Typed Architectures

Architectural Support for Lightweight Scripting

이재욱 (jaewlee@snu.ac.kr)
서울대학교 컴퓨터공학부

SIGPL 여름 워크샵
2017. 8. 9

* Typed Architecture team: 김찬노, 김성민, 김재혁, 김남호, 나기태, 김두영, 오영환, 장학범, 조현규

Motivation (1): Today's Scripting Languages

- **Already widely used in various application domains**
  - JavaScript: Web clients and servers
  - Lua: Game programming
  - R: Statistical computing, data analytics
  - Python, PHP, Perl, Ruby, MATLAB, etc.

- **Becoming general-purpose programming platforms**
  - Example: HTML5/JavaScript-based apps

Motivation (2): Today's Scripting Languages

- **(+) High productivity**
  - Dynamic type systems: flexible and extensible
  - High level of abstraction with powerful built-in functions
  - Object-oriented programming paradigm
  - Automatic memory management (e.g., garbage collection)

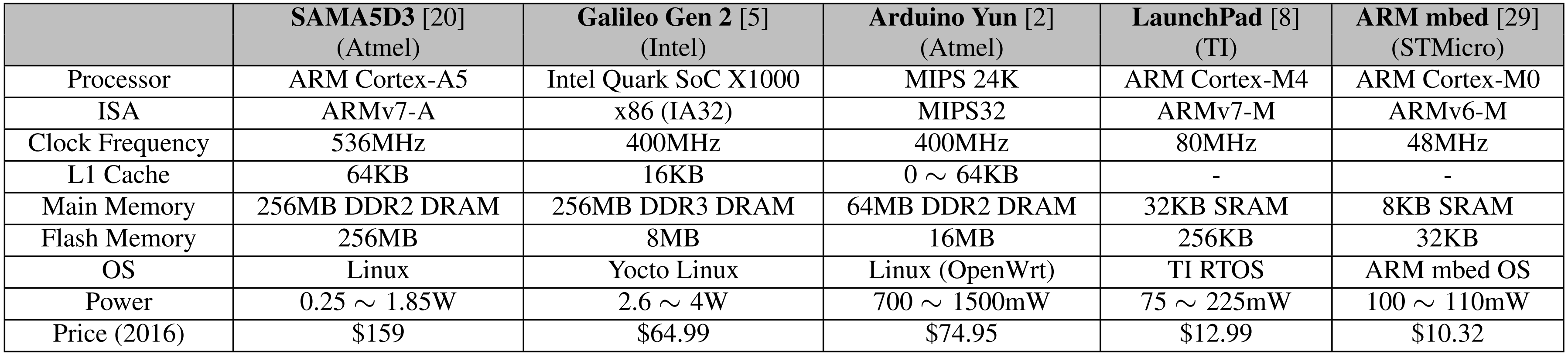
- **(–) Low efficiency**
  - Primarily due to dynamic type systems
  - Example: usage of polymorphic "**+**" operation
  - Increasing instruction count and memory footprint

```
function add(a, b) { return a + b; }

add(1, 2);     // (NUMBER::INT)    3
add(1.1, 2.2); // (NUMBER::DOUBLE) 3.3
add("a", "b"); // (STRING)         "ab"
```

Motivation (3): Scripting on Emerging IoT Platforms

- **Emerging single-board computers for so-called DIY electronics**
  - Arduino, Raspberry Pi, Intel Edison/Galileo, Samsung ARTIK, etc.
  - Platforms for emerging IoT applications: low cost, low power, small form factor
- **Scripting languages are too heavyweight for those platforms**
  - Heavily resource constrained
    - Single-core, in-order pipeline
    - Low MHz
    - Small memory

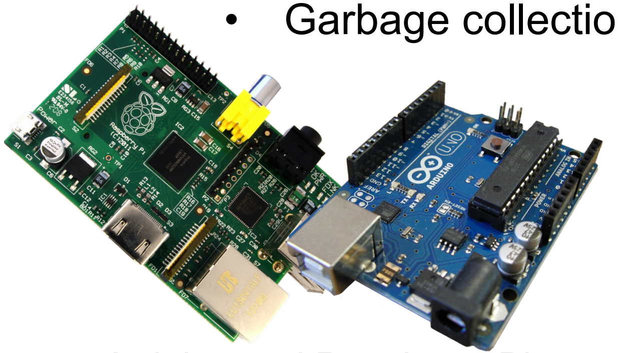


	SAMA5D3 [20] (Atmel)	**Galileo Gen 2** [5] (Intel)	**Arduino Yun** [2] (Atmel)	**LaunchPad** [8] (TI)	**ARM mbed** [29] (STMicro)
Processor	ARM Cortex-A5	Intel Quark SoC X1000	MIPS 24K	ARM Cortex-M4	ARM Cortex-M0
ISA	ARMv7-A	x86 (IA32)	MIPS32	ARMv7-M	ARMv6-M
Clock Frequency	536MHz	400MHz	400MHz	80MHz	48MHz
L1 Cache	64KB	16KB	0 ∼ 64KB	-	-
Main Memory	256MB DDR2 DRAM	256MB DDR3 DRAM	64MB DDR2 DRAM	32KB SRAM	8KB SRAM
Flash Memory	256MB	8MB	16MB	256KB	32KB
OS	Linux	Yocto Linux	Linux (OpenWrt)	TI RTOS	ARM mbed OS
Power	0.25 ∼ 1.85W	2.6 ∼ 4W	700 ∼ 1500mW	75 ∼ 225mW	100 ∼ 110mW
Price (2016)	$159	$64.99	$74.95	$12.99	$10.32

* Source: Kim et al., Typed Architectures: Architectural Support for Lightweight Scripting, ASPLOS 2017.

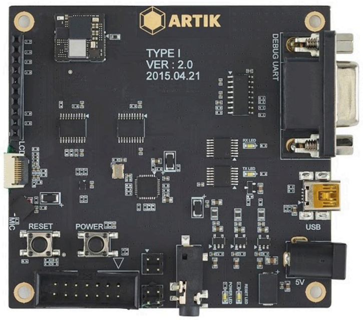
Motivation (4): Scripting Languages + Single-Board Computers

- **Productivity benefits for IoT programming**
  - Ease of programming and testing
  - Natural support for event-driven programming models
  - Seamless client-server integration (e.g., using HTML5/JavaScript)
- **But, too slow on IoT platforms**
  - **JIT compilation:** not viable due to severe resource constraints
  - **VM interpreter:** wastes CPU cycles for
    - Recurring cost of bytecode dispatch **[ISCA'16]**
    - Dynamic type checking **[ASPLOS'17]** **Today's focus**
    - Boxing/unboxing objects
    - Garbage collection



Arduino and Raspberry Pi

Intel Galileo and Edison



Samsung ARTIK

Related Research Activities @ SNU ARC

- **Our position: C will become the new assembly language!**

  (and >90% of the code will be written in higher-level languages)

- **Parallel mobile web browser @ stealth-mode start-up (2009-2011)**
- **Script optimization in SW (2012-2014)**
  - Input-aware specialization of scripting engines **[ASPLOS'13]**
  - CPU-GPU cooperative execution for data parallel JavaScript **[WWW'14] [PPoPP'15]**
  - Towards greener web: QoS-aware energy optimization for JavaScript **[ISLPED'14]**
- **Novel CPU architectures for scripting languages (2015-present)**
  - Reducing bytecode dispatch overhead **[ISCA'16]**
  - Reducing type checking overhead **[ASPLOS'17]**
  - Exploiting bytecode-level parallelism on multicores
  - Reducing GC overhead
  - etc.

들어가기 전에...

- **The rest of this talk is based on the following ASPLOS'17 paper**
  - Channoh Kim, Jaehyeok Kim, Sungmin Kim, Dooyoung Kim, Namho Kim, Gitae Na, Young H. Oh, Hyeon Gyu Cho, and Jae W. Lee, "Typed Architectures: Architectural Support for Lightweight Scripting", *22nd ACM Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Xi'an, China, April 2017.

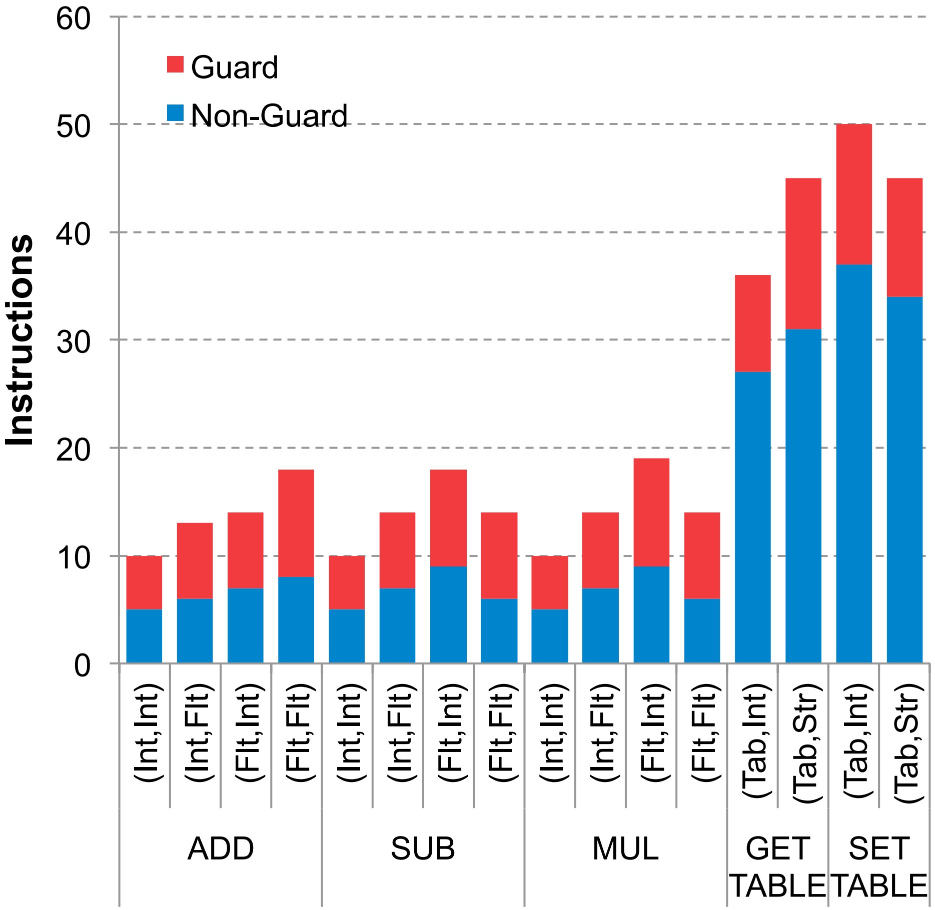
- **CAVEAT: This is an architecture paper (not a PL paper).**
  - Not dealing with the design of the type system of a language
  - Instead, focusing on accelerating specific implementations with hardware support
  - (Hopefully) understandable if you have taken undergraduate architecture course

