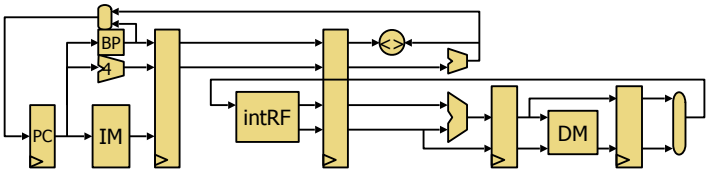
ECE 550
Computer Organization and Design

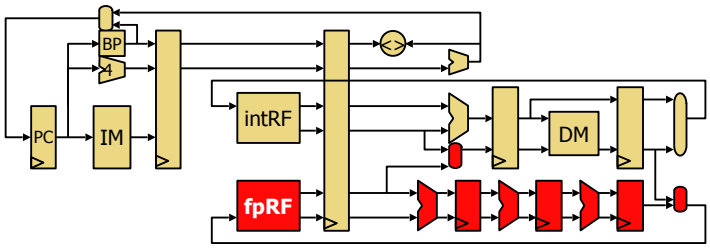
Fancy Pipelines: not just scalar in-order

Scalar Pipelines



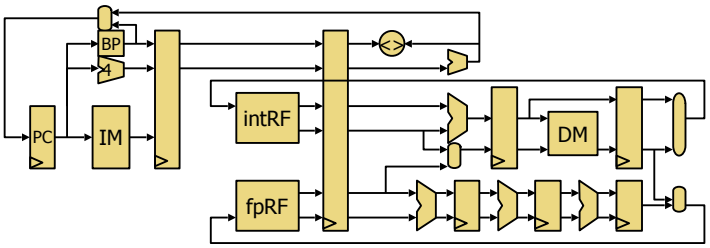
- So far we have looked at **scalar pipelines**
  - One insn per stage
  - With control speculation
  - With bypassing (not shown)

Floating Point Pipelines



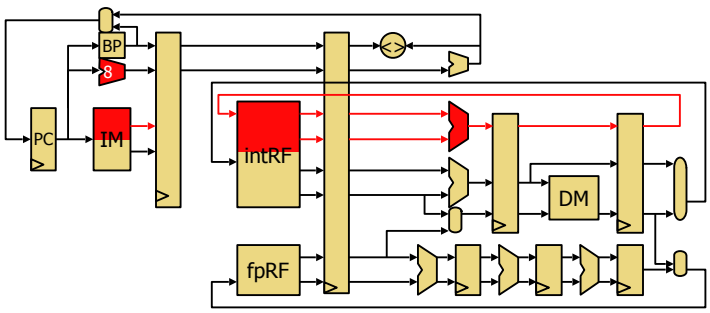
- Floating point (FP) insns typically use separate pipeline
  - Splits at decode stage: at fetch you don't know it's a FP insn
  - Most (all?) FP insns are multi-cycle (here: 3-cycle FP adder)
  - Separate FP register file
  - FP loads and stores execute on integer pipeline (address is integer)

The "Flynn Bottleneck"



– Performance limit of scalar pipeline is CPI = IPC = 1
  – Hazards → limit is not even achieved
  – Hazards + latch overhead → diminishing returns on "super-pipelining"

The "Flynn Bottleneck"



- Overcome IPC limit with **super-scalar pipeline**
  - Two insns per stage, or three, or four, or six, or eight…
  - Also called **multiple issue**
  - Exploit **"Instruction-Level Parallelism (ILP)"**

Superscalar Pipeline Diagrams

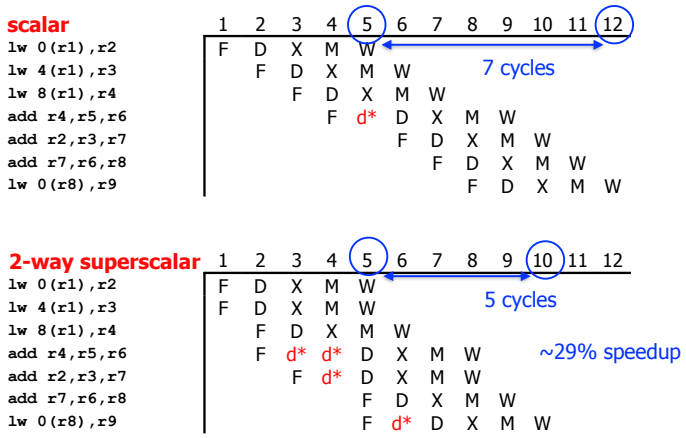
scalar

	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1),r2`	F	D	X	M	W							
`lw 4(r1),r3`		F	D	X	M	W						
`lw 8(r1),r4`			F	D	X	M	W					
`add r4,r5,r6`				F	d*	D	X	M	W			
`add r2,r3,r7`						F	D	X	M	W		
`add r7,r6,r8`							F	D	X	M	W	
`lw 0(r8),r9`								F	D	X	M	W

2-way superscalar

	1	2	3	4	5	6	7	8	9	10	11	12
`lw 0(r1),r2`	F	D	X	M	W							
`lw 4(r1),r3`	F	D	X	M	W							
`lw 8(r1),r4`		F	D	X	M	W						
`add r4,r5,r6`		F	d*	d*	D	X	M	W				
`add r2,r3,r7`			F	d*	D	X	M	W				
`add r7,r6,r8`					F	D	X	M	W			
`lw 0(r8),r9`					F	d*	D	X	M	W		

Superscalar Pipeline Diagrams



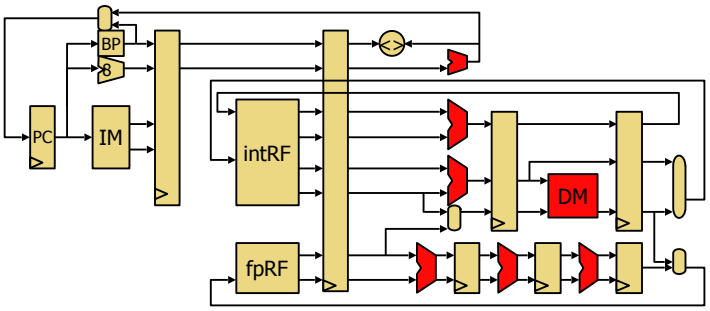
Superscalar CPI Calculations

- Base CPI for scalar pipeline is 1
- **Base CPI for N-way superscalar pipeline is 1/N**
  - Amplifies stall penalties
- Example: Branch penalty calculation
  - 20% branches, 75% taken, no explicit branch prediction
- Scalar pipeline
  - 1 + 0.2*0.75*2 = 1.3 → 1.3 / 1 = 1.3 → 30% slowdown
- 2-way superscalar pipeline
  - 0.5 + 0.2*0.75*2 = 0.8 → 0.8 / 0.5 = 1.6 → 60% slowdown
- 4-way superscalar
  - 0.25 + 0.2*0.75*2 = 0.55 → 0.55 / 0.25 = 2.2 → 120% slowdown

