# GPU Compilation:

## From Shader Code to Machine Instructions via LLVM AMDGPU Backend

Mirko Brkušanin, AMD

AMD

together we advance_

# Compiler Construction, 2026

Faculty of Mathematics, University of Belgrade

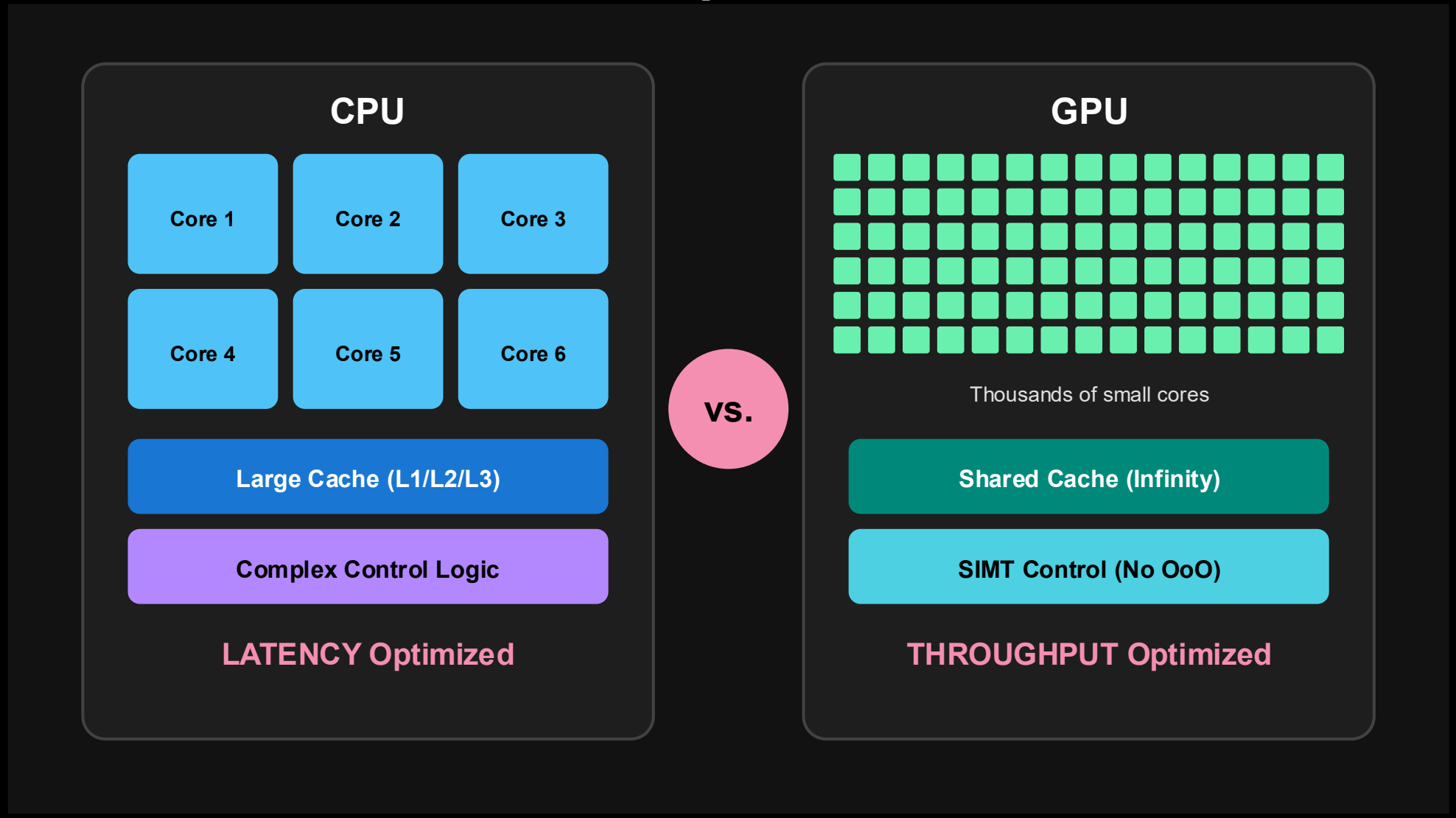
# Topics

- 1. Why is GPU compilation different from CPU?
  - SIMT model, wavefronts, divergence
- 2. GPU Architecture - What the compiler must know
  - Registers (VGPR/SGPR), memory, ALU
- 3. Compilation pipeline
  - GLSL/HLSL → SPIR-V → LLVM IR → AMDGPU ISA
- 4. Key LLVM passes for GPU
  - Structurization, lowering, instruction selection
- 5. Optimizations and challenges
  - Occupancy, register pressure, divergence



