At the heart of every AMD APU/GPU is a power aware high performance set of compute units that have been advancing to bring users new levels of programmability, precision and performance.
AGENDA ➔ AMD Graphic Core Next Architecture

- Unified Scalable Graphic Processing Unit (GPU) optimized for Graphics and Compute
  - Multiple Engine Architecture with Multi-Task Capabilities
  - Compute Unit Architecture
  - Multi-Level R/W Cache Architecture

- What will not be discussed
  - Roadmaps/Schedules
  - New Product Configurations
  - Feature Rollout
SCALABLE MULTI-TASK GRAPHICS ENGINE

- **GFX Command Processor**
- **Work Distributor**
- **Scalable Graphics Engine**
- **Unified Shader Core**

**Key Components:***
- **Primitives Pipes**: Primitive Pipe 0 and Primitive Pipe n
  - **HOS**
  - **Tessellate**
  - **Geometry**
- **Pixel Pipes**: Pixel Pipe 0 and Pixel Pipe n
  - **Scan Conversion**
  - **RB**
- **Memory HUB & L2**

**Abbreviations:**
- **HOS** - High Order Surface
- **RB** - Render Backend
- **CS** - Compute Shader
- **GFX** - Graphics

**Timeline:**
- **June 2011**
SCALABLE MULTI-TASK GRAPHICS ENGINE

- Primitive Scaling ➔ Multiple Primitive Pipelines
- Pixel Scaling ➔ Multiple Screen Partitions
- Multi-task graphics engine use of unified shader

Unified Shader Core

- CS Pipe
- Primitive Pipe 0
  - HOS
  - Tessellate
  - Geometry
- Primitive Pipe n
  - HOS
  - Tessellate
  - Geometry
- Pixel Pipe 0
  - Scan Conversion
  - RB
- Pixel Pipe n
  - Scan Conversion
  - RB

R/W L2

MC HUB & MEM
Asynchronous Compute Engine (ACE)

- Command Processor
  - Hardware Command Queue Fetcher
  - Device Coherent R/W Cache Access
    - Load Acquire/Store Release Semantics
  - Global Data Share Access
  - Hardware synchronization
- Independent & Concurrent Grid/Group Dispatcher
- Real time task scheduling
- Background task scheduling
- Compute Generated Task Graph Processing
  - User Queues
  - Hardware Scheduling
  - Task Queue Context Switching
- Error Detection & Correction (EDCC)
  - For GDDR and internal SRAM Pools
AMD GRAPHIC CORE NEXT ARCHITECTURE

COMPUTE UNIT ARCHITECTURE
SOME CODE EXAMPLES (1)

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

Optional:
Use based on the number of instruction in conditional section.
- Executed in branch unit

//Registers r0 contains “a”, r1 contains “b”
//Value is returned in r2

v_cmp_gt_f32  r0, r1           //a > b, establish VCC
s_mov_b64     s0, exec         //Save current exec mask
s_and_b64     exec, vcc, exec  //Do “if”

s_cbranch_vccz label0           //Branch if all lanes fail
v_sub_f32     r2, r0, r1       //result = a - b
v_mul_f32     r2, r2, r0       //result = result * a

label0:

s_andn2_b64   exec, s0, exec    //Do “else”(s0 & !exec)
s_cbranch_execz label1          //Branch if all lanes fail
v_sub_f32     r2, r1, r0       //result = b - a
v_mul_f32     r2, r2, r1       //result = result * b

label1:

s_mov_b64     exec, s0          //Restore exec mask
**COMPUTE UNIT ARCHITECTURE**

**Input Data:** PC/State/Vector Register/Scalar Register

- **Branch & MSG Unit**
- **Export/GDS Decode**
- **Vector Memory Decode**
- **Scalar Unit**
  - 8 KB Registers
  - Integer ALU
- **SIMD0**
  - 64 KB Registers
  - MP Vector ALU
- **SIMD1**
  - 64 KB Registers
  - MP Vector ALU
- **SIMD2**
  - 64 KB Registers
  - MP Vector ALU
- **SIMD3**
  - 64 KB Registers
  - MP Vector ALU