Challenges for Superscalar Pipelines

- So you want to build an N-way superscalar…
- **Hardware challenges**
  - Stall logic: N^2 terms
  - Bypasses: $2N^2$ paths
  - Register file: 3N ports
  - IMem/DMem: how many ports?
  - Anything else?
- **Software challenges**
  - Does program inherently have ILP of N?
  - Even if it does, compiler must schedule code to expose it
- Given these challenges, what is a reasonable N?
  - Current answer is 4–6

Superscalar "Execution"



- N-way superscalar = N of every kind of functional unit?
  - N ALUs? OK, ALUs are small and integer insns are common
  - N FP dividers? No, FP dividers are huge and `fdiv` is uncommon
  - How many loads/stores per cycle? How many branches?

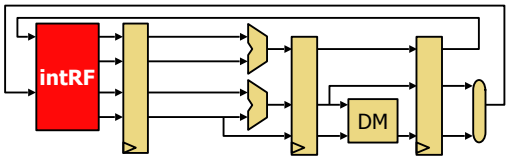
Superscalar Execution

- Common design: functional unit mix ∝ insn type mix
  - Integer apps: 20–30% loads, 10–15% stores, 15–20% branches
  - FP apps: 30% FP, 20% loads, 10% stores, 5% branches
  - Rest 40–50% are non-branch integer ALU operations
  - Intel Pentium (2-way superscalar): 1 any + 1 integer ALU
  - Alpha 21164: 2 integer (including 2 loads or 1 store) + 2 FP

DMem Bandwidth: Multi-Porting

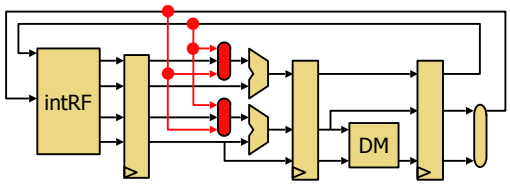
- Split IMem/Dmem gives you one dedicated DMem port
  - How to provide a second (maybe even a third) port?
- **Multi-porting**: just add another port
  - \+ Most general solution, any two reads/writes per cycle
  - – Latency, area ∝ #bits * $\#ports^2$
- Other approaches, not focusing too much on this.

Superscalar Register File



- Except DMem, execution units are easy
  - Getting values to/from them is the problem
- N-way superscalar register file: 2N read + N write ports
  - < N write ports: stores, branches (35% insns) don't write registers
  - < 2N read ports: many inputs come from immediates/bypasses
  - Still bad: latency and area ∝ #ports^2 ∝ (3N)^2

Superscalar Bypass



- Consider WX bypass for 1st input of each insn
  - 2 non-regfile inputs to bypass mux: in general N
  - 4 point-to-point connections: in general N^2
  - Bypass wires are difficult to route
  - And have high capacitive load (2N gates on each output)
  - And this is just one bypass stage and one input per insn!
- **N^2 bypass**

Superscalar Stall Logic

- Full bypassing → load/use stalls only
  - Ignore 2nd register input here, similar logic
- Stall logic for scalar pipeline
  (X/M.op==LOAD && D/X.rs1==X/M.rd)
- Stall logic for a 2-way superscalar pipeline
  - Stall logic for older insn in pair: also stalls younger insn in pair
    (X/M$_1$.op==LOAD && D/X$_1$.rs1==X/M$_1$.rd) ||
    (X/M$_2$.op==LOAD && D/X$_1$.rs1==X/M$_2$.rd)
  - Stall logic for younger insn in pair: doesn't stall older insn
    (X/M$_1$.op==LOAD && D/X$_2$.rs1==X/M$_1$.rd) ||
    (X/M$_2$.op==LOAD && D/X$_2$.rs1==X/M$_2$.rd) ||
    (D/X$_2$.rs1==D/X$_1$.rd)
- 5 terms for 2 insns: **N^2 dependence cross-check**
  - Actually N^2+N–1

Superscalar Pipeline Stalls

- If older insn in pair stalls, younger insns must stall too
- What if younger insn stalls?
  - Can older insn from next group move up?
  - **Fluid**: yes
    ± Helps CPI a little, hurts clock a little
  - **Rigid**: no
    ± Hurts CPI a little, but doesn't impact clock

Rigid

	1	2	3	4	5
`lw 0(r1),r4`	F	D	X	M	W
`addi r4,1,r4`	F	d*	d*	D	X
`sub r5,r2,r3`				F	D
`sw r3,0(r1)`				F	D
`lw 4(r1),r8`					F

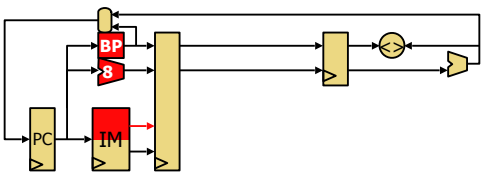
Fluid

	1	2	3	4	5
`lw 0(r1),r4`	F	D	X	M	W
`addi r4,1,r4`	F	d*	d*	D	X
`sub r5,r2,r3`		F	p*	D	X
`sw r3,0(r1)`				F	D
`lw 4(r1),r8`				F	D

Not All N^2 Problems Created Equal

- N^2 bypass vs. N^2 dependence cross-check
  - Which is the bigger problem?
- N^2 bypass … by a lot
  - 32- or 64- bit quantities (vs. 5-bit)
  - Multiple levels (MX, WX) of bypass (vs. 1 level of stall logic)
  - Must fit in one clock period with ALU (vs. not)
- Dependence cross-check not even 2nd biggest N^2 problem
  - Regfile is also an N^2 problem (think latency where N is #ports)
  - And also more serious than cross-check

Superscalar Fetch



- What is involved in fetching N insns per cycle?
  - Mostly wider IMem data bus
  - Most tricky aspects involve branch prediction