AMD
together we advance_

# CPU vs. GPU Architecture - Simplified overview



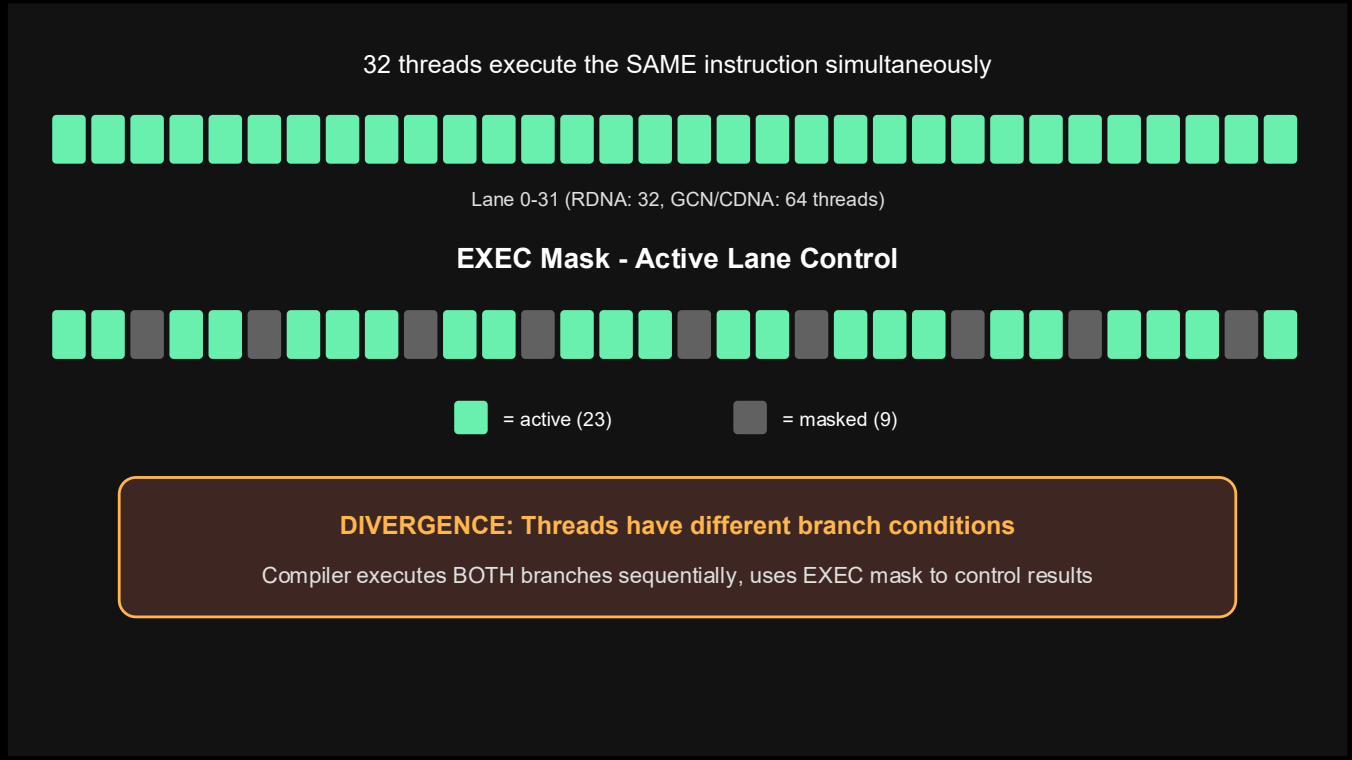



AMD
together we advance_

# SIMT Execution Model

- What is a “wavefront”?
  - Group of threads executing the SAME instruction
  - 32 threads (RDNA™) or 64 threads (GCN™/CDNA™)
  - ALL threads in wavefront execute the same instruction

- SIMT = Single Instruction, Multiple Threads
- Unlike CPU SIMD (SSE/AVX), the programmer writes scalar code
- Hardware executes it on 32/64 threads at once




Cycle 1: v_add_f32 v0, v1, v2 → all 32 threads do addition
Cycle 2: v_mul_f32 v3, v0, v4 → all 32 threads do multiplication

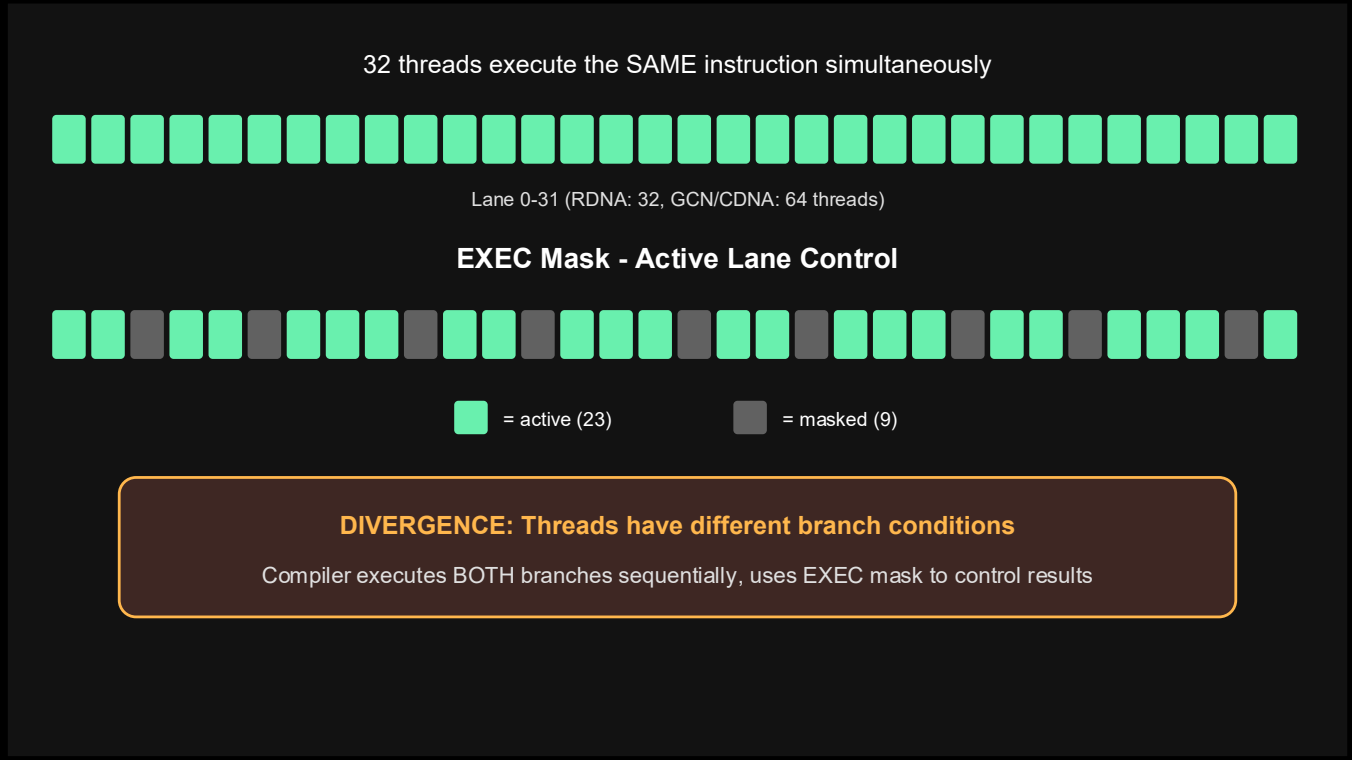


AMD
together we advance_

# Divergence - GPU Compiler's Biggest Problem

```
if (x > 0) {
    a = sqrt(x);
} else {
    a = 0;
}
```

- Problem: Some threads have x>0, others don't
  - Thread 0: x = 5.0 → wants THEN
  - Thread 1: x = -2.0 → wants ELSE
  - Thread 2: x = 3.0 → wants THEN
  - Thread 3: x = -1.0 → wants ELSE
  - …
- Solution: Execute BOTH branches, mask results
- EXEC MASK = hardware bitmask that tells which threads execute




Step 1: All threads: evaluate x > 0, create EXEC mask
Step 2: THEN branch: only threads where x > 0 compute
Step 3: ELSE branch: only threads where x <= 0 compute
Step 4: Both merge - all threads continue together



AMD
together we advance_

# Registers - VGPR and SGPR

## VGPR (Vector)

v0 =  ...

32 values (one per lane)

**Different for each thread!**

**Used for**

- Pixel coordinates
- Per-thread data
- Intermediate results
- Interpolated attributes

**256 VGPR per wavefront**

⚠️ More VGPR = lower occupancy!

64 VGPR = 8KB per wavefront

## SGPR (Scalar)

s0 = **single value**

One value for all lanes

**Same for ALL threads!**

**Used for**

- Constants
- Buffer pointers
- Uniform values
- Control flow

**108 SGPR available**

✓ Use SGPR when possible!

