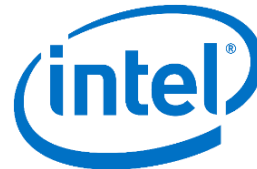


# Architecture Exploration for HLS-Oriented FPGA Debug Overlays

Al-Shahna Jamal, Jeffrey Goeders, Steve Wilton

FPGA'18 - Monterey, CA

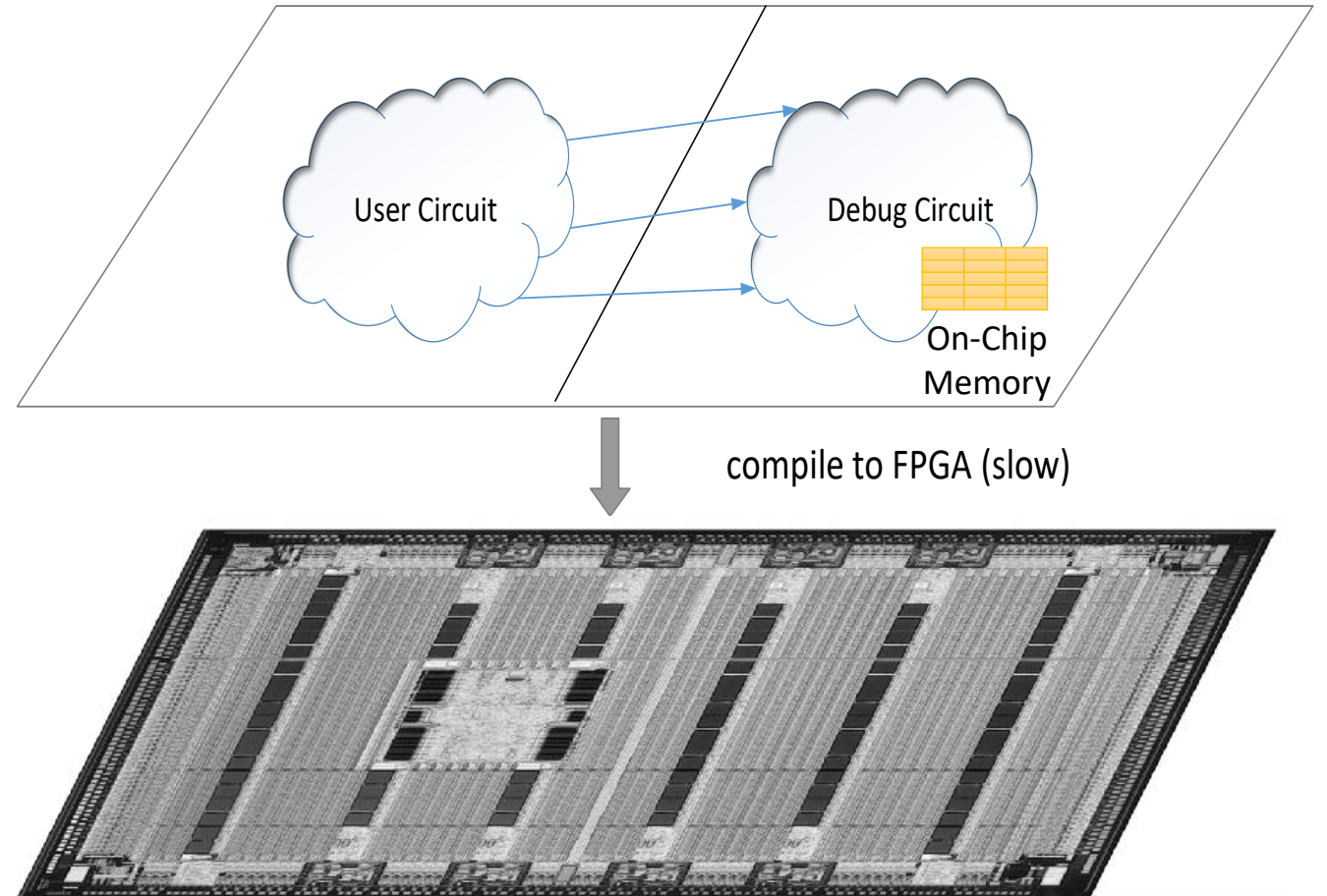


THE UNIVERSITY  
OF BRITISH COLUMBIA

# What this talk is about...

*Recent work:* Source-level, in-system debugging of HLS circuits

- Debug instrumentation is inserted at compile time
- Changing this instrumentation (to trace new data) requires a *recompile*



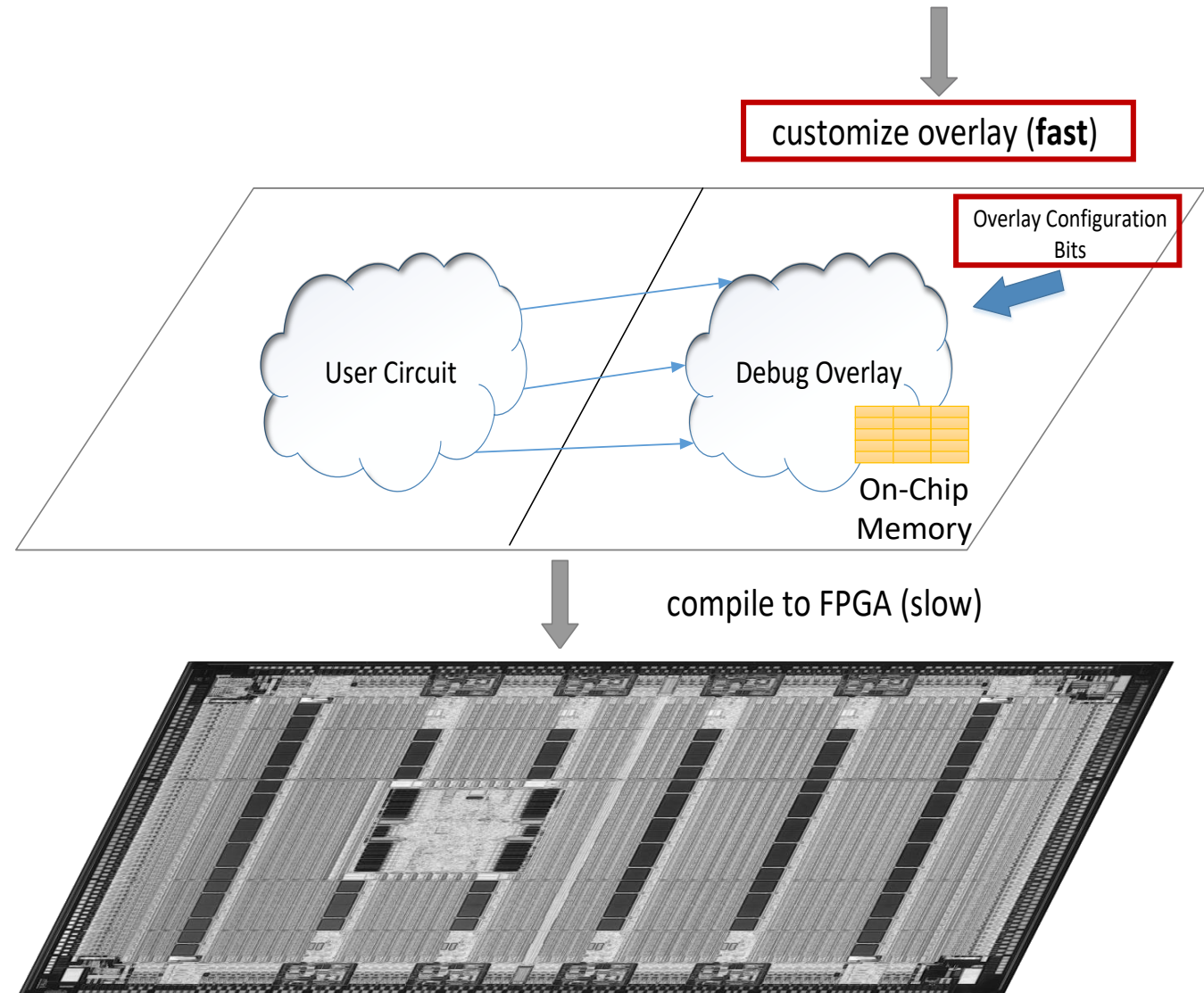
# What this talk is about...

*Recent work:* Source-level, in-system debugging of HLS circuits

- Debug instrumentation is inserted at compile time
- Changing this instrumentation (to trace new data) requires a *recompile*

*In this work:* Debug instrumentation still inserted at compile time BUT can be configured at runtime (**fast customization**)

**Impact: Achieves software like compile times (~1sec) between debug iterations**



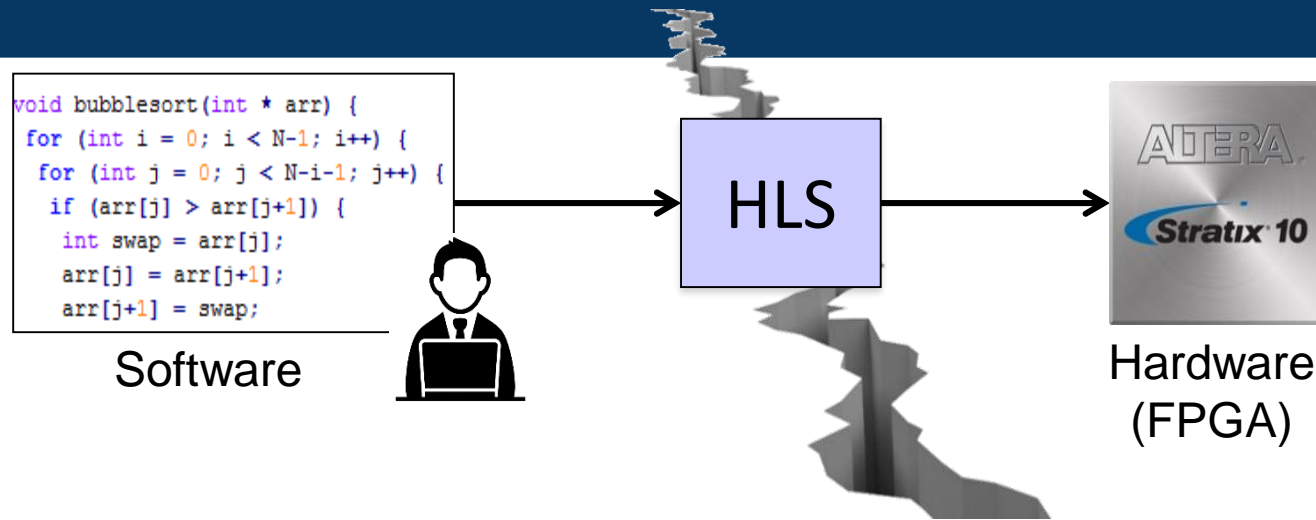
# Outline

- Motivation for In-System Debug
- Previous Work: In-System Debug Framework for HLS
  - Debug Instrumentation at compile time
- This paper: HLS Debug Overlay to allow customization at runtime
- Evaluation
- Future Work

# Outline

- Motivation for In-System Debug
- Previous Work: In-System Debug Framework for HLS
  - Debug Instrumentation at compile time
- This paper: HLS Debug Overlay to allow customization at runtime
- Evaluation
- Future Work

# High-Level Synthesis



Software designers need a full ecosystem of tools:

