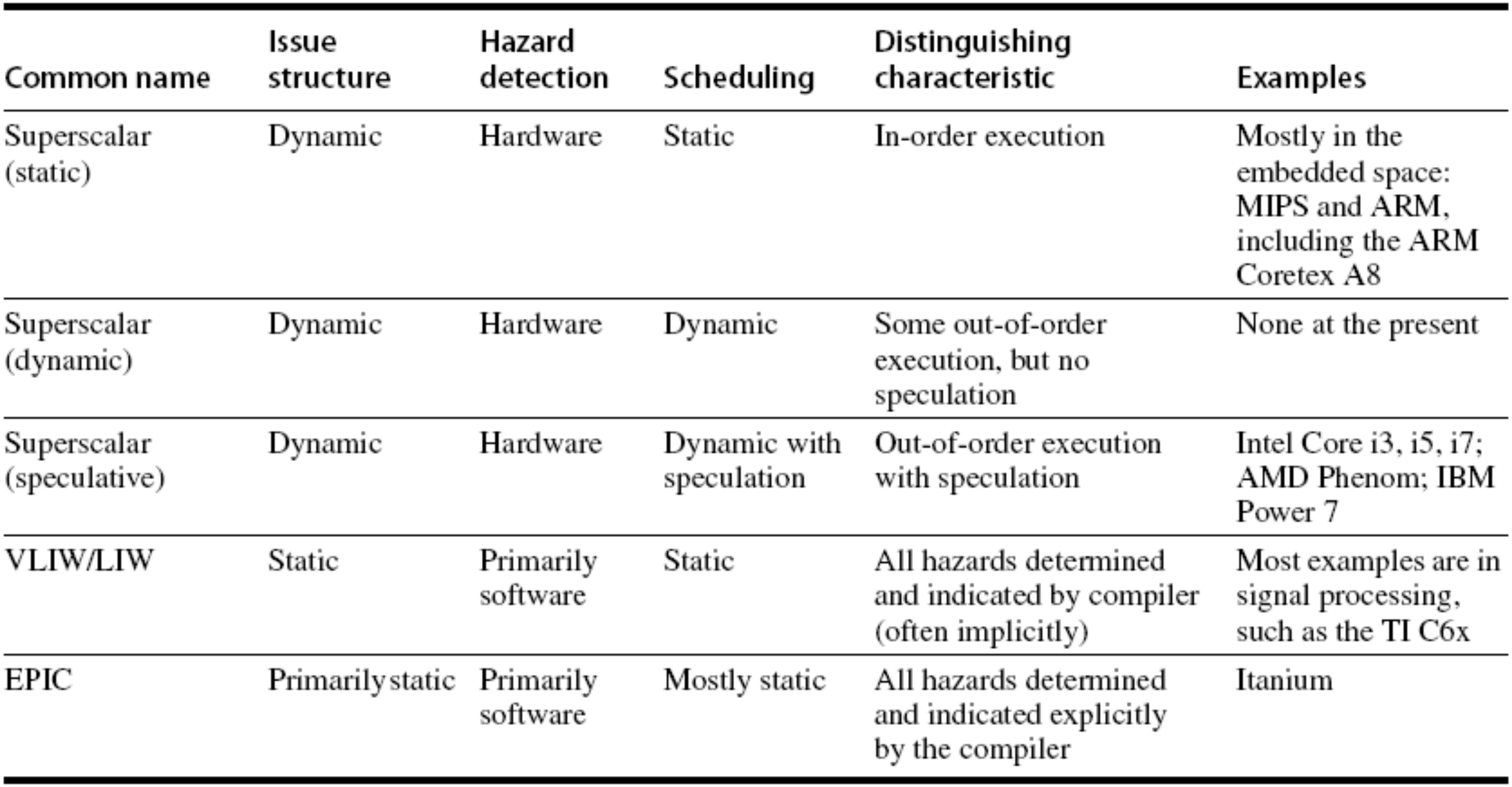
CSE 431
Computer Architecture
Fall 2022

Static SuperScalar (SS) Datapaths

Kiwan Maeng

[Adapted from *Computer Organization and Design, 5th Edition*,
Patterson & Hennessy, © 2014, MK
With additional thanks/credits to Amir Roth, Milo Martin, CIS/UPenn]

Review: Taxonomy of Multiple-Issue Machines



Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Coretex A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

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 be reduced by duplicating the resource or by pipelining the resource

Review: Overview of Dependence Analysis

- To what extent can the compiler (or the datapath) reorder instructions? Are there execution-order constraints?

original	possible?	possible?

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

- Instruction dependencies imply that reordering instructions is not possible
  - true dependence (or, data dep., flow dep.) (cannot reorder)

```
a = .
. = a        RAW, read after write
```

  - anti-dependence (renaming allows reordering)

```
. = a
a = .        WAR, write after read
```

  - output dependence (renaming allows reordering)

```
a = .
a = .        WAW, write after write
```

Multiple Instruction Issue Possibilities

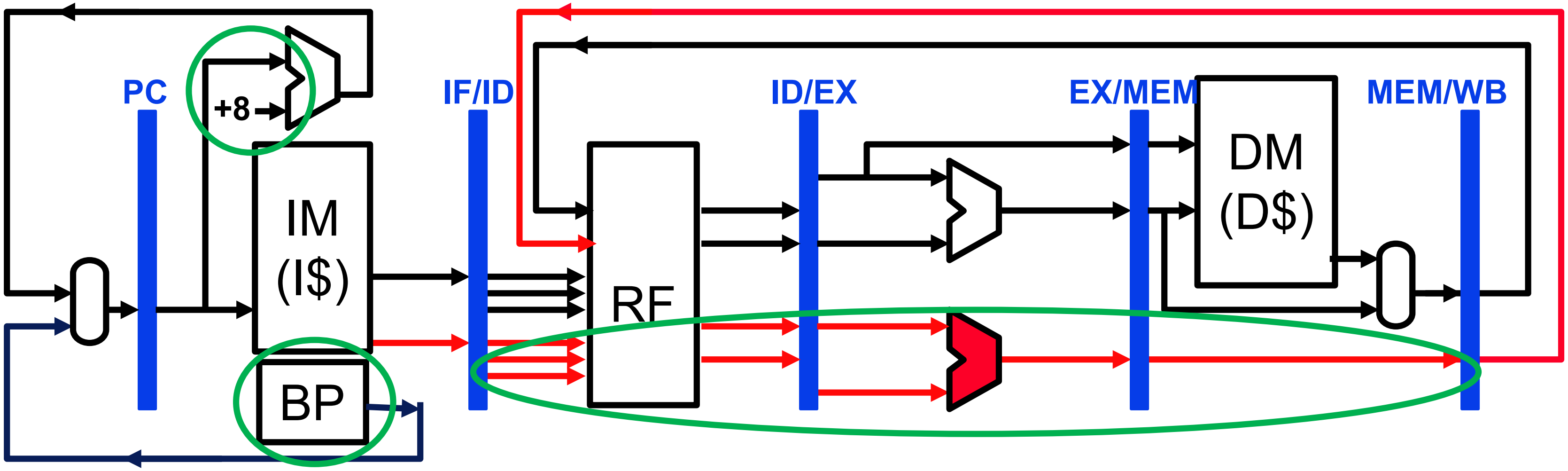
- Fetch and issue **more than one** instruction in a cycle

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)

Shortcomings of VLIW

- VLIW instruction sets are *not* backward compatible between implementations
  - When you move to a new architecture, you need a new binary (recompilation)
- Load instructions do *not* have a deterministic delay, making static scheduling of load instructions by the compiler very difficult
- Statically-scheduled superscalar architectures address the first shortcoming

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

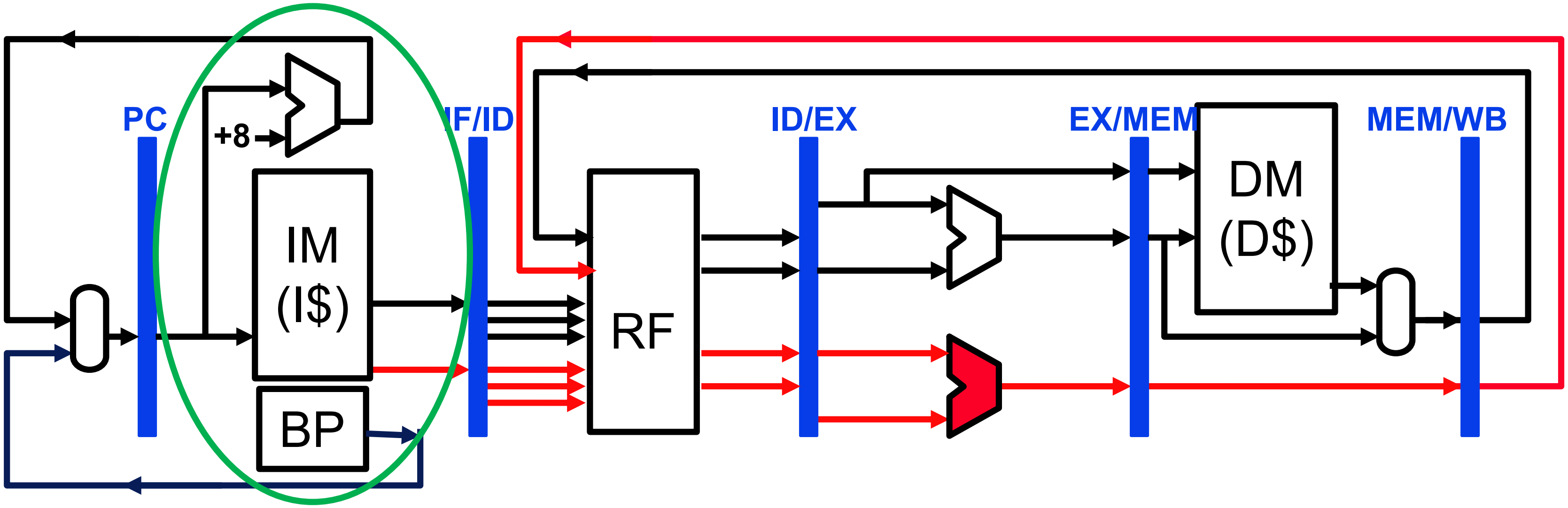
Branches and Instruction-Fetch Inefficiencies

