Review

- Last Class:
  - Tomasulo speculative OOO execution
  - Multi-issue OOO execution
- Today's class:
  - Multi-threading
  - SMT
  - Review of memory hierarchy
- Announcement and reminder
  - HW2 will be distributed on Canvas tonight

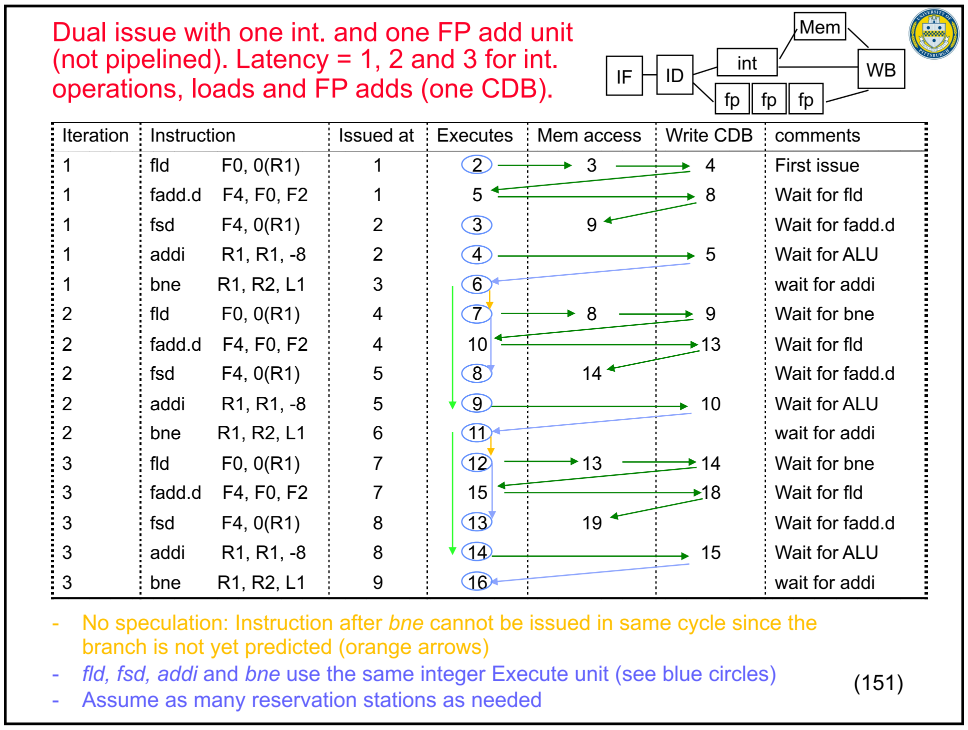
(149)

Dynamic scheduling (§3.8)

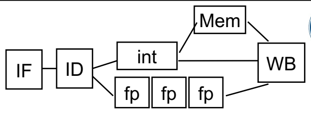
- Extends Tomasulo's algorithm to issue two (or more) instructions simultaneously to reservation stations.
- Either issue an instruction every half clock cycle, or double the logic to handle two instructions at once.
- Use the same logic. Some restrictions may be used to simplify hardware.
  - Example: issue only one FP and one int. operation every clock cycle. This reduces the load on the register files.
  - Do not issue dependent instructions in the same cycle
- To deal with control hazards without speculation (no ROBs): instructions following a branch can be issued but cannot start execution before the branch is resolved.
- Will look at the execution of

```
L1:  fld      F0, 0(R1)
     fadd.d   F4, F0, F2
     fsd      F4, 0(R1)
     addi     R1, R1, -8
     bne      R1, R2, L1
```

(150)



Dual issue with one int. and one FP add unit (not pipelined). Latency = 1, 2 and 3 for int. operations, loads and FP adds (one CDB).