Superscalar Fetch with Branches

- Three related questions
  - How many branches are predicted per cycle?
  - If multiple insns fetched, which is assumed to be the branch?
  - Can we fetch across the branch if it is predicted "taken"?
- Simplest design: "one", "doesn't matter", "no"
  - One prediction, discard post-branch insns if prediction is "taken"
  - Doesn't matter: associate prediction with non-branch to same effect
  - – Lowers effective fetch bandwidth width and IPC
  - Average number of insns per taken branch? ~8–10 in integer code
- Compiler can help
  - Reduce taken branch frequency: e.g., unroll loops

Predication

- Branch mis-predictions hurt more on superscalar
  - Replace difficult branches with something else…
  - Usually: conditionally executed insns also conditionally fetched...
- **Predication**
  - Conditionally executed insns unconditionally fetched
  - **Full predication** (ARM, IA-64)
    - Can tag every insn with predicate, but extra bits in instruction
  - **Conditional moves** (Alpha, IA-32)
    - Construct appearance of full predication from one primitive

```
cmoveq r1,r2,r3        // if (r1==0) r3=r2;
```

  - – May require some code duplication to achieve desired effect
  - + Only good way of adding predication to an existing ISA
- **If-conversion**: replacing control with predication

Insn Level Parallelism (ILP)

- No point to having an N-way superscalar pipeline…
- …if average number of parallel insns per cycle (ILP) << N
  - Theoretically, ILP is high…
    - Integer apps: ~50, FP apps: ~250
  - In practice, ILP is much lower
    - Branch mis-predictions, cache misses, etc.
    - Integer apps: ~1–3, FP apps: ~4–8
- Sweet spot for hardware around 4–6
  - Rely on compiler to help exploit this hardware
  - Improve performance and utilization

Utilization

- **Utilization**: actual performance / peak performance
  - Important metric for performance/cost
  - No point to paying for hardware you will rarely use
  - Adding hardware usually improves performance & reduces utilization
    - Additional hardware can only be exploited some of the time
    - Diminishing marginal returns
  - Compiler can help make better use of existing hardware
    - Important for superscalar

Code Example: SAXPY

- **SAXPY** (Single-precision A X Plus Y)
  - Linear algebra routine (used in solving systems of equations)
  - Part of early "Livermore Loops" benchmark suite

```
for (i=0;i<N;i++)
  Z[i]=A*X[i]+Y[i];
```

```
0: ldf X(r1),f1       // loop
1: mulf f0,f1,f2      // A in f0
2: ldf Y(r1),f3       // X,Y,Z are constant addresses
3: addf f2,f3,f4
4: stf f4,Z(r1)
5: addi r1,4,r1       // i in r1
6: blt r1,r2,0        // N*4 in r2
```


SAXPY Performance and Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`ldf X(r1),f1`	F	D	X	M	W															
`mulf f0,f1,f2`		F	D	d*	E*	E*	E*	E*	E*	W										
`ldf Y(r1),f3`			F	p*	D	X	M	W												
`addf f2,f3,f4`					F	D	d*	d*	d*	E	E	W								
`stf f4,Z(r1)`						F	p*	p*	p*	D	X	M	W							
`addi r1,4,r1`										F	D	X	M	W						
`blt r1,r2,0`											F	D	X	M	W					
`ldf X(r1),f1`												F	D	X	M	W				

- Scalar pipeline
  - Full bypassing, 5-cycle E*, 2-cycle E+, branches predicted taken
  - Single iteration (7 insns) latency: 16–5 = 11 cycles
  - **Performance**: 7 insns / 11 cycles = 0.64 IPC
  - **Utilization**: 0.64 actual IPC / 1 peak IPC = 64%

SAXPY Performance and Utilization

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`ldf X(r1),f1`	F	D	X	M	W															
`mulf f0,f1,f2`	F	D	d*	d*	E*	E*	E*	E*	E*	W										
`ldf Y(r1),f3`		F	D	p*	X	M	W													
`addf f2,f3,f4`		F	p*	p*	D	d*	d*	d*	d*	E	E	W								
`stf f4,Z(r1)`			F	p*	D	p*	p*	p*	p*	d*	X	M	W							
`addi r1,4,r1`					F	p*	p*	p*	p*	p*	D	X	M	W						
`blt r1,r2,0`					F	p*	p*	p*	p*	p*	D	d*	X	M	W					
`ldf X(r1),f1`											F	D	X	M	W					

- 2-way superscalar pipeline (fluid)
  - Same + any two insns per cycle + embedded taken branches
  - \+ **Performance**: 7 insns / 10 cycles = 0.70 IPC
  - – **Utilization**: 0.70 actual IPC / 2 peak IPC = 35%
  - – More hazards → more stalls
  - – Each stall is more expensive

(Compiler) Instruction Scheduling

- Idea: place independent insns between slow ops and uses
  - Otherwise, pipeline stalls while waiting for RAW hazards to resolve
  - Have already seen pipeline scheduling
- To schedule well you need … **independent insns**
- **Scheduling scope**: code region we are scheduling
  - The bigger the better (more independent insns to choose from)
  - Once scope is defined, schedule is pretty obvious
  - Trick is creating a large scope (must schedule across branches)
- Compiler scheduling (really scope enlarging) techniques
  - Loop unrolling (for loops)

Loop Unrolling SAXPY

- Goal: separate dependent insns from one another
- SAXPY problem: not enough flexibility within one iteration
  - Longest chain of insns is 9 cycles
    - Load (1)
    - Forward to multiply (5)
    - Forward to add (2)
    - Forward to store (1)
  - – Can't hide a 9-cycle chain using only 7 insns
  - But how about two 9-cycle chains using 14 insns?
- **Loop unrolling**: schedule two or more iterations together
  - Fuse iterations
  - Pipeline schedule to reduce RAW stalls
  - Pipeline schedule introduces WAR violations, rename registers to fix

Unrolling SAXPY I: Fuse Iterations

- Combine two (in general K) iterations of loop
  - Fuse loop control: induction variable (`i`) increment + branch
  - Adjust (implicit) induction uses: constants → constants + 4

```
ldf X(r1),f1
mulf f0,f1,f2
ldf Y(r1),f3
addf f2,f3,f4
stf f4,Z(r1)
addi r1,4,r1
blt r1,r2,0
ldf X(r1),f1
mulf f0,f1,f2
ldf Y(r1),f3
addf f2,f3,f4
stf f4,Z(r1)
addi r1,8,r1
blt r1,r2,0
```