- Branches impede the ability of the processor to fetch instructions, because they make instruction fetching *dependent* on the results of instruction execution

- When the outcome of a branch is not known, the instruction fetcher
  - is stalled, *or*
  - may fetch incorrect instructions

- Instruction misalignment may prevent the decoder from operating at full capacity

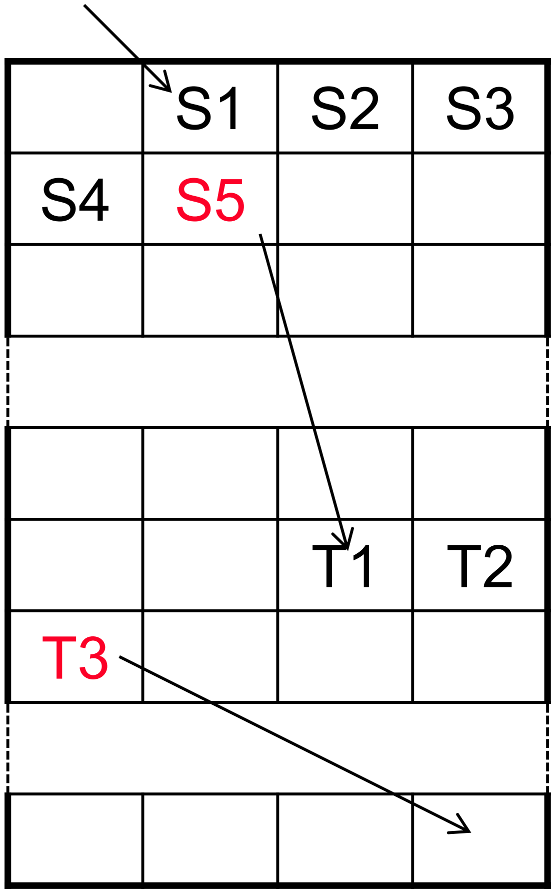
Static SS IF Stage Challenges



- Wide instruction fetch: Fetching a 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* – very good idea in this context

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 4 instr's per cycle with branch prediction)
  - 8 instructions in 4 cycles – so a actual rate of only 2 instr's/cycle
- Experimental Data: the average run length is about six instructions – half of the instructions runs are four instructions or less



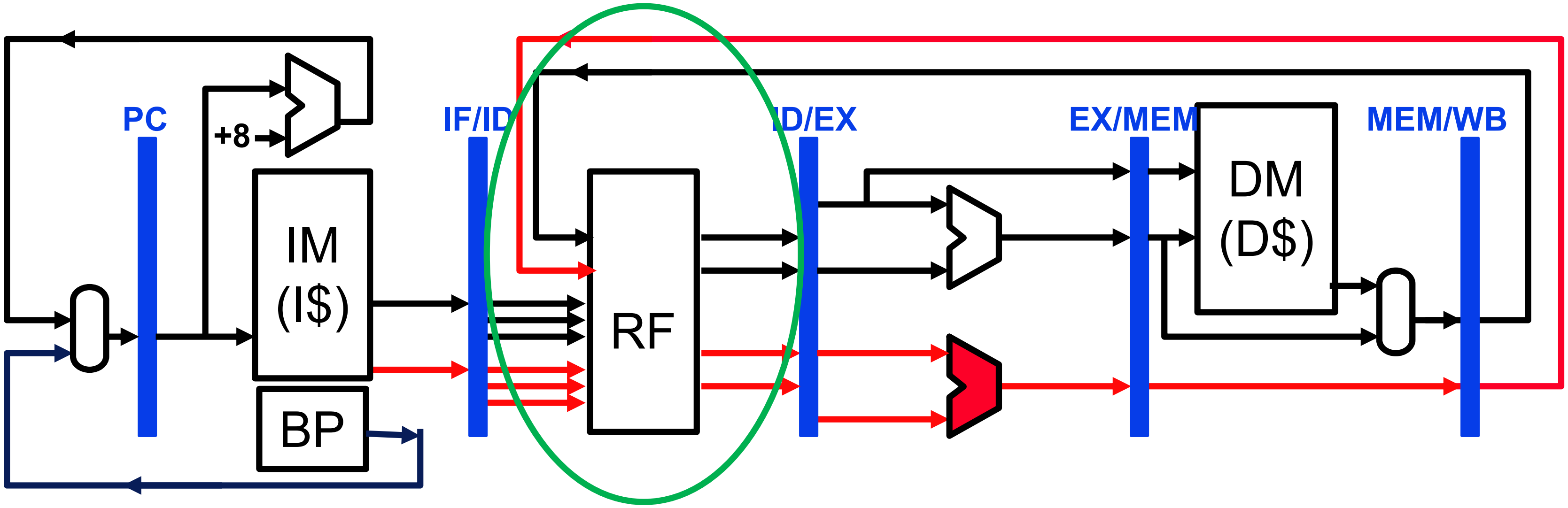
S1 S2 S3

S4 S5

T1 T2

T3

Static SS Dec Stage Challenges



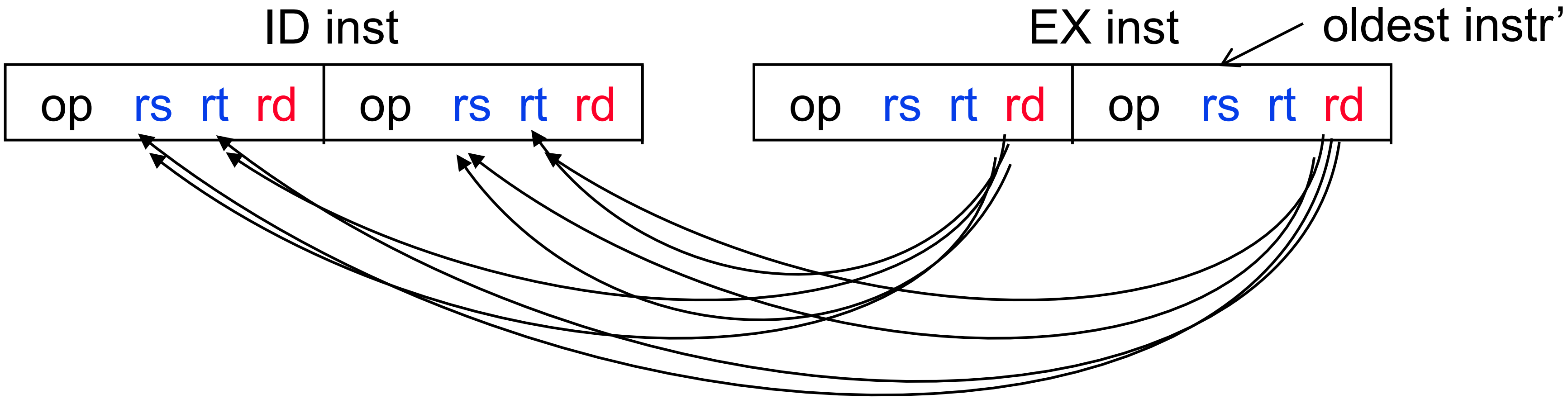
- Have to decode 2 to 8 instr’s *at once* and decide which can issue (be sent to the Exec stage) *in parallel*
  - Duplicated decoders
  - Logic to determine if there are structural hazards and/or data dependencies in the current instr bundle or load-use hazards with the previous instr bundle
  - Logic 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, higher latency, higher power, etc.