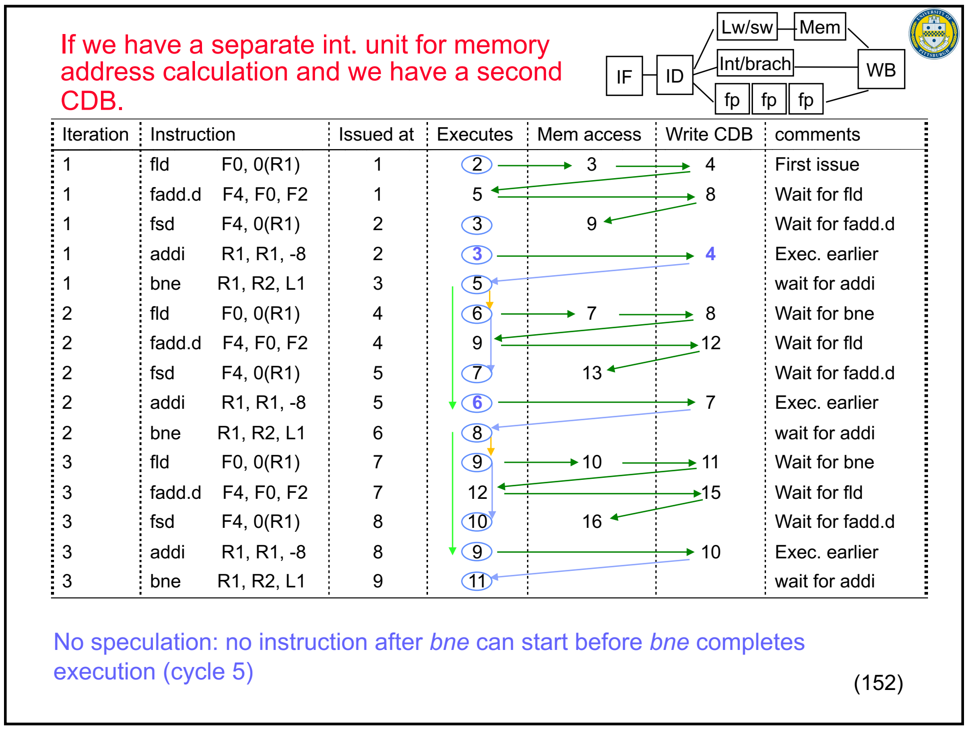


Iteration	Instruction		Issued at	Executes	Mem access	Write CDB	comments
1	fld	F0, 0(R1)	1	2	3	4	First issue
1	fadd.d	F4, F0, F2	1	5		8	Wait for fld
1	fsd	F4, 0(R1)	2	3	9		Wait for fadd.d
1	addi	R1, R1, -8	2	4		5	Wait for ALU
1	bne	R1, R2, L1	3	6			wait for addi
2	fld	F0, 0(R1)	4	7	8	9	Wait for bne
2	fadd.d	F4, F0, F2	4	10		13	Wait for fld
2	fsd	F4, 0(R1)	5	8	14		Wait for fadd.d
2	addi	R1, R1, -8	5	9		10	Wait for ALU
2	bne	R1, R2, L1	6	11			wait for addi
3	fld	F0, 0(R1)	7	12	13	14	Wait for bne
3	fadd.d	F4, F0, F2	7	15		18	Wait for fld
3	fsd	F4, 0(R1)	8	13	19		Wait for fadd.d
3	addi	R1, R1, -8	8	14		15	Wait for ALU
3	bne	R1, R2, L1	9	16			wait for addi

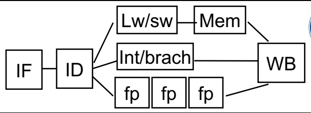
- No speculation: Instruction after *bne* cannot be issued in same cycle since the branch is not yet predicted (orange arrows)
- *fld, fsd, addi* and *bne* use the same integer Execute unit (see blue circles)
- Assume as many reservation stations as needed

(151)

151



If we have a separate int. unit for memory address calculation and we have a second CDB.

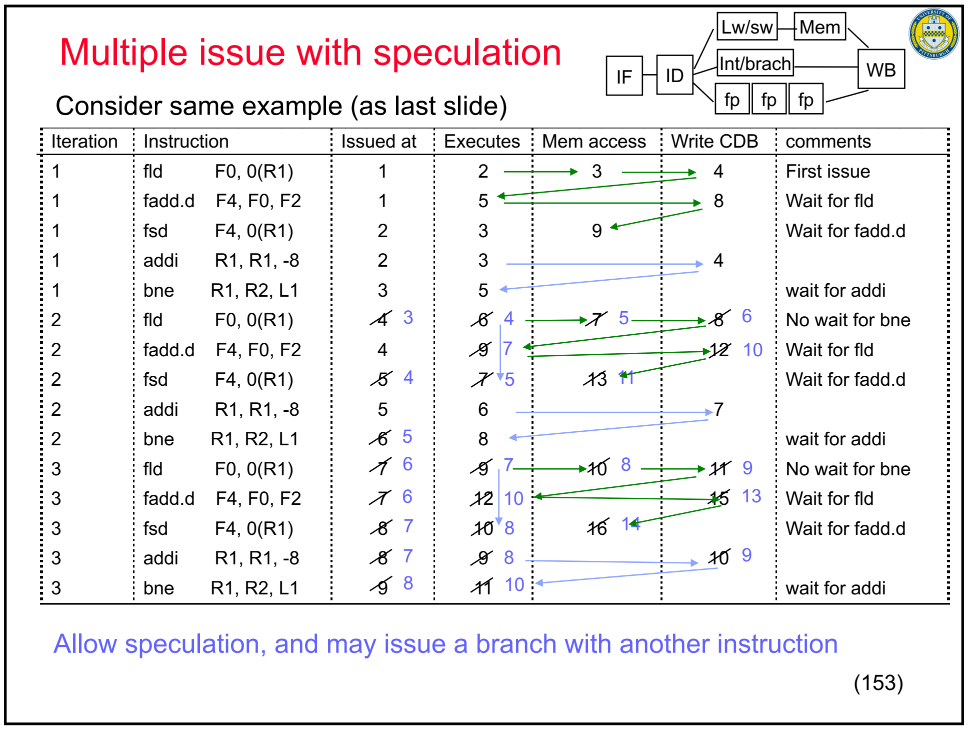


Iteration	Instruction		Issued at	Executes	Mem access	Write CDB	comments
1	fld	F0, 0(R1)	1	2	3	4	First issue
1	fadd.d	F4, F0, F2	1	5		8	Wait for fld
1	fsd	F4, 0(R1)	2	3	9		Wait for fadd.d
1	addi	R1, R1, -8	2	3		4	Exec. earlier
1	bne	R1, R2, L1	3	5			wait for addi
2	fld	F0, 0(R1)	4	6	7	8	Wait for bne
2	fadd.d	F4, F0, F2	4	9		12	Wait for fld
2	fsd	F4, 0(R1)	5	7	13		Wait for fadd.d
2	addi	R1, R1, -8	5	6		7	Exec. earlier
2	bne	R1, R2, L1	6	8			wait for addi
3	fld	F0, 0(R1)	7	9	10	11	Wait for bne
3	fadd.d	F4, F0, F2	7	12		15	Wait for fld
3	fsd	F4, 0(R1)	8	10	16		Wait for fadd.d
3	addi	R1, R1, -8	8	9		10	Exec. earlier
3	bne	R1, R2, L1	9	11			wait for addi

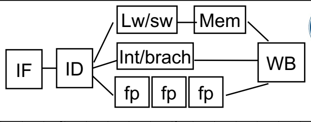
No speculation: no instruction after *bne* can start before *bne* completes execution (cycle 5)

(152)

