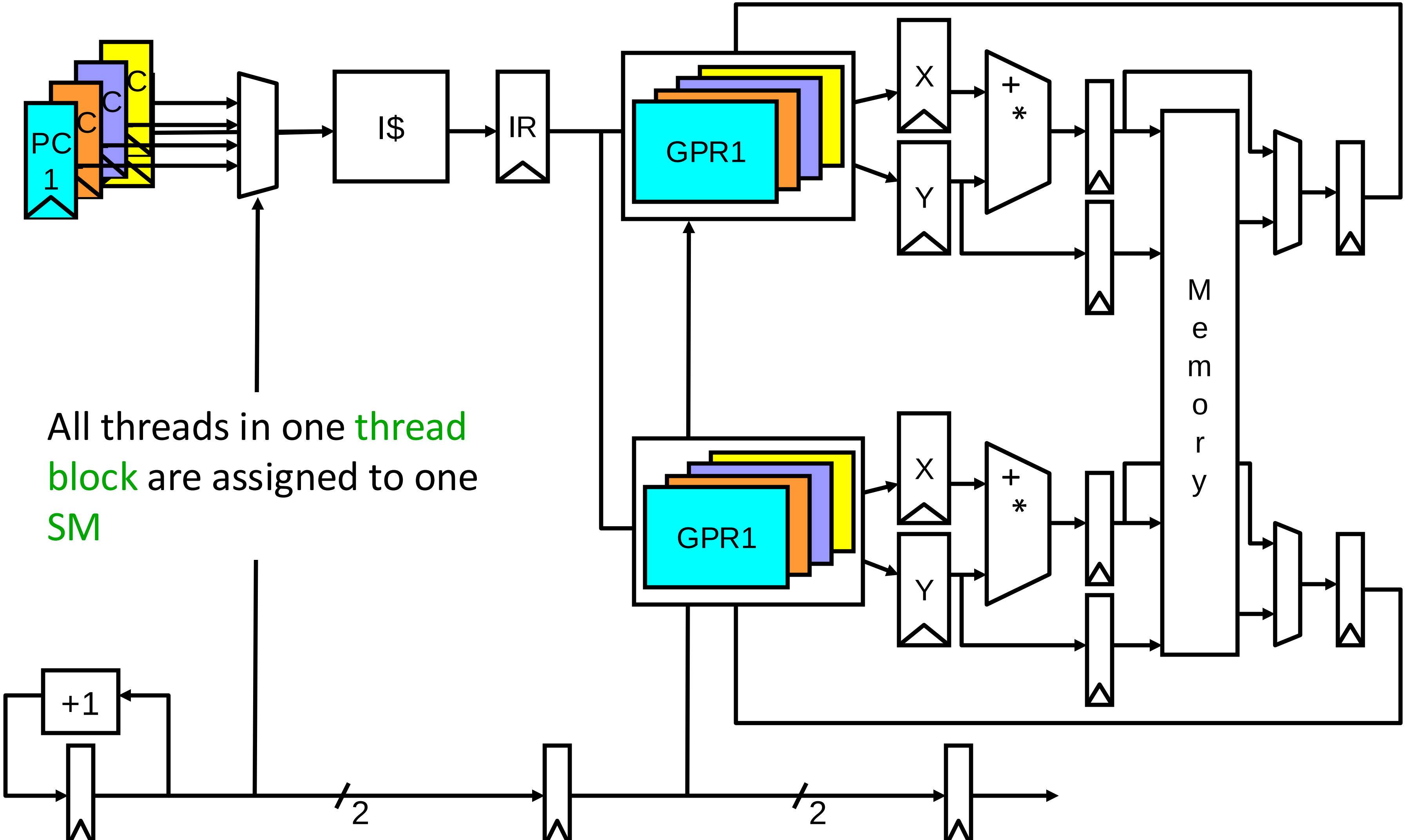
…and *maskable* vector instructions

- vector operation becomes NOP at elements where mask bit is clear

Code example:

```
CVM              # Turn on all elements
LV vA, rA        # Load entire A vector
SGTVS.D vA, F0   # Set bits in mask register where A>0
LV vA, rB        # Load B vector into A under mask
SV vA, rA        # Store A back to memory under mask
```

GPU pipeline



All threads in one thread block are assigned to one SM

GPU memory system

- Memory types (with different scopes)
  - Per-thread memory
  - Scratchpad shared memory
  - Global memory
- Memory primitives: gathers and scatters
- Efficient code requires reducing conflicts

GPU caches

- Goal: saving bandwidth instead of reducing latency
  - Also enables data compression

- Allows flexible and power-efficient designs

Transactional memory

- Use speculation to provide atomicity and isolation without losing concurrency
- Properties of transactions
  - Atomicity (all or nothing)
  - Isolation
  - Serializability
- Declarative synchronization
- System implements synchronization