Dependency Checking

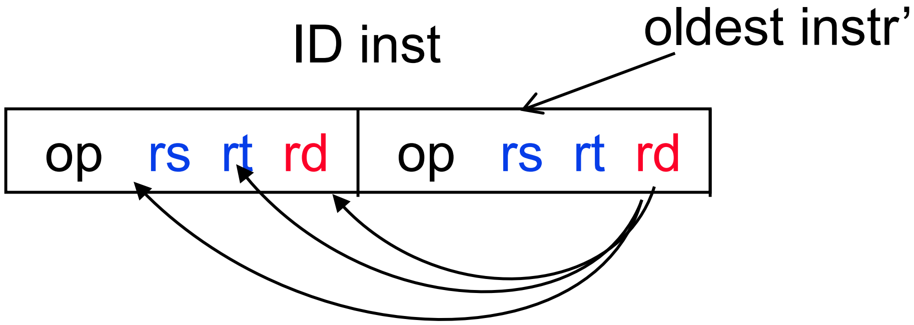
- Need to check for structural hazards (do the 2 (or 4, or 8) instr's need same FU's in EX ?)
  - If so, need to either duplicate the FU's or stall one (or more) of the instr's in the bundle.
- Need to cross check for load-use hazards of the instr's in ID (the "use" instr's – for both of their src operands) to the instr's in EX (the "load" 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)

2-way Dependency Checking

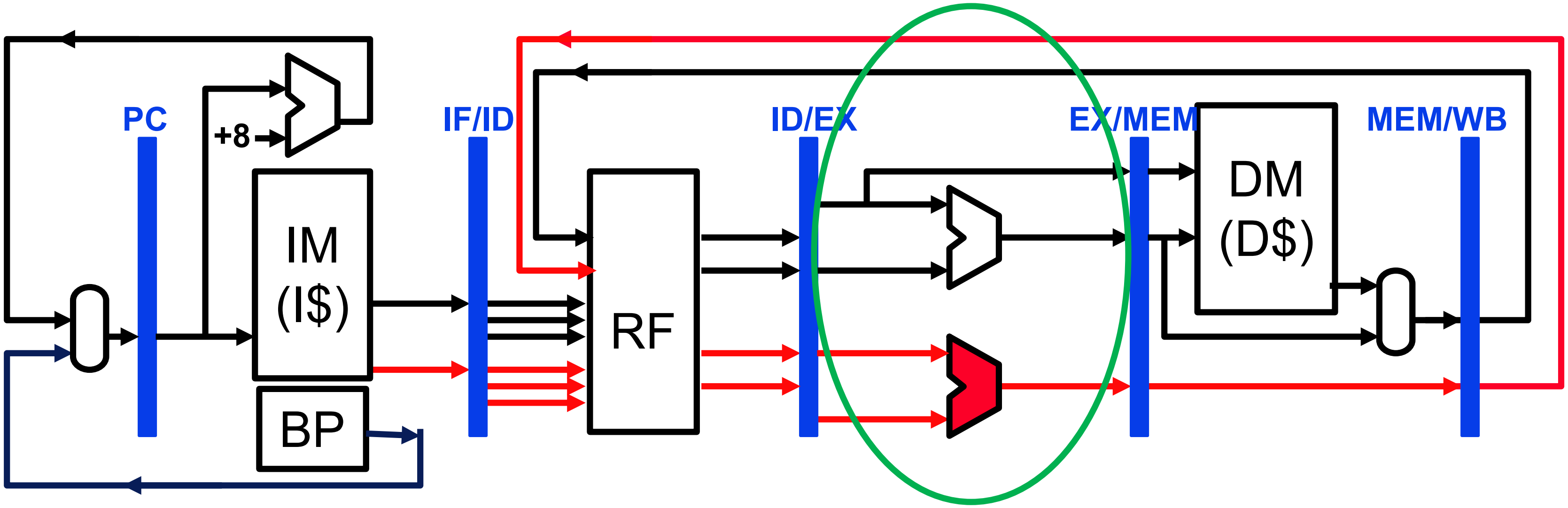
- Cross check for load-use hazards of the 2 instr's in ID (for both src's) to the 2 instr's in EX which gives __8__ load-use dependency checks



- 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



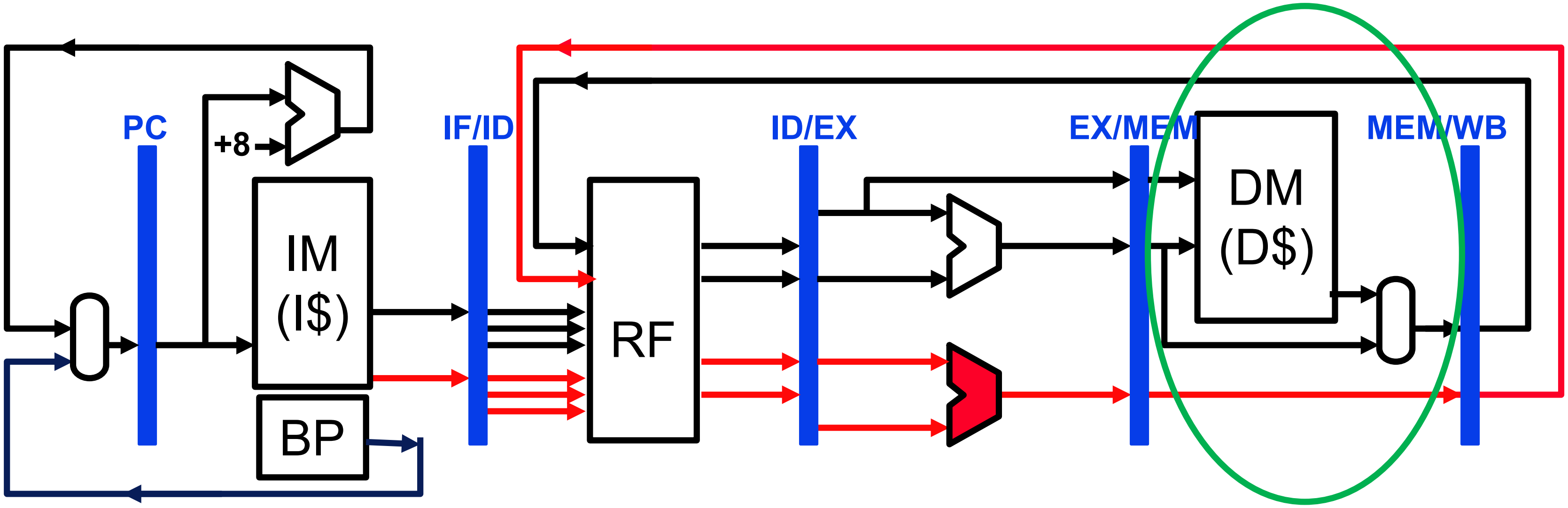
Static SS Exec Stage Challenges



- Need multiple execution units, do we need N (N = # instr in an instr bundle) of every kind?
  - ALUs?   FP dividers?
    - The FP and integer multiplier functional units themselves are pipelined and take multiple stages (this topic is for another day/course!)
  - How many branches per bundle? (we already decided only one)
  - How many loads and/or stores per bundle?

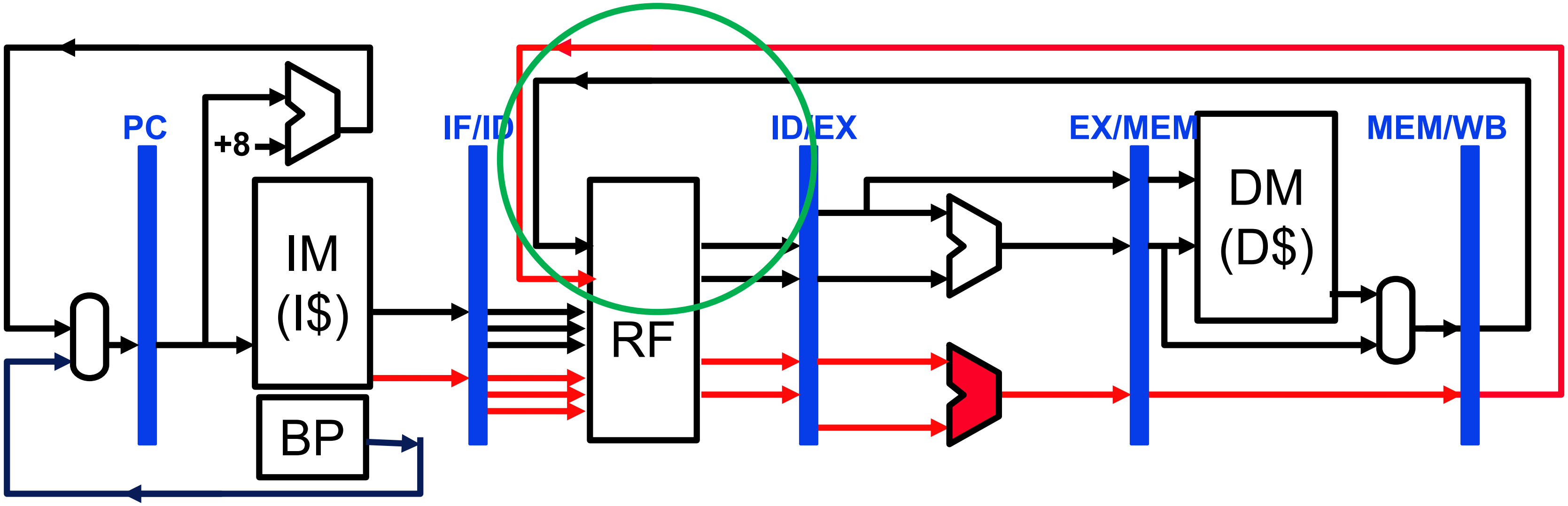
- Usually some mix proportional to the instr mix
  - 2-way: 1 integer (branch, load, store, int) + 1 ALU (int, fp)
  - 4-way: 2 integer + 2 ALU

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 30%), 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
    - Just like it is in register files
  - 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

