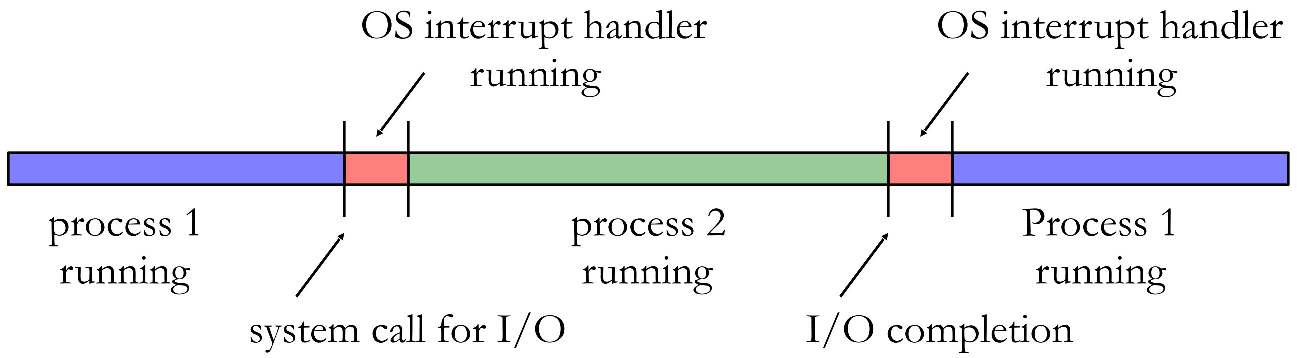
# Lect. 9: Multithreading

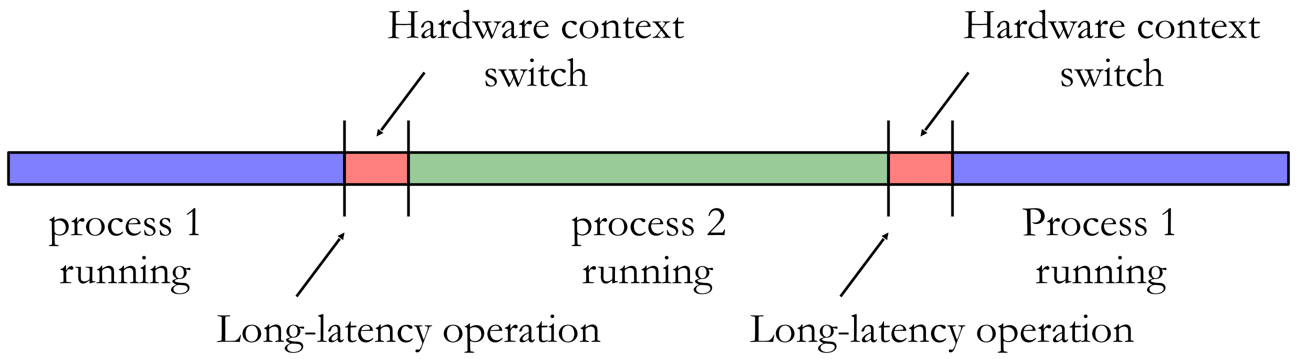
- Memory latencies and even latencies to lower level caches are becoming longer w.r.t. processor cycle times
- There are basically 3 ways to hide/tolerate such latencies by overlapping computation with the memory access
  - Dynamic out-of-order scheduling
  - Prefetching
  - Multithreading
- OOO execution and prefetching allow overlap of computation and memory access within the same thread (these were covered in CS3 Computer Architecture)
- Multithreading allows overlap of memory access of one thread/process with computation by another thread/process

# Blocked Multithreading

- Basic idea:
  - Recall multi-tasking: on I/O a process is context-switched out of the processor by the OS



  - With multithreading a thread/process is context-switched out of the pipeline by the hardware on longer-latency operations

