CMPEN 431
Computer Architecture
Fall 2018

Static Superscalar Datapaths

Jack Sampson(www.cse.psu.edu/~sampson)

[Slides adapted from work by Mary Jane Irwin, in turn adapted from *Computer Organization and Design, Revised 4th Edition*,

Patterson & Hennessy, © 2011, Morgan Kaufmann & 5th edition, © 2014

With additional thanks/credits to Amir Roth, Milo Martin, CIS/UPenn]

Review: Extracting Yet *More* Performance

- Increase the depth of the pipeline to increase the clock rate (CPI still 1, IC unchanged) – superpipelining
  - The more stages in the pipeline, the more forwarding/hazard hardware needed and the more pipeline latch overhead (i.e., the pipeline latch accounts for a larger and larger percentage of the clock cycle time)
- Fetch (and execute) more than one instructions at one time (expand every pipeline stage to accommodate multiple instructions) – multiple-issue
  - The instruction execution rate, CPI, will be less than 1, so instead we use IPC: instructions per clock cycle
    - E.g., a 6 GHz, four-way multiple-issue processor can execute at a peak rate of 24 billion instructions per second with a best case CPI of 0.25 or a best case IPC of 4
  - If the datapath has a five stage pipeline, how many instructions are active in the pipeline at any given time?
- Both superpipelining and multiple-issue

Review: Multiple-Issue Datapath Responsibilities

- Must handle, with a combination of hardware and software fixes, the fundamental limitations of
  - How many instructions to issue (send for execution) in one clock cycle
  - Storage (data) dependencies → data hazards
    - Limitation more severe in a in-order SuperScalar/VLIW processor due to (usually) low ILP
  - Procedural dependencies → control hazards
    - Ditto, but even more severe
    - Use dynamic branch prediction to help resolve the ILP issue
    - Use loop unrolling (in the compiler) to increase ILP
  - Resource conflicts → structural hazards
    - A multiple-issue datapath has a much larger number of potential resource conflicts
    - Functional units may have to arbitrate for result buses and RF write ports
    - Resource conflicts can often be eliminated by duplicating the resource or by pipelining the resource

Review: Data Dependence Analysis

original	possible?	possible?

`instr 1`
`instr 2`
consecutive
	`instr 2`
	`instr 1`
	consecutive
		`instr 1` and `instr 2`
		simultaneous

- To exploit ILP must determine which instructions can be executed in parallel (without any stalls) – but must preserve program order
  - RAW, true dependence (cannot reorder)

```
a = .      lw    $t0,0($s1)        sw    $t0,0($s1)
. = a      addu  $t0,$t0,$s2       lw    $t1,0($s1)
```

  - WAR, anti-dependence (renaming allows reordering)

```
. = a      lw    $t0,0($s1)        lw    $t0,0($s1)
a = .      addu  $s1,$s2,$s3       sw    $t1,0($s1)
```

  - WAW, output dependence (renaming allows reordering)

```
a = .      lw    $t0,0($s1)        sw    $t0,0($s1)
a = .      addu  $t0,$s2,$s3       sw    $t1,0($s1)
```

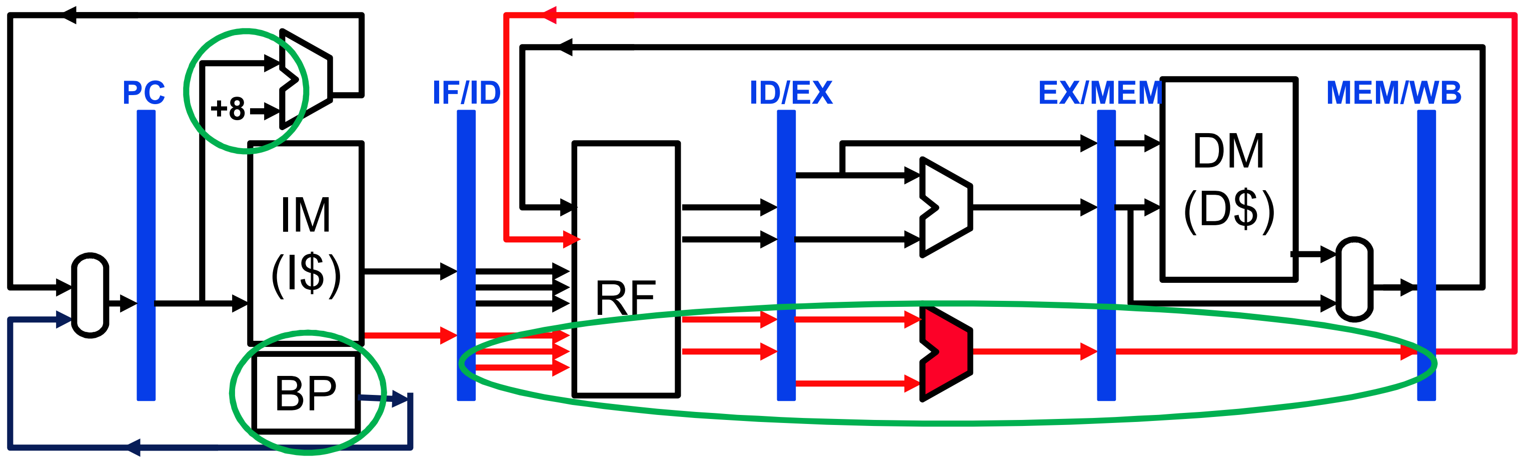
Multiple Instruction Issue Possibilities

- Fetch and issue **more than one** instruction in a cycle

1. **Statically-scheduled (in-order)**
   - **Very Long Instruction Word (VLIW)** e.g., TransMeta (4-wide)
     - Compiler figures out what can be done in parallel, so the hardware can be dumb and low power
     - Compiler must group parallel instr's, requires new binaries
   - **SuperScalar** e.g., Pentium (2-wide), ARM CortexA8 (2-wide)
     - Hardware figures out what can be done in parallel
     - Executes unmodified sequential programs
   - **Explicitly Parallel Instruction Computing (EPIC)** e.g., Intel Itanium (6-wide)
     - A compromise: compiler does some, hardware does the rest

2. **Dynamically-scheduled (out-of-order) SuperScalar**
   - Hardware dynamically determines what can be done in parallel (can extract much more ILP with OOO processing)
   - E.g., Intel Pentium Pro/II/III (3-wide), IBM Power7 (8-wide)

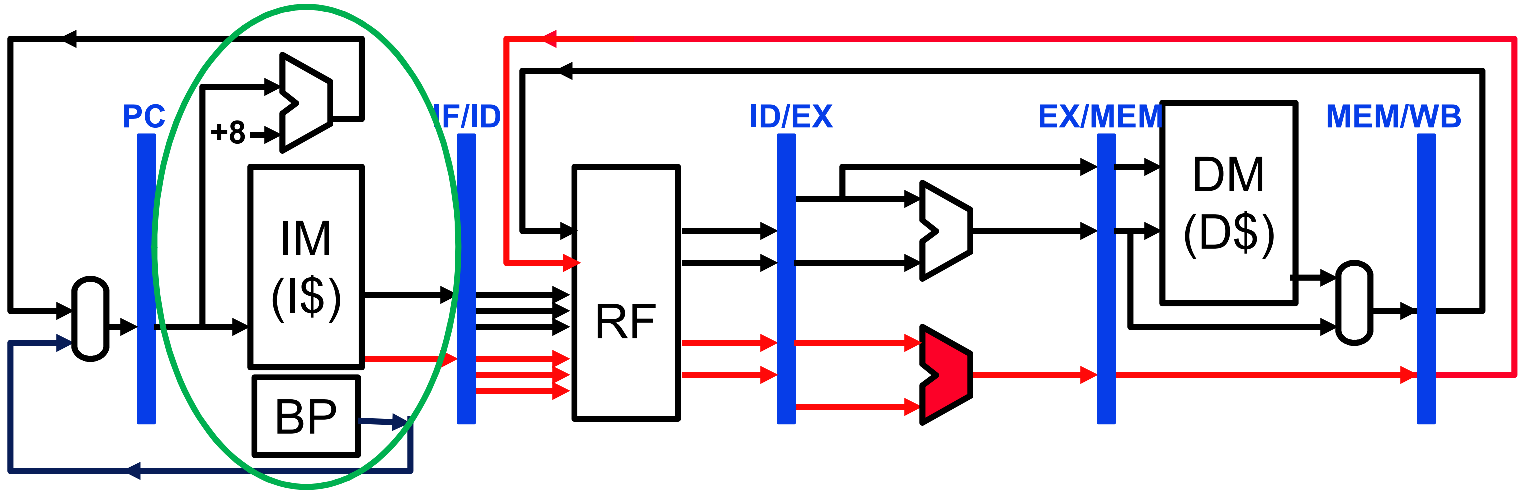
A (Simplified) Multiple Issue (In-Order) Pipeline