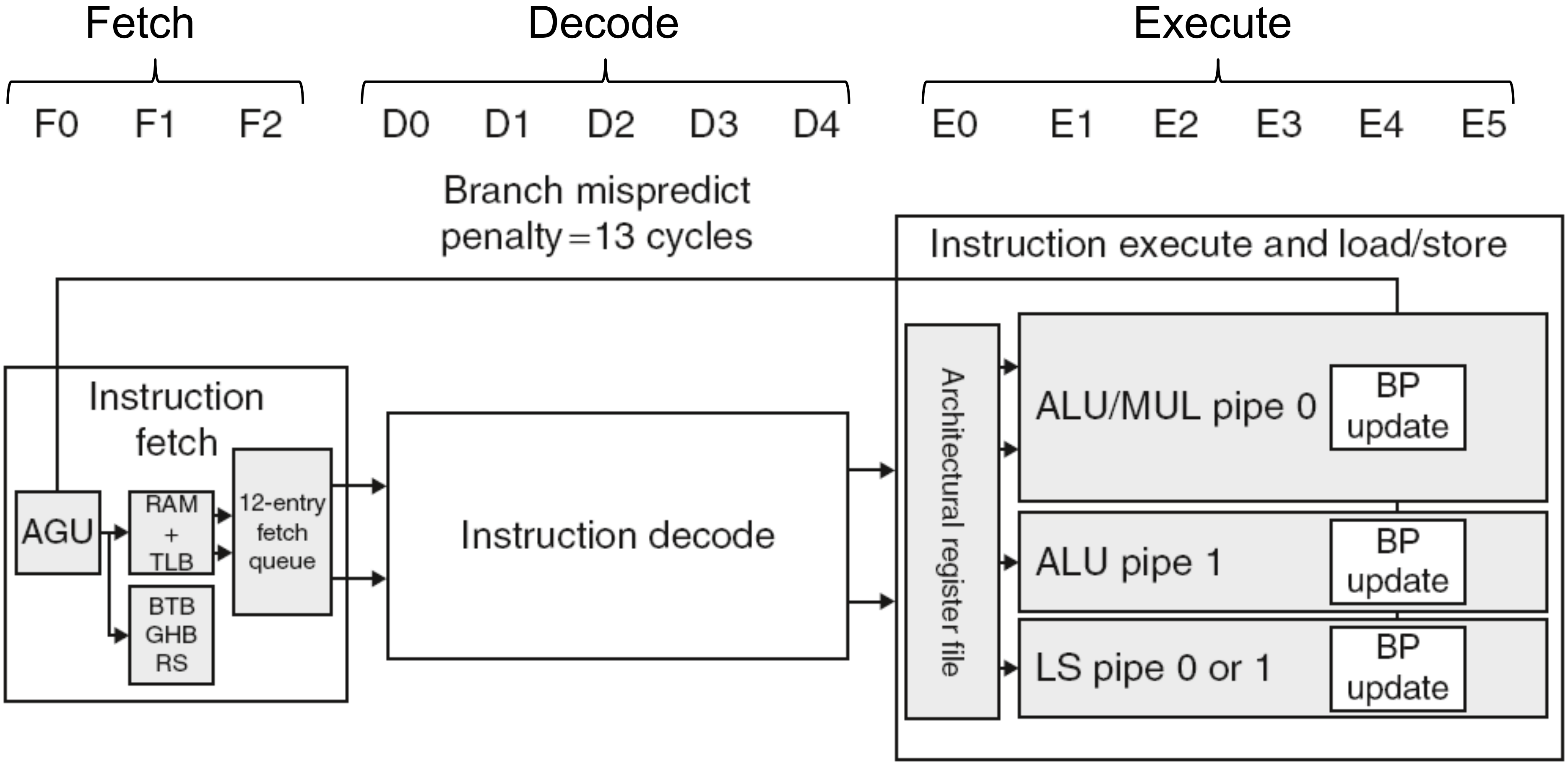
Trends in Static SS Datapath Design

	Pentium	PentiumII	Pentium4	Itanium	ItaniumII	Core2
Year	1993	1998	2001	2002	2004	2006
Width	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
  - There exist 10-way issue machines today
  - Hardware or compiler “scheduling” needed to exploit 4- to 6-way effectively
    - VLIW or EPIC (Itanium)
- Low-power cores usually have an issue width of 2
  - So, advanced scheduling techniques not needed
  - Use multi-threading (stay tuned) to help cope with load-use hazards and cache misses

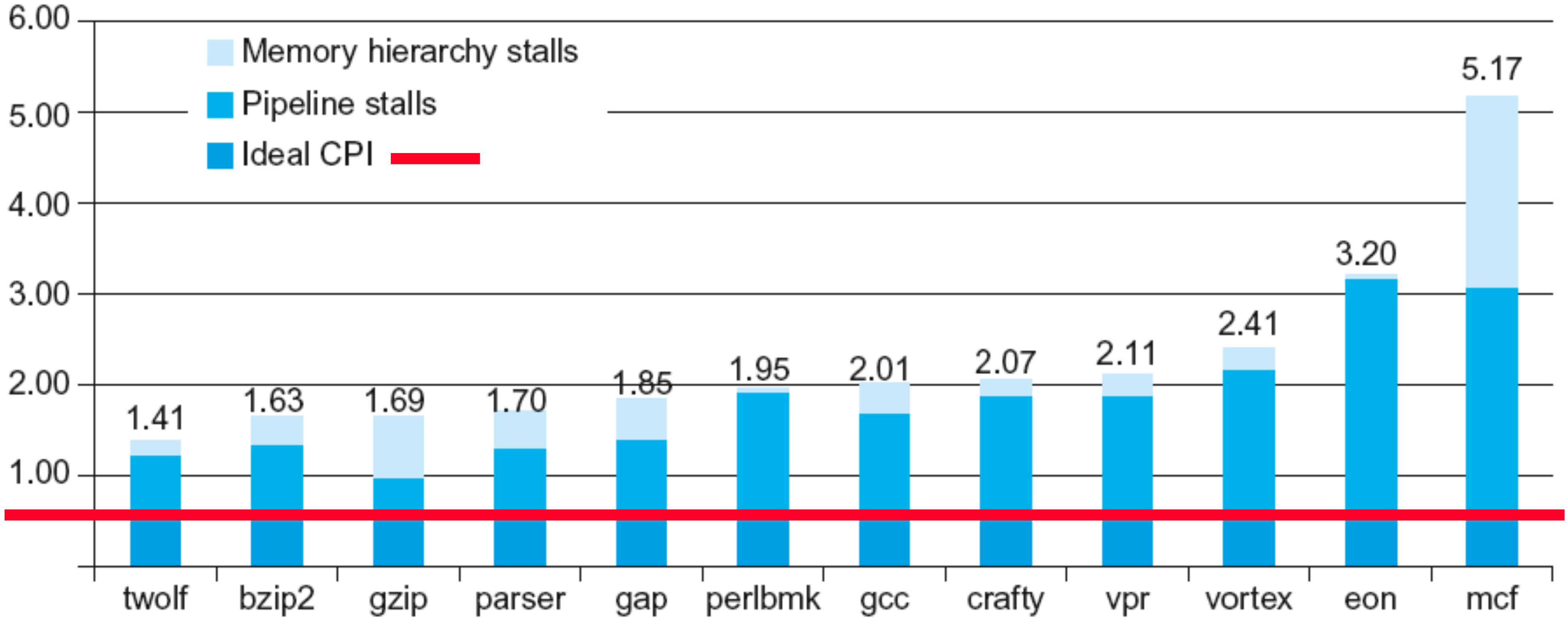
ARM Cortex A8 Pipeline

- 2-wide static (in-order) superscalar, 14-stage pipeline, 1GHz clock
- 12-instruction prefetch buffer; the fetch stage tries to keep it full
- The five stages of the decode pipeline determine if there are dependences between a pair of instructions, which would force sequential execution, and in which pipeline of the execution stages to send the instruction



ARM Cortex A8 Performance (Minnespec Benchmarks)

- 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
  - CPI impact of memory hierarchy is significantly underestimated, as a result of smaller data sizes



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

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)

EPIC

- Explicitly Parallel Instruction Computing (EPIC)

  - Jointly developed by Intel & Hewlett-Packard (HP)

- 64 bit architecture
  - Not extension of x86 series
  - Not adaptation of HP 64bit RISC architecture
- Exploits increasing chip transistors and increasing speeds
- This results in a more complex task for the compiler
- Hardware support for communication of meta-information

  → speculation, predication, and branch hints

EPIC vs VLIW

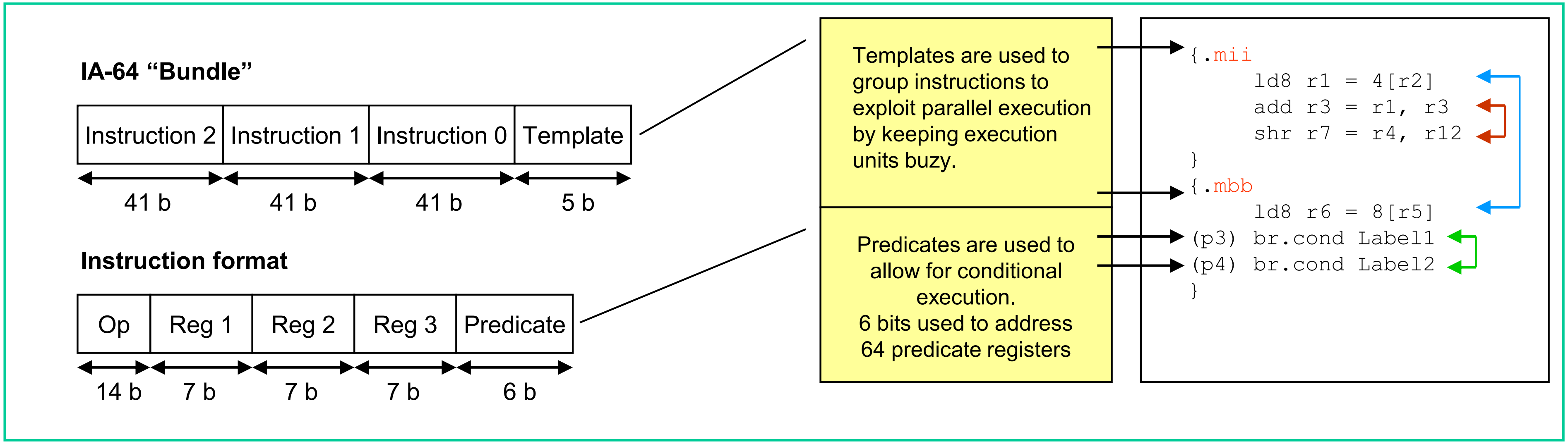
- Shortcomings of VLIW
  - VLIW instruction sets are *not* backward compatible between implementations
  - Load instructions do *not* have a deterministic delay, making static scheduling of load instructions by the compiler very difficult
- EPIC solution
  - Each group of multiple software instructions is called a *bundle*. Each of the bundles has a stop bit indicating if this set of operations is dependent upon by the subsequent bundle. With this capability, future implementations can be built to issue multiple bundles in parallel.
  - The dependency information is calculated by the compiler, so the hardware does not have to perform operand dependency checking.
  - A speculative load instruction is used to speculatively load data before it is known whether it will be used, (bypassing control dependencies), or whether it will be modified before it is used (bypassing data dependencies).