152



Multiple issue with speculation

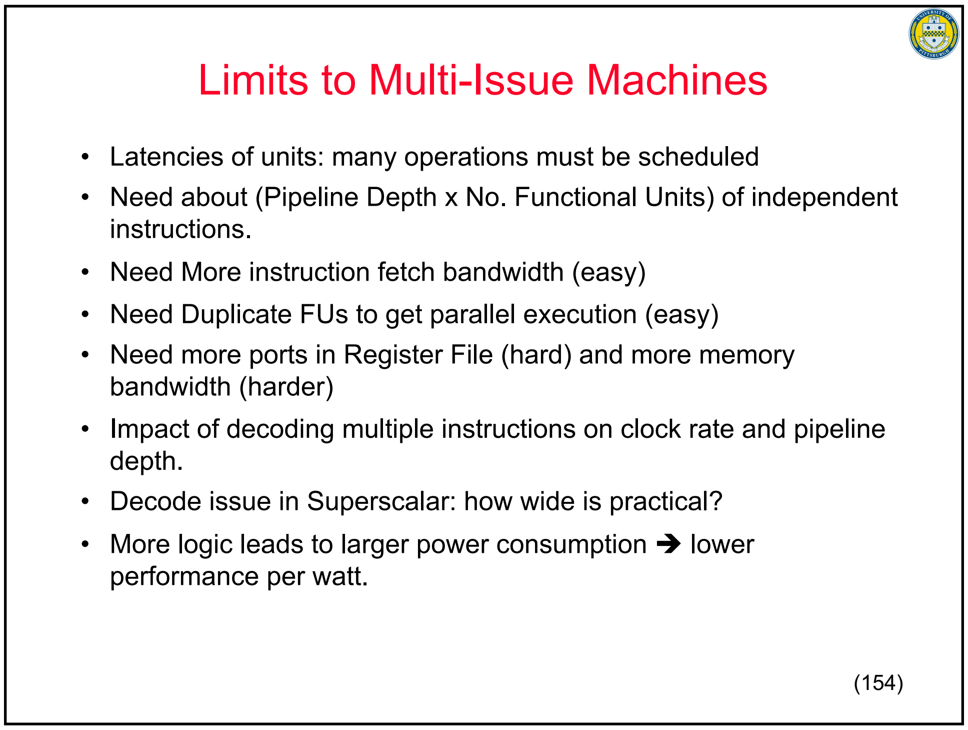


Consider same example (as last slide)

Iteration	Instruction		Issued at	Executes	Mem access	Write CDB	comments
1	fld	F0, 0(R1)	1	2	3	4	First issue
1	fadd.d	F4, F0, F2	1	5		8	Wait for fld
1	fsd	F4, 0(R1)	2	3	9		Wait for fadd.d
1	addi	R1, R1, -8	2	3		4	
1	bne	R1, R2, L1	3	5			wait for addi
2	fld	F0, 0(R1)	~~4~~ 3	~~6~~ 4	~~7~~ 5	~~8~~ 6	No wait for bne
2	fadd.d	F4, F0, F2	4	~~9~~ 7		~~12~~ 10	Wait for fld
2	fsd	F4, 0(R1)	~~5~~ 4	~~7~~ 5	~~13~~ 11		Wait for fadd.d
2	addi	R1, R1, -8	5	6		7	
2	bne	R1, R2, L1	~~6~~ 5	8			wait for addi
3	fld	F0, 0(R1)	~~7~~ 6	~~9~~ 7	~~10~~ 8	~~11~~ 9	No wait for bne
3	fadd.d	F4, F0, F2	~~7~~ 6	~~12~~ 10		~~15~~ 13	Wait for fld
3	fsd	F4, 0(R1)	~~8~~ 7	~~10~~ 8	~~16~~ 14		Wait for fadd.d
3	addi	R1, R1, -8	~~8~~ 7	~~9~~ 8		~~10~~ 9	
3	bne	R1, R2, L1	~~9~~ 8	~~11~~ 10			wait for addi

Allow speculation, and may issue a branch with another instruction

(153)



Limits to Multi-Issue Machines

- Latencies of units: many operations must be scheduled
- Need about (Pipeline Depth x No. Functional Units) of independent instructions.
- Need More instruction fetch bandwidth (easy)
- Need Duplicate FUs to get parallel execution (easy)
- Need more ports in Register File (hard) and more memory bandwidth (harder)
- Impact of decoding multiple instructions on clock rate and pipeline depth.
- Decode issue in Superscalar: how wide is practical?
- More logic leads to larger power consumption ➔ lower performance per watt.

(154)

Return address prediction

- Most unconditional branches come from function returns
- The same procedure can be called from multiple sites
  - Causes the buffer to potentially forget about the return address from previous calls

Integrated instruction Fetch unit that performs

- Branch prediction
- Instruction prefetch

Energy Efficiency

- Speculation is only energy efficient when it significantly improves performance

Value Prediction

- Loads that load from a constant pool
- Instruction that produces a value from a small set of values
- Not incorporated into modern processors

(155)

Dynamically scheduling memory ops

- Compilers must schedule memory ops conservatively
- Options for hardware:
  1. Don't execute any load until all prior stores execute (conservative)
  2. Execute loads as soon as possible, detect violations (aggressive)
     - When a store executes, it checks if any later loads executed too early (to the same address). If so, flush pipeline and restart
  3. Learn violations over time, selectively reorder (predictive)

```
Conservative                Aggressive (Wrong Outcome?)
ld r2,4(sp)                 ld r2,4(sp)
ld r3,8(sp)                 ld r3,8(sp)
add r3,r2→r1                ld r5,0(r8)  //does r8==sp?
st r1,0(sp)                 add r3,r2→r1
ld r5,0(r8) //stall         ld r6,4(r8)  //does r8+4==sp?
ld r6,4(r8)                 st r1,0(sp)
sub r5,r6→r4                sub r5,r6→r4
st r4,8(r8)                 st r4,8(r8)
```

(156)

Store Queue with load bypassing

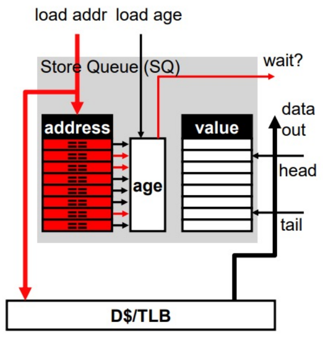
- Loads can be allowed to bypass stores (if no aliasing)
  - Requires checking addresses of older stores
  - Addresses of older stores must be known in order to check
- To implement, use separate load queue (LQ) and store queue (SQ)
  - Think of separate RS for loads and stores
- Need to know the relative order of instructions in the queues – “Age”: new field added to both queues
  - Age represents position of load/store in the program
  - A simple counter incremented during the in-order dispatch

