# Quiz 4 Review

4/7/2016

Section 11

Colin Schmidt

# Agenda

- Quiz Review
  - VLIW
  - Multithreading
  - Vectors(?)
- Problem Set Review

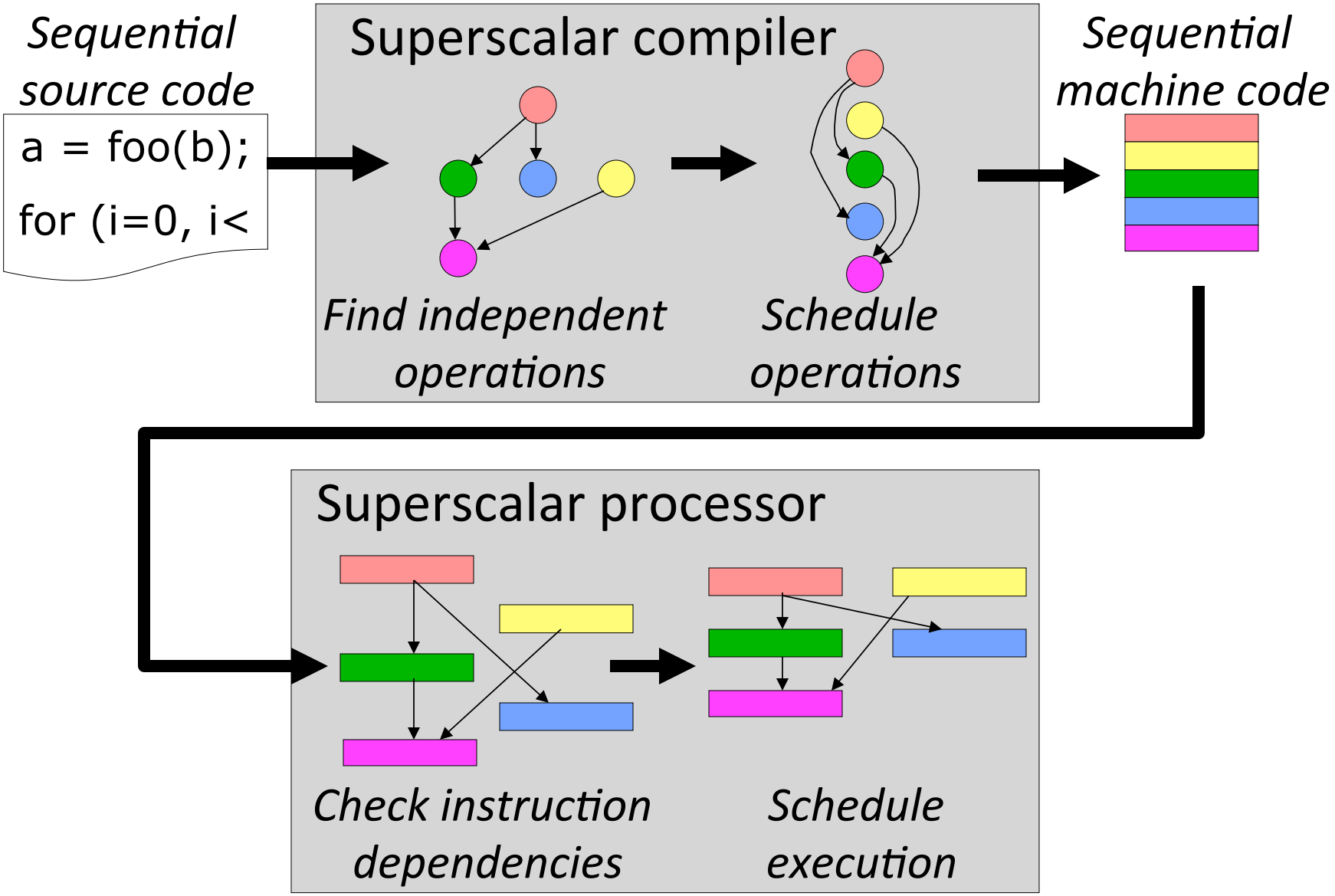
# Parallelism

- Several families 3-4
  - Instruction Level Parallelism
  - Data Level Parallelism
  - TLP
    - Thread Level Parallelism
    - Task Level Parallelism
- How to exploit?

