DWARF Extensions for Optimized SIMT/SIMD (GPU) Debugging

GNU Tools Cauldron @ Linux Plumbers Conference 2021

Tony Tye, Scott Linder, Zoran Zaric
AMD Open Source Debugging Tools Project

• AMD ROCm ROCgdb debugger
  • https://github.com/ROCm-Developer-Tools/ROCgdb

• DWARF Extensions For Heterogeneous Debugging
  • https://llvm.org/docs/AMDGPUDwarfExtensionsForHeterogeneousDebugging.html

• User Guide for AMDGPU Backend: DWARF Debug Information
  • https://llvm.org/docs/AMDGPUUsage.html#dwarf-debug-information
Heterogeneous Computing Debugging Challenges

- GPUs and other heterogeneous computing devices have:
  - Multiple memory address spaces
  - Many wide vector registers
  - Many scalar registers
  - Language threads of execution => lanes of a SIMD/SIMT execution model

Example: GPU Hardware

SIMD/SIMT Execution Model
Heterogeneous Computing Debugging Challenges (cont.)

- Variables more often spread across pieces of different storage kinds
- SIMD/SIMT execution needs runtime selection of pieces of vector registers
- Complex expressions benefit from factorization of location definitions
  - Currently not possible => duplicate parts of location definitions
Existing DWARF 5 Limitations

• Unable to describe variables in combinations of parts of registers => no static or runtime indexing
• Some features only work when located in memory => type attribute expressions requiring a base object
• DWARF procedures can only accept global memory address arguments
• No vector base types needed to describe vector registers
• Cannot create memory locations in address spaces
• CFI does not allow composite locations
• CFI does not support address spaces
• Bit field offsets are not supported for all location kinds
How to fix this?

• Explored numerous approaches to overcome limitations
  • Approach chosen was simplest and provided most benefits
• Based on:
  • Generalizing execution model
  • Composable and consistent operations
• Results in small number of new operations that compose generally
• As opposed to adding many specialized operations and rules
  • Causes contextual semantics and corner cases
  • Harder for compilers and debuggers
• Major aspect is to allow locations to be manipulated on the stack
Main Goals

• Upstreamable:
  • GDB debugger
  • LLVM compiler
  • GCC compiler
• Supportable by other tools:
  • TotalView debugger
• Backwards compatible with DWARF 5
• Support optimized code for GPUs
• Benefit non-GPU targets too
What Is DWARF?

• A standard way to specify debug information
  • Describes source language entities: compilation units, functions, types, variables, etc.
  • Embedded in sections of code object executables
• Maps source program language entities to the hardware representation:
  • Program counter <=> source line
  • Source function => entry point program counter
  • Source language variable => location at a particular program counter
  • Source function call stack virtual unwinding => locations of callee saved registers
  • Etc.
DWARF Expressions

- Great diversity in locations of entities
- Locations involve runtime values
- Variable location could be:
  - In register
  - At memory address
  - At Stack Pointer + offset
  - Optimized away known value => such as compile time value
  - Optimized away unknown value => such as unused variables
  - Spread across combination of locations
  - At memory address, but also transiently loaded into registers
- Consequently, DWARF includes rich expression language
  - Expression evaluated on simple stack machine
  - Expression evaluated in a context => value/location result kind, process/thread/lane/frame/pc, initial stack, etc.
  - Some context defined by place expression used => result kind, initial stack, etc.
DWARF 5: Dynamic Array Size (1 of 3)

Example: Runtime size of a dynamic array
- Context result kind => value
- Expression evaluated one operation at a time using a stack
• **DW_OP_regval_type** => reads register and pushes value on stack
• Each stack element is a value and associated type
• Type must be a DWARF base type => specifies encoding, byte ordering, and size of value
  • Defaults to *Generic* type => architecture specific integral encoding/byte ordering, the size of global memory address
DWARF 5: Dynamic Array Size (3 of 3)