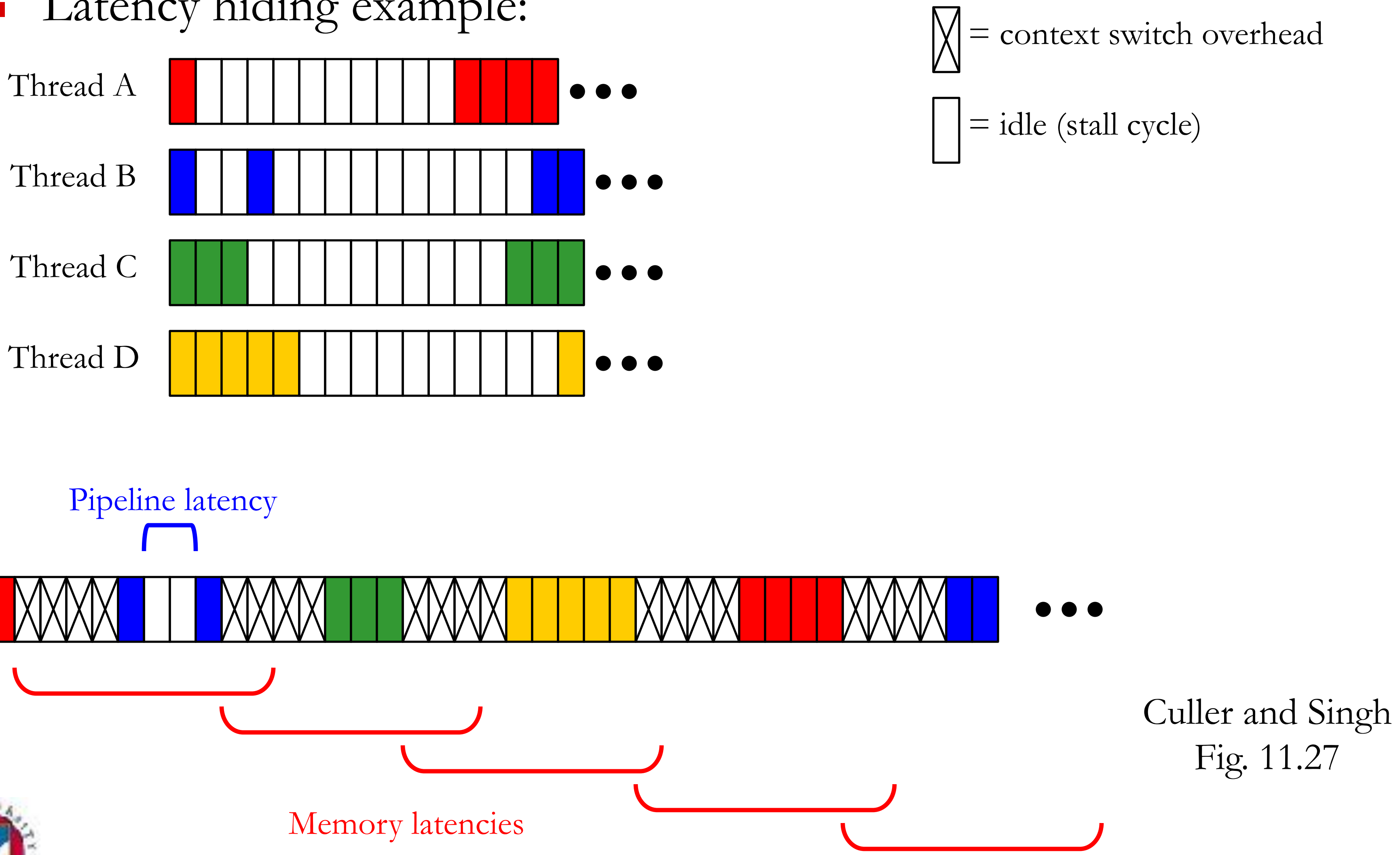


## Blocked Multithreading

- Basic idea:
  - Unlike in multi-tasking, context is still kept in the processor and OS is not aware of any changes
  - Context switch overhead is minimal (usually only a few cycles)
  - Unlike in multi-tasking, the completion of the long-latency operation does not trigger a context switch (the blocked thread is simply marked as ready)
  - Usually the long-latency operation is a L1 cache miss, but it can also be others, such as a fp or integer division (which takes 20 to 30 cycles and is unpipelined)
- Context of a thread in the processor:
  - Registers
  - Program counter
  - Stack pointer
  - Other processor status words
- Note: the term "multithreading" is commonly used to mean simply the fact that the system supports multiple threads

# Blocked Multithreading

- Latency hiding example:



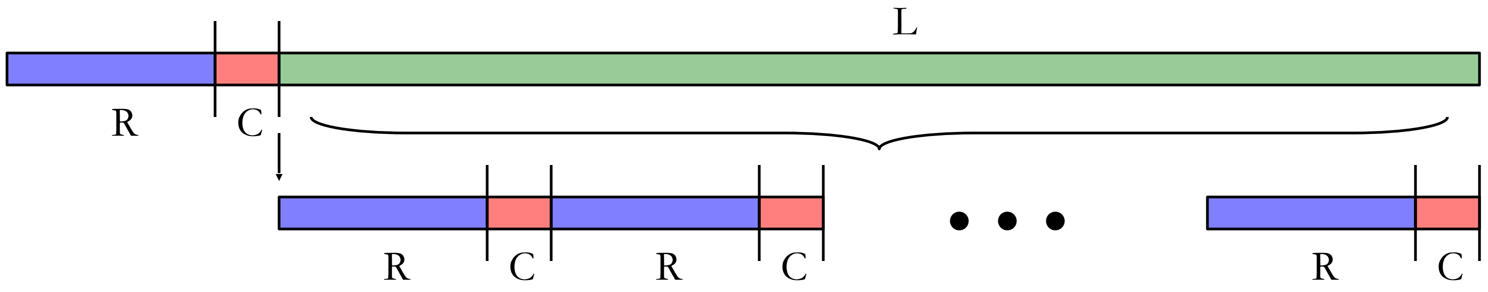
Culler and Singh
Fig. 11.27

# Blocked Multithreading

- Hardware mechanisms:
  - Keeping multiple contexts and supporting fast switch
    - One register file per context
    - One set of special registers (including PC) per context
  - Flushing instructions from the previous context from the pipeline after a context switch
    - Note that such squashed instructions add to the context switch overhead
    - Note that keeping instructions from two different threads in the pipeline increases the complexity of the interlocking mechanism and requires that instructions be tagged with context ID throughout the pipeline
  - Possibly replicating other microarchitectural structures (e.g., branch prediction tables)
- Employed in the Sun T1 and T2 systems (a.k.a. Niagara)

# Blocked Multithreading

- Simple analytical performance model:
  - Parameters:
    - Number of threads (N): the number of threads supported in the hardware
    - Busy time (R): time processor spends computing between context switch points
    - Switching time (C): time processor spends with each context switch
    - Latency (L): time required by the operation that triggers the switch
  - To completely hide all L we need enough N such that (N-1)*R + N*C = L
    - Fewer threads mean we can't hide all L
    - More threads are unnecessary



  - Note: these are only average numbers and ideally N should be bigger to accommodate variation

# Blocked Multithreading

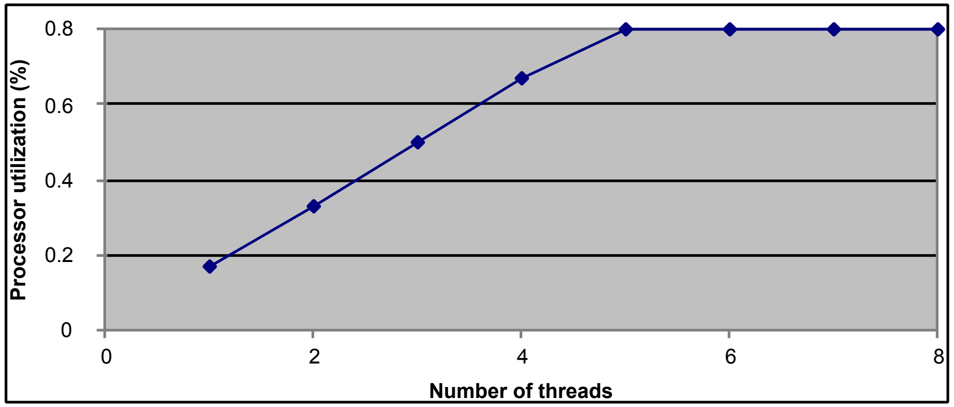
- Simple analytical performance model:
  - The minimum value of N is referred to as the saturation point ($N_{sat}$)

$$N_{sat} = \frac{R + L}{R + C}$$

  - Thus, there are two regions of operation:
    - Before saturation, adding more threads increase processor utilization linearly
    - After saturation, processor utilization does not improve with more threads, but is limited by the switching overhead