- Statically-scheduled in-order SuperScalar (SS)
  - Hardware figures out what can be done in parallel
  - Executes unmodified sequential programs
  - Instructions issue, execute and commit (change machine state) in order

- Today, typically 2-wide (like above) or 4-wide
  - 2-wide: Pentium, ARM CortexA8 (for low power)
  - 4-wide: Intel Core2, AMD Opteron
  - Some more (IBM Power5 is 5-wide)

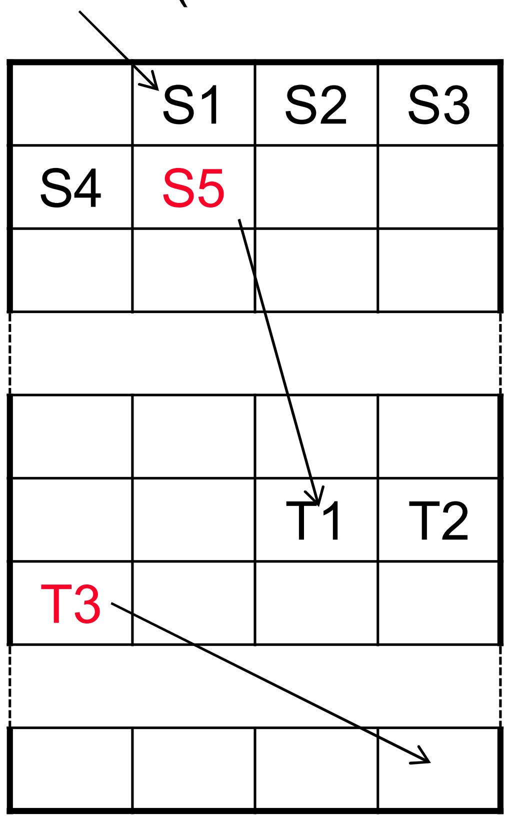
Static SS IF Stage Challenges



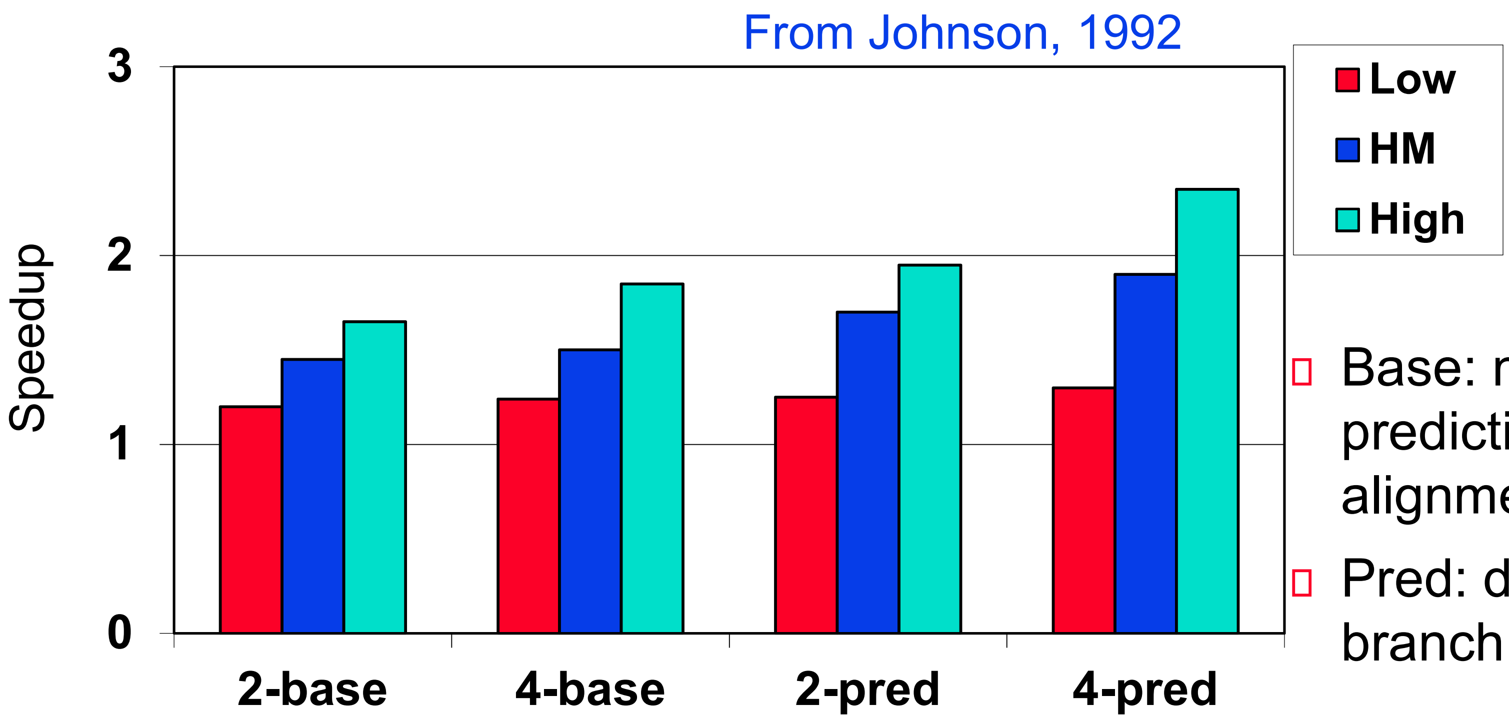
- Wide instruction fetch: Fetching 8B to 32B (2 to 8 instr's assuming 32b (4B) instr's) from the IM at once
  - Have to design the IM (I$) to support wide fetch in one cycle
- How many branches do we allow in a fetch bundle? Answer is usually only one (so that we only have to build one branch predictor).
  - Discard post-branch instr's in the fetch bundle if the prediction is "taken" which lowers the effective fetch width and the IPC
  - As we have seen, the compiler can help reduce the branch frequency with loop unrolling

Instruction Fetch Sequences

- Instruction run – number of (sequential) instructions (run length) fetched between taken branches
  - Instruction fetcher operates most efficiently when processing long runs – unfortunately runs are usually quite short (about six instr's)
- Example: for a 4-way fetcher, instr fetch bandwidth of only 2 instr's per cycle (with dynamic branch prediction)
  - 8 instructions in 4 cycles
- Fetcher can merge instr's from different runs, if the fetcher has the sufficient bandwidth (i.e., the fetch rate is faster than the decode rate, `fetch:speed<ratio>`)
  - 8 instructions in 3 cycles



