Slide Set 18: GPUs, cont'd

CPEN411

based on slides from Tor M. Aamodt

NVIDIA Tegra X1 die photo

CUDA Code

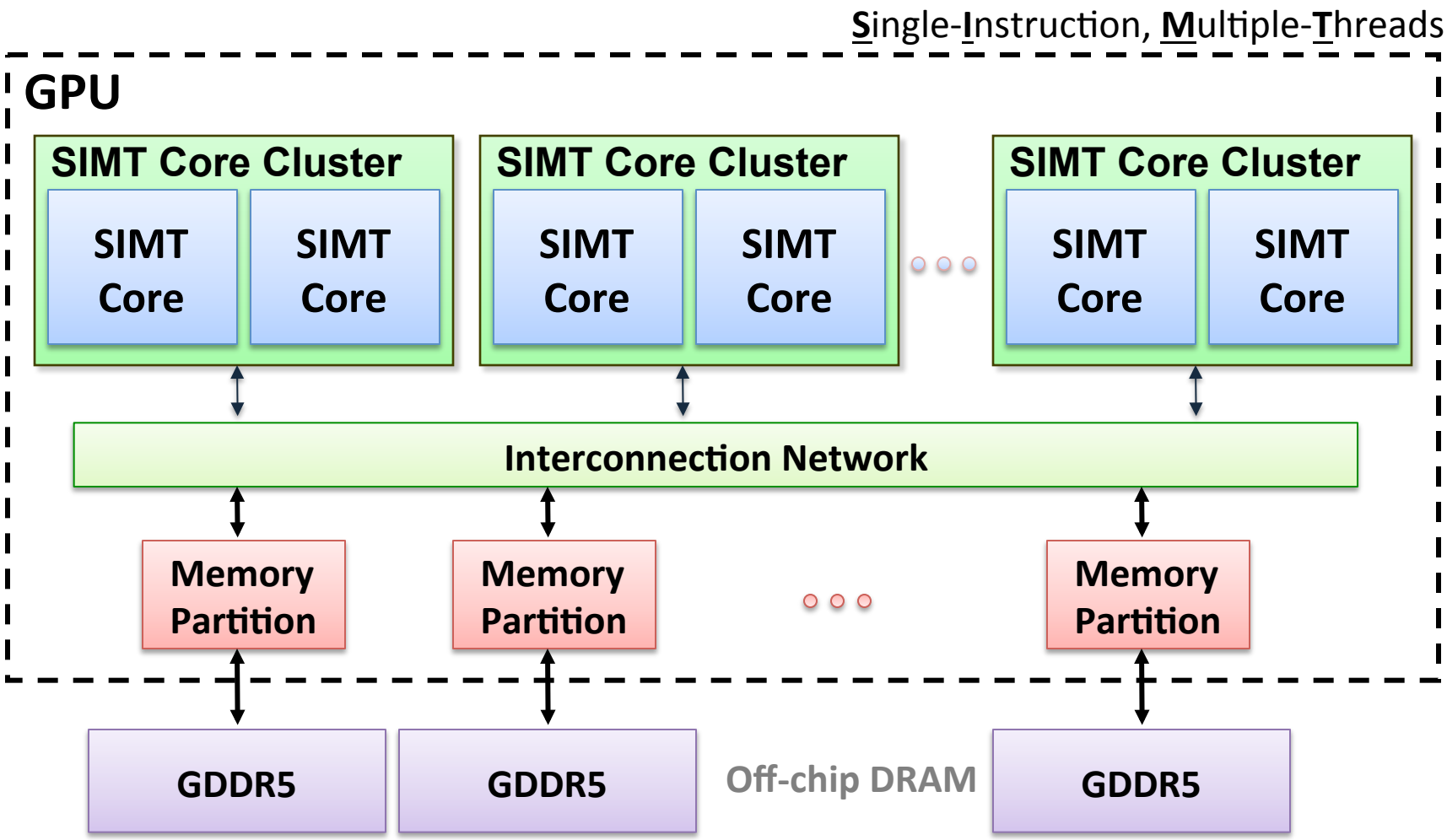
```
__global__ void saxpy(int n, float a, float *x, float *y) {
   int i = blockIdx.x*blockDim.x + threadIdx.x;
   if(i<n) y[i]=a*x[i]+y[i];
}
```

Runs on GPU

```
int main() {
  // omitted: allocate and initialize memory
  int nblocks = (n + 255) / 256;

  cudaMalloc((void**) &d_x, n);
  cudaMalloc((void**) &d_y, n);
  cudaMemcpy(d_x,h_x,n*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_y,h_y,n*sizeof(float),cudaMemcpyHostToDevice);
  saxpy<<<nblocks, 256>>>(n, 2.0, d_x, d_y);
  cudaMemcpy(h_y,d_y,n*sizeof(float),cudaMemcpyDeviceToHost);
  // omitted: using result
}
```