**Instruction Arbitration**

- **Instruction Fetch**
- **Instruction Decode**

**Arbitration**

- **Msg Bus**
- **Rqst Arb**
- **R/W L2**

**Memory**

- **64 KB LDS Memory**
- **4 CU Shared 16KB Scalar Read Only L1**
- **4 CU Shared 32KB Instruction L1**
Program Counter (PC) / Wavefront (Wave) – Unit of execution, 64 Work-items
- Each Work item can follow a unique code path via predication
- At least 10 Waves/SIMD, 40 Waves/CU ➔ 2560 Work-items per CU
- Independent Instruction Buffer Per Wave
- Instruction fetch Arbitration – Cycle per SIMD (age, priority, emptiness)

I-Cache 32 KB – 64B lines, 4 Way Assoc, Least Recently Used (LRU)
- Peak Bandwidth per CU is 32 bytes/cycle (Optimally 8 instructions)

Special Instructions consumed in 1 cycle in the Instruction Buffers
- s_nop, s_sleep, s_waitcnt, s_barrier, s_setprio, s_setvskip, s_halt

16 Work Group Barrier resources supported per CU
INSTRUCTION ARBITRATION AND DECODE

- A Kernel freely mixes instruction types (Simplistic Programming Model, no weird rules)
  - Scalar/Scalar Memory, Vector, Vector Memory, Shared Memory, etc.

- A CU will issue the instructions of a kernel for a wave-front sequentially
  - Use of predication & control flow enables any single work-item a unique execution path

- Every clock cycle, waves on one SIMDs are considered for instruction issue.
- At most, one instruction from each category may be issued.
- At most one instruction per wave may be issued.
- Up to a maximum of 5 instructions can issue per cycle, not including “internal” instructions.
  - 1 Vector Arithmetic Logic Unit (ALU)
  - 1 Scalar ALU or Scalar Memory Read
  - 1 Vector memory access (Read/Write/Atomic)
  - 1 Branch/Message - s_branch and s_cbranch_<cond>
  - 1 Local Data Share (LDS)
  - 1 Export or Global Data Share (GDS)
  - 1 Internal (s_nop, s_sleep, s_waitcnt, s_barrier, s_setprio)
Independent scalar assist unit to handle special classes of instructions concurrently

- Branch
  - Unconditional Branch (s_branch)
  - Conditional Branch (s_cbranch_<cond>)
    - Condition \( \Rightarrow \) SCC==0, SCC=1, EXEC==0, EXEC!=0, VCC==0, VCC!=0
  - 16-bit signed immediate dword offset from PC provided

- Messages
  - s_msg \( \Rightarrow \) CPU interrupt with optional halt (with shader supplied code and source),
  - debug msg (perf trace data, halt, etc)
  - special graphics synchronization messages
**INTEGER SCALAR UNIT**

- A fully functional “scalar unit” with independent arbitration and decode
  - One scalar ALU or scalar memory read instruction processed per cycle
  - 32/64 bit Integer ALU with memory read support
  - 512 SGPR per SIMD shared between waves, \{SGPRn+1, SGPR\} pair provide 64 bit register

- Scalar Data Cache 16 KB – 64B lines, 4 Way Assoc, LRU replacement policy
  - Peak Bandwidth per CU is 16 bytes/cycle
**SCALAR INSTRUCTIONS / ALU Operations**