Basic Concepts Behind EPIC

- Instruction level parallelism (ILP)
  - EXPLICIT in machine instruction, rather than determined at runtime by processor
- Long or very long instruction words (LIW/VLIW)
  - Fetch bigger chunks already “preprocessed”
- Predicated Execution
  - Marking groups of instructions for a late decision on “execution”.
- Control Speculation
  - Go ahead and fetch & decode instructions, but keep track of them so the decision to “issue” them, or not, can be practically made later
- Data Speculation (or Speculative Loading)
  - Go ahead and load data early so it is ready when needed, and have a practical way to recover if speculation proved wrong
- Software Pipelining
  - Multiple iterations of a loop can be executed in parallel
- “Revolvable” Register Stack
  - Stack Frames are programmable and used to reduce unnecessary movement of data on procedure calls

Epic Resources and Instructions

- Instruction encoding



IA-64 "Bundle"

Instruction 2	Instruction 1	Instruction 0	Template
41 b	41 b	41 b	5 b

Instruction format

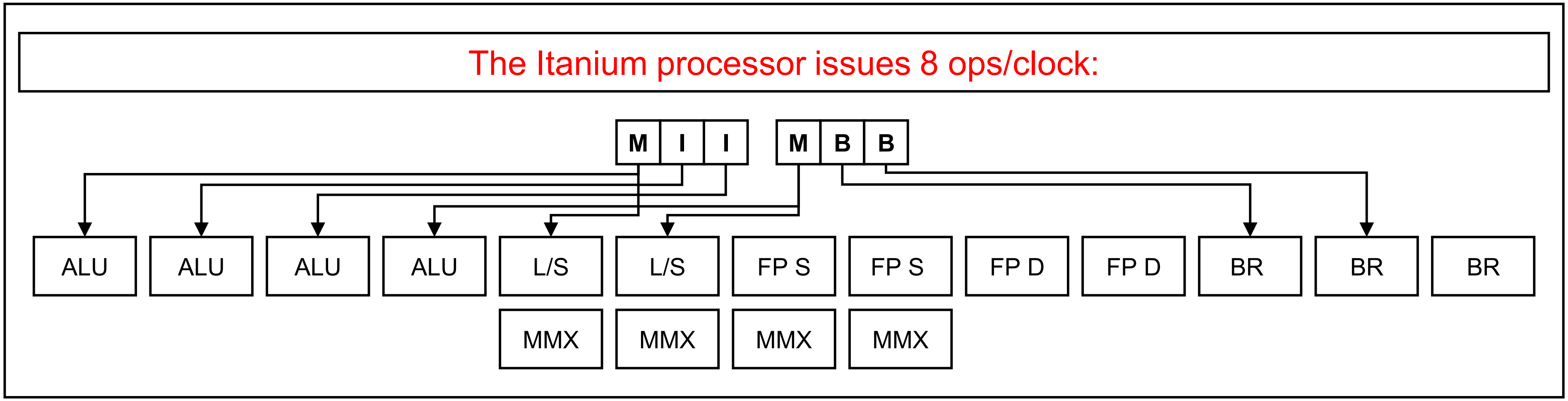
Op	Reg 1	Reg 2	Reg 3	Predicate
14 b	7 b	7 b	7 b	6 b

Templates are used to group instructions to exploit parallel execution by keeping execution units buzy.

Predicates are used to allow for conditional execution.
6 bits used to address 64 predicate registers

```
{.mii
     ld8 r1 = 4[r2]
     add r3 = r1, r3
     shr r7 = r4, r12
}
{.mbb
     ld8 r6 = 8[r5]
(p3) br.cond Label1
(p4) br.cond Label2
}
```

The Itanium processor issues 8 ops/clock:



[Frans Dondorp 2001]

Branch Removal

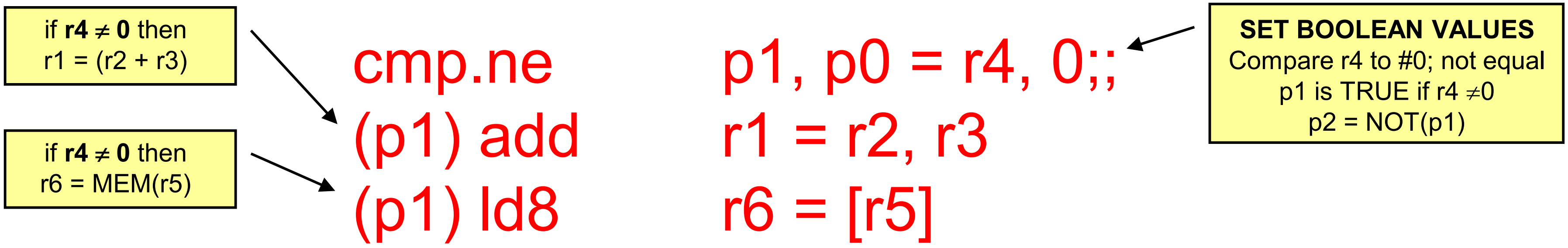
- Branch-prediction is costly
- Cost of misprediction is proportional to pipeline length

Optimizing the use of prediction resources can significantly improve the overall performance

Conditional Instructions can eliminate the need for branches

Predication

Predication: tagging instructions with a boolean value



if **r4 ≠ 0** then
r1 = (r2 + r3)

if **r4 ≠ 0** then
r6 = MEM(r5)

```
cmp.ne       p1, p0 = r4, 0;;
(p1) add     r1 = r2, r3
(p1) ld8     r6 = [r5]
```

SET BOOLEAN VALUES
Compare r4 to #0; not equal
p1 is TRUE if r4 ≠0
p2 = NOT(p1)

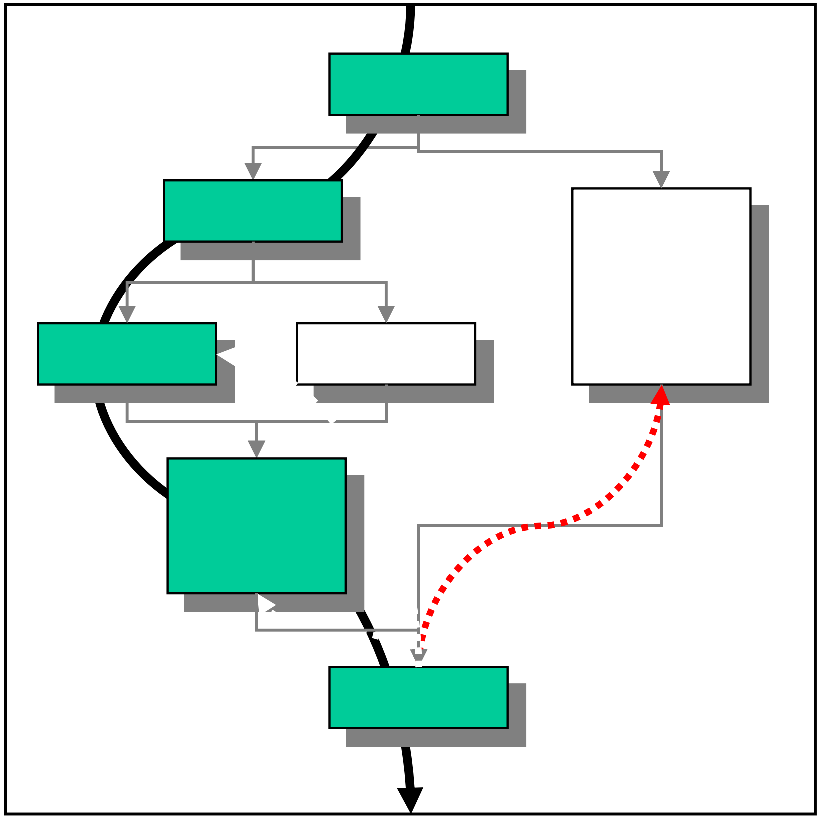
The limitations of conditional instructions are decreased by predication: with predication the amount of conditions to test on equals the number of predicate registers

Speculative Execution

The compiler selects commonly executed blocks

Instruction selection, prioritization and reordering

To enable agressive code-motion done by the compiler, explicitly speculative instructions must be available



