DWARF Extensions for Optimized SIMT/SIMD (GPU) Debugging

GNU Tools Cauldron @ Linux Plumbers Conference 2021

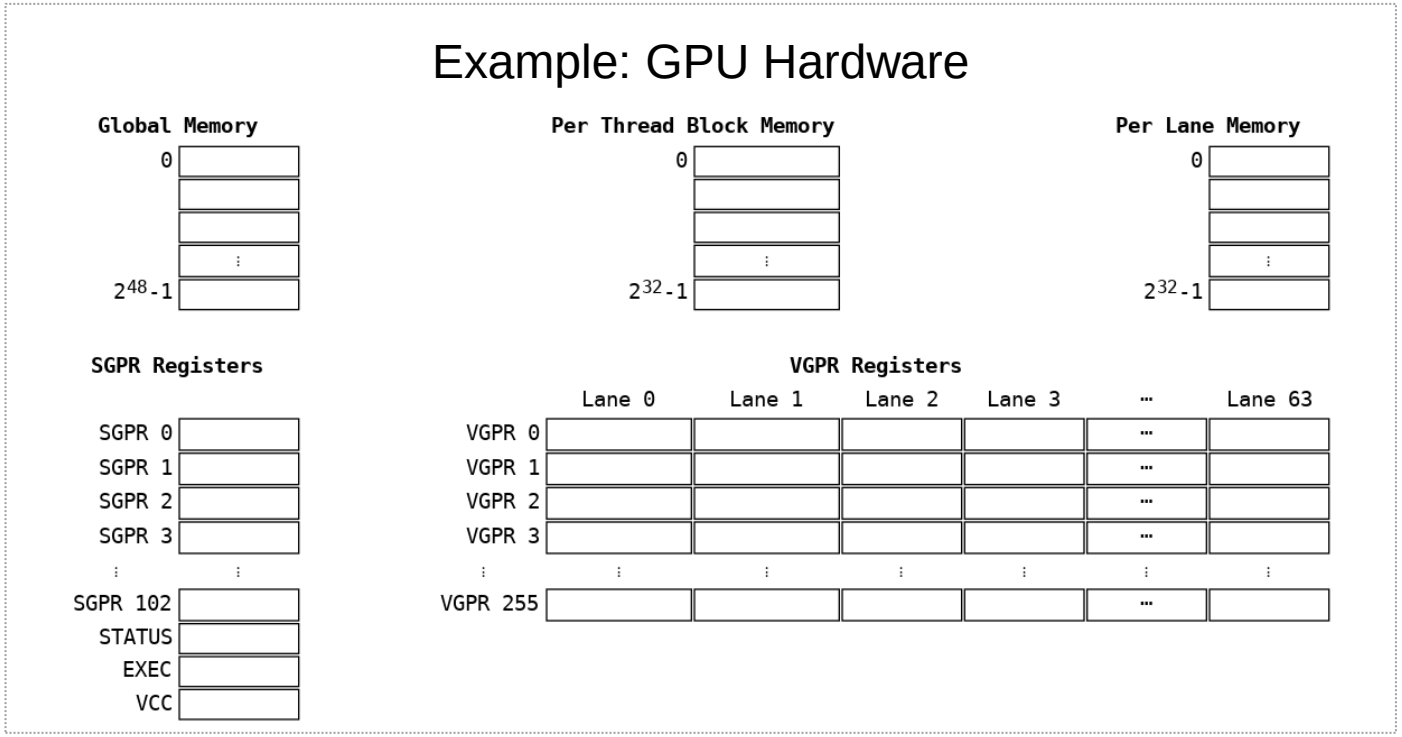
Tony Tye, Scott Linder, Zoran Zaric

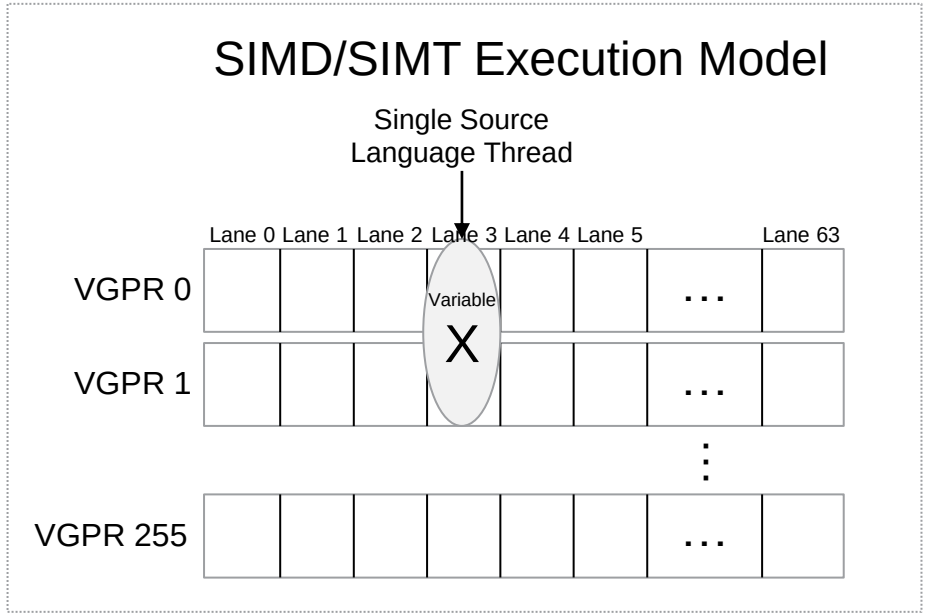
AMD Open Source Debugging Tools Project

- 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





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

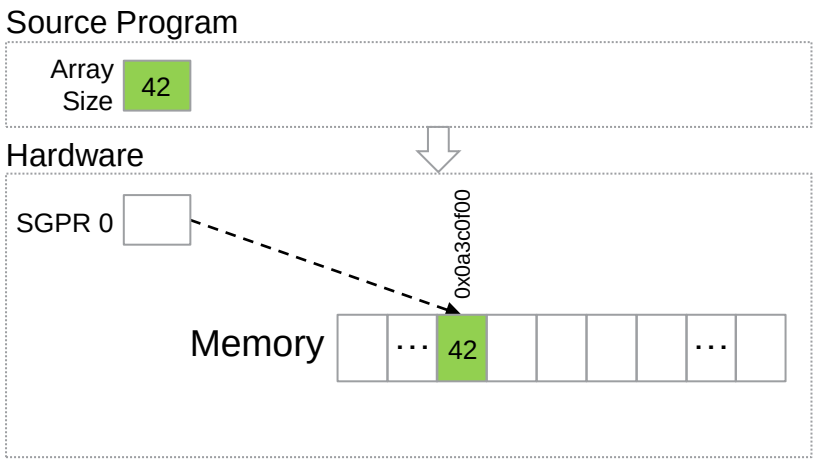
- Great diversity in locations of entities
- Locations involve runtime values
- Variable location could be:
  - In register
  - At memory address
  - At Stack Pointer + offset
  - Optimized away known value => such as compile time value
  - Optimized away unknown value => such as unused variables
  - 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)

	Expression	Stack	Context
DWARF PC →	`DW_OP_regval_type SGPR0 Generic` `DW_OP_deref`		**Result:** *Value*

Example: Runtime size of a dynamic array

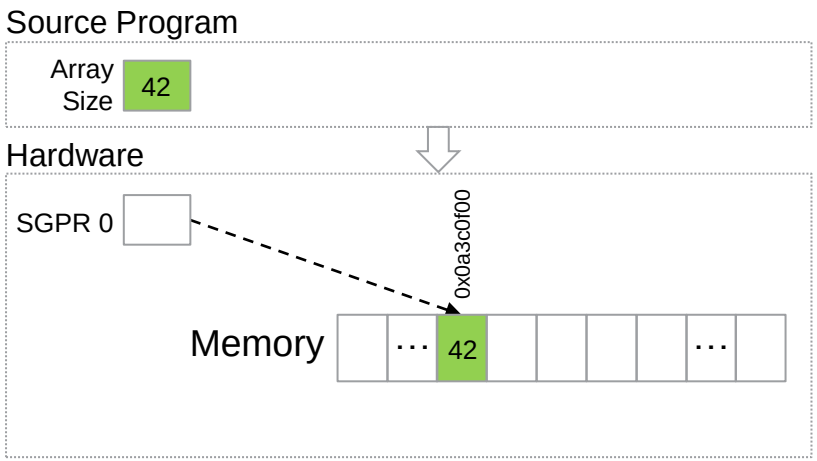
- Context result kind => value
- Expression evaluated one operation at a time using a stack



DWARF 5: Dynamic Array Size (2 of 3)

Expression	Stack	Context
DWARF PC → `DW_OP_regval_type SGPR0 Generic` `DW_OP_deref`	**Value** *Type*: Generic *Value*: 0x0a3c0f00 *Top*	**Result:** *Value*

- `DW_OP_regvalue_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)

Expression

```
DW_OP_regval_type SGPR0 Generic
DW_OP_deref
```

DWARF PC →

Stack

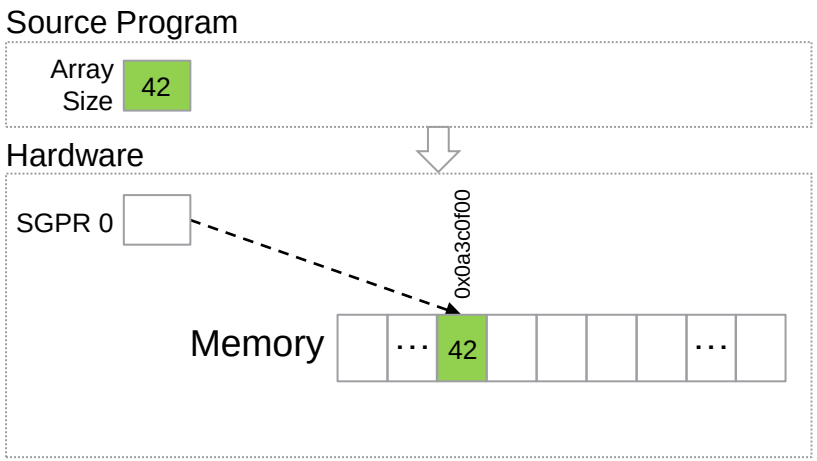
Value	
Type	*Value*
Generic	42

Top

Context

Result: *Value*

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



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

	Expression	Stack	Location	Context
DWARF PC →	`DW_OP_regx SGPR0`			**Result:** *Location*