- **Scalar ALU instructions**
  - 32-bit arithmetic integer ops
    - Basic signed/unsigned integer arithmetic
    - Support for extended wide integer arithmetic
    - Compare (integer, bit, min/max), move and conditional move (select operations based on SCC)
  - Rich set of 32b/64b bit-wise, bit-field, and bit manipulation ops
  - Constants Operands
    - 32b literals and float or integer hardware constants
  - Scalar Condition Code (SCC) register
Control Flow Instructions (SOPP, some in SOP1, SOPK, SOPC)
- Branch and Msg ➔ Executed in the Branch Unit
- Vector Skip (VSKIP), Barrier, Sleep, Debug, etc ➔ Executed in the Instruction Buffer
- Conditional branch fork & join Instructions ➔ Executed in the Scalar ALU
  - Special Hardware assisted SGPR-stack based conditional branch with fork & join operations
  - Enables optimized control flow and unstructured code input
- Call/Return ➔ Scalar ALU (s_swap_pc) ➔ exchange PC with SGPR pair
- Jump ➔ Scalar ALU (s_setpc) ➔ set PC to absolute address in SGPR pair
SCALAR INSTRUCTIONS | Memory Read Instructions

- Read-Only data from memory via Scalar Data L1 Cache
  - Read 1, 2, 4, 8 or 16 dwords from data cache into adjacent SGPRs
  - Use either simple 64-bit address or resource constant
  - Takes either an immediate offset or offset from SGPR

- Special Ops
  - Read 64b hardware Time Counter
  - Invalidate Scalar R/O L1 data cache
VECTOR ALU UNIT

- Multi-Precision (MP) Single Instruction Multiple Data (SIMD) Units
  - 16 wide Single Precision IEEE floats or 32-bit integers operations per cycle
  - Selectable Double Precision rate options (determined at product build/configuration time)
  - 256 VGPRs shared across waves in SIMD, adjacent pairs form 64 bit registers
### 4 Way VLIW SIMD

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 Single Precision MAC</td>
<td></td>
</tr>
<tr>
<td>VGPR ➔ 64 * 4 * 256-32bit ➔ 256KB</td>
<td></td>
</tr>
<tr>
<td>1 VLIW Instruction * 4 Ops ➔ Dependencies limitations</td>
<td></td>
</tr>
<tr>
<td>3 SRC GPRs, 1 Vector Destination</td>
<td></td>
</tr>
<tr>
<td>Compiler manage VGPR port conflicts</td>
<td>No VGPR port conflicts</td>
</tr>
<tr>
<td>VALU Instruction Bandwidth ➔ 1-7 dwords (~2 dwords/clk)</td>
<td>VALU Instruction Bandwidth ➔ 1-2 dwords/cycle</td>
</tr>
<tr>
<td>Interleaved wavefront instruction required</td>
<td>Vector back-to-back wavefront instruction issue</td>
</tr>
<tr>
<td>Specialized complicated compiler scheduling</td>
<td>Standard compiler scheduling &amp; optimizations</td>
</tr>
<tr>
<td>Difficult assembly creation, analysis, &amp; debug</td>
<td>Simplified assembly creation, analysis, &amp; debug</td>
</tr>
<tr>
<td>Complicated tool chain support</td>
<td>Simplified tool chain development and support</td>
</tr>
<tr>
<td>Less predictive results and performance</td>
<td>Stable and predictive results and performance</td>
</tr>
</tbody>
</table>

### 4 SIMD non-VLIW SIMD
VECTOR INSTRUCTIONS

- 1 - 3 source operands from VGPR without bank conflicts
- 1 broadcast constant from literal, hdw constant, SGPR, or LDS location
- 32 bit instruction encoding for most common vector ALU ops
- 64b encoding enables input/output modifiers and the rich set of ops in current products
- IEEE Vector Compare operations return
  - 1 bit per work-item to a named SGPR pair called Vector Condition Code (VCC)
  - Optionally update the Exec mask (bit per lane)
- Vector Op performs move between VGPR lane to SGPR

