Register File Organization

Objective

- To understand the organization of large register files used in GPUs
- Identify the performance bottlenecks and opportunities for optimization in accessing the register file

Reading

- S. Liu et.al, "Operand Collector Architecture," US Patent 7,834,881
  - Perspective of a lane
- J. H. Choquette, et. Al., "Methods and Apparatus for Source Operand Caching," US Patent 8,639,882
  - Perspective of instruction scheduling
- B. Coon, et.al, "Tracking Register Usage During Multithreaded Processing Using A Scoreboard Having Separate Memory Regions and Storing Sequential Register Size Indicators," US Patent 7,434,032 B1, October 2008
  - From a dependency perspective
- T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text, Sections 3.2, 3.3

Reading (cont.)

- S. Mittal, A Survey of Techniques for Architecting and Managing GPU Register File, *IEEE TPDS*, January 2017 (Sections 1 & 2)
- GPGPUSim, http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual#Introduction

CPU vs. GPU Register Files

CPU/Core

- Optimized for latency
- Tens of threads
- Small number of ports
- Small relative to caches

GPU/SM

- Optimized for throughput
- Tens of thousands of threads
- Large number of ports (implied)
- Larger relative to caches

(5)

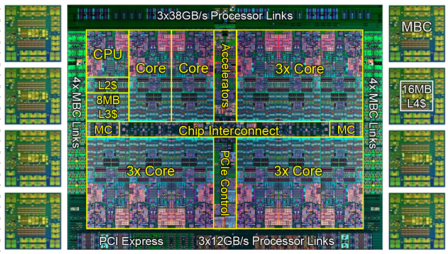
Register File vs. Cache

NVIDIA Pascal

guru3d.com

- 3584 cores/GPU
- Register File is 256 KB/SM
- 14336 KB/GPU
- 4MB L2

IBM Power8

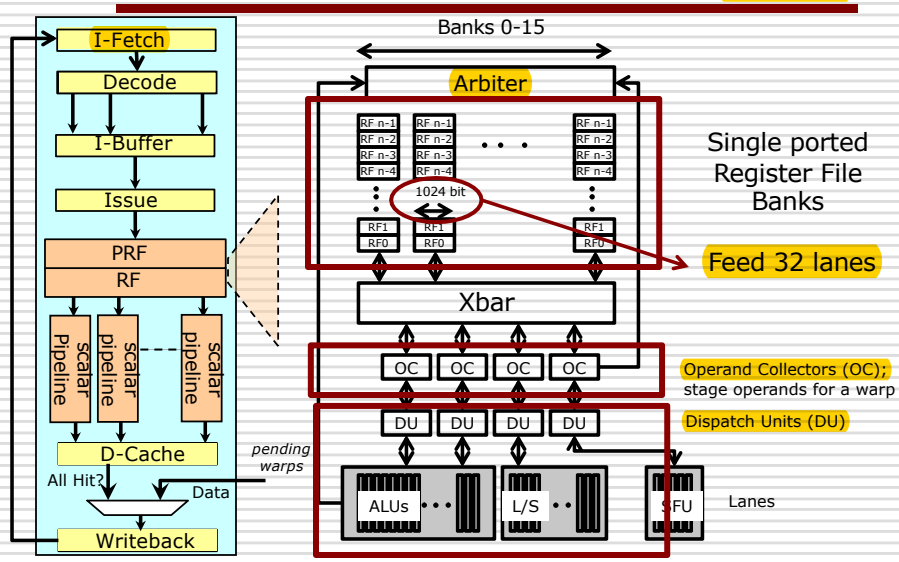


extremetech.com

- 12 cores
- 8KB of RF (+ other registers)
- 96 KB/core L1
- 512KB/core L2, and
- 8MB/core L3

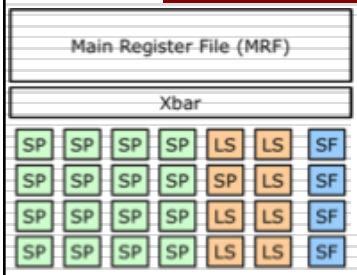
(6)

Register File Access: Recap



(7)

The SM Register File: Bandwidth



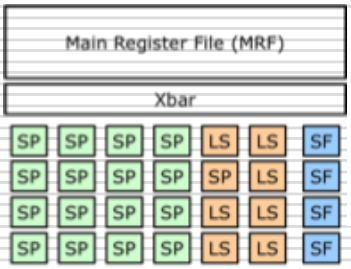
	Architecture	CC	L1 size per SM	L2 size	RF size per SM	# of SMs	Total RF size
G80	Tesla	1.0	None	None	32	16	512
GT200	Tesla	1.3	None	None	64	30	1920
GF100	Fermi	2.0	48	768	128	16	2048
GK110	Kepler	3.5	48	1536	256	15	3840
GK210	Kepler	3.7	48	1536	512	15	7680
GM204	Maxwell	5.2	48	2048	256	16	4096

*From **

- NVIDIA Pascal: 256KB/SM - 64K, 32-bit registers
- Throughput-optimized design
  - 32 instructions/warp, up to 96 operands/warp/cycle
  - Compare to an out-of-order core
- Research Q: Area efficient RFs

(8)

The SM Register File: Power*



	Archi-tecture	CC	L1 size per SM	L2 size	RF size per SM	# of SMs	Total RF size
G80	Tesla	1.0	None	None	32	16	512
GT200	Tesla	1.3	None	None	64	30	1920
GF100	Fermi	2.0	48	768	128	16	2048
GK110	Kepler	3.5	48	1536	256	15	3840
GK210	Kepler	3.7	48	1536	512	15	7680
GM204	Maxwell	5.2	48	2048	256	16	4096

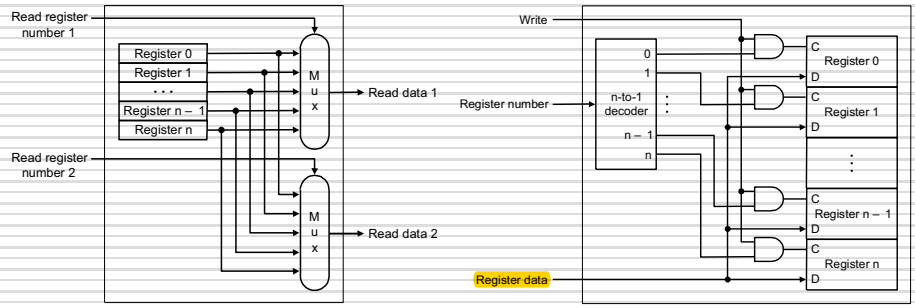
*From **

- Constructed with fast (leaky) transistors
- Up to 20% of chip dynamic power consumption can be consumed by the RF

Research Q: How do you architect, organize, & manage RFs for energy efficiency?