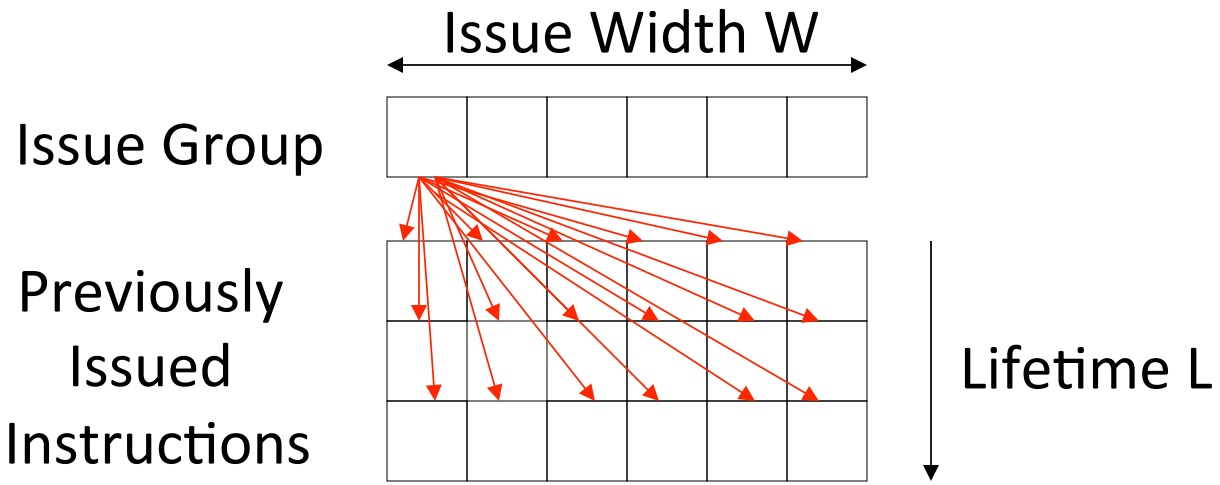
# ILP

- Out-of-order
- Super-scalar
- Both?
- Costs
  - Scheduling

# Sequential ISA Bottleneck



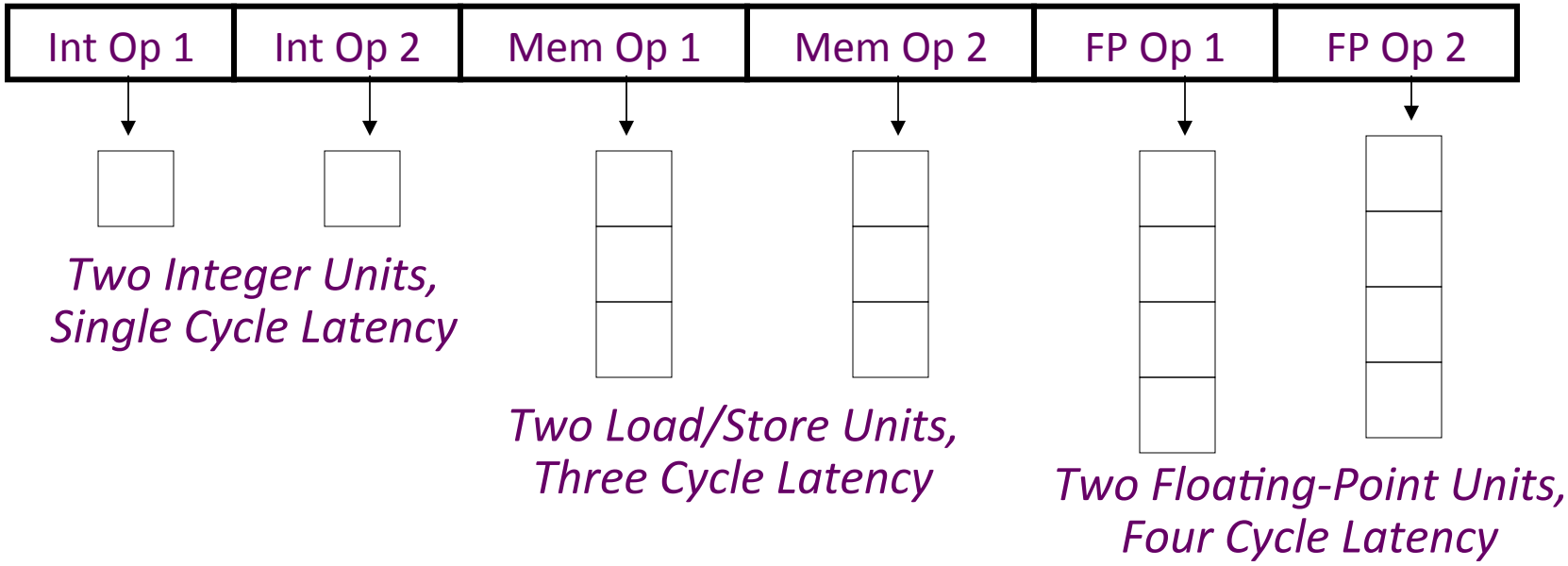
# Superscalar Control Logic Scaling




- Each issued instruction must somehow check against W*L instructions, i.e., growth in hardware ∝ W*(W*L)
- For in-order machines, L is related to pipeline latencies and check is done during issue (interlocks or scoreboard)
- For out-of-order machines, L also includes time spent in instruction buffers (instruction window or ROB), and check is done by broadcasting tags to waiting instructions at write back (completion)
- As W increases, larger instruction window is needed to find enough parallelism to keep machine busy => greater L