- Value result kind => result is the top stack value

Expression

- DW_OP_regval_type SGPRO Generic
- DW_OP_deref

Stack

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>42</td>
</tr>
</tbody>
</table>

Context

Result: Value

Source Program

Array Size: 42

Hardware

SGPR 0

Memory

... 42 ...
## DWARF 5: Variable Location in Register (1 of 2)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Stack</th>
<th>Location</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW_OP_regx SGPR0</td>
<td></td>
<td></td>
<td>Result: Location</td>
</tr>
</tbody>
</table>

### Example: Source variable located in a register

- **Context result kind** => location
  - *Note: DWARF uses term location description*
• **DW_OP_regx** => creates register location
• **DWARF** conceptually has a separate location area
  • Does not use the stack
• **Location result kind** => result is the location area
Example: Source variable in stack frame memory at address stack pointer + 0x10
DWARF 5: Variable Location Memory (2 of 4)

- `DW_OP_regval_type` => pushes value read from stack pointer register

```
DWARF PC

Expression

DW_OP_regval_type SGPR0 Generic
DW_OP_plus_uconst 0x10

Stack

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>0xa3c0f00</td>
</tr>
</tbody>
</table>

Location

Result: Location

Source Program

Variable

Hardware

SGPR 0

Memory

Stack Frame
``
DWARF 5: Variable Location Memory (3 of 4)

- Variable location is 0x10 bytes from the base of the stack frame
- `DW_OP_plus_uconst` => pop value, add 0x10, and push result
• Location area empty when location result kind => convert top stack element to memory location
  • Use value as global memory address
DWARF 5: Variable Spread Across Different Locations (1 of 7)

Example: Source variable that is partly in a register, partly undefined, and partly in memory

- Composite location => 1 or more parts
  - Each part specifies a location and number of bytes used from it
DWARF 5: Variable Spread Across Different Locations (2 of 7)

- DW_OP_regx => creates register location

<table>
<thead>
<tr>
<th>Expression</th>
<th>Stack</th>
<th>Location</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW_OP_regx SGPR3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_piece 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_piece 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_bregx SGPR0 0x10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_piece 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source Program

Hardware

Memory

Stack Frame

Register
Number
SPGR3

Result: Location
DWARF 5: Variable Spread Across Different Locations (3 of 7)

- **DW_OP_piece** => first use creates incomplete composite location
  - Location in location area used in first part
  - Size 4 indicates number of bytes used from beginning of part’s location
• **DW_OP_piece** => subsequent use adds part to already created incomplete composite location
  • Parts form a contiguous set of bytes
  • If no other location in location area, and no value on stack => part implicitly the undefined location
  • 2 indicates there are 2 undefined bytes

• Undefined location => used to indicate part that has been optimized away
DWARF 5: Variable Spread Across Different Locations (5 of 7)

- **DW_OP_bregx** => read register as Generic type, add 0x10, and push value
DWARF 5: Variable Spread Across Different Locations (6 of 7)

- DW_OP_piece => add part to already created incomplete composite location
- If no other location in location area, but value on stack => part is memory location with address popped from stack
- 2 indicates there are 2 bytes from memory

<table>
<thead>
<tr>
<th>Expression</th>
<th>Stack</th>
<th>Location</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW_OP_regx SGPR3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_piece 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_piece 2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_bregx SGPR0 0x10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_piece 2</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source Program

Hardware

Memory

Stack Frame

null

<table>
<thead>
<tr>
<th>Size</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Register Number</td>
</tr>
<tr>
<td></td>
<td>SPGR3</td>
</tr>
<tr>
<td>2</td>
<td>Undef</td>
</tr>
<tr>
<td>2</td>
<td>Memory Address</td>
</tr>
<tr>
<td></td>
<td>0x0a3c6f10</td>
</tr>
</tbody>
</table>
DWARF 5: Variable Spread Across Different Locations (7 of 7)

- Incomplete composite location implicitly converted to complete composite location
Example: Offsetting a composite location not supported

- Extend previous example to offset location built so far
- **Variable Location in Memory** example used `DW_OP_plus` => convenient way to offset memory address

DWARF 5: Offsetting a Composite Location (1 of 2)
DWARF 5: Offsetting a Composite Location (2 of 2)

- However, `DW_OP_plus` cannot be used to offset a composite location => it only operates on the stack
- Compiler would need to make a different composite location => starting at the part corresponding to offset
  - `DW_OP_piece 1`
  - `DW_OP_bregx SGPR0 0x10`
  - `DW_OP_piece 2`
- Operations on values are not composable with locations
What Is A Location?

- Location storage is contiguous linear organization of certain number of bytes
- All location kinds can be viewed the same way:
  - Global memory => linear stream of bytes of the architecture’s address size