Unrolling SAXPY II: Pipeline Schedule

- Pipeline schedule to reduce RAW stalls
  - Have already seen this: pipeline scheduling

```
ldf X(r1),f1
mulf f0,f1,f2
ldf Y(r1),f3
addf f2,f3,f4
stf f4,Z(r1)
ldf X+4(r1),f1
mulf f0,f1,f2
ldf Y+4(r1),f3
addf f2,f3,f4
stf f4,Z+4(r1)
addi r1,8,r1
blt r1,r2,0
```

```
ldf X(r1),f1
ldf X+4(r1),f1
mulf f0,f1,f2
mulf f0,f1,f2
ldf Y(r1),f3
ldf Y+4(r1),f3
addf f2,f3,f4
addf f2,f3,f4
stf f4,Z(r1)
stf f4,Z+4(r1)
addi r1,8,r1
blt r1,r2,0
```

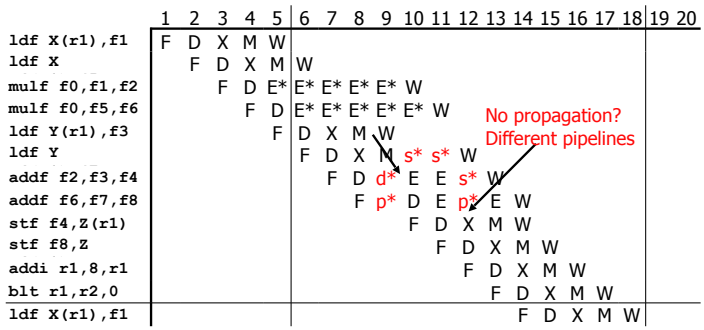

Unrolling SAXPY III: Rename Registers

- Pipeline scheduling causes WAR violations
  - Rename registers to correct

```
ldf X(r1),f1
ldf X+4(r1),f1
mulf f0,f1,f2
mulf f0,f1,f2
ldf Y(r1),f3
ldf Y+4(r1),f3
addf f2,f3,f4
addf f2,f3,f4
stf f4,Z(r1)
stf f4,Z+4(r1)
addi r1,8,r1
blt r1,r2,0
```

```
ldf X(r1),f1
ldf X+4(r1),f5
mulf f0,f1,f2
mulf f0,f5,f6
ldf Y(r1),f3
ldf Y+4(r1),f7
addf f2,f3,f4
addf f6,f7,f8
stf f4,Z(r1)
stf f8,Z+4(r1)
addi r1,8,r1
blt r1,r2,0
```

Unrolled SAXPY Performance/Utilization



+ Performance: 12 insn / 13 cycles = 0.92 IPC
+ Utilization: 0.92 actual IPC / 1 peak IPC = 92%
+ **Speedup**: (2 * 11 cycles) / 13 cycles = 1.69

Loop Unrolling Shortcomings

- Static code growth → more I$ misses (limits degree of unrolling)
- Needs more registers to resolve WAR hazards
- **Doesn't handle recurrences** (inter-iteration dependences)
- Doesn't handle non-loops…

```
for (i=0;i<N;i++)
  X[i]=A*X[I-1];
```

```
ldf X-4(r1),f1
mulf f0,f1,f2
stf f2,X(r1)
addi r1,4,r1
blt r1,r2,0
ldf X-4(r1),f1
mulf f0,f1,f2
stf f2,X(r1)
addi r1,4,r1
blt r1,r2,0
```

→

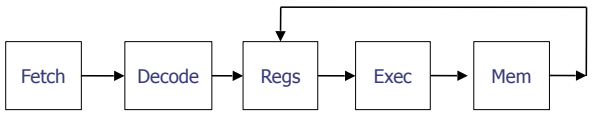
```
ldf X-4(r1),f1
mulf f0,f1,f2
stf f2,X(r1)
mulf f0,f2,f3
stf f3,X+4(r1)
addi r1,4,r1
blt r1,r2,0
```

- Two `mulf`'s are not parallel

Anything Compiler Can Do…

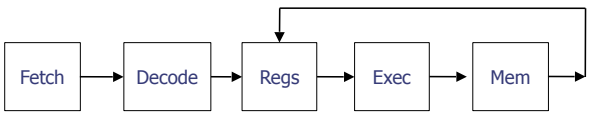
- **Dynamically-scheduled superscalar**
  - Hardware re-schedules insns…
  - …within a sliding window of VonNeumann insns
  - Does loop unrolling transparently
  - Does equivalent of loop unrolling on non-loop code
    - Uses branch prediction to "unroll" branches
  - Can handle data cache misses (don't know what that is yet), but…
    - Can flexibly schedule insns around uncertain latencies
  - Pentium Pro/II/III (3-wide), Core/2 (4-wide), Alpha 21264 (4-wide), MIPS R10000 (4-wide), Power5 (5-wide)
  - Not going to cover in detail, but… quick overview

Out-of-order 10K foot view



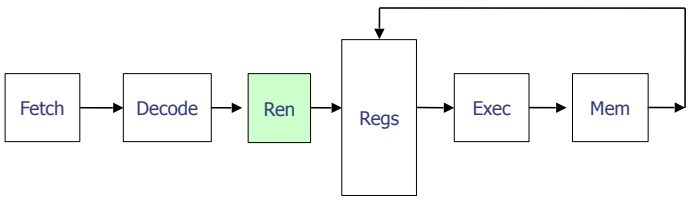
- Let's revisit the in-order pipeline…
  - Stall when reading registers for dependences
    - Load-use or dependent insns together
  - May be younger, independent insns, but can't let them around:
    - Hardware doesn't support it (no "jump past")
    - Program expects its insns done in order…
- Re-order, but maintain **illusion** of in-order?