<table>
<thead>
<tr>
<th>VOP2</th>
<th>0</th>
<th>OP₆</th>
<th>VSRC₁₈</th>
<th>SRC₀₀</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOP1</td>
<td>0 1 1 1 1 1</td>
<td>VDST₆</td>
<td>OP₃</td>
<td>SRC₀₀</td>
</tr>
<tr>
<td>VOPC</td>
<td>0 1 1 1 1 1 0</td>
<td>OP₃</td>
<td>VSRC₁₈</td>
<td>SRC₀₀</td>
</tr>
</tbody>
</table>

sop2, sop1, sopc, vop2, vop1, vopc can be followed by a 32-bit literal constant

<table>
<thead>
<tr>
<th>VOP3</th>
<th>1 1 1 1 1 0 0</th>
<th>OP₉</th>
<th>5</th>
<th>ABS</th>
<th>VDST₈</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>NEG</td>
<td>OMOD</td>
<td>SRC₂₀</td>
<td>SRC₁₀</td>
<td>SRC₀₀</td>
</tr>
</tbody>
</table>

| VINTRP | 1 1 0 0 0 1 0 | VDST₈ (acc) | OP₂ | ATTR₂ | ATIR | CHAN₉ | VSRC₂₉ (IJ) |

2 Operand
1 Operand
Compare Ops
3 Operand
LOCAL SHARED MEMORY (LDS)

- 64 kb, 32 bank Shared Memory
- Direct mode
  - Vector Instruction Operand ➔ 32/16/8 bit broadcast value
  - Graphics Interpolation @ rate, no bank conflicts
- Index Mode – Load/Store/Atomic Operations
  - Bandwidth Amplification, upto 32 – 32 bit lanes serviced per clock peak
  - Direct decoupled return to VGPRs
  - Hardware conflict detection with auto scheduling
- Software consistency/coherency for thread groups via hardware barrier
- Fast & low power vector load return from R/W L1
LOCAL SHARED MEMORY INSTRUCTIONS

- Encoding for Local Data Share (LDS) indexed ops, and Global Data Share (GDS) ops
- Fully decoupled from ALU instructions
- Instructions
  - Load, Store, Atomic (including fpmin, fpmax, fpcmepswp)
  - 2 source operands Ops from shared memory for accelerated reduction ops
  - Special lane swizzle ops without memory usage
  - GDS Global Wave Sync, Ordered Count Ops – executed by export unit
VECTORS EXPORT INSTRUCTIONS

- Exports move data from 1-4 VGPRs to Graphic Pipeline
  - Color (MRT0-7), Depth, Position, and Parameter
- Global shared memory Ops
VECTOR MEMORY OPERATIONS

- Read/Write/Atomic request are routed to R/W cache hierarchy
  - Variable size addresses /data (4-128b, 8-64b, 16-32b)/cycle
- Addressing unit
  - Address coalescing
  - Image and filter dependant address generation
  - Write Data format conversion
- L1 16KB R/W Vector Data cache
  - 64B cache line, 4 sets x 64 way, LRU Replacement
  - Read-Write Cache (write-through at end of wavefront)
  - Decompression on cache read out
- Return data processing to VGPRs
  - Data filtering, format conversions
  - Optional gather return directly to LDS
## Vector Memory Instructions

MUBUF

<table>
<thead>
<tr>
<th>0000 1111</th>
<th>OP7</th>
<th>LS</th>
<th>A64</th>
<th>LC</th>
<th>VDATA8</th>
<th>VADDR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOFFSET8 (sgr)</td>
<td>TFE</td>
<td>LB</td>
<td>SRSRC6 (T#sgr)</td>
<td>VDATA8 (vgrpr, src or ds)</td>
<td>VADDR8 (vgrpr)</td>
<td></td>
</tr>
</tbody>
</table>