  - Register => linear stream of bytes of the size of each architecture’s register
  - Composite location => linear stream of the bytes defined by the parts
  - Implicit location => linear stream of bytes of the value using the type’s byte ordering
  - Undefined location => infinitely sized storage where every byte is undefined
- A location is comprised of:
  - A kind (memory, register, etc.)
  - A reference to a specific location storage of that kind
  - A zero-based offset within the location storage
Stack Location Operations

- If location could be allowed on the stack:
  - Define new operations to work on locations in compossible manner
  - Example: new `DW_OP_LLVM_offset` => updates offset of any location kind
- Existing operations can be generalized => act on locations of any kind
  - Example: `DW_OP_deref` => pop a location (rather than memory address value), read it
  - Backwards compatibility via implicit conversions
- Key part of extension is allowing locations on stack
  - DWARF 5 expressions can be evaluated unchanged and yield same results
  - Extension allows greater expressiveness => see following examples
Extension: Variable spilled to part of VGPR (1 of 3)

Example: Compiler spills variable stored in SGPR register to fixed lane of VGPR register

- In this case lane 5 of VGPR0 => each VGPR lane is 4 bytes
  - So index is 5 * 4 = 20
Extension: Variable spilled to part of VGPR (2 of 3)

- `DW_OP_regx` => now pushes location on stack with byte offset of 0
Extension: Variable spilled to part of VGPR (3 of 3)

- DW_OP_LLVM_offset => can offset register location
- DWARF 5 does not support specifying register offset => can only have locations starting at beginning
- Defining register names for every part of every register => not practical for GPUs due to sheer number
- Separate register names would not allow computed runtime indexing of register parts
- GPU compilers frequently locate variable in parts of the numerous wide vector registers
  - Especially in optimized code to avoid memory accesses
  - Runtime indices used to support SIMT execution model
- Result is top stack entry

Source Program

<table>
<thead>
<tr>
<th>Variable</th>
<th>Hardware</th>
</tr>
</thead>
<tbody>
<tr>
<td>SGPR 0</td>
<td>Lane 0 + Lane 1 + Lane 2 + Lane 3 + Lane 4 + Lane 5 + Lane 63</td>
</tr>
</tbody>
</table>

New

- DW_OP_LLVM_offset => can offset register location
- DWARF 5 does not support specifying register offset => can only have locations starting at beginning
- Defining register names for every part of every register => not practical for GPUs due to sheer number
- Separate register names would not allow computed runtime indexing of register parts
- GPU compilers frequently locate variable in parts of the numerous wide vector registers
  - Especially in optimized code to avoid memory accesses
  - Runtime indices used to support SIMT execution model
- Result is top stack entry
Example: Source variable located across a SIMT lane of multiple VGPR registers

- GPU compiler maps language threads to VGPR lanes in SIMT manner
- Thread’s variable is spread across the same lane of multiple VGPRs
- Context specifies lane => lane corresponds to focused source thread
Extension: Variable across multiple VGPRs (2 of 14)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Stack</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td>DW_OP_regs VGPR0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_LLVM_push_lane</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_uconst 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_mul</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_LLVM_offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_piece 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_regs VGPR1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_LLVM_push_lane</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_uconst 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_mul</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_LLVM_offset</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DW_OP_piece 4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Source Program

Variable

Hardware

Focused Thread

Focused Lane

Lane 0  Lane 1  Lane 2  Lane 3  Lane 4  Lane 5  Lane 63

VGPR 0  ...  VGPR 1  ...

DWARF PC

Result: Location Lane: 5
Extension: Variable across multiple VGPRs (3 of 14)

- DW_OP_LLVM_push_lane => pushes value of context’s lane

New

Source Program

Stack

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>VGPR0</td>
</tr>
</tbody>
</table>

Context

Result: Location
Lane: 5

Variable

Hardware

Focused Thread

Focused Lane

Lane 0  Lane 1  Lane 2  Lane 3  Lane 4  Lane 5  Lane 63

VGPR 0  

VGPR 1  

...  

...  

DVGR 0

DVGR 1
Extension: Variable across multiple VGPRs (4 of 14)

- Each VGPR lane is 4 bytes => lane must be multiplied by 4 to get register byte index
Extension: Variable across multiple VGPRs (5 of 14)

Source Program

Hardware

Variable

Focused Thread

Focused Lane

Lane 0  Lane 1  Lane 2  Lane 3  Lane 4  Lane 5  Lane 6

VGPR 0

VGPR 1

DWARF PC

Expression

DW_OP_regx VGPR0
DW_OP_LLVM_push_lane
DW_OP_uconst 4
DW_OP_mul
DW_OP_LLVM_offset
DW_OP_piece 4
DW_OP_regx VGPR1
DW_OP_LLVM_push_lane
DW_OP_uconst 4
DW_OP_mul
DW_OP_LLVM_offset
DW_OP_piece 4

Stack

Context

Result: Location
Lane: 5

Register

Offset | Number

0     | VGPR0

Value

Type  | Value

Generic | 20

Top

Extension: Variable across multiple VGPRs (6 of 14)

Expression