*=> Out-of-order control logic grows faster than $W^2$ (~$W^3$)*

# VLIW: Very Long Instruction Word




- Multiple operations packed into one instruction
- Each operation slot is for a fixed function
- Constant operation latencies are specified
- Architecture requires guarantee of:
  - Parallelism within an instruction => no cross-operation RAW check
  - No data use before data ready => no data interlocks

# VLIW Compiler Responsibilities

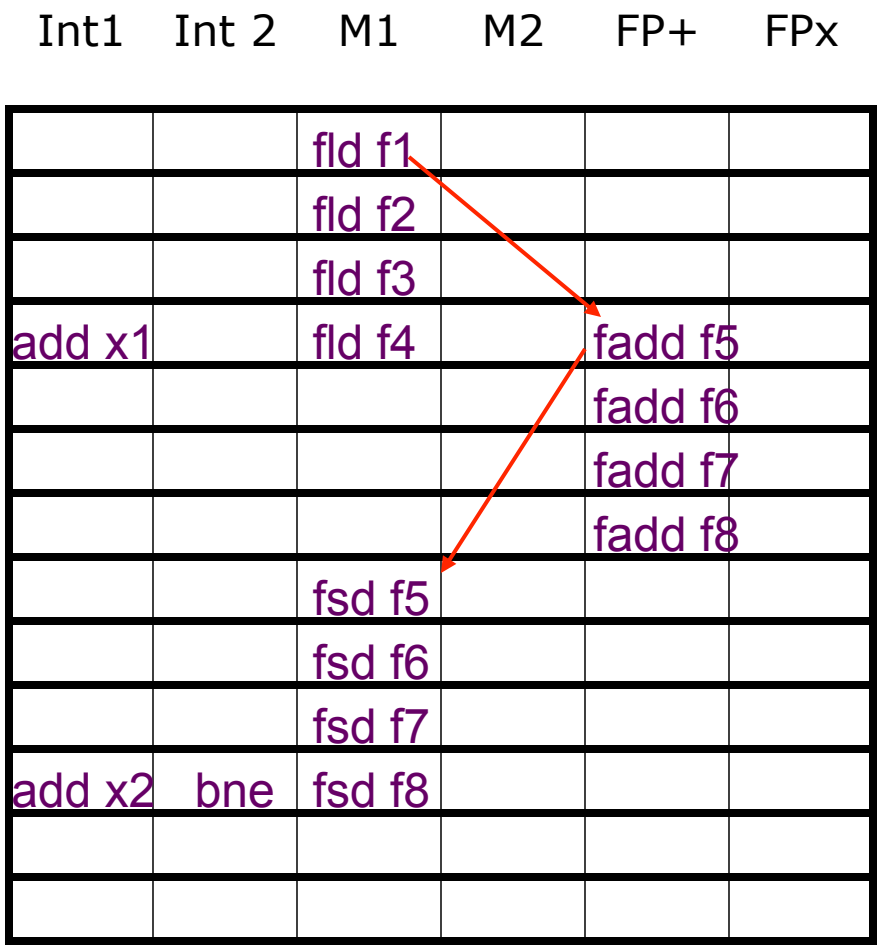
- Schedule operations to maximize parallel execution
- Guarantees intra-instruction parallelism
- Schedule to avoid data hazards (no interlocks)
  - Typically separates operations with explicit NOPs