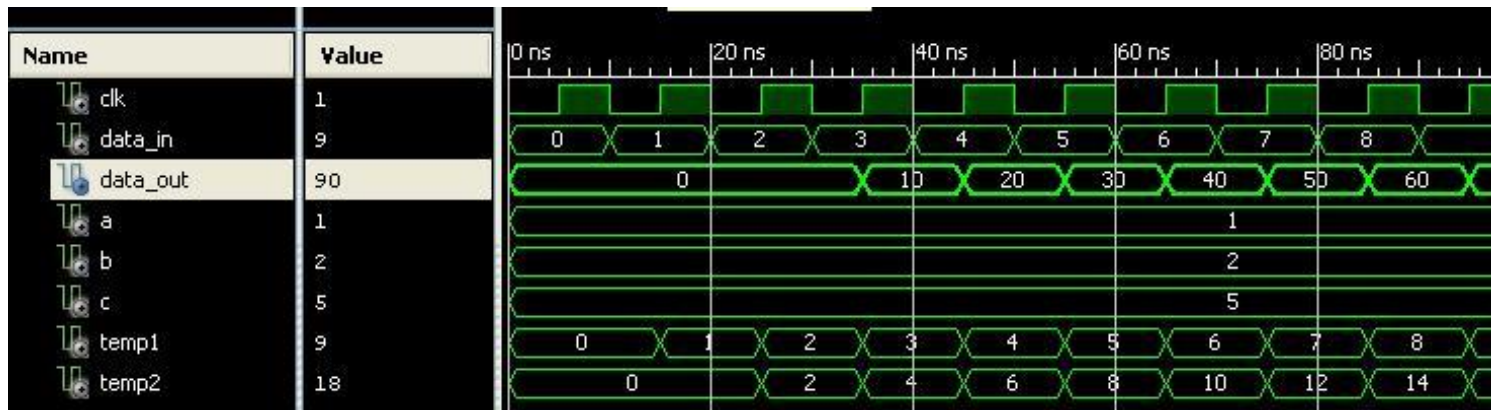
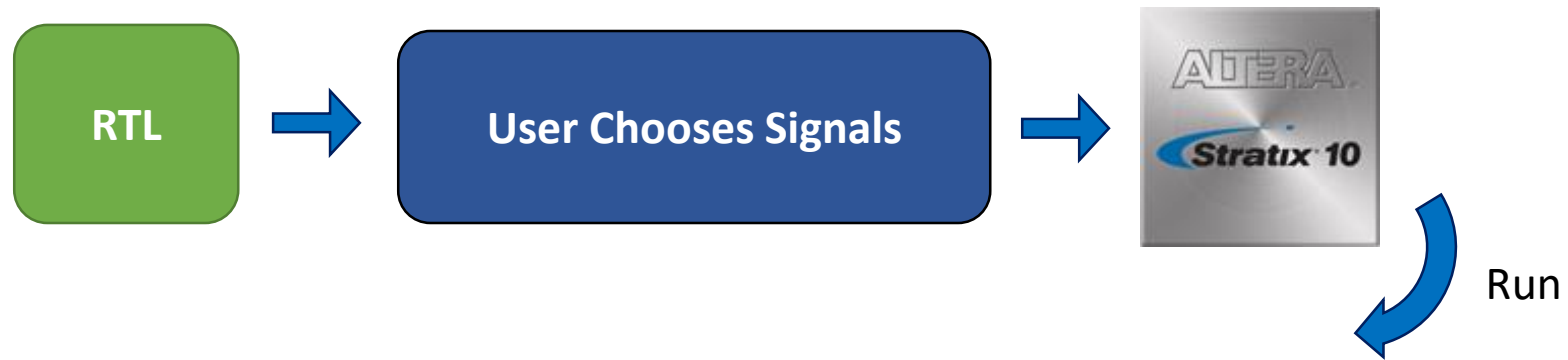
- Testing, debugging, optimization....

**Debugging:** When do we have to do in-system debug?

- Simulation may take too long
- Bug may be dependent on system interactions, IO traffic, etc.

**For certain bugs we have to perform in-system debug, observing the actual hardware**

# Hardware Debug Tools



Not practical for a software designer!

# Outline

- Motivation for In-System Debug
- Previous Work: In-System Debug Framework for HLS
  - Debug Instrumentation at compile time
- This paper: HLS Debug Overlay to allow customization at runtime
- Evaluation
- Future Work



# Previous Work: In-System Debug Framework for HLS

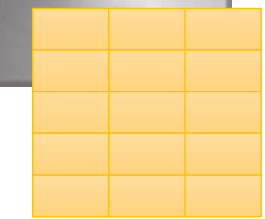
Capture system-level bugs → Need to run at-speed, on-chip

## Solution: Record and Replay

1. User selects variables, tool determines signals, inserts instrumentation

```
void qSort(int *arr) {  
    int piv, beg[N], end[N];  
    int i=0;  
    int L, R, swap;  
    ...  
}
```

2. Compile



On-Chip Memory

3. Execute and record

4. Stop and retrieve

The screenshot shows a debugger window titled "Execution Mode FPGA Replay". The code editor displays the following C code:

```
13 }  
14  
15 void quickSort(int *arr, int elements) {  
16     int piv, beg[15], end[15], i = 0, L, R, swap;  
17  
18     beg[0] = 0;  
19     end[0] = elements;  
20     while (i >= 0) {  
21         L = beg[i];  
22         R = end[i] - 1;  
23         if (L < R) {  
24             piv = arr[L];  
25             while (L < R) {  
26                 while (arr[R] >= piv && L < R)  
27                     R--;  
28                 if (L < R)  
29                     arr[L++] = arr[R];  
30                 while (arr[L] <= piv && L < R)  
31                     L++;  
32                 swap(arr[L], arr[R]);  
33             }  
34         }  
35     }  
36 }
```

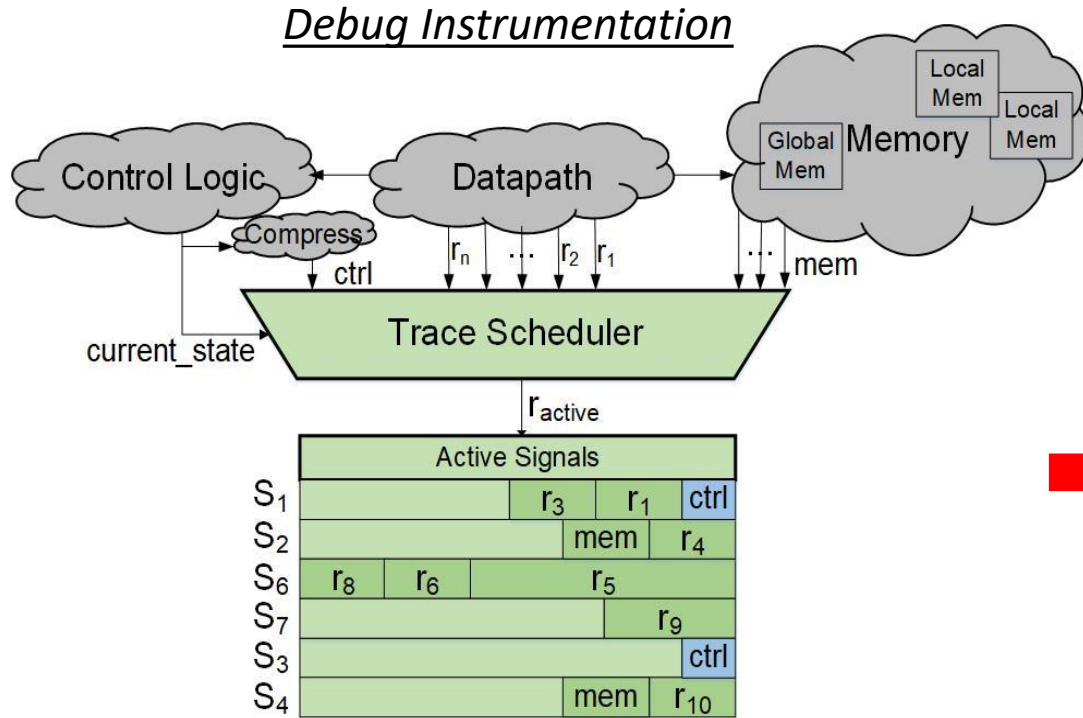
The variable table on the right shows the following values:

Variable	Value
--Globals--	
main	
i	20
beg	
end	
arr	
elements	20
i	1
L	18
R	19
piv	53704
swap	20
correct	<N/A>

5. Software-like debug using recorded data

Limited on-chip memory → Need to select what we want to record and use memory efficiently

# Previous Work: Taking Advantage of HLS Scheduling



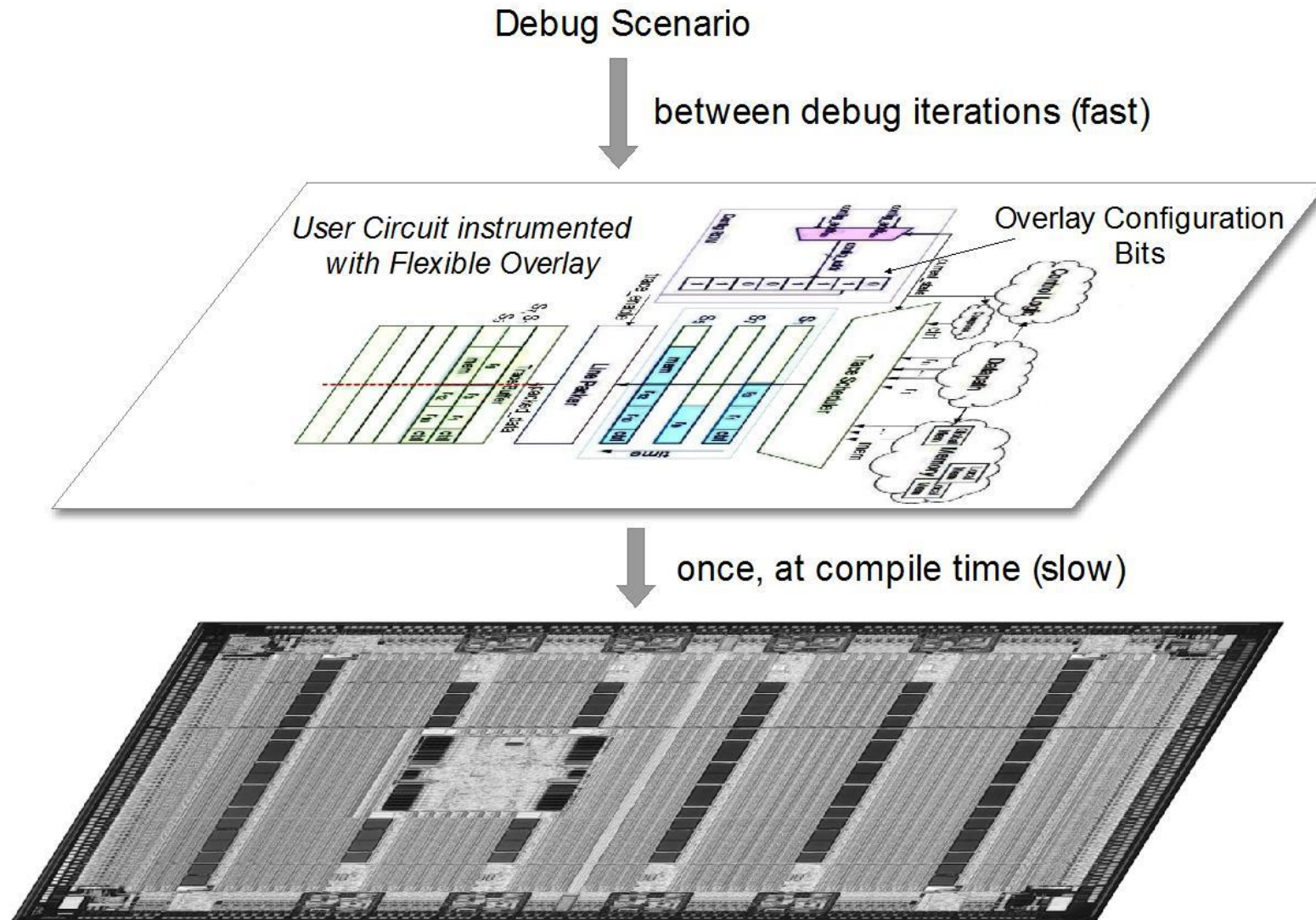
- Recorded signals change each cycle
- 50x-100x more memory efficient than traditional Embedded Logic Analyzer (ELA) approach

- Circuit-by-Circuit custom compression
- Based on signals selected for tracing (compression algorithms)
- Selecting a different subset of signals requires a **recompile**