$$U_{sat} = \frac{R}{R + C}$$

  - E.g.:
    for R=40,
    L=200,
    and C=10



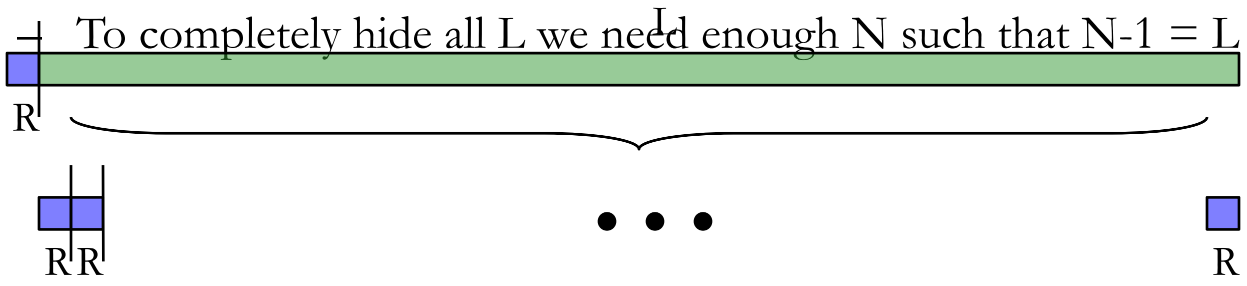
Culler and Singh
Fig. 11.25

# Fine-grain or Interleaved Multithreading

- Basic idea:
  - Instead of waiting for long-latency operation, context switch on every cycle
  - Threads waiting for a long latency operation are marked not ready and are not considered for execution
  - With enough threads no two instructions from the same thread are in the pipeline at the same time → no need for pipeline interlock at all
- Advantages and disadvantages over blocked multithreading:
  - + No context switch overhead (no pipeline flush)
  - + Better at handling short pipeline latencies/bubbles
  - – Possibly poor single thread performance (each thread only gets the processor once every N cycles)
  - – Requires more threads to completely hide long latencies
  - – Slightly more complex hardware than blocked multithreading (if we want to permit multiple instructions from the same thread in the pipeline)
- Some machines have taken this idea to the extreme and eliminated caches altogether (e.g., Cray MTA-2, with 128 threads per processor)

# Fine-grain or Interleaved Multithreading

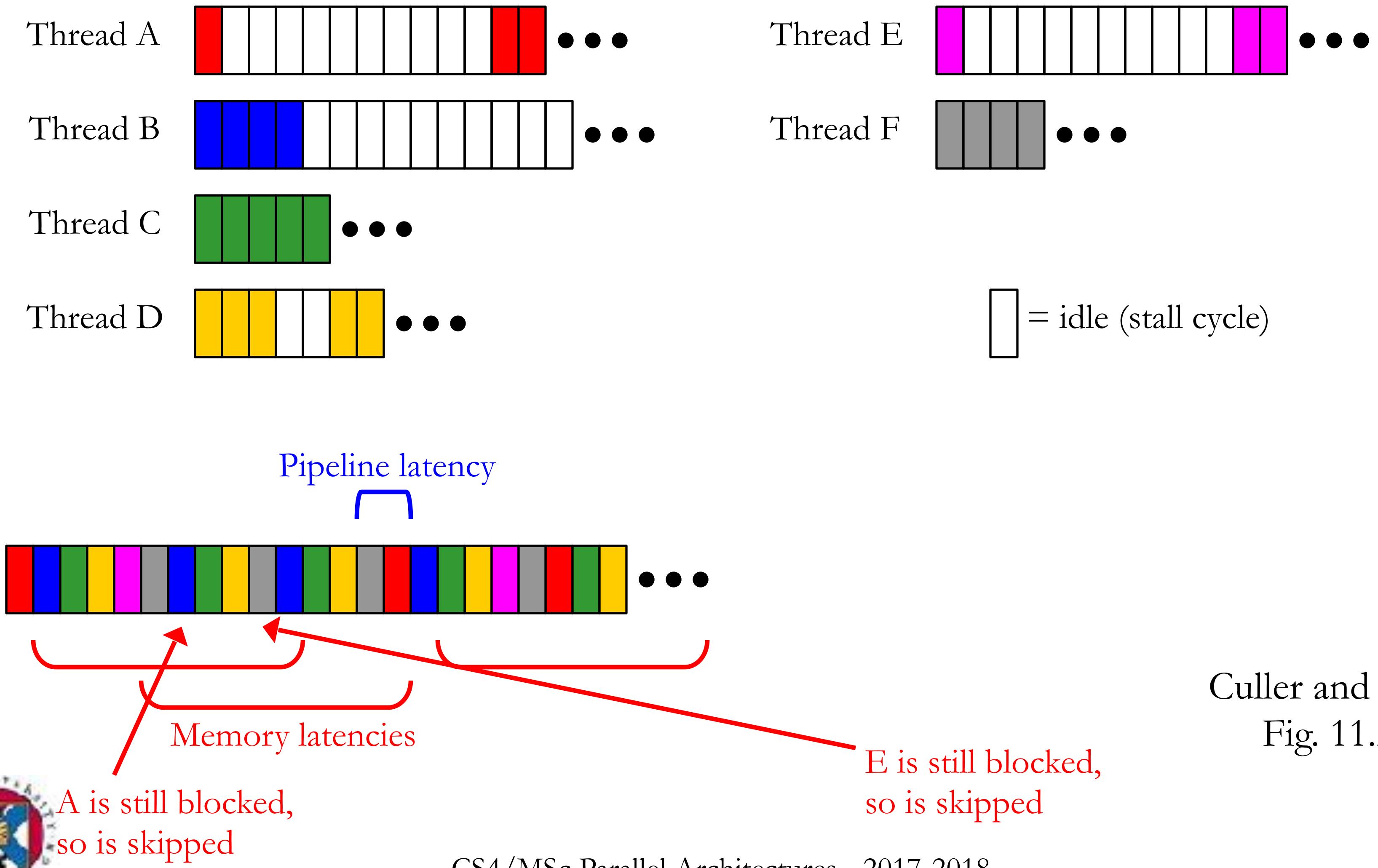
- Simple analytical performance model
- Assumption: no caches, 1 in 2 instruction is a memory access
  - Parameters:
    - Number of threads (N) and Latency (L)
    - Busy time (R) is now 1 and switching time (C) is now 0
  - To completely hide all L we need enough N such that N-1 = L