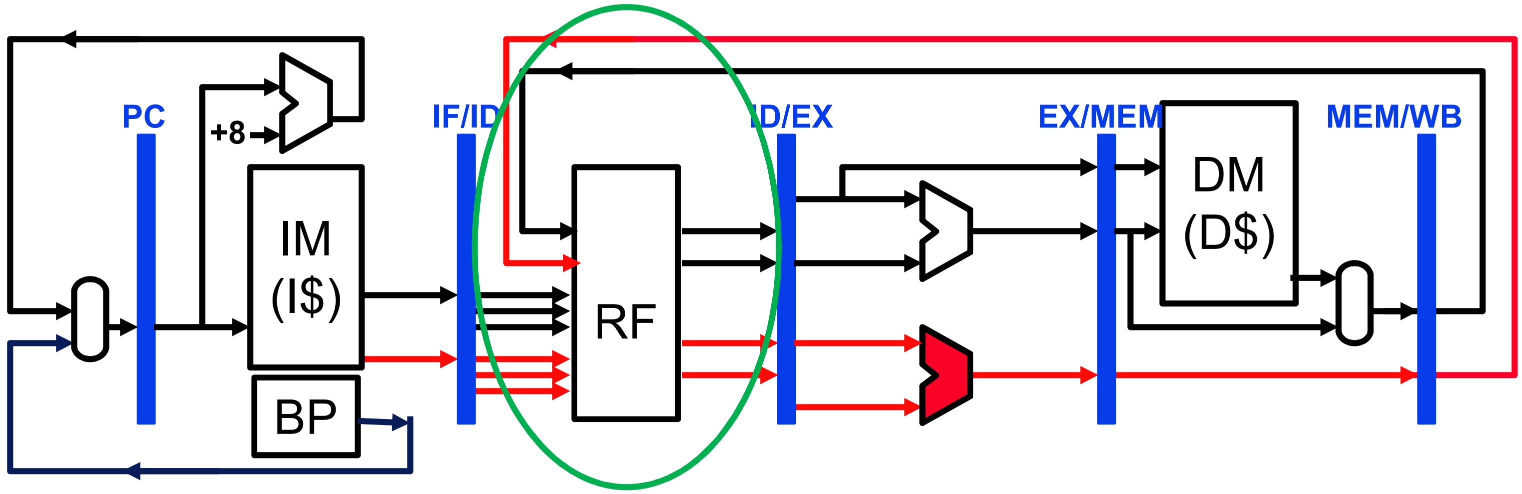
Speedups of Fetch Alternatives



- Base: no prediction and no alignment
- Pred: dynamic branch prediction

- A 4-way instr fetcher outperforms a 2-way instr fetcher
  - It has twice the potential instruction bandwidth
  - But it requires twice as much decoder hardware to keep up

Static SS Dec Stage Challenges



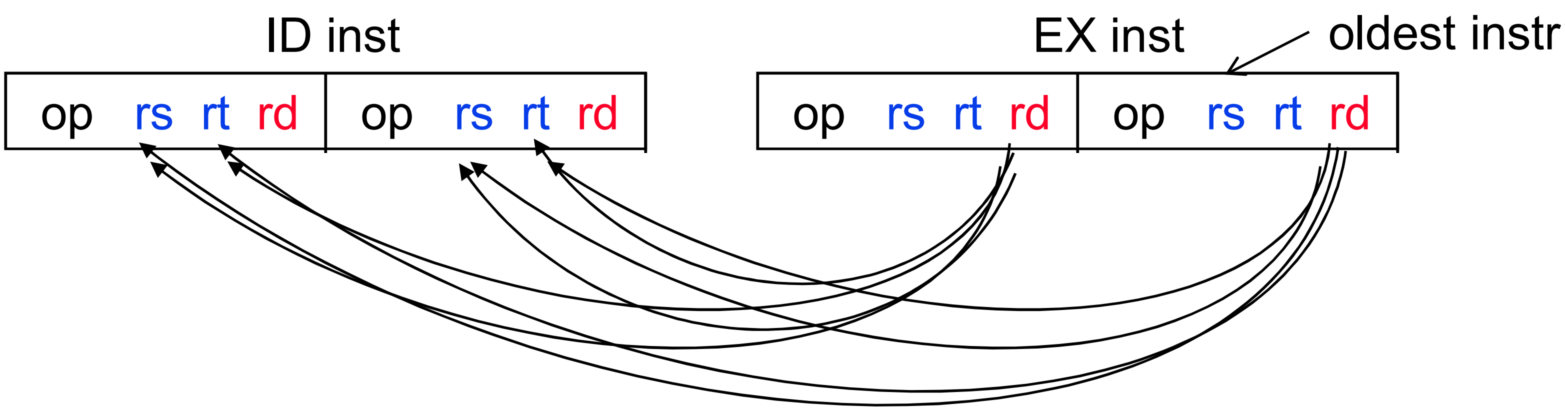
- Have to decode 2 to 8 instr's *at once* and decide which can issue (be sent to the Exec stage) *in parallel*
  - Duplicated decoders
  - Logic to determine if there are structural hazards and/or data dependencies in the instr bundle or load-use hazards with the previous instr bundle
  - Logic to stall conflicted instr's (and instr's in Fetch) for a cycle
- Multiported RF – 4 read ports/2 write ports (2 instr's) up to 16 read ports/8 write ports (8 instr's)
  - Larger area, latency, power, …

Dependency Checking

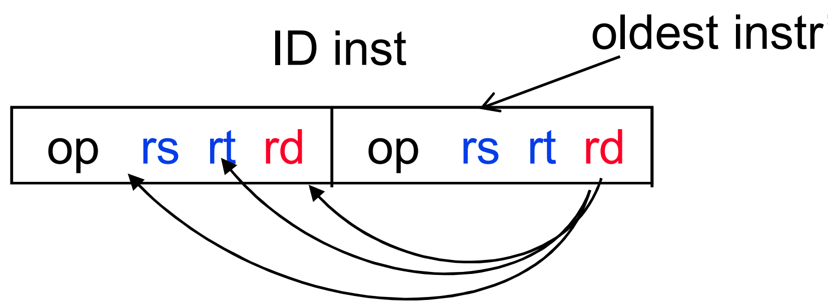
- Need to check for structural hazards (do the 2 (or 4, or 8) instr's need same FU's in EX ?)
  - If so need to either duplicate the FU's or stall one (or more) of the instr's in the bundle.
- Need to cross check for load-use hazards of the instr's in ID (the "use" instr's - for both of their src operands) to the instr's in EX (the "loads" instr's). We have forwarding logic that can take care of all other inter-bundle RAW data hazards.
- And need to check for dst-src (RAW) and dst-dst (WAW) dependencies between the instr's in the **same** instruction bundle in ID (intra-bundle RAW and WAW)
- Don't really have to check for WAR intra-bundle dependencies, why not?