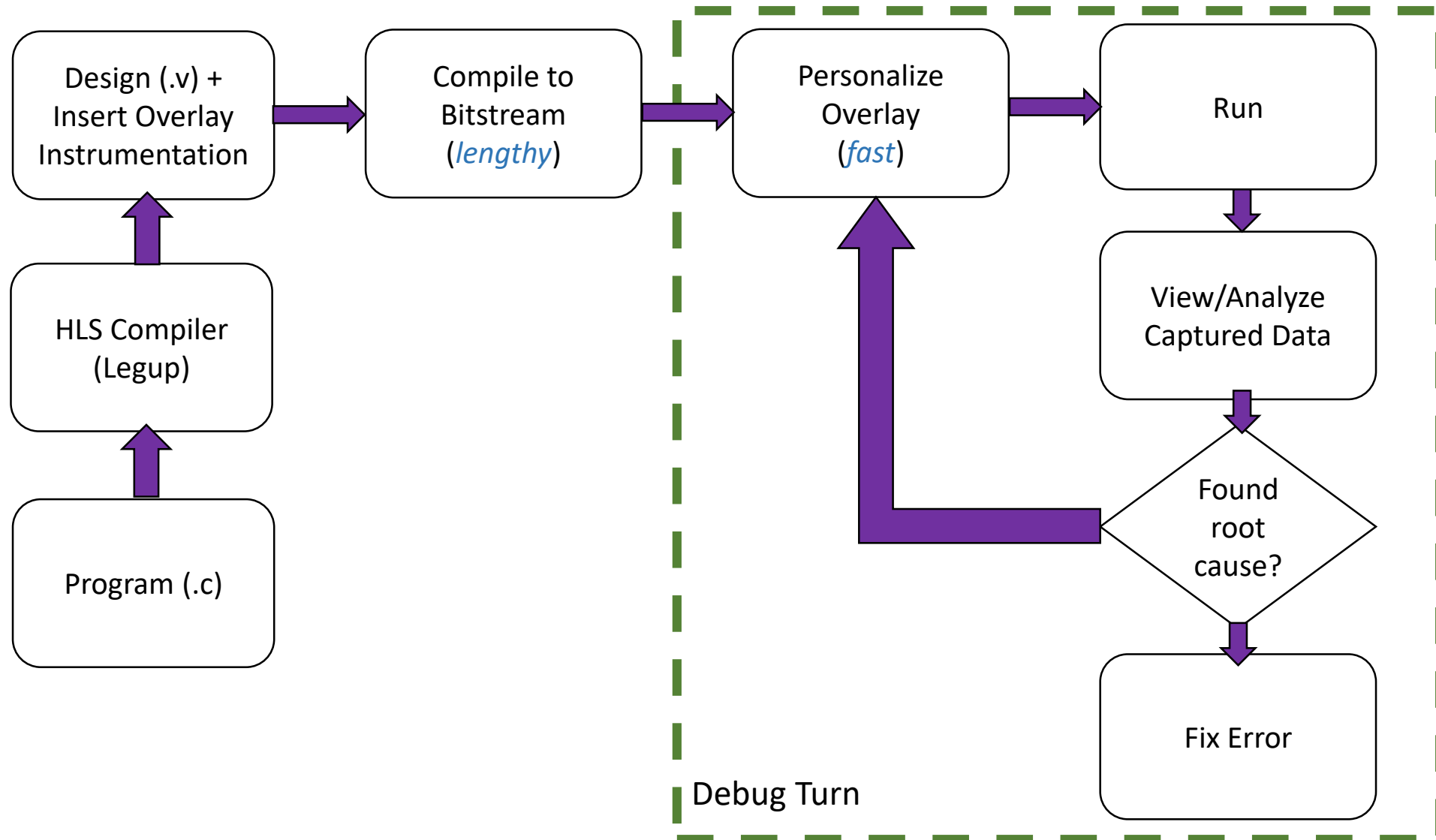
# Outline

- Motivation for In-System Debug
- Previous Work: In-System Debug Framework for HLS
  - Debug Instrumentation at compile time
- This paper: HLS Debug Overlay to allow customization at runtime
- Evaluation
- Future Work

# HLS Overlays: Software-like Debug Turn-Around Times



# Workflow Using the Debug Overlay



**Key:** The more general/flexible the overlay – the larger the area overhead

**Our Approach:** determine a set of **useful capabilities**, and architect an overlay that is *just flexible enough* to implement these

# What can this overlay do?

**Our approach:** determine a set of **useful capabilities**, and architect an overlay that is *just flexible enough* to implement these.

1. Selective Variable Tracing
  - Select user visible variables to trace
2. Selective Function Tracing
  - Select region of code to trace
3. Conditional Buffer Freeze
  - Specify a condition on the circuit that, when true, causes recording in the trace buffer to halt.

# Selective Variable Tracing: User Perspective

The screenshot displays the HLS Debugger interface for a SHA256 implementation. The top section shows the execution mode as "FPGA Replay Execution" and the design state as "Paused". The function being traced is "sha256\_transform" at FSM State #12. The main window is split into three panes: source code, a timing diagram, and a selective variable tracing table.

**Source Code (sha256\_labeled.c):**

```
74 }
75 for (; i < 64; ++i) {
76     s = m[i - 2];
77     sig1 = ROTRIGHT(s, 17);
78     sig1 ^= ROTRIGHT(s, 19);
79     sig1 ^= s >> 10;
80
81     s = m[i - 15];
82     sig0 = ROTRIGHT(s, 7);
83     sig0 ^= ROTRIGHT(s, 18);
84     sig0 ^= s >> 3;
85
86     temp = sig1;
87     temp ^= m[i - 7];
88     temp += sig0;
89     temp += m[i - 16];
90     n[i] = temp;
91 }
92
93 a = ctx->state[0];
94 b = ctx->state[1];
95 c = ctx->state[2];
96 d = ctx->state[3];
97 e = ctx->state[4];
98 f = ctx->state[5];
99 g = ctx->state[6];
100 h = ctx->state[7];
101
102 for (i = 0; i < 64; ++i) {
103     ep0 = ROTRIGHT(a, 2);
104     ep0 ^= ROTRIGHT(a, 13);
105     ep0 ^= ROTRIGHT(a, 22);
106     ep1 = ROTRIGHT(e, 6);
107     ep1 ^= ROTRIGHT(e, 11);
108     ep1 ^= ROTRIGHT(e, 25);
109     ch = (e & f) ^ (~e & g);
110     naj = (a & b) ^ (a & c) ^ (b & c);
111     t1 = h + ep1 + ch + k[i] + m[i];
112     t2 = ep0 + maj;
113
114 }
```

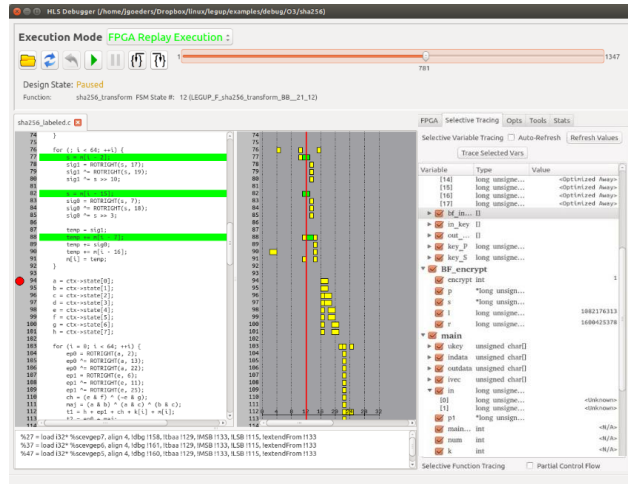
**Selective Variable Tracing Table:**

Variable	Type	Value
[14]	long unsigne...	<Optimized Away>
[15]	long unsigne...	<Optimized Away>
[16]	long unsigne...	<Optimized Away>
[17]	long unsigne...	<Optimized Away>
bf_in...	[]	
in_key	[]	
out_...	[]	
key_P	long unsigne...	
key_S	long unsigne...	
BF_encrypt	encrypt int	1
p	*long unsign...	
s	*long unsign...	
l	long unsigne...	1082176313
r	long unsigne...	1600425378
main		
ukey	unsigned char[]	
indata	unsigned char[]	
outdata	unsigned char[]	
ivec	unsigned char[]	
in	long unsigne...	
[0]	long unsigne...	<Unknown>
[1]	long unsigne...	<Unknown>
p1	*long unsign...	
main...	int	<N/A>
num	int	<N/A>
k	int	<N/A>

Select/de-select variables from pane in Debug GUI



# Architecture to Support Capability

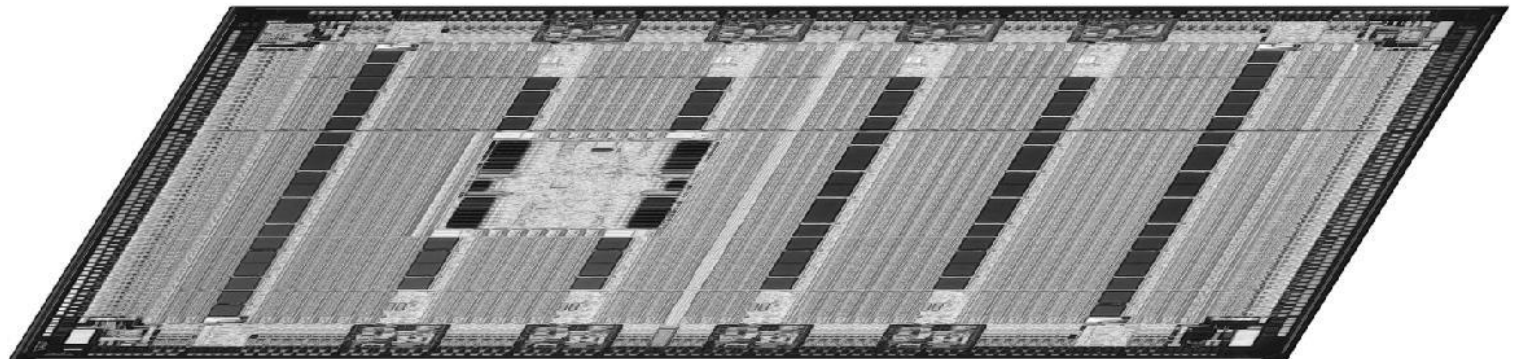


Debug Scenario

between debug iterations (fast)



once, at compile time (slow)



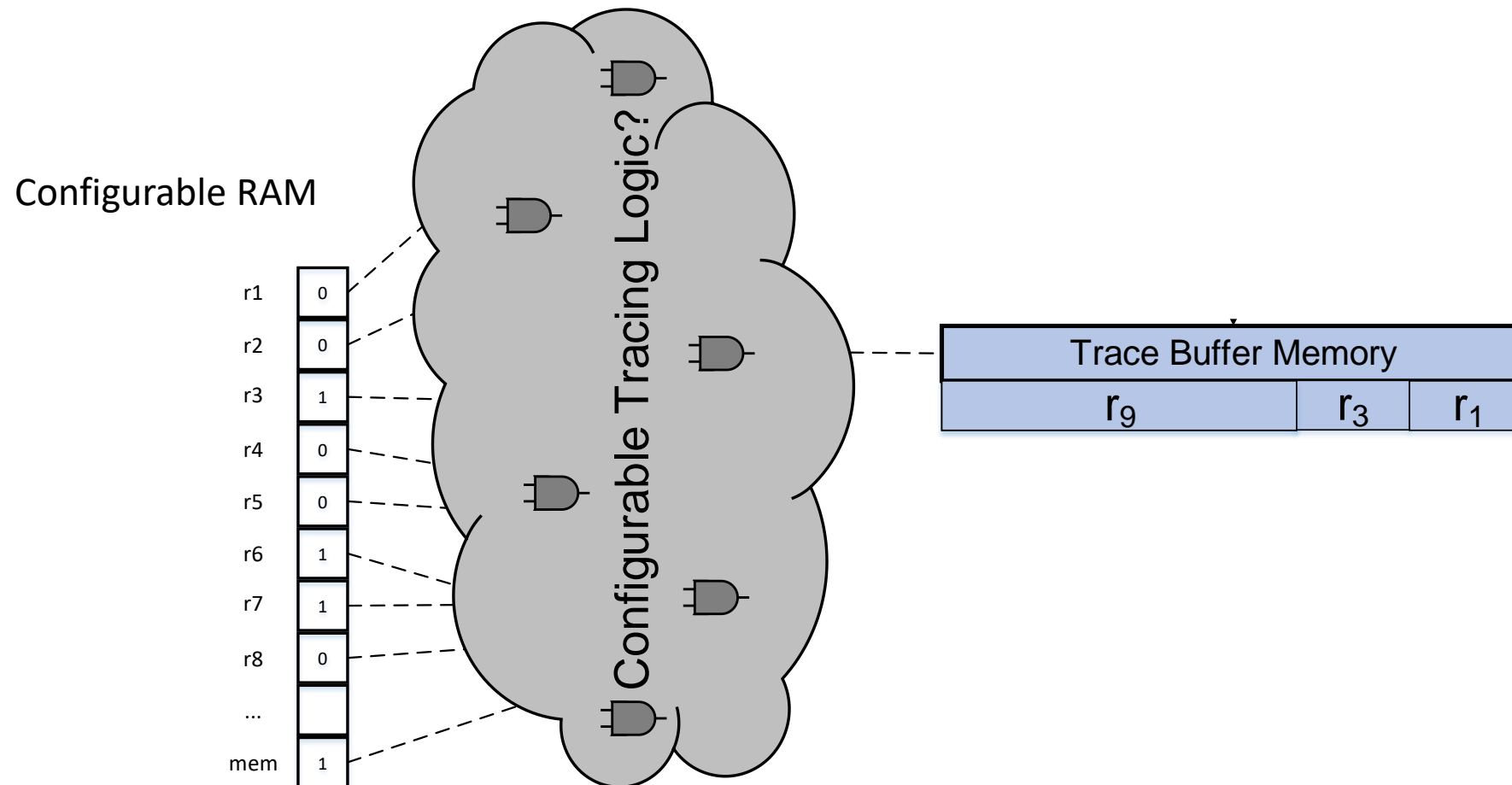
# Selective Variable Tracing Architecture – Initial Ideas...

Could have a configurable memory that enables which RTL signals (that map to C code variables) we want to trace. Program this memory at runtime...

