Handout 3

HSAIL and A SIMT GPU Simulator

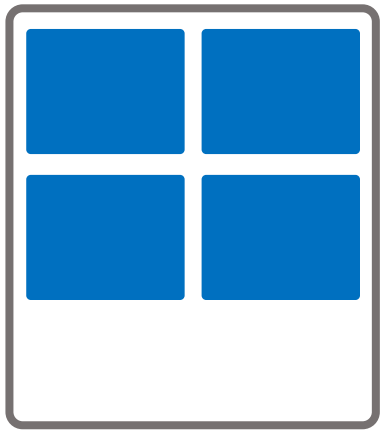
Outline

- Heterogeneous System
- Introduction of HSA Intermediate Language (HSAIL)
- A SIMT GPU Simulator
- Summary

Heterogeneous System

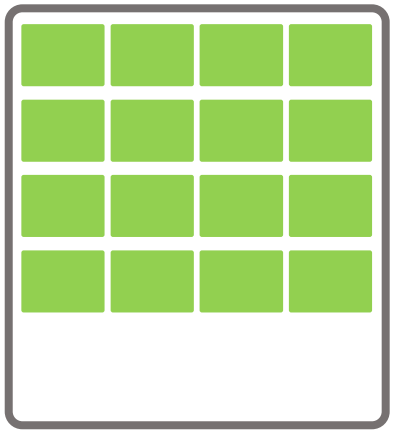
CPU & GPU

CPU



- CPU wants GPU's capability (SIMD)
- Sequential thread with limited data parallelism
- 8 ~ 16 cores

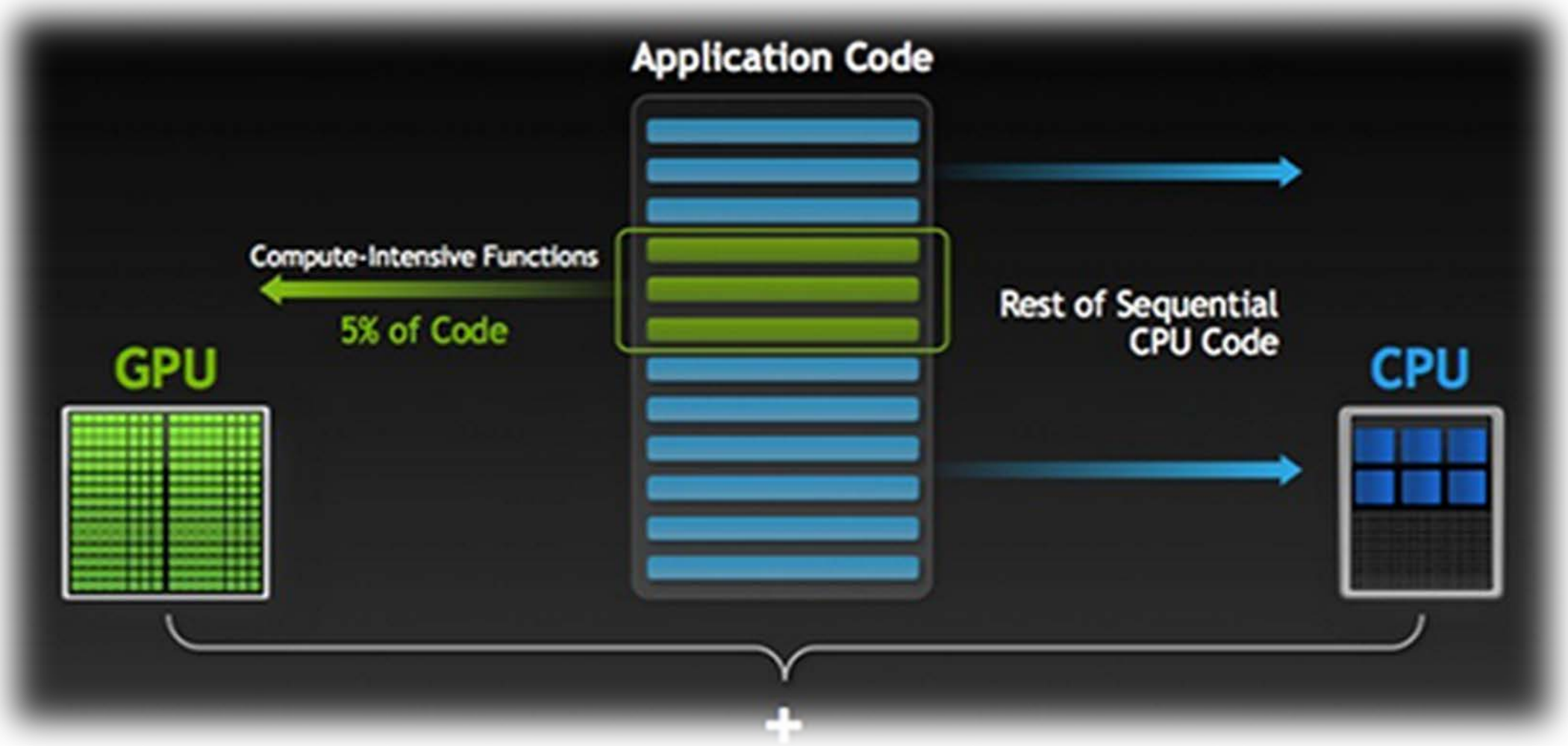
GPU



- GPU wants to have CPU's features (GPGPU)
- Data parallel processing
- 192 ~ cores (NV Kepler)

Heterogeneous Computing

Acceleration based on data parallelism

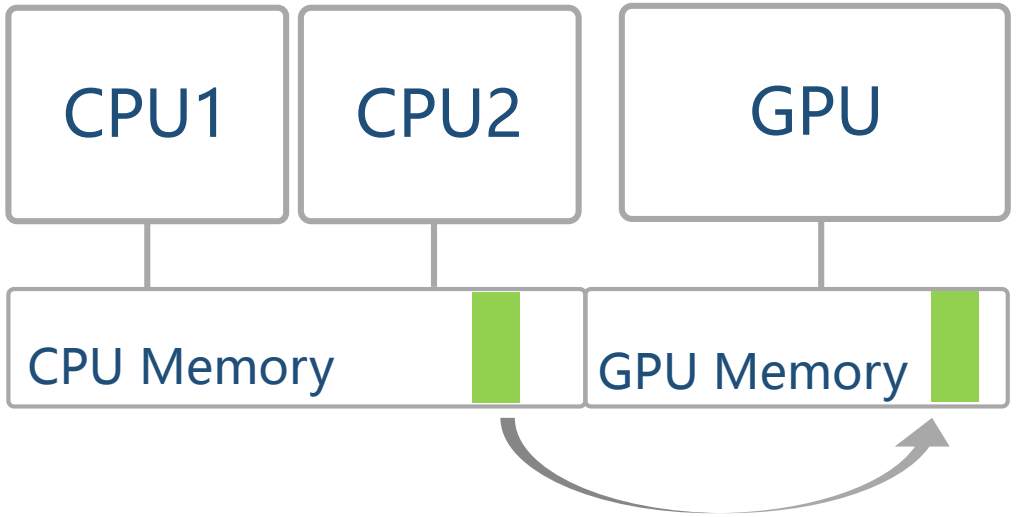


Shared Memory or not

Challenge of Traditional Heterogeneous Systems

Support only dedicated address space

- Require cumbersome copy operations
- Prevent the use of pointer
- DMA copy



Pointer addressable: load/store access

Heterogeneous Systems Architecture

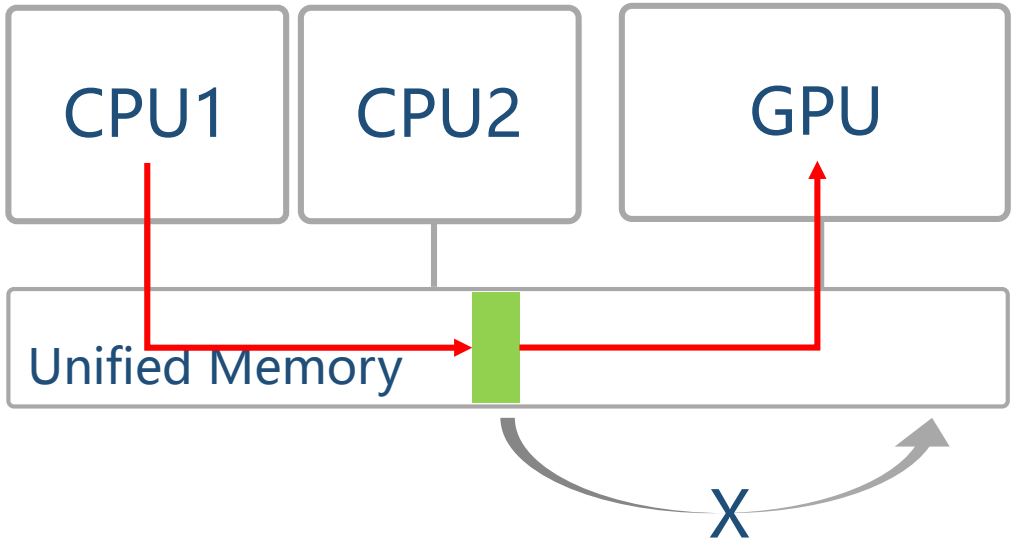
- Unified Memory Address Space (hMMU)

  Both CPU and GPU and access the memory directly.

  Remove the overhead due to memory copy.