# Scheduling Loop Unrolled Code

*Unroll 4 ways*

```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       fsd f8, 24(x2)
       add x2, 32
       bne x1, x3, loop
```

*Schedule*



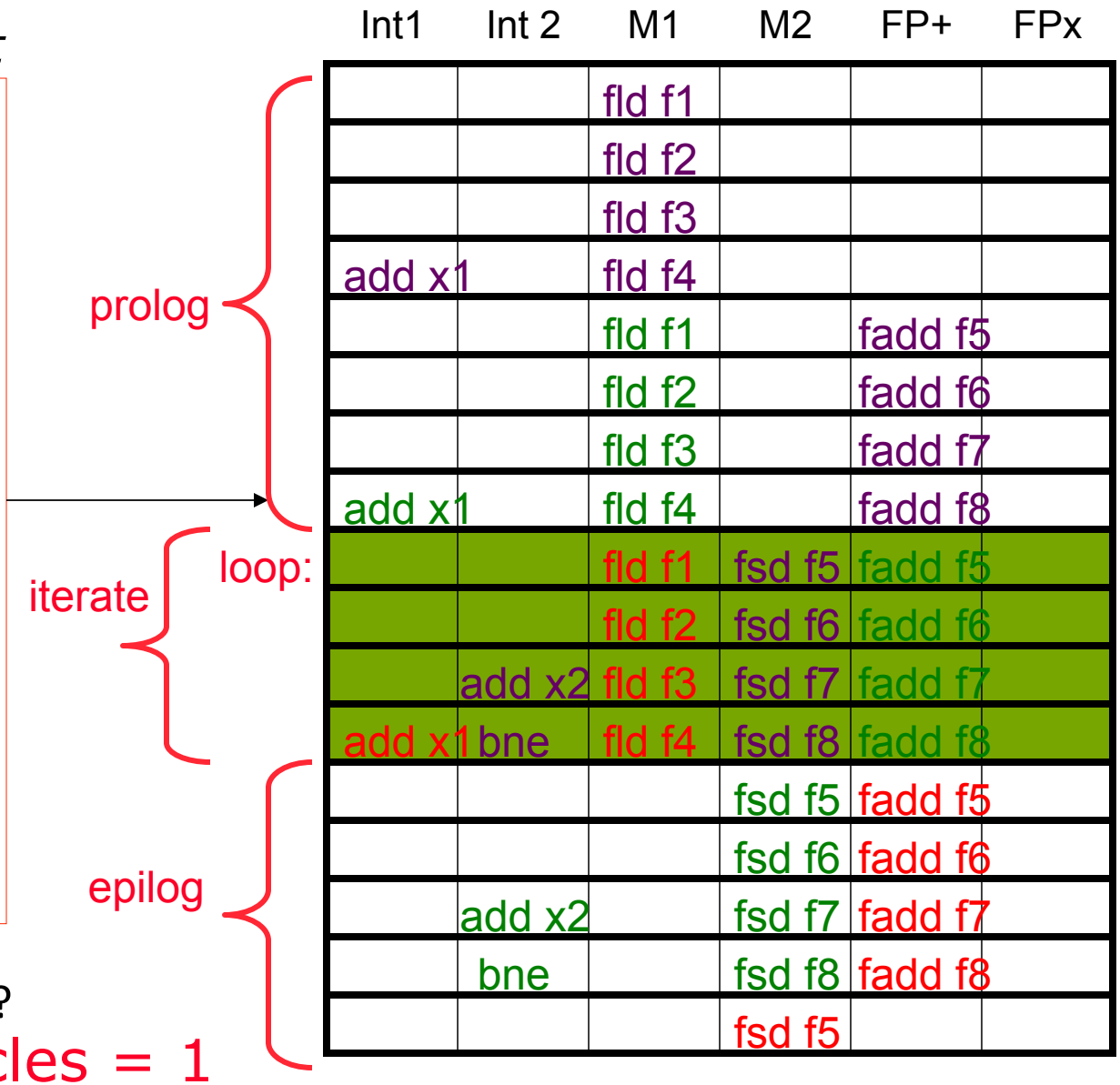

How many FLOPS/cycle?