Aside: Intel's In-System Memory Content Editor

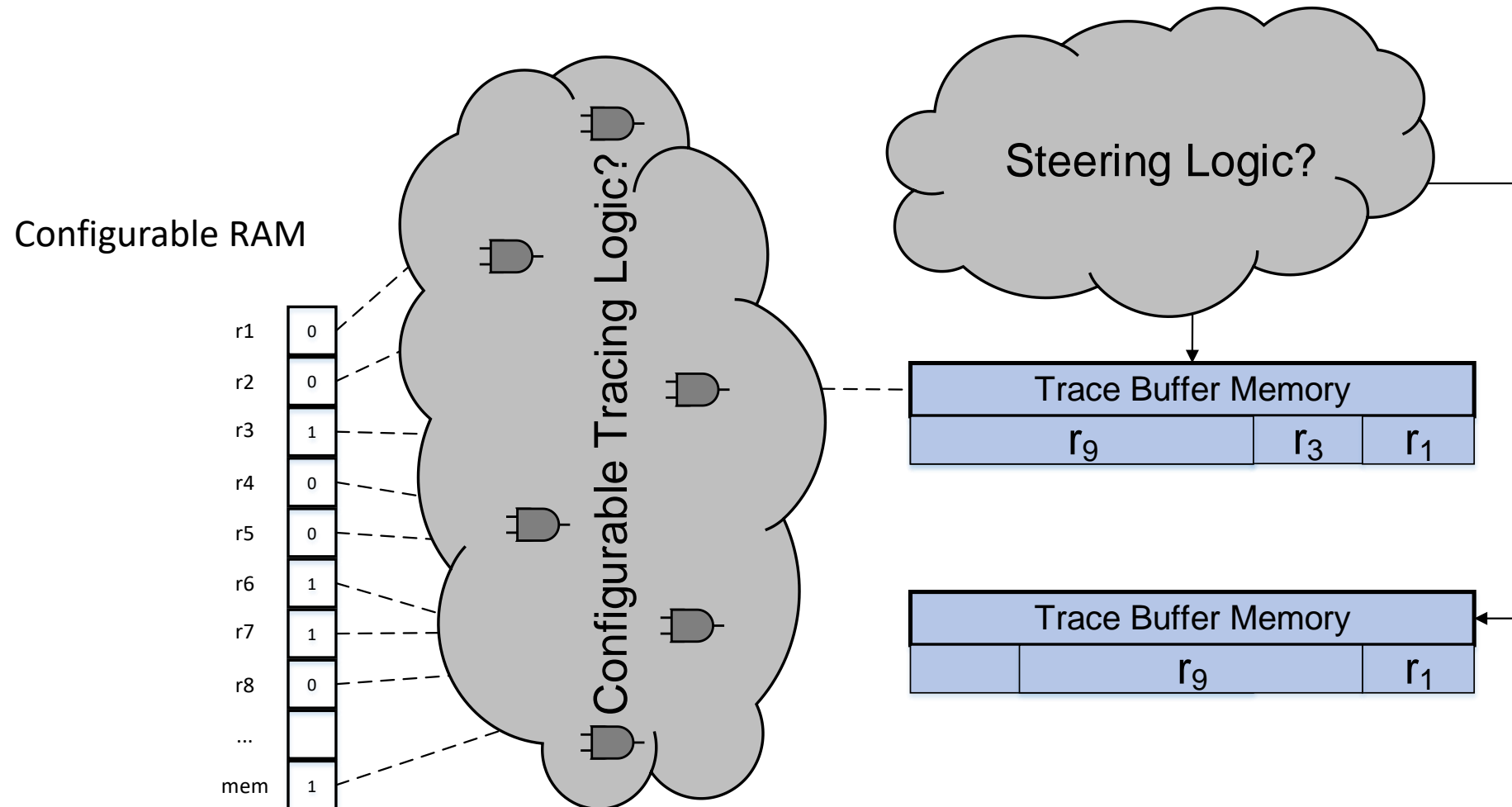
# Selective Variable Tracing Architecture – Initial Ideas...

Could associate a bit in Config RAM with each RTL signal that corresponds to a C code variable...



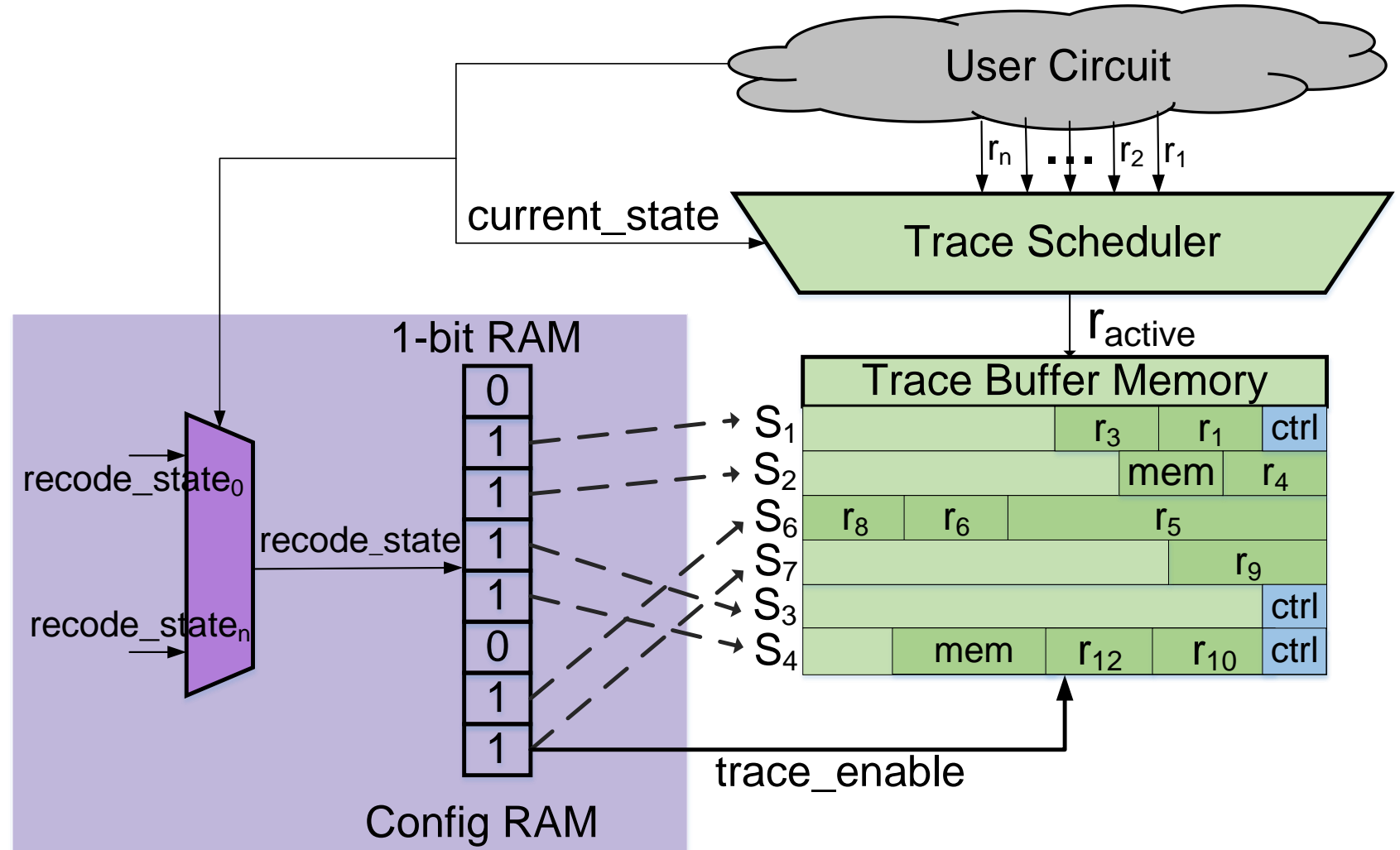
# Selective Variable Tracing Architecture – Initial Ideas...

Could associate a bit in Config RAM with each RTL signal that corresponds to a C code variable...

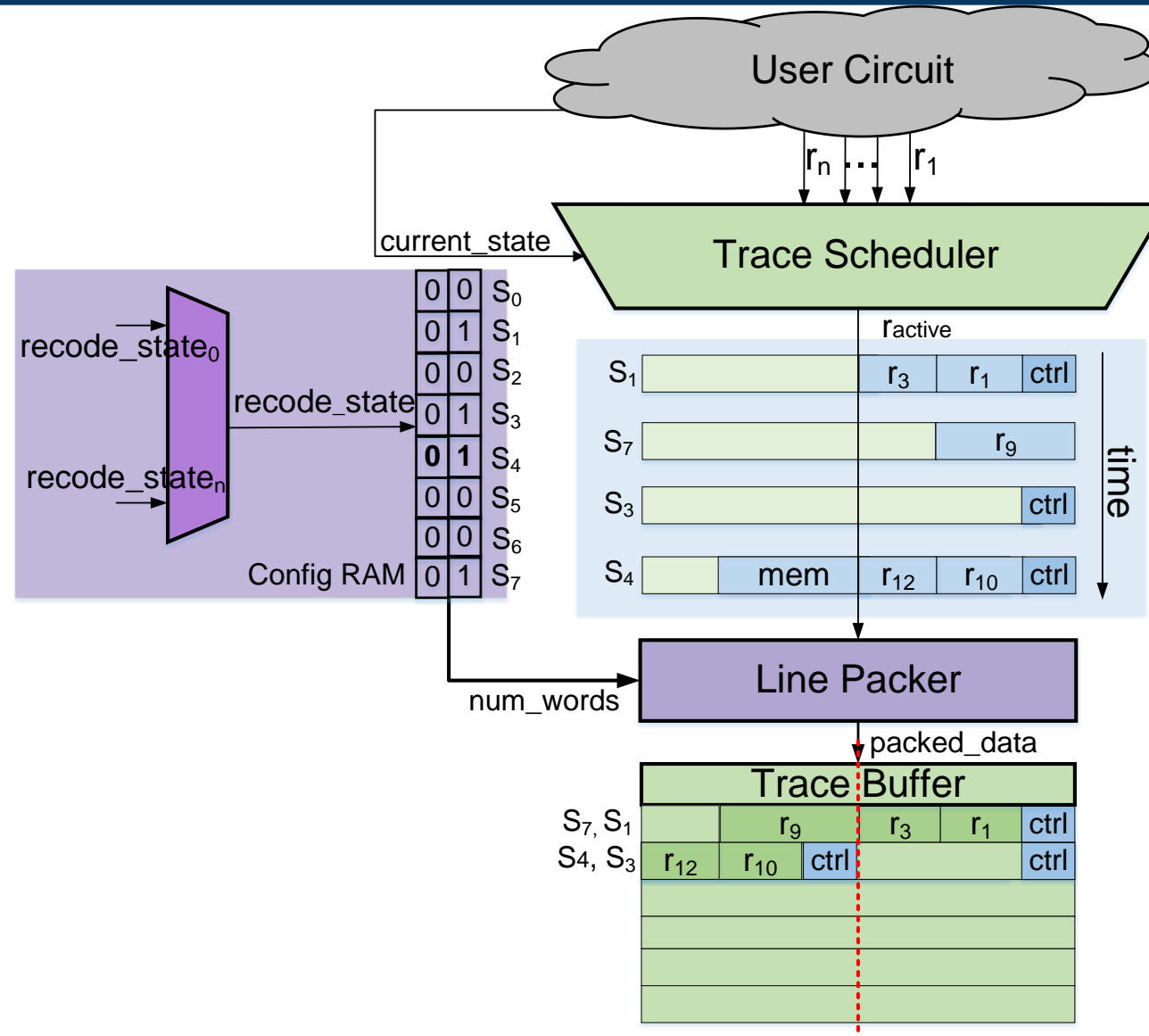


# Selective Variable Tracing Architecture: Variant A

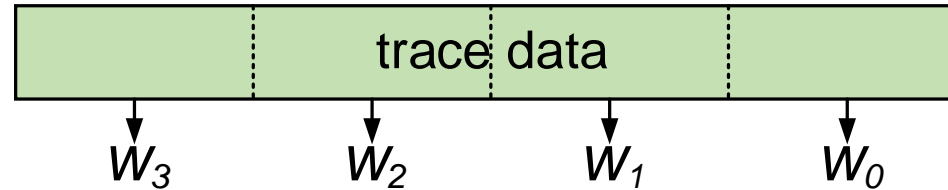
**Key:** Every *bit* is associated with a *state* in the user circuit