(157)

157

Store Queue with load bypassing

- Loads: for the oldest ready load in LQ, check the “Addr” of older stores in SQ
  - If any store with an uncomputed or matching “Addr”, load cannot issue
  - Check SQ in parallel with accessing D$
- • Stores: can always execute when at ROB head



(158)

158

Memory forwarding

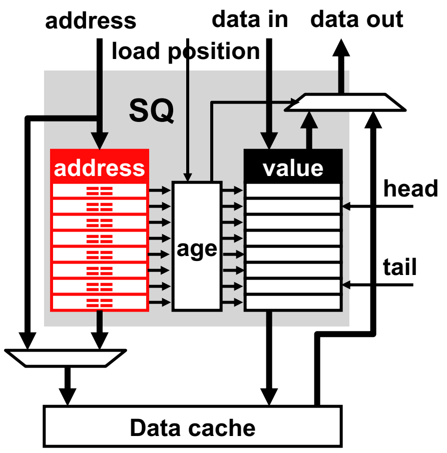
- Stores write cache at Commit (until then hold store value and address in a Store Queue)
  - Commit is in-order, so allows stores to (also) be “undone” on branch mis-predictions, etc.
- Loads read cache
  - Early execution of loads is critical
- Forwarding
  - Allow store to (following) load communication before store Commits
  - Conceptually like register fowarding, but different implementation
    - Why? Addresses unknown until Execute

(159)

159

Store Queue with load forwarding

- Store Queue (SQ)
  - Holds all in-flight stores
  - CAM: searchable by address
  - Age logic: determine *youngest* matching store that is *older* than the load
- Store execution
  - Write into SQ
    - address + data value
  - Store from SQ into D$ at Commit
- Load execution
  - Assoc search SQ addresses
    - Match? Forward youngest matching store value to load request
  - Else read value from D$



(160)

160

Load speculation

- Allow load to proceed when there are undetermined store addresses in the SQ.
- Speculation requires two things ...
  1. Detection of mis-speculations (guessed that memory addresses didn't match and it turns out that they did)
     - How can we do this?
  2. Recovery from mis-speculations
     - Squash instr's after offending load (they may be using the load data)
     - Saw how to squash instr's after mispredicated branches: same method

(161)

LSQ = Store Queue + Load Queue

- Store Queue: handles forwarding
  - Written into by stores (@ store execute)
  - Searched by loads (@ load execute) for store forwarding
  - Write to data cache (@ store Commit)
- Load Queue: detects ordering violations
  - Written into by loads (@ load execute)
  - Searched by stores (@ store execute) to detect load ordering violation
- Both together
  - Allow aggressive load scheduling
    - Stores don't constrain load execution
  - Help us improve performance significantly

(163)

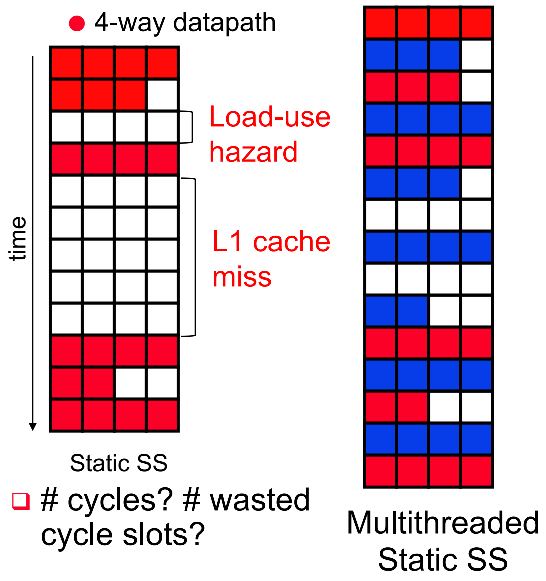
Multi-threading (MT)

- Even moderate static superscalars (e.g., 4-way) are not fully utilized
  - Average sustained IPC: 1.5–2 → < 50% utilization due to
    - Mispredicted branches (branch frequency?)
    - Cache misses, especially L1 (very frequent)
    - Data dependences, load-use data hazards
- Multi-threading (MT) to the rescue
  - Improve utilization of datapath components by multiplexing multiple (process) threads on single datapath
  - If one thread cannot fully utilize the datapath, maybe 2 or 4 (or 100) can

(165)

Multithreading Example

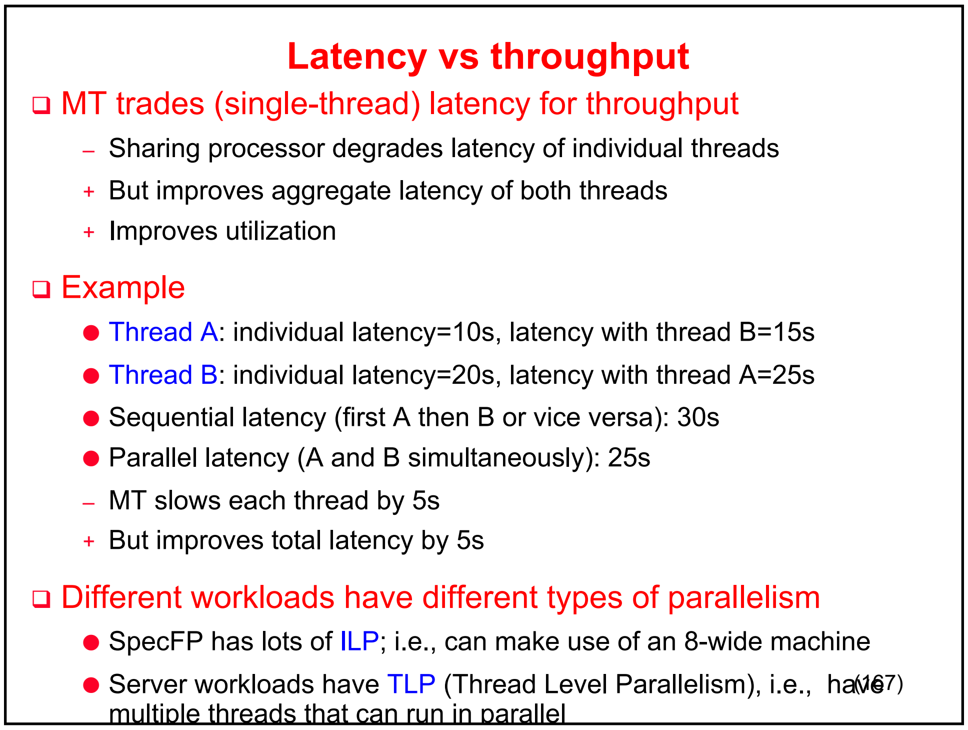
- Time evolution of issue slot
  - 4-way datapath