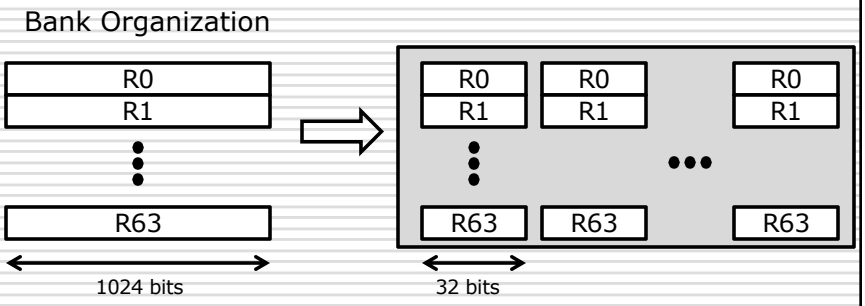
*S. Mittal, A Survey of Techniques for Architecting and Managing GPU Register File, *IEEE TPDS*, January 2017

Multi-ported Register Files



- Multi-ported register file organization
- Area and delay grows with #ports
- Use multiple banks with single read and write ports to emulate multiple ports

Multiple Banks

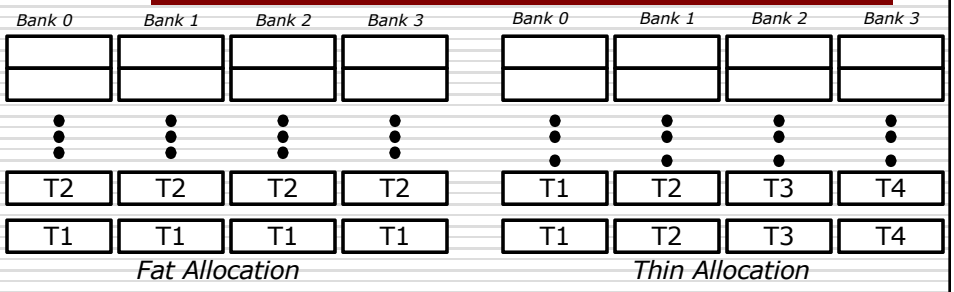


- 1R/1W – single read port and single write port per bank
- Each access to a register bank produces the same named register per lane, e.g., R0
- Concurrently access multiple banks

S. Mittal, A Survey of Techniques for Architecting and Managing GPU Register File, IEEE TPDS, January 2017

(11)

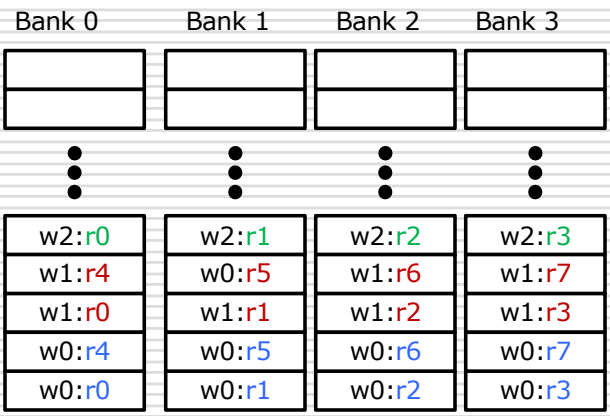
Thread Register Allocation



- Interpret thread as warp (in patents)
- Operands (registers) of a thread can be mapped across register banks in different ways
  - Thin, fat, and mixed
- Skewed allocation: Goal is maximum bandwidth
- # of cycles to access operands for an instruction?

(12)

Allocation Example



(13)

Skewed Allocation: Example

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Skewed allocation

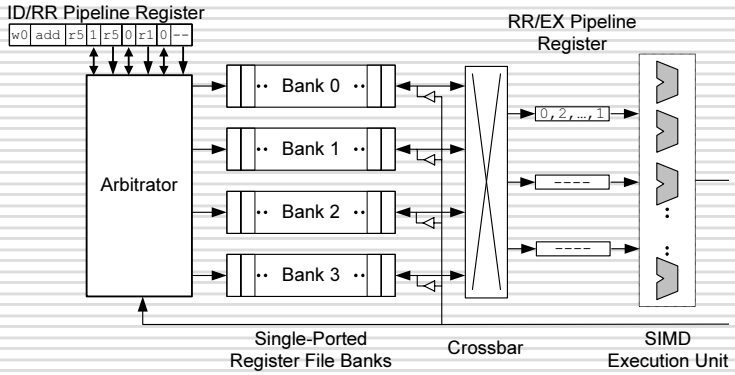
Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	~~w1:r4~~	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
~~w0:r4~~	~~w0:r5~~	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

(14)

Register File Access

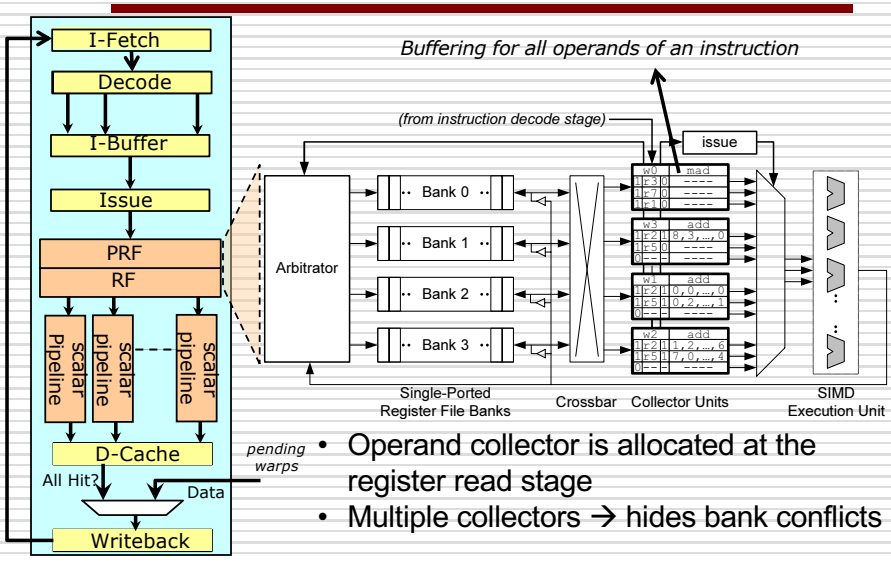
- Why and when do bank conflicts occur?