```
DW_OP_regx VGPR0
DW_OP_LLVM_push_lane
DW_OP_uconst 4
DW_OP_mul
DW_OP_LLVM_offset
DW_OP_piece 4
DW_OP_regx VGPR1
DW_OP_LLVM_push_lane
DW_OP_uconst 4
DW_OP_mul
DW_OP_LLVM_offset
DW_OP_piece 4
```

Stack

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>VGPR0</td>
</tr>
</tbody>
</table>

Context

Result: Location
Lane: 5

Source Program

```
Variable
```

Hardware

```
Lane 0  Lane 1  Lane 2  Lane 3  Lane 4  Lane 5  Lane 63
VGPR 0   VGPR 1
```

Focused Thread

Focused Lane
Extension: Variable across multiple VGPRs (7 of 14)

- **DW_OP_piece** => now pops location from stack to use for the new part
- For backwards compatibility: if stack is empty or the top element is an incomplete composite location, implicitly uses the undefined location; if top element is a Generic value, it is implicitly converted to a global memory location

- Checks if the top stack element is an incomplete composite location
  - If not, creates new incomplete composite location with no parts
  - Otherwise, pops previously created incomplete composite location
- Adds new part to incomplete composite location and pushes on stack

**Source Program**
- `Lane 0` to `Lane 63`
- `VGPR 0` to `VGPR 1`

**DWARF PC**
- `Lane:5`

**Stack**
- **Composite (incomplete)**
  - **Offset**
  - **Size**
  - **Parts**
  - **Location**
    - **Register**
    - **Offset**
    - **Number**
  - **Top**

**Context**
- **Result**: Location
  - **Lane**: 5

**Hardware**
- **Focused Lane**
  - `Lane 0` to `Lane 63`
  - `VGPR 0`
  - `VGPR 1`
Extension: Variable across multiple VGPRs (8 of 14)

**Source Program**
- Variable
- Hardware
- Focused Thread
- Focused Lane

**Extension: Variable across multiple VGPRs**

**Expression**
- DW_OP_regx VGPR0
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4
- DW_OP_regx VGPR1
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Parts</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Register Offset Number Location</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>20 VGPR0</td>
<td></td>
</tr>
</tbody>
</table>

**Context**
- Result: Location Lane: 5

**DWARF PC**