- # cycles? # wasted cycle slots?

- Fill in with instructions from other threads – in this example we have 2 threads and change threads every cycle
  - Completely removes load-use hazard empty slots
  - Takes longer for the “red” thread to finish
    - With more threads, would take even longer
  - Still have some noop slots (so wasted performance – stay tuned)

(166)

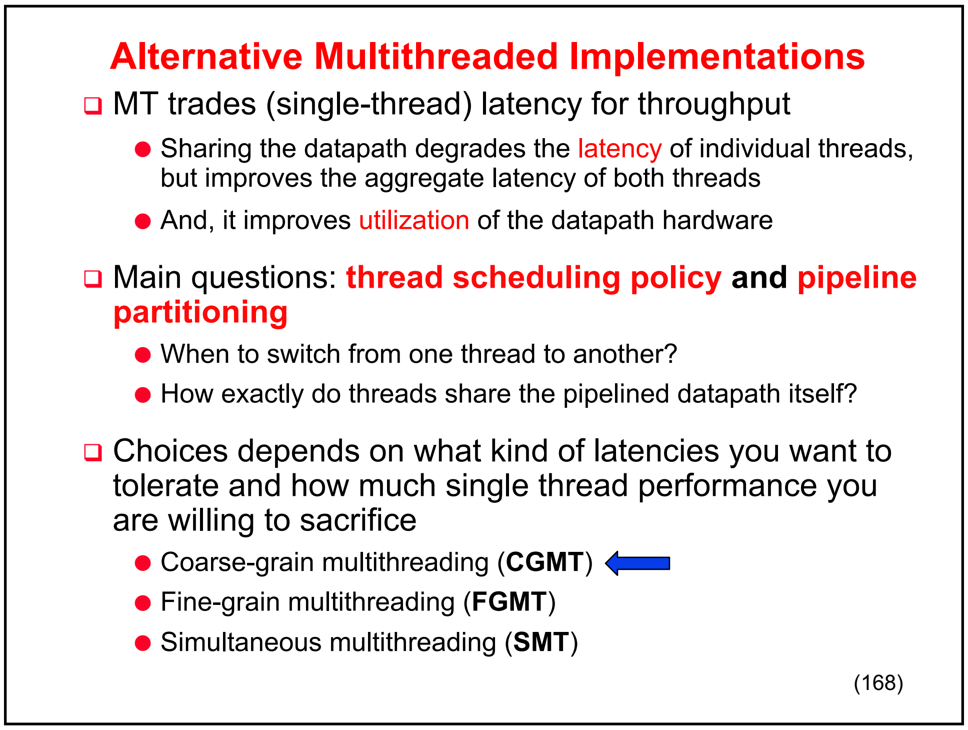


Latency vs throughput

- MT trades (single-thread) latency for throughput
  - – Sharing processor degrades latency of individual threads
  - + But improves aggregate latency of both threads
  - + Improves utilization
- Example
  - Thread A: individual latency=10s, latency with thread B=15s
  - Thread B: individual latency=20s, latency with thread A=25s
  - Sequential latency (first A then B or vice versa): 30s
  - Parallel latency (A and B simultaneously): 25s
  - – MT slows each thread by 5s
  - + But improves total latency by 5s
- Different workloads have different types of parallelism
  - SpecFP has lots of ILP; i.e., can make use of an 8-wide machine
  - Server workloads have TLP (Thread Level Parallelism), i.e., have multiple threads that can run in parallel

(167)

167



Alternative Multithreaded Implementations

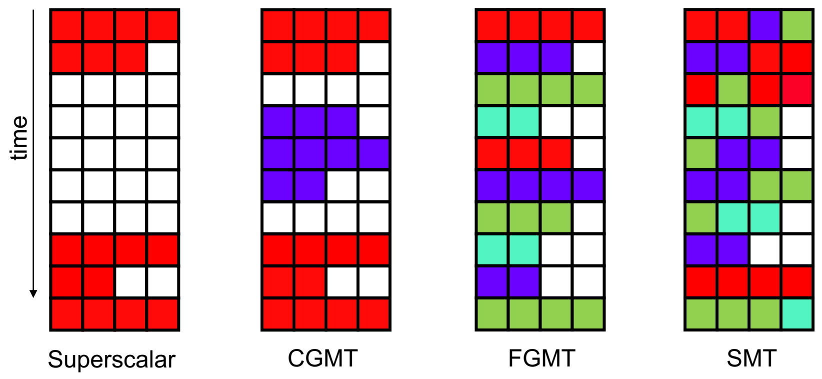
- MT trades (single-thread) latency for throughput
  - Sharing the datapath degrades the latency of individual threads, but improves the aggregate latency of both threads
  - And, it improves utilization of the datapath hardware
- Main questions: **thread scheduling policy** and **pipeline partitioning**
  - When to switch from one thread to another?
  - How exactly do threads share the pipelined datapath itself?
- Choices depends on what kind of latencies you want to tolerate and how much single thread performance you are willing to sacrifice
  - Coarse-grain multithreading (**CGMT**)
  - Fine-grain multithreading (**FGMT**)
  - Simultaneous multithreading (**SMT**)