Out-of-order 10K foot view



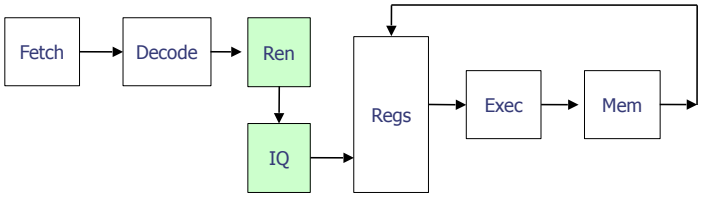
- Problem: Write-after-write (WAW) + Write-after-read (WAR)
  - 1: add $**r3**, $r1, $r2
  - 2: ld $r4, 0 ($**r3**)
  - 3: ld $**r3**,0 ($r6)
- WAW: 3 then 1: now $r3 has wrong value later
- WAR: 1 then 3, then 2: now insn 2 reads wrong value
- Sure would be nice if compiler picked different reg…

Out-of-order 10K foot view



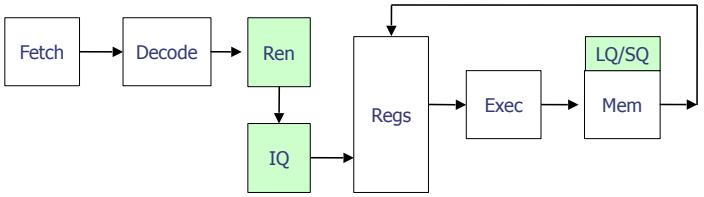
- Solution: register renaming (add a level of indirection)
  - Map logical names to physical names
  - Have more physical registers than logical registers
  - Must recover mapping on branch mis-prediction
    - Cleverly takes care of "undoing" wrong-path reg writes

Out-of-order 10K foot view



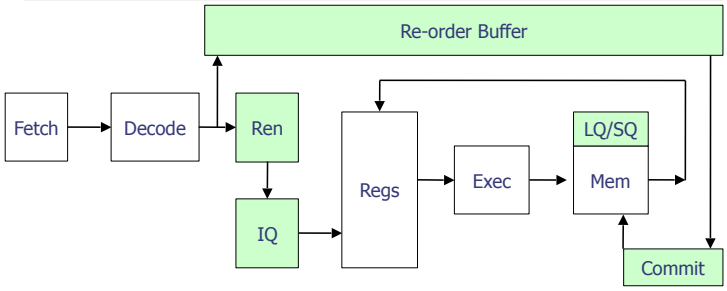
- Problem 2: How to pick what to issue?
  - **Issue Queue**: tracks "ready" status of insns per input reg
  - Insns broadcast destination physical register # at right time
    - "Wakeup" dependents
    - Parallel search/match circuit: CAM

Out-of-order 10K foot view



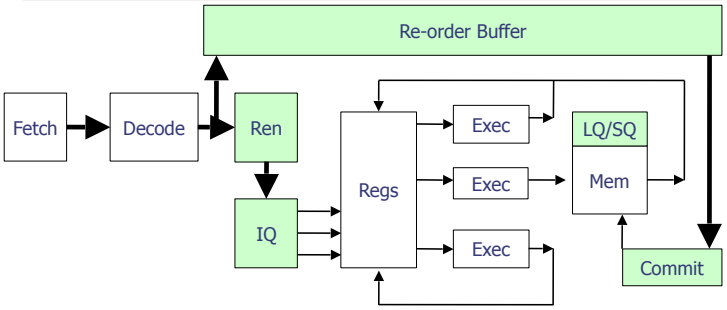
- Problem 3: Loads and stores
  - Stores cannot be "undone" once in Dmem
    - Need to buffer (Store Queue)
    - Loads must search: CAM
  - Register dependences: explicit (named in insn word)
  - Memory dependences: same address (depends on reg values)
    - Known after execute, speculate
    - Stores search Load Queue for incorrect loads

Out-of-order 10K foot view



- Problem 3: Need in-order notion of "really done"
  - Add Re-order Buffer
  - Track all in-flight instructions
  - Used for recovery (undo mappings in reverse order)
  - Also **commit**: instruction is done "for real"

Out-of-order 10K foot view



- Works well with super-scalar too!

Other Kinds of Parallelism

- So far have been talking about ILP
- Architects love _LP
  - ILP = Instruction Level Parallelism
  - DLP = Data Level Parallelism
  - TLP = Thread Level Parallelism
  - MLP = Memory Level Parallelism

Single Instruction Multiple Data

- One form of DLP: SIMD ("Vectors")
  - Vector reg holds 4 ints instead of 1
  - vadd $v1, $v2, $v3 : add 4 ints in $v2 to 4 ints in $v3, store in $v1
    - 1 instruction 4x the work -> ¼ the insns when used
    - Cheaper than super-scalar
      - Bypassing complexity
      - Reg read/write (wider read, not more ports)
- On x86: SSE
  - 2x 64-bit ints or 4x 32-bit ints per register

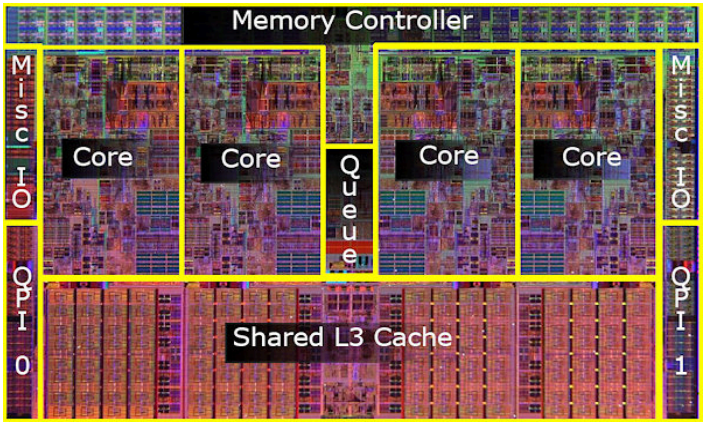
Thread Level Parallelism (TLP)

- Another type of parallelism: Thread Level Parallelism
  - ILP is fine-grained: individual instructions
  - TLP is coarse-grained: large independent tasks
- Imagine writing web-server
  - Process many requests
  - Most requests independent of each other
    - I load one page
    - You load another
  - Good candidate for multi-threading:
    - Handle different requests on different threads

Simultaneous Multi-Threading (SMT)

- SMT (Intel calls HyperThreading)
  - Interleave different threads in the pipeline
  - Increase utilization:
    - Slip other thread into stall cycles
- Very popular technique:
  - Intel's processors SMT-2 [two threads]
  - Power7 SMT-4 [four threads]
  - Works very well with superscalar
  - Also works well with out-of-order
    - Rename maps per-thread
    - Past rename, just looks like independent insns (except mem)