2-way Dependency Checking

- Cross check for load-use hazards of the 2 instr's in ID (for both src's) to the 2 instr's in EX which gives __8__ load-use dependency checks



- And check for __2__ dst-src (RAW) and __1__ dst-dst (WAW) dependencies between the 2 instr's in the **same** instr bundle in ID



Handling of RAW (and WAW) Stalls (Lockstep)

- Cycle 1: RAW hazard detected

ID stage		EX stage
op \$2 rt rd	op rs rt \$2	

- Cycle 2: W instr moves to EX stage, R instr in ID and instructions in FETCH stalled for one cycle

ID stage		EX stage	
op \$2 rt rd			op rs rt \$2

- Cycle 3: EX/MEM to EX data forwarding will now happen

ID stage		EX stage		MEM stage	
op rs rt rd	op rs rt rd	op \$2 rt rd			op rs rt \$2

Handling of Load-Use Stalls

- Cycle 1: Load-use hazard detected

ID stage		EX stage	
op rs rt rd	op \$2 rt rd	op rs rt rd	lw rs \$2

- Cycle 2: Load-use hazard bubble insertion (instructions in FETCH stage and ID stage are stalled for one cycle)

ID stage		EX stage	MEM stage	
op rs rt rd	op \$2 rt rd		op rs rt rd	lw rs \$2

- Cycle 3: MEM/WB to EX data forwarding will now happen

ID stage		EX stage		MEM stage
op rs rt rd	op rs rt rd	op rs rt rd	op \$2 rt rd	

Handling of RAW (and WAW) Stalls (sliding queue)

- Cycle 1: RAW hazard detected

ID stage		EX stage
op $2 rt rd	op rs rt $2	

- Cycle 2: W instr moves to EX stage, R instr in ID and instructions in FETCH stalled for one cycle

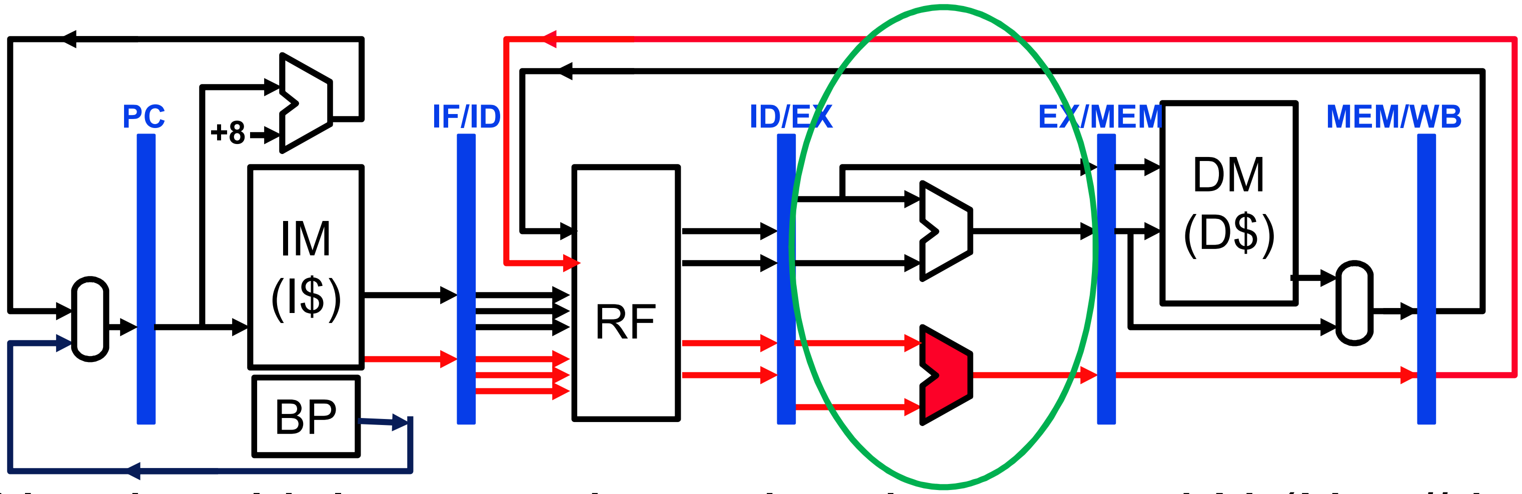
ID stage		EX stage	
op rs rt rd	op $2 rt rd		op rs rt $2

- Cycle 3: EX/MEM to EX data forwarding will now happen

ID stage		EX stage		MEM stage	
op rs rt rd	op rs rt rd	op rs rt rd	op $2 rt rd		op rs rt $2

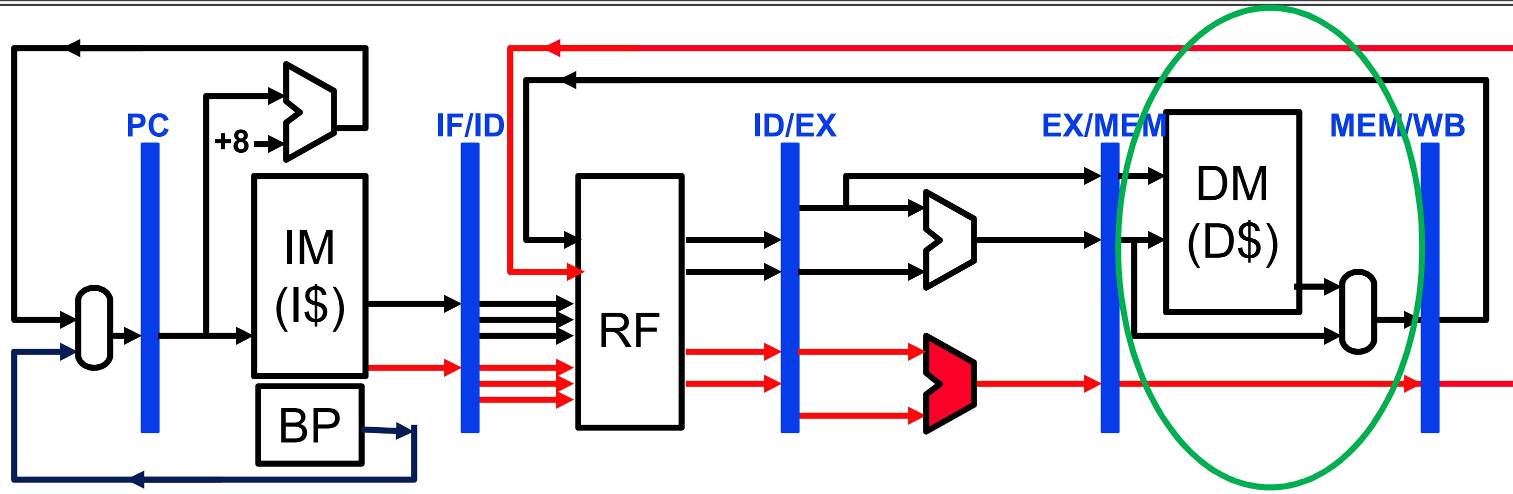
Loses alignment properties of lockstep – more complicated MUXing/control