Advantages of TM

- Easy-to-use synchronization
- High performance
- Composability

TM implementation

- Choices
  - Hardware transactional memory (HTM)
  - Software transactional memory (STM)
  - Hybrid transactional memory

- Basic implementation
  - Version management
  - Conflict detection
  - Conflict resolution

Version management

- Eager versioning
  - Undo-log based
  - Fast commits and slow aborts

- Lazy versioning
  - Write-buffer based
  - Slow commits and fast aborts

Conflict detection

- Read-write and write-write conflicts

- Pessimistic detection
  - Checks during loads/stores
  - Typical resolution: requester wins/stalls
  - Detects conflicts early
  - Requires more to guarantee forward progress

- Optimistic detection
  - Checks when attempting to commit
  - Typical resolution: committer wins
  - Guarantees forward progress (still has fairness issues)
  - Detects conflicts late

HTM implementation

- Version management: use caches
  - Caching write-buffer or undo-log
  - Tracking read-set and write-set

- Conflict detection: use the cache coherence protocols

- Pros:
  - Low implementation overheads
  - Simplifies consistency
- Cons:
  - Performance pathologies
  - Capacity limitations
  - Interaction with Irrevocable execution
  - …

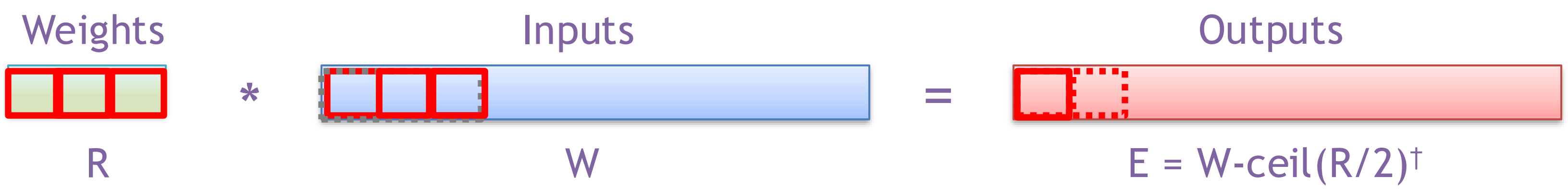
Accelerators

- Why are they useful?
  - Use limited number of transistors more efficiently
  - Trade-off of flexibility vs. efficiency

Accelerators

- Dataflow
  - Mainly categorized by type of reuse
  - Output/Input/Weight stationary
- Sparsity
  - Format
  - Gating
  - Skipping

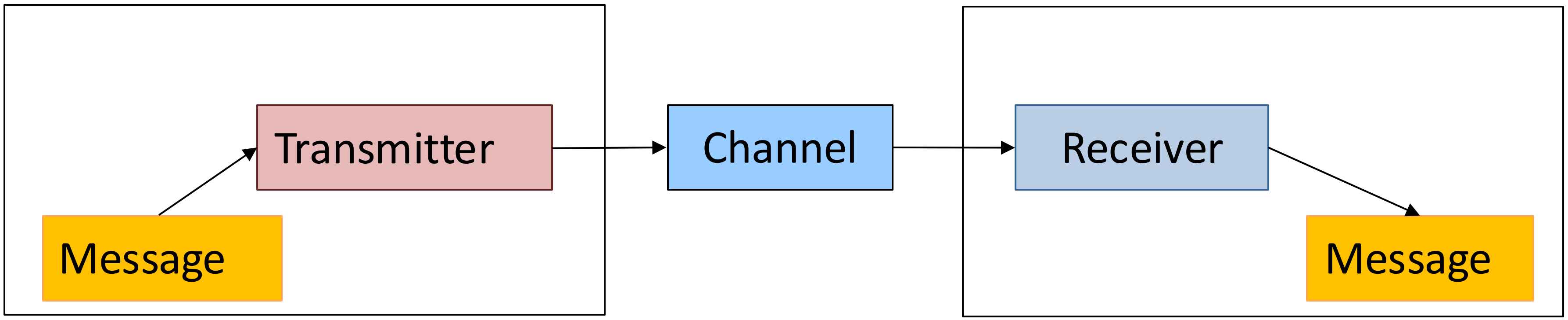
What type of dataflow is this?




```
int i[W];      # Input activations
int f[S];      # Filter weights
int o[Q];      # Output activations

for q in [0, Q):
    for s in [0, S):
        w = q+s
        o[q] += i[w]*f[s];
```

Security



- Transmitter accepts message
- Transmitter modulates channel
- Receiver detects modulation on channel
- Receiver decodes modulation as message.

Security

- Should be able to identify
  - The transmitter & the secret
  - The channel
  - Which part of the code modulates the channel
  - How can the receiver decode the secret
  - Does the receiver need to be active (i.e., does the channel need to be preconditioned)

Wish you all the best!