(168)

168

The standard multithreading pictures

- Time evolution of issue slots
  - Color = thread



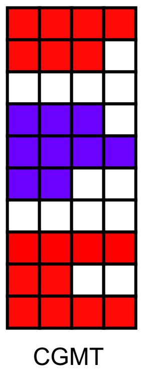
(169)

Coarse-Grain MultiThreading (CGMT)

+ Sacrifices very little single thread performance (of one thread)

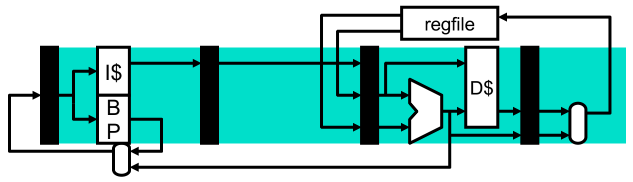
– Tolerates only long latencies (e.g., L2 misses)

- Thread scheduling policy
  - Designate a “preferred” thread (e.g., thread A)
  - Switch to thread B on thread A L2 miss
  - Switch back to A when A L2 miss returns
- Pipeline partitioning
  - None, flush on switch
  - – So can’t tolerate very short latencies
    - - Need short in-order pipeline for good performance
- Example: IBM Northstar/Pulsar

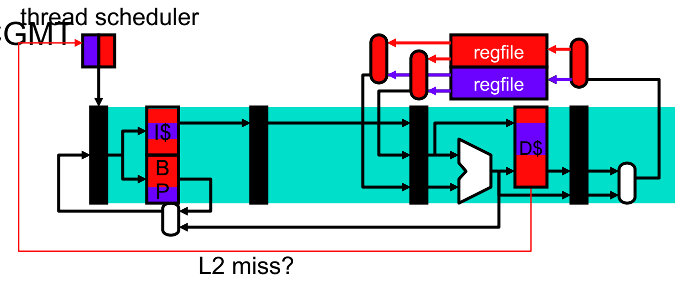


(170)

Coarse-Grain MultiThreaded architecture



- ❑ CGMT



(171)

Alternative Multithreaded Implementations

- ❑ MT trades (single-thread) latency for throughput
  - Sharing the datapath degrades the latency of individual threads, but improves the aggregate latency of both threads
  - And, it improves utilization of the datapath hardware
- ❑ Main questions: **thread scheduling policy and pipeline partitioning**
  - When to switch from one thread to another?
  - How exactly do threads share the pipelined datapath itself?
- ❑ Choices depends on what kind of latencies you want to tolerate and how much single thread performance you are willing to sacrifice
  - Coarse-grain multithreading (**CGMT**)
  - Fine-grain multithreading (**FGMT**) ⇐
  - Simultaneous multithreading (**SMT**)

(172)

Fine-Grain MultiThreading (FGMT)

– Sacrifices significant single thread performance

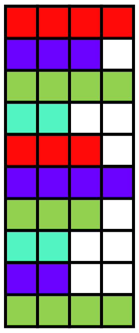
+ Tolerates latencies (e.g., load-use hazards, L1 misses, mispredicted branches, etc.)

- Thread scheduling policy
  - Switch threads every cycle (round-robin, can skip stalled threads)
- Pipeline partitioning
  - Dynamic, <u>no</u> pipeline flushing between threads

– Need a lot of threads

- Extreme example: Denelcor HEP
  - So many threads (100+), it didn't even need caches
  - Targeted for DoD, not successful commercially

  http://en.wikipedia.org/wiki/Heterogeneous_Element_Processor
- Sun's UltraSPARC T1 (Niagara)
  - Many threads → many RF http://en.wikipedia.org/wiki/UltraSPARC_T1


FGMT

(173)

173

FGMT Sharing Implementations Issues

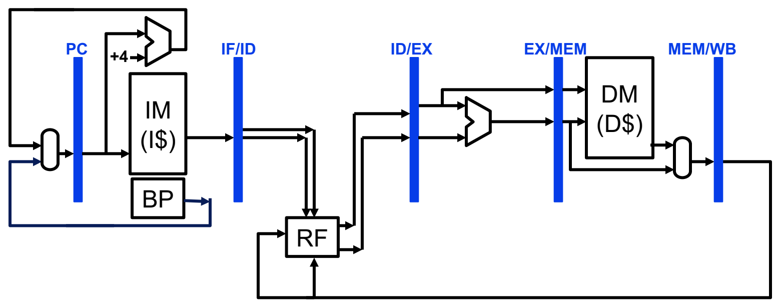
- How do multiple threads share a single datapath?
  - Different sharing mechanisms for different kinds of structures, depending on what kind of state the structure stores
- No state: ALUs
  - So, can be dynamically shared
- Persistent hard state (aka thread "context"): PC, RFile
  - So must be **replicated**
- Persistent soft state: caches, TLBs, branch prediction structures (BTB, BHT)
  - Dynamically partitioned (like on a multi-programmed uni-processor)
    - TLBs need thread ids, caches/branch prediction table (BHT) don't
- Transient state: pipeline latches
  - Must be partitioned … somehow

(174)

174

FGMT Datapath (Single Issue)

- What do we have to add to our datapath to support FGMT?

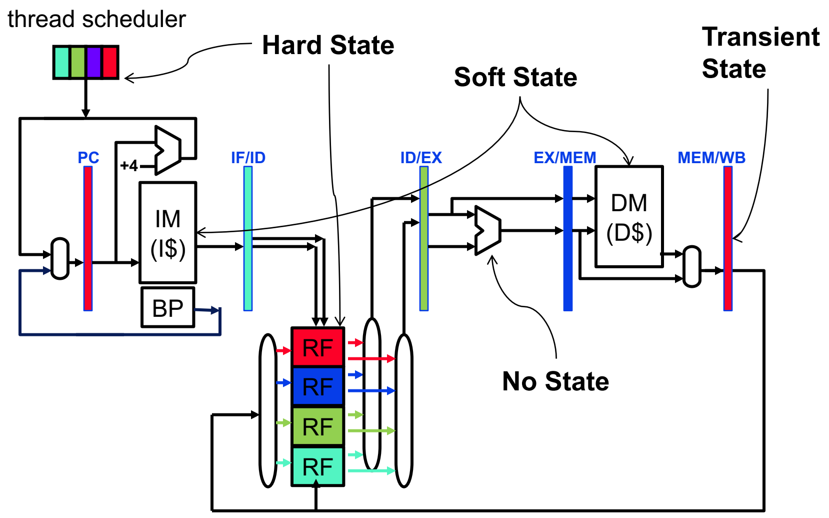


(175)

175

FGMT Datapath (Single Issue)

- What do we have to add to our datapath to support FGMT?