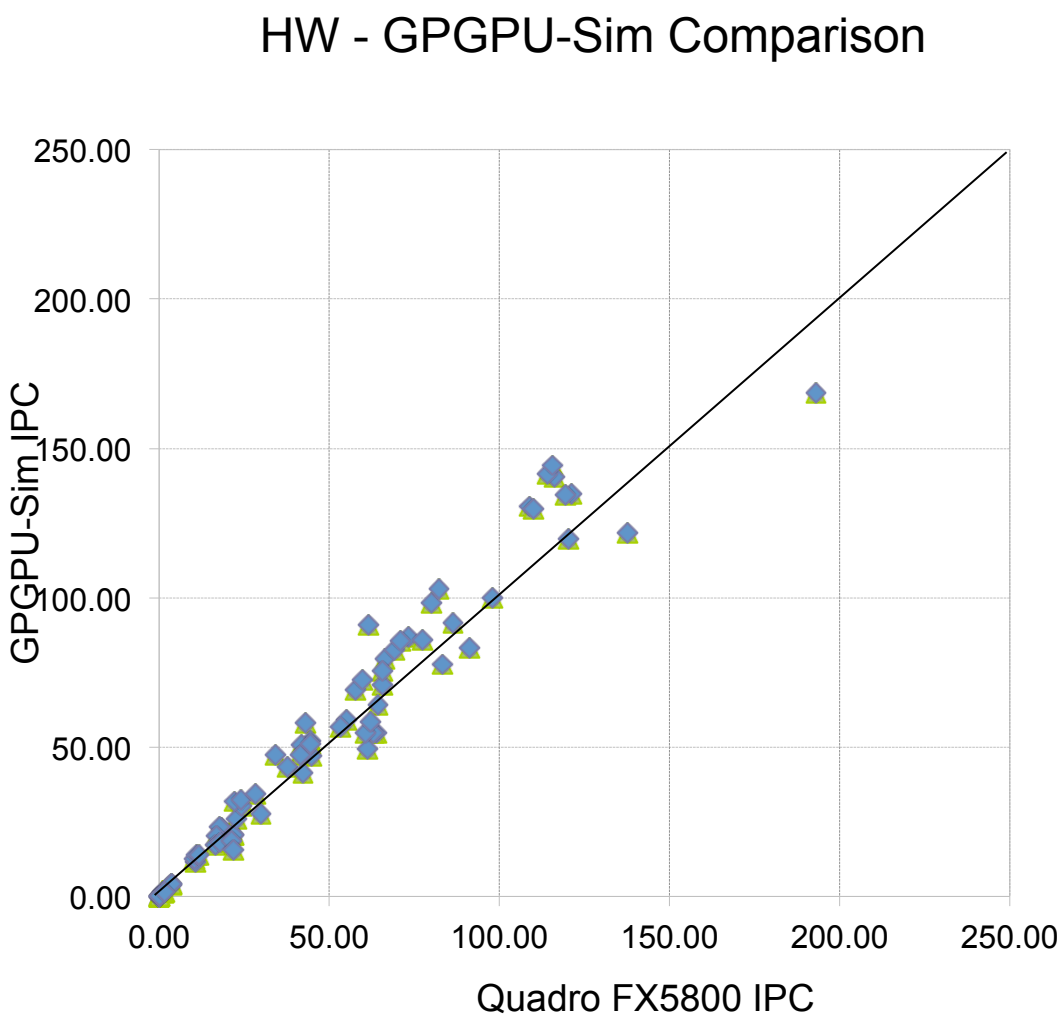
GPU Microarchitecture Overview



GPU Microarchitecture

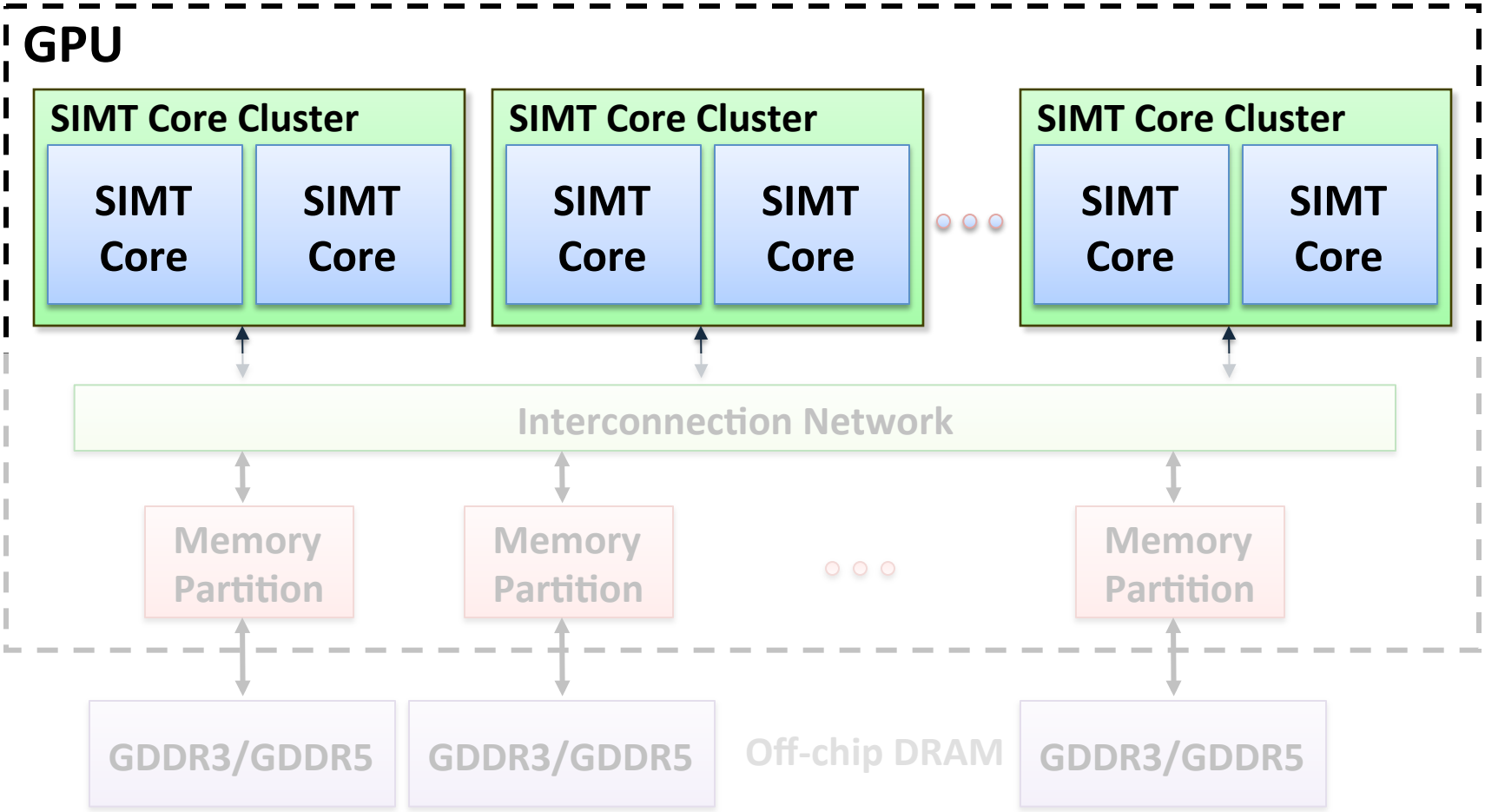
- Companies tight lipped about details of GPU microarchitecture.
- Several reasons:
  - Competitive advantage
  - Fear of being sued by "non-practicing entities"
  - The people that know the details too busy building the next chip
- Model described next, embodied in GPGPU-Sim, developed from: white papers, programming manuals, IEEE Micro articles, patents.