Much more efficient



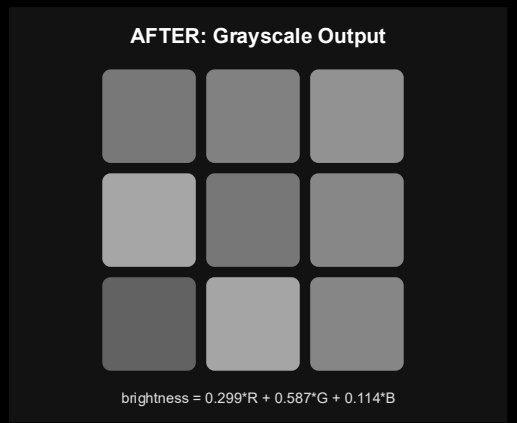
AMD
together we advance_

# Example

```
#version 450
layout(location=0) in vec3 fragColor;
layout(location=0) out vec4 outColor;

void main() {
  float brightness = dot(fragColor,
                vec3(0.299, 0.587, 0.114));
  outColor = vec4(vec3(brightness), 1.0);
}
```

```
s_mov_b32 s0, 0x3e991687    ; 0.299f
s_mov_b32 s1, 0x3f1645a2    ; 0.587f
s_mov_b32 s2, 0x3de978d5    ; 0.114f
v_mul_f32 v3, s0, v0        ; r * 0.299
v_mul_f32 v4, s1, v1        ; g * 0.587
v_mul_f32 v5, s2, v2        ; b * 0.114
v_add_f32 v3, v3, v4
v_add_f32 v3, v3, v5   ; brightness
v_mov_b32 v0, v3       ; R = G = B = brightness
v_mov_b32 v1, v3
v_mov_b32 v2, v3
v_mov_b32 v3, 1.0         ; alpha = 1.0
exp mrt0 v0, v1, v2, v3 ; export to render target
```



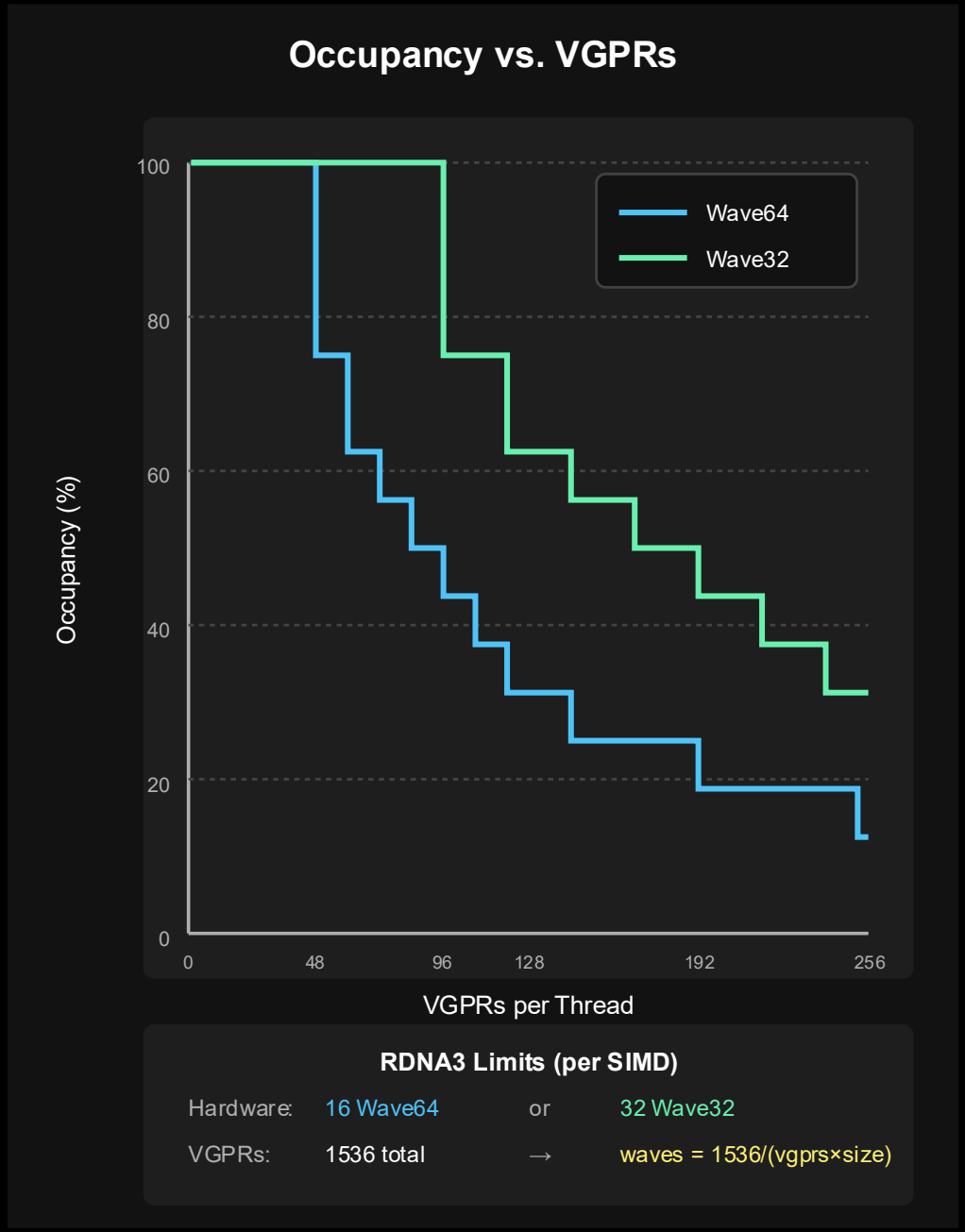

AMD
together we advance_

# Occupancy and Register Pressure

- OCCUPANCY = $\frac{\text{active wavefronts}}{\text{maximum wavefronts}}$
- Why it matters
  - GPU hides latency by switching between wavefronts
  - More wavefronts = better latency hiding
  - Fewer wavefronts = GPU stalls waiting for memory

- What limits occupancy
  - VGPR usage (most common limiter)
  - SGPR usage
  - LDS usage
  - Workgroup size