Dynamic Type Checking (1)

- **Significant fraction of instructions are spent executing type guards**
  - Profiled five most frequently used bytecodes in Lua

```
// from Lua interpreter

case ADD:
 Value *rb = RB(Bytecode);
 Value *rc = RC(Bytecode);
 Number nb, nc;
 if (isInt(rb) && isInt(rc)) {
   ival(ra) = ival(rb)+ival(rc);
   type(ra) = INT;
 }
 else if (toNumber (rb, &nb) &&
          toNumber (rc, &nc)) {
   fval(ra) = nb + nc;
   type(ra) = FLT;
 }
 else {
   /* do exception */
 }
```

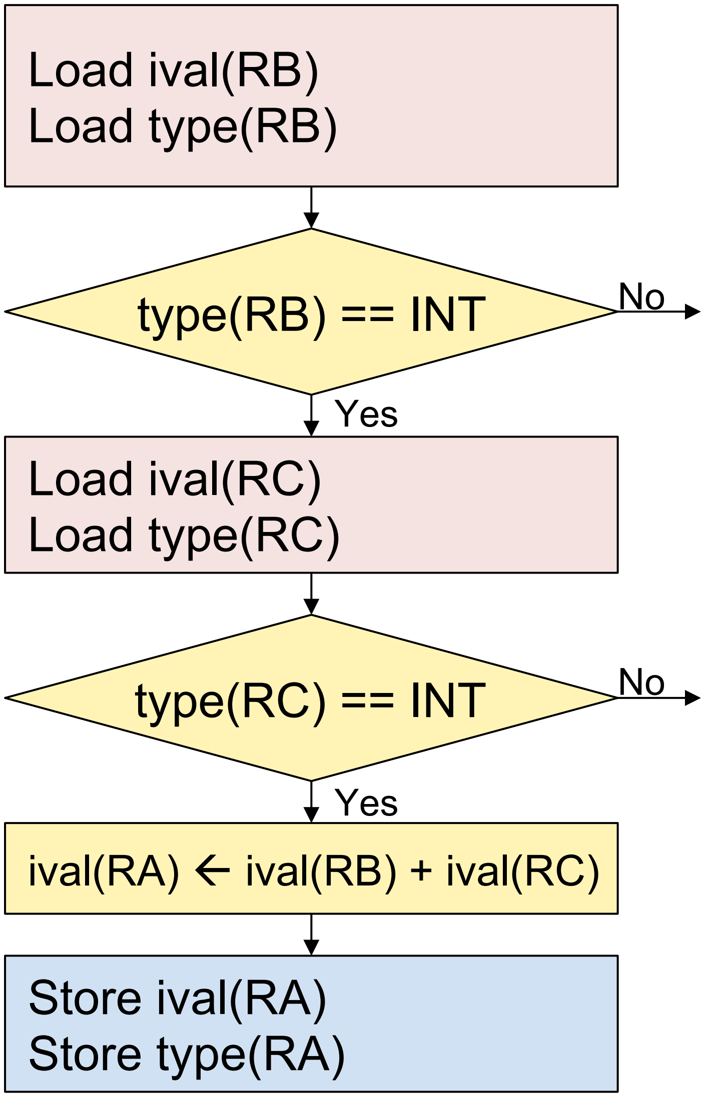


Dynamic Type Checking (2)

```
// from Lua interpreter

case ADD:
 Value *rb = RB(Bytecode);
 Value *rc = RC(Bytecode);
 Number nb, nc;
 if (isInt(rb) && isInt(rc)) {
   ival(ra) = ival(rb) + ival(rc);
   type(ra) = INT;
 }
 else if (toNumber (rb, &nb) &&
          toNumber (rc, &nc)) {
   fval(ra) = nb + nc;
   type(ra) = FLT;
 }
 else {
   /* do exception */
 }
```

Bytecode ADD



Load ival(RB)
Load type(RB)

type(RB) == INT

No

Yes

Load ival(RC)
Load type(RC)

type(RC) == INT

No

Yes

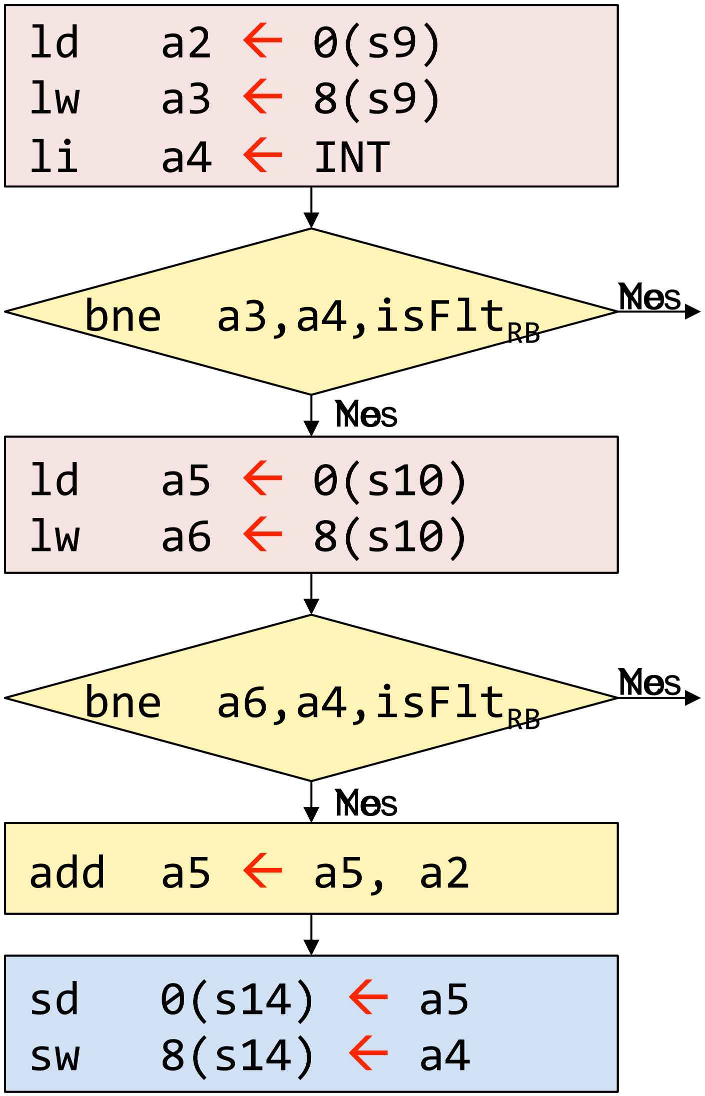
ival(RA) ← ival(RB) + ival(RC)

Store ival(RA)
Store type(RA)

Dynamic Type Checking (3)

- **Tag extraction**
  - Extraction of an operand's type tag
- **Tag checking**
  - Checking the type tags to execute the correct version of the operator
- **Tag insertion**
  - Storing the calculated value with type tag

Bytecode ADD



```
ld    a2 ← 0(s9)
lw    a3 ← 8(s9)
li    a4 ← INT
```

bne a3,a4,$isFlt_{RB}$

```
ld    a5 ← 0(s10)
lw    a6 ← 8(s10)
```

bne a6,a4,$isFlt_{RB}$

```
add   a5 ← a5, a2
```

```
sd    0(s14) ← a5
sw    8(s14) ← a4
```

Our Proposal: Typed Architectures

- **A high-efficiency, low-cost execution substrate for dynamic script languages**
  - **Key idea**: Retaining high-level type information of a variable at an ISA level
  - Dynamic type checking in parallel with value calculation

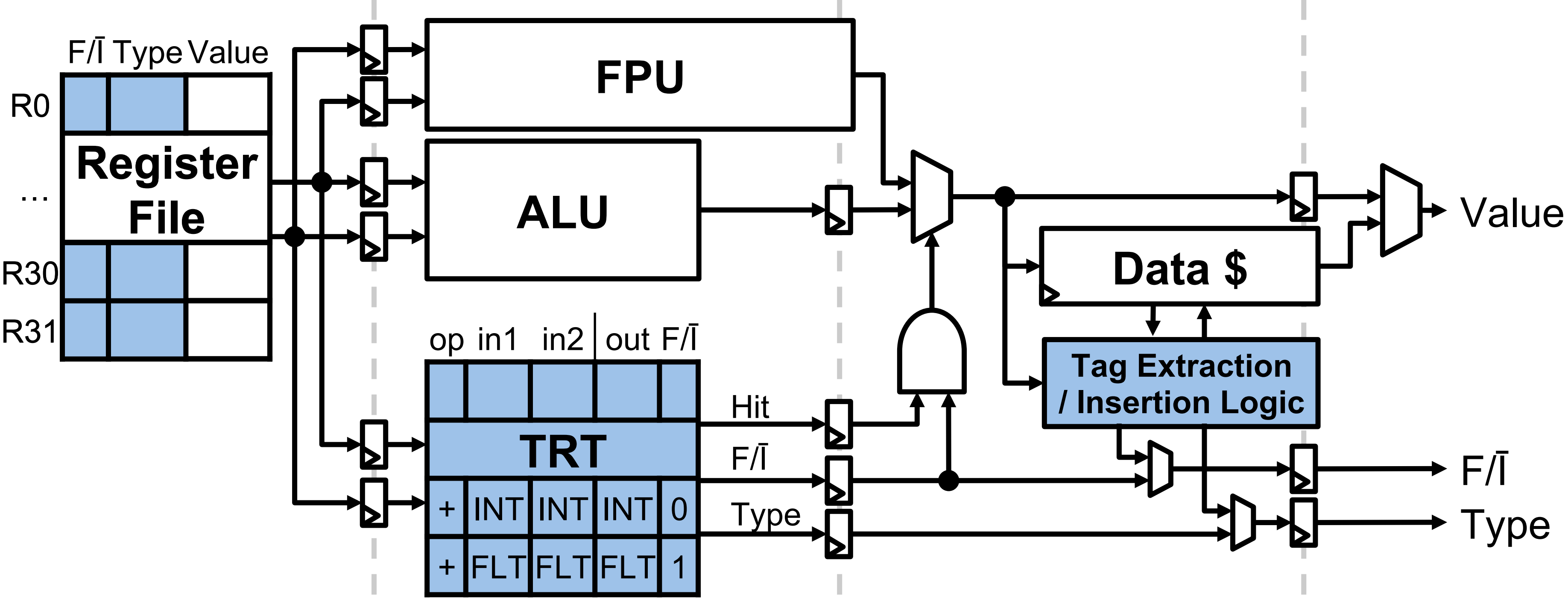
- **Key results**
  - Geomean (Max.) speedups: **14.1% (46.0%)** for Lua, **11.7% (29.9%)** for JavaScript
  - Incurs minimal hardware cost (**1.6%** area overhead)