4 fadds / 11 cycles = 0.36

# Software Pipelining

*Unroll 4 ways first*

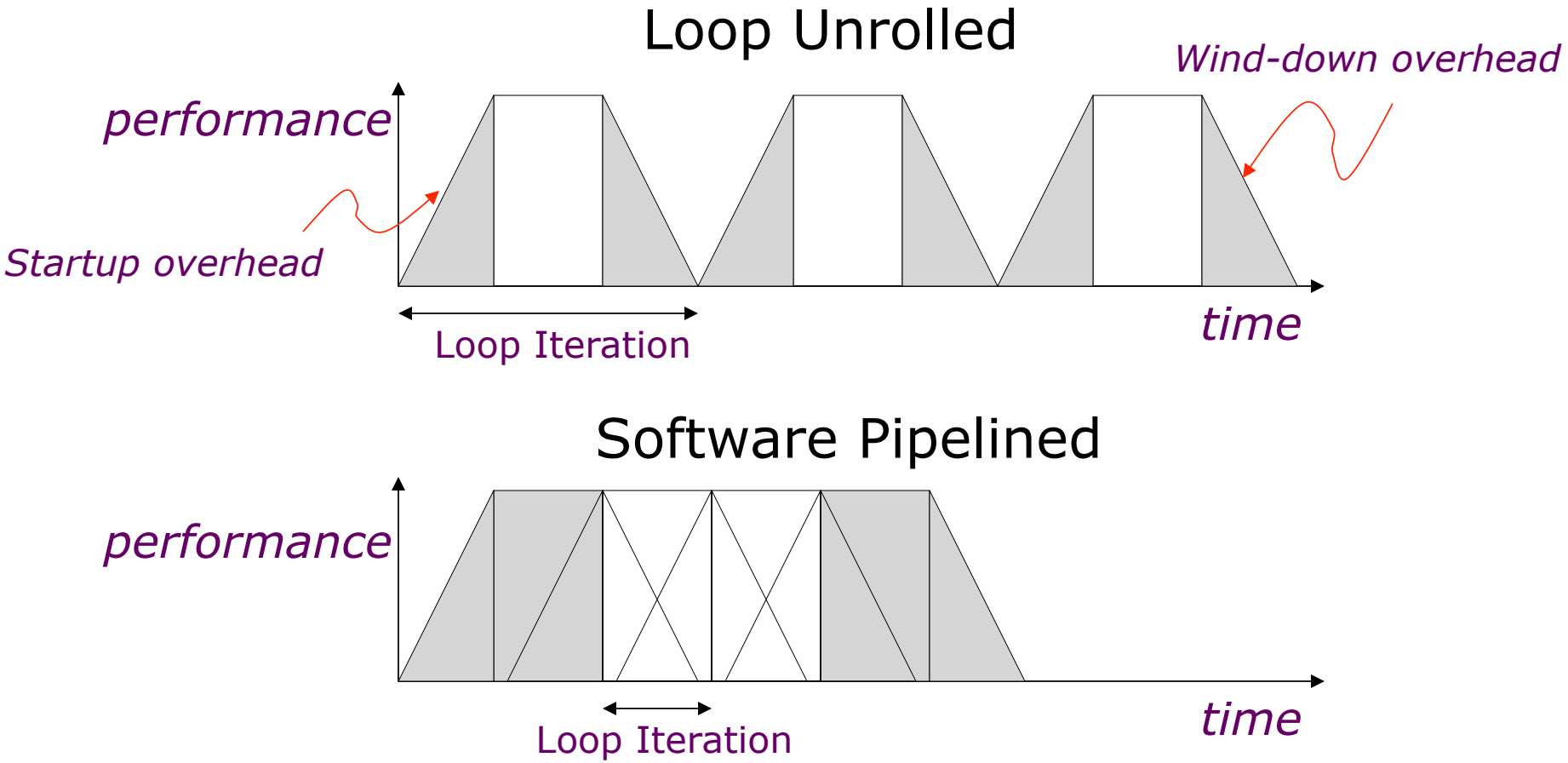
```
loop:  fld f1, 0(x1)
       fld f2, 8(x1)
       fld f3, 16(x1)
       fld f4, 24(x1)
       add x1, 32
       fadd f5, f0, f1
       fadd f6, f0, f2
       fadd f7, f0, f3
       fadd f8, f0, f4
       fsd f5, 0(x2)
       fsd f6, 8(x2)
       fsd f7, 16(x2)
       add x2, 32
       fsd f8, -8(x2)
       bne x1, x3, loop
```




How many FLOPS/cycle?