Static SS Exec Stage Challenges



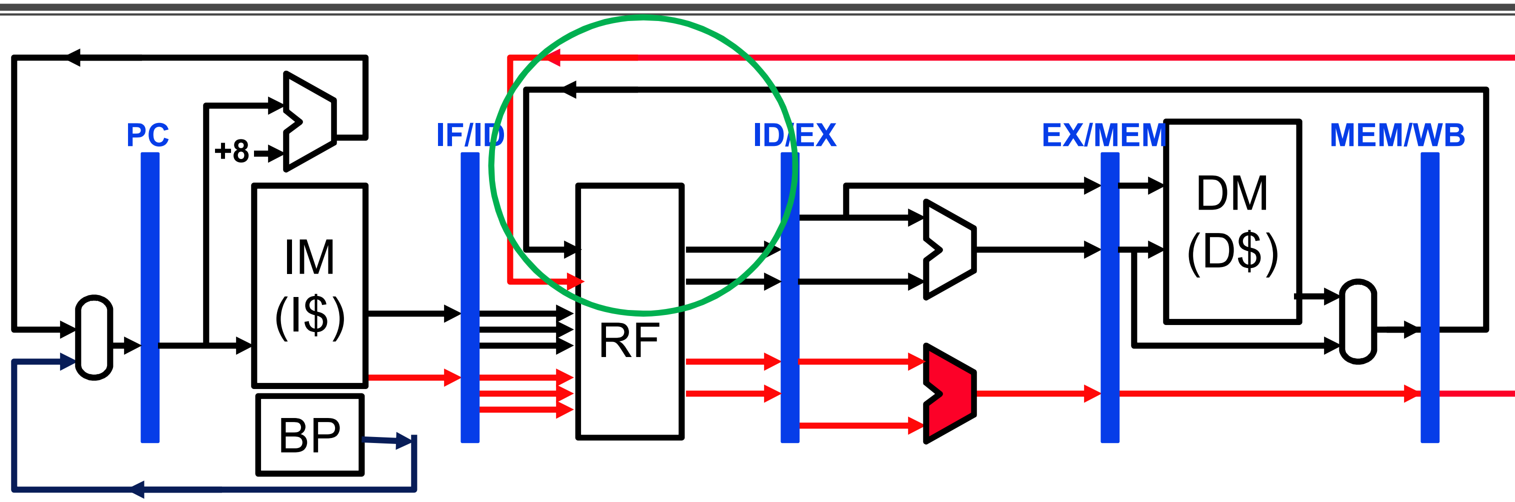
- Need multiple execution units, do we need N (N = # instr in an instr bundle) of every kind?
  - ALUs? FP dividers?
  - How many branches per bundle? (we already decided only one)
  - How many loads and/or stores per bundle?
- Usually some mix proportional to the instr mix
  - 2-way: 1 integer (branch, load, store, int) + 1 ALU (int, fp)
  - 4-way: 2 integer + 2 ALU

Static SS Mem Stage Challenges



- What about multiple loads and/or stores per cycle?
  - Probably only needed in 4-wide or greater
  - More important to support multiple loads than multiple stores
    - Instr mix: loads (~20% to 25%), stores (~10% to 15%)
- Have to design the DM (D$) to support multiple loads/stores in one cycle (have assumed only one DM port to this point)
  - Multi-porting is expensive in terms of latency, area, and power
  - Banked (interleaved) memories

Static SS WB Stage Challenges



- For an N-wide machine, need 2N RF read ports and N write ports
  - Read ports: area, latency ~ $(2N)^2$
  - Write ports: area, latency ~ N^2
- May not use the max number of read and write ports
  - Read ports: not all instr's use two source operands; forwarding supplies many of the read values (but don't know that at RF read time, so it doesn't help reduce read port count)
  - Write ports: stores, branches (~35%) don't write to the RF

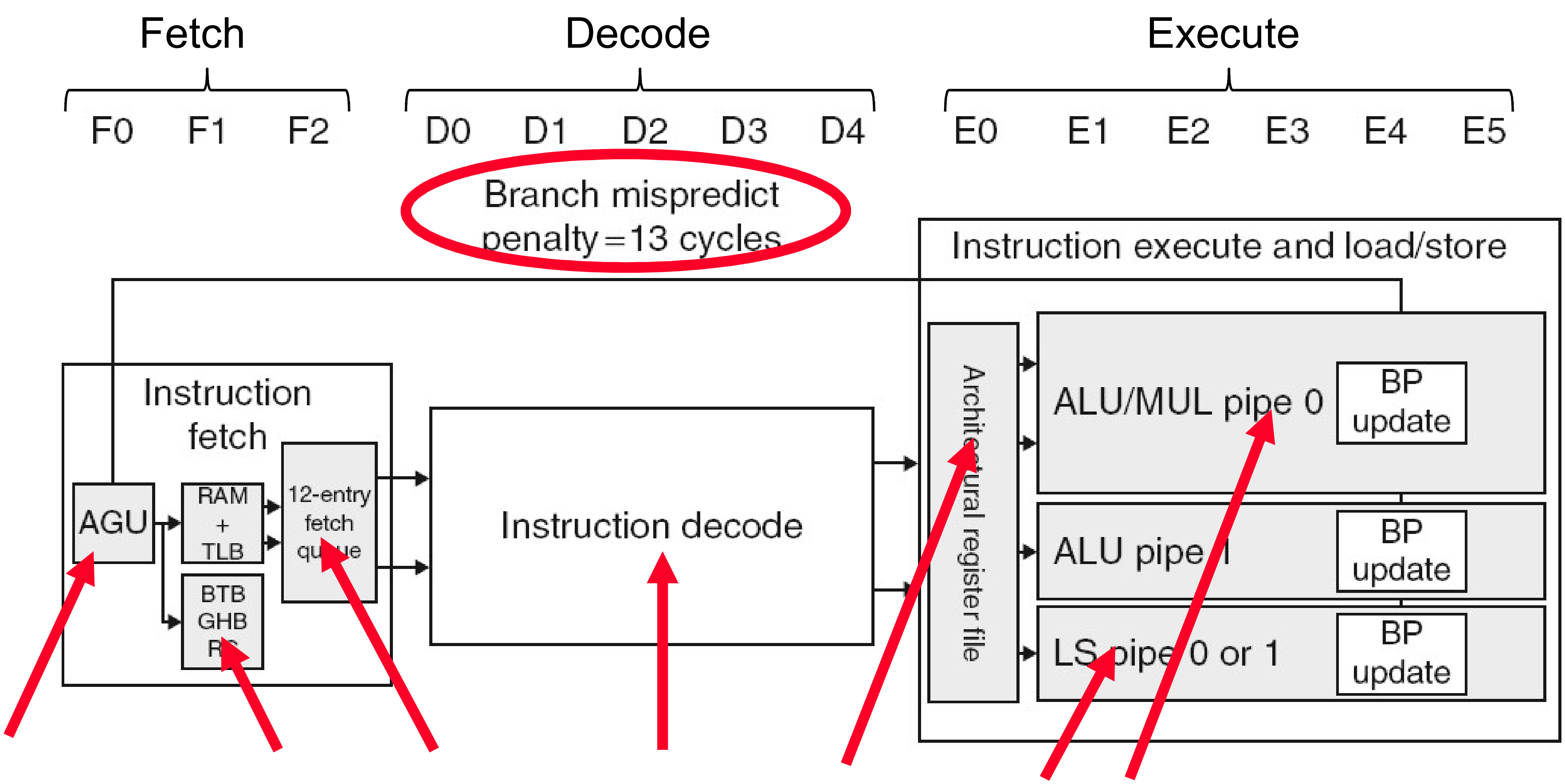
Trends in Static SS Datapath Design

	486	Pentium	PentiumII	Pentium4	Itanium	ItaniumII	Core2
Year	1989	1993	1998	2001	2002	2004	2006
Width	1	2	3	3	3*	6*	4

- Issue width has saturated at 4- to 6-way for high-performance cores
  - The canceled Alpha 21464 was an 8-way issue
  - Hardware or compiler “scheduling” needed to exploit 4- to 6-way effectively
    - Out-of-order execution (later lecture) (or VLIW or EPIC)
- For good-performing, low-power cores, issue width is ~2
  - So advanced scheduling techniques not needed
  - Use multi-threading (next …) to help cope with load-use hazards and cache misses