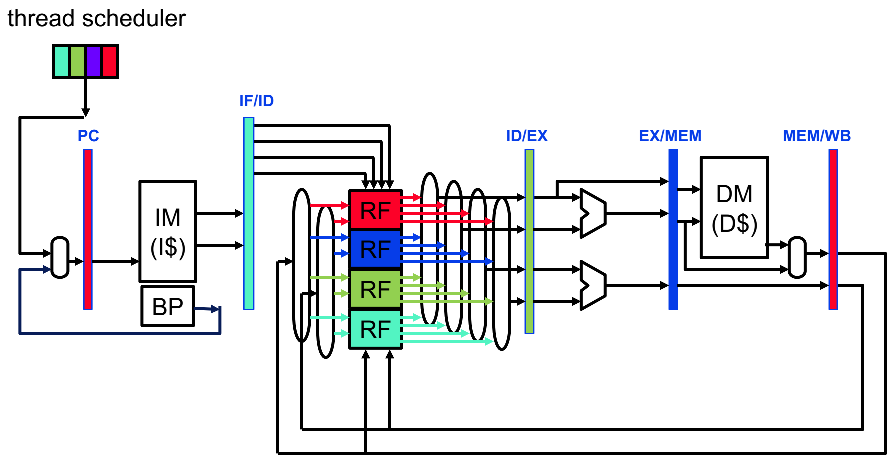


(176)

176

FGMT Datapath (2-Way Issue)

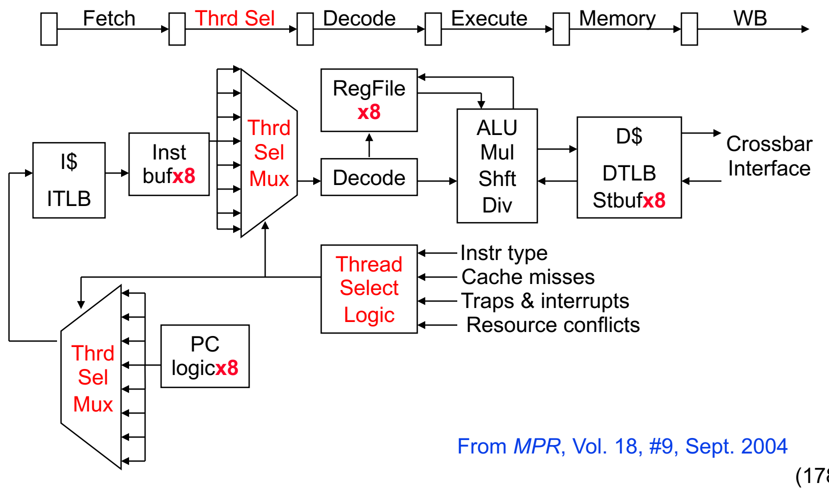
- What do we have to add to our datapath to support FGMT?



(177)

Sun Niagara's FGMT Integer pipeline

- Cores are simple (single-issue, 6 stage, no branch prediction), small, and power-efficient

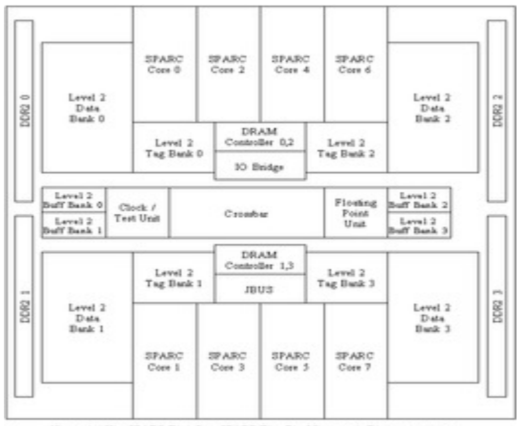


From *MPR*, Vol. 18, #9, Sept. 2004

(178)

Sun Niagara's Architecture

- 8 SPARC FGMT datapath cores

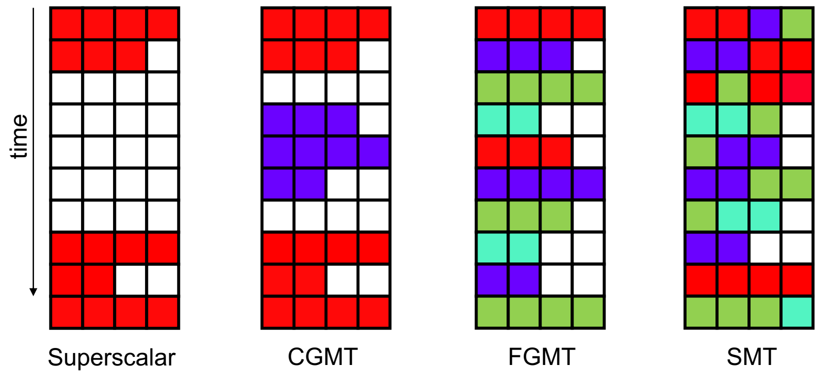


Niagra 1 / UltraSPARC T1 / OpenSPARC T1 - Die Micrograph Diagram ([illegible])

(179)

The standard multithreading pictures

- Time evolution of issue slots
  - Color = thread



(180)

Simultaneous Multithreading (§3.10)