- HSA Intermediate Language

  1. Enable Unified address space access
  2. Provide a unified intermediate language for high-level languages and different hardware ISA



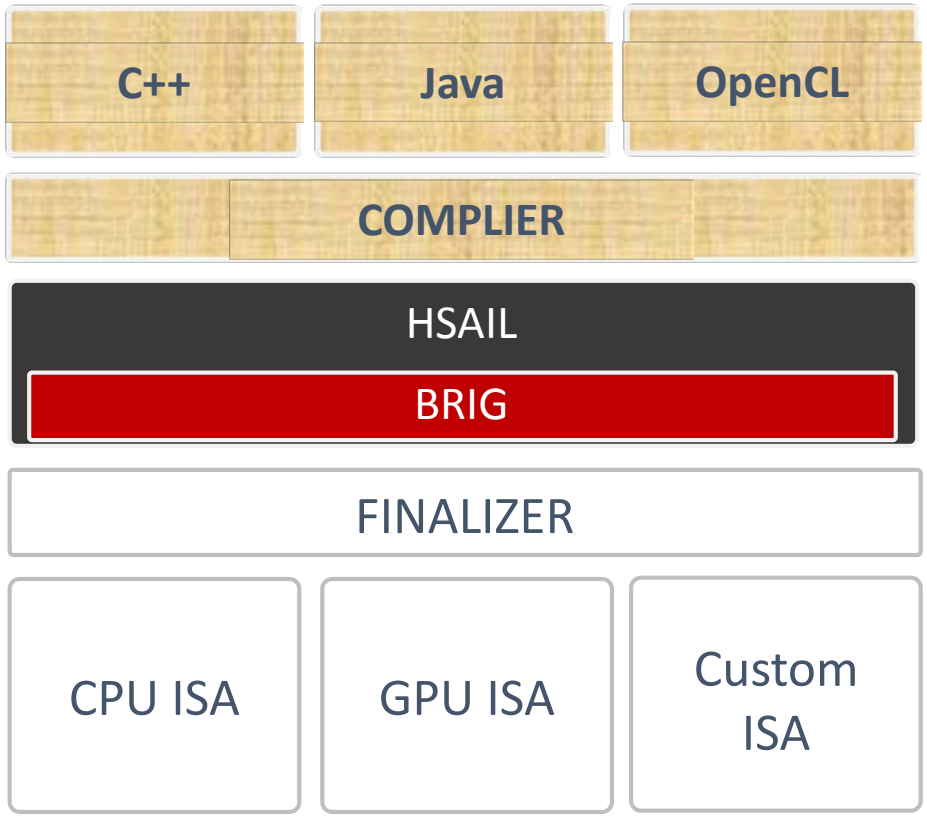
HSA INTERMEDIATE LANGUAGE

HSAIL Overview

- Introduced by HSA Foundation
- A virtual ISA with many operations.
- A textual representation of instructions and a binary format called BRIG.
- ISA, programming model, memory model, machine model, profile…..

HSAIL: Virtual ISA Abstraction

HSAIL: A virtual ISA abstraction for popular programming languages.



Programmers can program in languages they already know, and with the features and tools they expect.

Complier that generates HSAIL can be assured that the resulting code will be able to run on different target platforms

The conversion from HSAIL to machine ISA is more of a translation than a complex complier optimization.

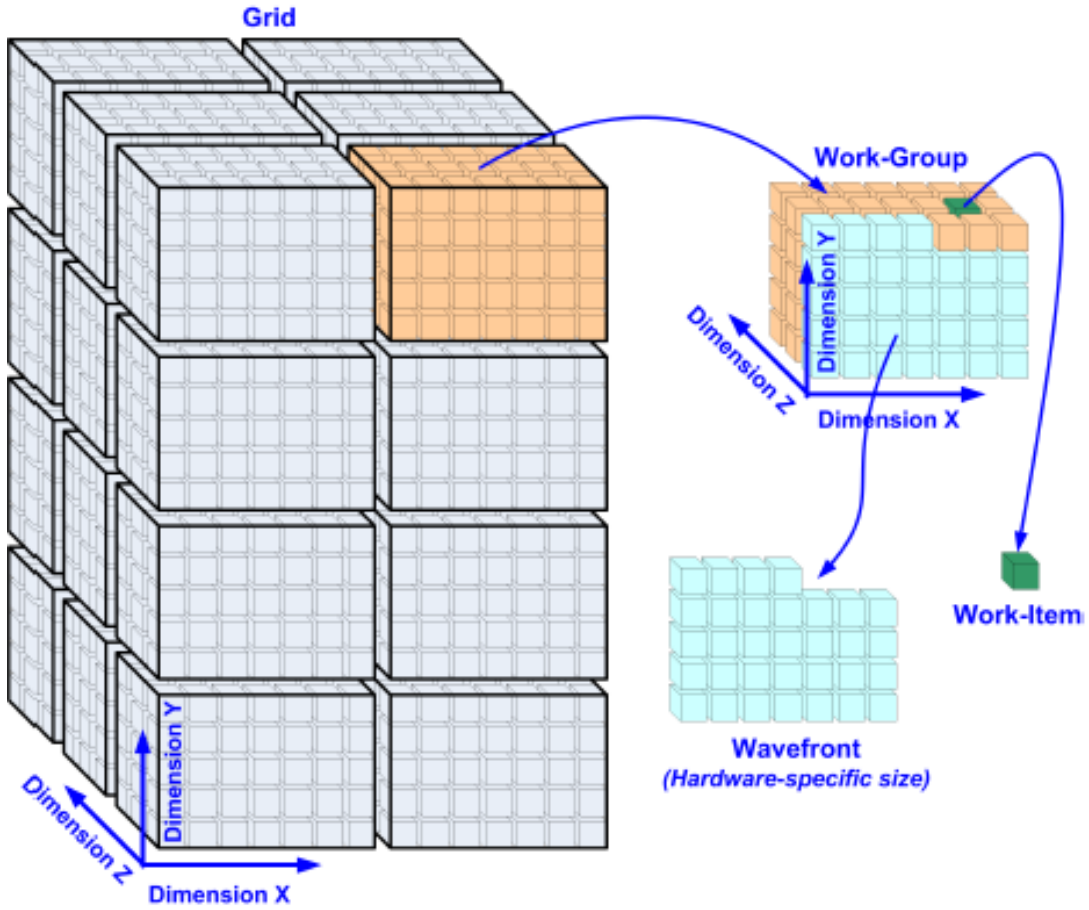
HSAIL: Programming Model

Sequential program expanded in three dimensional parallel execution model.

Grids are divided into one or more work-groups.

Work-groups are composed of work-items.

Work-items in the same work-group can efficiently communicate and synchronize with each other through the “group” memory.



HSAIL: Single instruction Multiple Data

Sequential programming run in parallel

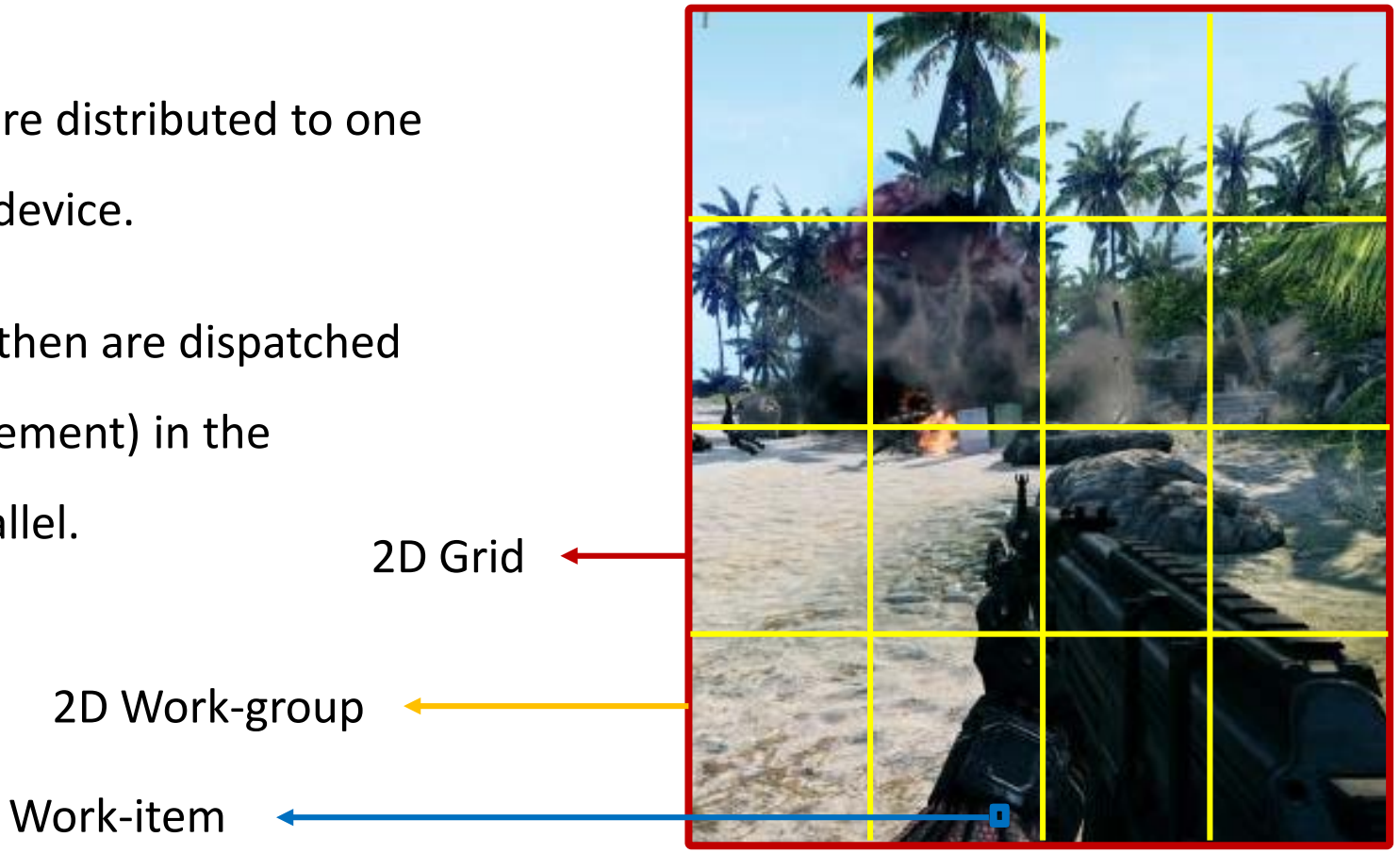
- When the parallel task is dispatched, the dispatch command specifies the number of work-items that should be executed.
- Each work-item has a unique identifier specified with x, y, z coordinate.
- HSAIL contains instructions so that each work-item can determine its unique coordinate and operate on certain part of the data.

HSAIL: Example

Parallel workgroups, parallel work-items

When a grid executes, work-groups are distributed to one or more compute units in the target device.

Work-items in the same work-group then are dispatched to each execution unit (processing element) in the computing unit, and executed in parallel.