Outline

- Motivation and key idea
- **Typed Architecture**
  - Component #1: Unified Register File
  - Component #2: Tagged ALU Instructions
  - Component #3: Tagged Memory Instructions
  - Table Access
- **Evaluation**
- **Summary**

Typed Architecture: Overview

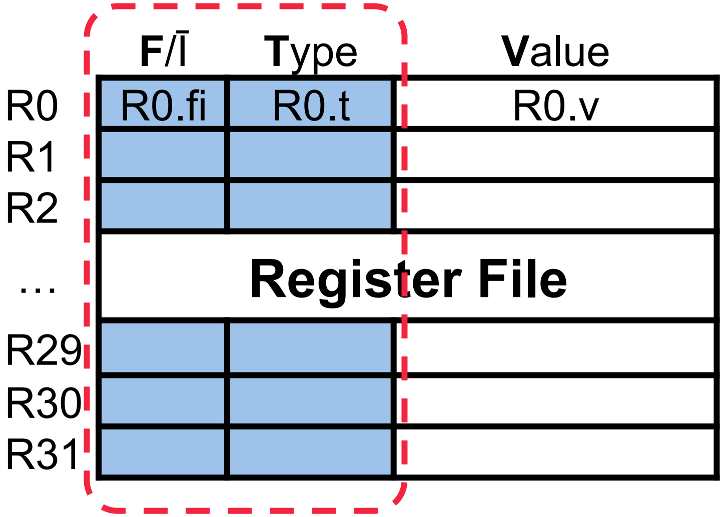
- **Extending ISA**
  - Unified register file
  - Tagged ALU instructions (Type Rule Table)
  - Tagged memory instructions (tag extraction/insertion)
- **Special-purpose registers**
  - Three registers for flexible tag extraction and insertion; one for type miss handling



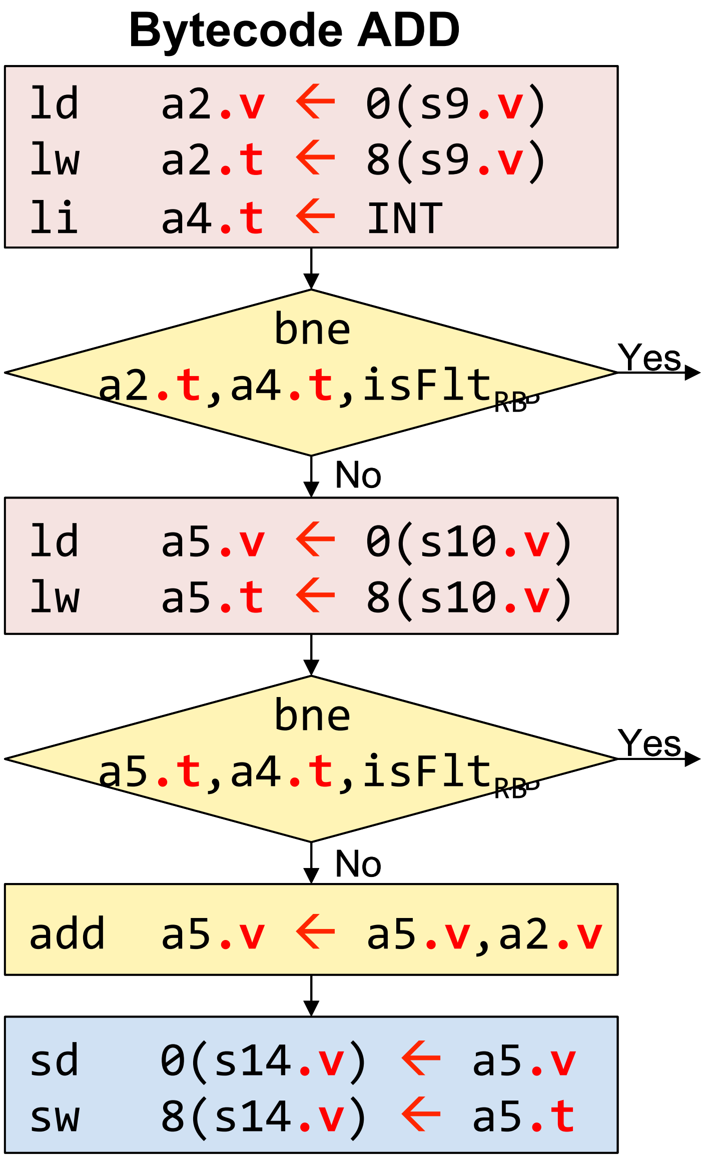
op	in1	in2	out	F/Ī
+	INT	INT	INT	0
+	FLT	FLT	FLT	1

Component #1: Unified Register File

- **Both integer and floating-point values stored in a unified register file**
- **Each entry extended with two fields**
  - **Type field** (8 bits): stores type encoding of the value
  - **F/Ī** (1 bit): indicates whether it is an **integer (0)** or **floating-point (1)** subtype



	F/Ī	Type	Value
R0	R0.fi	R0.t	R0.v
R1			
R2			
…	Register File		
R29			
R30			
R31			



Bytecode ADD

```
ld   a2.v ← 0(s9.v)
lw   a2.t ← 8(s9.v)
li   a4.t ← INT
```

```
bne
a2.t,a4.t,isFltRB
```

```
ld   a5.v ← 0(s10.v)
lw   a5.t ← 8(s10.v)
```

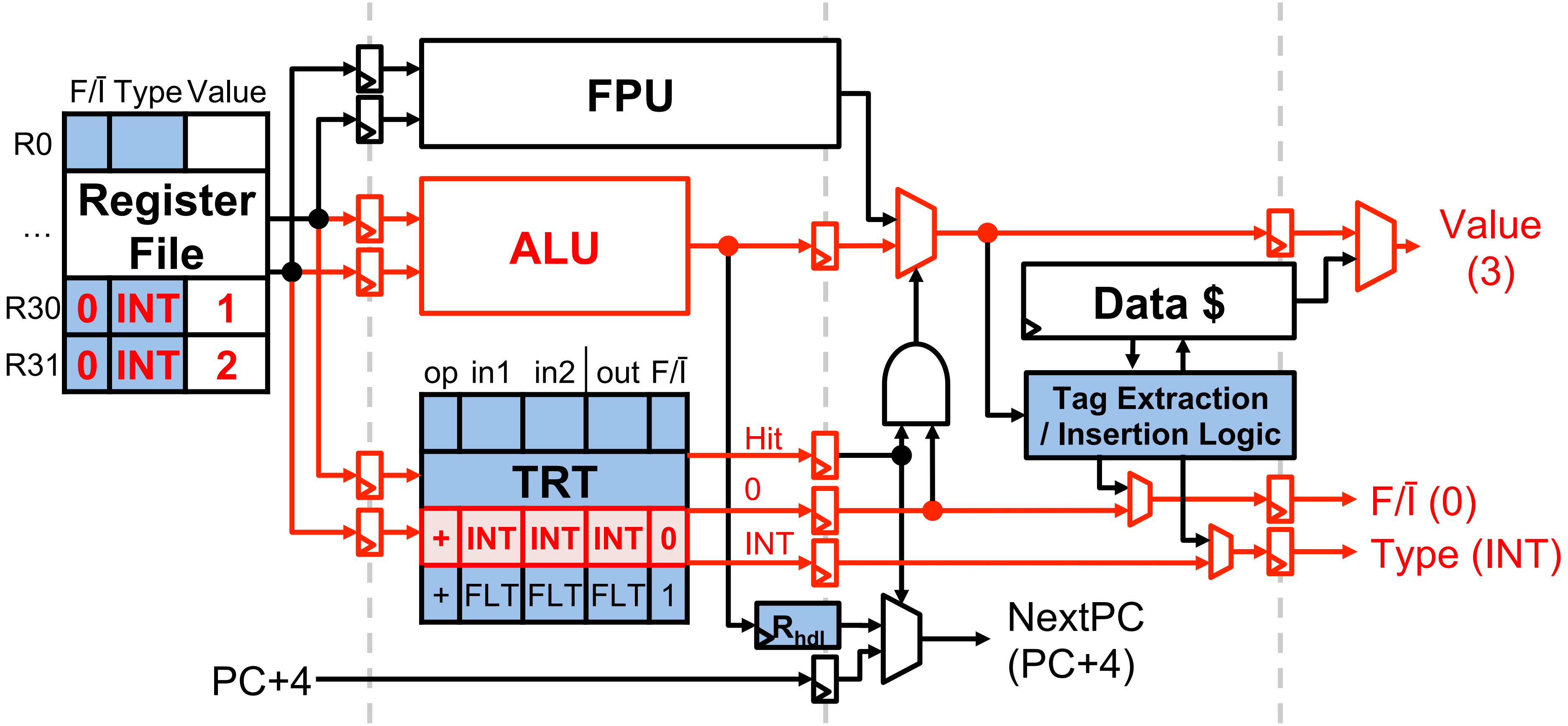
```
bne
a5.t,a4.t,isFltRB
```

```
add  a5.v ← a5.v,a2.v
```

```
sd   0(s14.v) ← a5.v
sw   8(s14.v) ← a5.t
```