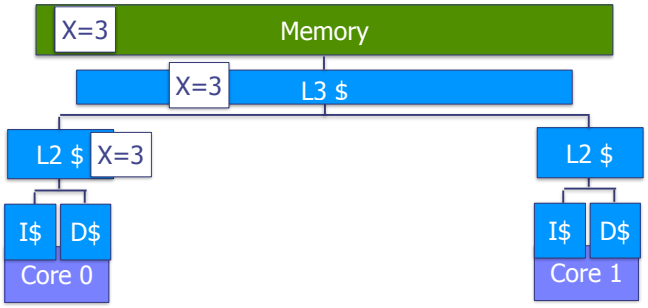
Die-photo: Core i7



Multi-core

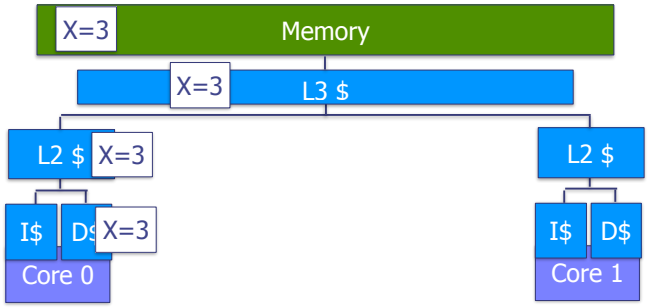
- Previous picture: 4 cores
  - Another way to exploit TLP: run 1 thread per core
  - …or actually 2 per core (SMT-2 on each core)
- Why not just SMT-8?
  - Have 4x the execution capability
  - At 4x the cost, not 16x!
    - All independent: no bypassing between cores
    - Also, nice for design cost: design once, replicate 4x
  - ..but…

Caches and Multi-Core



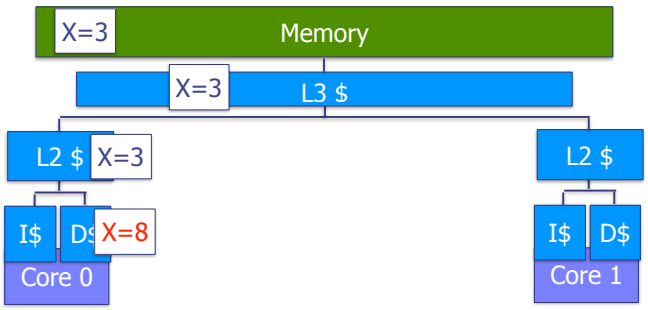
- What if…
  - Core 0 accesses X
    - Cache miss—request from memory

Caches and Multi-Core



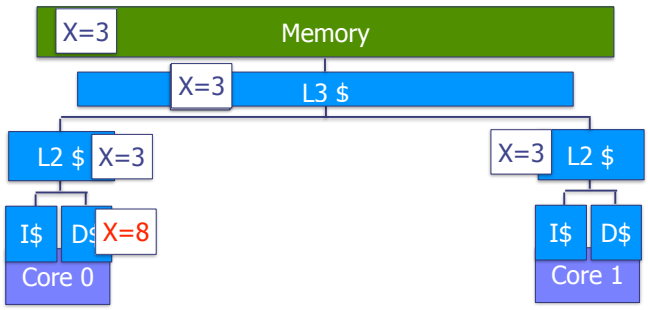
- Now, Core 0 has X in its D$..
  - So far, so good…

Caches and Multi-Core



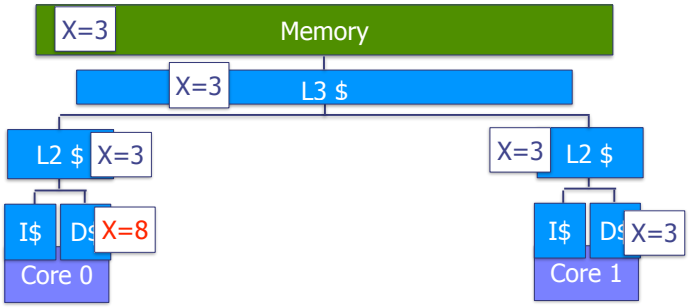
- Core 0 does a store X=8
  - Data is dirty in D$ (fine: write back cache)

Caches and Multi-Core



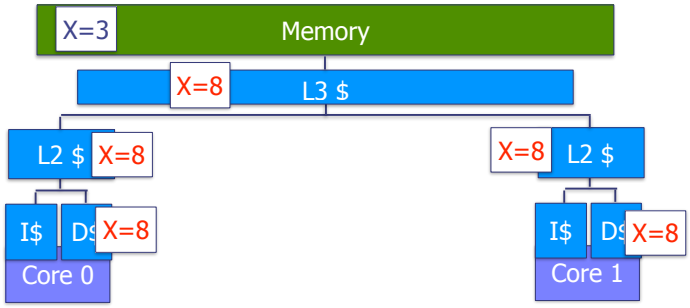
- Core 1 does a load of X
  - Also misses its cache, but hits the shared L3

Caches and Multi-Core



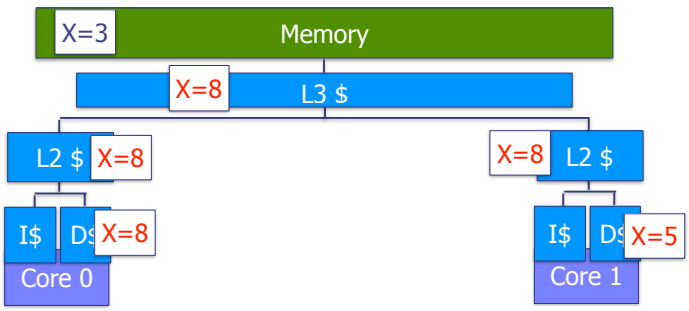
- Now we have a problem
  - Core 1 loaded the wrong value!
  - Stale data from L3: should be X = 8!