MTBUF

<table>
<thead>
<tr>
<th>0010 1111</th>
<th>NFMT3</th>
<th>DFMT4</th>
<th>OP3</th>
<th>A64</th>
<th>C3</th>
<th>VDATA8</th>
<th>VADDR8</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOFFSET8 (sgr)</td>
<td>TFE</td>
<td>LB</td>
<td>SRSRC6 (T#sgr)</td>
<td>VDATA8 (vgrpr, src or ds)</td>
<td>VADDR8 (vgrpr)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

MIMG

<table>
<thead>
<tr>
<th>1111 1111</th>
<th>OP7</th>
<th>TS7</th>
<th>ACNT</th>
<th>C3</th>
<th>EM</th>
<th>DMASK</th>
<th>LWE</th>
<th>TFE</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSAMP5 (S#sgr)</td>
<td>SRSRC6 (T#sgr)</td>
<td>VDATA8 (vgrpr, src or ds)</td>
<td>VADDR8 (vgrpr)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Three simple classes of Vector memory instructions that will:

- **MUBUF** – read from or perform write/atomic to an un-typed memory buffer/address
  - Data type/size is specified by the instruction operation
- **MTBUF** – read from or write to a typed memory buffer/address
  - Data type is specified in the resource constant
- **MIMG** – read/write/atomic operations on elements from an image surface.
  - Image objects (1-4 dimensional addresses and 1-4 dwords of homogenous data)
  - Image objects use resource and sampler constants for access and filtering
AMD GRAPHIC CORE NEXT ARCHITECTURE
MULTI-LEVEL READ/WRITE CACHE ARCHITECTURE
MULTI-LEVEL READ/WRITE CACHE ARCHITECTURE

- **CU L1 R/W Cache**
  - 16 KB L1, 64B lines, 4 sets x 64 way
  - ~64B/CLK per compute unit bandwidth
  - Write-through – alloc on write (no read) w/dirty byte mask
  - Instruction GLC bit defines cache behavior
    - GLC = 0;
      - Local caching (full lines left valid)
      - Shader write back invalidate instructions
    - GLC = 1;
      - Global coherent (hits within wavefront boundaries)

- **L2 R/W Cache**
  - 64-128KB L2, 64B lines, 16 way set associative
  - ~64B/CLK per channel for L2/L1 bandwidth
  - Write-back - alloc on write (no read) w/ dirty byte mask
  - Acquire/Release semantics control data visibility across CUs (GLC bit on load/store)
    - L2 coherent = all CUs can have the same view of data
  - Remote Atomic Operations
    - Common Integer set & float Min/Max/CmpSwap
### COMPUTE UNIT ARCHITECTURE

- **Input Data:** PC/State/Vector Register/Scalar Register

#### Instruction Fetch Arbitration
- SIMD PC & IB
- SIMD PC & IB
- SIMD PC & IB
- SIMD PC & IB

#### Instruction Arbitration
- SIMD PC & IB
- SIMD PC & IB
- SIMD PC & IB

#### Branch & MSG Unit
- 8 KB Registers
- Integer ALU

#### Scalar Decode
- SIMD 0
- SIMD 1
- SIMD 2
- SIMD 3
- 64 KB Registers
- MP Vector ALU
- MP Vector ALU
- MP Vector ALU
- MP Vector ALU