```
Variable
 Hardware
 Focused Thread
 Focused Lane
```

```
Lane 0  Lane 1  Lane 2  Lane 3  Lane 4  Lane 5  Lane 63

VGPR 0
 VGPR 1
...    ...
```
Extension: Variable across multiple VGPRs (9 of 14)

### Expression
- DW_OP_regx VGPR0
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4
- DW_OP_regx VGPR1
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4

### Stack

#### Composite (Incomplete)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Parts</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Register Offset Number</td>
<td>Location</td>
</tr>
<tr>
<td>20</td>
<td>VGPR0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Register

<table>
<thead>
<tr>
<th>Offset</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VGPR1</td>
</tr>
</tbody>
</table>

#### Value

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>5</td>
</tr>
</tbody>
</table>

### Context

Result: Location
Lane: 5

Source Program

Hardware

Focused Lane

Focused Thread

DWARF PC
Extension: Variable across multiple VGPRs (10 of 14)

**Expression**

- DW_OP_regx VGPR0
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4
- DW_OP_regx VGPR1
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4

**Stack**

<table>
<thead>
<tr>
<th>Composite (Incomplete)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Offset</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>VGPR0</td>
</tr>
</tbody>
</table>

**Context**

- Result: Location
- Lane: 5

**Source Program**

- Variable

**Hardware**

- Focused Thread
- Focused Lane

**Stack Diagram**

```
Lane 0  Lane 1  Lane 2  Lane 3  Lane 4  Lane 5  Lane 63
VGPR 0  
VGPR 1  
```

**Register**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VGPR1</td>
</tr>
</tbody>
</table>

**Value**

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>5</td>
</tr>
</tbody>
</table>

**Top**

- 4
Extension: Variable across multiple VGPRs (11 of 14)

**Expression**
- DW_OP_rex VGPR0
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4
- DW_OP_rex VGPR1
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4

**Context**
- Result: Location
  - Lane: 5

**Stack**
- Composite (incomplete)
  - Offset: 0
    - Size: 4
    - Parts: Register
      - Offset: 0
      - Number: 20
        - VGPR0

- Register
  - Offset: 0
  - Number: VGPR1

- Value
  - Type: Generic
  - Value: 20

**Source Program**
- Variable
- Focused Thread

**Hardware**
- Focused Lane
- VGPR 0
- VGPR 1

**Focused Lane**
- Lane 0, Lane 1, Lane 2, Lane 3, Lane 4, Lane 5, ..., Lane 63
Extension: Variable across multiple VGPRs (12 of 14)

Expression

- DW_OP_regx VGPR0
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4
- DW_OP_regx VGPR1
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4

DWARF PC

Stack

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Parts</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>Register</td>
<td>Offset Number</td>
</tr>
<tr>
<td></td>
<td></td>
<td>20</td>
<td>VGPR0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>20</td>
<td>VGPR1</td>
</tr>
</tbody>
</table>

Context

Result: Location
Lane: 5

Source Program

Variable

Focused Thread

Hardware

Focused Lane

Lane 0 Lane 1 Lane 2 Lane 3 Lane 4 Lane 5 Lane 63

VGPR 0

VGPR 1
Extension: Variable across multiple VGPRs (13 of 14)

- **DW_OP_piece** => pops location and adds as new part to incomplete composite location on top of stack

**Expression**
- DW_OP_regx VGPR0
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4
- DW_OP_regx VGPR1
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4

**Stack**

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Parts Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>4</td>
<td>Register Offset Number 20 VGPR0</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>Register Offset Number 20 VGPR1</td>
</tr>
</tbody>
</table>

**Context**

Result: Location Lane:5

**Source Program**
- Variable
- Hardware

**Focused Lane**
- VGPR 0
- VGPR 1
Extension: Variable across multiple VGPRs (14 of 14)

- If top of stack is incomplete composite location => implicitly converted to complete composite location

Source Program

Hardware

Focused Thread

Focused Lane

Lane 0 Lane 1 Lane 2 Lane 3 Lane 4 Lane 5 Lane 63

Variable

VGPR 0

VGPR 1
Extension: Variable across multiple kinds of locations (1 of 7)

Example: Source variable located across register, memory, and implicit locations

- Same as previous example, except last 4 bytes are from memory and a constant value

Source Program