Example: Source variable located in a register

- Context result kind => location
  - *Note: DWARF uses term location description*

DWARF 5: Variable Location in Register (2 of 2)

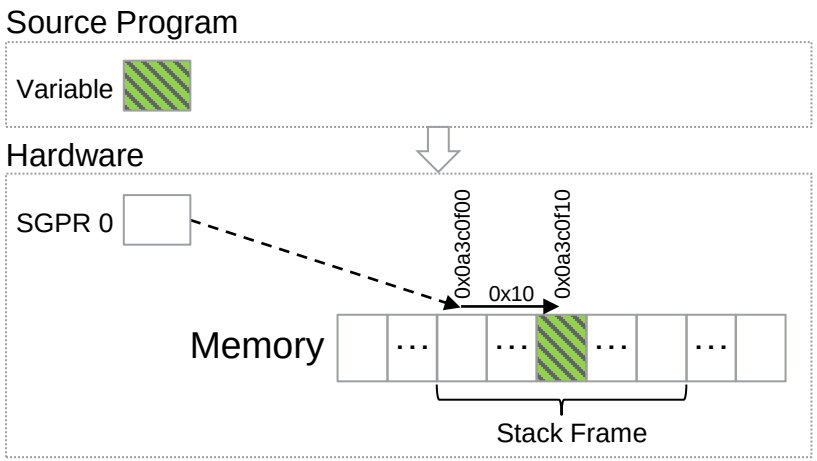
Expression	Stack	Location	Context
DWARF PC → `DW_OP_regx SGPR0`		**Register** *Number* SGPR0	**Result:** *Location*

- `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

DWARF 5: Variable Location in Memory (1 of 4)

	Expression	Stack	Location	Context
DWARF PC →	`DW_OP_regval_type SGPR0 Generic` `DW_OP_plus_uconst 0x10`			**Result:** *Location*

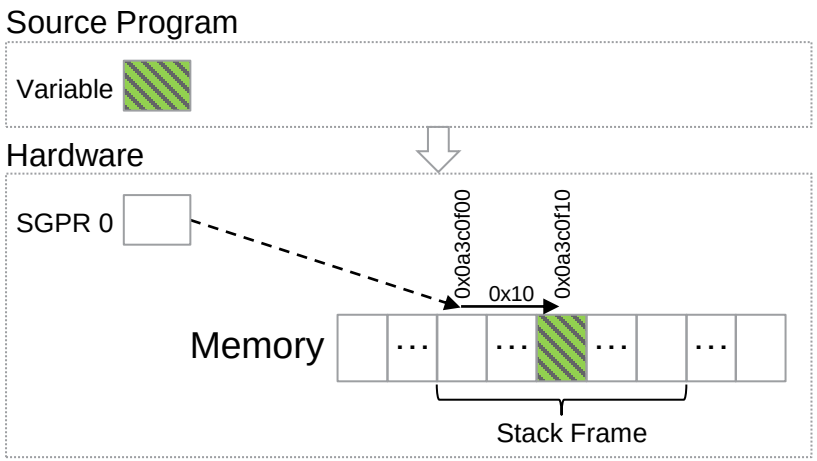
Example: Source variable in stack frame memory at address stack pointer + 0x10



DWARF 5: Variable Location Memory (2 of 4)

Expression	Stack	Location	Context
DWARF PC → `DW_OP_regval_type SGPR0 Generic` `DW_OP_plus_uconst 0x10`	**Value** *Type* | *Value* Generic | 0x0a3c0f00 *Top*		**Result:** *Location*

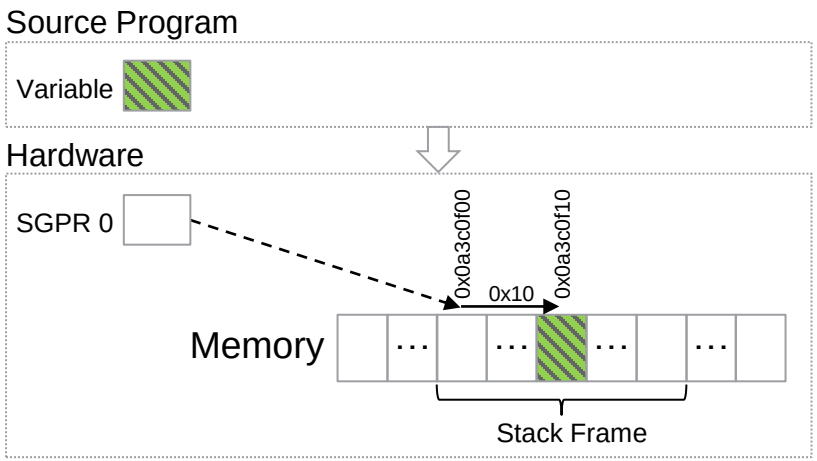
- `DW_OP_regvalue_type` => pushes value read from stack pointer register



DWARF 5: Variable Location Memory (3 of 4)

Expression	Stack	Location	Context
DWARF PC → `DW_OP_regval_type SGPR0 Generic` `DW_OP_plus_uconst 0x10`	**Value** *Type*: Generic *Value*: 0x0a3c0f10 *Top*		**Result:** *Location*

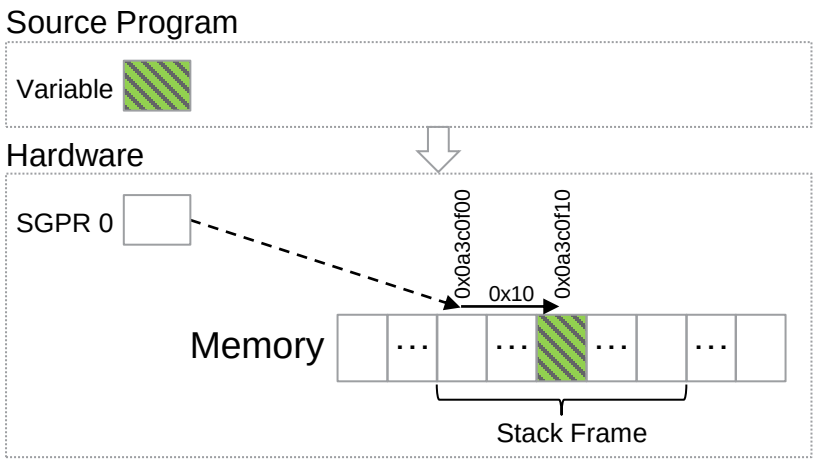
- Variable location is 0x10 bytes from the base of the stack frame
- `DW_OP_plus_uconst` => pop value, add 0x10, and push result



DWARF 5: Variable Location Memory (4 of 4)

Expression	Stack	Location	Context
DWARF PC → `DW_OP_regval_type SGPR0 Generic` `DW_OP_plus_uconst 0x10`		**Memory** *Address* `0x0a3c0f10`	**Result:** *Location*

- Location area empty when location result kind => convert top stack element to memory location
  - Use value as global memory address



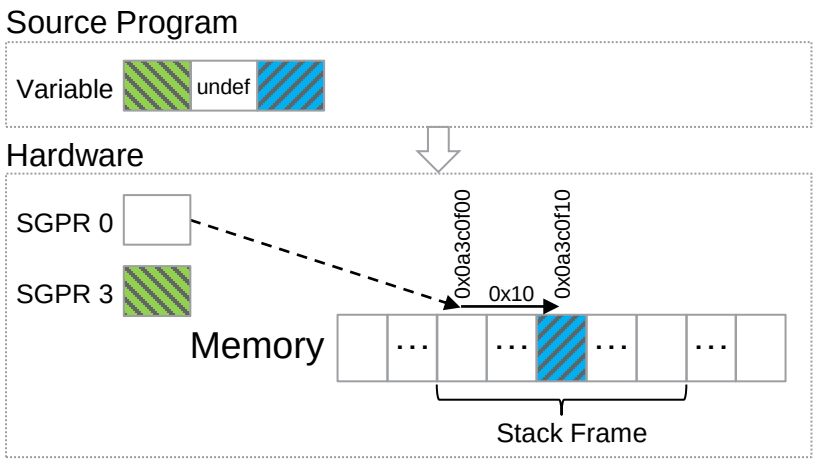
DWARF 5: Variable Spread Across Different Locations (1 of 7)

	Expression	Stack	Location	Context

DWARF PC →	`DW_OP_regx SGPR3`
	`DW_OP_piece 4`
	`DW_OP_piece 2`
	`DW_OP_bregx SGPR0 0x10`
	`DW_OP_piece 2`
				Result: *Location*

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

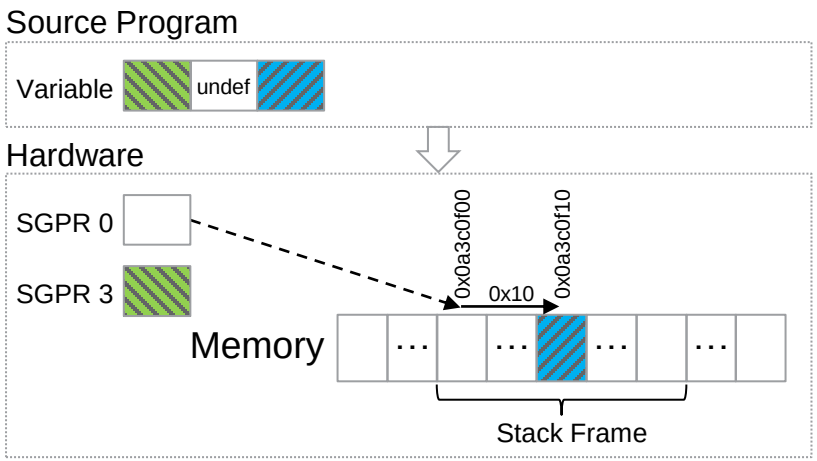


DWARF 5: Variable Spread Across Different Locations (2 of 7)