GPGPU-Sim v3.x w/ SASS

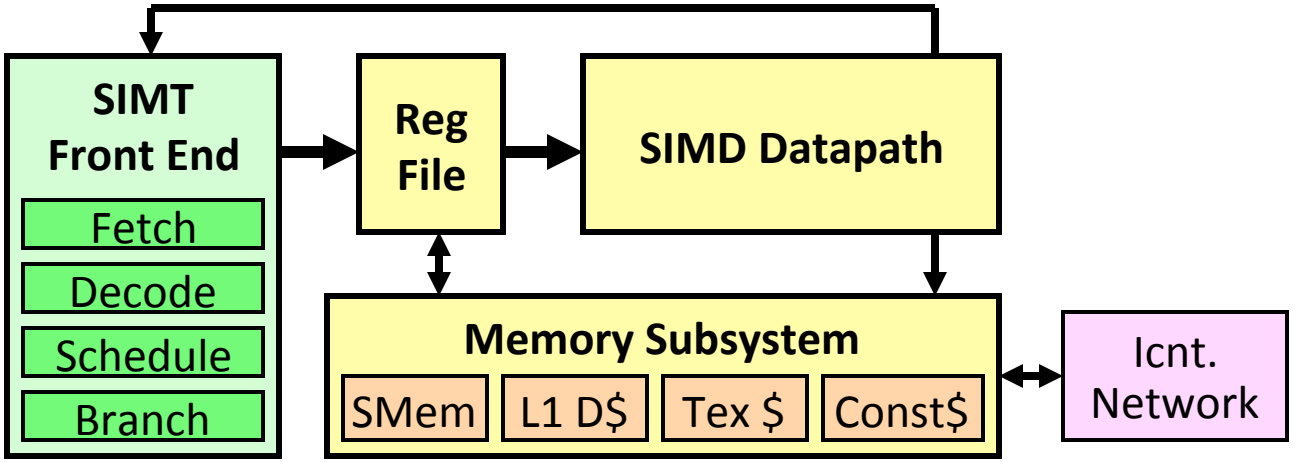


Correlation
~0.976

GPU Microarchitecture Overview

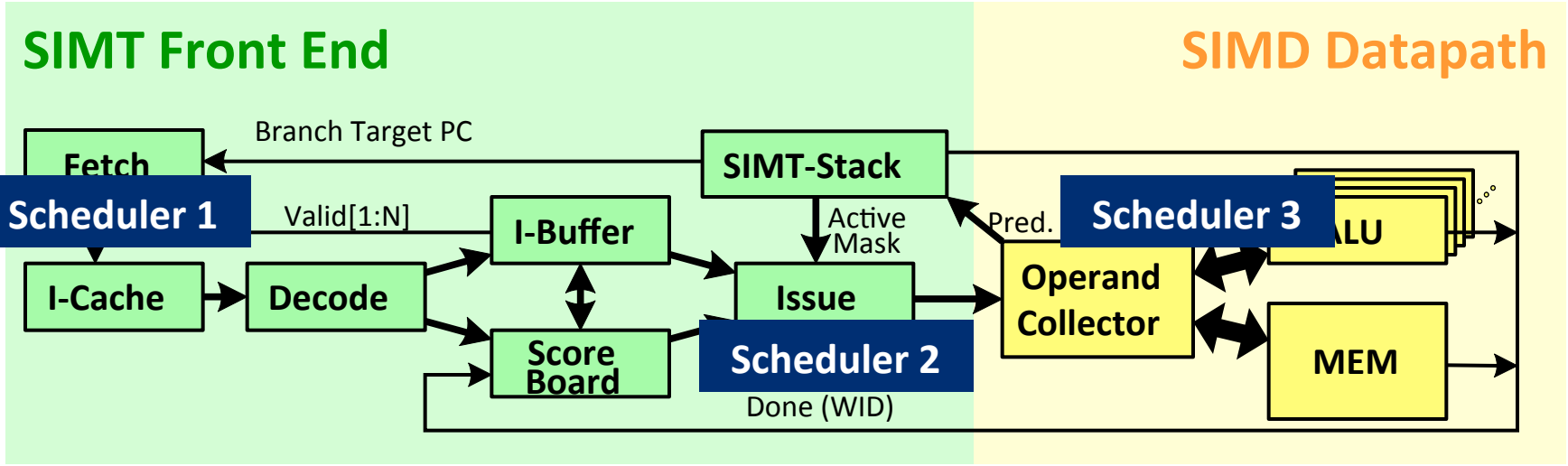


Inside a SIMT Core



- SIMT front end / SIMD backend
- Fine-grained multithreading
  - Interleave warp execution to hide latency
  - Register values of all threads stays in core