Operand access for a warp may take several cycles → need a way to collect operands!

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

Register File Access



- Operand collector is allocated at the register read stage
- Multiple collectors → hides bank conflicts

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

Bank Conflicts (1)

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Fat allocation

```
i1:  mad   r2_2, r5_1, r4_0, r6_2
i2:  add   r5_1, r5_1, r1_1
```

Decode Cycle	Warp	Instruction
0	w3	i1: mad r2, r5, r4, r6
1	w0	i2: add r5, r5, r1
4	w1	i2: add r5, r5, r1

Bank conflict (read vs. write)

Bank	Cycle 1	2	3	4	5	6
0	w3:i1:r4					
1	w3:i1:r5	w0:i2:r1	w0:i2:r5	w1:i2:r1	w0:i2:r5	w1:i2:r5
2	w3:i1:r6		w3:i1:r2			
3						
EU		w3		w0		

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

Bank Conflicts (2)

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Not Skewed

Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Bank	Cycle 1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

Improved concurrency

- Skewed allocation
- Bank conflicts remain within a warp
- Bank conflicts improved across warps

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

Bank Conflicts (2)

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Cycle

Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

- Skewed allocation
- Bank conflicts remain within a warp
- Bank conflicts improved across warps

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

(19)

Bank Conflicts (2)

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Cycle

Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

- Skewed allocation
- Bank conflicts remain within a warp
- Bank conflicts improved across warps

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

(20)

Bank Conflicts (2)

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Cycle

Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

- Skewed allocation
- Bank conflicts remain within a warp
- Bank conflicts improved across warps

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

Bank Conflicts (2)

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r7	w1:r4	w1:r5	w1:r6
w1:r3	w1:r0	w1:r1	w1:r2
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

```
i1:  add   r1, r2, r5
i2:  mad   r4, r3, r7, r1
```

Cycle	Warp	Instruction
0	w1	i1: add $r1_2$, $r2_3$, $r5_2$
1	w2	i1: add $r1_3$, $r2_0$, $r5_3$
2	w3	i1: add $r1_0$, $r2_1$, $r5_0$
3	w0	i2: mad $r4_0$, $r3_3$, $r7_3$, $r1_1$

Cycle

Bank	1	2	3	4	5	6
0		w2:r2		w3:r5		w3:r1
1			w3:r2			
2		w1:r5		w1:r1		
3	w1:r2		w2:r5	w0:r3	w2:r1	w0:r7
EU			w1	w2	w3	

In bank 0

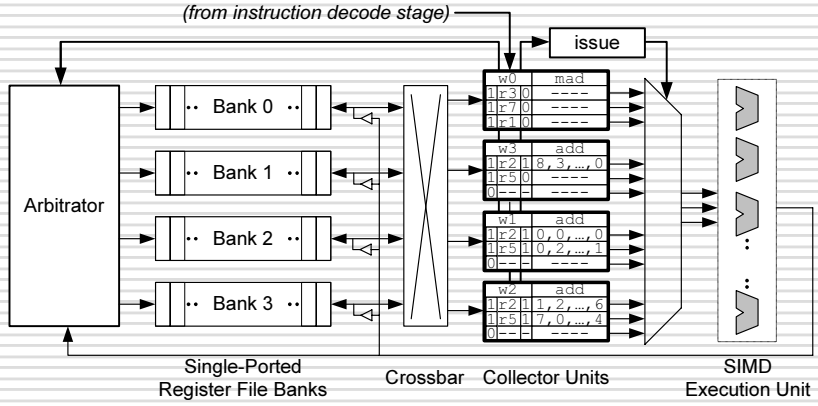
In bank 2

In bank 3

- Skewed allocation
- Bank conflicts remain within a warp
- Bank conflicts improved across warps

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

Potential Hazards



- With multiple instructions in flight from a warp, WAR hazards are possible with repeated bank conflicts
  - Two instructions from the same warp in the OC

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

(23)

Structural Hazards

- Structural hazards cause instruction replays
  - Instructions are maintained in the instruction buffer until it completes
  - Switch to another warp rather than stall the pipeline
- Examples:
  - OC is full (analogy with ROB being full)
  - Load/store miss in the cache: replay is to fetch the instruction from the cache
  - Register bank conflicts
  - Shared memory bank conflicts
  - Others
- NVIDIA profiler will provide statistics on instruction repay

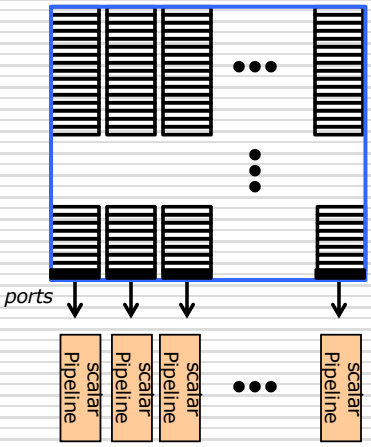
(24)

Bank and Register File Access Organization

- How do we organize per thread RFs within the main register file (MRF)
  - Maximize concurrency
  - Minimize bank conflicts
  - Cost grows multiplicatively with the number of ports
  - Using lower port count memories to emulate high port count memories
- Access organization should support dynamic warp formation
- Goal: Full bandwidth access to lanes

(25)

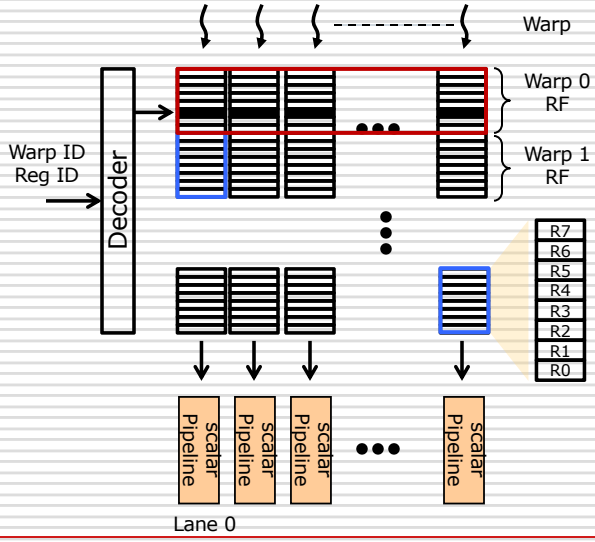
A Multi-ported Register File



- Multi-ported register file
- Expensive in area and energy
- Feasibility in practice
- 32 * 3 read operands/cycle