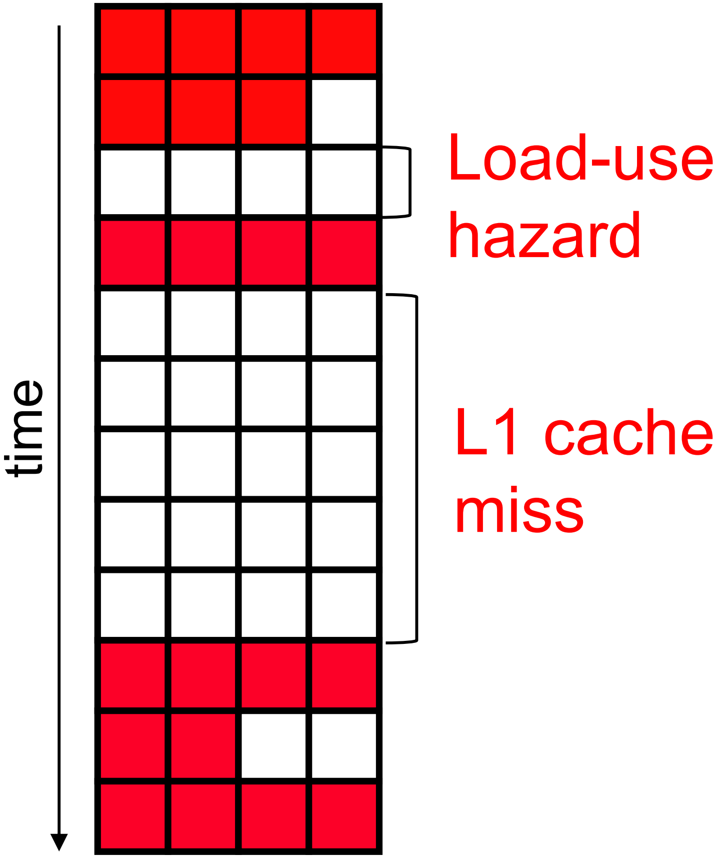
Multithreading (MT)

- Even moderate static superscalars (e.g., 4-way) are not fully utilized
  - Average sustained IPC: 1.5–2 → < 50% utilization due to
    - Mispredicted branches
    - 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

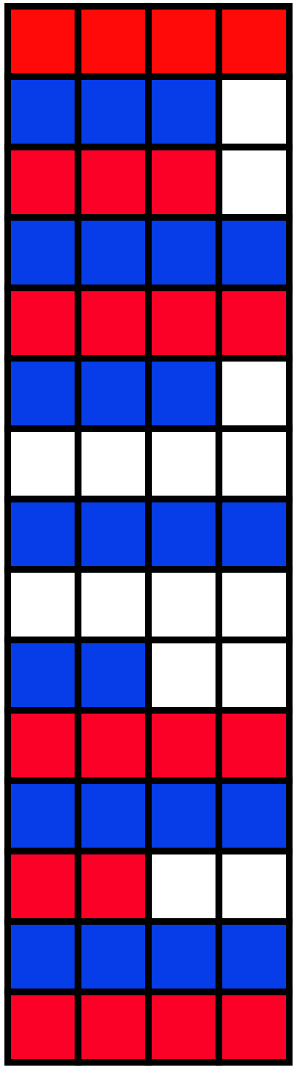
Multithreading Example

- Time evolution of issue slot
  - 4-way datapath



Static SS

- # cycles? # wasted cycle slots?



Multithreaded Static SS

- 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)

Latency vs Throughput

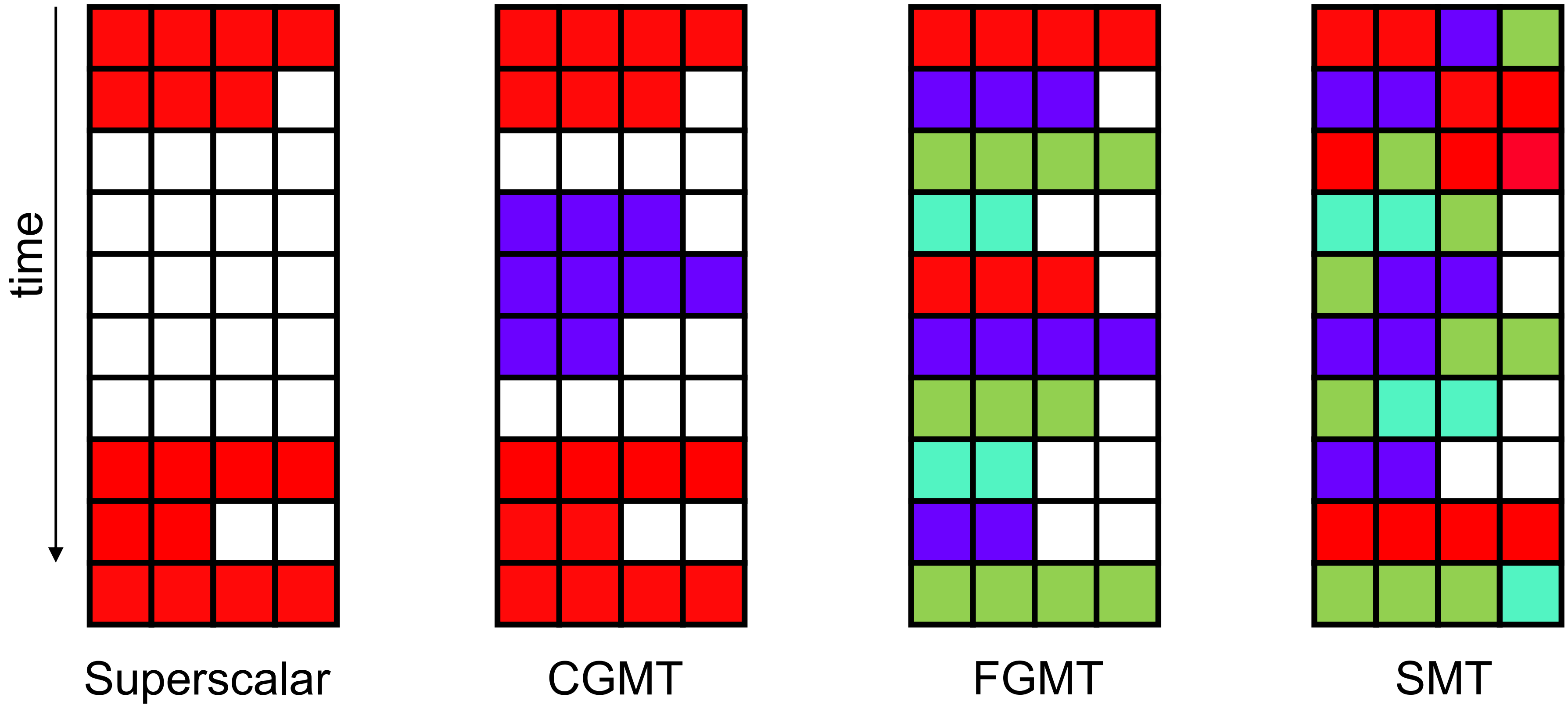
- MT trades (single-thread) latency for throughput
  - – Sharing processor degrades latency of individual threads
  - + But improves aggregate latency of both threads
  - + Improves utilization
- Example
  - Thread A: individual latency=10s, latency with thread B=15s
  - Thread B: individual latency=20s, 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

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
  - Fine-grain multithreading (**FGMT**)
  - Coarse-grain multithreading (**CGMT**)
  - Simultaneous multithreading (**SMT**)

Time Evolution of Issue Slots

- Color = thread



Fine-Grain Multithreading (FGMT)

– Sacrifices significant single thread performance

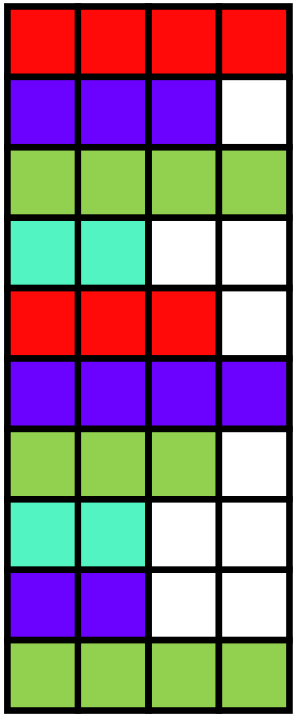
+ Tolerates latencies (e.g., load-use hazards, L1 misses, mispredicted branches, etc.)