Component #2: Tagged ALU Instructions (1)

- **Example:** `xadd r30 ← r30, r31 // for polymorphic ”+” operator`
  - Case 1: R30 (Integer) + R31 (Integer)
  - Vaules dispatched to integer ALU and type checking performed in parallel (**type hit**)

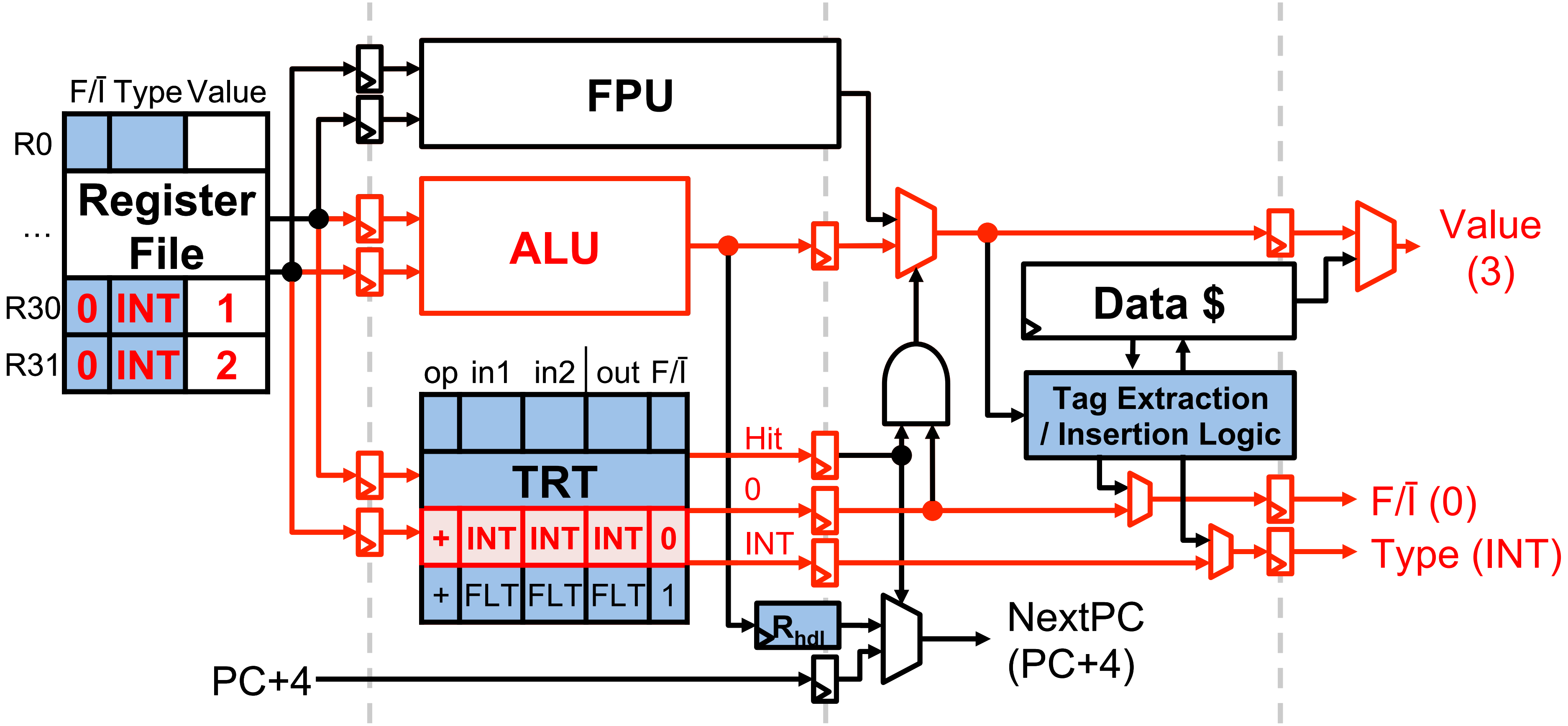


	F/Ī	Type	Value
R0			
…			
R30	0	INT	1
R31	0	INT	2

op	in1	in2	out	F/Ī
+	INT	INT	INT	0
+	FLT	FLT	FLT	1

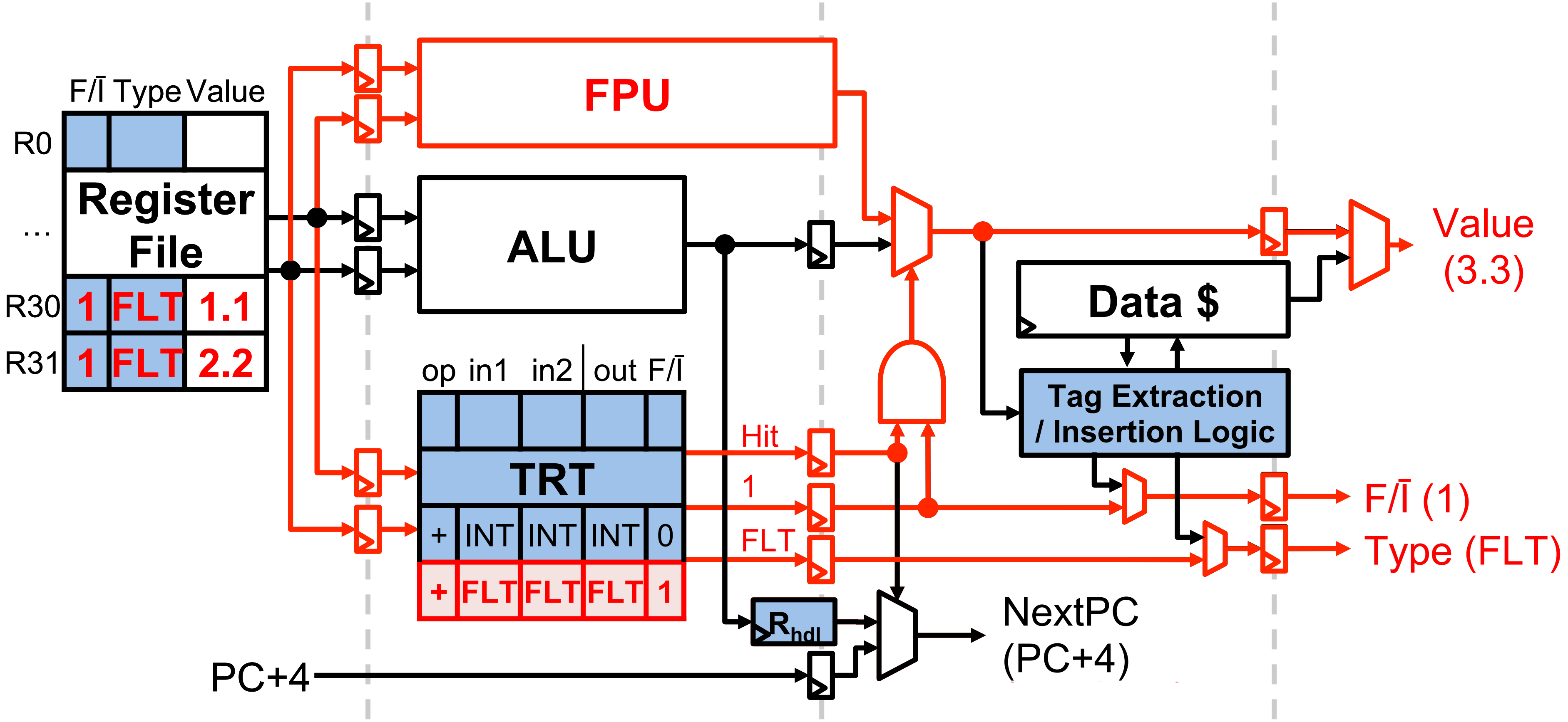
Component #2: Tagged ALU Instructions (2)

- **Example:** `xadd r30 ← r30, r31 // for polymorphic “+” operator`
  - Case 2: R30 (Float) + R31 (Float)
  - Vaules dispatched to FP Unit and type checking performed in parallel (**type hit**)



Component #2: Tagged ALU Instructions (3)

- **Example:** **`xadd r30 ← r30, r31 // for polymorphic "+" operator`**
  - Case 3: R30 (Integer) + R31 (Float)
  - PC is redicrected to the slow path pointed to by R_{hdl} (**type miss**)