Expression	Stack	Location	Context

DWARF PC → `DW_OP_regx SGPR3`
`DW_OP_piece 4`
`DW_OP_piece 2`
`DW_OP_bregx SGPR0 0x10`
`DW_OP_piece 2`
		Register *Number*
		SPGR3
			Result: *Location*

- `DW_OP_regx` => creates register location

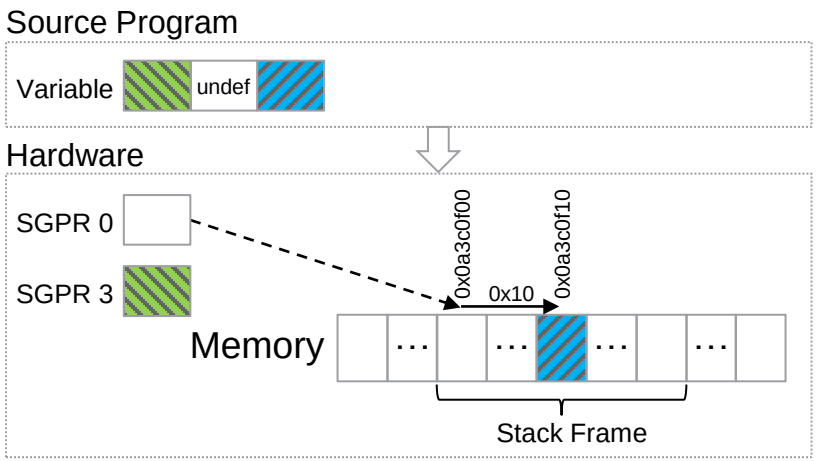


DWARF 5: Variable Spread Across Different Locations (3 of 7)

Expression	Stack	Location	Context

DWARF PC → `DW_OP_regx SGPR3`
`DW_OP_piece 4`
`DW_OP_piece 2`
`DW_OP_bregx SGPR0 0x10`
`DW_OP_piece 2`
		Composite *(incomplete)* *Parts*
		Size | *Location*
		4 | **Register** *Number* SPGR3
			Result: *Location*

- `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



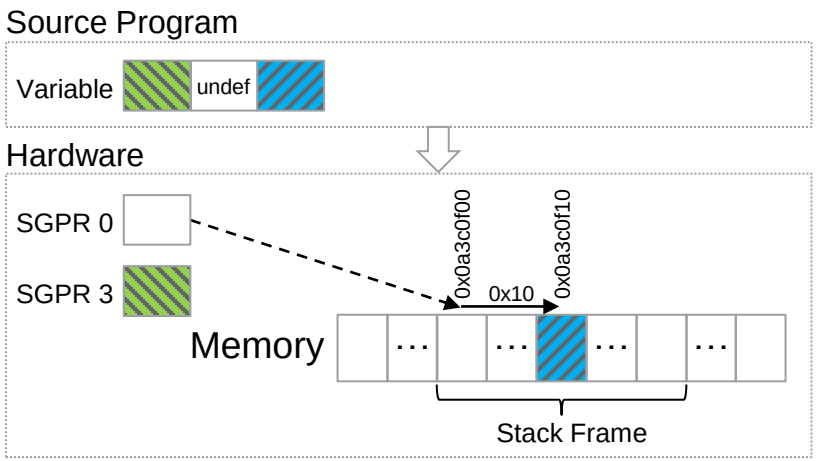
DWARF 5: Variable Spread Across Different Locations (4 of 7)

Expression	Stack	Location	Context

DWARF PC → `DW_OP_regx SGPR3`
`DW_OP_piece 4`
`DW_OP_piece 2`
`DW_OP_bregx SGPR0 0x10`
`DW_OP_piece 2`
		Composite *(incomplete)* *Parts*	**Result:** *Location*

Size	*Location*
4	**Register** *Number* SPGR3
2	**Undef**

- `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)

Expression

```
DW_OP_regx SGPR3
DW_OP_piece 4
DW_OP_piece 2
DW_OP_bregx SGPR0 0x10
DW_OP_piece 2
```

DWARF PC → DW_OP_bregx SGPR0 0x10

Stack

Value	
Type	*Value*
Generic	0x0a3c0f10

Top

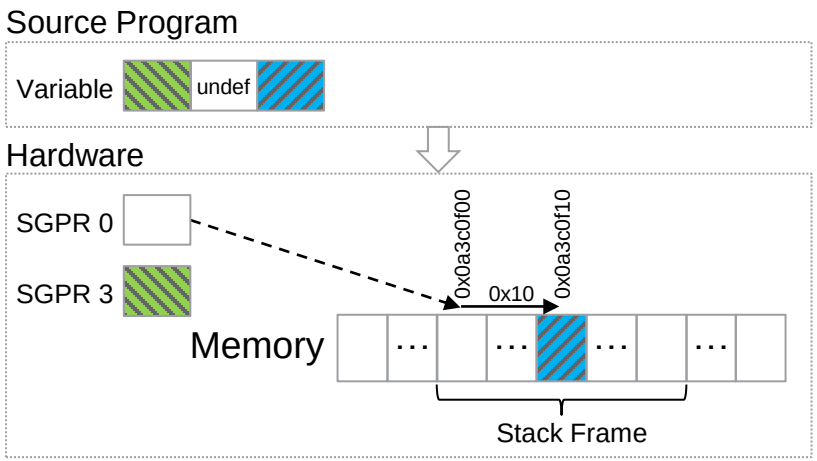
Location

Composite *(incomplete)*	
Parts	
Size	*Location*
4	**Register** *Number* SPGR3
2	**Undef**

Context

Result: *Location*

- `DW_OP_bregx` => read register as Generic type, add 0x10, and push value



DWARF 5: Variable Spread Across Different Locations (6 of 7)

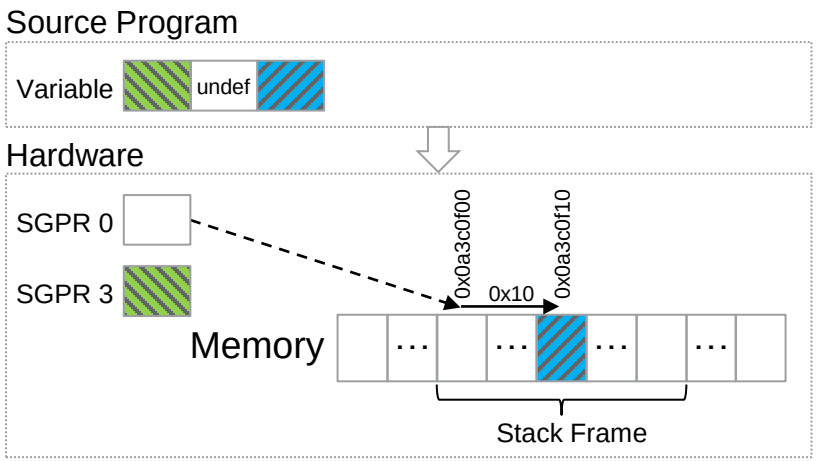
Expression	Stack	Location	Context
DW_OP_regx SGPR3 DW_OP_piece 4 DW_OP_piece 2 DW_OP_bregx SGPR0 0x10 DW_OP_piece 2			**Result:** *Location*

DWARF PC →

Composite *(incomplete)*

Size	*Parts* *Location*
4	**Register** *Number* SPGR3
2	**Undef**
2	**Memory** *Address* 0x0a3c0f10

- `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

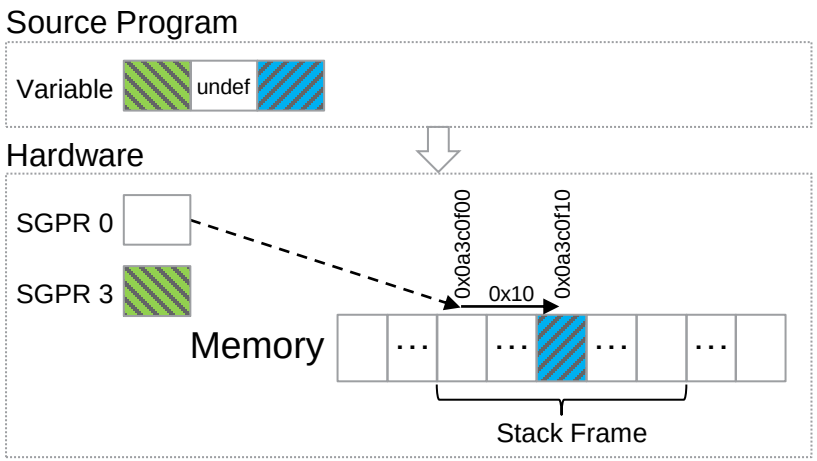


DWARF 5: Variable Spread Across Different Locations (7 of 7)

Expression	Stack	Location	Context
DW_OP_regx SGPR3 DW_OP_piece 4 DW_OP_piece 2 DW_OP_bregx SGPR0 0x10 DWARF PC → DW_OP_piece 2		**Composite** *Parts*	**Result:** *Location*

Size	*Location*
4	**Register** *Number* SPGR3
2	**Undef**
2	**Memory** *Address* 0x0a3c0f10

- Incomplete composite location implicitly converted to complete composite location



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

Expression	Stack	Location	Context

DWARF PC → `DW_OP_regx VGPR0`
`DW_OP_piece 4`
`DW_OP_piece 2`
`DW_OP_bregx SGPR0 0x10`
`DW_OP_piece 2`
`DW_OP_plus_uconst 5`
		Composite *(incomplete)* *Parts*	**Result:** *Location*

Size	*Location*
4	**Register** *Number* VGPR0
2	**Undef**
2	**Memory** *Address* 0x0a3c0f10

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 (2 of 2)

Expression	Stack	Location	Context

`DW_OP_regx VGPR0`
`DW_OP_piece 4`
`DW_OP_piece 2`
`DW_OP_bregx SGPR0 0x10`
`DW_OP_piece 2`
`DW_OP_plus_uconst 5`
			Result: *Location*

DWARF PC →

Invalid Expression:
DW_OP_plus_uconst only
operates on values.