```
Example: Source variable located across register, memory, and implicit locations

• Same as previous example, except last 4 bytes are from memory and a constant value
```
Extension: Variable across multiple kinds of locations (2 of 7)

- DW_OP_addr => now pushes a memory location
Extension: Variable across multiple kinds of locations (3 of 7)

**Expression**

- DW_OP_regx VGPR0
- DW_OP_LLVM_push_lane
- DW_OP_uconst 4
- DW_OP_mul
- DW_OP_LLVM_offset
- DW_OP_piece 4
- DW_OP_addr 0xbeef
- DW_OP_piece 2
- DW_OP_uconst 0xf00d
- DW_OP_stack_value
- DW_OP_piece 2
- DW_OP_LLVM_piece_end

**Stack**

<table>
<thead>
<tr>
<th>Composite (incomplete)</th>
<th>Offset</th>
<th>Size</th>
<th>Parts</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>4</td>
<td>Register</td>
<td>Offset</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>2</td>
<td>Memory</td>
<td>Offset</td>
</tr>
</tbody>
</table>

**Context**

Result: Location
Lane: 5

- DW_OP_piece => adds memory location as next piece of composite location
Extension: Variable across multiple kinds of locations (4 of 7)

- Last 2 bytes are constant value 0xf00d
Extension: Variable across multiple kinds of locations (5 of 7)

- DW_OP_stack_value =>
  - Pops value
  - Creates implicit location storage using value’s base type size and byte order
  - Pushes implicit location referencing implicit location storage

Source Program

Hardware

Variable

Focused Thread

Memory

Composite (incomplete)

<table>
<thead>
<tr>
<th>Offset</th>
<th>Size</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>Register</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Offset</td>
</tr>
<tr>
<td></td>
<td>20</td>
<td>VGPR0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Offset</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0xf00d</td>
</tr>
</tbody>
</table>

Focused Lane
Extension: Variable across multiple kinds of locations (6 of 7)

- **DW_OP_piece** => *adds implicit location as next piece of composite location*
Extension: Variable across multiple kinds of locations (7 of 7)

- DW_OP_LLVM_piece_end => explicitly completes incomplete composite location on top of stack
  - Permits location operations to be used => such as DW_OP_LLVM_offset
  - Permits creation of multiple composite locations on stack => used to pass to DW_OP_call*
### Extension: Address Spaces (1 of 5)

---

**Example: Source variable in stack frame address space memory at address stack pointer + 0x10**

- Devices can have multiple hardware supported address spaces
  - Specific hardware instructions to access address spaces
- **DWARF 5** `DW_OP_xderef` => dereferences a memory address using an address space
  - No way to create address in a specific address space
  - No way to include address space memory locations in parts of composite locations

---

<table>
<thead>
<tr>
<th>Expression</th>
<th>Stack</th>
<th>Context</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>DWARF PC</code></td>
<td><code>DW_OP_regval_type SGPR0 Generic</code></td>
<td><strong>Result:</strong> Location</td>
</tr>
<tr>
<td><code>DW_OP_uconst 1</code></td>
<td><code>DW_OP_LLVM_form_aspace_address</code></td>
<td></td>
</tr>
<tr>
<td><code>DW_OP_LLVM_offset 0x10</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Extension: Address Spaces (2 of 5)

- GPUs use separate address space for per lane managed storage => used by stack pointer
- DW_OP_regval_type => push stack pointer address

Expression

- DW_OP_regval_type
- SGPR0
- Generic

Stack

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>0x0a3c0f00</td>
</tr>
</tbody>
</table>

Context

Result: Location

Source Program

variable

Hardware

SGPR 0

Per Lane Memory

Global Memory

Stack Frame
Extension: Address Spaces (3 of 5)