(26)

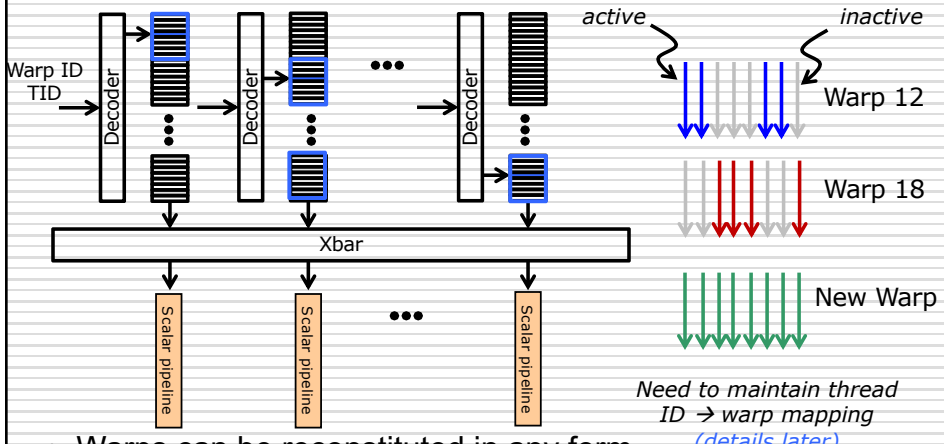
A Baseline Register File



- Wide register format
- Static assignment of threads to lanes
- Single decoder across the bank
- Comprised of 1R/1W banks

(27)

Register File: Warp Formation (1)

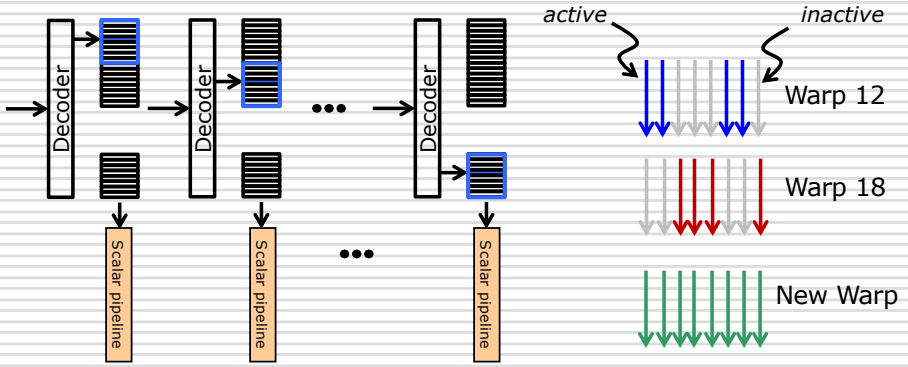


Need to maintain thread ID → warp mapping
(details later)

- Warps can be reconstituted in any form
- Bank conflicts can reduce performance
- Number of banks vs. number of lanes?

(28)

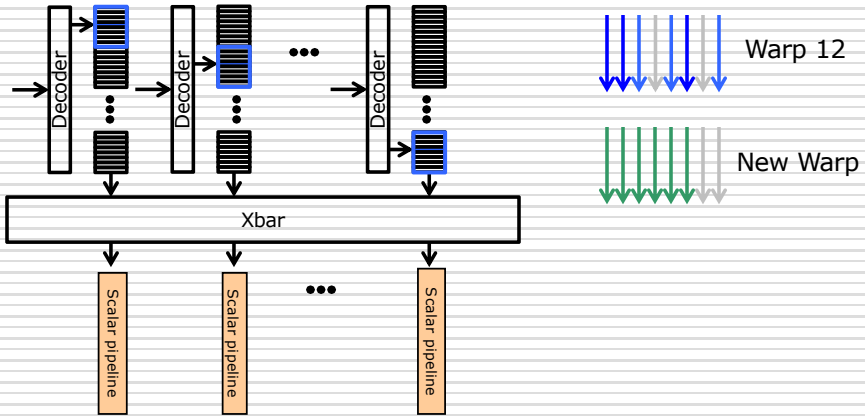
Register File: Warp Formation (2)



- Warps can be reformed but must conform to static lane assignments → want full bandwidth access to lanes
- More on dynamic warp formation later

(29)

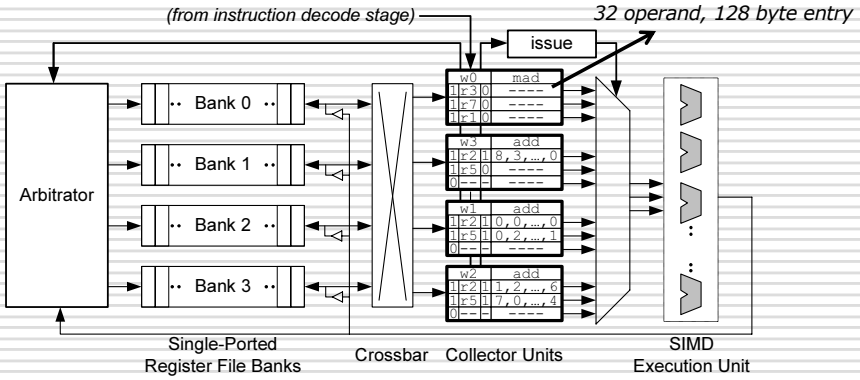
Register File: Warp Compaction



- Can enable sub-warp optimizations such as power gating, intra-chip communication optimizations (TBD), etc.
- Remove the the Xbar with lane-aware compaction

(30)