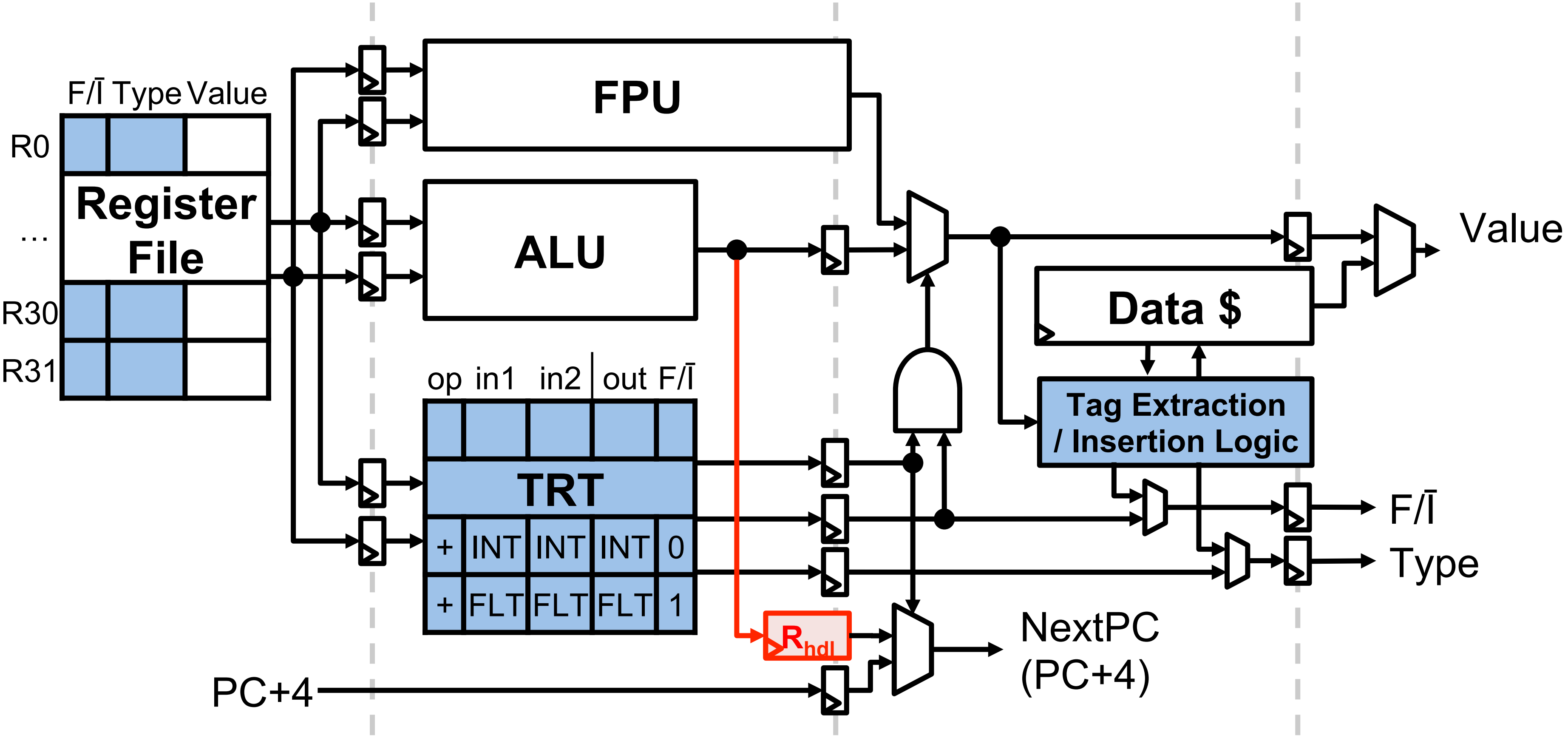


	F/Ī	Type	Value
R0			
…	Register File		
R30	1	FLT	1.1
R31	1	FLT	2.2

op	in1	in2	out	F/Ī
+	INT	INT	INT	0
+	FLT	FLT	FLT	1

Component #2: Tagged ALU Instructions (4)

- **Setting the value of the type miss handler register (`thdl`)**
  - Format: `thdl .LABEL`
  - Loads the starting address of the slow path (`.LABEL`) into $\mathbf{R_{hdl}}$



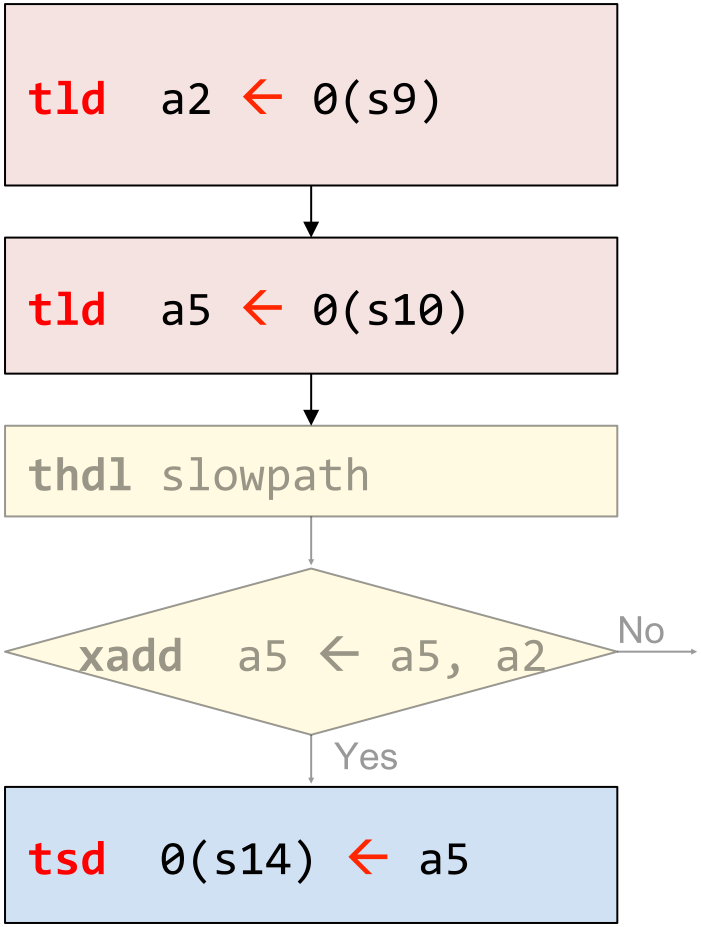
op	in1	in2	out	F/Ī
+	INT	INT	INT	0
+	FLT	FLT	FLT	1

Component #3: Tagged Memory Instructions (1)

- **Tagged memory instructions (`tld/tsd`)**
  - Loads/stores a value with type tag

- **Special-purpose registers**
  - $\mathbf{R_{offset}}$ (Offset Register): indicates which quad-word the tag will be extracted from
  - $\mathbf{R_{shift}}$ (Shift Amount Register): holds the starting position of type field
  - $\mathbf{R_{mask}}$ (Mask Register): holds 8-bit mask to extract 8-bit type tag

Bytecode ADD



Component #3: Tagged Memory Instructions (2)

- **Tagged load instruction**
  - Example: `tld   a2 ← 0(s9)`
  - Loads the value of a variable with its type tag

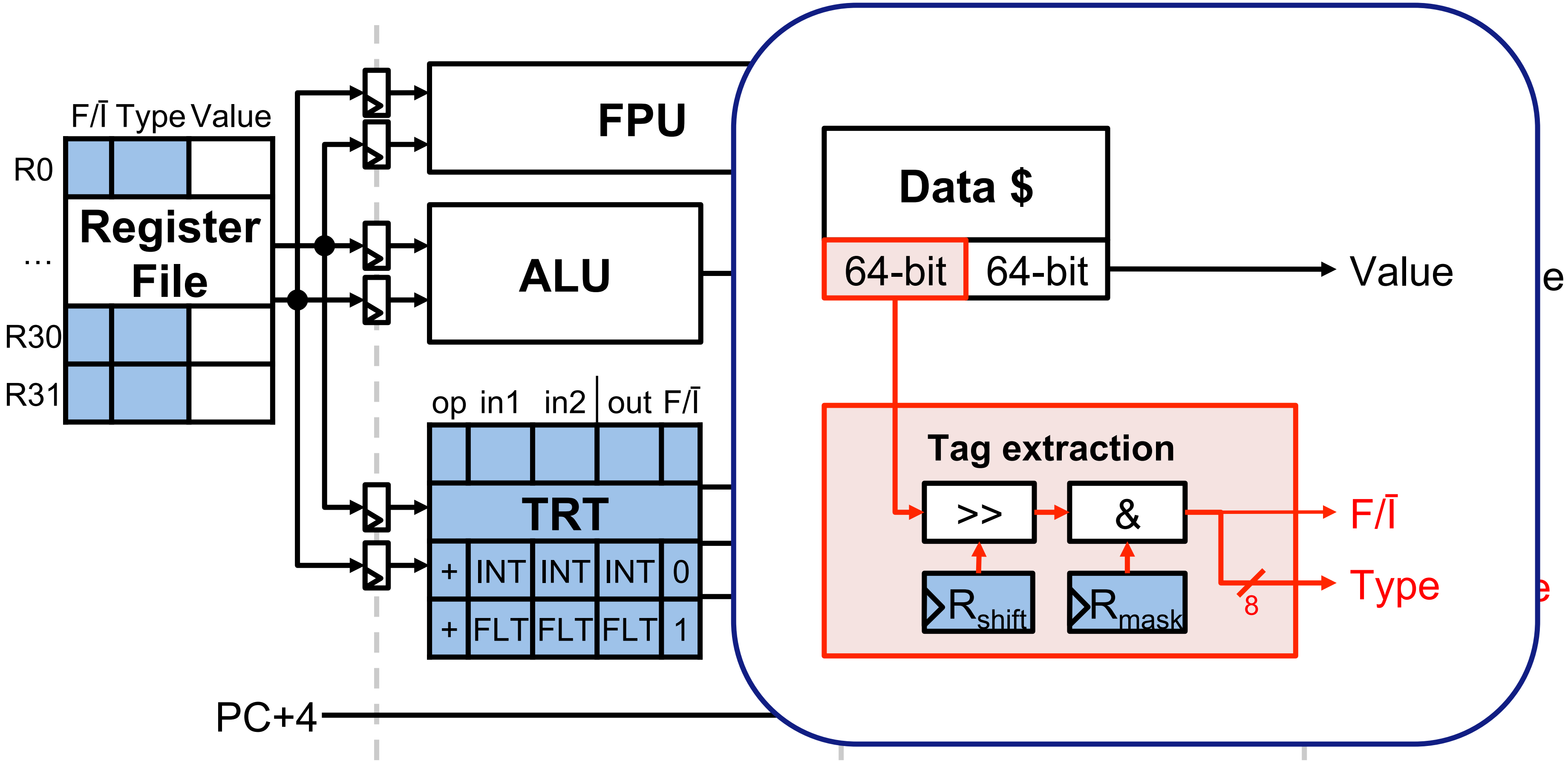


Table Access (1)