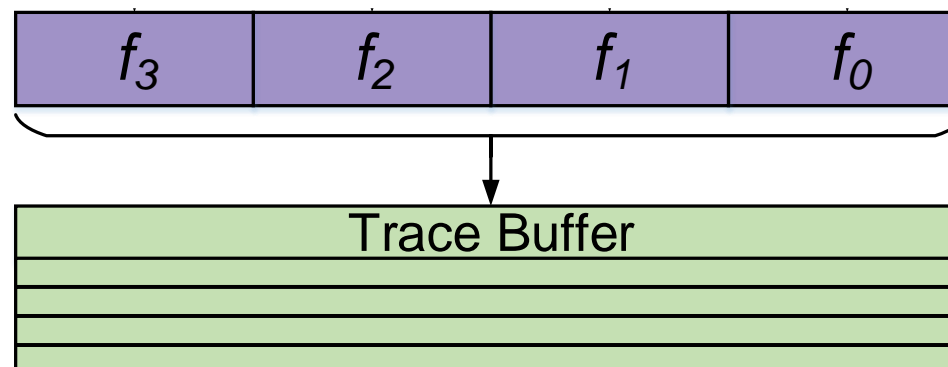
# Selective Variable Tracing Architecture: Variant B



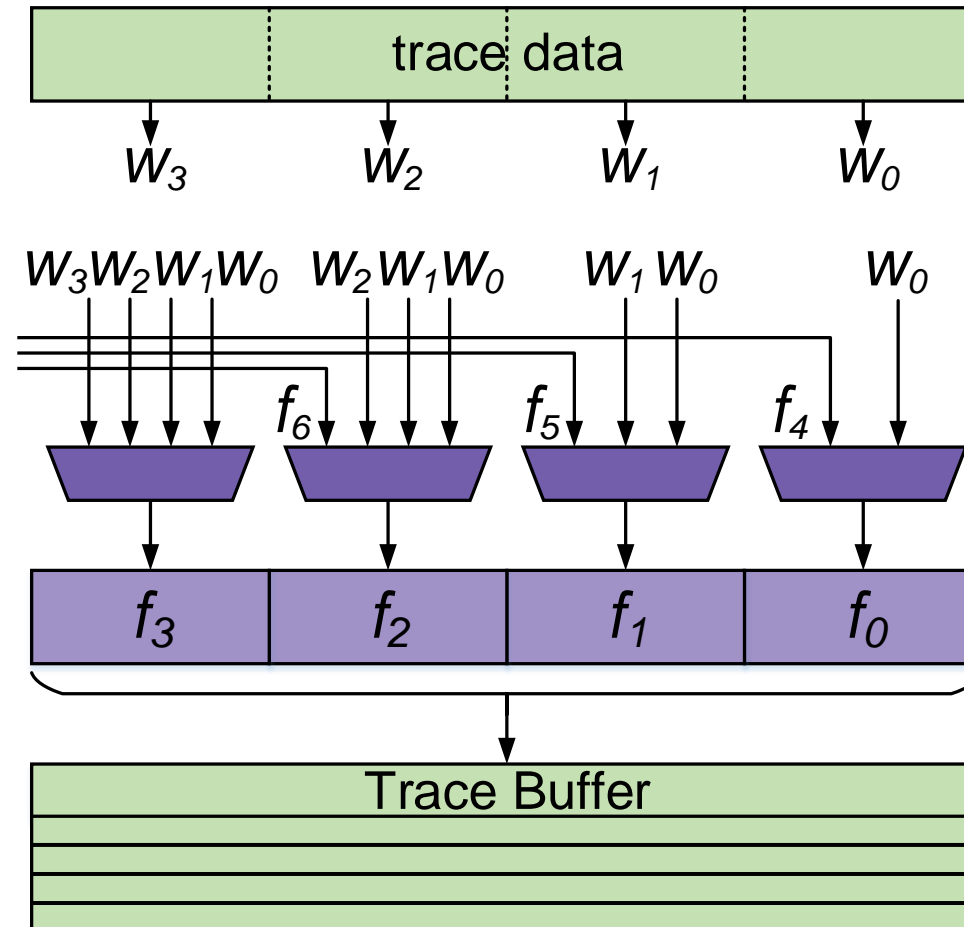
# Variant B: Line Packer – Architectural Parameter “G”



- $G$ : *granularity*
- Increasing  $G$  splits the incoming trace data into smaller words – more fine grained packing
- Increasing  $G$  also increases the steering logic/area overhead



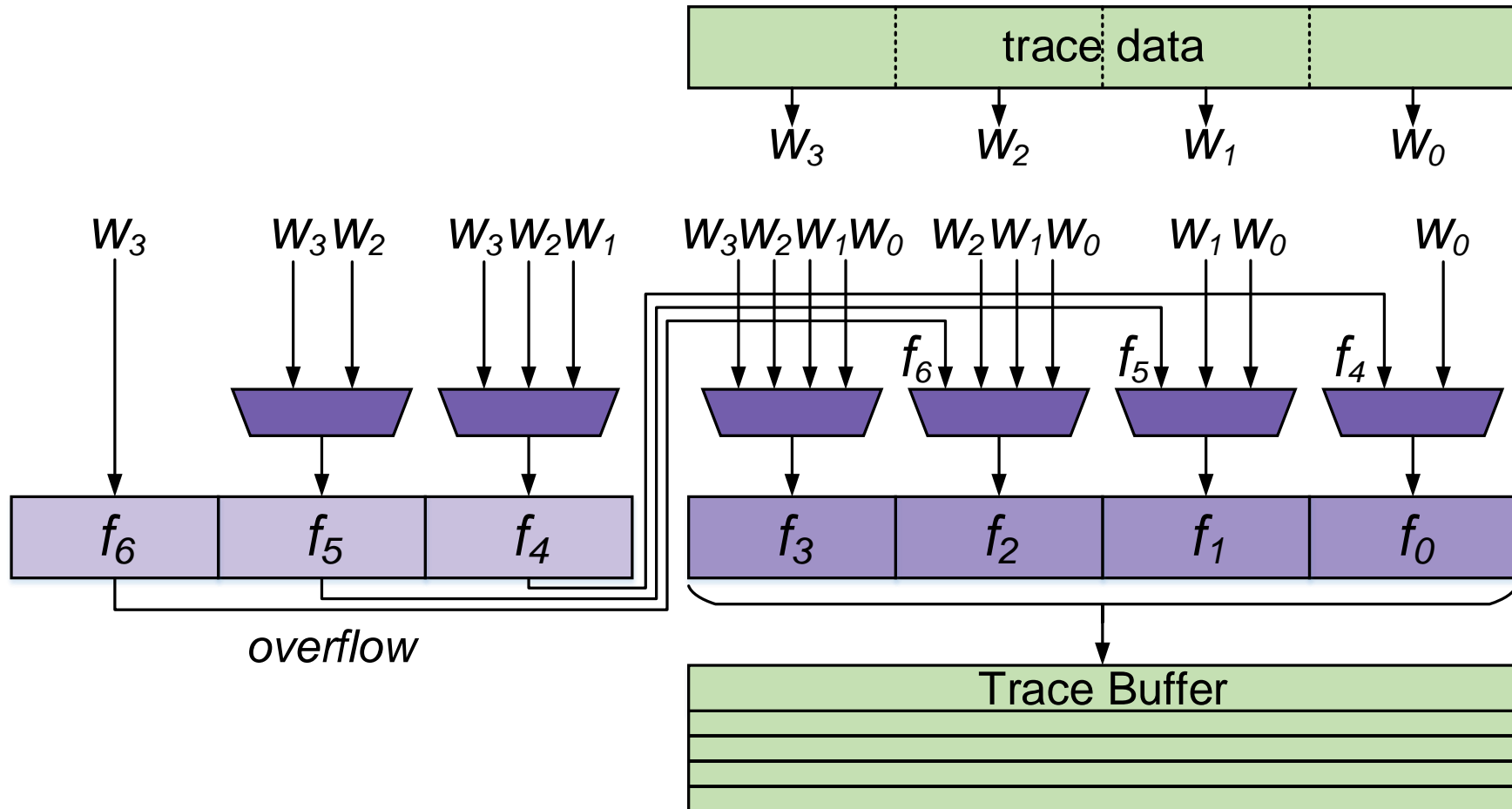
# Variant B: Line Packer – Architectural Parameter “G”



- $G$ : *granularity*
- Increasing  $G$  splits the incoming trace data into smaller words – more fine grained packing
- Increasing  $G$  also increases the steering logic/area overhead

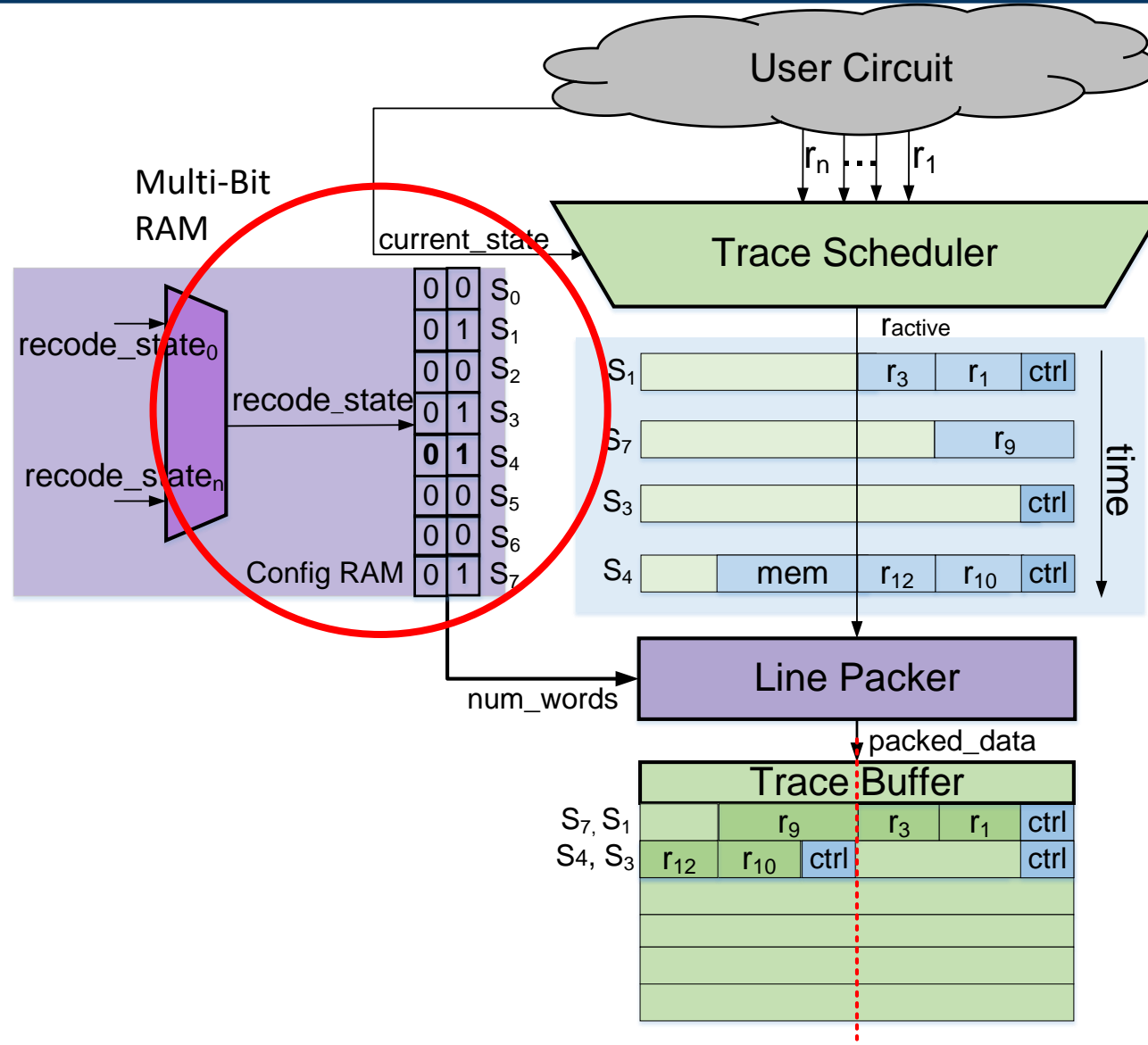


# Variant B: Line Packer – Architectural Parameter “G”



- $G$ : *granularity*
- Increasing  $G$  splits the incoming trace data into smaller words – more fine grained packing
- Increasing  $G$  also increases the steering logic/area overhead

# Variant B – Multi-Bit Configuration ROM



# Selective Function Tracing: User Perspective

The screenshot displays the HLS Debugger interface for a SHA256 implementation. The top bar shows the execution mode as "FPGA Replay Execution" and the design state as "Paused". The function being traced is "sha256\_transform" at FSM State #12. The main window is split into three panes: source code, a timing diagram, and a selective tracing table.