- 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

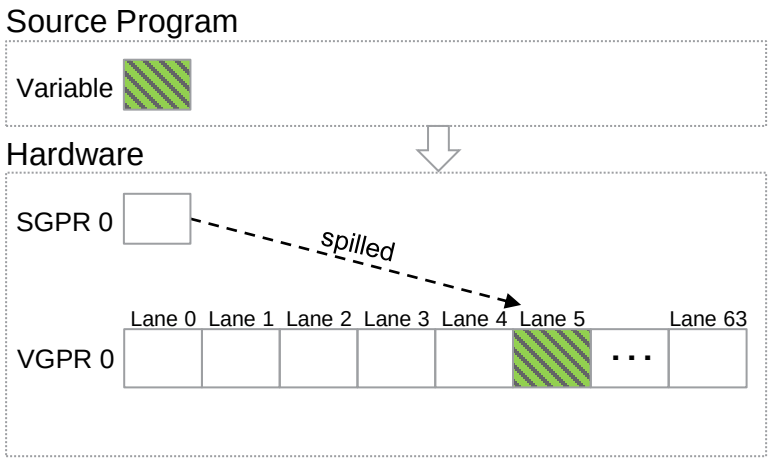
- If location could be allowed on the stack:
  - Define new operations to work on locations in compossible manner
  - Example: new `DW_OP_LLVM_offset` => updates offset of any location kind
- Existing operations can be generalized => act on locations of any kind
  - Example: `DW_OP_deref` => pop a location (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)

	Expression	Stack	Context
DWARF PC →	`DW_OP_regx VGPR0` `DW_OP_LLVM_offset_uconst 20`		**Result:** *Location*

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)

Expression

DWARF PC →

```
DW_OP_regx VGPR0
DW_OP_LLVM_offset_uconst 20
```

Stack

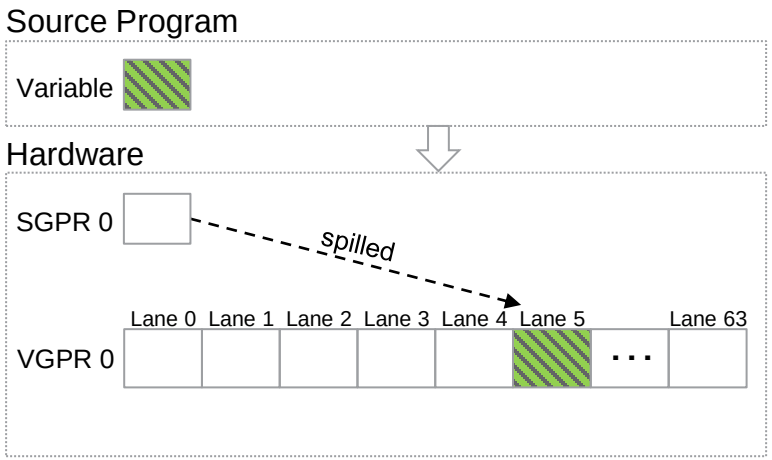
Register	
Offset	*Number*
0	VGPR0

Top

Context

Result: *Location*

- `DW_OP_regx` => now pushes location on stack with byte offset of 0



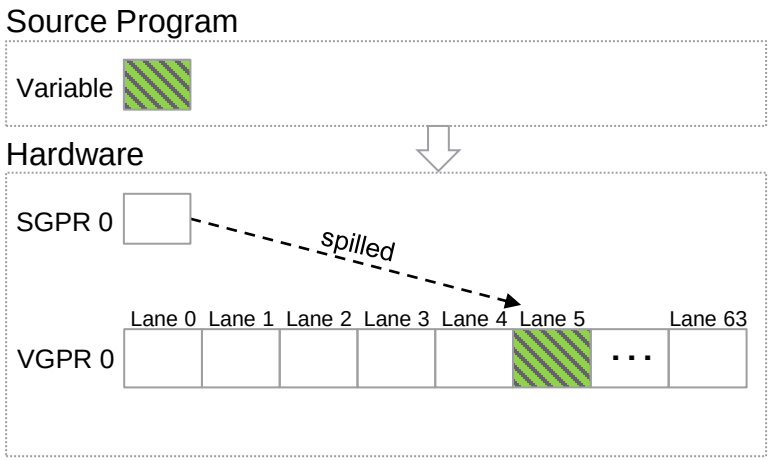
Extension: Variable spilled to part of VGPR (3 of 3)

Expression	Stack	Context
DW_OP_regx VGPR0 DW_OP_LLVM_offset_uconst 20	Register Offset: 20 Number: VGPR0 Top	**Result:** *Location*

DWARF PC →

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

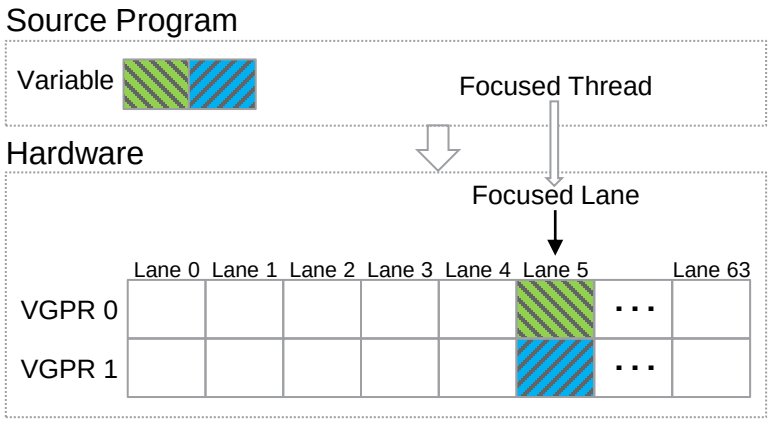


Extension: Variable across multiple VGPRs (1 of 14)

	Expression	Stack	Context
DWARF PC →	`DW_OP_regx VGPR0`		**Result:***Location*
	`DW_OP_LLVM_push_lane`		**Lane:***5*
	`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`		

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)

Expression

DWARF PC →

```
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
```

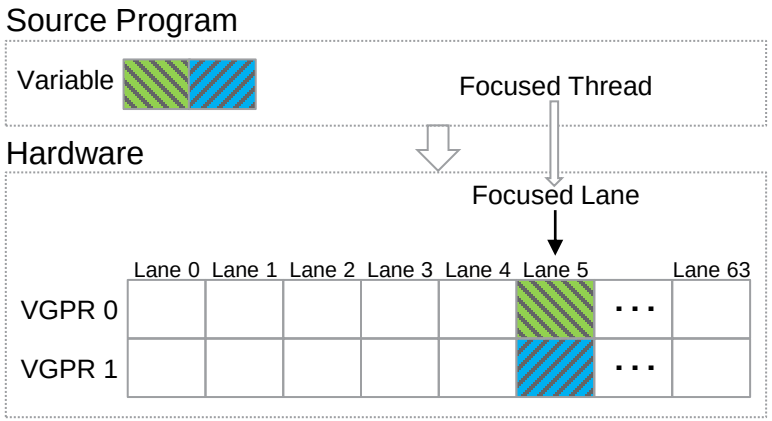
Stack

Register	
Offset	*Number*
0	VGPR0

Top

Context

Result: *Location*
Lane: *5*



Extension: Variable across multiple VGPRs (3 of 14)