4 fadds / 4 cycles = 1

# Software Pipelining vs. Loop Unrolling




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

# Problems with "Classic" VLIW

- Object-code compatibility
  - have to recompile all code for every machine even if differences are slight (e.g., latency of one functional unit)
- Object code size
  - instruction padding wastes instruction memory/cache
  - loop unrolling/software pipelining replicates code
- Scheduling variable latency memory operations
  - caches and/or memory bank conflicts impose statically unpredictable variability
- Knowing branch probabilities
  - Profiling requires an significant extra step in build process
- Scheduling for statically unpredictable branches
  - optimal schedule varies with branch path
  - i.e., the result of a branch can affect how to schedule instructions before the branch

# Limits of Static Scheduling

- Unpredictable branches
- Variable memory latency (unpredictable cache misses)
- Code size explosion
- Compiler complexity

Despite several attempts, VLIW has failed in general-purpose computing arena (so far).

- More complex VLIW architectures close to in-order superscalar in complexity, no real advantage on large complex apps

Successful in embedded DSP market

- Simpler VLIWs with more constrained environment, friendlier code.

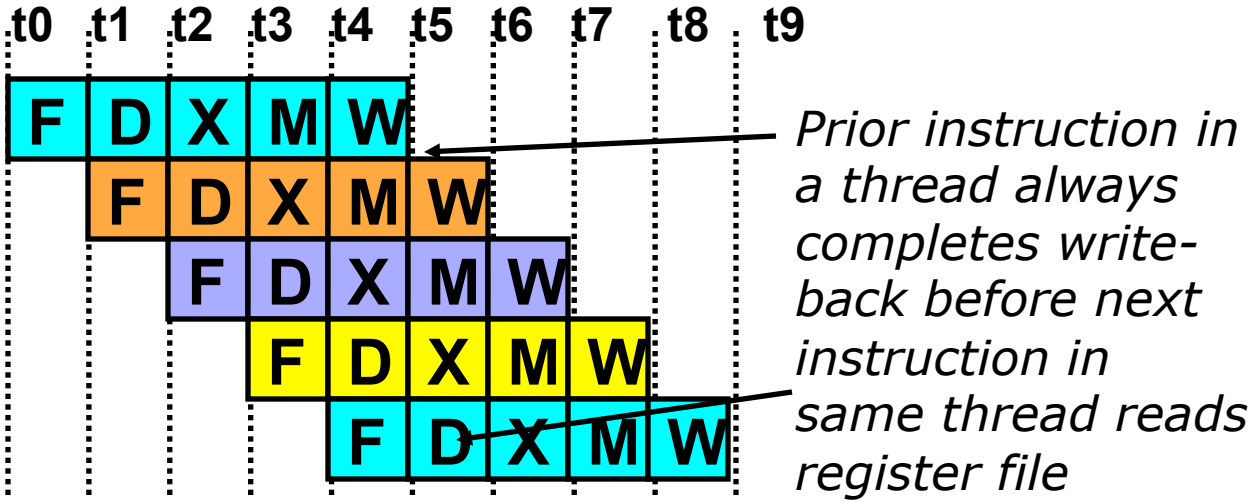
# Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