**Source Code (sha256\_labeled.c):**

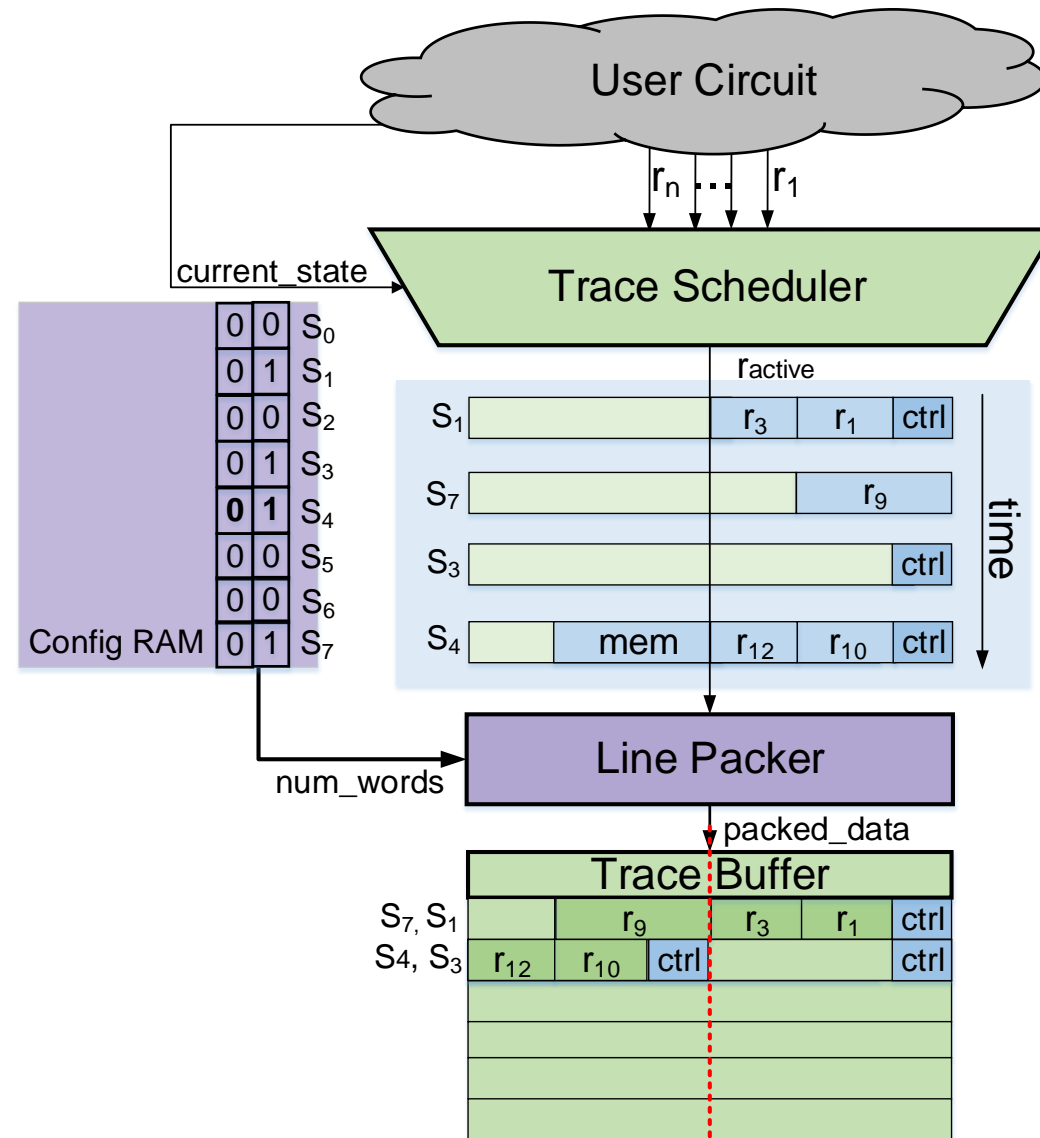
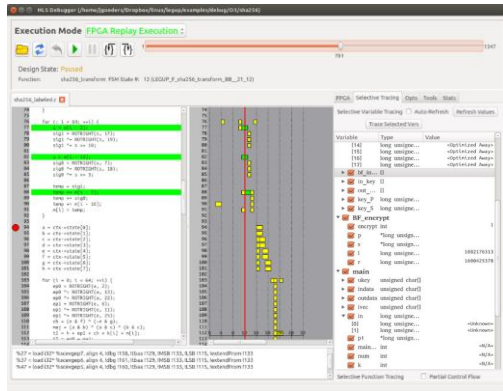
```
74 }
75 for (; i < 64; ++i) {
76     s = m[i - 2];
77     sig1 = ROTRIGHT(s, 17);
78     sig1 ^= ROTRIGHT(s, 19);
79     sig1 ^= s >> 10;
80
81     s = m[i - 15];
82     sig0 = ROTRIGHT(s, 7);
83     sig0 ^= ROTRIGHT(s, 18);
84     sig0 ^= s >> 3;
85
86     temp = sig1;
87     temp ^= m[i - 1];
88     temp += sig0;
89     temp += m[i - 16];
90     n[i] = temp;
91 }
92
93 a = ctx->state[0];
94 b = ctx->state[1];
95 c = ctx->state[2];
96 d = ctx->state[3];
97 e = ctx->state[4];
98 f = ctx->state[5];
99 g = ctx->state[6];
100 h = ctx->state[7];
101
102 for (i = 0; i < 64; ++i) {
103     ep0 = ROTRIGHT(a, 2);
104     ep0 ^= ROTRIGHT(a, 13);
105     ep0 ^= ROTRIGHT(a, 22);
106     ep1 = ROTRIGHT(e, 6);
107     ep1 ^= ROTRIGHT(e, 11);
108     ep1 ^= ROTRIGHT(e, 25);
109     ch = (e & f) ^ (~e & g);
110     naj = (a & b) ^ (a & c) ^ (b & c);
111     t1 = h + ep1 + ch + k[i] + m[i];
112     t2 = ep0 + maj;
113
114 }
```

**Selective Tracing Table:**

Variable	Type	Value
[14]	long unsigne...	<Optimized Away>
[15]	long unsigne...	<Optimized Away>
[16]	long unsigne...	<Optimized Away>
[17]	long unsigne...	<Optimized Away>
bf_in...	[]	
in_key	[]	
out_...	[]	
key_P	long unsigne...	
key_S	long unsigne...	
BF_encrypt		
encrypt	int	1
p	*long unsign...	
s	*long unsign...	
l	long unsigne...	1082176313
r	long unsigne...	1600425378
main		
ukey	unsigned char[]	
indata	unsigned char[]	
outdata	unsigned char[]	
ivec	unsigned char[]	
in	long unsigne...	
[0]	long unsigne...	<Unknown>
[1]	long unsigne...	<Unknown>
p1	*long unsign...	
main...	int	<N/A>
num	int	<N/A>
k	int	<N/A>

Select Functions from pane in Debug GUI

# Selective Function Tracing: Same architecture!



# Conditional Buffer Freeze – User Perspective

Execution Mode: FPGA Replay Execution

Design State: Paused

Function: sha256\_transform FSM State #: 12 (LEGUP\_F\_sha256\_transform\_BB\_\_21\_12)

sha256\_labeled.c

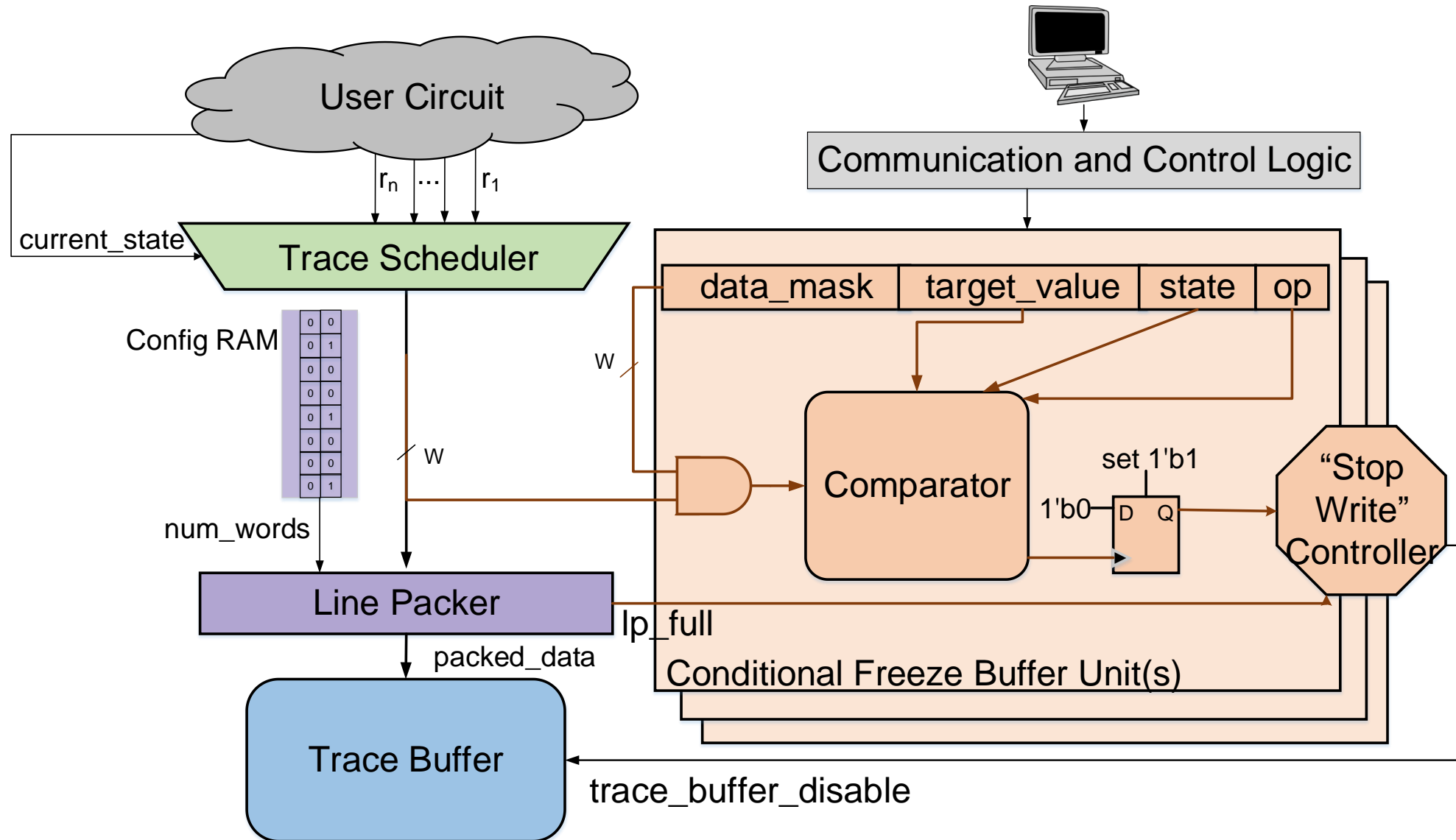
```
74 }
75
76 for (; i < 64; ++i) {
77     s = m[i - 2];
78     sig1 = ROTRIGHT(s, 17);
79     sig1 ^= ROTRIGHT(s, 19);
80     sig1 ^= s >> 10;
81
82     s = m[i - 15];
83
84
85
86
87
88
89
90
91
92
93
94     a = ctx->state[0];
95     b = ctx->state[1];
96     c = ctx->state[2];
97     d = ctx->state[3];
98     e = ctx->state[4];
99     f = ctx->state[5];
100    g = ctx->state[6];
101    h = ctx->state[7];
102
103    for (i = 0; i < 64; ++i) {
104        ep0 = ROTRIGHT(a, 2);
105        ep0 ^= ROTRIGHT(a, 13);
106        ep0 ^= ROTRIGHT(a, 22);
107        ep1 = ROTRIGHT(e, 6);
108        ep1 ^= ROTRIGHT(e, 11);
109        ep1 ^= ROTRIGHT(e, 25);
110        ch = (e & f) ^ (~e & g);
111        maj = (a & b) ^ (a & c) ^ (b & c);
112        t1 = h + ep1 + ch + k[i] + m[i];
113        t2 = ep0 + maj;
114
```