The Operand Collector

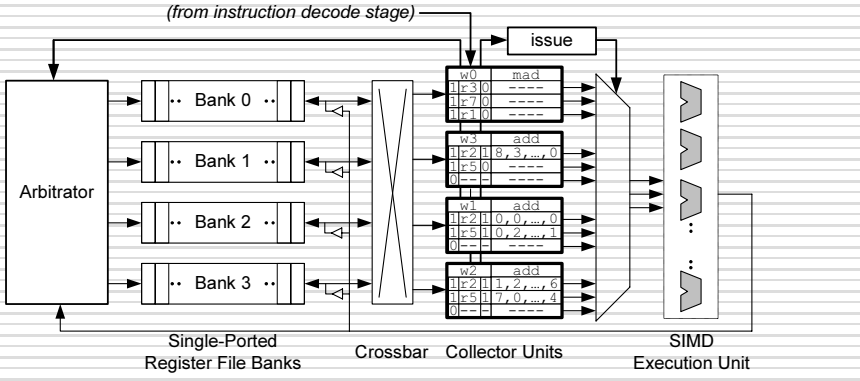


- Register values for a warp may arrive at difference times due to bank conflicts
- Staging of operands in a warp prior to issue

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

(31)

Caching in the Operand Collector



- Operands for each warp are collected over time (bank conflicts)
- Sharing of operands across collectors → cache?

Figure from T. Aamodt, W. Fung, and T. Rodgers, "General Purpose Graphics Processor Architectures," Draft Text

(32)

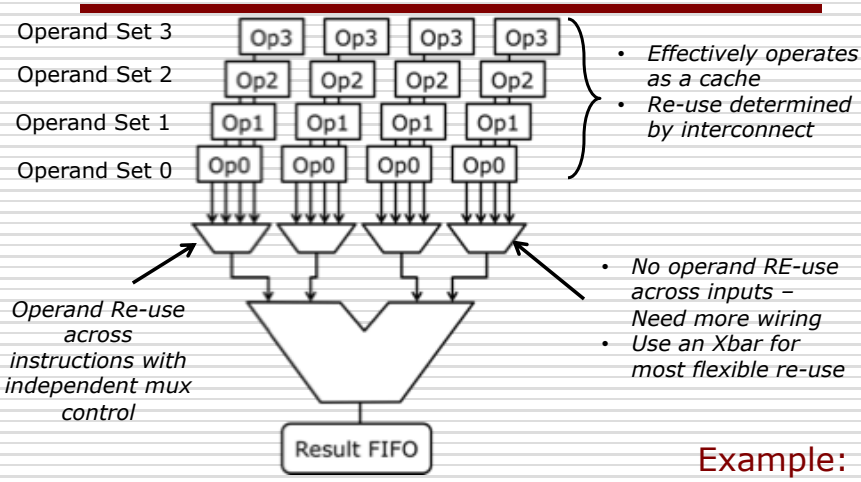
Example

Op	Dest	Op1	Op2	Op3	Op3	Op4	Op5	OP6
FMA	R1	R13	R11	R14	X	X	X	X
Add	R2	R6	R7	R8	X	X	X	X
Mul	R3	R6	R11	R13	X	X	X	X

- Capacity of the OC can reduce register file pressure
  - Store more instructions operands → sharing extends across both instructions, e.g. to registers R11 and R13
  - Coherence? → operands copied into multiple locations
- Reuse
  - Depends on connectivity with lanes
  - Ports to the OC

From B. Coon, et.al, "Tracking Register Usage During Multithreaded Processing Using A Scoreboard Having Separate Memory Regions and Storing Sequential Register Size Indicators," US Patent 7,434,032 B1, October 2008

Collecting Operands

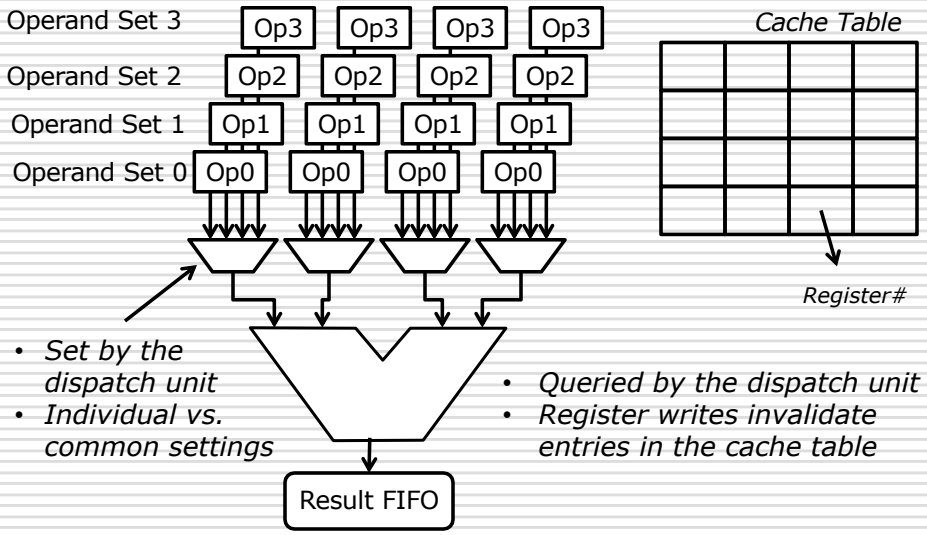


Example:

- From the perspective of a single lane/ALU

From B. Coon, et.al, "Tracking Register Usage During Multithreaded Processing Using A Scoreboard Having Separate Memory Regions and Storing Sequential Register Size Indicators," US Patent 7,434,032 B1, October 2008

Operand Caching

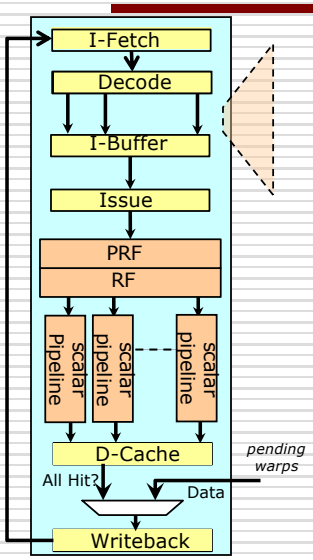


- *Set by the dispatch unit*
- *Individual vs. common settings*

- *Queried by the dispatch unit*
- *Register writes invalidate entries in the cache table*

(35)