Caches and Multi-Core



- What needs to happen (part 1)
  - Get dirty data from Core 0
  - …but wait…

Caches and Multi-Core



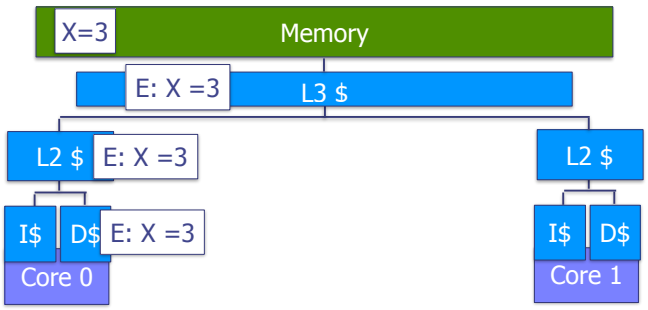
- What if Core 1 now does store X = 5?
- …and then Core 0 does a load?
  - It will still see X = 8! (also broken)

Solution: Cache Coherence

- Add cache coherence protocol
  - Key invariant: **single writer OR multiple readers**
  - Instead of just valid/invalid: coherence state
    - Many protocols, we'll just cover MESI

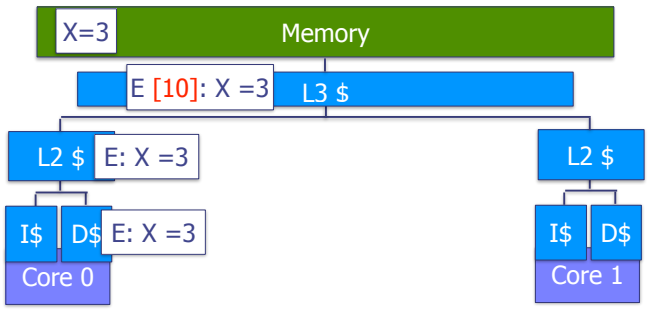
	Dirty?	Readable?	Writeable?
Modified	Yes	Yes	Yes
Exclusive	No	Yes	Yes
Shared	No	Yes	No
Invalid	—	No	No

Caches and Multi-Core



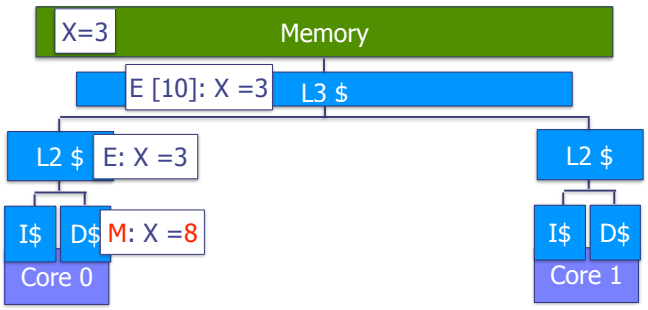
- Go back to the start
  - Core 0 gets block in E (Exclusive)
    - Readable/Writeable/Clean

Caches and Multi-Core



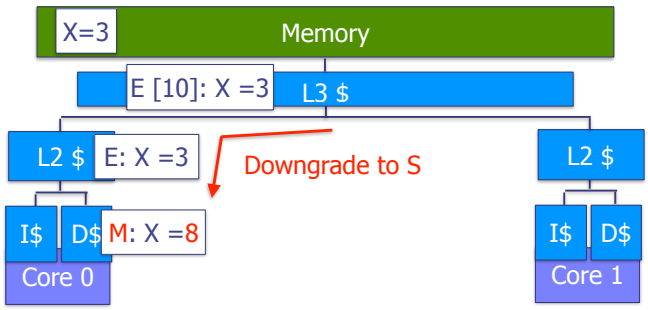
- In this design: L3 tracks **sharers vector**
  - Bit vector of which cores have the block
    - 1 = core (maybe) has, 0 = does not have

Caches and Multi-Core



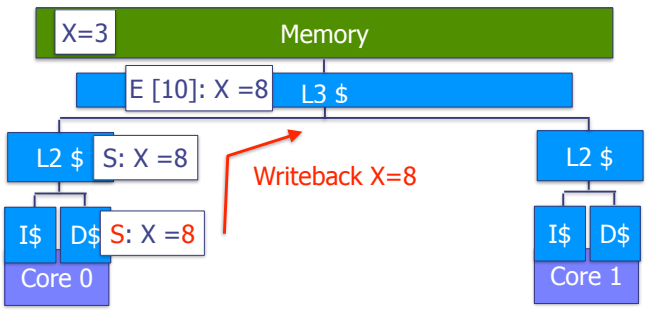
- Core 0 stores X = 8
  - Silently transitions to M
    - Silently = no messages needed

Caches and Multi-Core



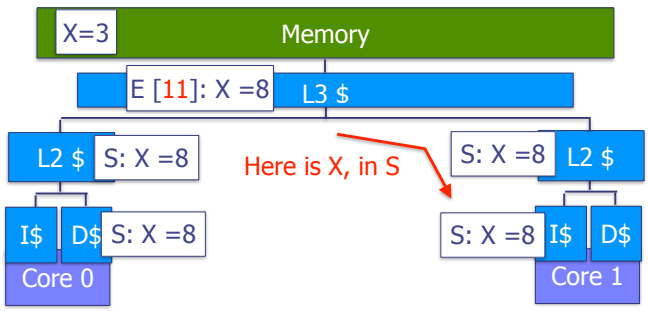
- Core 1 requests X ("getS X"), misses to L3
  - L3 sees that Core 0 has exclusive permissions
  - Must **downgrade** Core 0's permissions

Caches and Multi-Core



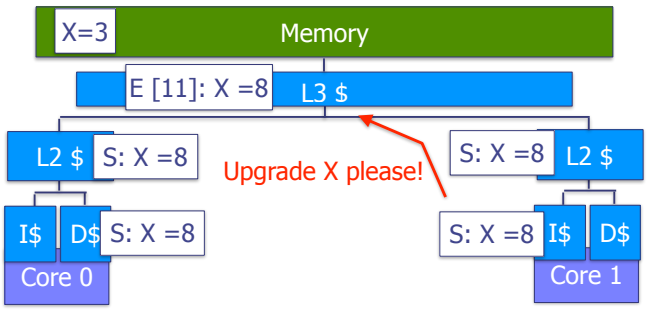
- Core 1 requests X ("getS X"), misses to L3
  - L3 sees that Core 0 has exclusive permissions
  - Must **downgrade** Core 0's permissions