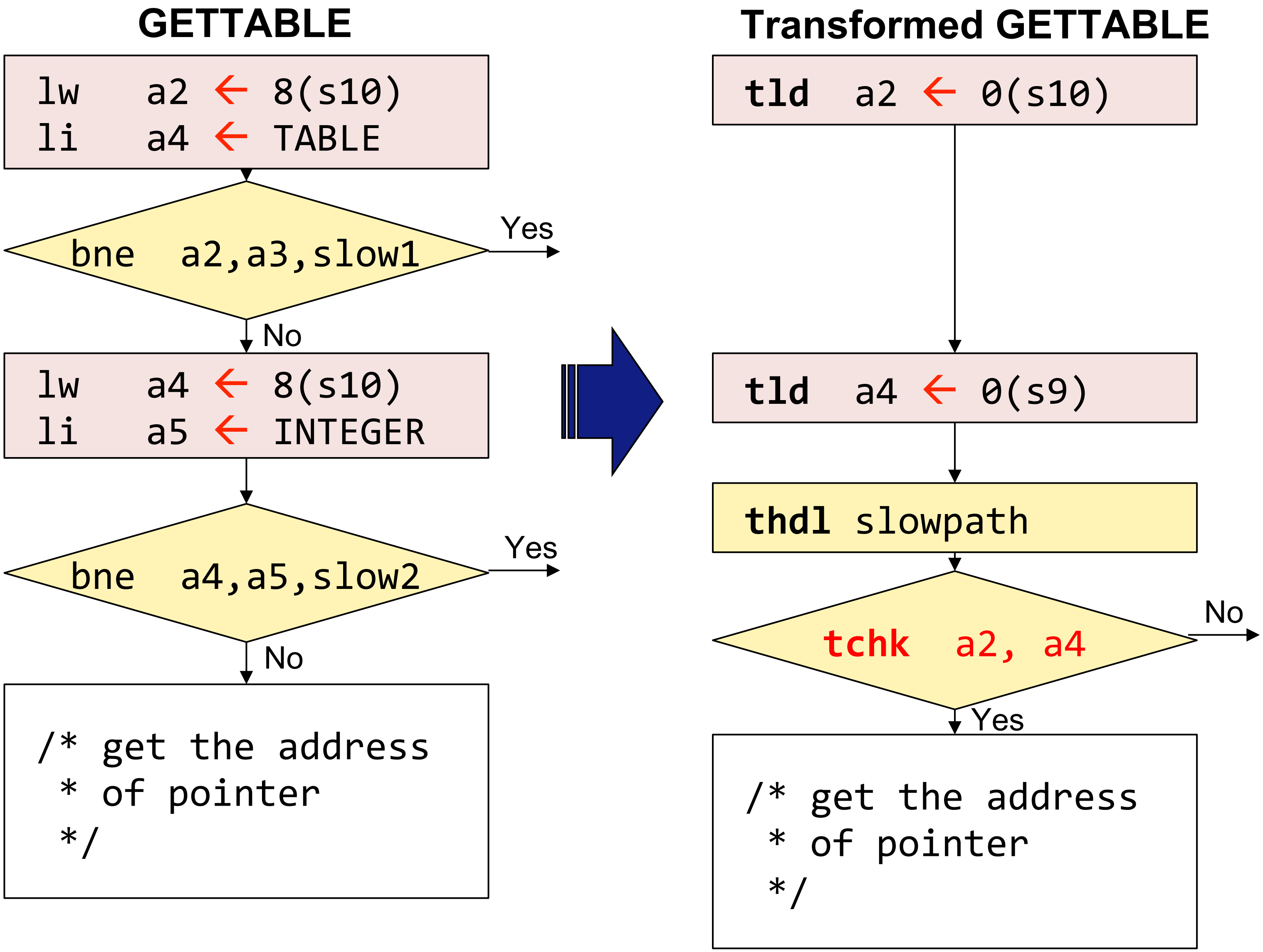
```
// Table access example in JavaScript
arr = [1, 2, 3];

arr[0] = 0;     // 0 (table update)
print(arr[0]); // 0 (table access)
```

- **Table access bytecodes**
  - Lua: GETTABLE/SETTABLE
  - JavaScript (SpiderMonkey): GETELEM/SETELEM

- **Each bytecode consists of two parts:**
  - Address calculation for the requested element
  - Element access/update with boundary checking

Table Access (2)



GETTABLE

```
lw    a2 ← 8(s10)
li    a4 ← TABLE
```

```
bne  a2,a3,slow1
```

Yes / No

```
lw    a4 ← 8(s10)
li    a5 ← INTEGER
```

```
bne  a4,a5,slow2
```

Yes / No

```
/* get the address
 * of pointer
 */
```

Transformed GETTABLE

```
tld   a2 ← 0(s10)
```

```
tld   a4 ← 0(s9)
```

```
thdl slowpath
```

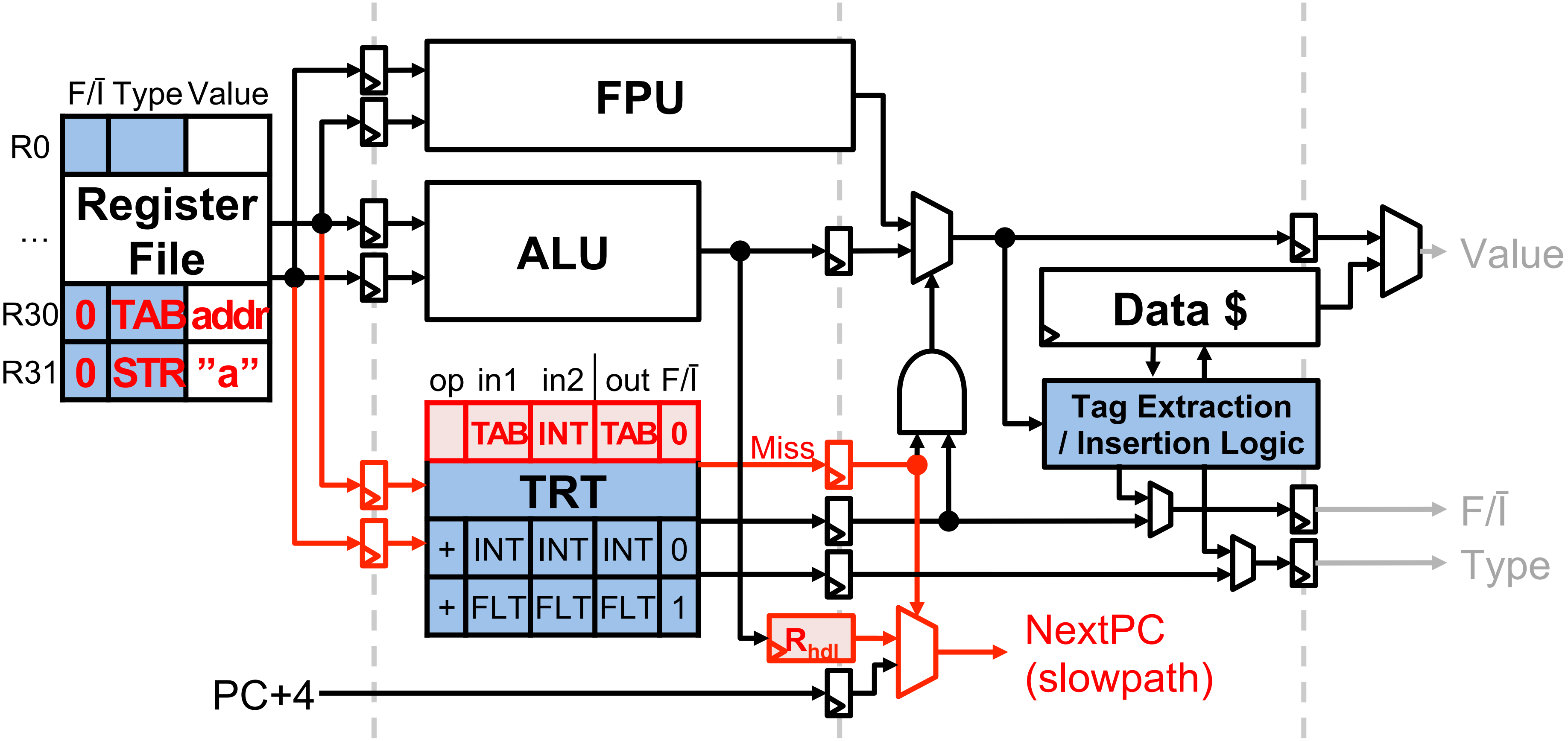
```
tchk  a2, a4
```

No / Yes

```
/* get the address
 * of pointer
 */
```

Table Access (3)

- **Tag check instruction**
  - Example: `tchk a2, a4`
  - Checks only type tags of two operands



	F/Ī	Type	Value
R0			
…	Register File		
R30	0	TAB	addr
R31	0	STR	"a"

op	in1	in2	out	F/Ī
	TAB	INT	TAB	0
+	INT	INT	INT	0
+	FLT	FLT	FLT	1

Topics Not Covered in This Presentation

Please refer to the paper for the following information:

- **Details of pipeline design**
- **Code transformation for Lua and JavaScript**
- **OS context switching**
- **Legacy code execution**
- **Detailed power and area analysis using synthesizable RTL**
- **etc.**

Outline

- Motivation and key idea
- Typed Architecture
- **Evaluation**
  - Methodology
  - Performance Results
  - Area and Power Overhead
- **Summary**