Expression

DWARF PC →

```
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

Register	
Offset	*Number*
0	VGPR0

Value	
Type	*Value*
Generic	5

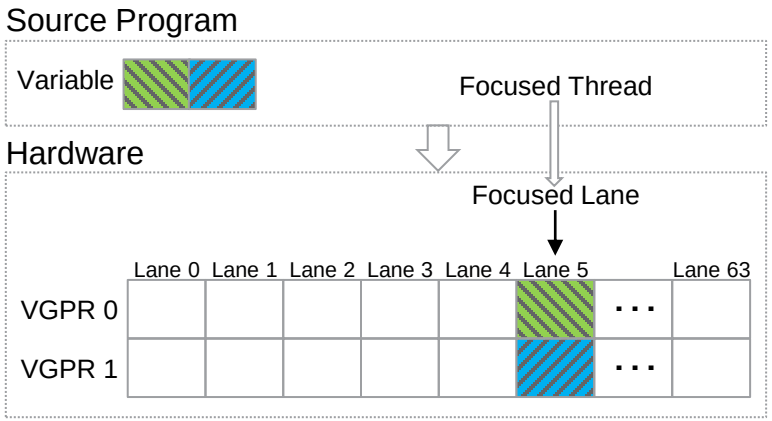
Top

Context

Result: *Location*
Lane: *5*

New

- `DW_OP_LLVM_push_lane` => pushes value of context's lane



Extension: Variable across multiple VGPRs (4 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

Register	
Offset	*Number*
0	VGPR0

Value	
Type	*Value*
Generic	5

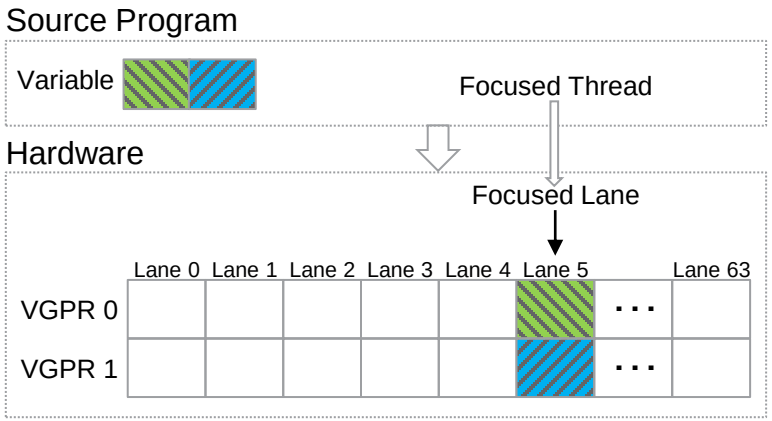
Value	
Type	*Value*
Generic	4

Top

Context

Result: *Location*
Lane: *5*

- 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)

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 → DW_OP_LLVM_offset

Stack

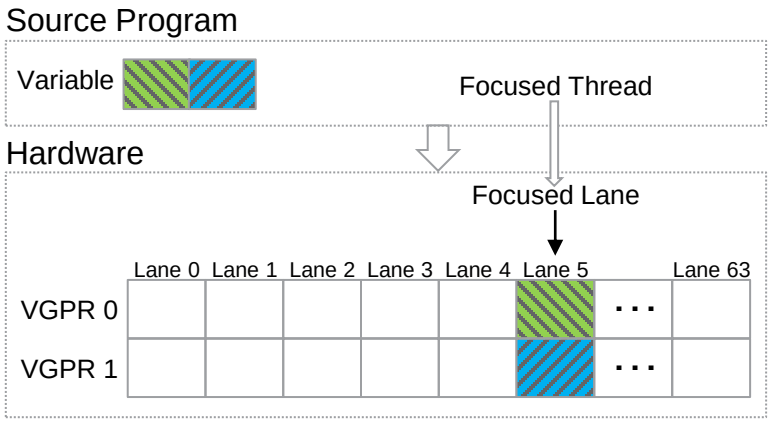
Register	
Offset	*Number*
0	VGPR0

Value	
Type	*Value*
Generic	20

Top

Context

Result: *Location*
Lane: *5*



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
```

DWARF PC → `DW_OP_LLVM_offset`

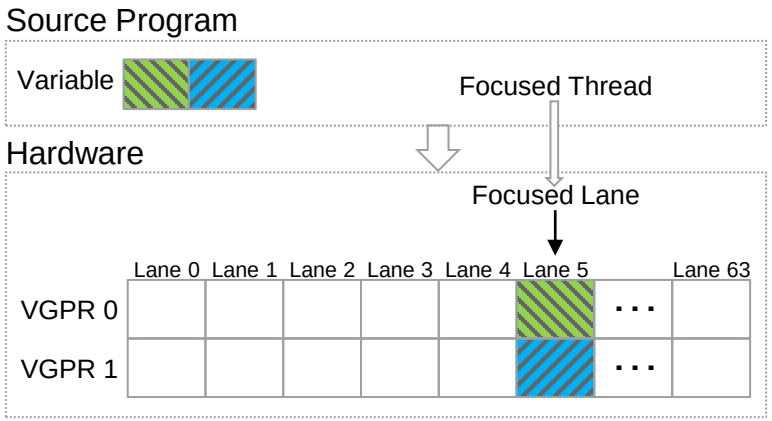
Stack

Register	
Offset	*Number*
20	VGPR0

Top

Context

Result: *Location*
Lane: *5*



Extension: Variable across multiple VGPRs (7 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 → DW_OP_regx VGPR1

Stack

Composite (incomplete)		
Offset	Parts	
0	Size	Location
	4	Register
		Offset: 20 / Number: VGPR0

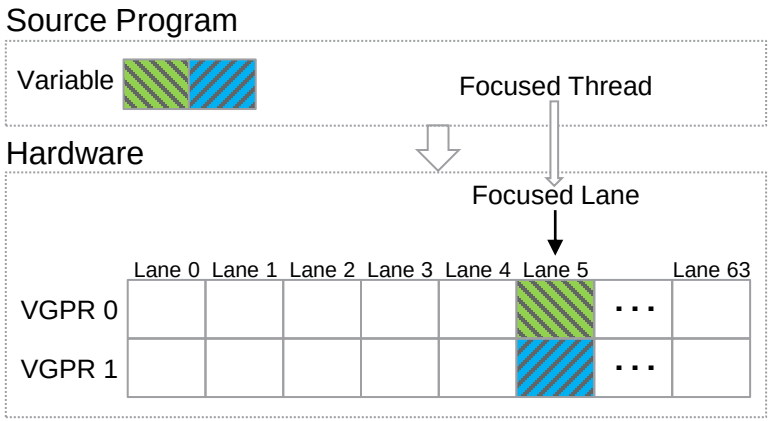
Top

Context

Result: *Location*
Lane: *5*

- `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



Extension: Variable across multiple VGPRs (8 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 → DW_OP_regx VGPR1

Stack

Composite *(incomplete)*

Offset	*Parts*	
	Size	*Location*
0	4	**Register**

Offset	*Number*
20	VGPR0

Register

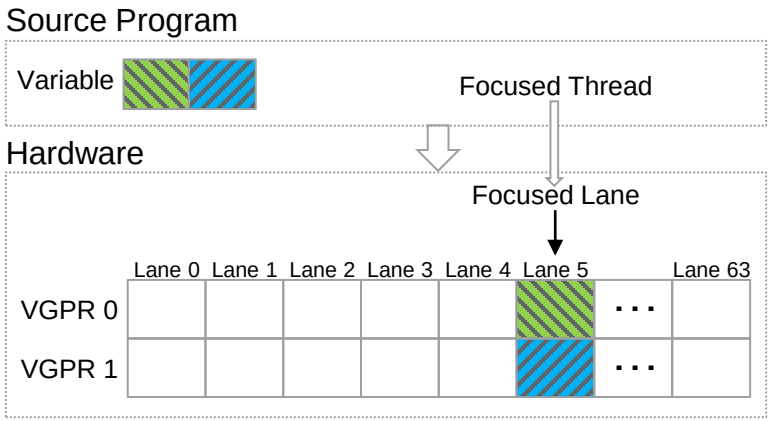
Offset	*Number*
0	VGPR1

Top

Context

Result: *Location*

Lane: *5*



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
```

DWARF PC → DW_OP_LLVM_push_lane

Stack

Composite *(incomplete)*

Offset	*Parts*	
0	*Size*	*Location*
	4	**Register**
		Offset / *Number*
		20 / VGPR0

Register

Offset	*Number*
0	VGPR1

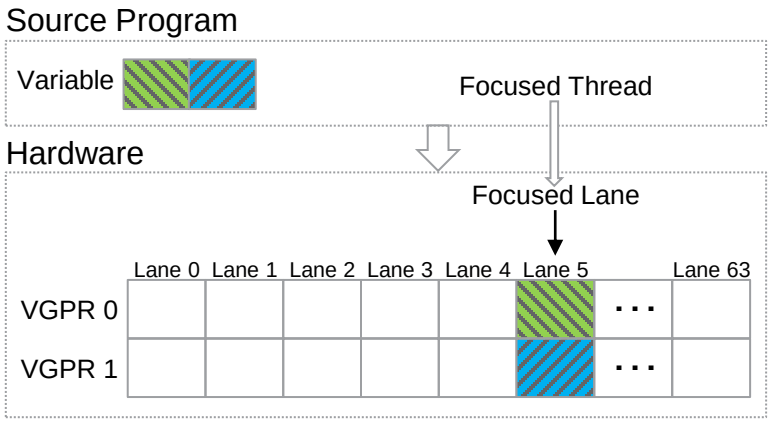
Value

Type	*Value*
Generic	5

Top

Context

Result: *Location*
Lane: *5*



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
```

DWARF PC →

Stack

Composite *(incomplete)*

Offset	*Parts*	
	Size	*Location*
0	4	**Register**

Offset	*Number*
20	VGPR0

Register

Offset	*Number*
0	VGPR1

Value

Type	*Value*
Generic	5

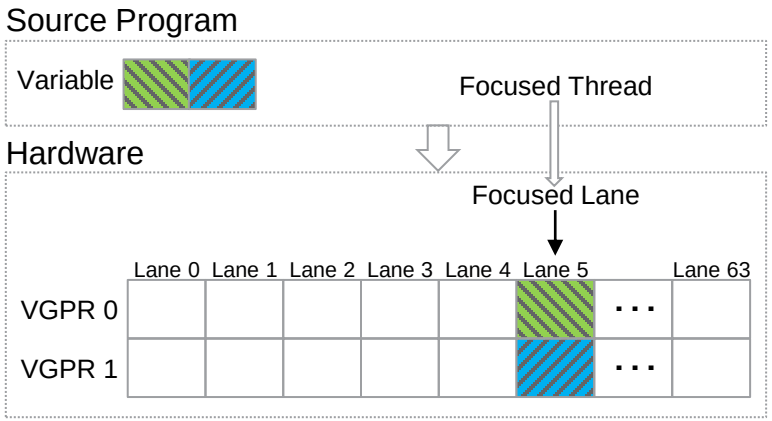
Value

Type	*Value*
Generic	4

Top

Context

Result: *Location*
Lane: *5*



Extension: Variable across multiple VGPRs (11 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 → DW_OP_mul

Stack

Composite *(incomplete)*

Offset	*Parts*	
0	*Size*	*Location*
	4	**Register**
		Offset / *Number*
		20 / VGPR0