Condition

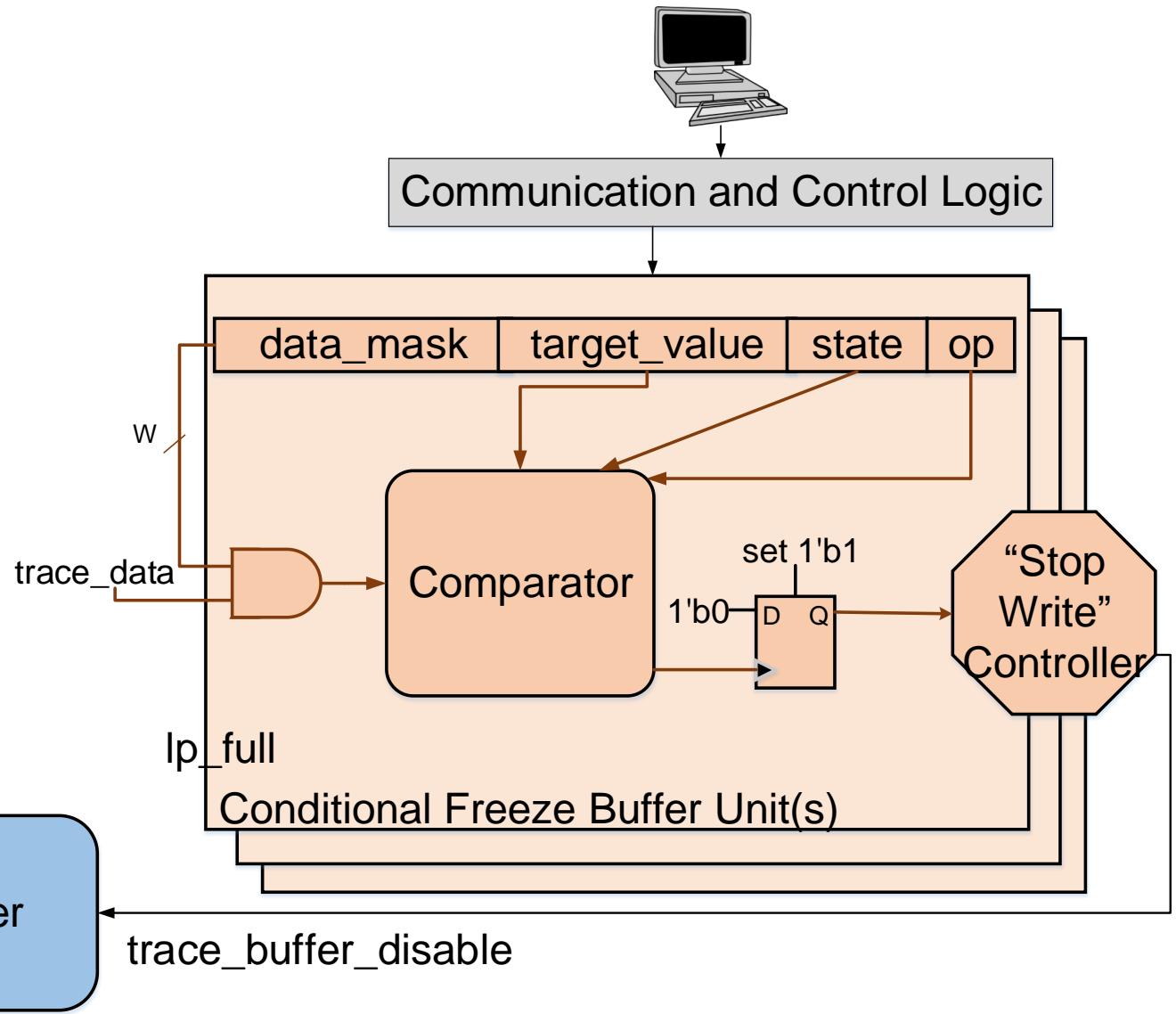
a < 0, line 94

74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114

# Conditional Buffer Freeze



# Conditional Buffer Freeze – Architectural Parameter “C”



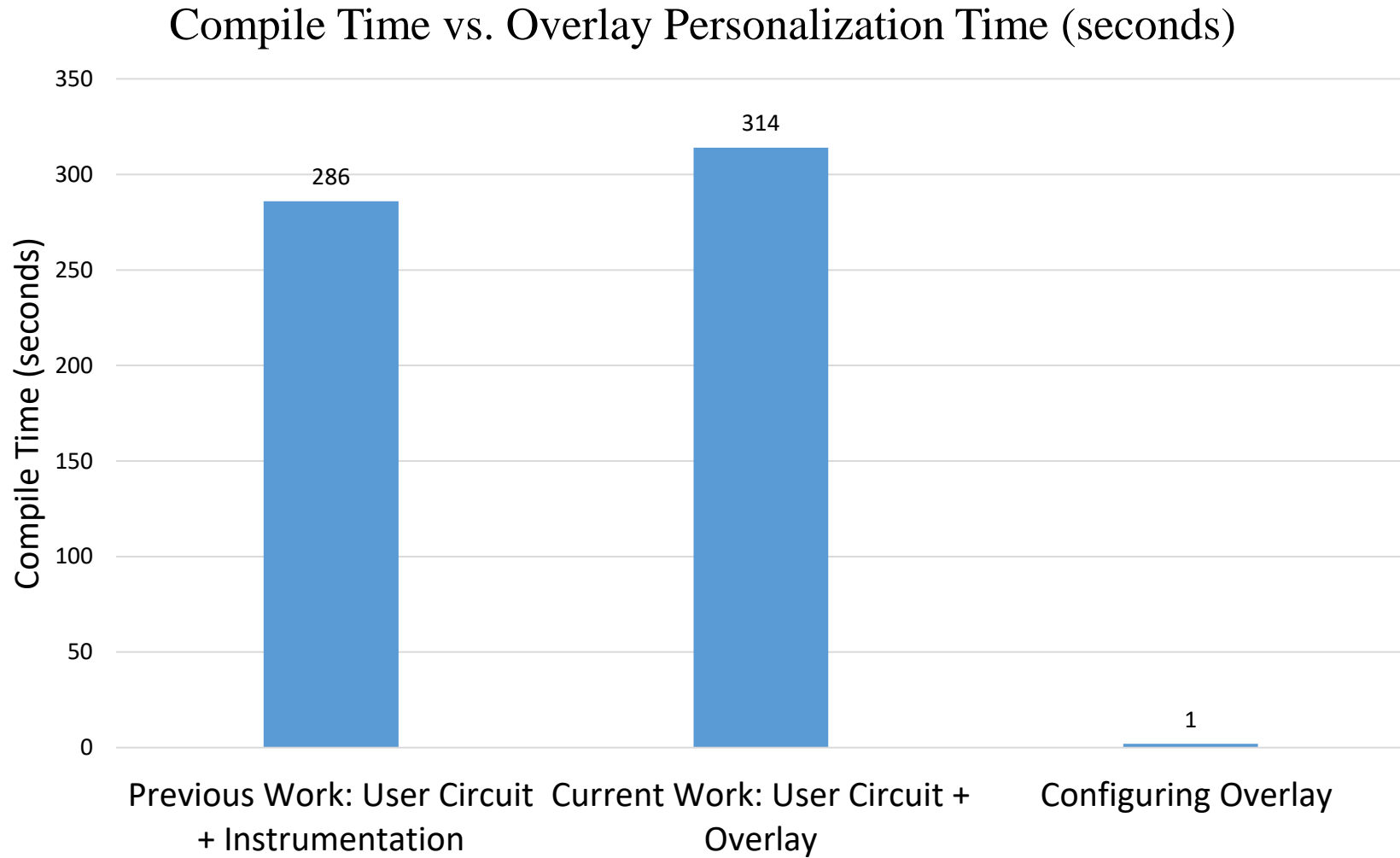
- Increase C units to express a more complex condition
- Example: Stop tracing when *err flag 1* OR *err flag 2* goes high
- “Stop write controller” receives signals from all C units – OR trigger function

# Outline

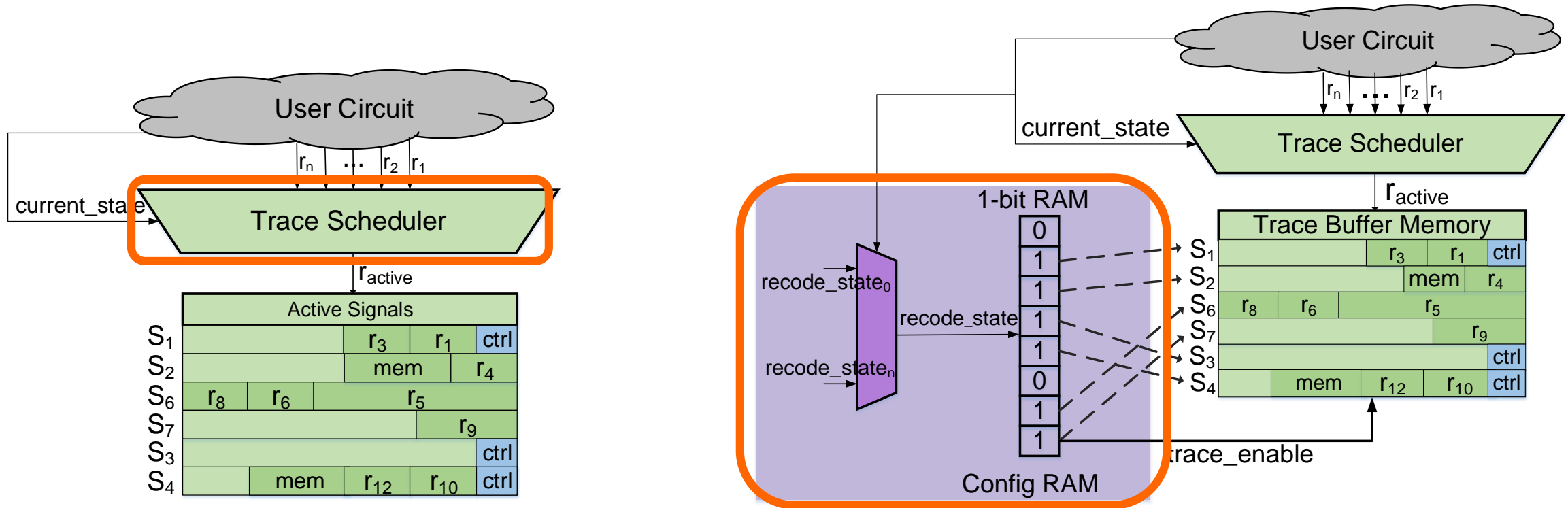
- Motivation for In-System Debug
- Previous Work: In-System Debug Framework for HLS
  - Debug Instrumentation at compile time
- This paper: HLS Debug Overlay to allow customization at runtime
- Evaluation
- Future Work



# Evaluation: Run-Times



# Variant A Overlay – Impact on Area

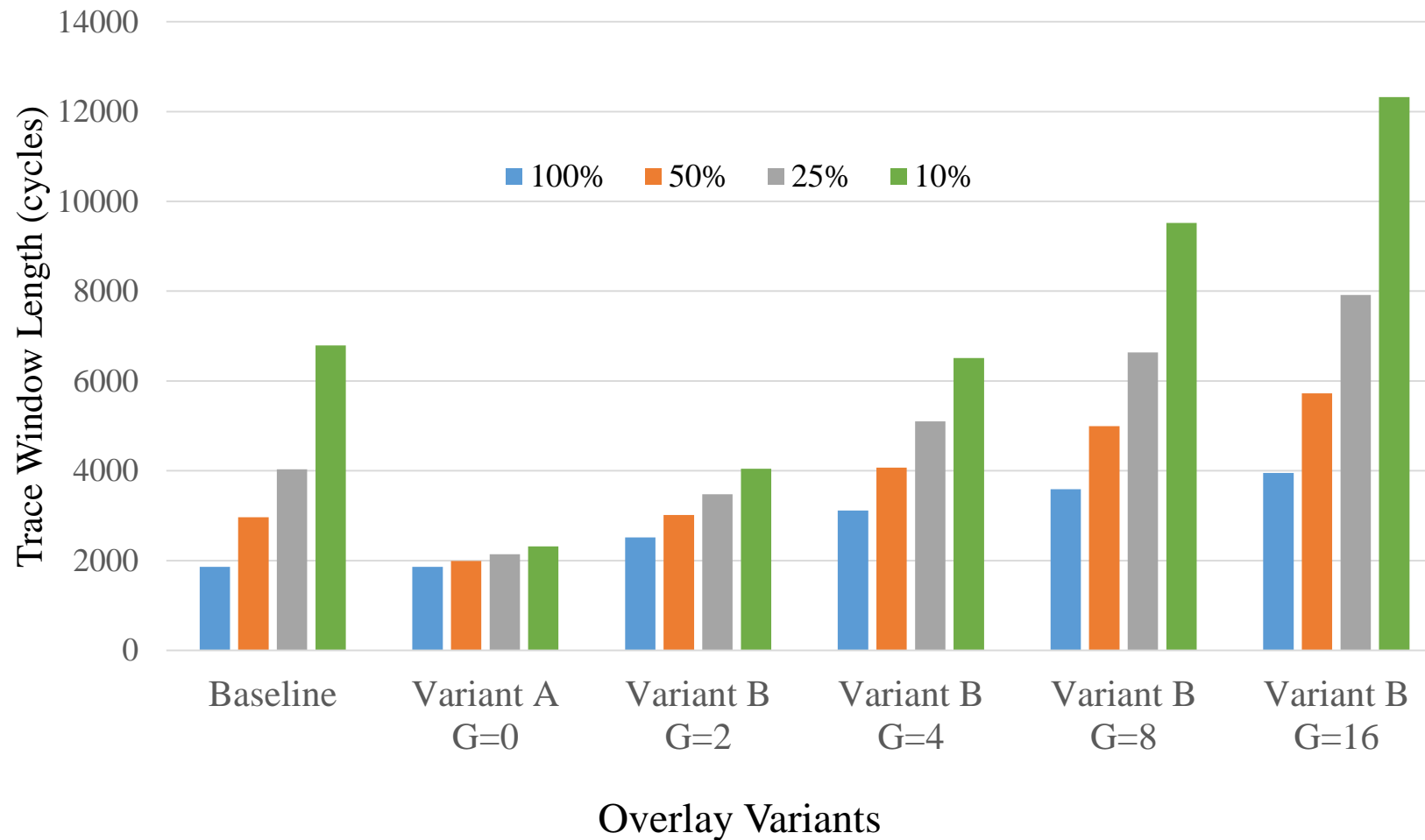


Baseline debug instrumentation is 20% size of the user circuit\*

Variant A increases the size by 39 ALMs on average, and 1 M9K – cheap!