  - The minimum value of N (i.e., N=L+1) is the saturation point ($N_{sat}$)
  - Again, there are two regions of operation:
    - Before saturation, adding more threads increase processor utilization linearly
    - After saturation, processor utilization does not improve with more threads, but is 100% (i.e., $U_{sat}$ = 1)

# Fine-grain or Interleaved Multithreading

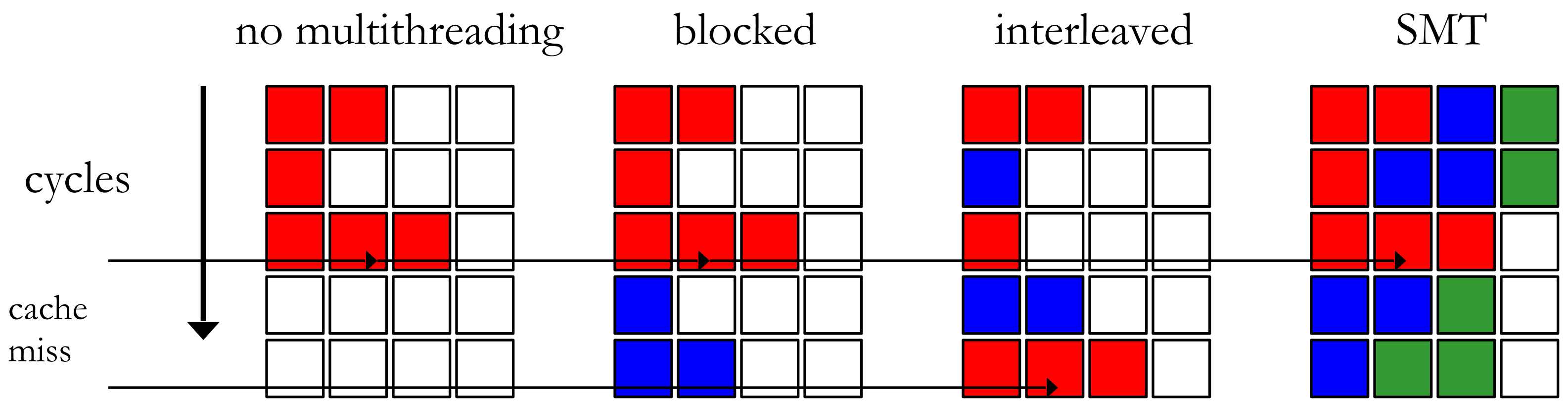
- Latency hiding example:



Culler and Singh
Fig. 11.28

# Simultaneous Multithreading (SMT)

- Basic idea:
  - Don't actually context switch, but on a superscalar processor fetch and issue instructions from different threads/processes simultaneously
  - E.g., 4-issue processor



- Advantages:
  - + Can handle not only long latencies and pipeline bubbles but also unused issue slots
  - + Full performance in single-thread mode
  - – Most complex hardware of all multithreading schemes

# Simultaneous Multithreading (SMT)

- Fetch policies:
  - Non-multithreaded fetch: only fetch instructions from one thread in each cycle, in a round-robin alternation
  - Partitioned fetch: divide the total fetch bandwidth equally between some of the available threads (requires more complex fetch unit to fetch from multiple I-cache lines; see Lecture 3)
  - Priority fetch: fetch more instructions for specific threads (e.g., those not in control speculation, those with the least number of instructions in the issue queue)
- Issue policies:
  - Round-robin: select one ready instruction from each ready thread in turn until all issue slots are full or there or no more ready instructions

    (note: should remember which thread was the last to have an instruction selected and start from there in the next cycle)
  - Priority issue:
    - E.g., threads with older instructions in the issue queue are tried first
    - E.g., threads in control speculative mode are tried last
    - E.g., issue all pending branches first