Register

Offset	*Number*
0	VGPR1

Value

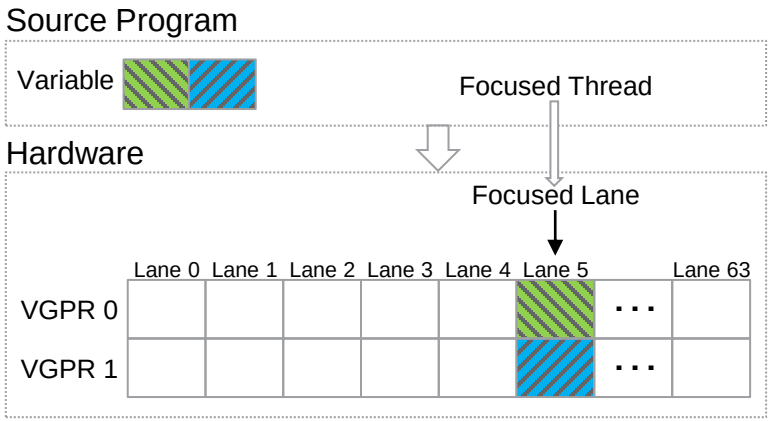
Type	*Value*
Generic	20

Top

Context

Result: *Location*

Lane: *5*



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 → DW_OP_LLVM_offset

Stack

Composite *(incomplete)*

Offset	*Parts*	
0	*Size*	*Location*
	4	**Register**
		Offset / *Number*
		20 / VGPR0

Register

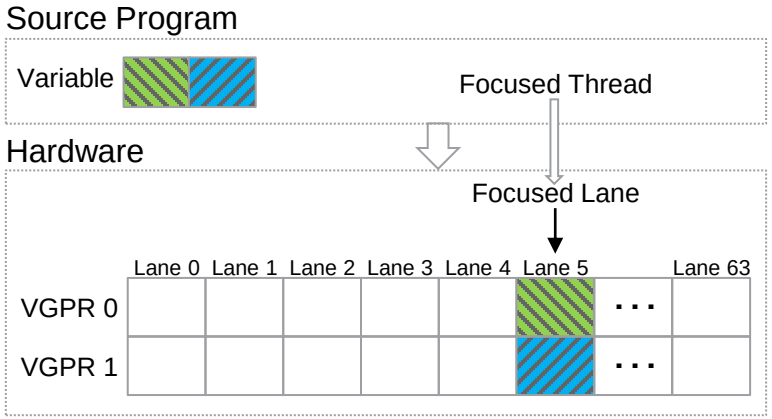
Offset	*Number*
20	VGPR1

Top

Context

Result: *Location*

Lane: *5*



Extension: Variable across multiple VGPRs (13 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

Composite *(incomplete)*

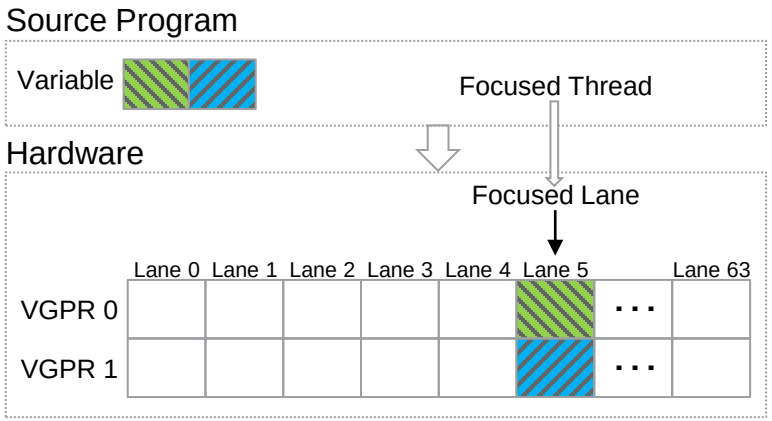
Offset	*Parts*	
0	*Size*	*Location*
	4	**Register** *Offset* 20, *Number* VGPR0
	4	**Register** *Offset* 20, *Number* VGPR1

Top

Context

Result: *Location*
Lane: *5*

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



Extension: Variable across multiple VGPRs (14 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

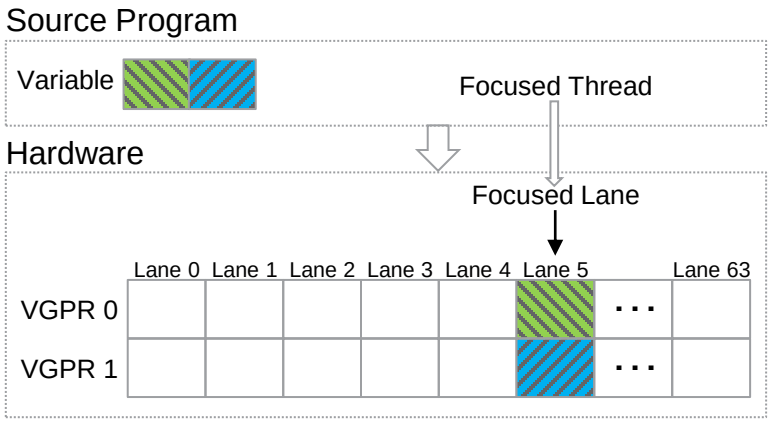
Composite		
Offset	*Parts*	
0	*Size*	*Location*
	4	**Register** *Offset*: 20, *Number*: VGPR0
	4	**Register** *Offset*: 20, *Number*: VGPR1

Top

Context

Result: *Location*
Lane: *5*

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



Extension: Variable across multiple kinds of locations (1 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
```

DWARF PC → `DW_OP_addr 0xbeef`

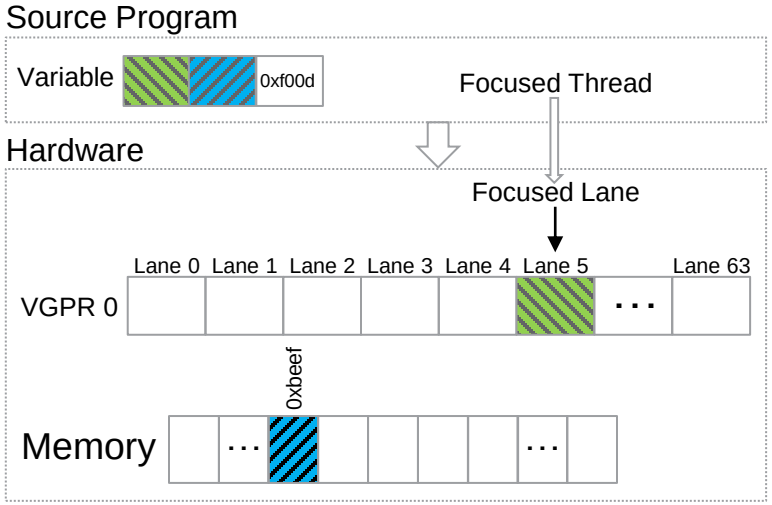
Stack

Composite *(incomplete)*		
Offset	*Parts*	
0	*Size*	*Location*
	4	**Register**
		Offset / *Number*
		20 / VGPR0

Top

Context

Result: *Location*
Lane: *5*



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)

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
```

DWARF PC → DW_OP_addr 0xbeef

Stack

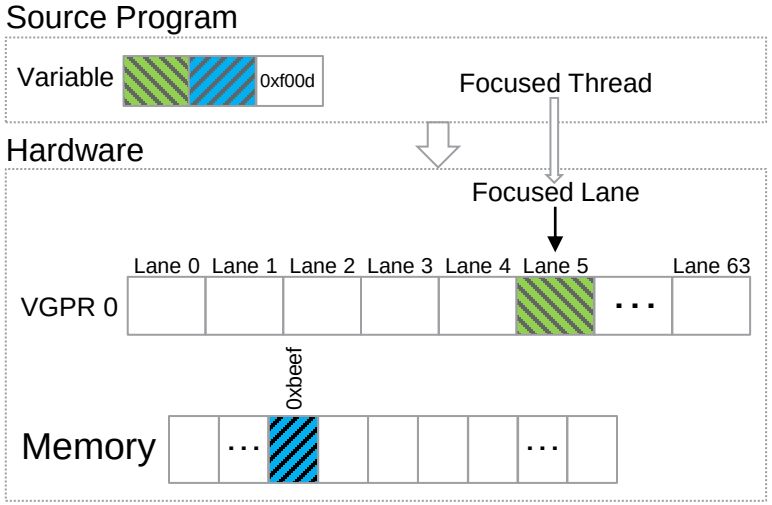
Composite *(incomplete)*		
Offset	*Parts*	
0	*Size*	*Location*
	4	**Register**
		Offset / *Number*
		20 / VGPR0

Memory
Offset
0xbeef

Top

Context

Result: *Location*
Lane: *5*



- 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
```

DWARF PC → `DW_OP_piece 2`

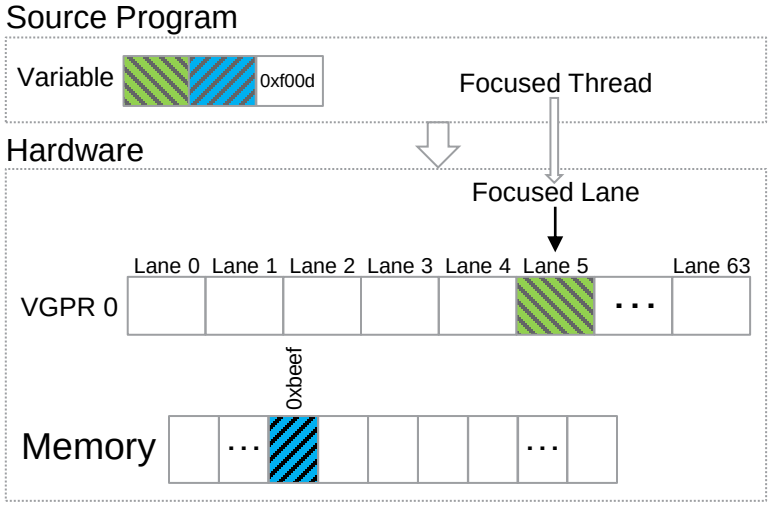
Stack

Composite *(incomplete)*		
Offset	*Parts*	
0	*Size*	*Location*
	4	**Register**
		Offset / *Number*
		20 / VGPR0
	2	**Memory**
		Offset
		0xbeef

Top

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)

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
```