- **DW_OP_uconst** => push address space number
- Architecture defines numbers => address space 1 is per lane memory

### Source Program

- Variable

### Hardware

- SGPR 0
- Per Lane Memory
- Global Memory
- Stack Frame

### DWARF PC

- DW_OP_regval_type SGPR0 Generic
- DW_OP_uconst 1
- DW_OP_LLVM_form_aspace_address
- DW_OP_LLVM_offset 0x10

### Extension: Address Spaces (3 of 5)

<table>
<thead>
<tr>
<th>Type</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generic</td>
<td>0x0a3c0f00</td>
</tr>
<tr>
<td>Generic</td>
<td>1</td>
</tr>
</tbody>
</table>

**Result:** Location
Extension: Address Spaces (4 of 5)

- DW_OP_regval_type: SGPRO Generic
- DW_OP_Uconst 1
- DW_OP_LLVM_form_aspace_address
- DW_OP_LLVM_offset 0x10

**Expression**

DWARF PC

**Stack**

<table>
<thead>
<tr>
<th>Memory</th>
<th>Offset</th>
<th>ASpace</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0x0a3c0f00</td>
<td>1</td>
</tr>
</tbody>
</table>

**Result:** Location

- DW_OP_LLVM_form_aspace_address => pops value and address space number, and pushes memory location which includes the address space
  - Each address space is a separate memory location storage
  - All operations on locations work with memory locations regardless of address space
  - Every architecture defines address space 0 => default global memory address space
- Generalization avoids creating specialized operations to work with address spaces
The source variable is at byte 0x10 in the frame.

DW_OP_LLVM_offset => works the same with memory locations that have an address space.
Example: Variable in bit field of a register

- Locations specify an offset within associated location storage => extension allows bit offsets
  - DWARF 5 does not support general bit offset => only supports bit fields in composites with `DW_OP_bit_piece`
  - DWARF 5 only supports locations that start at the beginning of a register
- Supporting bit offsets benefits all targets
Extension: Bit Offsets (2 of 4)

**Expression**
- DW_OP_regx SGPR3
- DW_OP_uconst 20
- DW_OP_LLVM_bit_offset

**Stack**

<table>
<thead>
<tr>
<th>Register</th>
<th>Offset</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0 bits</td>
<td>SGPR3</td>
</tr>
</tbody>
</table>

**Context**
- Result: Location

**Source Program**
- Variable

**Hardware**
- SGPR 3
Extension: Bit Offsets (3 of 4)

- `DW_OP_uconst` => push bit offset on stack
- This could also be a runtime calculation
Extension: Bit Offsets (4 of 4)

- DW_OP_LLVM_bit_offset => pop value and location, update location’s offset using value as a bit offset, push updated location
  - Bit ordering, like byte ordering, is architecture specific
  - Base type’s ordering can specify both byte and bit ordering
  - Works on any location kind
- Locations with bit offsets allowed in composite location parts just like any other location
Other benefits of generalizing locations on the stack

- DWARF 5 only supports memory locations on the stack => uses global memory address:
  - DW_AT_data_member_location => evaluates expression with type instance object address as initial stack value
  - DW_OP_push_object_address => pushes location of context’s program object defined by the attribute
  - DW_OP_call* operations => values can be passed in/out to called DWARF procedure on stack

- Generalization allows any location kind
  - Necessary to support optimized code on GPUs => compiler allocates objects in registers, different address spaces, and composites of them
  - Allows bit fields and implicit locations to be supported => can occur through optimization on any target

- GPU compiler uses DWARF procedures to factorize location expressions => SIMT divergent control flow information
  - Reduces DWARF size
  - More convenient to generate
Call Frame Information (CFI)