Caches and Multi-Core



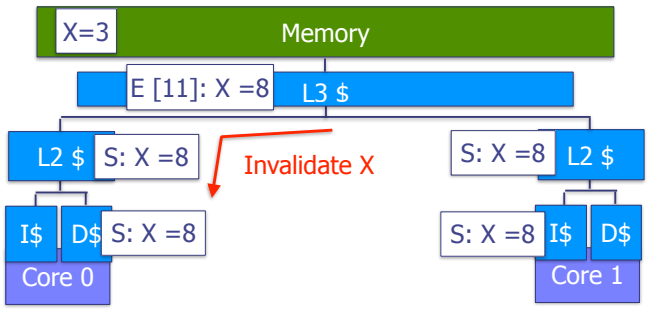
- Core 1 requests X ("getS X"), misses to L3
  - L3 sees that Core 0 has exclusive permissions
  - Must **downgrade** Core 0's permissions

Caches and Multi-Core



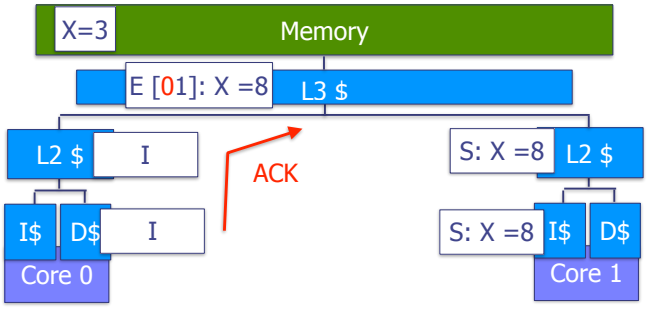
- Now Core 1 wants to write to X (store X = 5)
  - Has insufficient permissions
  - Must **upgrade** (kind of like a cache miss): ask L3

Caches and Multi-Core



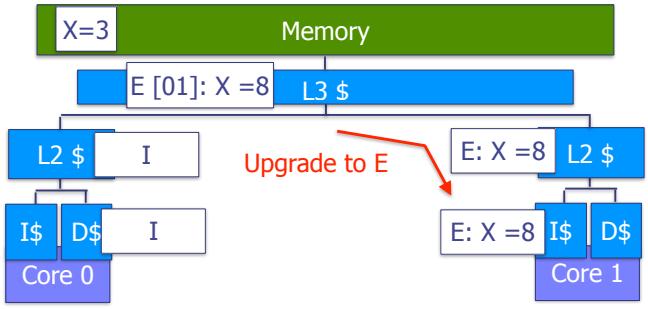
- L3 knows which caches might have data (sharers vector)
  - Sends invalidations

Caches and Multi-Core



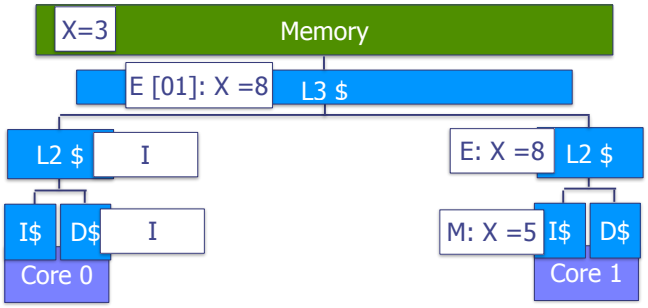
- Other caches send acknowledgements
  - L3 collects acknowledgements

Caches and Multi-Core



- Grants upgrade request to core 1
  - Now core 1 has in E
  - May complete its store and go to M

Caches and Multi-Core



- Core 0 cannot read block now (state = I)
  - Read or write will miss to L3
  - L3 knows Core 1 has it

Coherence More Complex

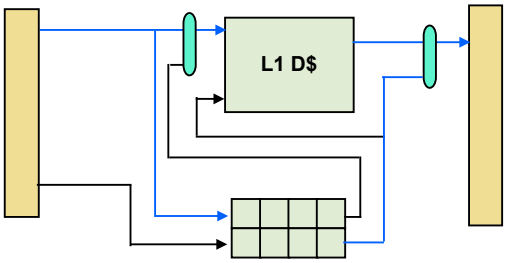
- Coherence is a more complex topic than that
  - Quick intro
- A variety of different protocols
- In reality:
  - Things happen in real time
    - Core 2 requests X in the middle of Core 1s request
  - Must be correct!

- Understanding coherence is crucial to writing parallel code
  - Key to performance
- Much more on coherence in 552!

Another Issue: Consistency

- Another multi-core issue: Memory Consistency
  - Do not confuse with cache coherence
- Memory consistency model: part of ISA
  - Specifies allowable **observable** re-orderings of lds/sts
    - Can reorder anything as long as not observable
  - Memory ops within a thread must be in order
    - Or at least appear so

Store Buffer



- Remember the store buffer?
  - Loads may read a value **before** an older store completes!
  - Can only do this if MC model allows it
    - What we talked about is basically TSO (store buffer is FIFO)
      - Loads may complete before older stores

Memory Consistency Models

- Very strong: sequential consistency
  - All memory operations (appear to) complete in order
    - No visible reordering allowed
- Strong: TSO
  - Loads may complete before older stores
  - Loads/load ordering remains
  - Store/store ordering remains
- Weak orderings
  - Allow more combinations
- Need to enforce ordering?
  - Use a **fence** instruction: explicit ordering instruction

Memory Consistency and Programming

Thread 0 — Start: A= 0, B = 0, C= 0 — Thread 1

```
A = 1;      //store        B=1; //store
int b = B; //load          C=1; //store
int c = C; //load          int a = A; //load
printf("b=%d\n", b);       printf("a=%d\n",a);
printf("c=%d\n", c);
```

- What does the above code print?
  - Better question: what **can** it print?

It depends… (on what/how?)

Memory Consistency and Programming

Thread 0 — Start: A= 0, B = 0, C= 0 — Thread 1

```
A = 1;      //store        B=1; //store
int b = B; //load          C=1; //store
int c = C; //load          int a = A; //load
printf("b=%d\n", b);       printf("a=%d\n",a);
printf("c=%d\n", c);
```

A	B	C	Legal Under
0	0	0	TSO, Weak
0	0	1	Weak
0	1	0	TSO, Weak
0	1	1	SC, TSO, Weak
1	0	0	SC, TSO, Weak
1	0	1	SC, TSO, Weak
1	1	0	SC, TSO, Weak
1	1	1	SC, TSO, Weak

(Much more in 552)

Wrap-up

- Quick look at "fancy" features in real processors
  - Super-scalar
  - Out-of-order
  - SIMD
  - SMT
  - Multi-core:
    - Coherence
    - Consistency
- Learn more in 552!

- Want to be able to make code fast? **Know hardware!**
  - Also, take Performance/Optimization/Parallelism class