Instruction Dispatch



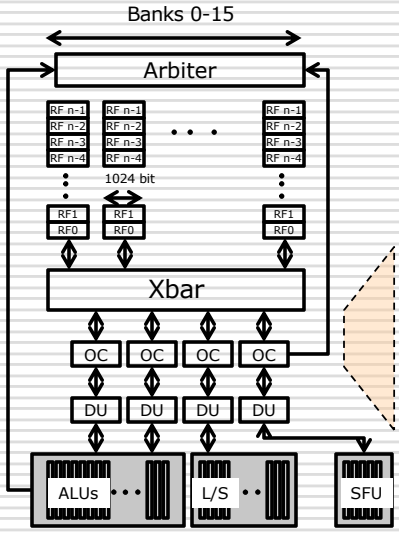
- OC allocated and initialized after decode
- Source operand requests are queued at arbiter

Instruction				
V	Reg	RDY	WID	Operand (128 bytes)
V	Reg	RDY	WID	Operand (128 bytes)
V	Reg	RDY	WID	Operand (128 bytes)

- Operands/cycle →OC limited by interconnect

(36)

Pipeline Stage



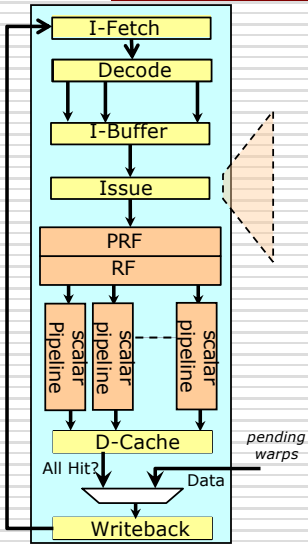
- OC allocated and initialized after decode
- Source operand requests are queued at arbiter

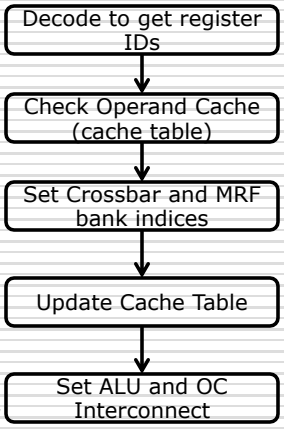
Instruction				
V	Reg	RDY	WID	Operand (128 bytes)
V	Reg	RDY	WID	Operand (128 bytes)
V	Reg	RDY	WID	Operand (128 bytes)

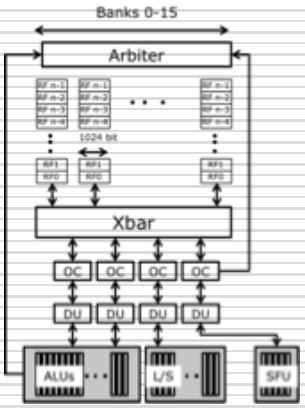
- Operands/cycle →OC limited by interconnect

(37)

Instruction Dispatch

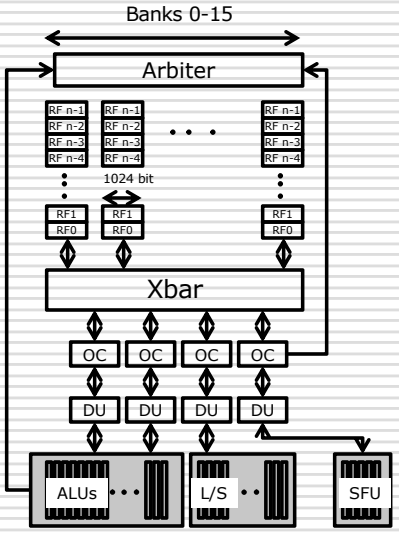






(38)

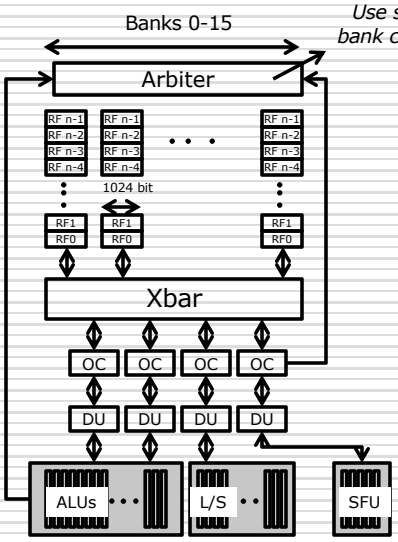
Functional Unit Level Collection



- Operand collectors are associated with different functional unit types
- Can naturally support heterogeneity
- Dedicated vs. shared OC units
  - Connectivity consequences
- Other sources of operands
  - Constant cache, read-only cache

(39)

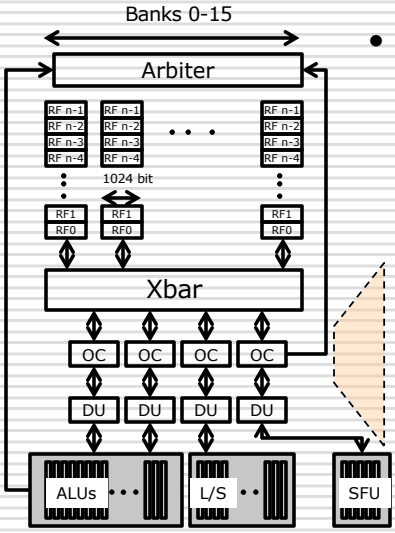
Instruction Perspective



- OC request read operands
- Prioritize writes over reads
- Schedule read requests for maximum BW

(40)

Register File Access: Coherency



- What happens when a new value is written back the Main Register File (MRF)?
  - OC values must be invalidated

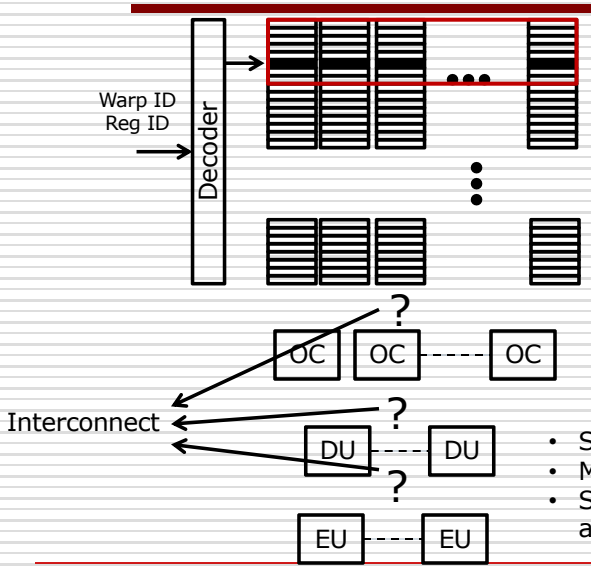
Instruction				
V	Reg	RDY	WID	Operand (128 bytes)
V	Reg	RDY	WID	Operand (128 bytes)
V	Reg	RDY	WID	Operand (128 bytes)