- Compiler can use `-amdgpu-num-vgpr` to limit VGPR and increase occupancy






AMD
together we advance_

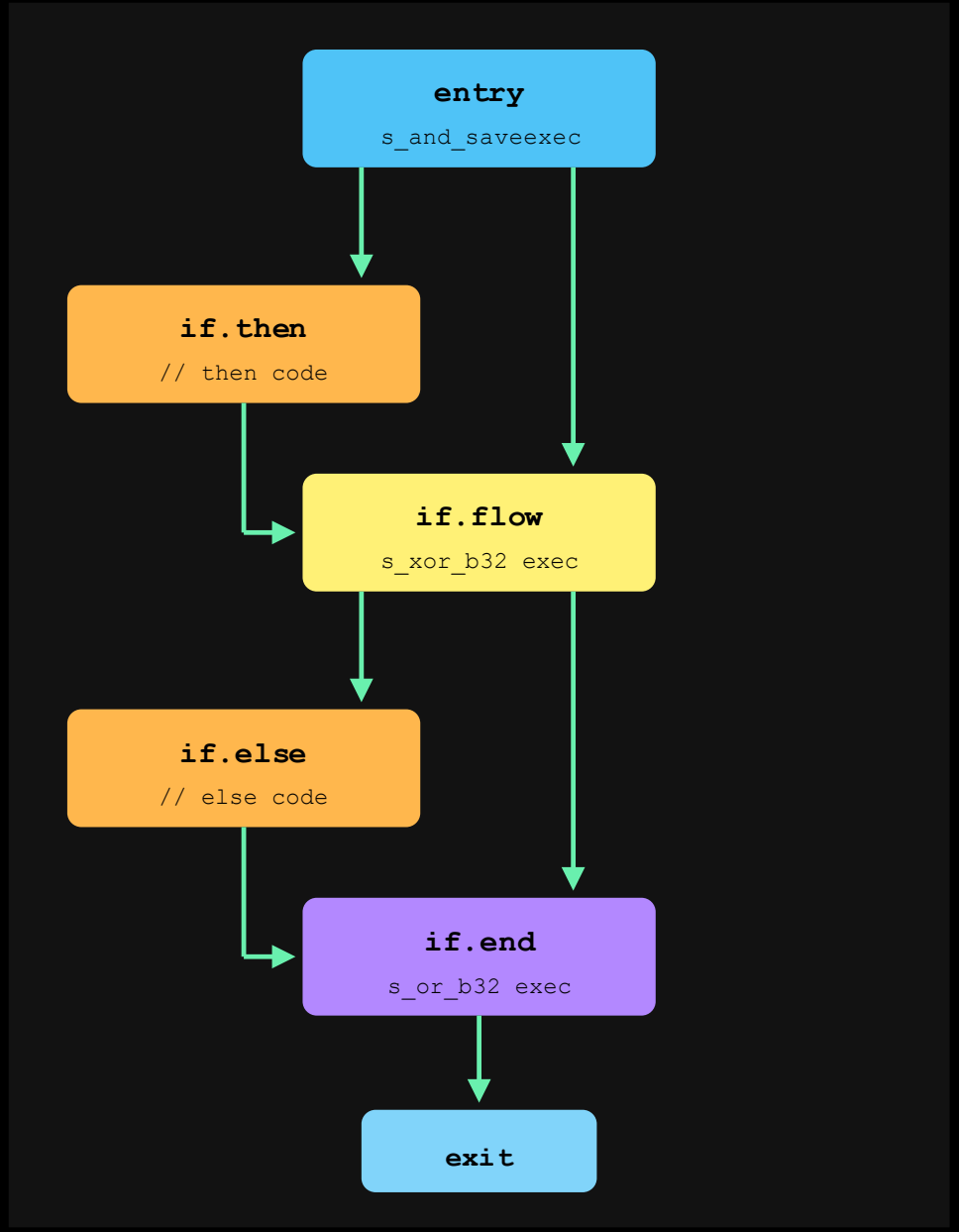
# CFG Structurization

- GPU ISA does NOT require structured control flow
- AMD ISA has `s_branch`, `s_cbranch_*` for arbitrary jumps

- Why structurize anyway?
  - LLVM's SelectionDAG/GlobalISel work better with structured CFG
  - Easier to generate correct EXEC mask save/restore code
  - Simplifies register allocation and instruction scheduling
  - Avoids irreducible CFG (loops with multiple entry points)
    - Makes analysis and optimization much harder

- Solution: StructurizeCFG pass
  - Transforms arbitrary CFG to single-entry-single-exit regions
  - Inserts "flow" variables to track which path was taken
  - Ensures all divergent paths reconverge

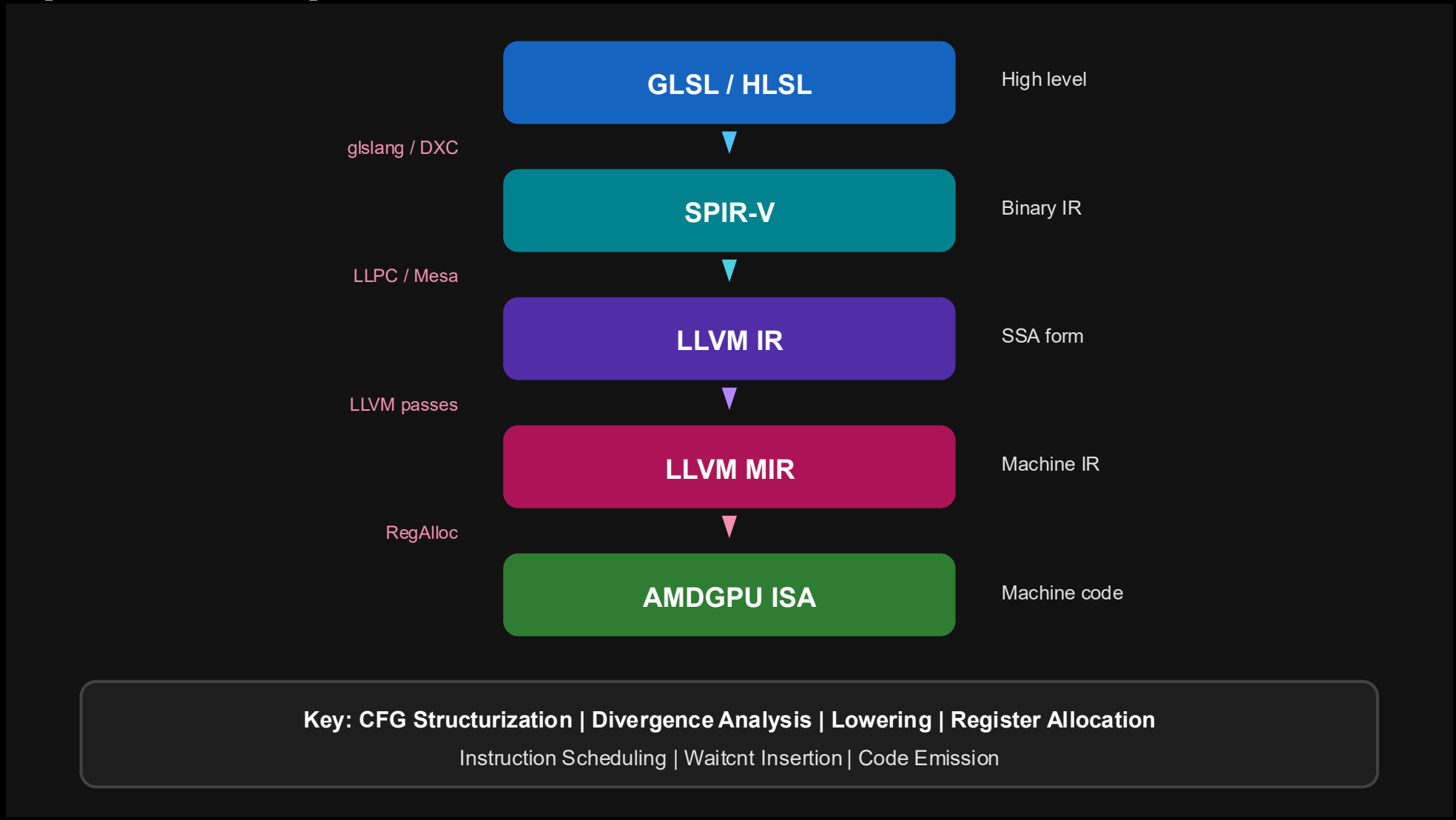
- Passes: AMDGPUUnifyDivergentExitNodes, StructurizeCFG