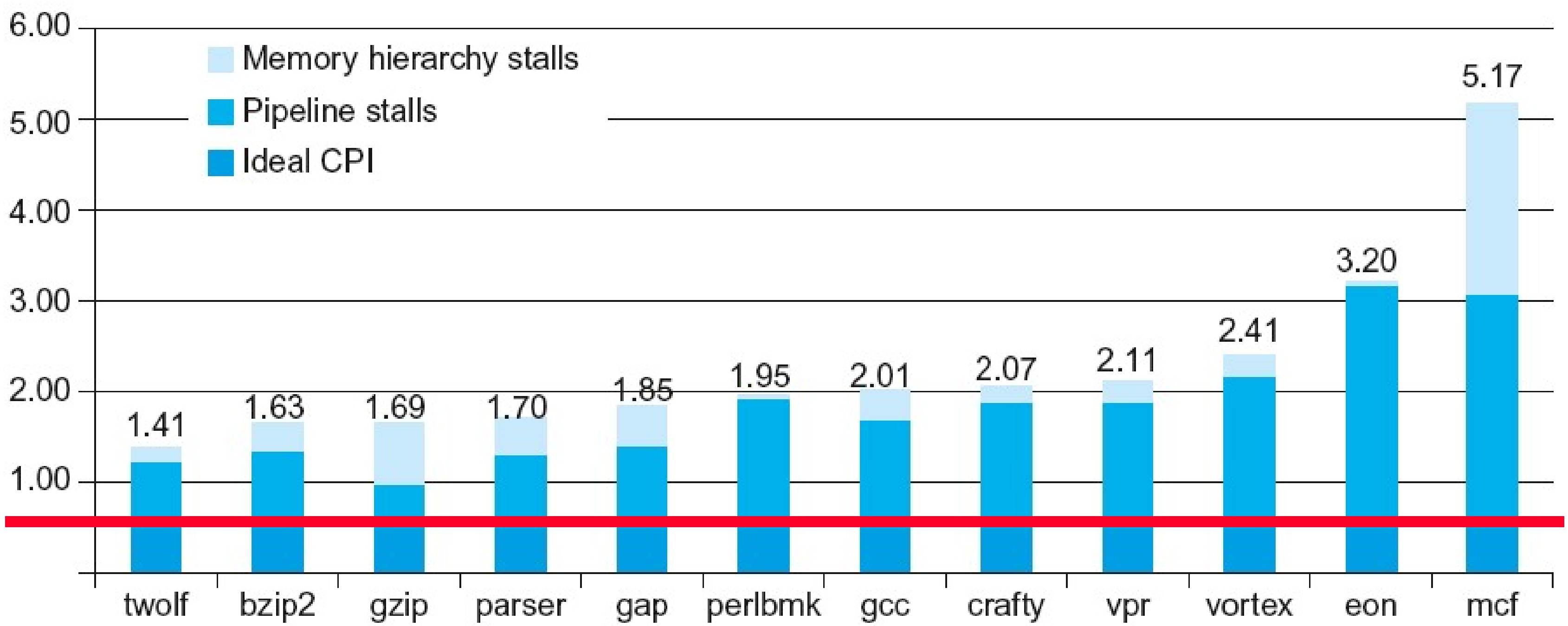
ARM Cortex A8 Pipeline

- 2-wide static (in-order) superscalar, 14-stage pipeline, 1GHz clock



ARM Cortex A8 Performance

- Ideal CPI is 0.5. For the median case (`gcc`), 80% of the stalls are due to pipeline hazards, 20% to memory stalls
  - Pipeline hazards are from branch mispredictions, structural hazards, and data dependencies between pairs
    - The compiler is the only thing that can help with structural hazards and data dependencies



Aside: CISC vs RISC vs Static SS vs VLIW

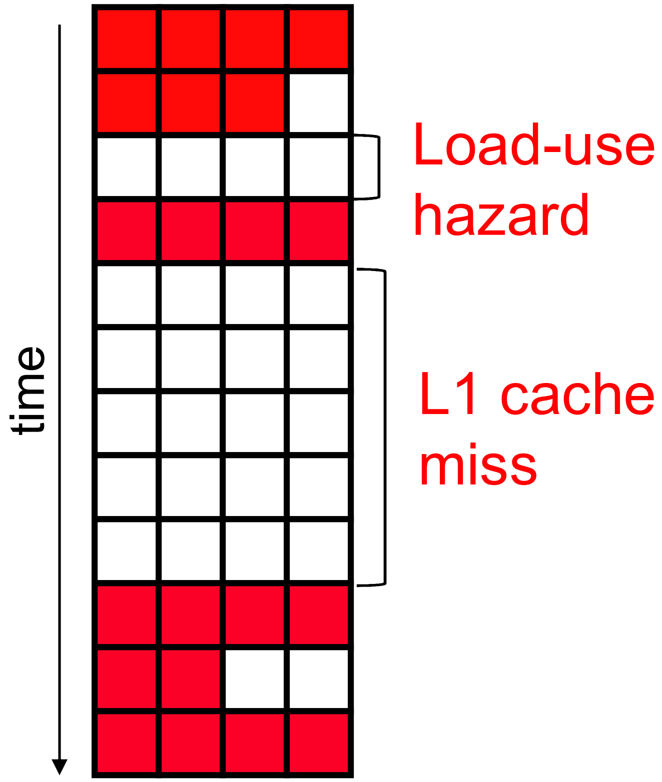
	CISC	RISC	Static Superscalar	VLIW
Instr size	variable size	fixed size	fixed size	fixed size (but large)
Instr format	variable format	fixed format	fixed format	fixed format
Registers	few, some special Limited # of ports	Many GP Limited # of ports	Many (more) GP Many ports	Many, many GP Many ports
Memory reference	embedded in many instr's	load/store	load/store	load/store
Key Issues	decode complexity	data forwarding, hazards	hardware instr dependency checks, data forwarding	(compiler) code scheduling

Multi-threading (MT)

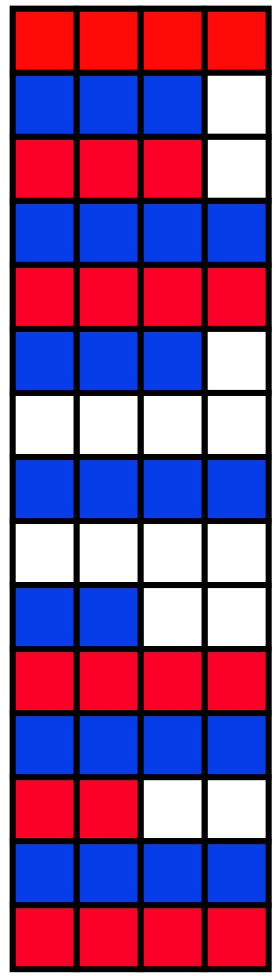
- Even moderate static superscalars (e.g., 4-way) are not fully utilized
  - Average sustained IPC: 1.5–2 $\rightarrow$ < 50% utilization due to
    - Mispredicted branches
    - Cache misses, especially L1
    - Data dependences (load-use data hazards)
- Multi-threading (MT) to the rescue
  - Improve *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

Multithreading Example

- Time evolution of issue slot
  - 4-way datapath



Static SS



Multithreaded Static SS

- # 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 hazards
  - Takes longer for the “red” thread to finish now
    - With more threads, would take even longer
  - Still have some noop “slots” (so wasted performance – stay tuned)

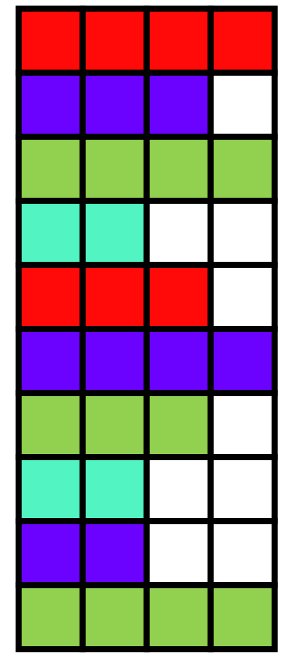
Alternative Multithreaded Implementations

- 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**)

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 threads)
- Pipeline partitioning
  - Dynamic, *no* 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