- DWARF defines call frame information (CFI) => used to virtually unwind call stack
- Extended CFI rules to support:
  - All location kinds
  - Address spaces
- GPU only saves active lanes of VGPR callee saved registers
  - DW_OP_LLVM_select_bit_piece => used by unwind expressions to inspect the bits in EXEC register
  - DW_OP_LLVM_call_frame_entry_reg => used to get EXEC register value on entry to function
Multiple Places

- DWARF 5 supports loclists => can specify a location is in multiple places at same time
- DW_OP_call* and DW_OP_implicit_pointer => can specify DIE that has a loclist
- Location extended to allow one or more single locations
- Location operations extended to act on multiple places
  - DW_OP_LLVM_offset => adjusts offset of all the single locations
- DWARF 5 defines operation expressions and loclist expressions separately
  - Works in DWARF 5 as locations can only be the last step of an expression
  - Extension generalizations made unification fall out naturally =>
    unification necessary as locations now allowed at any step of an expression
Extension Operation Summary

Core Extensions

• Expression operations:
  • DW_OP_LLVM_form_aspace_address
  • DW_OP_LLVM_push_lane
  • DW_OP_LLVM_offset
  • DW_OP_LLVM_offset_uconst
  • DW_OP_LLVM_bit_offset
  • DW_OP_LLVM_call_frame_entry_reg
  • DW_OP_LLVM_undefined
  • DW_OP_LLVM_aspace_bregx
  • DW_OP_LLVM_aspace_implicit_pointer
  • DW_OP_LLVM_piece_end
  • DW_OP_LLVM_extend
  • DW_OP_LLVM_select_bit_piece

• CFI operations:
  • DW_CFA_LLVM_def_aspace_cfa
  • DW_CFA_LLVM_def_aspace_cfa_sf

• DIE Attributes:
  • DW_AT_LLVM_active_lane
  • DW_AT_LLVM_lanes
  • DW_AT_LLVM_lane_pc

Divergent Lane Support Extensions

• DIE Attributes:
  • DW_AT_LLVM_vector_size
Current Progress

- Ongoing development:
  - AMD ROCm ROCgdb debugger
    https://github.com/ROCM-Developer-Tools/ROCgdb

- In development:
  - AMD ROCm LLVM compiler
  - Perforce TotalView debugger
  - Mentor Graphics GCC compiler

- Further Information:
  - DWARF Extensions For Heterogeneous Debugging
    https://llvm.org/docs/AMDGPUDwarfExtensionsForHeterogeneousDebugging.html
  - User Guide for AMDGPU Backend: DWARF Debug Information
    https://llvm.org/docs/AMDGPUUsage.html#dwarf-debug-information
Acknowledgements

- Would like to thank the following for collaboration, feedback and support:
  - AMD ROCm Debugger Team: Tony Tye, Laurent Morichetti, Zoran Zaric
  - AMD ROCm Compiler Team: Scott Linder, Konstantin Zhuravlyov, Ram Nalamothu, Brian Sumner
  - ARM: Louise Spellecy, Richard Brunt, Dirk Schubert
  - GDB Maintainers: Pedro Alves, Simon Marchi
  - HPE: Andrew Gontarek, Deepak Eachempati, John Vogt, Jeff Sandoval
  - Intel: Markus Metzer
  - Lawrence Livermore National Laboratory: Dong Ahn
  - Mentor Graphics: Andrew Stubbs
  - Oak Ridge National Laboratory
  - Perforce: John DelSignore, Steve Lawrence
Summary

- DWARF expressions are generalized to allow locations on the stack
- New operators that are composable, consistent, and backward compatible
- Provides support needed by GPUs and other heterogeneous devices
- Improves debugging of optimized code for both CPUs and GPUs
DISCLAIMER AND ATTRIBUTIONS

Disclaimer
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. 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 NONINFRINGEMENT, 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 CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

©2021 Advanced Micro Devices, Inc. All rights reserved.

AMD®, the AMD Arrow logo, AMD Instinct®, Radeon®, ROCm®, and combinations thereof are trademarks of Advanced Micro Devices, Inc. ARM® is a registered trademark of ARM Limited in the EU and other countries. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.