An example OC[1]

[1]http://gpgpu-sim.org/manual/index.php/GPGPU-Sim_3.x_Manual#Register_Access_and_the_Operand_Collector

(41)

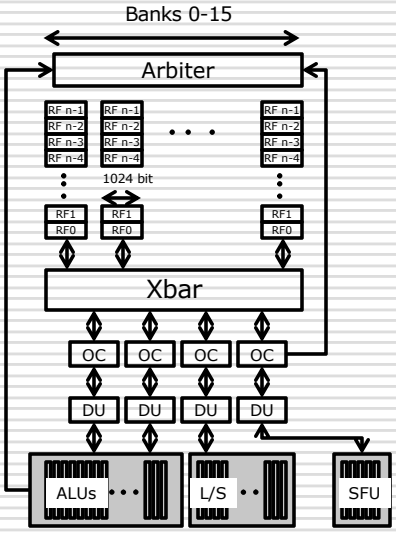
Scheduling and Allocating OCs



- Scheduling of OCs –policy?
- Mitigate bank conflicts
- Support for heterogeneous architecture

(42)

The Operand Collector: Summary



- Buffer (warp size) operands at the collector
- Sharing of operands across instructions
- Operates as an operand cache for "collecting" operands across bank conflicts
- Simplifies scheduling accesses to the MRF

Example Flow:

(43)

Project Ideas

- Goals
  - Reduce MRF Bank Conflicts
  - Allocation policies across Warps
  - Hierarchical Register Files – implement from papers
- Possible Methodologies
  - Get instructions traces from GPGPUSim or Harmonica
  - Implement simple register file analysis
    - What is the degree of actual sharing that is taking place across registers?
    - What do the register dependency chains look like?
    - What is the headroom for reducing bank conflicts?
    - Impact of relationships between #banks and #lanes

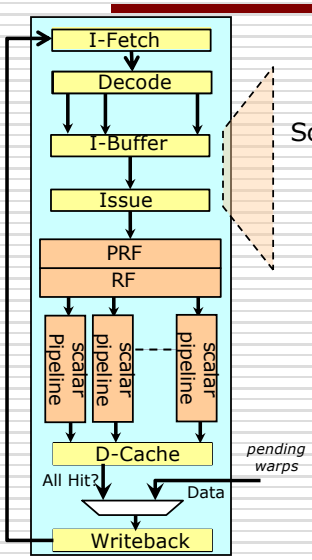
(44)

Summary: Register File

- Register file management and operand dispatch has multiple interacting components
- Performance complexity tradeoff
  - Concurrency increase requires increasing interconnect complexity
  - Stalls/conflicts require buffering and bypass to increase utilization of the execution units
- Good register file allocation in critical

(45)

The Scoreboard



- Scoreboard implemented @issue → register read operand step
- Adaptation to high register count of SIMT/GPUs

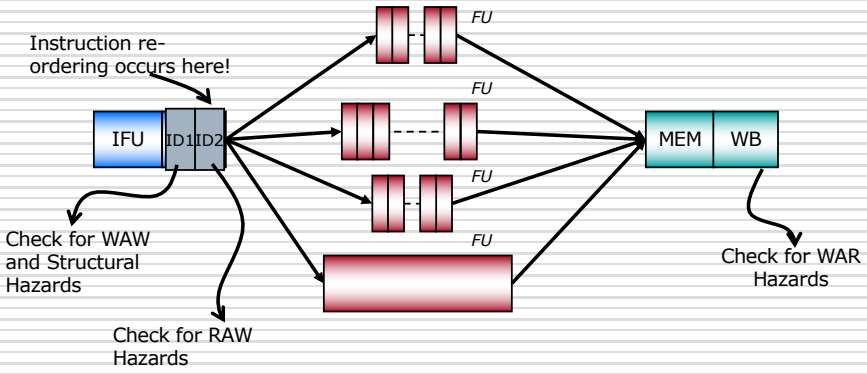
(46)

Review: Generic Scoreboard

- Classic: Comprised of several data structures
  - Instruction status table
  - Function status table
  - Result Table
- What are the challenges in straightforward adaptation of CPU concepts?

(47)

Scoreboard Operation



- Separate read operand stage from issue logic
- Issue logic tracks register dependencies
- Issue logic tracks structural dependencies

(48)

Instruction Status Table

- Keeps the information about which activities of the execution process an instruction is currently in.
  - `rdopd?` - has it completed reading its operands?
  - `issue?` - is the instructions issued?
  - `exec?` - has it completed its execution?
  - `wrback?` - has it completed its writeback?
- Impact on warp level tracking

(49)

Functional Unit Status Table

Function unit producing value | Source Registers have value?

dest reg | src1 | src2

Name	Busy	Op	F_i	F_j	F_k	Q_j	Q_k	R_j	R_k
Integer	Yes	Load	F2	R3				No	
Add	Yes	Sub	F8	F6	F2		Integer	Yes	No

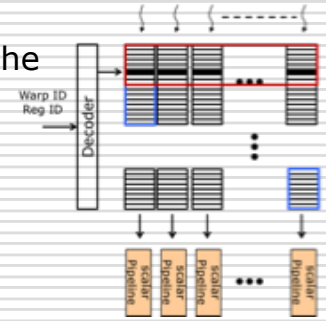
- Has an entry for each functional unit and there are 9 fields for each entry:
  - `busy?` - indicates if the functional unit is busy;
  - `op` - the kind of operation being performed;
  - `dest` - the destination register;
  - `src1, src2` - the two source registers;
  - `Func1` (Q_i), `func2` (Q_j) - the functional units producing the results in the two source registers;
  - `ready1?, ready2?` - indicates if `src1` and `src2` is ready;

(50)