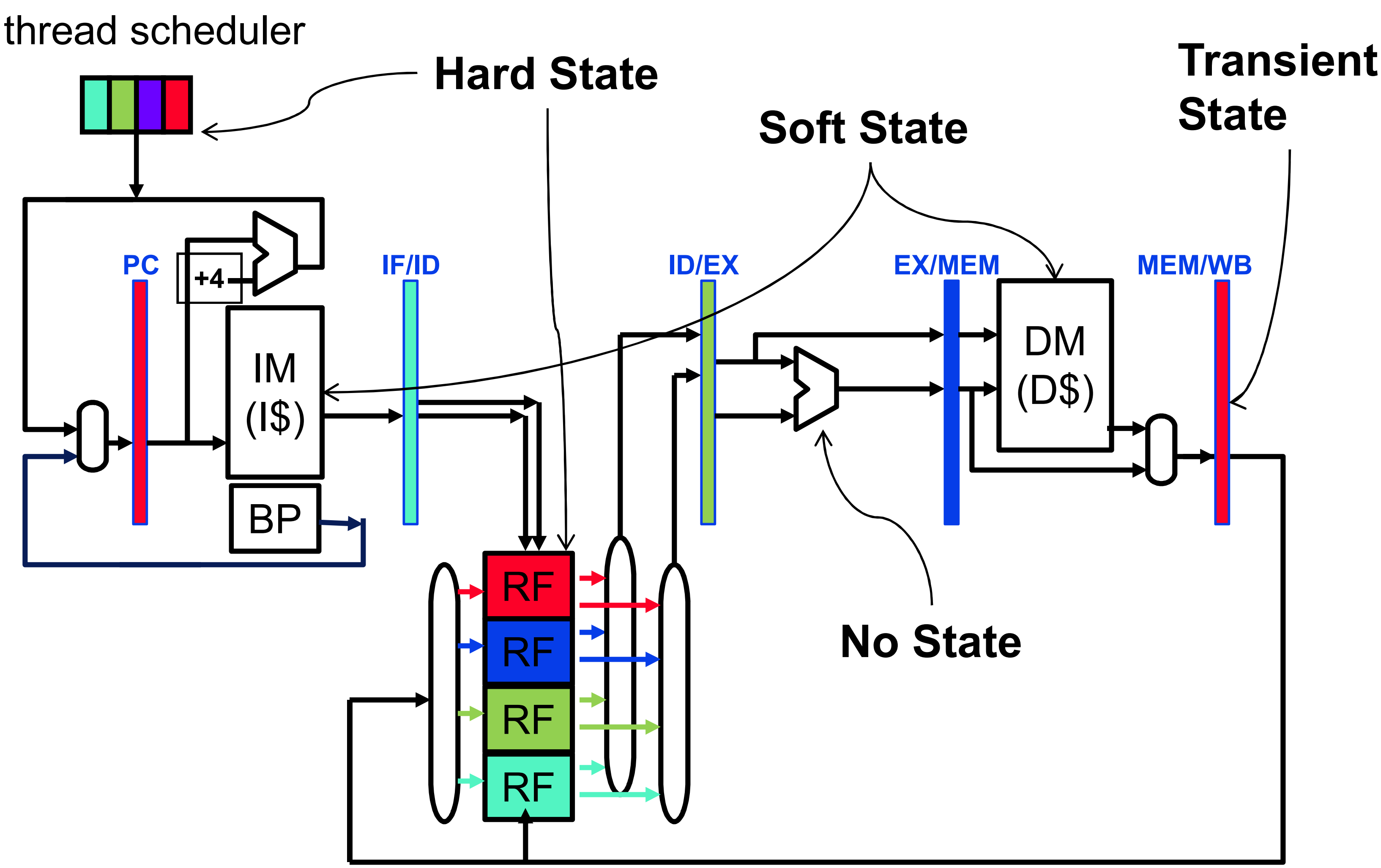
FGMT

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, bpred (BTB, BHT)
  - Dynamically partitioned (like on a multi-programmed uni-processor)
    - TLBs need thread ids, caches/bpred tables don’t
  - Except ordered “soft” state (RAS) is replicated
- Transient state: pipeline latches
  - Must be partitioned … somehow

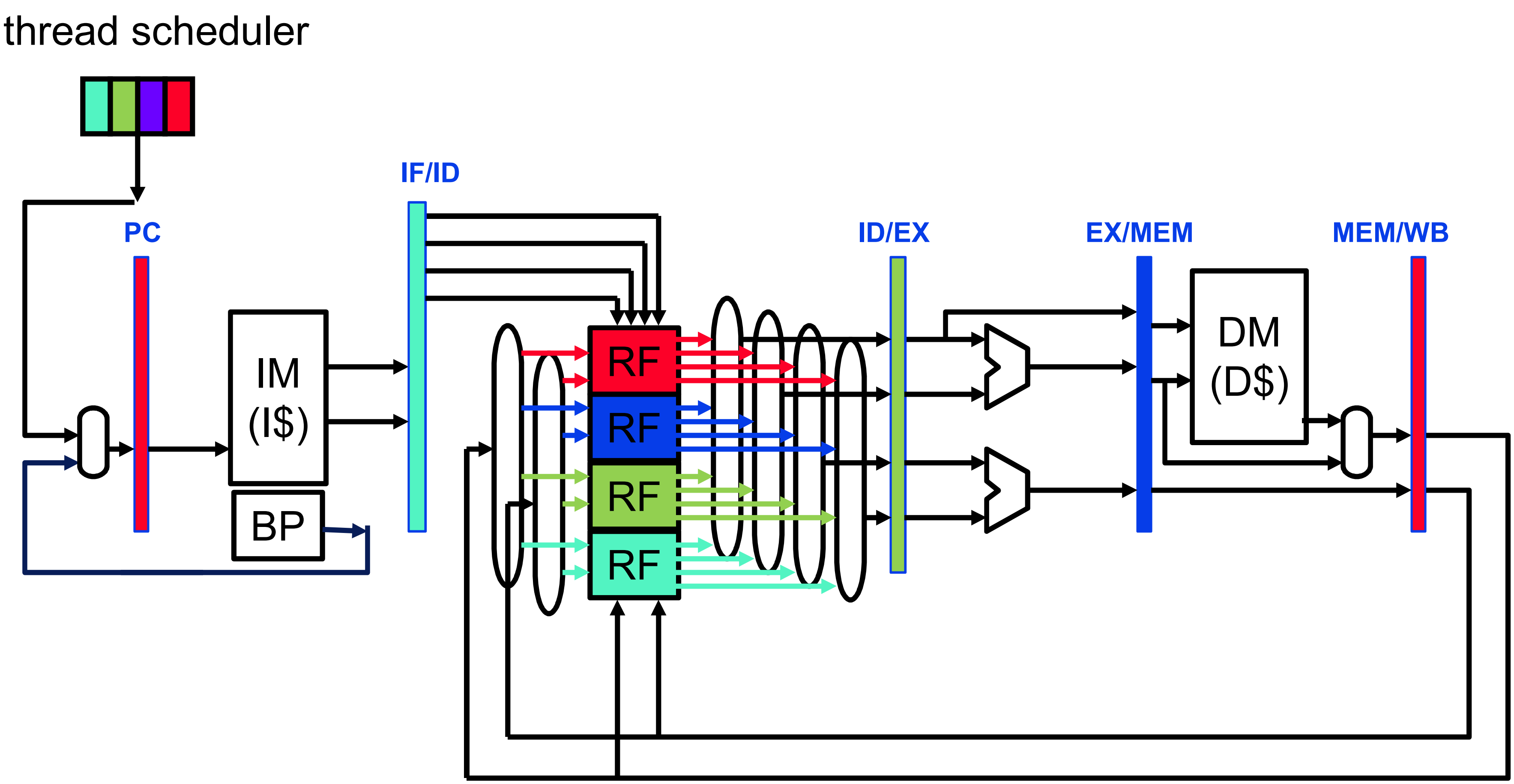
FGMT Datapath (Single Issue)