#### Vector Decode
- SIMD Memory Decode
- Export/GDS Decode

#### LDS Decode
- 64 KB LDS Memory

#### LDS Memory
- 64 KB LDS Memory

#### Bus
- Export Bus
- R/W L1
- R/W L2

#### Req Arb
- R/W L2

#### Rqst
- R/W L2

#### Export
- 16KB

#### 4 CU Shared 16KB Scalar Read Only L1
- 4 CU Shared 32KB Instruction L1
<table>
<thead>
<tr>
<th>4 Way VLIW SIMD</th>
<th>4 SIMD non-VLIW</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 Single Precision MAC</td>
<td>64 Single Precision MAC</td>
</tr>
<tr>
<td>VGPR ➔ 64 * 4 * 256-32bit ➔ 256KB</td>
<td>VGPR ➔ 4 * 64 * 256-32bit ➔ 256KB</td>
</tr>
<tr>
<td>1 VLIW Instruction * 4 Ops ➔ Dependencies limitations</td>
<td>4SIMD * 1 ALU Operation ➔ Occupancy limitations</td>
</tr>
<tr>
<td>3 SRC GPRs, 1 Vector Destination</td>
<td>3 SRC GPRs, 1 Vector\1Scalar Register Destination</td>
</tr>
<tr>
<td>Compiler manage VGPR port conflicts</td>
<td>No VGPR port conflicts</td>
</tr>
<tr>
<td>VALU Instruction Bandwidth ➔ 1-7 dwords(~2 dwords/clk)</td>
<td>VALU Instruction Bandwidth ➔ 1-2 dwords/cycle</td>
</tr>
<tr>
<td>Interleaved wavefront instruction required</td>
<td>Vector back-to-back wavefront instruction issue</td>
</tr>
<tr>
<td>Specialized complicated compiler scheduling</td>
<td>Standard compiler scheduling &amp; optimizations</td>
</tr>
<tr>
<td>Difficult assembly creation, analysis, &amp; debug</td>
<td>Simplified assembly creation, analysis, &amp; debug</td>
</tr>
<tr>
<td>Complicated tool chain support</td>
<td>Simplified tool chain development and support</td>
</tr>
<tr>
<td>Less predictive results and performance</td>
<td>Stable and predictive results and performance</td>
</tr>
</tbody>
</table>
SCALAR + VECTOR

- Simpler ISA compared to previous generation
  - No more clauses
  - No more VLIW packing
  - Control flow more directly programmed

- Scalar engine
  - Lower latency wavefront issue from distributed sequencing of wavefronts
  - Improved performance in previously clause bound cases
  - Lower power handling of control flow as control is closer

- Improved debug support
  - Added HW functionality to improve debug support
**SCALAR + VECTOR**

- Enhanced extended ALU operations
  - Media ops
  - Integer ops
  - Floating point atomics (min, max, cmpxchg)
- Advanced language feature support
  - Exception support
  - Function calls
  - Recursion

- Better architecture consistency
- Easier to understand architecture
- More predictable performance behavior
**R/W CACHE**

- Reads and writes cached
  - Bandwidth amplification
  - Improved behavior on more memory access patterns
  - Improved write to read reuse performance

- Relaxed memory model
  - Consistency controls available to control locality of load/store

- GPU Coherent
  - Acquire/Release semantics control data visibility across the machine (GLC bit on load/store)
  - L2 coherent = all CUs can have the same view of data

- Global atomics
  - Performed in L2 cache
```c
float fn0(float a, float b) {
    if (a > b)
        return ((a - b) * a);
    else
        return ((b - a) * b);
}
```

// Registers r0 contains “a”, r1 contains “b”
// Value is returned in r2

```c
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 // Branch if all lanes fail
v_sub_f32 r2, r0, r1 // result = a - b
v_mul_f32 r2, r2, r0 // result = result * a

s_andn2_b64 exec, s0, exec // Do “else” (s0 & !exec)
s_cbranch_execz label1 // Branch if all lanes fail
v_sub_f32 r2, r1, r0 // result = b - a
v_mul_f32 r2, r2, r1 // result = result * b
s_mov_b64 exec, s0 // Restore exec mask
```
float fn0(float a, float b) {
    if (a > b) {
        return ((a - b) * a);
    } else {
        return ((b - a) * b);
    }
}