- 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



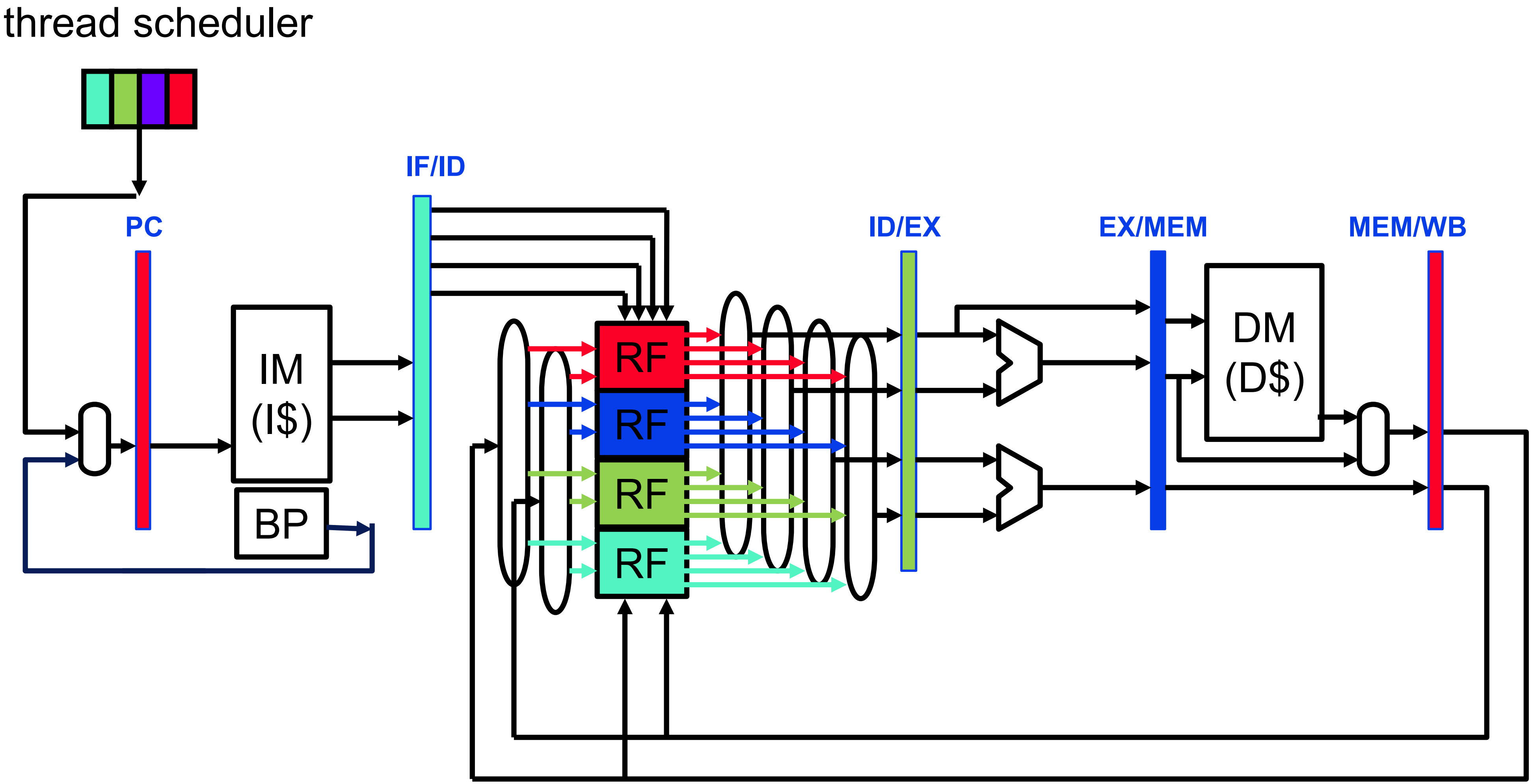
FGMT

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

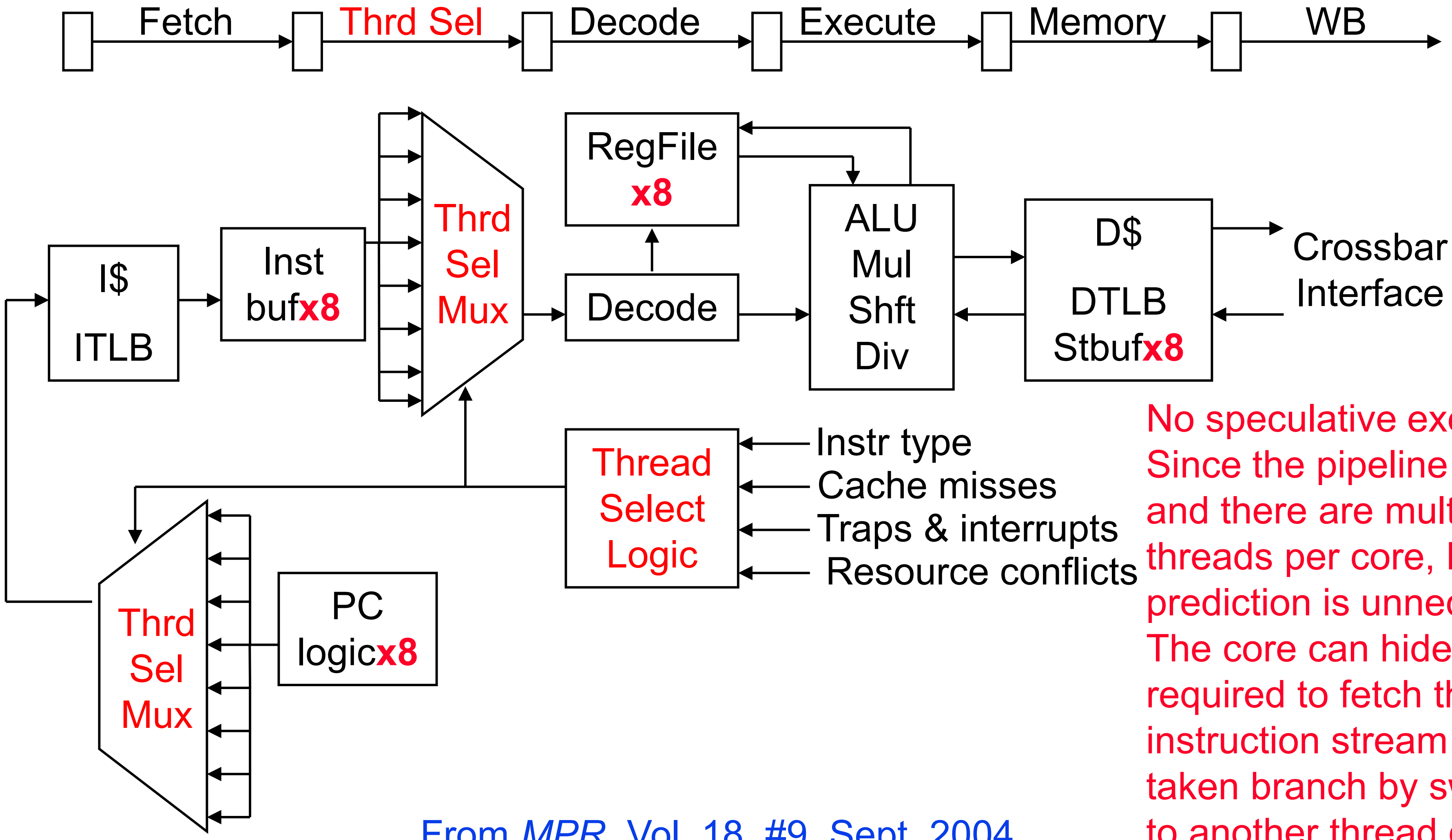
FGMT Datapath

- 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

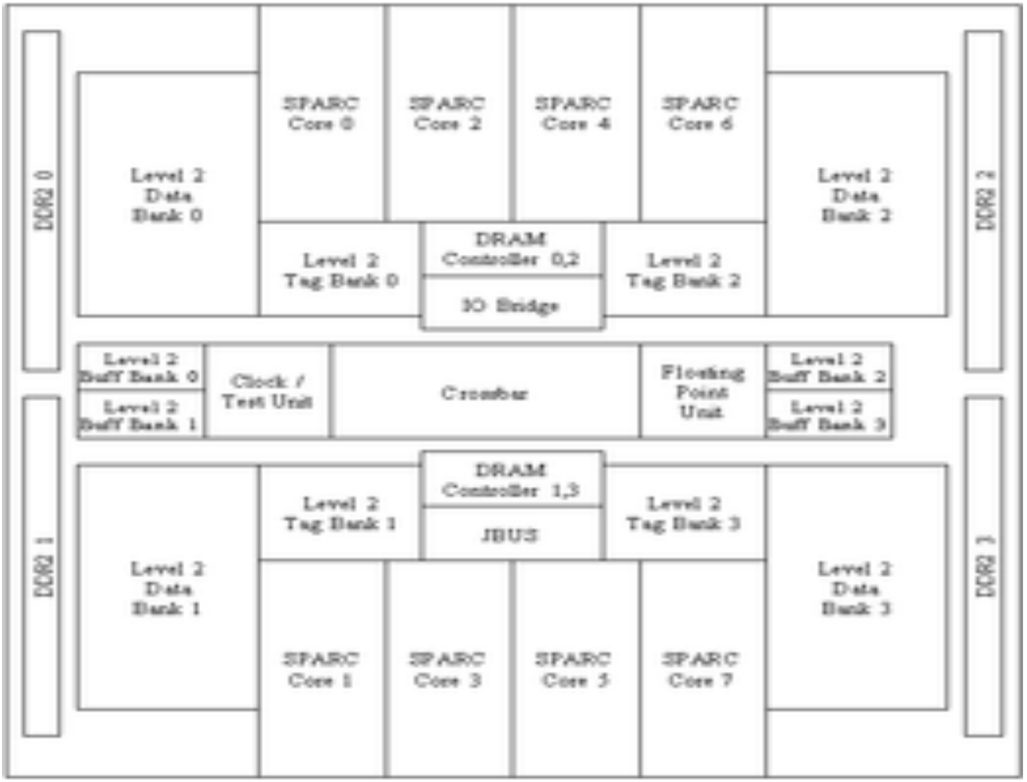


From *MPR*, Vol. 18, #9, Sept. 2004

No speculative execution. Since the pipeline is short and there are multiple threads per core, branch prediction is unnecessary. The core can hide the time required to fetch the new instruction stream on a taken branch by switching to another thread during the clock delay.