AMD
together we advance_

# Compilation Pipeline

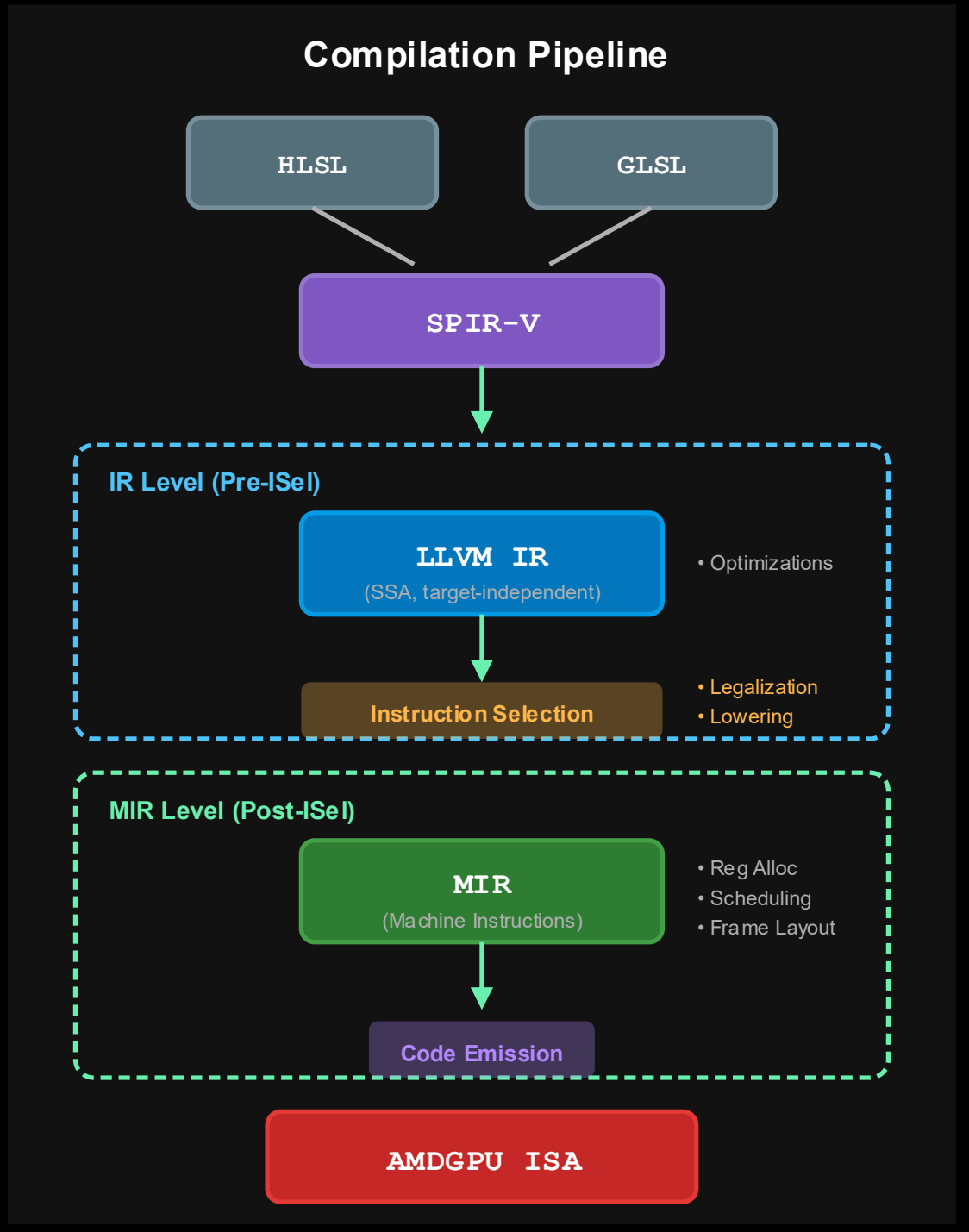





AMD
together we advance_

# Key LLVM Passes for AMDGPU

- IR Level
  - AMDGPUUnifyDivergentExitNodes - ensures single exit
  - InferAddressSpaces - optimizes memory access
  - AMDGPUPropagateAttributes - propagates function attributes
- Pre-ISel
  - SIAnnotateControlFlow - marks divergent branches
  - AMDGPUCodeGenPrepare - prepares IR for codegen
- Post-ISel
  - SILowerControlFlow - lowers to EXEC mask operations
  - SIWholeQuadMode - handles derivative operations
  - SIInsertWaitcnts - inserts memory synchronization
- Register Allocation
  - SIRegisterInfo - defines register classes
  - GCNRegPressure - tracks register pressure



# Divergence Analysis

- Goal: Determine which values differ across threads

- Uniform values
  - Loop induction variables (sometimes)
  - Values loaded from uniform buffers
  - Results of scalar operations

- Divergent values
  - Thread ID (SV_DispatchThreadID)
  - Pixel coordinates
  - Values from per-vertex attributes
  - Results of operations on divergent values

- Why it matters:
  - Uniform → can use SGPR, scalar instructions
  - Divergent → must use VGPR, vector instructions

- LLVM pass: AMDGPUAnnotateUniformValues



AMD
together we advance_

# Instruction Selection for AMDGPU

- From LLVM IR to machine instructions
- LLVM IR

```
%result = fadd float %a, %b
```

- SelectionDAG (or GlobalISel)

```
(fadd f32:$a, f32:$b)
```

- Pattern matching

```
def :
 Pat<(fadd f32:$src0, f32:$src1),
     (V_ADD_F32_e32 VGPR32:$src0, VGPR32:$src1)>;
```

- Output

```
v_add_f32 v0, v1, v2
```

Selection considerations

- VOP2 vs. VOP3 encoding
- Immediate operands
- Modifiers (neg, abs)

```
v_xor_b32     v3, 0x80000000, v1
v_and_b32     v4, 0x7FFFFFFF, v2
v_add_f32_e32 v0, v3, v4

v_add_f32_e64 v0, -v1, |v2|
```