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



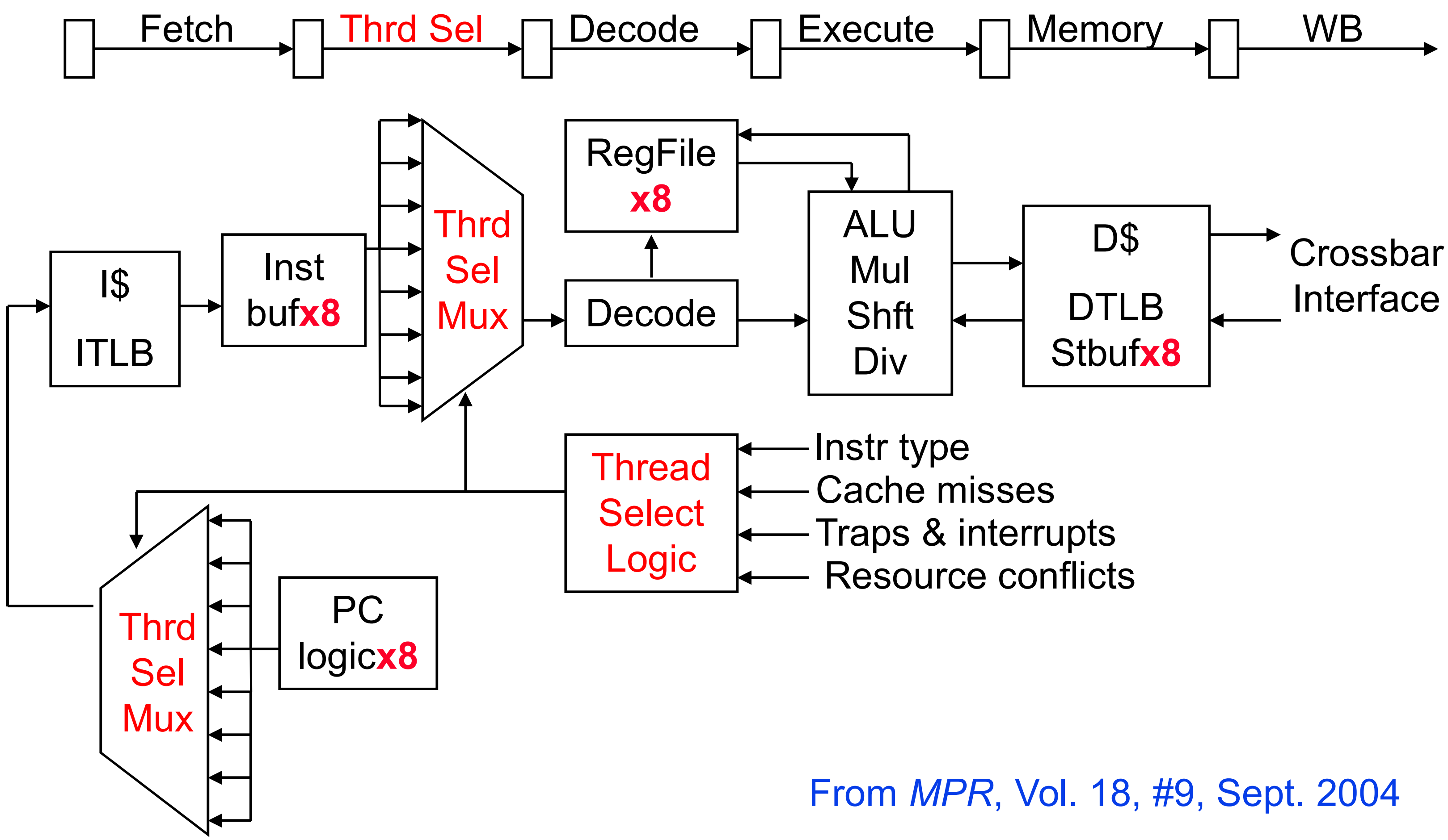
FGMT Datapath (2-Way Issue)

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



Sun Niagara's FGMT Integer pipeline

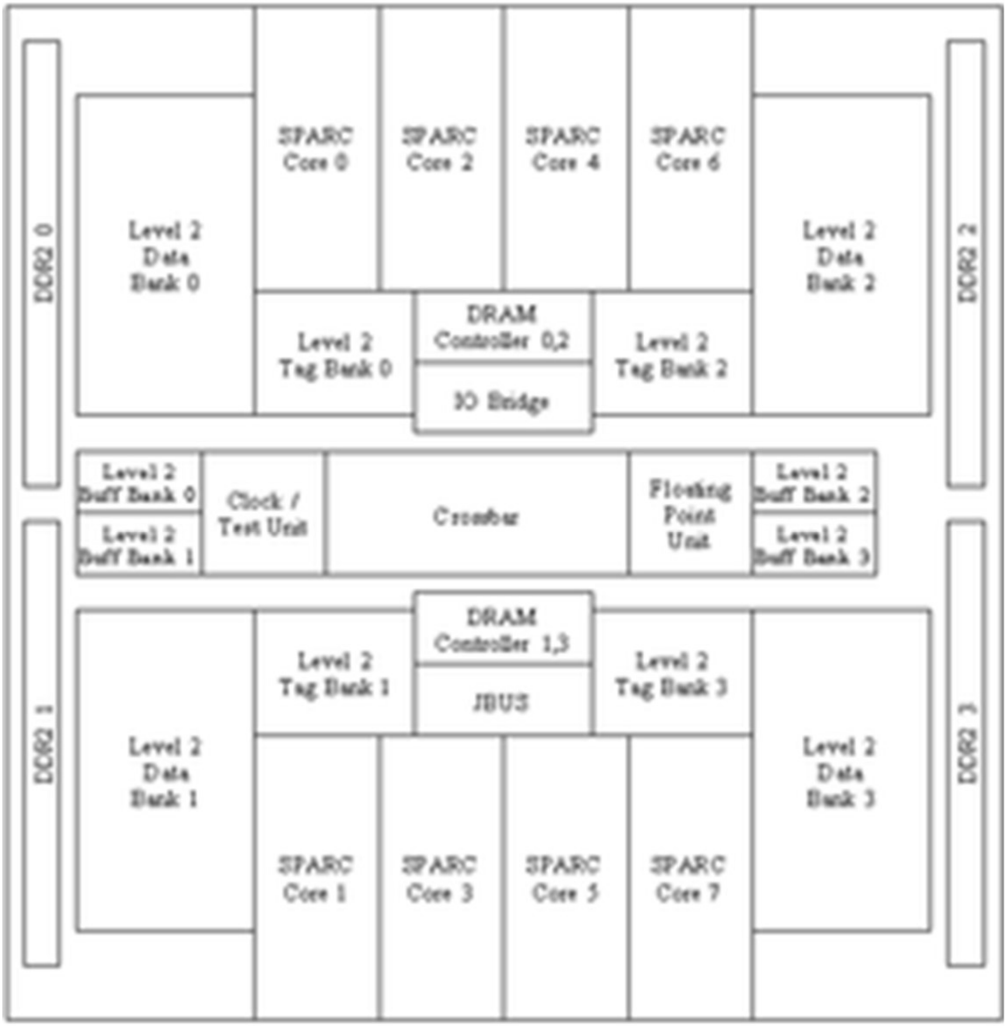
- Cores are simple (single-issue, 6 stage, no branch prediction), small, and power-efficient



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

Sun Niagara's Architecture

- 8 SPARC FGMT datapath cores



Niagra 1 / UltraSPARC T1 / OpenSPARC T1 - Die Micrograph Diagram