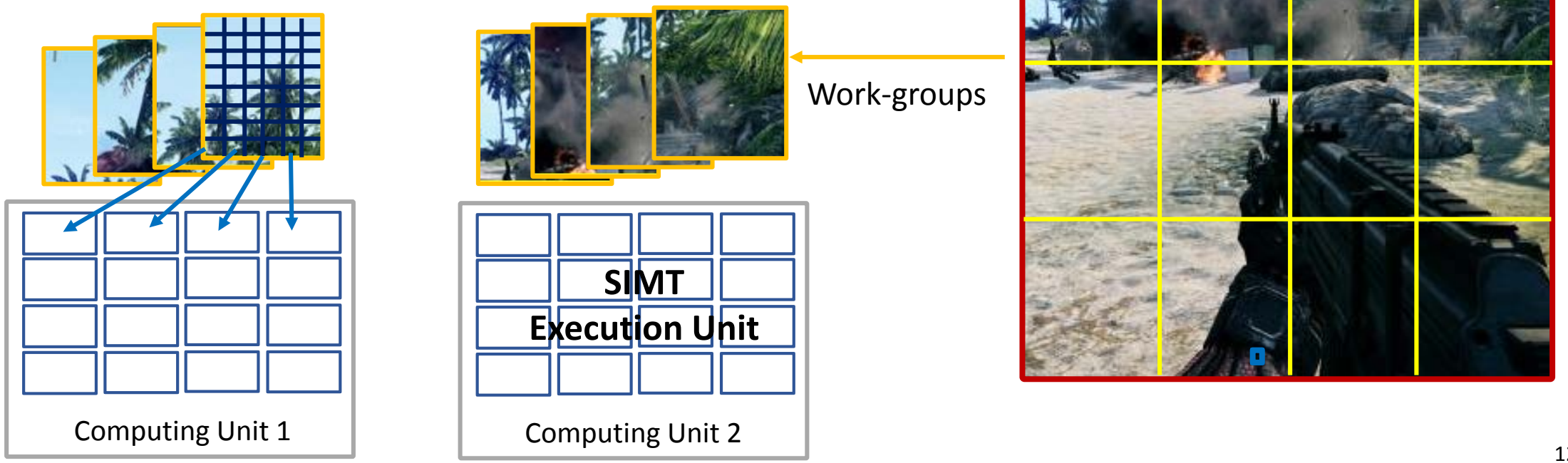


http://www.gamesaktuell.de/Crysis-Xbox360-237049/News/Crysis-3D-Support-und-Parallax-Occlusion-Mapping-fuer-die-Konsolen-Version-844757/

HSAIL: Scheduling Granularity (1)

Top level dispatch example

The grid is always scheduled in work-group-sized granularity. Work-group thus encapsulates a piece of parallel work.

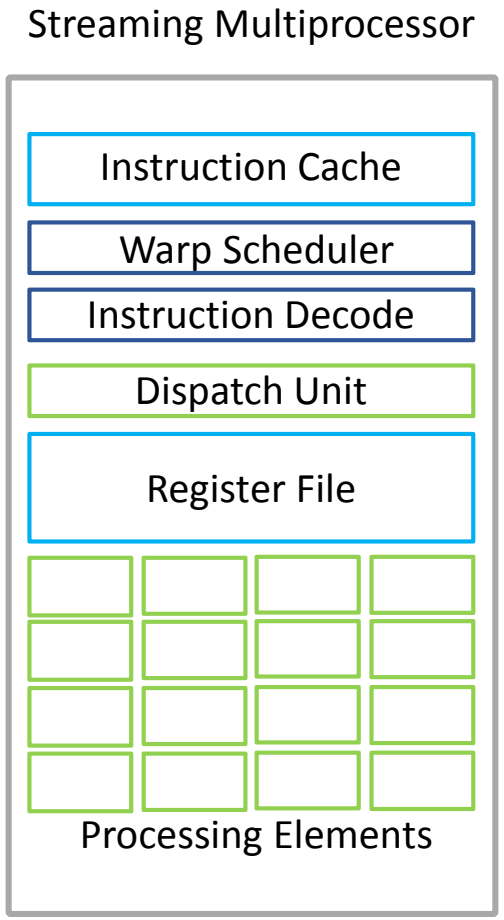


HSAIL: Scheduling Granularity (2)

Warp or Wavefront: An independent instruction stream which works on multiple data.

- The wavefront is a hardware concept indicating the number of work-items that are scheduled together.
- Wavefront width is implementation dependent.

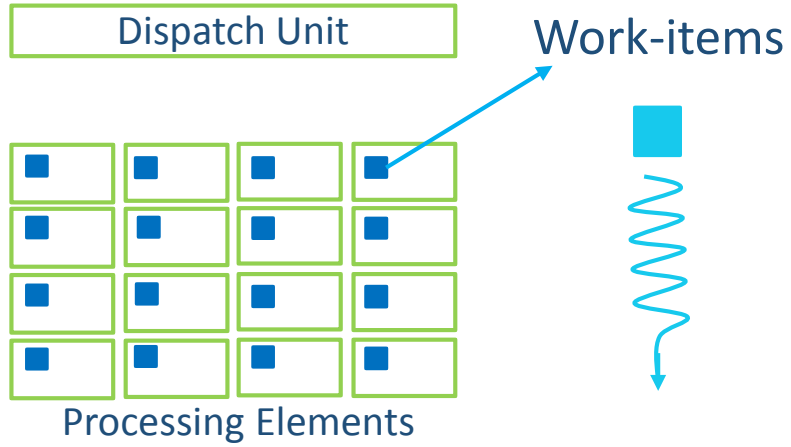
- HSAIL also provides cross-lane operations that combine results from several work-items in the work-group. (cross lane transfer of data…)



HSAIL: A RISC-like ISA, inherent for SIMT Operations

Effective Parallel Processing Comes from Machine.

- Each work-item in the HSA execution model represents a single thread of execution.
- HSAIL thus looks like a sequential program.
- Parallelism is expressed by the grid and the work-groups, which specify how many work-items to run, rather than expressed in the HSAIL code itself.



HSAIL: ISA

- Writing HSAIL code is similar to writing in assembly language for a RISC CPU

```
add_s32 $s0,$s2,$s6;
```

- The HSAIL language provides about 140 operations :

  Fundamental arithmetic operations (integer/floating point)

  Load/Store operations

  Branch operations (call, ret….)

  Multi-media operations (image operations..)

  Synchronization operations (barrier, atomic, …)

  Cross-lane operations (get register value from other lane, e.g.,)

  Special operations (get work-item ID…..)

HSAIL: Data Type

- Base Data Types :
  - Floating point (single, double, half), Integer (32-bit, 64-bit), Bit data type
- Packed data type (SIMD)
- Opaque Data Type (incomplete defined data type, image handle, for image object)

Table 4–2 Base Data Types

Type	Description	Possible lengths in bits
b	Bit type	1, 8, 16, 32, 64, 128
s	Signed integer type	8, 16, 32, 64
u	Unsigned integer type	8, 16, 32, 64
f	Floating-point type	16, 32, 64

Table 4–3 Packed Data Types and Possible Lengths

Type	Description	Lengths for 32-bit types	Lengths for 64-bit types	Lengths for 128-bit types
s	Signed integer	8x4, 16x2	8x8, 16x4, 32x2	8x16, 16x8, 32x4, 64x2
u	Unsigned integer	8x4, 16x2	8x8, 16x4, 32x2	8x16, 16x8, 32x4, 64x2
f	Floating-point	16x2	16x4, 32x2	16x8, 32x4, 64x2

HSAIL: Memory Model

Segment address, flat address

HSAIL provides 7 segments of memory :

- Global Segment : Visible to all HSA agents and work-items
- Group Segment : Shared among all work-items in the same work-group
- Private: Per work item
- Spill, Argument Segments: (not part of flat address space, meaning: using segment addressing mode)
- Kernarg Segment : Programmers use this segment to pass arguments to kernel (not part of flat address space)
- Read-only Segment : const. (not part of flat address space)

Global	Group	Spill	Private	Arg.	Kernarg	Read-Only

HSAIL Register: 128 32-bits per work-item

- HSAIL uses a fixed-size register pool.
- HSAIL provides 4 classes of registers:

  C Register : 1-bit control registers which are used to store the output of the comparison operations

  S Register : 32-bit registers that can store a 32-bit integer or a single-precision floating point value

  D Register : 64-bit registers that can store a 64-bit integer or a double-precision floating point value

  Q Register :128-bit registers that store packed values (8*16bit / 4*32bit)

- HSAIL provides up to 8 “C registers”.
- The “S”, ”D”, “Q” registers share a single pool of resources which support up to 128 “S” registers

HSAIL: Machine Model & Profile

- HSAIL intends to support a wide arrange of devices. To make sure that HSAIL can be implemented efficiently on multiple market segments, the HSA Foundation introduced the concepts of machine models and profiles.
- Machine model specifies which machine model is used during finalization.

  Small model : 32-bit address space

  Large model : 64-bit address space

- Profiles focus on features and precision requirements.