AMD
together we advance_

# GPU Memory Spaces

**Private**

Per-thread

Scratch (in VRAM)

**HIGH LATENCY ⚠**

**LDS (Local)**

Per-workgroup

Shared memory

**LOW LATENCY ✓**

**Global (VRAM)**

All threads

Device memory

**HIGH LATENCY**

**Constants (SMEM, cached)**

**LLVM Address Spaces**

```
addrspace(1) = global
addrspace(5) = private
addrspace(3) = LDS
addrspace(4) = constants
```



AMD
together we advance_

# Memory Operations

- Buffer operations (global memory)

```
buffer_load_dword v0, v1, s[0:3], 0
buffer_store_dword v0, v1, s[0:3], 0
```

- LDS operations (shared memory)

```
ds_read_b32 v0, v1
ds_write_b32 v1, v0
```

- Scalar memory (constants)

```
s_load_dword s0, s[2:3], 0x0
```

- Flat operations (generic address)

```
flat_load_dword v0, v[1:2]
```

- Image operations (textures)

```
image_sample v[0:3], v[4:5], s[0:7], s[8:11]
```



AMD
together we advance_

# Example - Memory operations

```
; RDNA4 (GFX12) - Multiple LDS loads + Global load
  ds_load_b32 v10, v0                ; LDS load #1 (dscnt++)
  ds_load_b32 v11, v1                ; LDS load #2 (dscnt++)
  ds_load_b32 v12, v2                ; LDS load #3 (dscnt++)
  …
  global_load_b32 v20, v[3:4], off ; Global load (loadcnt++)
  …
  v_add_f32 v30, v40, v41
  v_mul_f32 v31, v42, v43
  …
  s_wait_dscnt 0x2                   ; Wait for dscnt ≤ 2
  …                                  ; (v10 ready, v11/v12 in flight)
  v_add_f32 v50, v10, v30            ; USE v10
  …
  s_wait_loadcnt 0x0                 ; Wait for all loads complete
  v_mul_f32 v51, v20, v50            ; USE v20
  …
  s_wait_dscnt 0x0                   ; Wait for remaining LDS
  v_add_f32 v52, v11, v12            ; USE v11, v12
```



AMD
together we advance_

# Key takeaways

1. GPU executes in SIMT model
   - Wavefronts of 32/64 threads execute same instruction
2. Divergence is expensive
   - Compiler must handle if-else with EXEC masking
3. Register pressure affects occupancy
   - More VGPR = fewer wavefronts = less latency hiding
4. Multiple memory spaces
   - Each has different latency and requires different instructions
5. Structured control flow required
   - CFG must be transformed to ensure convergence
6. LLVM AMDGPU backend handles all of this!
   - From SPIR-V to machine code