Sun Niagara’s Architecture

- 8 SPARC FGMT datapath cores



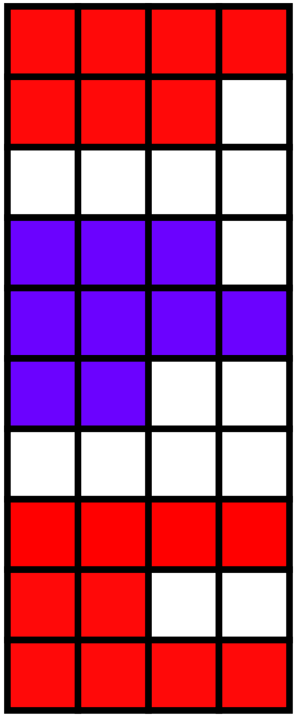
Niagra 1 / UltraSPARC T1 / OpenSPARC T1 - Die Micrograph Diagram ([illegible])

Coarse-Grain Multithreading (CGMT)

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

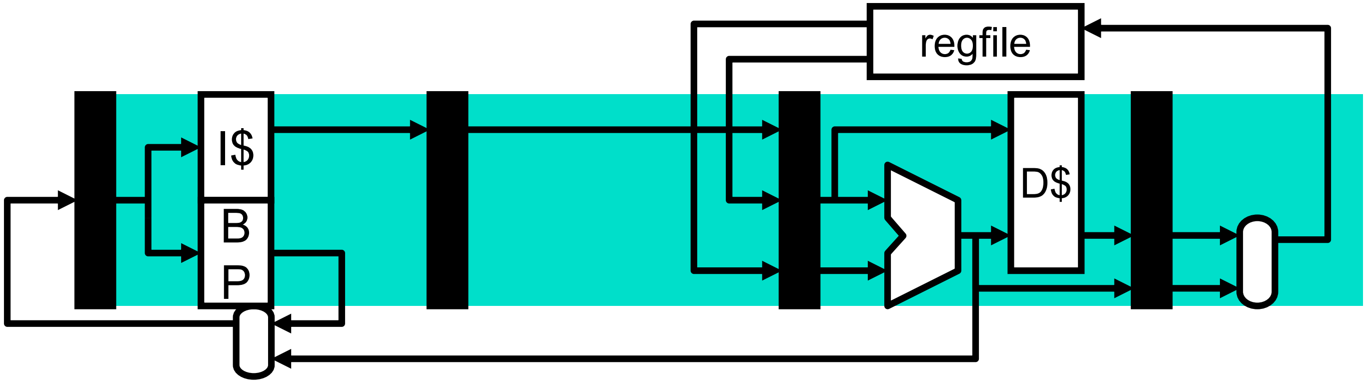
– Tolerates only long latencies (e.g., L2 misses)

- 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

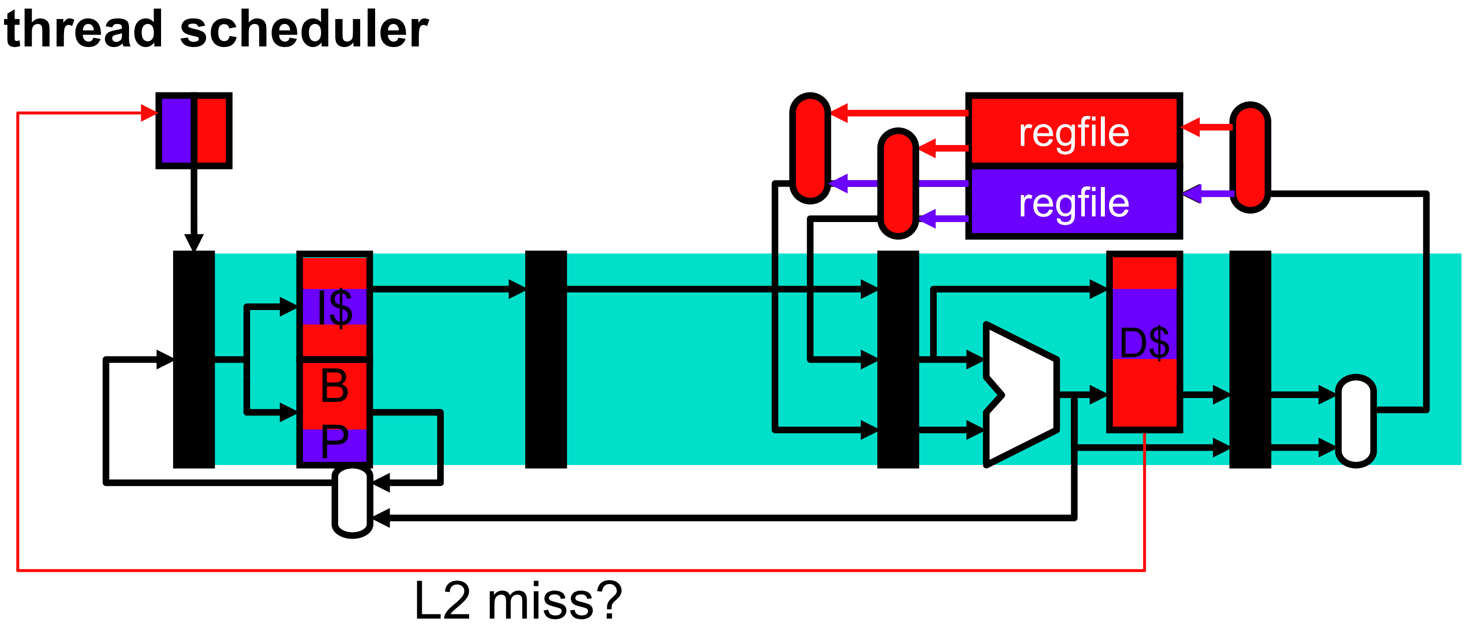


CGMT

Coarse-Grain Multithreaded Architecture

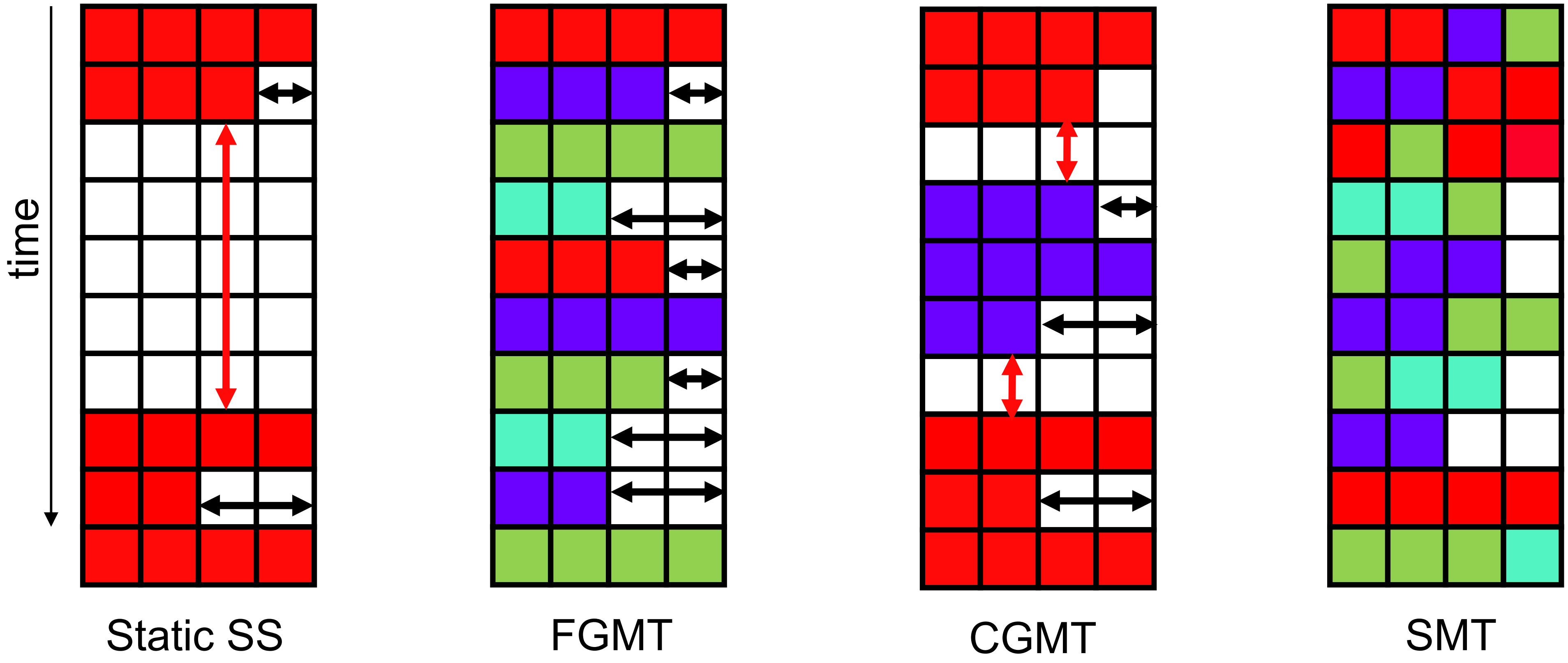


- CGMT



Vertical and Horizontal Under-Utilization

- FGMT and CGMT reduce **vertical under-utilization**
  - Loss of all slots in an issue cycle
- They don’t help with **horizontal under-utilization**
  - Loss of some slots in an issue cycle (in a static SS)



stay tuned…