  Base profile : Smaller systems having power efficiency without sacrificing performance

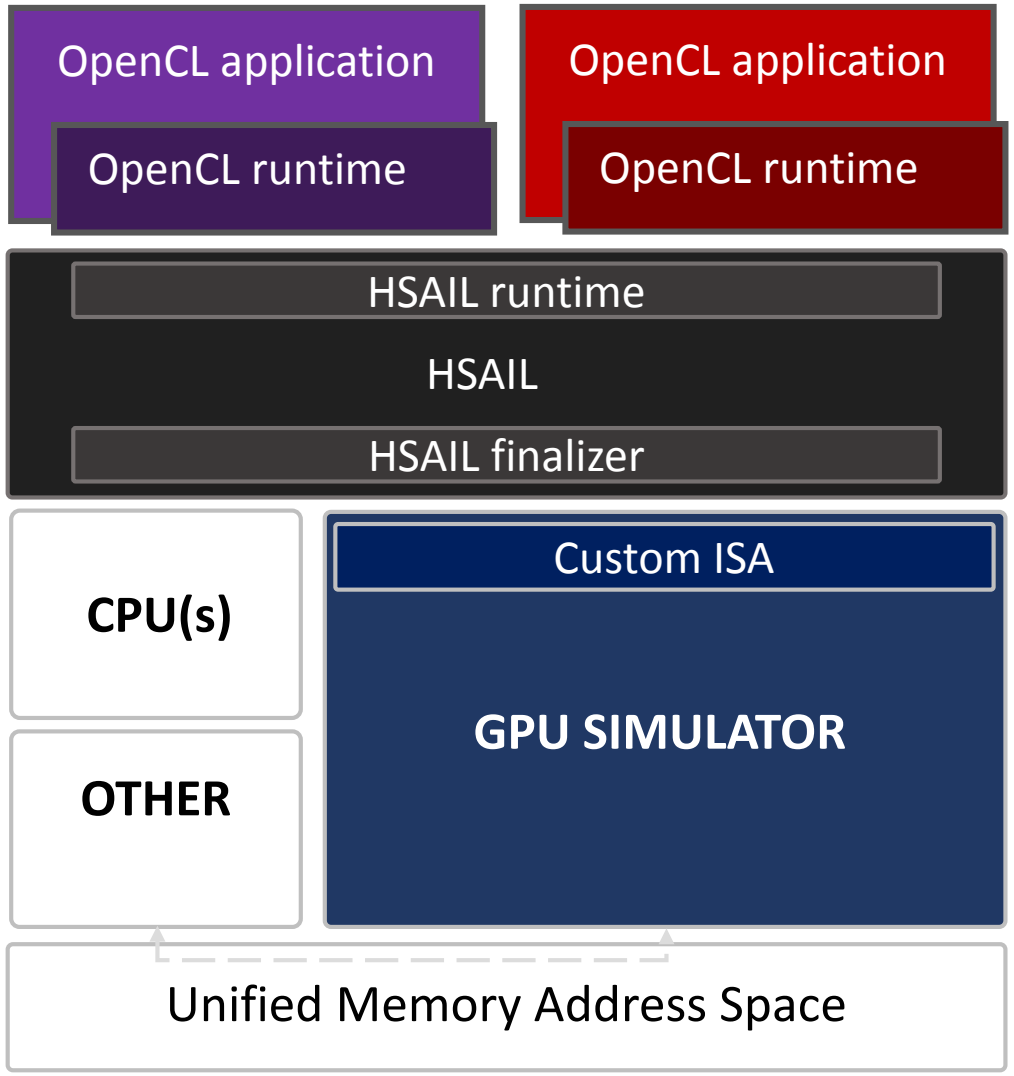
  Full profile : Larger systems having higher precision without sacrificing performance

A SIMT GPU Simulator Based on HSAIL BRIG

Processor Designed by Computer Architecture and System LAB
National Cheng-Kung University

NCKUEE dual-core pipelined CPU
Boot Linux Successfully.

Get started with a GPU SIMULATOR



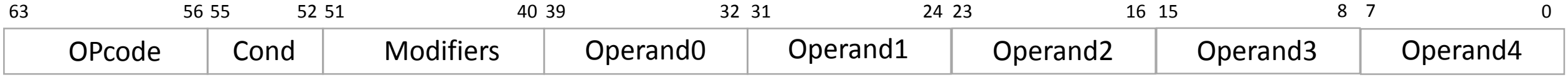
First step: a GPU ISS, which defines correct operations for SIMT.

Second: a soft GPU IP.

Third: a GPU IC.

GPU Binary ISA

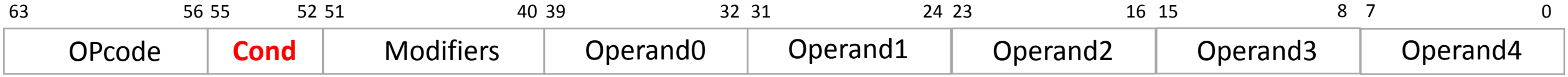
DETAIL OF THE SIMULATOR – CUSTOM ISA



- Our custom ISA is based on HSAIL BRIG (140 x variants = huge #), binary representation of HSAIL textual)
- The instructions are 64-bit long.
- Every instruction has common Opcode and Conditional field. Number of modifier and operand fields tailor to the definition in HSAIL.
- Currently in our custom ISA, there are 7 groups of instructions including : Arithmetic Operations, Memory Operations (flat address model), Branch Operations, Image (Multi-Media) Operations, Synchronization Operations, Cross-lane Operations and Special Operations.

Add Predicate to Instructions

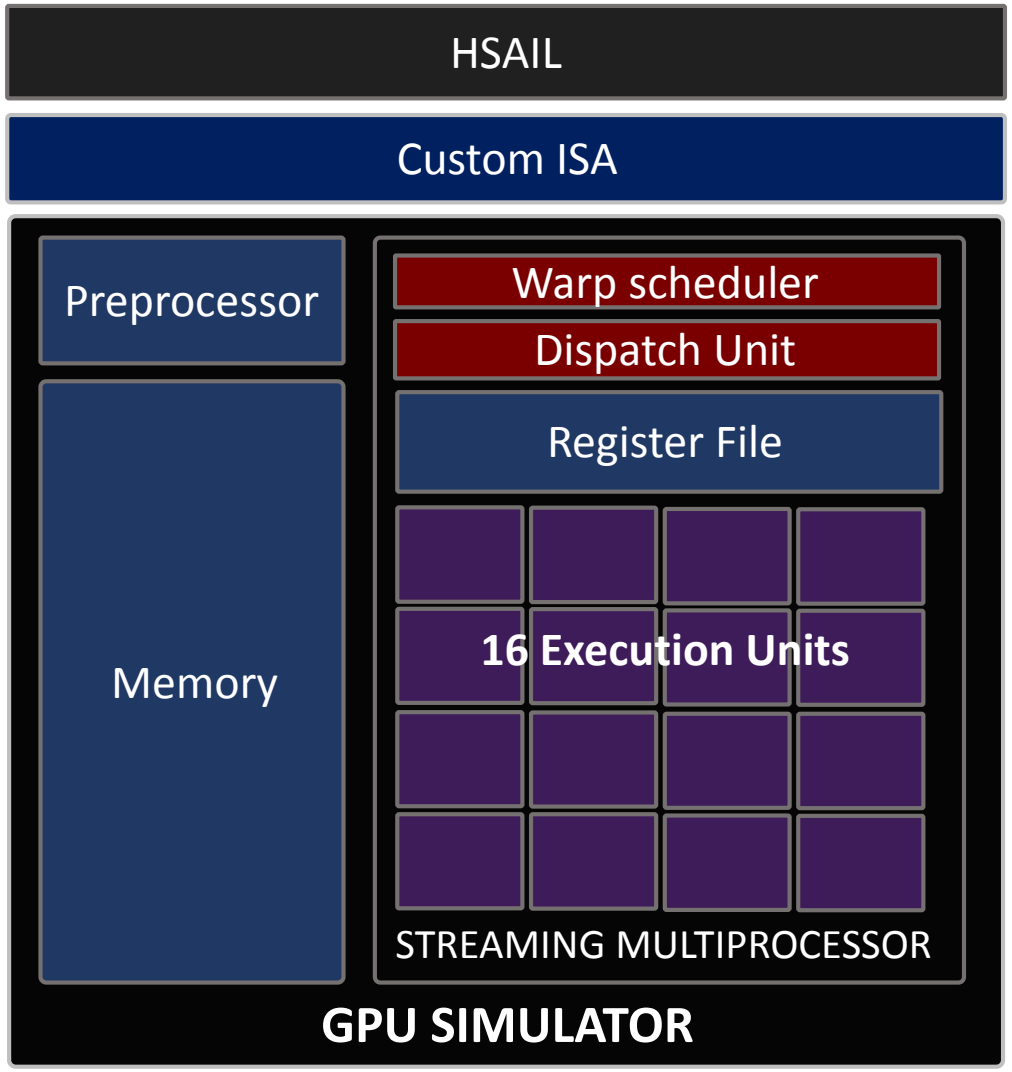
DETAIL OF THE SIMULATOR – CUSTOM ISA (Conditional Field)



Our custom ISA has an additional field called “conditional field”, which does not appear in HSAIL.

- The Parallel Thread Execution (PTX) language used in Nvidia’s CUDA programming environment supports conditional field.
- One can use predicate flag to deal with divergence control flow.

Build Core ISS First



SIMULATOR ARCHITECTURE

Develop custom Binary ISA based on HSAIL BRIG

The GPU Simulator includes:

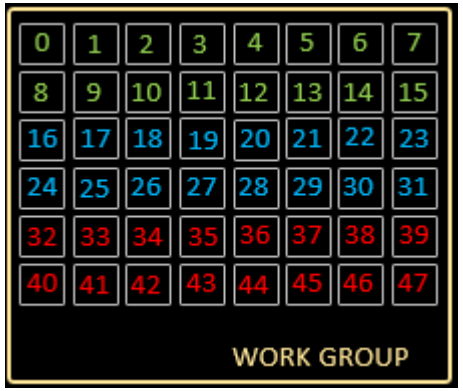
- Preprocessor Unit which passes hsail binary code (.hbin) to GPU memory.
- A simple memory model for our GPU simulation.
- A shader core includes warp scheduler, dispatch unit, register file, and 16 execution units.

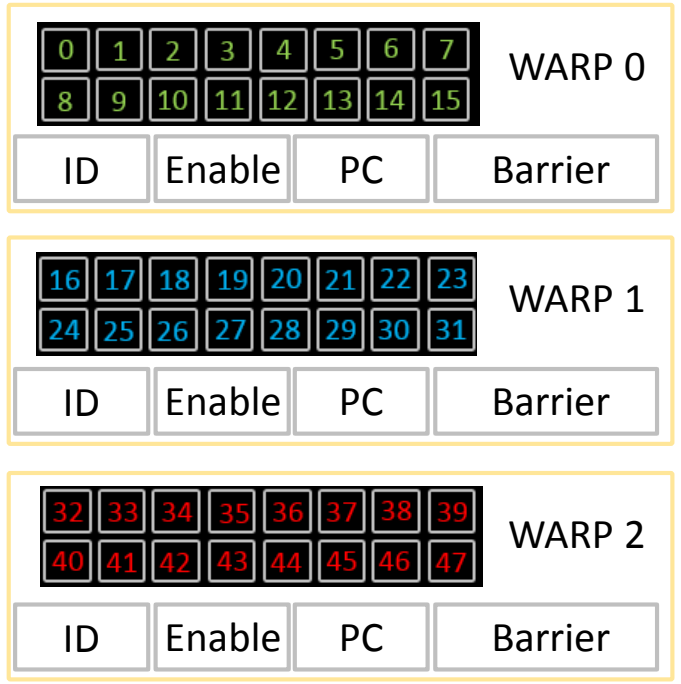
Group work-items (thread) into wavefront (warp)

DETAIL OF THE SIMULATOR – WARP SCHEDULER

Before GPU runs, the programmer specifies the number of work-items required in the kernel (work-group).