DWARF PC → DW_OP_stack_value

Stack

Composite *(incomplete)*

Offset	*Parts*	
0	*Size*	*Location*
	4	**Register** *Offset* 20, *Number* VGPR0
	2	**Memory** *Offset* 0xbeef

Value

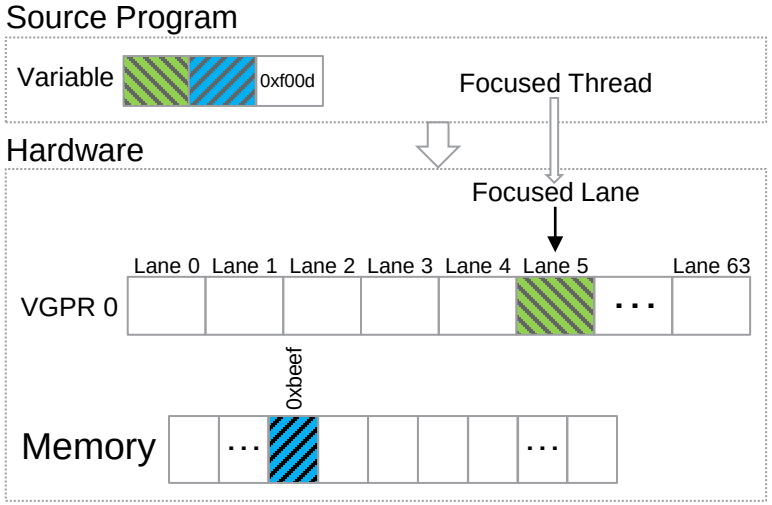
Type	*Value*
Generic	0xf00d

Top

Context

Result: *Location*

Lane: *5*



- Last 2 bytes are constant value `0xf00d`

Extension: Variable across multiple kinds of locations (5 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
```

DWARF PC → DW_OP_stack_value

Stack

Composite *(incomplete)*

Offset	*Parts*	
0	*Size*	*Location*
	4	**Register** *Offset* 20, *Number* VGPR0
	2	**Memory** *Offset* 0xbeef

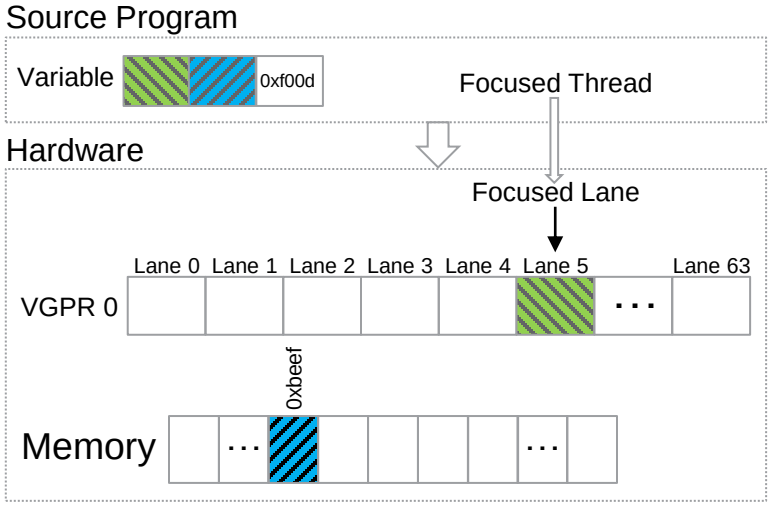
Implicit

Offset	*Value*
0	0xf00d

Top

Context

Result: *Location*
Lane: *5*



- `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

Extension: Variable across multiple kinds of locations (6 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
```

DWARF PC → DW_OP_piece 2

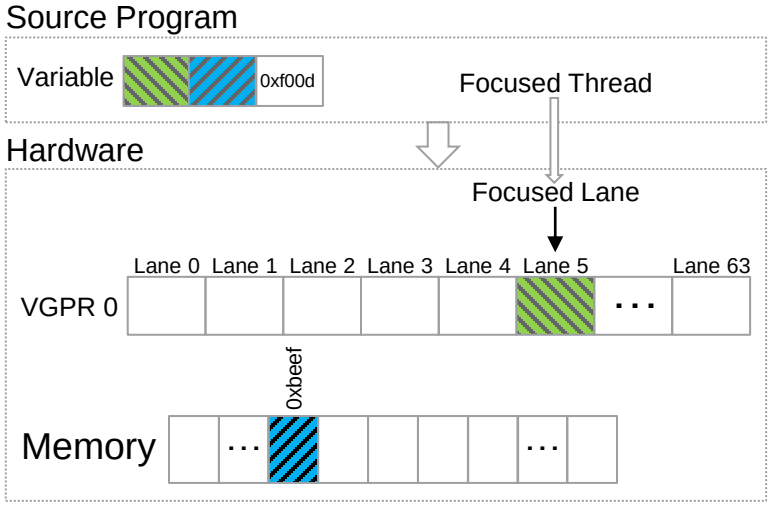
Stack

Composite *(incomplete)*		
Offset	*Parts*	
0	*Size*	*Location*
	4	**Register**
		Offset / *Number*
		20 / VGPR0
	2	**Memory**
		Offset
		0xbeef
	2	**Implicit**
		Offset / *Value*
		0 / 0xf00d