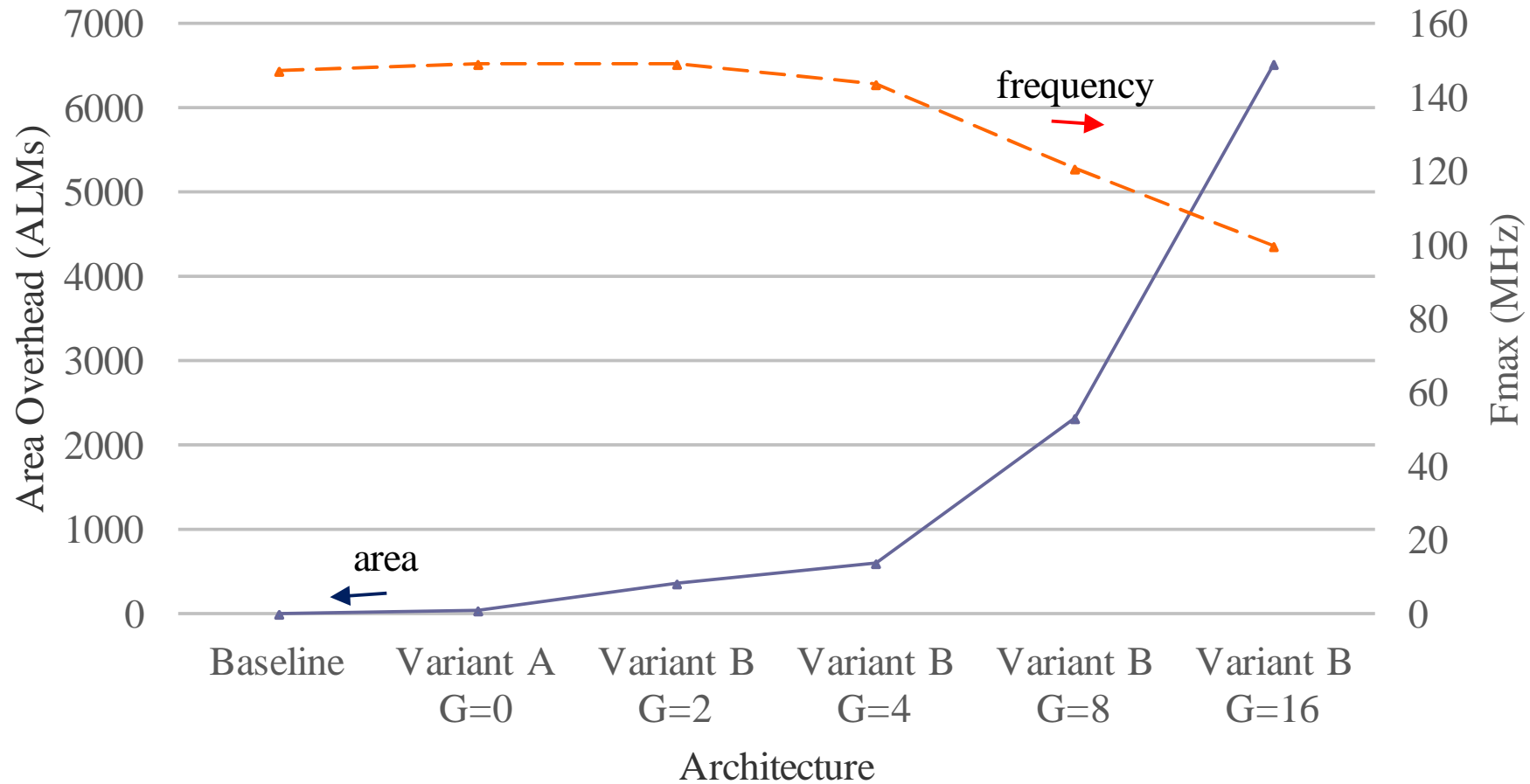
\*Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits". IEEE TCAD 2017. J Goeders, SJE Wilton.

# Architecture vs. Trace Window Length



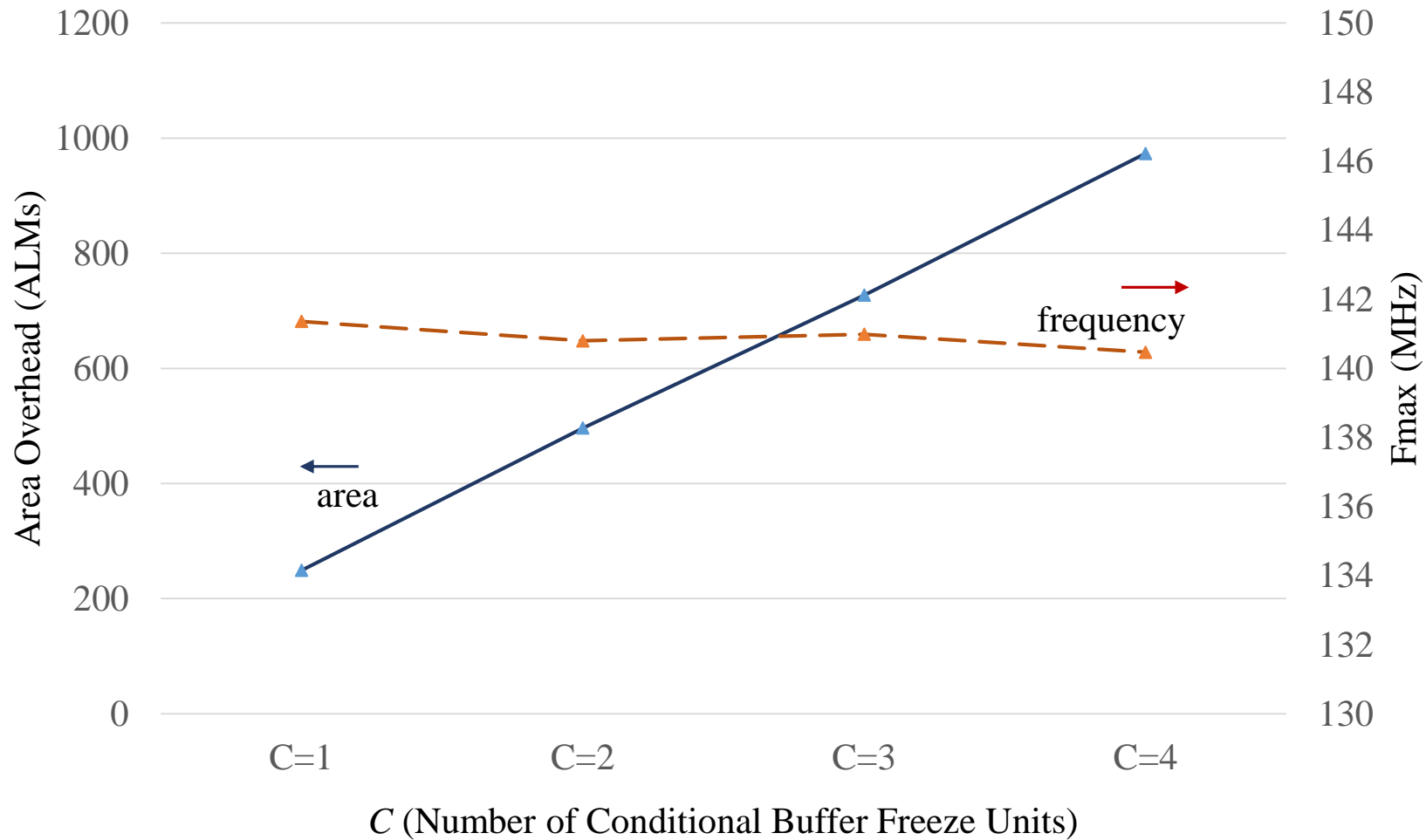
Architectural enhancements improve trace window length

# Overhead: Variant B vs. Variant A



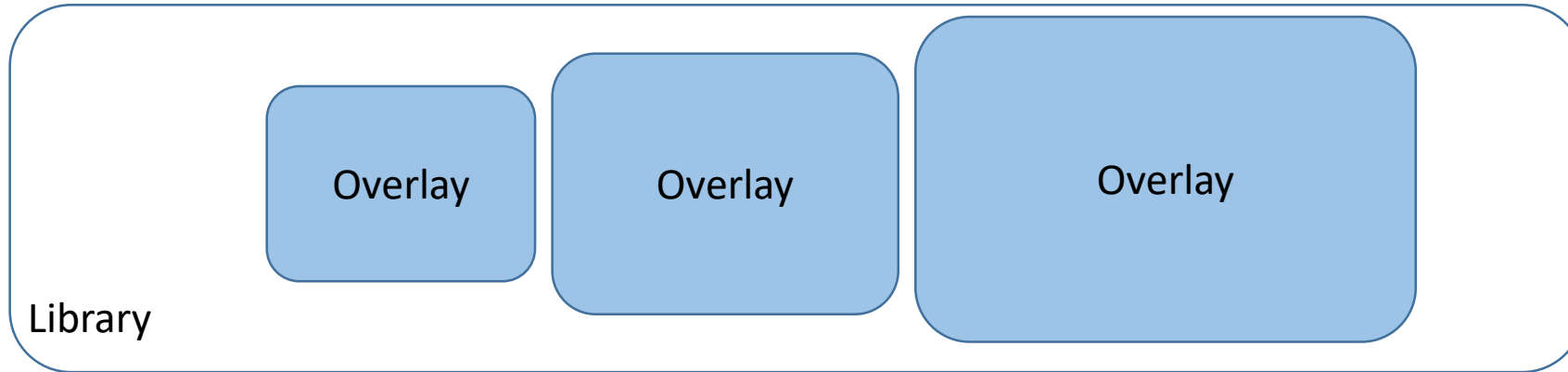
Area goes up dramatically for high granularity in line packer

# Overhead: Conditional Units



Area increases with number of C units with small decrease in Fmax

# How can a FPGA vendor use these results?



Provide a library of overlays.

Depending on the user's debugging needs, and resources available – select appropriate library:

- **Economy Library:** cheaper overlay (i.e. only selective variable tracing)
- **Deluxe Library:** supports more capabilities (i.e. conditional trigger functions)

Can also take advantage of:

- User input / estimates to user
- Variable reconstruction

# Outline

- Motivation for In-System Debug
- Previous Work: In-System Debug Framework for HLS
  - Debug Instrumentation at compile time
- This paper: HLS Debug Overlay to allow customization at runtime
- Evaluation
- Future Work

# Future Work

**Currently, the user selects the overlay + capabilities to insert.**

- Next step – create a tool that automatically determines the type of overlay to insert based on *estimated unused resources*

**The overlay is passive (i.e. only *monitors* the user circuit)**

- Investigate limited *controllability*
- Allow for simple “what if” scenarios



# Summary

**Achieved software like compile times between debug turns in a limited context via an HLS oriented overlay**

- Can personalize the overlay at runtime without a recompile
- Overlay supports a set of capabilities (selective variable/function tracing, conditional buffer freeze)
- Overheads are significant (335 ALMs for Variant B/G=2 line packer, 249 ALMs for C=1 unit) on top of the Baseline instrumentation

**Worth it for the option to have software like compile times during debug**

**Thank you**

**Additional**

# Previous Work – Instrumentation Overhead

Circuit	User Module (ALMs)	Instrumentation (100%)		Proportion in Debug Partition
		Fixed hlsd (ALMs)	Trace Scheduler (ALMs)	
adpcm	7019	480	1749	24.1%
aes	7135	479	754	14.7%
blowfish	3038	528	1187	36.1%
dfadd	3605	495	1115	30.9%
dfdiv	6000	532	1124	21.6%
dfmul	1881	483	675	38.1%
dfsin	11864	529	2904	22.4%
gsm	4147	473	782	23.2%
jpeg	18735	506	2781	14.9%
mips	1441	505	419	39.1%
motion	6470	520	524	13.9%
sha	1720	514	334	33.0%
combined	66522	583	13525	17.5%
<b>Mean</b>	<b>10736</b>	<b>509</b>	<b>2114</b>	<b>25.4%</b>

Roughly ¼ is debug instrumentation.