The GPU simulator then encapsulates work-items into several warps based on the wavefront size.

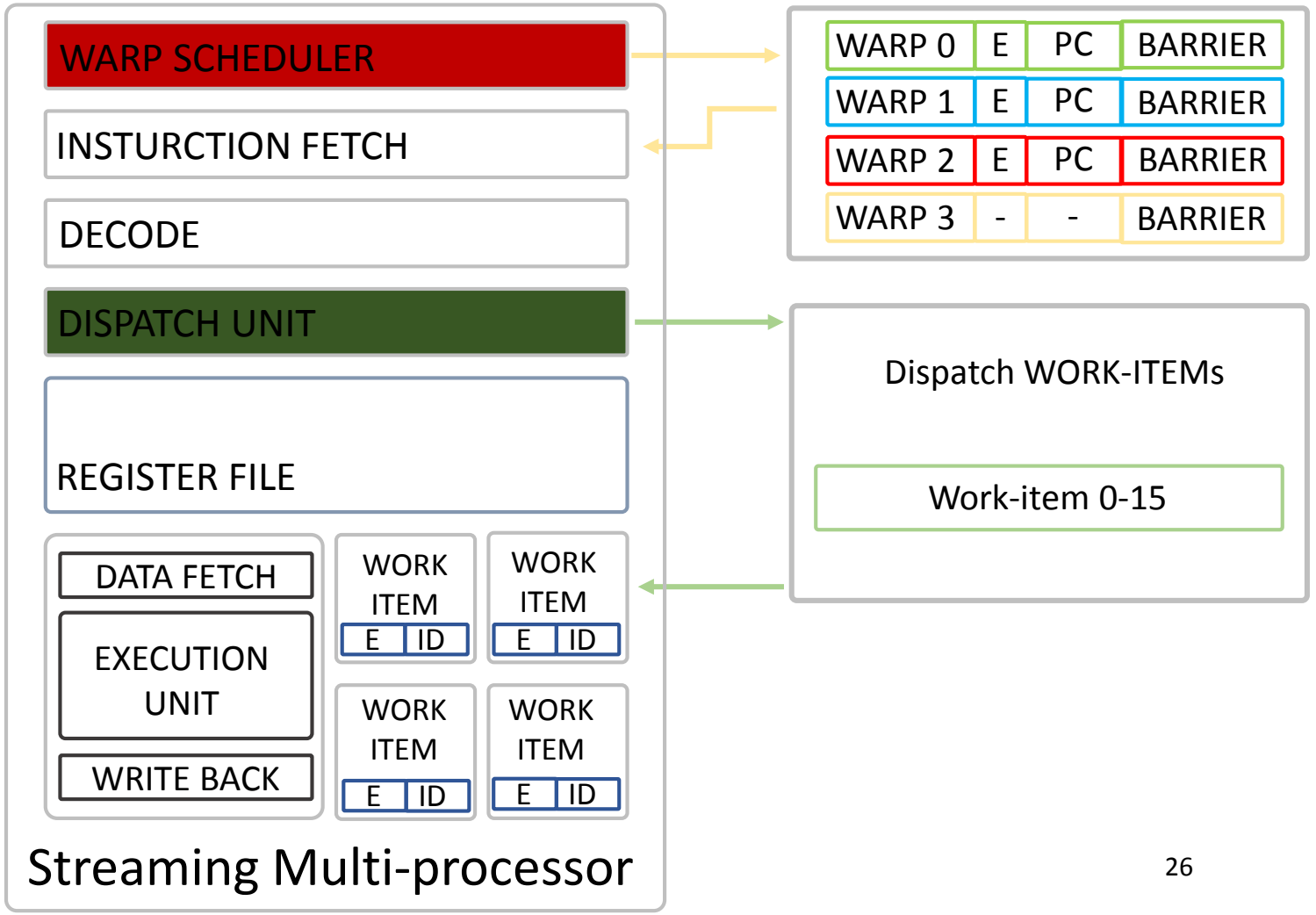




Run warp by warp

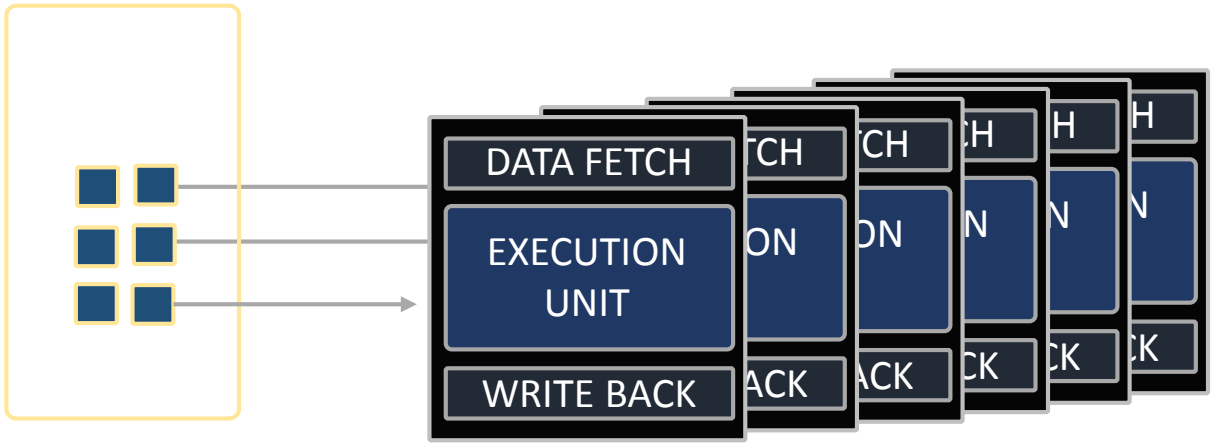
DETAIL OF THE SIMULATOR – WARP SCHEDULER & DISPATCH UNIT

Every cycle, warp scheduler selects a warp which is ready for execution.
The selected warp then goes through the Instruction Fetch and Decode stage.
Each of the work-items in the same warp is dispatched to a lane, Processing Element (execution unit) in dispatch unit.



Sufficient to Run HSAIL Benchmarks (HSA)

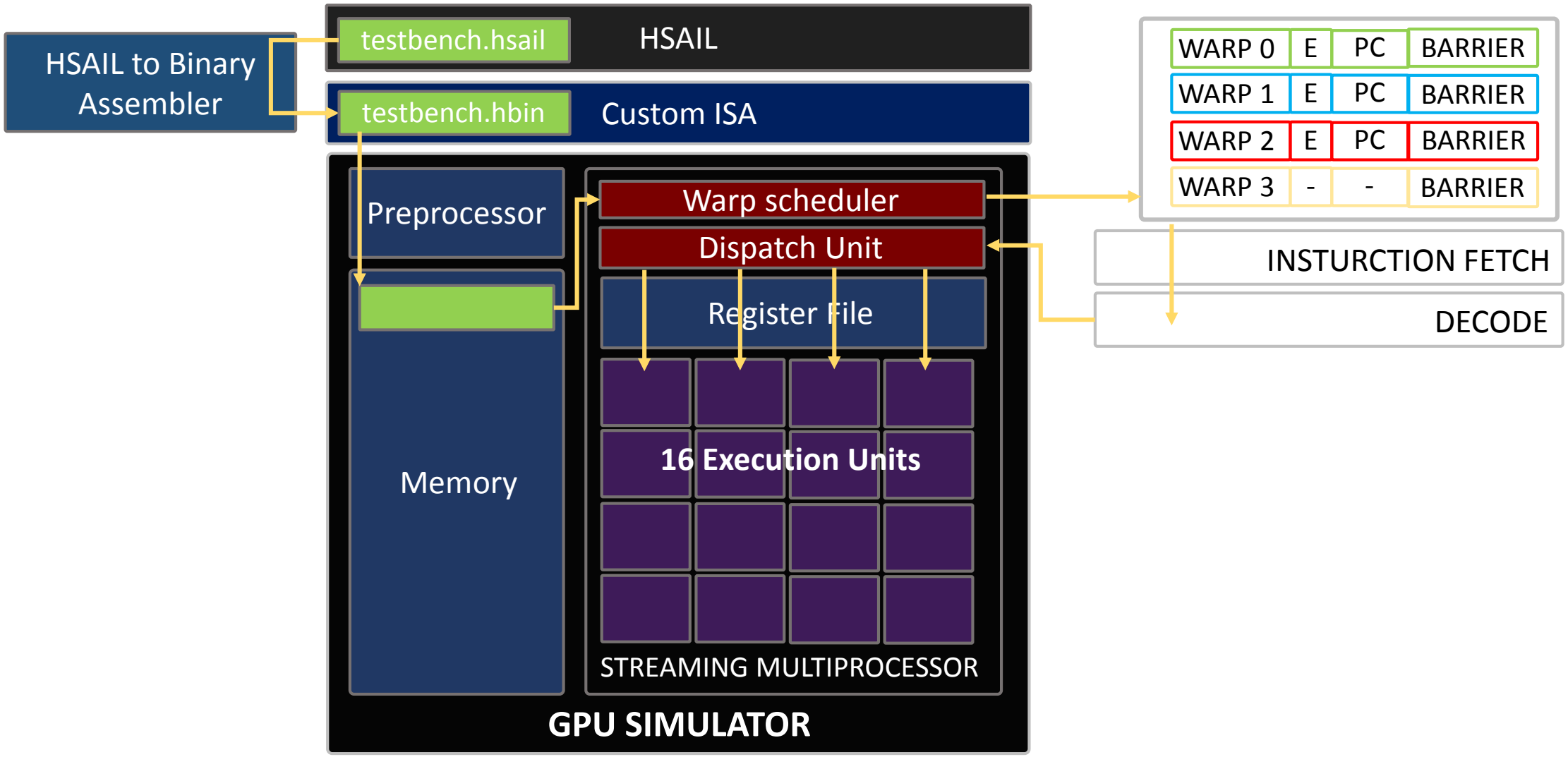
Each work-item is dispatched to a SIMD execution unit (processing element) then executed simultaneously.