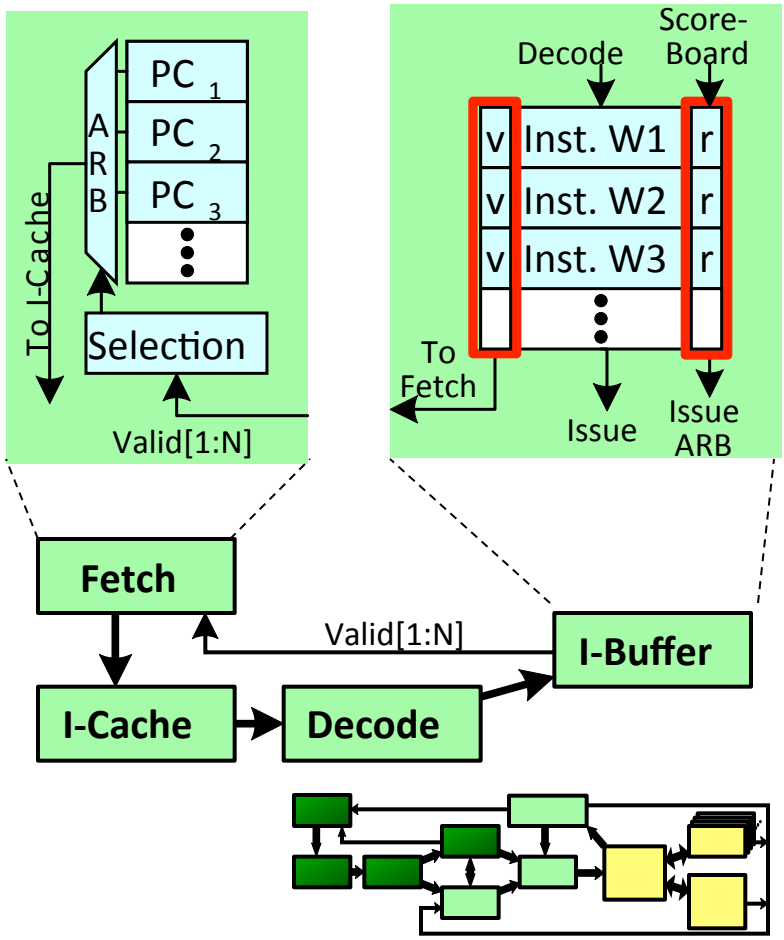
Inside an "NVIDIA-style" SIMT Core



- Three decoupled warp schedulers
- Scoreboard
- Large register file
- Multiple SIMD functional units

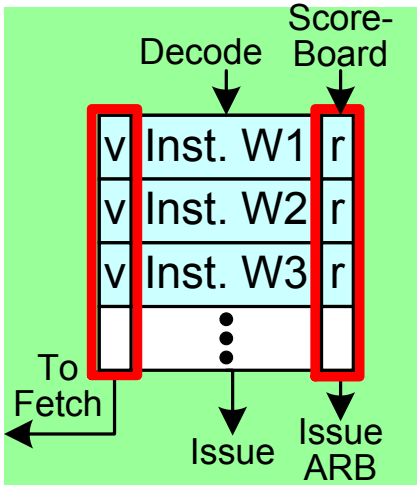
Fetch + Decode

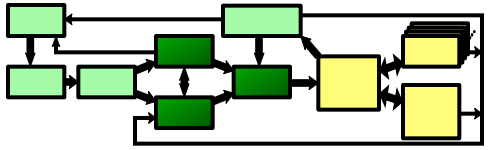
- Arbitrate the I-cache among warps
  - Cache miss handled by fetching again later
- Fetched instruction is decoded and then stored in the I-Buffer
  - 1 or more entries / warp
  - Only warp with vacant entries are considered in fetch



Instruction Issue

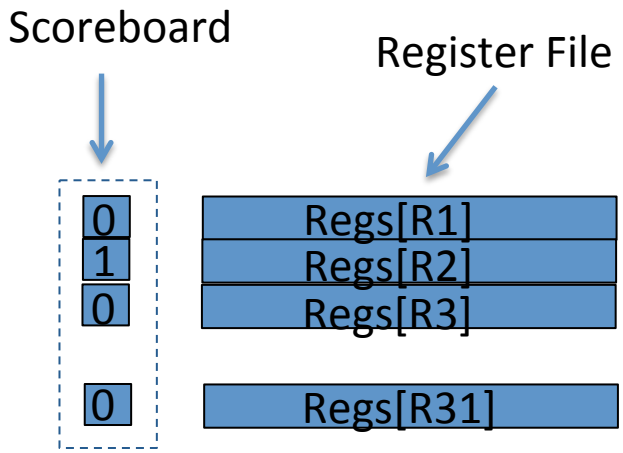
- Select a warp and issue an instruction from its I-Buffer for execution
  - Scheduling: Greedy-Then-Oldest (GTO)
  - GT200/later Fermi/Kepler: Allow dual issue (superscalar)
  - Fermi: Odd/Even scheduler
  - To avoid stalling pipeline might keep instruction in I-buffer until know it can complete (replay)





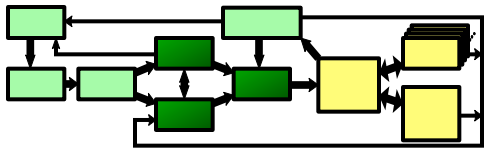
Review: In-order Scoreboard

- Scoreboard: a bit-array, 1-bit for each register
  - If the bit is *not* set: the register has valid data
  - If the bit is set: the register has stale data
    i.e., some outstanding instruction is going to change it
- Issue in-order: RD ← Fn (RS, RT)
  - If SB[RS] or SB[RT] is set → RAW, stall
  - If SB[RD] is set → WAW, stall
  - Else, dispatch to FU (Fn) and set SB[RD]
- Complete out-of-order
  - Update GPR[RD], clear SB[RD]



In-Order Scoreboard for GPUs?

- Problem 1: 32 warps, each with up to 128 (vector) registers per warp means scoreboard is 4096 bits.
- Problem 2: Warps waiting in I-buffer needs to have dependency updated every cycle.
- Solution?
  - Flag instructions with hazards as *not ready* in I-Buffer so not considered by scheduler
  - Track up to 6 registers per warp (out of 128)
  - I-buffer 6-entry bitvector: 1b per register dependency
  - Lookup source operands, set bitvector in I-buffer. As results written per warp, clear corresponding bit



Example

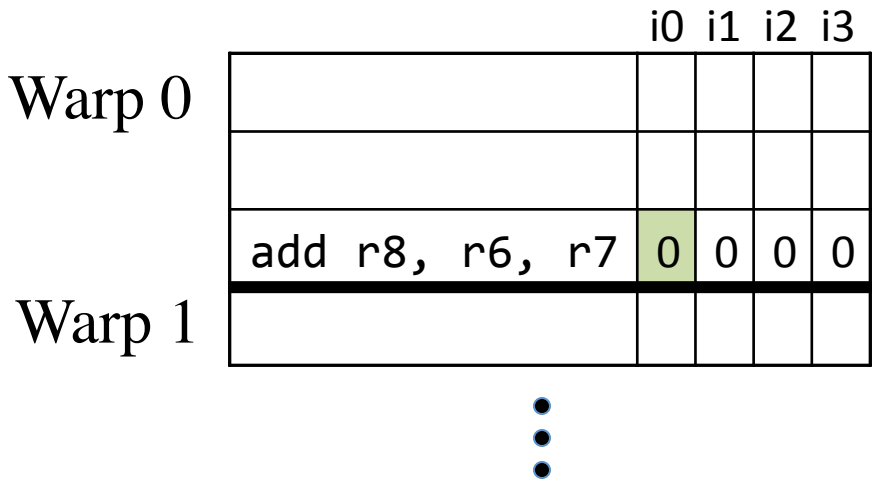
Code

```
ld  r7, [r0]
mul r6, r2, r5
add r8, r6, r7
```