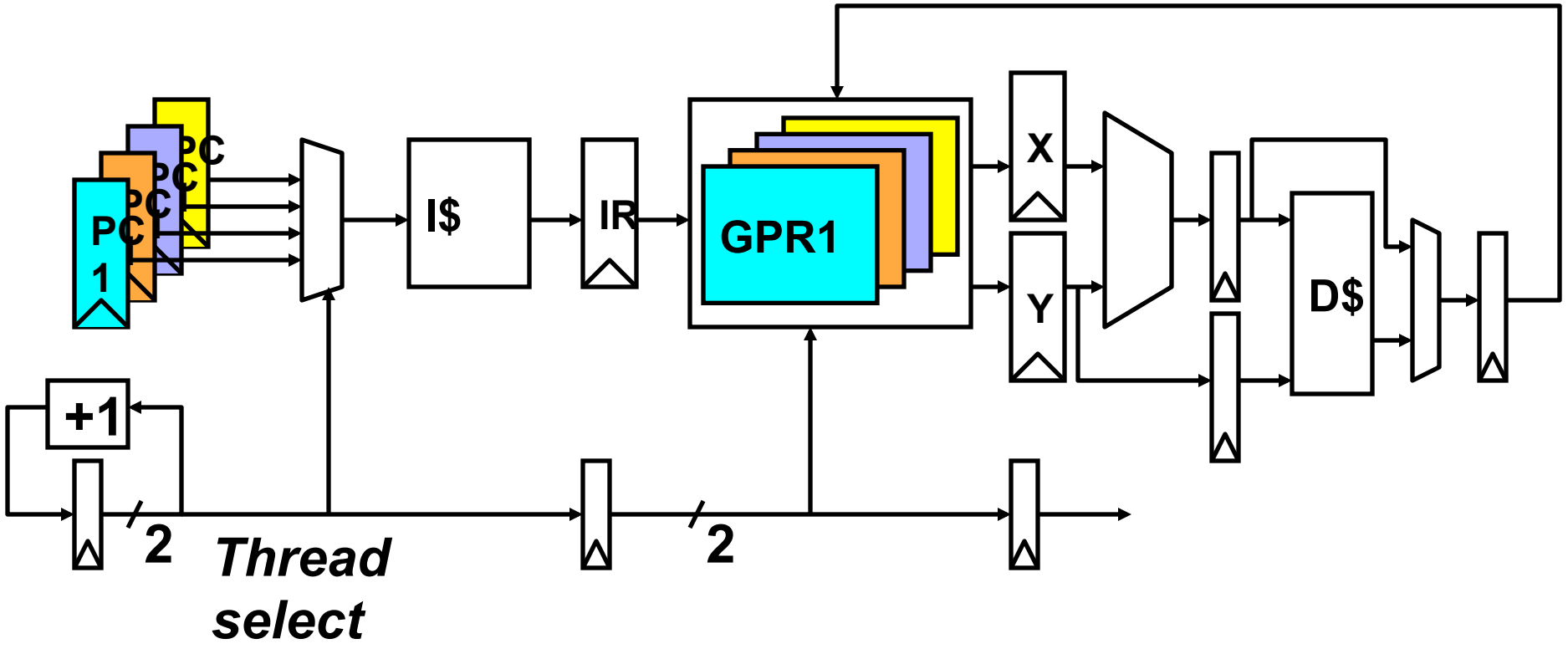
-- One way is to interleave execution of instructions from different program threads on same pipeline

*Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe*

T1: LW r1, 0(r2)
T2: ADD r7, r1, r4
T3: XORI r5, r4, #12
T4: SW 0(r7), r5
T1: LW r5, 12(r1)



# Simple Multithreaded Pipeline




- Have to carry thread select down pipeline to ensure correct state bits read/ written at each pipe stage
- Appears to software (including OS) as multiple, albeit slower, CPUs