Our simulator now supports most of the instructions defined by HSAIL, including Arithmetic Operations, Memory Operations, Branch Operation, Synchronization Operations and Special Operations.

ARITHMETIC OPERATION (71)									
COMMON ARITHMETIC (35)									
0	NOP	2	ABS	3	ADD	4	BORROW	5	CARRY
6	CEIL	7	COPYSIGN	8	DIV	9	FLOOR	A	FMA
B	FRACT	C	MAD	D	MAX	E	MIN	F	MUL
10	MULHI	11	NEG	12	REM	13	RINT	14	SQRT
15	SUB	16	TRUNC	17	MAD24	18	MAD24HI	19	MUL24
1A	MUL24HI	33	CMOV	35	NCOS	36	NEXP2	37	NFMA
38	NLOG2	39	NRCP	3A	NRSQRT	3B	NSIN	3C	NSQRT
ARITHMETIC WITH IMMEDIATE (8)									
90	ADDI	91	DIVI	92	MULI	93	SUBI	94	ANDI
95	ORI	96	XORI	97	NOTI				
BIT OPERATION (14)									
1B	SHL	1C	SHR	1D	AND	1E	NOT	1F	OR
20	POPCOUNT	21	XOR	22	BITEXTRACT	23	BITINSERT	24	BITMASK
25	BITREV	26	BITSELECT	27	FIRSTBIT	28	LASTBIT		
PACKED DATA OPERATION (5)									
2E	SHUFFLE	2F	UNPACKHI	30	UNPACKLO	31	PACK	32	UNPACK
OTHER OPERATION (9)									
29	COMBINE	2A	EXPAND	2D	MOV	34	CLASS	44	SEGMENTP
45	FTOS	46	STOF	47	CMP	48	CVT		
MEMORY OPERATION (7)									
49	LD	4A	ST	4B	ATOMIC	4C	ATOMICNORET	4D	SIGNAL
4E	SIGNALNORET	4F	MEMFENCE						
BRANCH OPERATION (6)									
5F	BRA		CBR		CALL		RET		ENDBRA
	EXIT								
IMAGE OPERATIONS (8)									
50	RDIMAGE	51	LDIMAGE	52	STIMAGE	53	ATOMICIMAGE	54	ATOMICIMAGENORET
55	QUERY	98	LDIMAGE_LEVEL	99	LDIMAGE_GRAD				
SYNCHRONIZATION OPERATION (8)									
60	BARRIER	61	WAVEBARRIER	62	ARRIVEFBAR	63	INITFBAR	64	JOINFBAR
65	LEAVEFBAR	66	RELEASEFBAR	67	WAITFBAR				
CROSS-LANE COMMUNICATION (4)									
69	ACTIVELANECOUNT	6A	ACTIVELANEID	6B	ACTIVELANEMASK	6C	ACTIVELANESHUFFLE		
SPECIAL OPERATION (10)									
70	DDQUEUEWRITEINDE	71	ASQUEUEWRITEINDE	72	CLEARDETECTEXCEPT	73	LDS	76	DEBUGTRAP
7C	LDQUEUEREADINDEX	7D	LDQUEUEWRITEINDEX	86	SETDETECTEXCEPT	87	STQUEUEREADINDEX	88	STQUEUEWRITEINDEX

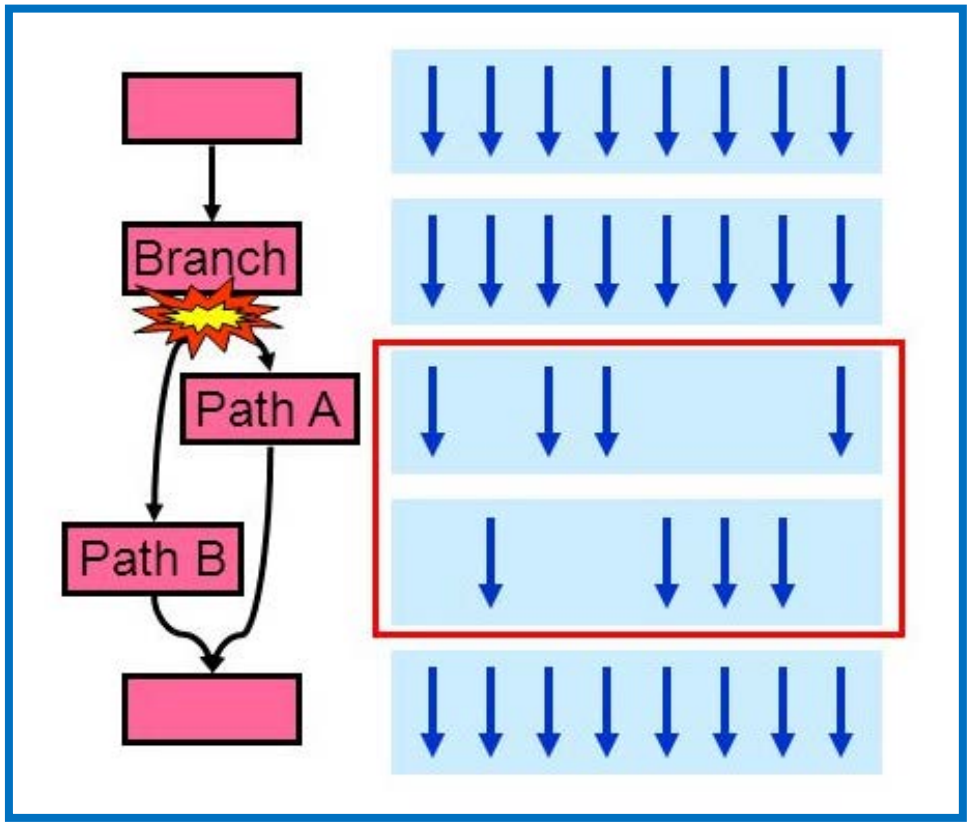
Custom Assembler (HSAIL Symbolic to Binary) used at the moment



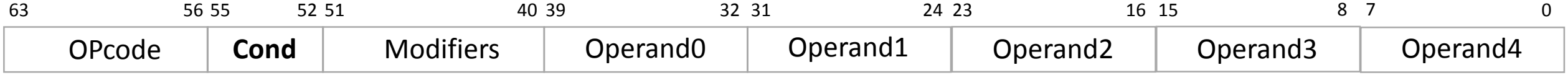
ISSUE– BRANCH OPERATIONS in SIMT

- 1: Predicate execution
- 2: Execute both paths and then re-converge, but…..

Big issue since many possible structures:
if-else, nested if-else, loop,
break, continue…



Remove Simple branch with predicate execution




```
1  lds workitemabsid $s0
2  ld $s1,[$s0+100];
3  ld $s2,[$s0+200];
4  if($s0>31)
5  {
6      add $s3,$s1,$s2;
7  }
8  else
9  {
10     sub $s3,$s1,$s2;
11 }
```

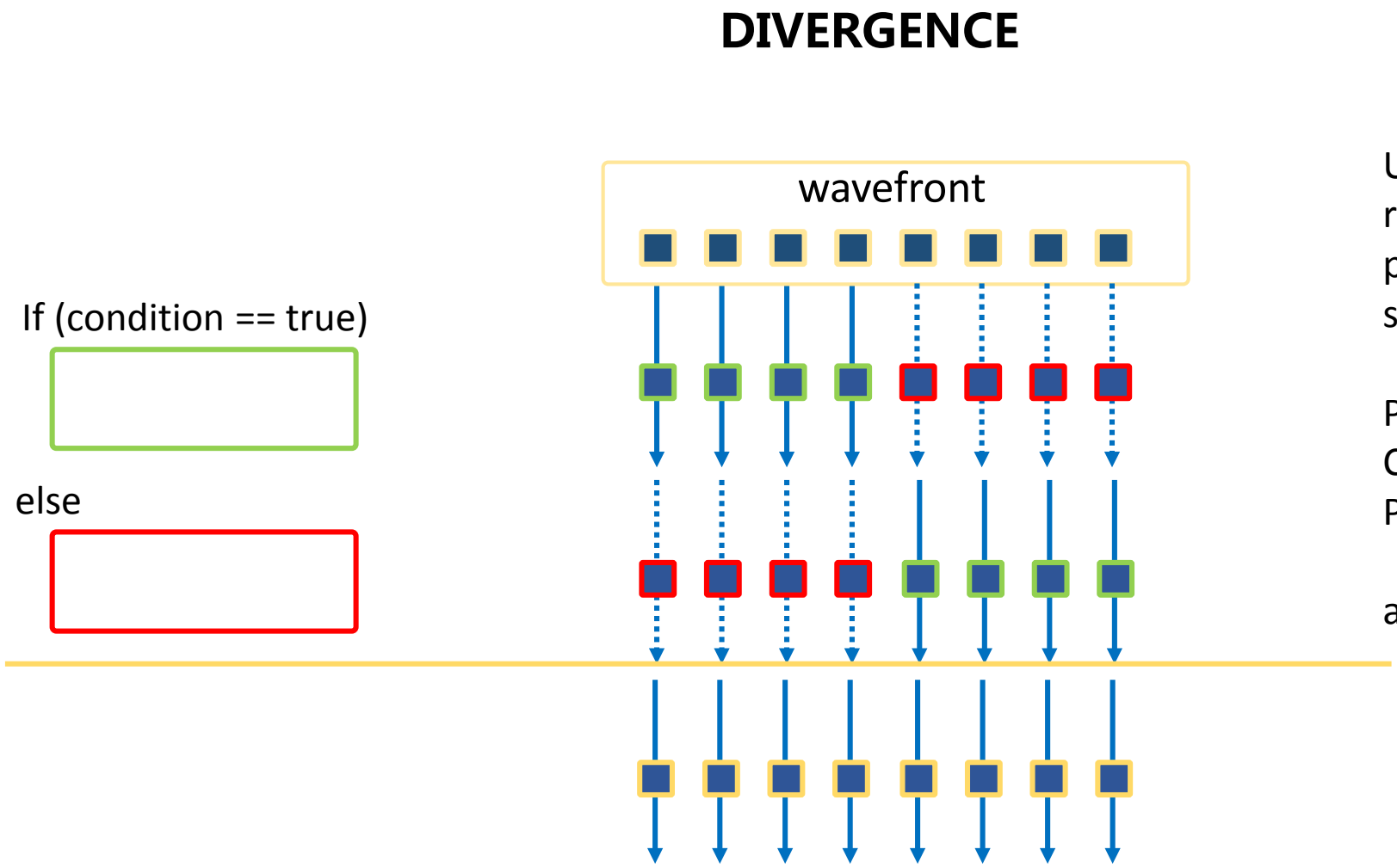
Without Conditional Field

```
cmp_lt_u32 $c0, $s0,31;
cbr $c0,@ELSE
add $s3,$s1,$s2;
@ELSE:
sub $s3,$s1,$s2;
endbranch
```