Result Status Table

Unit	F0	F2	F4	F6	F8	F10	F12	etc
FU	Mult1	Integer			Add	Divide		

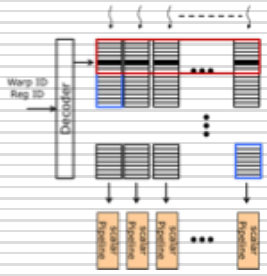
- Maintain an entry per register indicating the functional unit that will write the pending result into the register
- Impact on SIMT/GPU → large number of registers



(51)

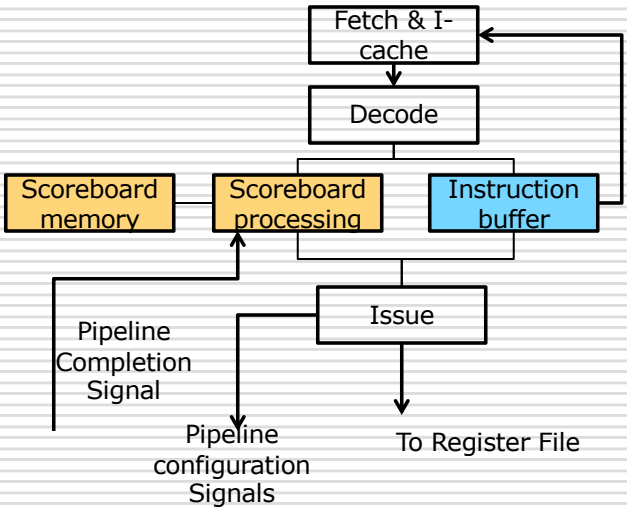
Challenges

- Scoreboard size
  - Tracking register dependencies for all registers → very large number of registers
- Query bandwidth
  - Checking scoreboard structures require high query bandwidth because of the number of registers



(52)

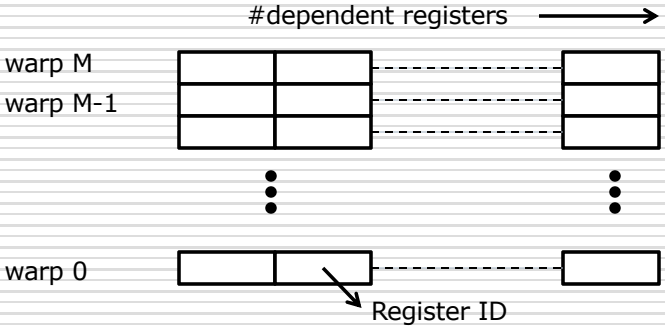
Adaptation to SIMT Processors



From B. Coon, et.al, "Tracking Register Usage During Multithreaded Processing Using A Scoreboard Having Separate Memory Regions and Storing Sequential Register Size Indicators," US Patent 7,434,032 B1, October 2008

(53)

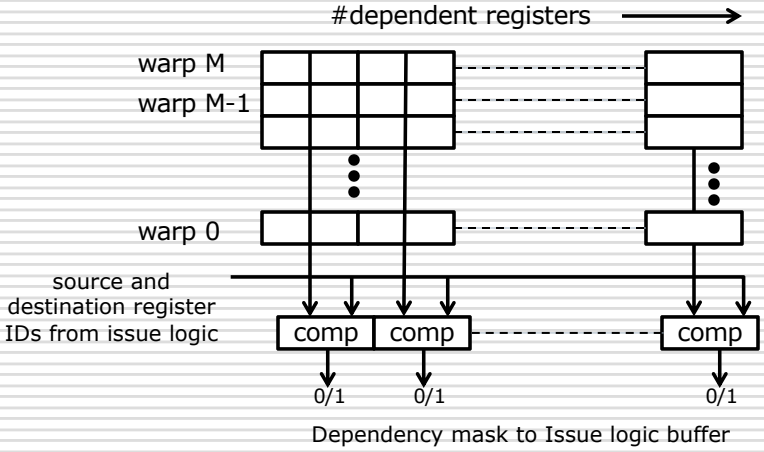
Scoreboard Memory Data Structure



- Each warp tracks number of dependent registers (issued instructions)
- Limit the number of dependencies rather than storage for tracking all possible dependencies

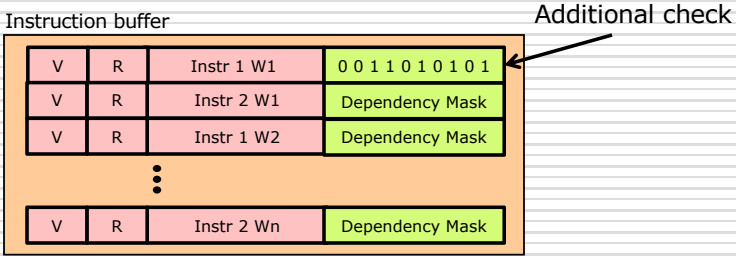
(54)

Scoreboard Memory Data Structure



(55)

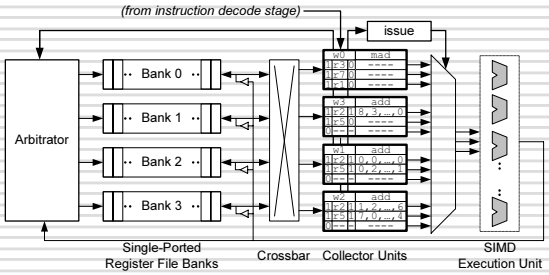
Issue Logic

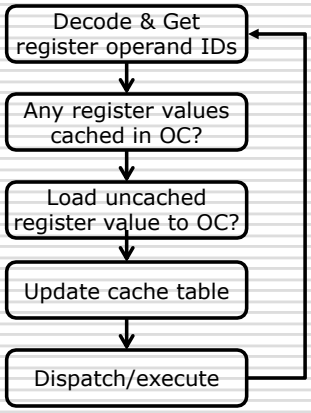


- Updated instruction buffer with the dependency mask for each instruction
- Additional issue constraint: Issue instruction when dependency mask is all 0
  - Can impact parallelism when there are a large number of dependencies

(56)

Operation





- At commit stage, update the OC/Issue/cache data structures for registers that have been written
  - Recompute warp dependency mask
- Update the scoreboard data structures

(57)

Summary

- Buffering at the OC to decouple MRF access/conflicts from dispatch
  - Throughput optimized design vs. latency optimized
- The OC and scoreboard are organized to handle in-order cores but with a large number of registers/threads/units
  - Trading dependencies for concurrency
  - Throughput optimized design

(58)