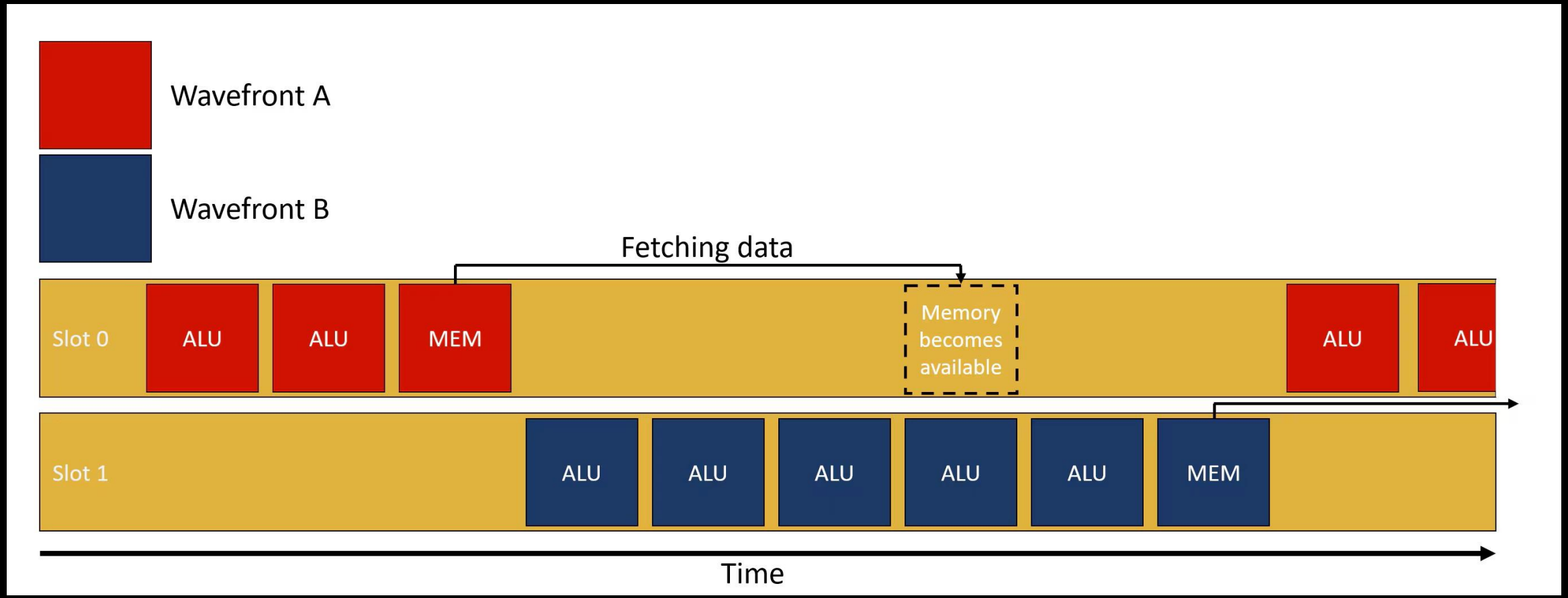


AMD
together we advance_

# Questions?

Thank you for your attention

# Bonus: Occupancy

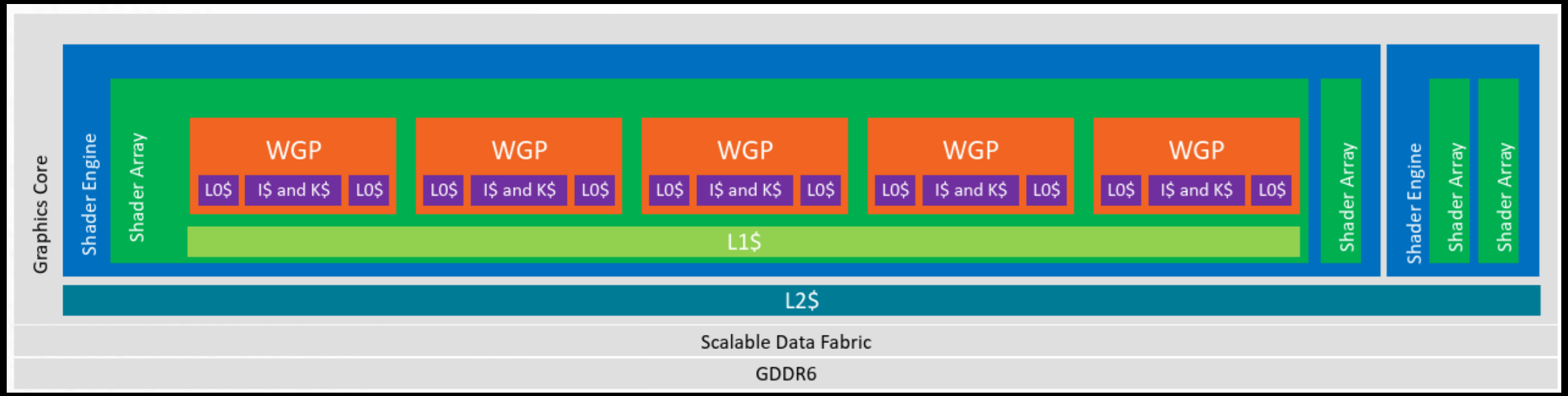



https://gpuopen.com/learn/occupancy-explained/



AMD
together we advance_

# Bonus: RDNA™ overview

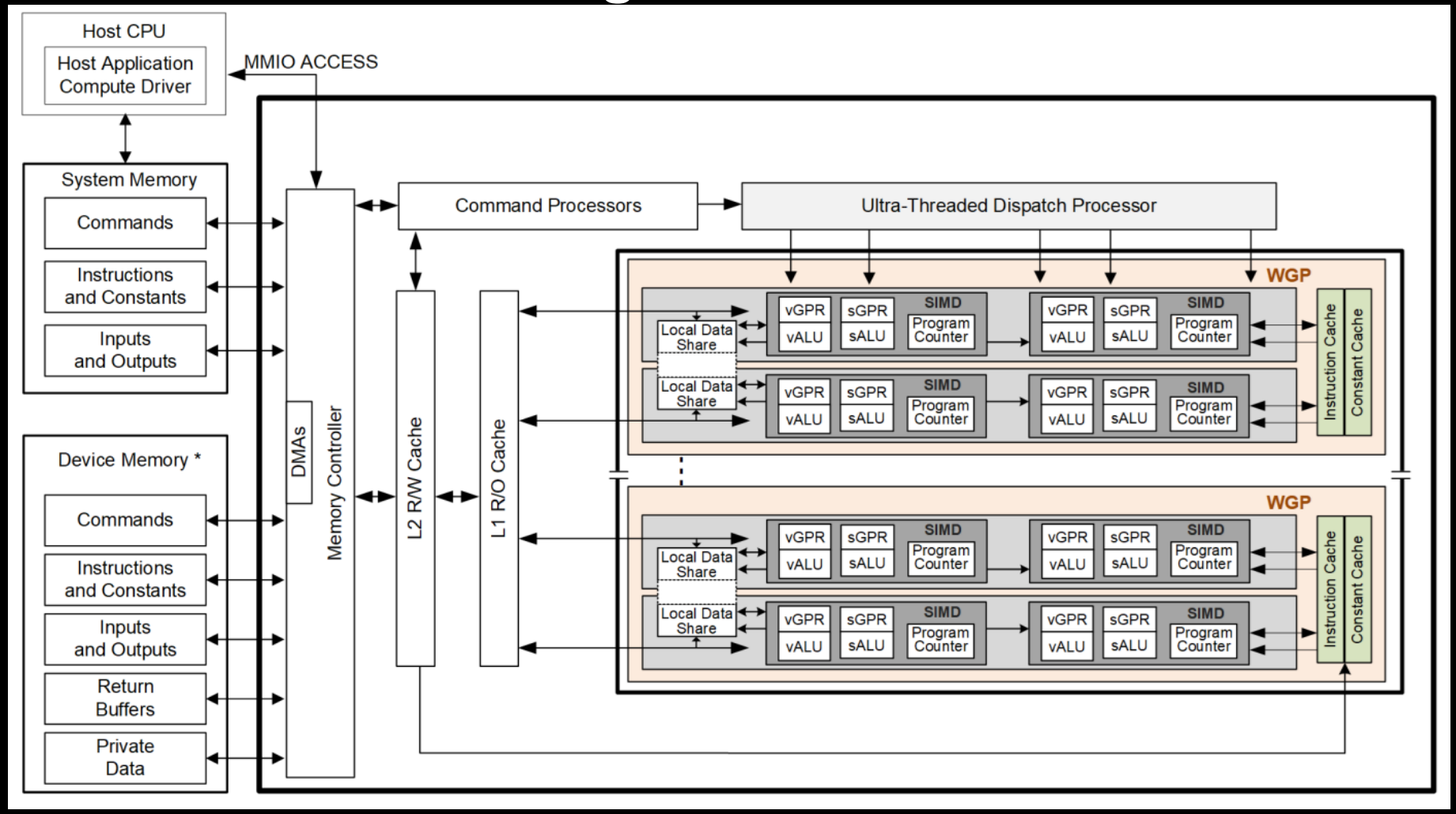



https://gpuopen.com/download/RDNA_Architecture_public.pdf



AMD
together we advance_

# Bonus: RDNA™ 4 Block Diagram

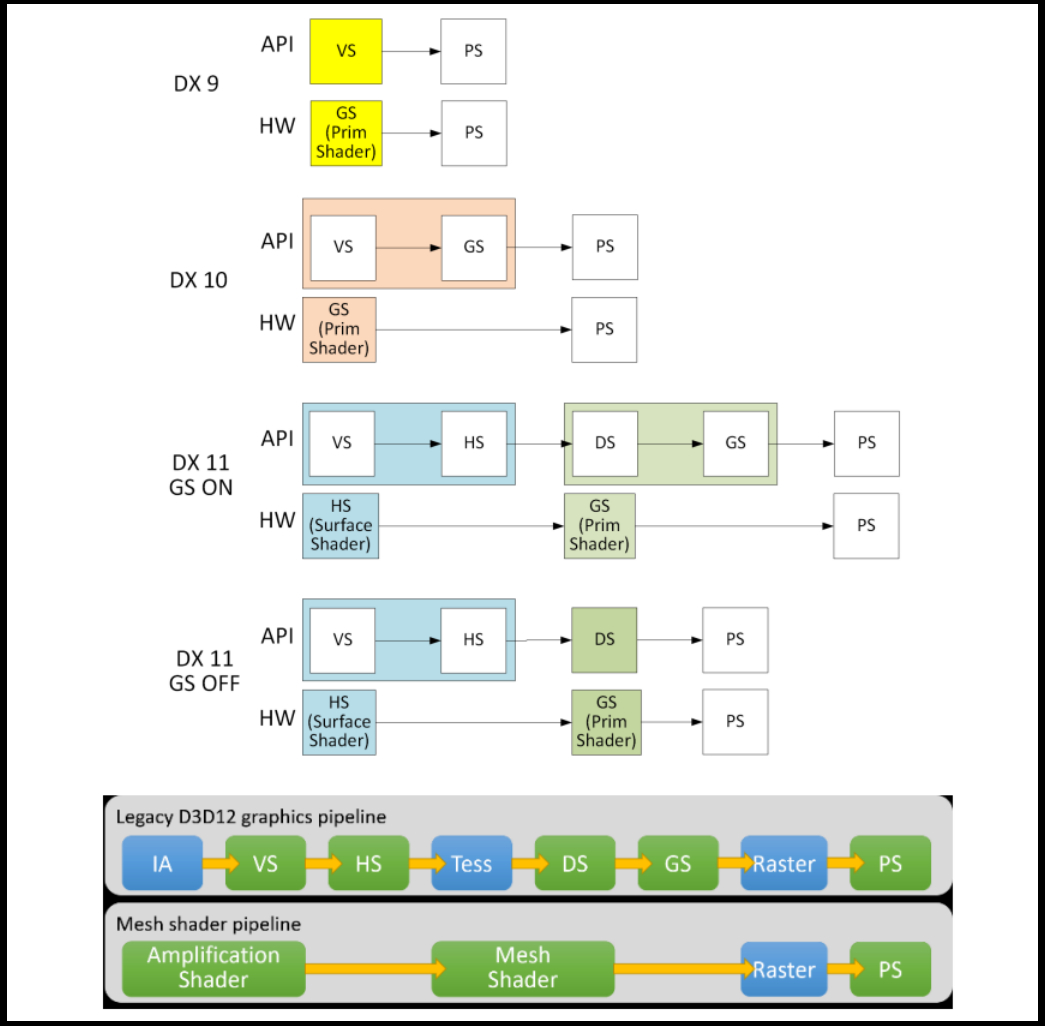



https://docs.amd.com/v/u/en-US/rdna4-instruction-set-architecture



AMD
together we advance_

# Bonus: Graphics pipeline

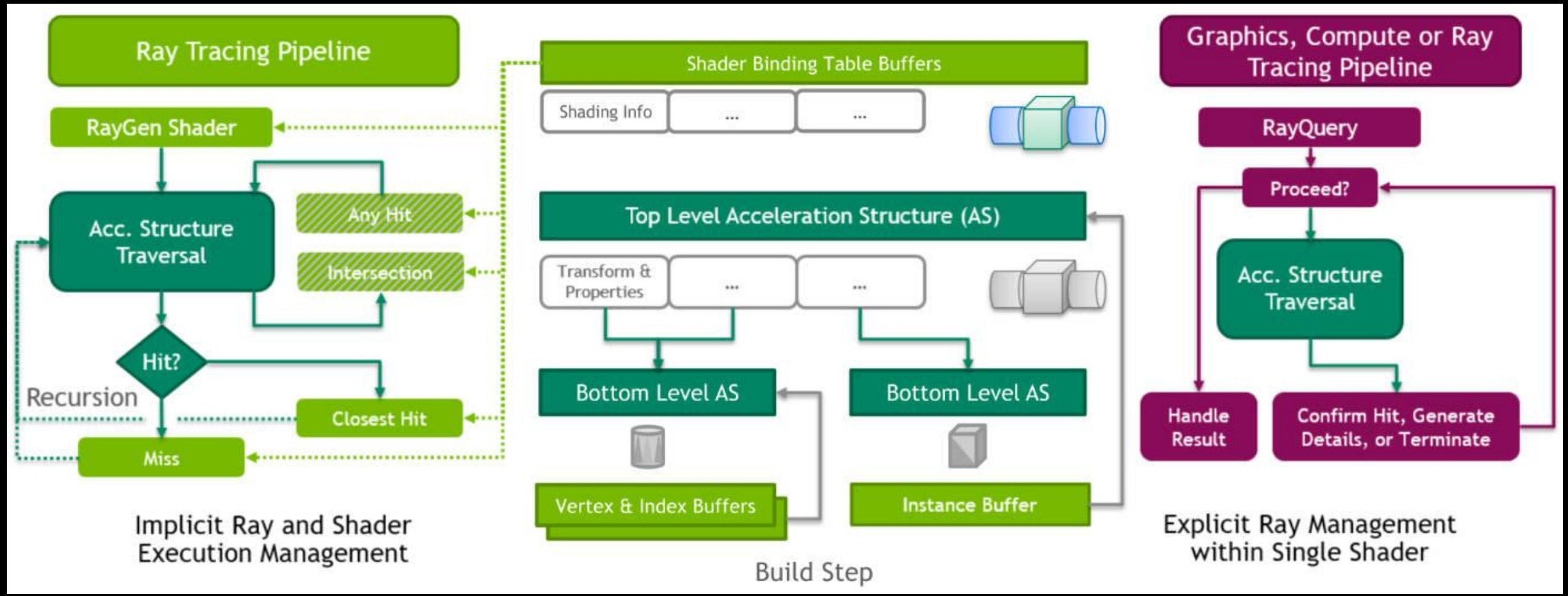



https://docs.amd.com/v/u/en-US/rdna4-instruction-set-architecture



AMD
together we advance_

# Bonus: Ray Tracing Pipeline




https://www.khronos.org/blog/ray-tracing-in-vulkan



AMD
together we advance_

# Public documents and references

- LLVM Project source code
  - https://github.com/llvm/llvm-project
- LLVM Language Reference Manual
  - https://llvm.org/docs/LangRef.html
- LLVM User Guide for AMDGPU Backend
  - https://llvm.org/docs/AMDGPUUsage.html
- AMD RDNA 2 Instruction Set Architecture
  - https://docs.amd.com/v/u/en-US/rdna2-shader-instruction-set-architecture
- AMD RDNA 3 Instruction Set Architecture
  - https://docs.amd.com/v/u/en-US/rdna3-shader-instruction-set-architecture-feb-2023_0
- AMD RDNA 4 Instruction Set Architecture
  - https://docs.amd.com/v/u/en-US/rdna4-instruction-set-architecture
- GPU Open - Occupancy explained
  - https://gpuopen.com/learn/occupancy-explained/
- RDNA Architecture overview
  - https://gpuopen.com/download/RDNA_Architecture_public.pdf



AMD
together we advance_

# Copyright and disclaimer

©2026 Advanced Micro Devices, Inc.  All rights reserved.

AMD, the AMD Arrow logo, RDNA and combinations thereof are trademarks of Advanced Micro Devices, Inc.  Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases,  for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated.  AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION



AMD
together we advance_


AMD