- Key idea

  Issue multiple instructions from multiple threads each cycle

- Features
  - Fully exploit thread-level parallelism and instruction-level parallelism.
  - Better Performance for
    - Mix of independent programs
    - Programs that are parallelizable

(181)

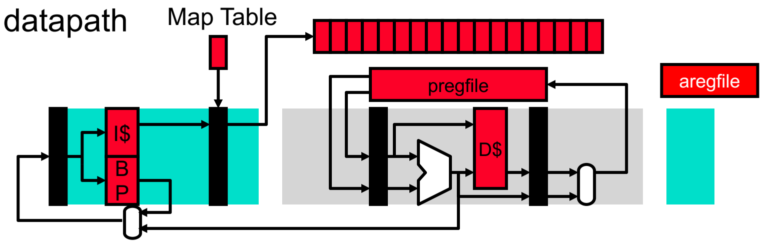
Simultaneous MultiThreading (SMT)

- What can issue instr's from multiple threads in one cycle?
  - Same thing that issues instr's from multiple parts of same thread …
  - …out-of-order execution !!
- Simultaneous multithreading (SMT): **OOO + FGMT**
  - Most common implementation of multithreading!
  - Aka (by Intel) "hyper-threading"
  - Once instr's are renamed, issuer doesn't care which thread they come from (well, for non-loads at least)
  - Some examples
    - IBM Power5: 4-way, 2 threads; IBM Power7: 4-way, 4 threads
    - Intel Pentium4: 3-way, 2 threads; Intel Core i7: 4-way, 2 threads
    - AMD Bulldozer: 4-way, 2 threads
    - Alpha 21464: 8-way issue, 4 threads (canceled)
    - Notice a pattern? #threads (T) * 2 = #issue width (N)

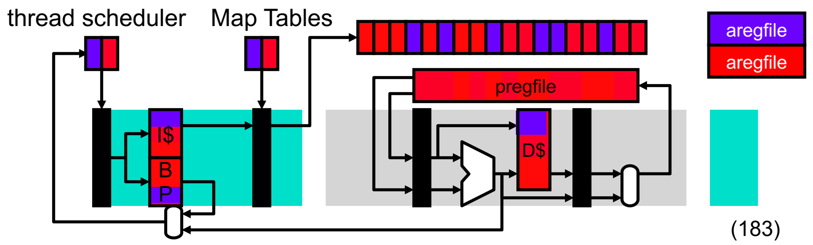
(182)

Simultaneous MultiThreaded architecture

- OoO datapath



- SMT
  - Threads share (larger) physical register file

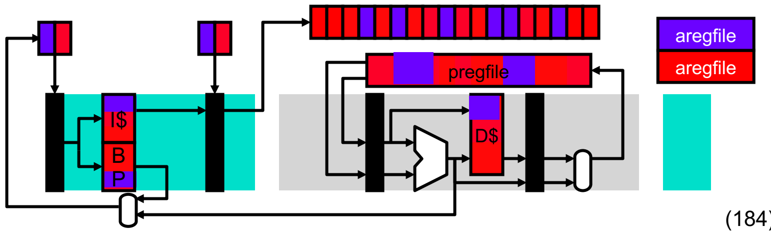


(183)

183

Static/dynamic ROB & LSQ partitioning

- **Static partitioning**
  - T equal-sized contiguous partitions
  - ± No starvation, sub-optimal utilization (fragmentation)
- **Dynamic partitioning**
  - P > T partitions, available partitions assigned on need basis
  - ± Better utilization, possible starvation
- Couple both with larger ROBs/LSQs



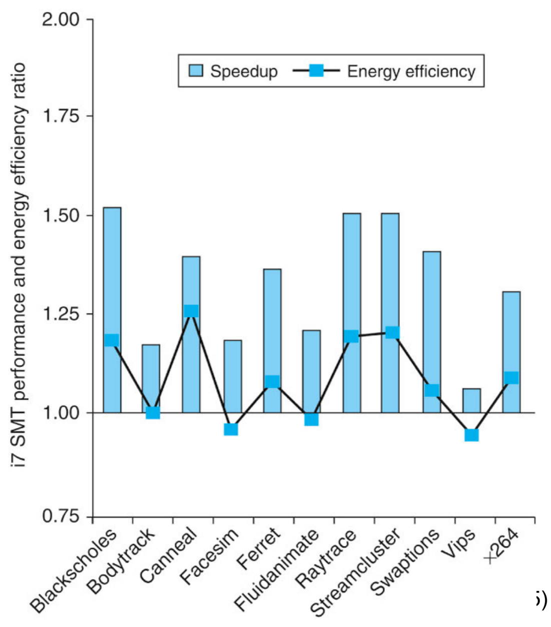
(184)

184

Multithreading Speed-Ups on the i7 (SMT)

Multithreading allows multiple threads tp share the functional units of a single processor in an overlapping fashion

Duplicates only private state – not everything



;)

Multithreading vs Multicore

- If you wanted to run multiple threads would you build a
  - A multicore: multiple separate pipelines?
  - A multithreaded processor: a single larger pipeline?
- **Both will get you throughput on multiple threads**
  - A multicore core will be simpler, possibly faster clock
    - Multicore is mainly a TLP (thread-level parallelism) engine
  - SMT will get you better performance (IPC) on a single thread
    - SMT is basically an ILP engine that converts TLP to ILP
- **Do both**
  - Intel’s Sandy (Ivy) Bridge and Haswell, IBM’s Power7 & 8
  - 4 to 8 OOO 4-way cores each of which supports 2 to 4 threads (SMT)
  - Private L1 and L2 caches, shared L3 cache
  - 3+ GHz clock rate
  - Graphics processors

(186)

Check Yourself

- ❑ Branch prediction
- ❑ Multiple issue
- ❑ VLIW
- ❑ Superscalar
- ❑ Static scheduling
- ❑ Dynamic scheduling
- ❑ In-order execution
- ❑ Out-of-order (OOO) execution
- ❑ Speculation
- ❑ Reorder buffer
- ❑ Register renaming

(187)

187