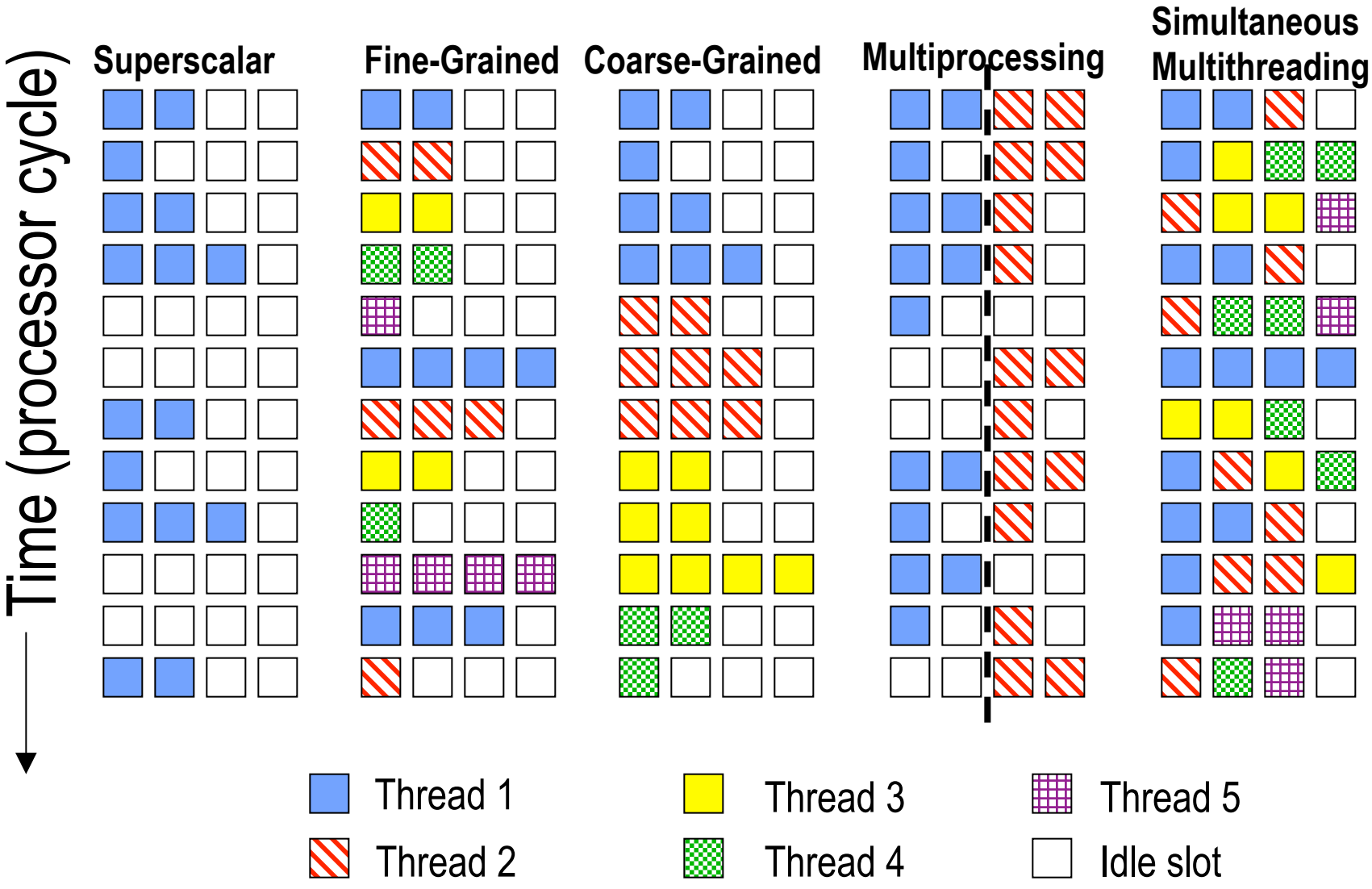
# Multithreading Costs

- Each thread requires its own user state
  - PC
  - GPRs
- Also, needs its own system state
  - Virtual-memory page-table-base register
  - Exception-handling registers
- *Other overheads:*
  - Additional cache/TLB conflicts from competing threads
  - (or add larger cache/TLB capacity)
  - More OS overhead to schedule more threads (where do all these threads come from?)

# Simultaneous Multithreading (SMT) for OoO Superscalars

- Techniques presented so far have all been "vertical" multithreading where each pipeline stage works on one thread at a time
- SMT uses fine-grain control already present inside an OoO superscalar to allow instructions from multiple threads to enter execution on same clock cycle. Gives better utilization of machine resources.

# Summary: Multithreaded Categories



# O-o-O Simultaneous Multithreading
**[Tullsen, Eggers, Emer, Levy, Stamm, Lo, DEC/UW, 1996]**

- Add multiple contexts and fetch engines and allow instructions fetched from different threads to issue simultaneously
- Utilize wide out-of-order superscalar processor issue queue to find instructions to issue from multiple threads
- OOO instruction window already has most of the circuitry required to schedule from multiple threads
- Any single thread can utilize whole machine

# Initial Performance of SMT

- Pentium 4 Extreme SMT yields 1.01 speedup for SPECint_rate benchmark and 1.07 for SPECfp_rate
  - Pentium 4 is dual threaded SMT
  - SPECRate requires that each SPEC benchmark be run against a vendor-selected number of copies of the same benchmark
- Running on Pentium 4 each of 26 SPEC benchmarks paired with every other ($26^2$ runs) speed-ups from 0.90 to 1.58; average was 1.20
- Power 5, 8-processor server 1.23 faster for SPECint_rate with SMT, 1.16 faster for SPECfp_rate
- Power 5 running 2 copies of each app speedup between 0.89 and 1.41
  - Most gained some
  - Fl.Pt. apps had most cache conflicts and least gains