Scoreboard

	Index 0	Index 1	Index 2	Index 3
Warp 0	-	-	r8	-
Warp 1	-	-	-	-

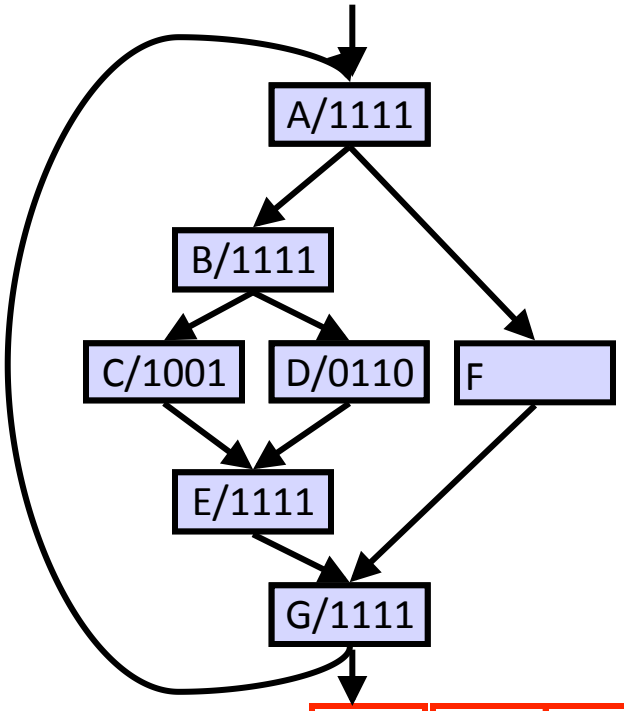
Instruction Buffer



SIMT Using a Hardware Stack

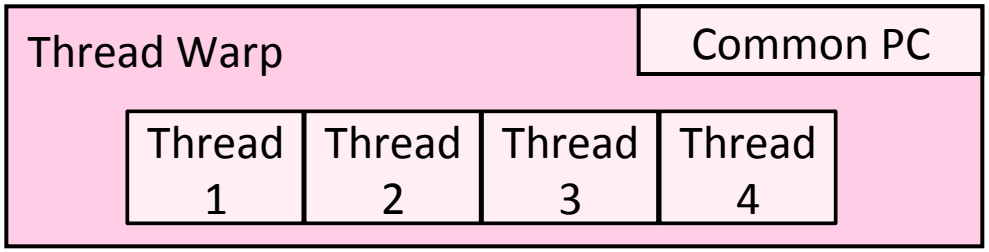
Stack approach invented at Lucafilm, Ltd in early 1980's

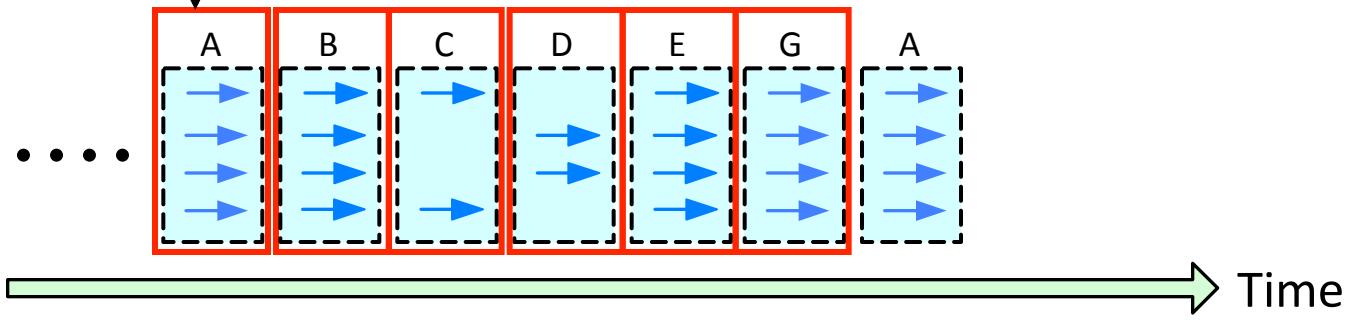
Version here from [Fung et al., MICRO 2007]



Stack

	Reconv. PC	Next PC	Active Mask
TOS →	-	E	1111
TOS →	E	D	0110
TOS →	E	E	1001





SIMT = SIMD Execution of Scalar Threads

SIMT Notes

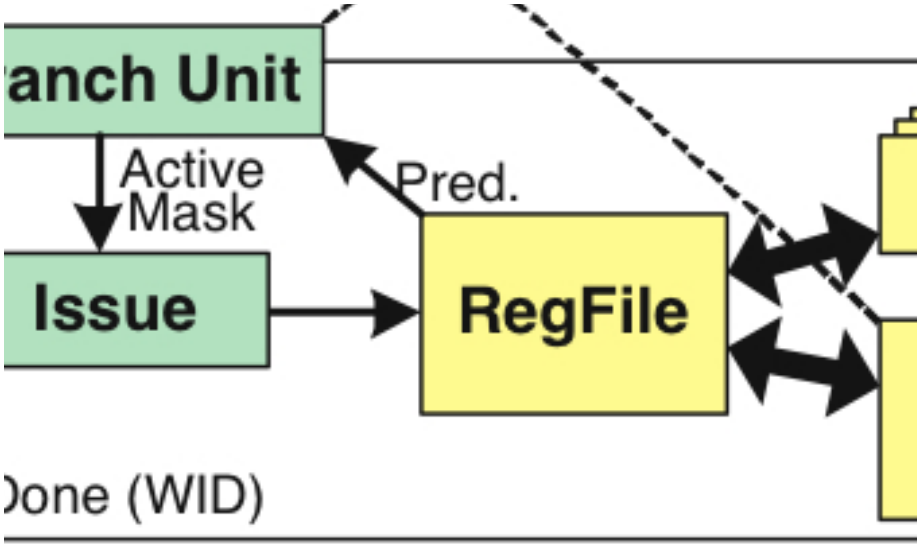
- execution mask stack implemented with special instructions to push/pop
- in practice augment stack with predication (lower overhead)
- leads to interesting corner cases (e.g., deadlock)

SIMT outside of GPUs?