Evaluation Methodology (1): Evaluation Platform

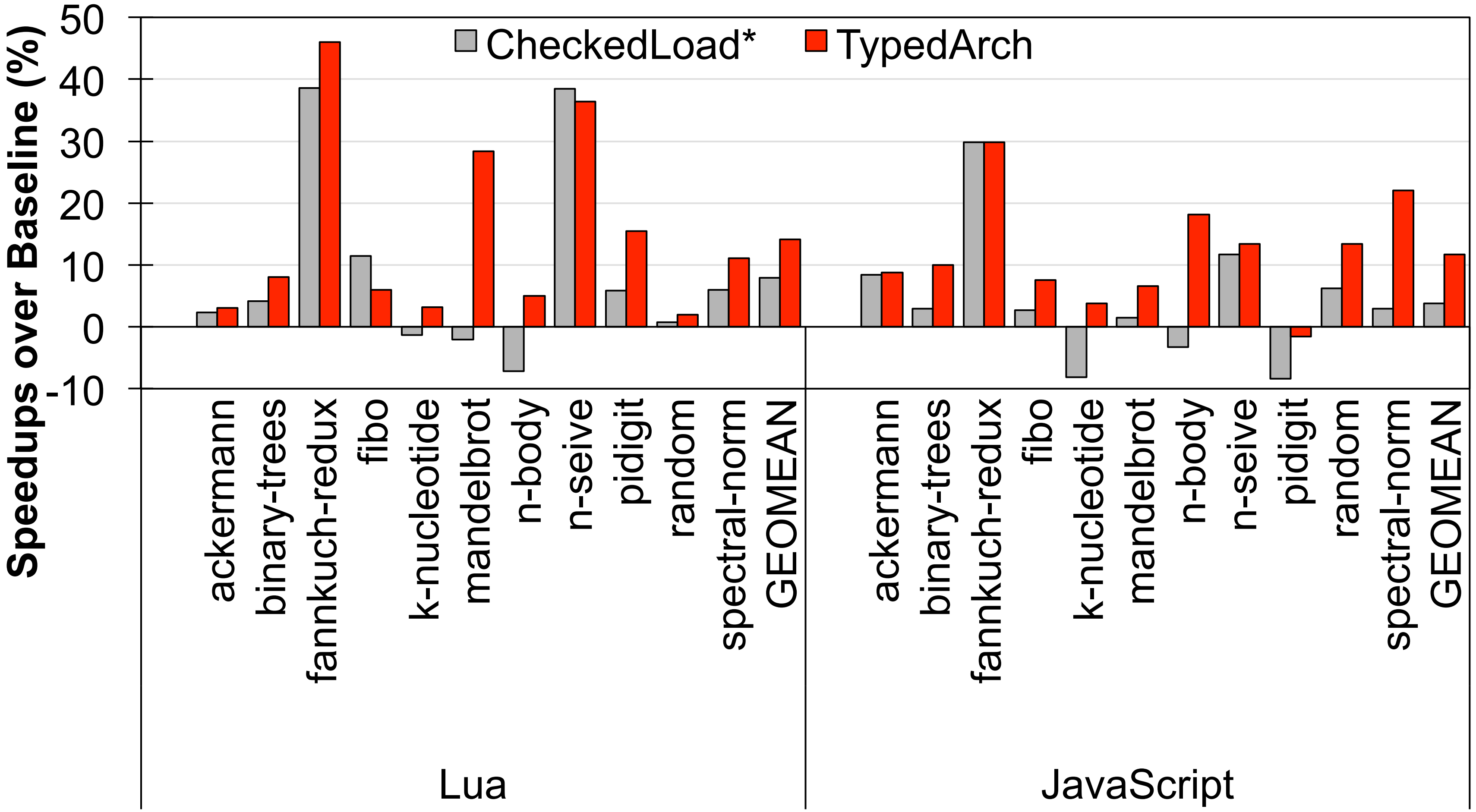
Platform	Xilinx Zynq ZC706 FPGA
Processor	64-bit RISC-V Rocket Core
Pipeline	Single-Issue In-Order, 50MHz Fetch/Decode/Execute/Mem/WB (5 stages)
Branch Predictor	32B predictor (128-entry gshare) 62-entry, fully-associative BTB with LRU replacement policy 2-entry return address stack 2-cycle branch miss penalty
Caches	16KB, 4-way, 1-cycle L1 I-cache 16KB, 4-way, 1-cycle L1 D-cache 8-entry I-TLB, 8-entry D-TLB 64B block size with LRU
Type Rule Table	**8-entry, 32B fully-associative table**

Evaluation Methodology (2): Workloads

- **Lua-5.3.0**
  - 47 distinct bytecodes
  - Modified bytecodes: ADD, SUB, MUL, GETTABLE, SETTABLE
- **SpiderMonkey-17.0 from FireFox (JavaScript)**
  - 229 distinct bytecodes
  - Modified bytecodes: ADD, SUB, MUL, GETELEM, SETELEM
- **JIT is disabled in both cases**
- **Benchmarks**
  - 11 scripts for each from Computer Language Benchmarks Game*

* http://benchmarksgame.alioth.debian.org

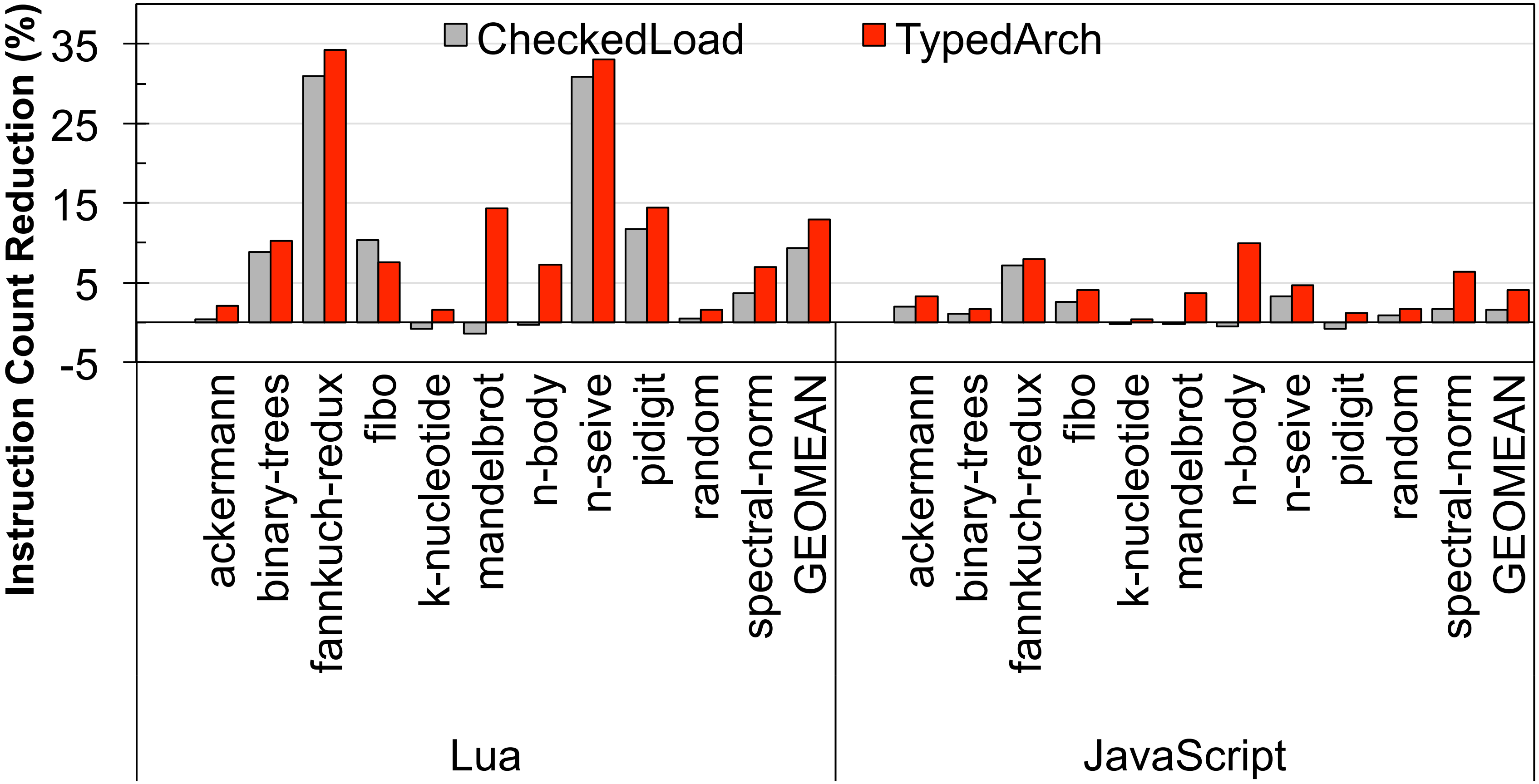
Overall Speedups



- **Geomean speedups (reflecting post-camera-ready updates)**
  - Lua: ~~9.9%~~ → 14.1% (Max: 46.0% for fannkuch-redux)
  - JavaScript: ~~11.2%~~ → 11.7% (Max: 29.9% for fannkuch-redux)

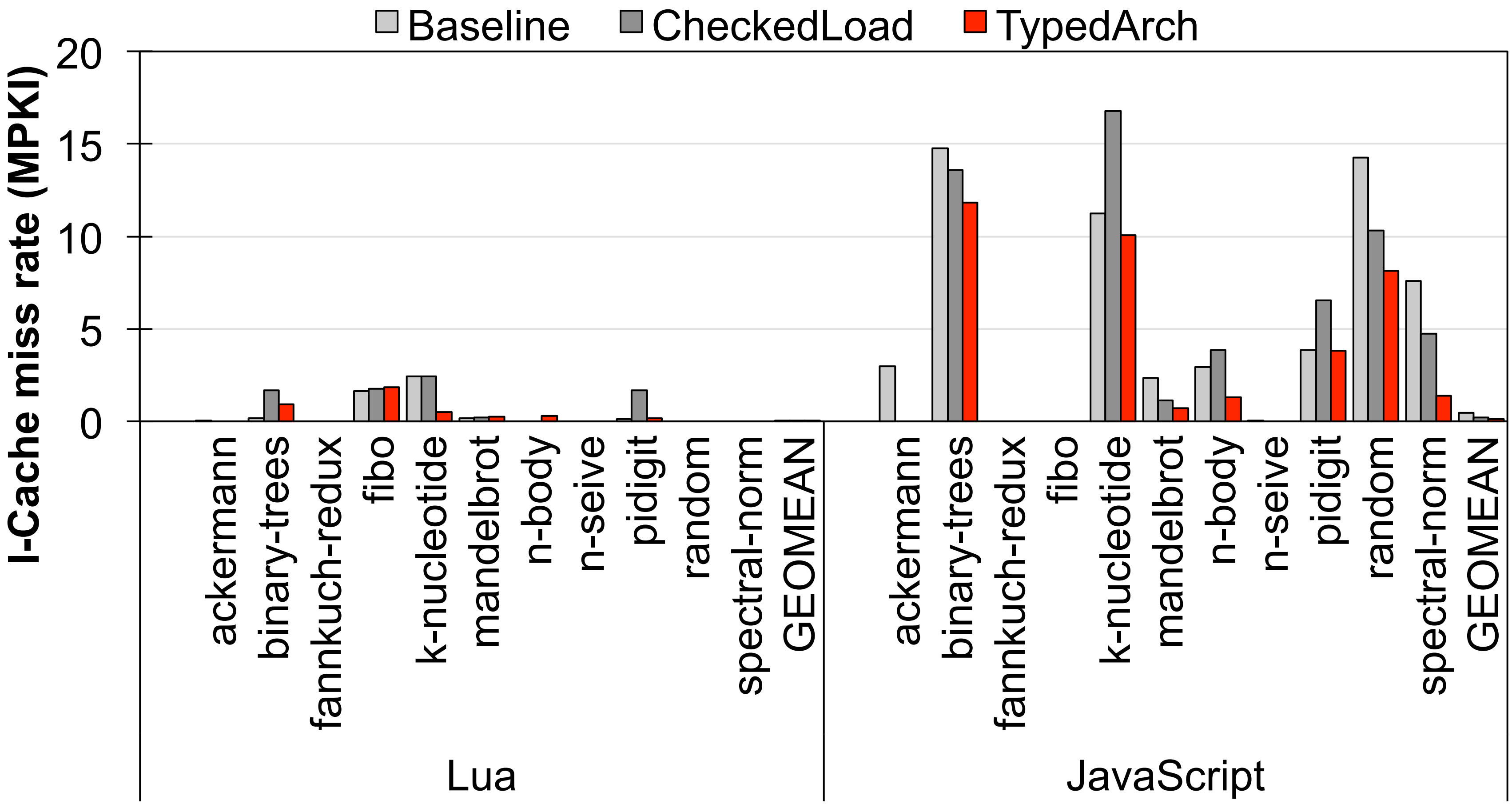
* [HPCA ’11] Checked Load: Architectural support for JavaScript type-checking on mobile processors