With Conditional Field

```
cmp_lt_u32 $c0, $s0,31;
($c0)add $s3,$s1,$s2;
(!$c0)sub $s3,$s1,$s2;
```

Divergence



Use stack-based mechanism to resolve the branch divergence problem (ex: Program counter stack, active lane mask stack)

Push (mask, else block address)
Complement (~mask, else block)
Pop. (Convergence)

active lane mask stack

EXAMPLE-BRANCH INSTRUCTION EXECUTION FLOW

Shader Assembly:

```
1.  @__OpenCL_vec_copy_kernel_entry:
2.          mov_b32   $s0, 1;
3.          mov_b32   $s1, 3;
4.          mov_b32   $s3, 0;
5.          workitemabsid_u32    $s2, 0 ;
6.          cmp_ge_b1_u32    $c0, $s2, $s1 ;
7.          cbr     $c0, @Btag1 ;
8.          add_u32  $s3, $s0, $s1 ;
9.          brn           @Btag2
10. @Btag1:
11.         sub_u32  $s3, $s2, $s1 ;
12. @Btag2:
13.         endbranch;
14.         mov_b32   $s4, 7;
15. };
```

Instruction Execution flow:

Active workitem	:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
#2 mov_32	:	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
#3 mov_32	:	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
#4 mov_32	:	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
#5 workitemabsid	:	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
#6 cmp_ge_b1_u32	:	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
#7 cbr	:				X	X	X	X	X	X	X	X	X	X	X	X	X
#11 sub_u32	:				X	X	X	X	X	X	X	X	X	X	X	X	X
#13 endbranch	:				X	X	X	X	X	X	X	X	X	X	X	X	X
#8 add_u32	:	X	X	X													
#9 brn	:	X	X	X													
#13 endbranch	:	X	X	X													
#14 mov_u32	:	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X

Event:

1.Push stack (Save active lane mask , PC of Else block), and go to @Btag1 (

2. Complement the masked control register and go to the PC for else block

3. Recover the masked control register, and pop stack

Divergence Control Schemes

Technique	Traversal order	Reconvergence	Indirect
NVIDIA	ISA+address stack	ISA+mask stack	Yes
AMD	ISA	ISA+mask stack	No
Intel Sandy Bridge	ISA	ISA+PC	No
Our work	ISA+address stack	ISA+mask satck	Yes

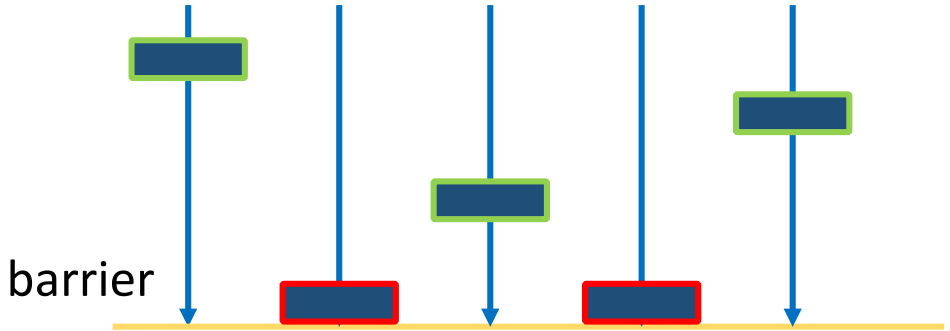
Common Control Divergence Structure

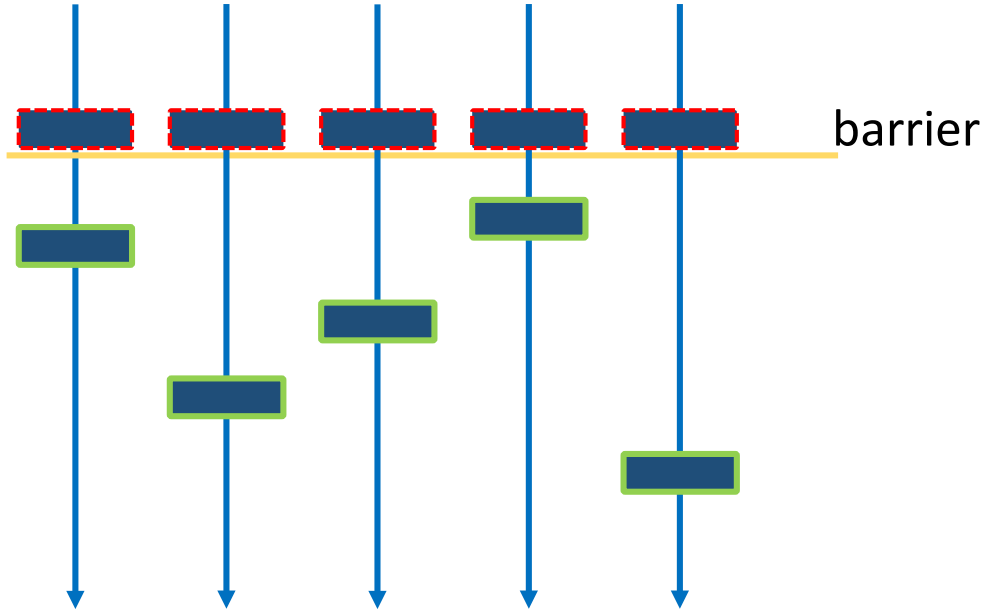
- If-else
- Nested if else
- Indirect jump (switch, case)
- For loop, while
- Break or continue in loop

Synchronization

ISSUE2: SYNCHRONIZATION

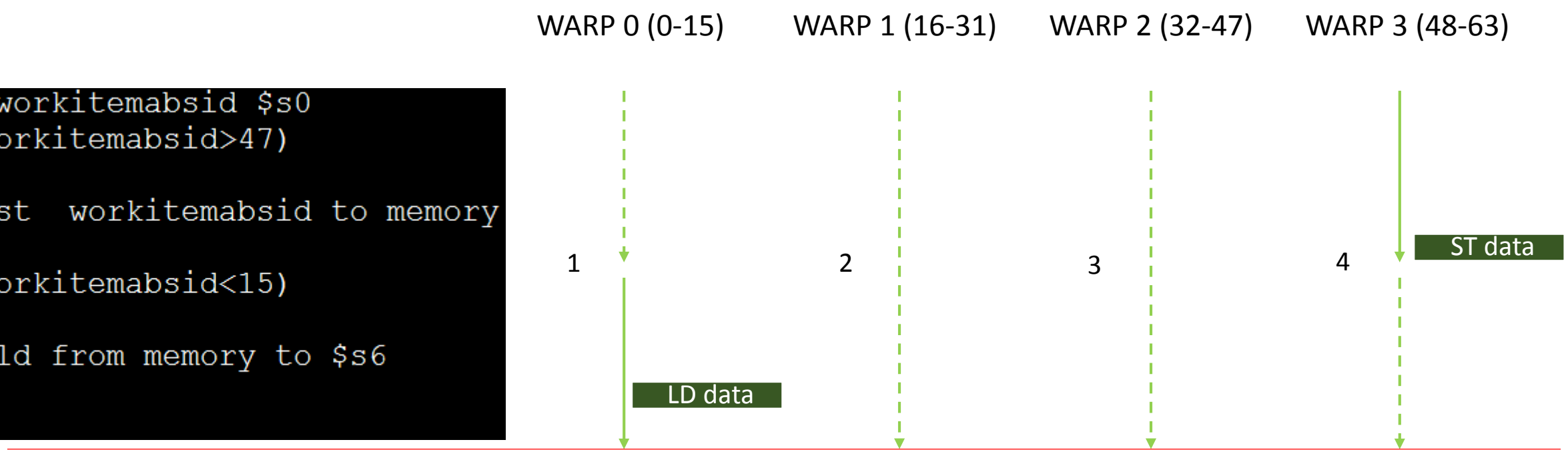
BARRIER operations provide an approach to synchronize all work-items in the same work-group.





Exchange data without synchronization

```
lds_workitemabsid $s0
if(workitemabsid>47)
{
    st  workitemabsid to memory
}
if(workitemabsid<15)
{
    ld from memory to $s6
}
NOP;
```