// Registers r0 contains “a”, r1 contains “b”
// Value is returned in r2
00 ALU_PUSH_BEFORE
    1 x: PREDGT ____, R0.x, R1.x
        UPDATE_EXEC_MASK UPDATE_PRED
01 JUMP ADDR(3)
02 ALU
    2 x: SUB ____ , R0.x, R1.x
    3 x: MUL_e R2.x, PV2.x, R0.x
03 ELSE POP_CNT(1) ADDR(5)
04 ALU_POP_AFTER
    4 x: SUB ____ , R1.x, R0.x
    5 x: MUL_e R2.x, PV4.x, R1.x
05 POP(1) ADDR(6)
float fn0(float \(a\), float \(b\), float \(c\))
{
    float \(d=0.0\);
    for(int \(i=0; i<100; i++\))
    {
        if(\(c>113.0\))
            break;
        \(c=c*a+b\);
        \(d=d+1.0\);
    }
    return (\(d\));
}
float fn0(float a, float b, float c) {
    float d = 0.0;
    for (int i = 0; i < 100; i++) {
        if (c > 113.0)
            break;
        c = c * a + b;
        d = d + 1.0;
    }
    return (d);
}

// Registers r0 contains “a”, r1 contains “b”, r2 contains “c” and r3 contains “d”. Value is returned in r3

00 ALU
    0 x: mov R3.x 0.0
01 LOOP i0 FAIL_JUMP_ADDR(6)
    02 ALU_BREAK
        1 x: SETGT_INT ____ 100, R0.z
        2 y: AND_INT r0.y, PV7.z, 1
        3 x: PREDNE_INT ____ R0.y, 0.0f
            UPDATE_EXEC_MASK UPDATE_PRED
    03 ALU_BREAK
        10 x: SETGT ____ R2.x,
             (0x42E20000, 113.0f).x
        11 w: AND_INT R0.w, PV10.x, 1
        12 x: PREDE_INT ____ R0.w, 0.0f
            UPDATE_EXEC_MASK UPDATE_PRED
04 ALU
    13 x: MULADD_e R2.x, R0.x, R2.x, R1.x
        z: ADD_INT R0.z, R0.z, 1
    14 x: ADD R3.x, R3.x, 1.0f
05 ENDLOOP i0 PASS_JUMP_ADDR(3)
AMD Graphic Core Next Compute Unit Architecture Summary

- A heavily multi-threaded Compute Unit (CU) architected for throughput
  - Efficiently balanced for graphics and general compute
  - Simplified coding for performance, debug and analysis
  - Simplified machine view for tool chain development
  - Low latency flexible control flow operations
  - Read/Write Cache Hierarchy improves I/O characteristics
  - Flexible vector load, store, and remote atomic operations
  - Coherency domains
  - Load acquire/Store release consistency controls
MULTI-TASK / MULTI-ENGINE UNIFIED COMPUTING GPU

Unified Shader Core

GFX Command Processor

Work Distributor

Scalable Graphics Engine

Primitives Pipe 0
- HOS
- Tessellate
- Geometry

Primitives Pipe n
- HOS
- Tessellate
- Geometry

Pixel Pipe 0
- Scan Conversion
- RB

Pixel Pipe n
- Scan Conversion
- RB

Dual DMA

MC HUB & MEM

L2

R/W
MULTI-TASK UNIFIED COMPUTING

- Asynchronous Compute Engine (ACE)
  - Command processor
    - Hardware Command Queue Fetcher
    - Device coherent R/W Cache Access
    - Can synchronize with other command queues and engines
  - Independent and concurrent grid/workgroup dispatch
  - Queue priority controls
  - Device resource allocation and control
  - Compute Generated Task Graph Processing
    - User task queues
    - HW scheduling of work and resources
    - Task queue context switching

- Multiple Concurrent Contexts
  - Multiple ACE paths
  - Lower overhead dispatch and submission
  - Improved resource management and allocation
  - Out-of-order completion to free up resources as early as possible
QUESTIONS
Disclaimer & Attribution

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 for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. There is no obligation to update or otherwise correct or revise this information. However, we reserve the right to revise this information and to make changes from time to time to the content hereof without obligation to notify any person of such revisions or changes.