- ARM Research looking at SIMT-ized ARM ISA.
- Intel MIC implements SIMT on top of vector hardware via compiler (ISPC)
- Possibly other industry players in future

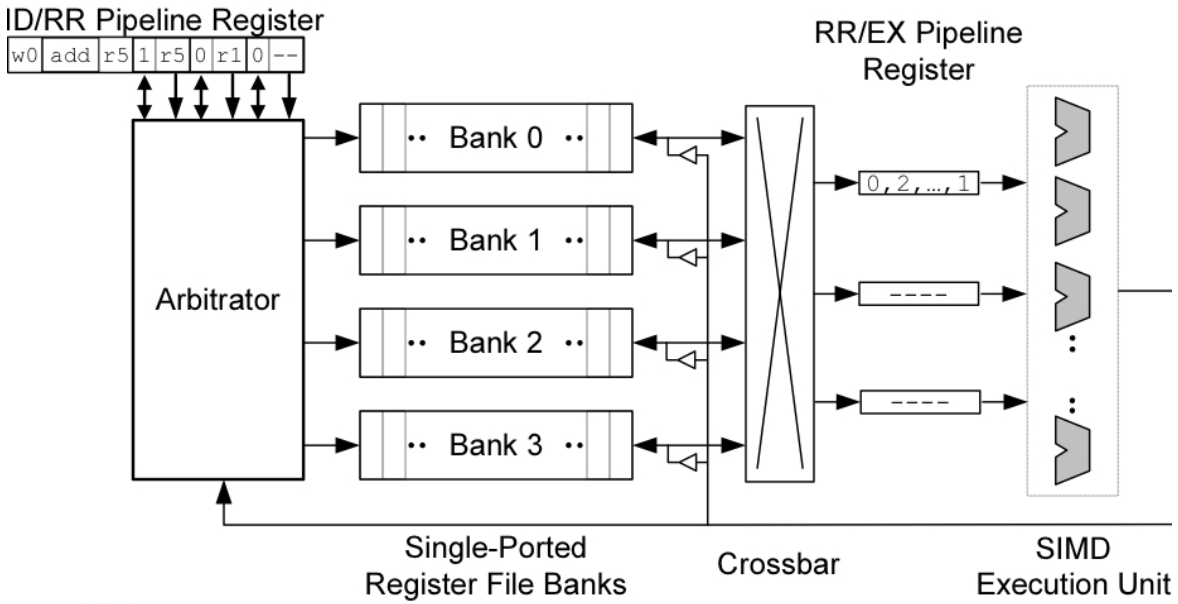
Register File

- 32 warps, 32 threads per warp, 16 x 32-bit registers per thread = 64KB register file.
- Need “4 ports” (e.g., FMA) greatly increase area.
- Alternative: banked single ported register file. How to avoid bank conflicts?



Banked Register File

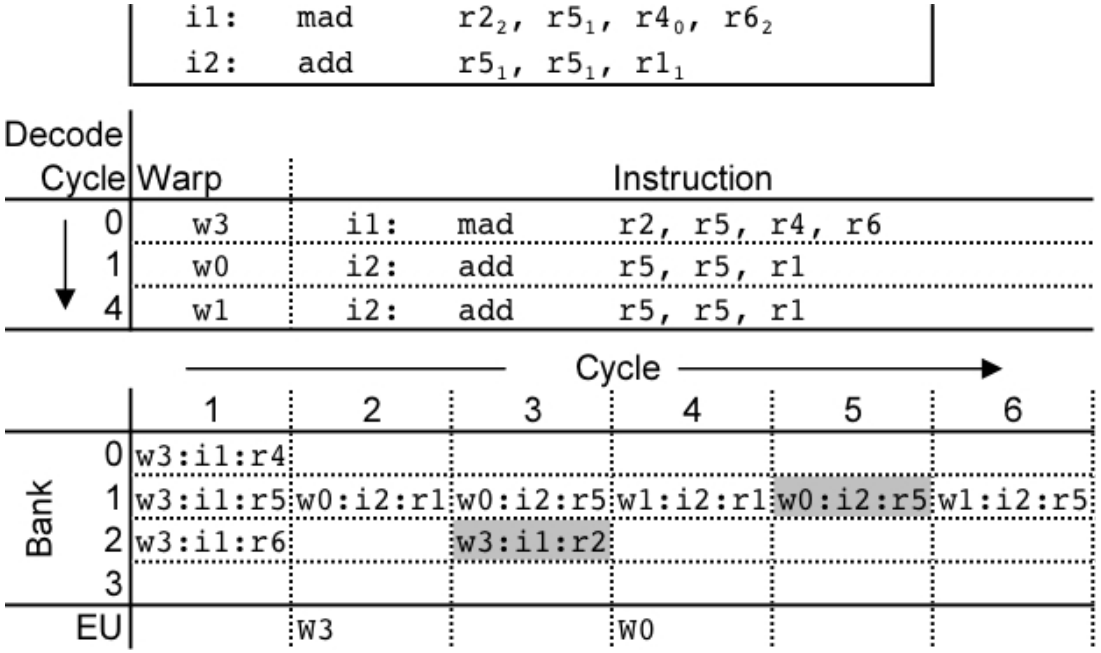
Strawman microarchitecture:



Register layout:

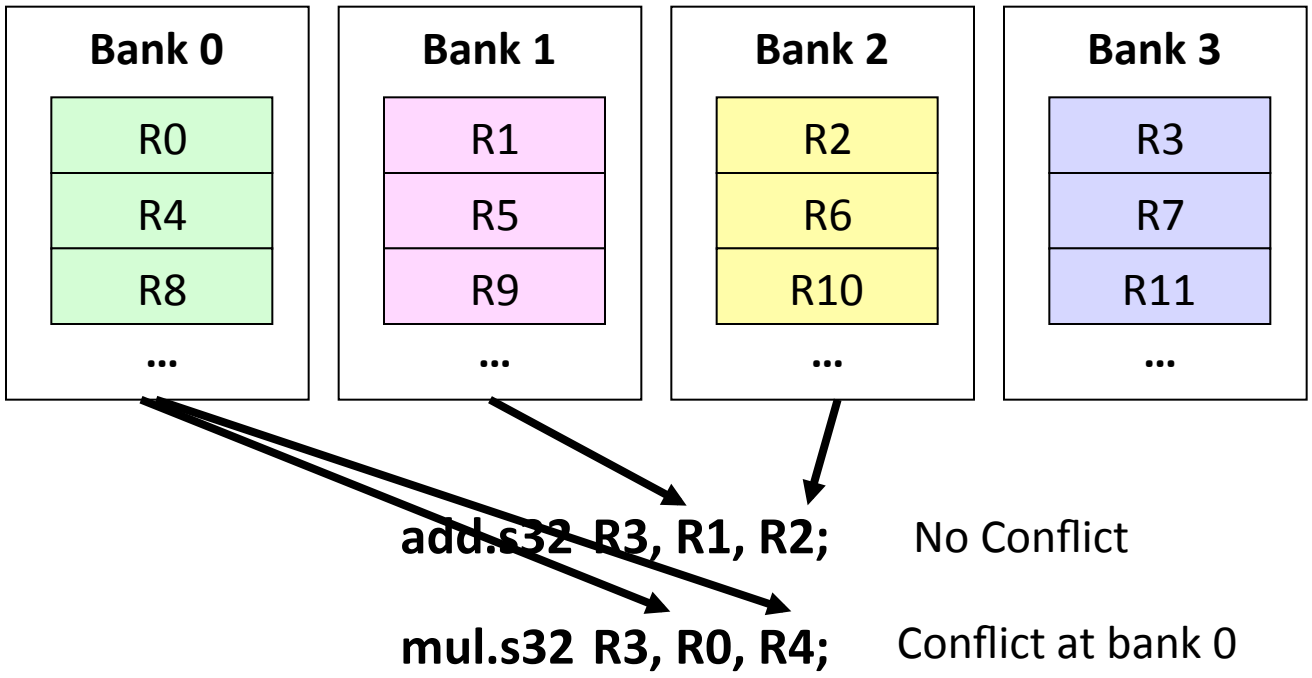
Bank 0	Bank 1	Bank 2	Bank 3
…	…	…	…
w1:r4	w1:r5	w1:r6	w1:r7
w1:r0	w1:r1	w1:r2	w1:r3
w0:r4	w0:r5	w0:r6	w0:r7
w0:r0	w0:r1	w0:r2	w0:r3

Register Bank Conflicts

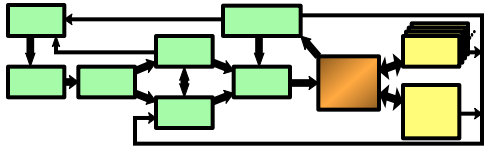


- warp 0, instruction 2 has two source operands in bank 1: takes two cycles to read.
- Also, warp 1 instruction 2 is same and is also stalled.
- Can use warp ID as part of register layout to help.

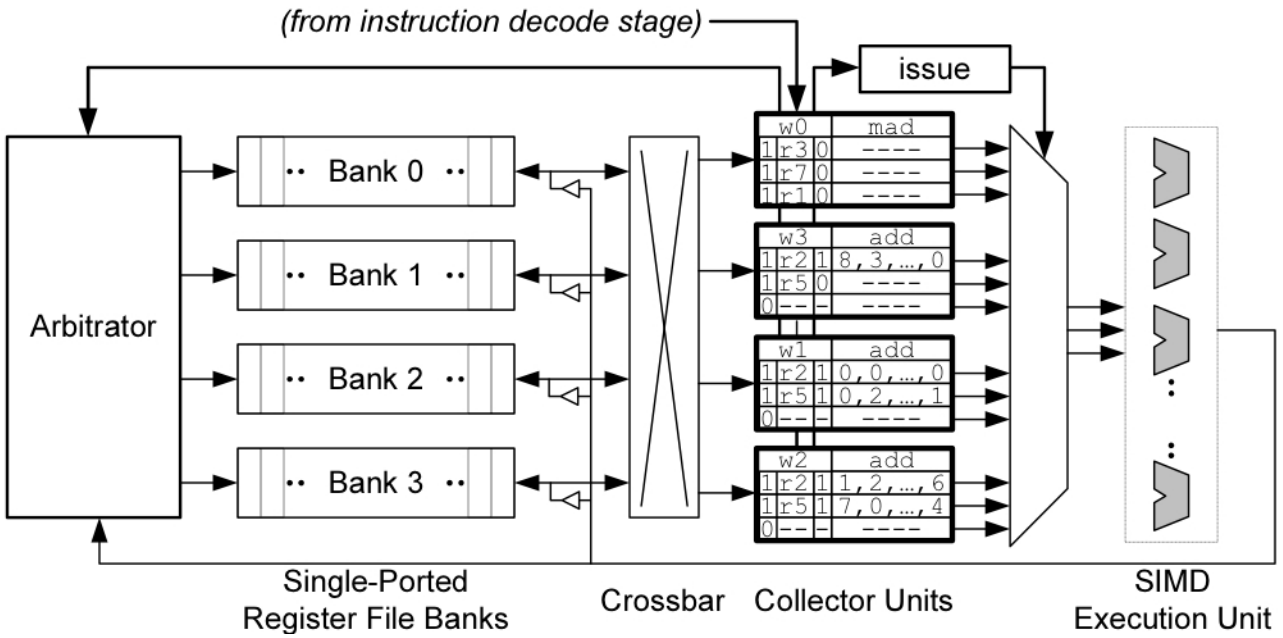
Operand Collector



- Term "Operand Collector" appears in figure in NVIDIA Fermi Whitepaper
- Operand Collector Architecture (US Patent: 7834881)
  - Interleave operand fetch from different threads to achieve full utilization



Operand Collector (1)



- Issue instruction to collector unit.
- Collector unit similar to reservation station in tomasulo's algorithm.
- Stores source register identifiers.
- Arbiter selects operand accesses that do not conflict on a given cycle.
- Arbiter needs to also consider writeback (or need read+write port)