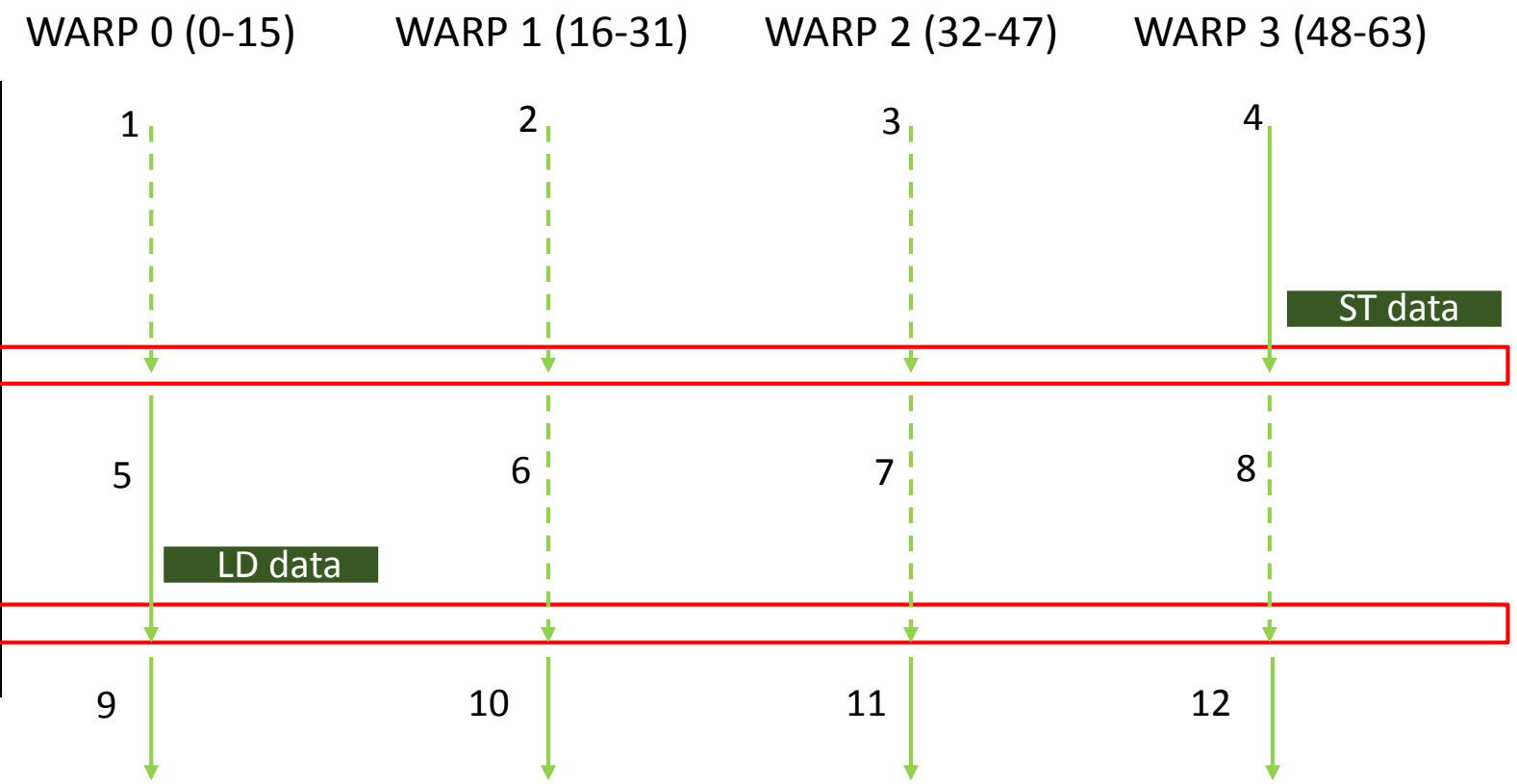
Assume wavefront size = 16,

Assume store memory x, load from memory x (group or global)

Load data before the data are written, which is incorrect!

Exchange data with barrier

```
lds_workitemabsid $s0
if(workitemabsid>47)
{
    st  workitemabsid to memory
}
BARRIER;
if(workitemabsid<15)
{
    ld from memory to $s6
}
BARRIER;
NOP;
```



Assume wavefront size = 16,

Scheduling ready warp or that reaching barrier count

Barrier operations can not be used in divergent control flow.

Barrier operations must be wavefront uniform.

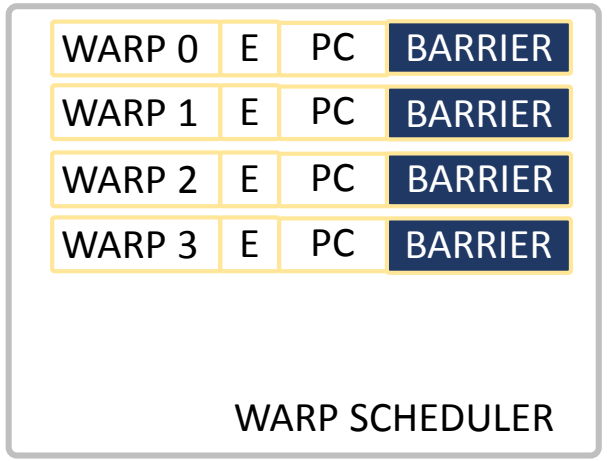
BARRIER OPERATION

WARP [ID] . BARRIER = 1

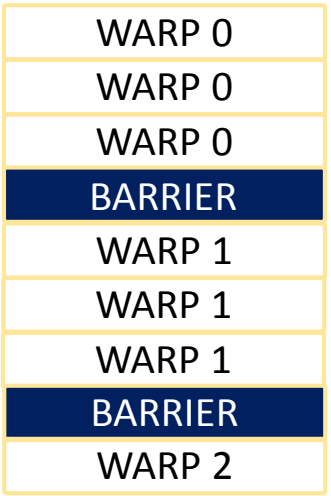
BARRIER COUNT + 1

If (WARP SIZE = BARRIER COUNT)
Reset BARRIER (free to go)

WARP SIZE	BARRIER COUNT



SCHEDULING MECHANISM



Fine-grained barrier

Fine-grained barriers (FBARRIERs) are used to synchronize some of the work-items within a work-group.
Wavefront uniform (all work-items in a wavefront join or not)

FBARRIER TABLE

FBARRIER OP.	Valid	member_count	arrive_count	wait_set	member_set
INITFBARRIER	1	0	0	0	0
JOINFBARRIER		+1			+ID
WAITFBARRIER			+1	+ID	
ARRIVEFBARRIER			+1		
LEAVEFBARRIER		-1			-ID
RELEASEFBARRIER	0				

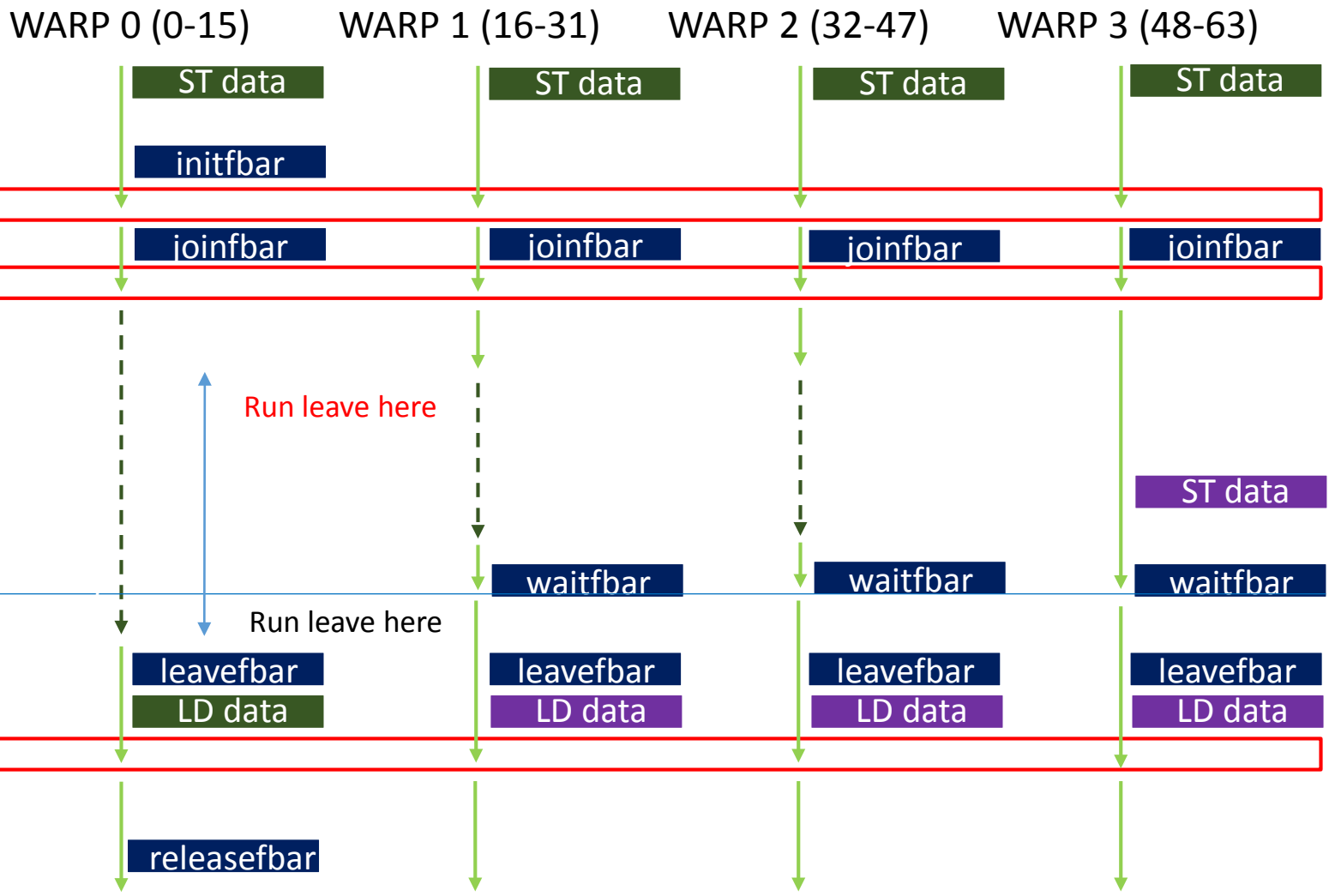
INITFBARRIER : Before fbarrier can be used, it must be initialized.
fbarrier can not be initialized twice.

WAITFBARRIER These three operations should check if (member_count = arrive_count)
ARRIVEFBARRIER : The fbarrier should be reset if the above condition is true.
LEAVEFBARRIER (release membership)

RELEASEFBARRIER : All fbarriers must be released before work group exits.
fbarrier can not be released twice.

Fbarrier example

```
1  st workitemabsid to memory
2  if(workitemabsid=0)
3      initfbar 0;
4  barrier;
5  joinfbar 0;
6  barrier;
7  if(workitemabsid>15)
8  {
9      if(workitemabsid>47)
10     {
11         $s1 = workitemabsid*10;
12         st $s1 to memory
13     }
14     waitfbar 0;
15 }
16 leavefbar 0;
17 ld workitemabsid from memory
18 barrier;
19
20 if(workitemabsid=0)
21     releasefbar 0;
```



Summary

1. We have developed a custom binary ISA based on the HSAIL BRIG.
2. Developed a SIMT simulator including warp scheduler, dispatch unit, and SIMD execution units for parallel computing .
3. Provided approaches to deal with divergence control and synchronization operations.