Top

Context

Result: *Location*
Lane: *5*



- `DW_OP_piece` => adds implicit location as next piece of composite location

Extension: Variable across multiple kinds of locations (7 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
```

DWARF PC

New

Stack

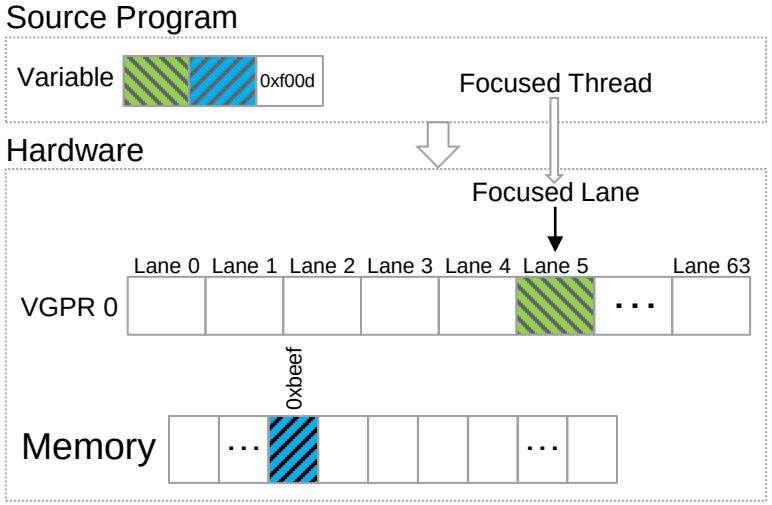
Composite		
Offset	*Parts*	
0	*Size*	*Location*
	4	**Register** *Offset* 20, *Number* VGPR0
	2	**Memory** *Offset* 0xbeef
	2	**Implicit** *Offset* 0, *Value* 0xf00d

Top

Context

Result: *Location*

Lane: *5*



- `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*`

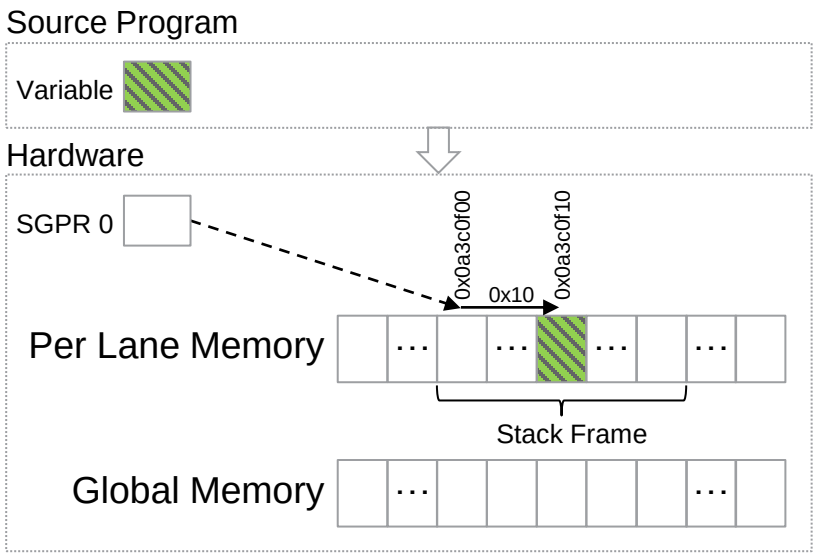
Extension: Address Spaces (1 of 5)

	Expression	Stack	Context

DWARF PC →	`DW_OP_regval_type SGPR0 Generic`
	`DW_OP_uconst 1`
	`DW_OP_LLVM_form_aspace_address`
	`DW_OP_LLVM_offset 0x10`
			Result: *Location*

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

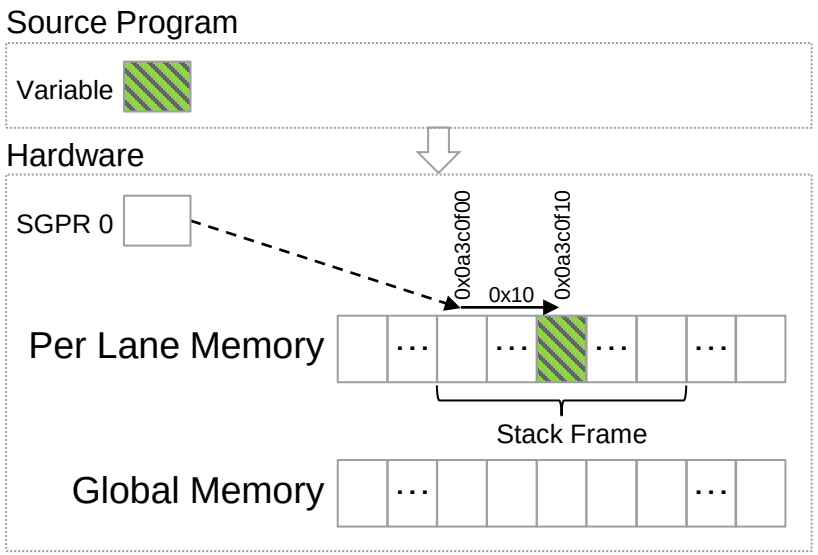


Extension: Address Spaces (2 of 5)

Expression	Stack	Context

DWARF PC → `DW_OP_regval_type SGPR0 Generic`
`DW_OP_uconst 1`
`DW_OP_LLVM_form_aspace_address`
`DW_OP_LLVM_offset 0x10`
	Value
	Type | *Value*
	Generic | 0x0a3c0f00
	Top
		Result: *Location*

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



Extension: Address Spaces (3 of 5)

Expression

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

DWARF PC → DW_OP_uconst 1

Stack

Value	
Type	*Value*
Generic	0x0a3c0f00

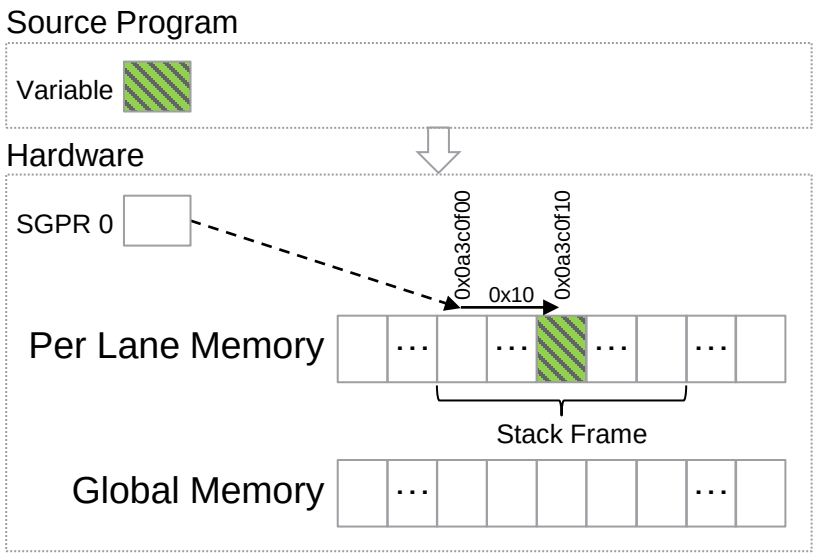
Value	
Type	*Value*
Generic	1

Top

Context

Result: *Location*

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



Extension: Address Spaces (4 of 5)

Expression

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

DWARF PC → DW_OP_LLVM_form_aspace_address

Stack

Memory	
Offset	*ASpace*
0x0a3c0f00	1

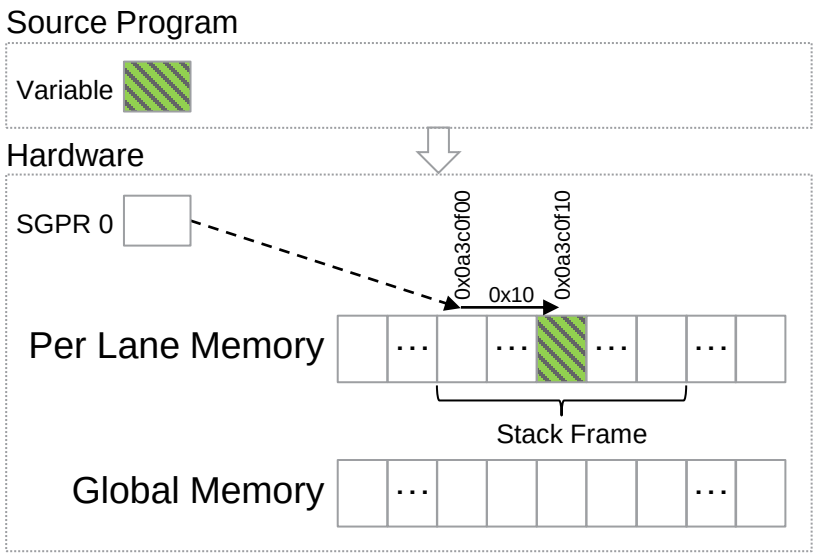
Top

Context

Result: *Location*

New

- `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



Extension: Address Spaces (5 of 5)

Expression

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

DWARF PC →

Stack

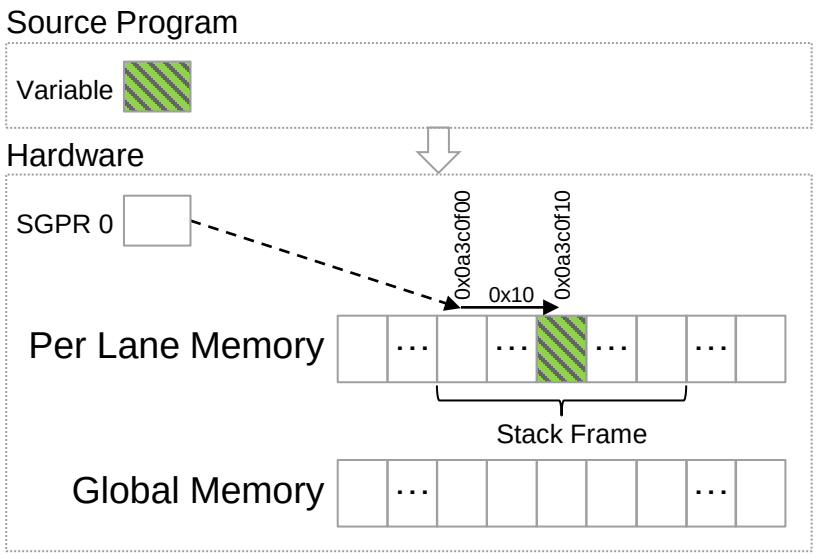
Memory	
Offset	*ASpace*
0x0a3c0f10	1

Top

Context

Result: *Location*

- 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



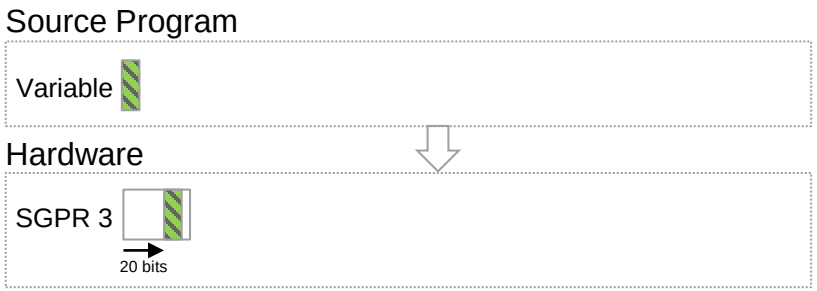
Extension: Bit Offsets (1 of 4)

	Expression	Stack	Context

DWARF PC →	`DW_OP_regx SGPR3`
	`DW_OP_uconst 20`
	`DW_OP_LLVM_bit_offset`
			Result: *Location*

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

DWARF PC →

```
DW_OP_regx SGPR3
DW_OP_uconst 20
DW_OP_LLVM_bit_offset
```

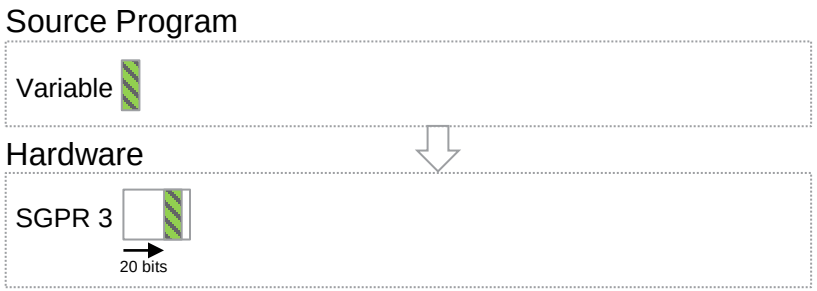
Stack

Register	
Offset	*Number*
0 bits	SGPR3

Top

Context

Result: *Location*



Extension: Bit Offsets (3 of 4)

Expression

```
DW_OP_regx SGPR3
DW_OP_uconst 20
DW_OP_LLVM_bit_offset
```

DWARF PC →

Stack

Register	
Offset	*Number*
0 bits	SGPR3

Value	
Type	*Value*
Generic	20

Top

Context

Result: *Location*

- `DW_OP_uconst` => push bit offset on stack
  - This could also be a runtime calculation

Extension: Bit Offsets (4 of 4)

Expression

```
DW_OP_regx SGPR3
DW_OP_uconst 20
DW_OP_LLVM_bit_offset
```

DWARF PC →

Stack

Register	
Offset	*Number*
20 bits	SGPR3

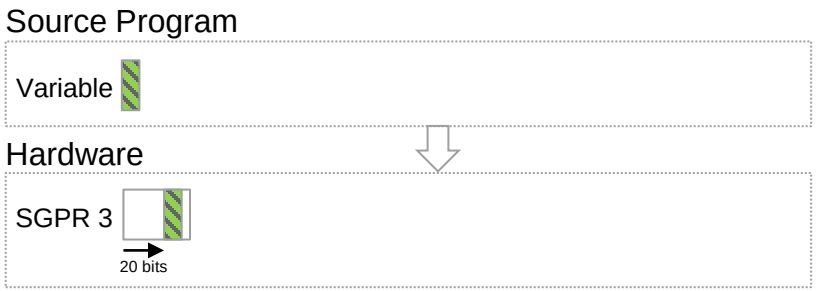
Top

Context

Result: *Location*

New

- `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
  - New `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_vector_size`

Divergent Lane Support Extensions

- DIE Attributes:
  - `DW_AT_LLVM_active_lane`
  - `DW_AT_LLVM_lanes`
  - `DW_AT_LLVM_lane_pc`

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