NO REPRESENTATIONS OR WARRANTIES ARE MADE WITH RESPECT TO THE CONTENTS HEREOF AND NO RESPONSIBILITY IS ASSUMED FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

ALL IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED. IN NO EVENT WILL ANY LIABILITY TO ANY PERSON BE INCURRED FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

AMD, the AMD arrow logo, and combinations thereof are trademarks of Advanced Micro Devices, Inc. All other names used in this presentation are for informational purposes only and may be trademarks of their respective owners.

OpenCL is a trademark of Apple Inc. used with permission by Khronos.

DirectX is a registered trademark of Microsoft Corporation.

© 2011 Advanced Micro Devices, Inc. All rights reserved.
## ARCHITECTURAL STATE VISIBLE TO A KERNEL

<table>
<thead>
<tr>
<th>PC ➔ Program Counter</th>
<th>EXEC [63:0] ➔ Execute mask – bit per vector lane</th>
</tr>
</thead>
<tbody>
<tr>
<td>VGPR (0-255) ➔ 32 bit, General Purpose Vector Register</td>
<td>SGPR (0-103) ➔ 32 bit General Purpose Scalar Register</td>
</tr>
<tr>
<td>LDS ➔ Local data share space. Up to 32KB</td>
<td>Status Register ➔ Misc status bits (EXECZ, VCCZ, etc)</td>
</tr>
<tr>
<td>VCC [63:0] ➔ Vector Condition Code – bit per vector lane</td>
<td>SCC ➔ Scalar condition code resulting from scalar-ALU op.</td>
</tr>
<tr>
<td>VMCNT, LGKMCNT, EXPCNT ➔ Dependency counters</td>
<td>ALLOC ➔ SGPR, VGPR, LDS Base and size</td>
</tr>
<tr>
<td>M0 ➔ Memory Descriptor register (Use shared memory access)</td>
<td>TIME ➔ 64 bit time</td>
</tr>
<tr>
<td>HDW_ID ➔ Hardware ID register</td>
<td>Mode ➔ FP modes, exception enables</td>
</tr>
<tr>
<td>TRAPSTS ➔ Exceptions status registers</td>
<td>TBA, TMA, TTMP ➔ Trap registers</td>
</tr>
</tbody>
</table>
VECTOR UNIT CHARACTERISTICS

- **FMA** (Fused Multiply Add), IEEE 754-2008 precise with all round modes, proper handling of Nan/Inf/Zero and full de-normal support in hardware for SP and DP
- **MULADD** single cycle issue instruction without truncation, enabling a MULieee followed by ADD ieee to be combined with round and normalization after both multiplication and subsequent addition
- **VCMP** A full set of operations designed to fully implement all the IEEE 754-2008 comparison predicates
- **IEEE Rounding Modes** (Round to nearest even, Round toward +Infinity, Round toward –Infinity, Round toward zero) supported under program control anywhere in the shader. Double and single precision modes are controlled separately. Applies to all slots in a VLIW.
- **De-normal Programmable Mode** control for SP and DP independently. Separate control for input flush to zero and underflow flush to zero.
- **DIVIDE ASSIST OPS** IEEE 0.5 ULP Division accomplished with macro in (SP/DP ~15/41 Instruction Slots respectively)
- **FP Conversion Ops** between 16-bit, 32-bit, and 64-bit floats with full IEEE 754 precision and rounding
- **Exceptions Support** in hardware for floating point numbers with software recording and reporting mechanism. Inexact, Underflow, Overflow, division by zero, de-normal, invalid operation, and integer divide by zero operation
- **64-bit Transcendental Approximation** Hardware based double precision approximation for reciprocal, reciprocal square root and square root
- **24 BIT INT MUL/MULADD/LOGICAL/SPECIAL** @ full SP rates
  - Heavy use for Integer thread group address calculation
  - 32-bit Integer MUL/MULADD @ DPFP Mul/FMA rate