Operand Collector (2)

- Combining swizzling and access scheduling can give up to ~ 2x improvement in throughput

```
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	

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

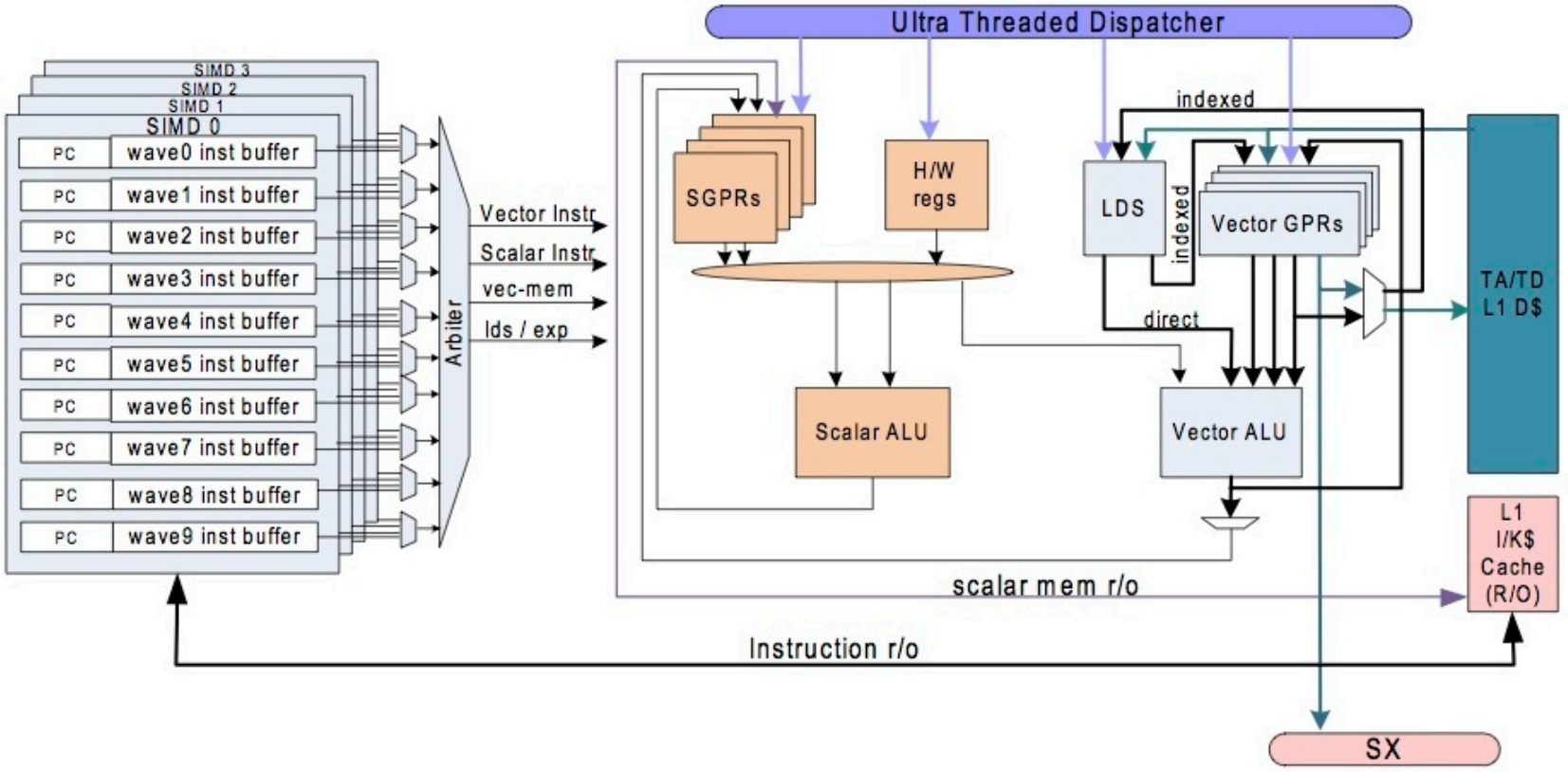
AMD Southern Islands

- SIMT processing often includes redundant computation across threads.

```
thread 0…31:
for(i=0; i < runtime_constant_N; i++) {
    /* do something with “i” */
}
```

AMD Southern Islands SIMT-Core

ISA visible scalar unit executes computation identical across SIMT threads in a wavefront



Example

```
float fn0(float a,float b)
{
   if(a>b)
      return (a * a – b);
   else
      return (b * b – a);
}
```

```
// Registers r0 contains “a”, r1 contains “b”
// Value is returned in r2
    v_cmp_gt_f32 r0, r1 // a>b
    s_mov_b64 s0, exec  // save current exec mask
    s_and_b64 exec, vcc, exec // do “if”
    s_cbranch_vccz label0 // early exit if all lanes fail
    v_mul_f32 r2, r0, r0 // result = a * a
    v_sub_f32 r2, r2, r1 // result = result - b
label0:
    s_not_b64 exec, exec // do “else”
    s_and_b64 exec, s0, exec // do “else”
    s_cbranch_execz label1 // early exit if all lanes fail
    v_mul_f32 r2, r1, r1 // result = b * b
    v_sub_f32 r2, r2, r0 // result = result - a
label1:
    s_mov_b64 exec, s0   // restore exec mask
```

[Southern Islands Series Instruction Set Architecture, Aug. 2012]

Southern Islands SIMT Stack?

- Instructions: `S_CBRANCH_*_FORK; S_CBRANCH_JOIN`
- Use for arbitrary (e.g., irreducible) control flow
- 3-bit control stack pointer
- Six 128-bit stack entries; stored in scalar general purpose registers holding {exec[63:0], PC[47:2]}
- `S_CBRANCH_*_FORK` executes path with fewer active threads first

Active area of research!

- warp scheduling?
  - GTO common, can we do better?
- memory access scheduling?
  - better than coalescing?
- branch divergence reduction?
  - also, proper semantics?
- coherent shared memory?
  - intra-GPU? CPU-GPU?
- cache replacement / prefetching?
  - lots of ways to control locality on a GPU
- virtual addressing?
  - effective TLB miss, fault strategies?