Normalized Instructions



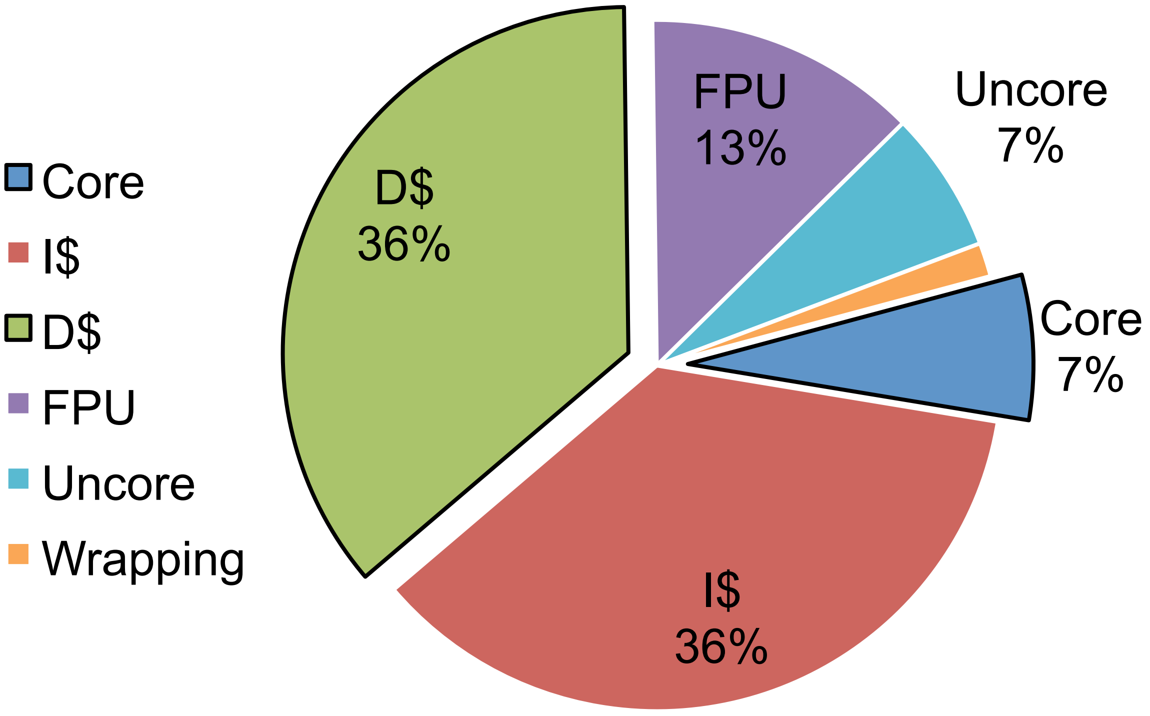
- **Reduction in dynamic instruction count**
  - Lua: **12.9%** (Max: 34.2% for fannkuch-redux)
  - JavaScript: **4.1%** (Max: 10.0% for n-body)

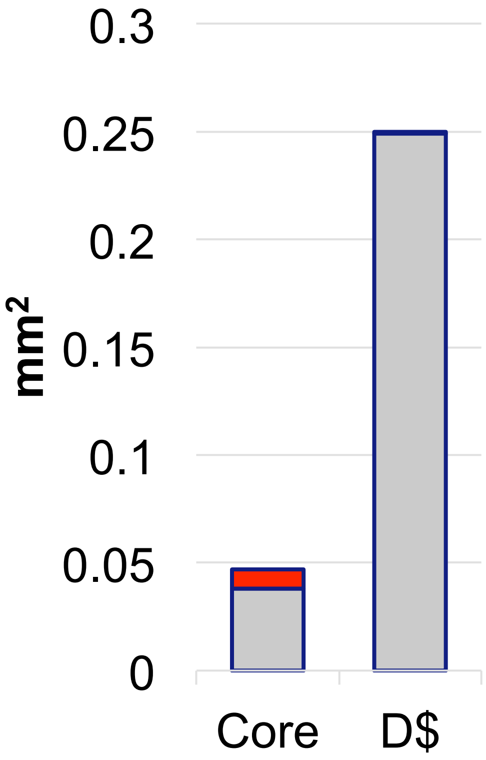
I-Cache Misses Per Kilo-Instructions (MPKI)



- **Significant reduction in I-cache miss rates (in MPKI)**
  - Lua: k-nucleotide (2.4 → 0.5), ackermann (0.03 → 0.01)
  - JavaScript: random (14.3 → 8.1), spectral-norm (7.6 → 1.4)

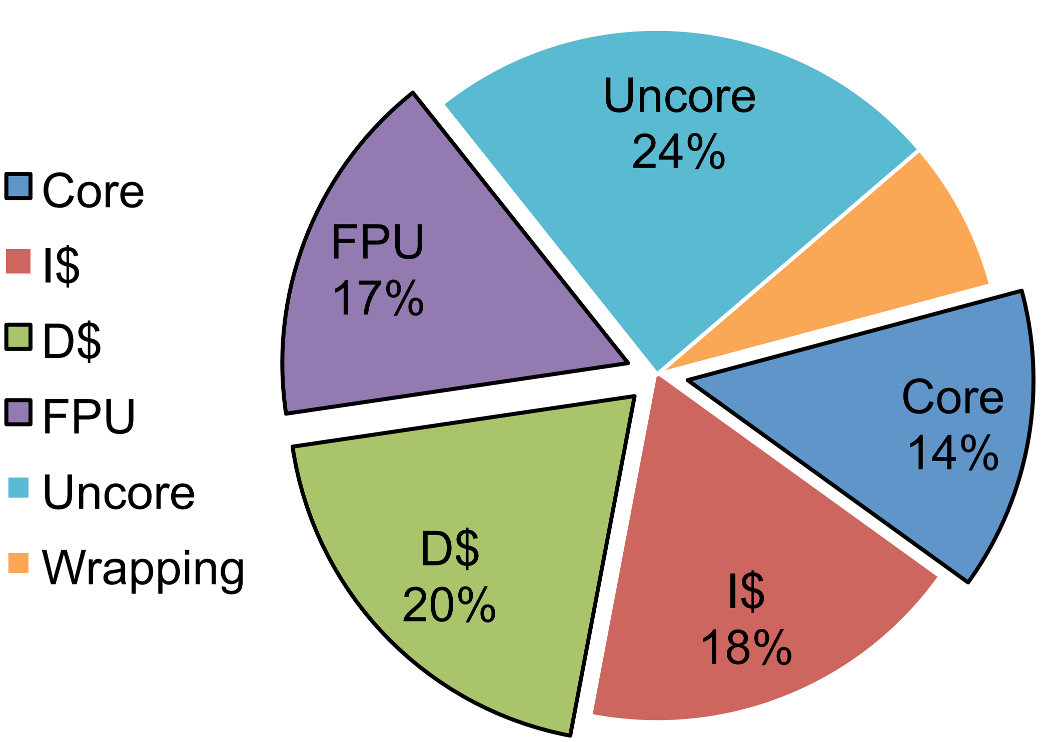
Area and Energy Overhead (1)

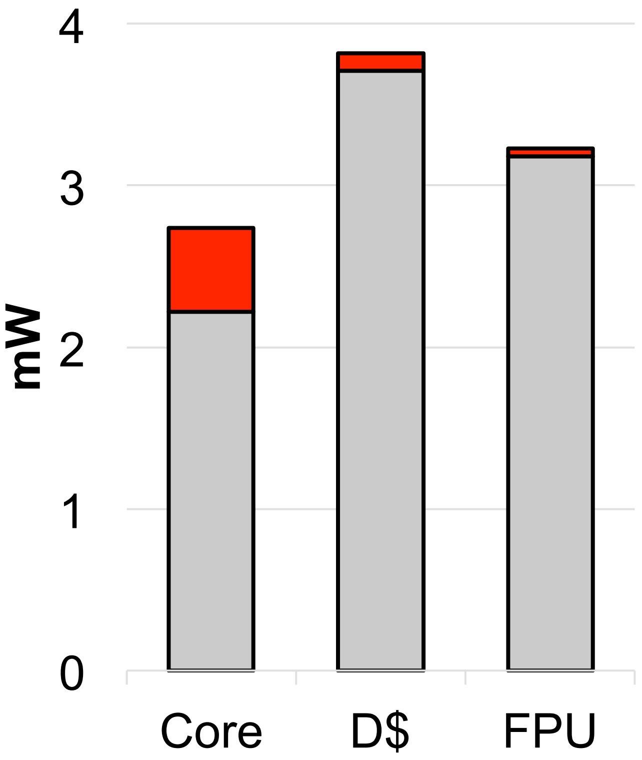




- **Minimal area/power costs (at TSMC 40nm technology node)**
  - Area overhead: **1.61%** (mostly in Core)

Area and Energy Overhead (2)





- **Minimal area/power costs (at TSMC 40nm technology node)**
  - Power overhead: **3.69%** (mostly in Core and Data Cache)
- **EDP improvement: 20.6%** (Lua), **17.1%** (JavaScript)

Summary

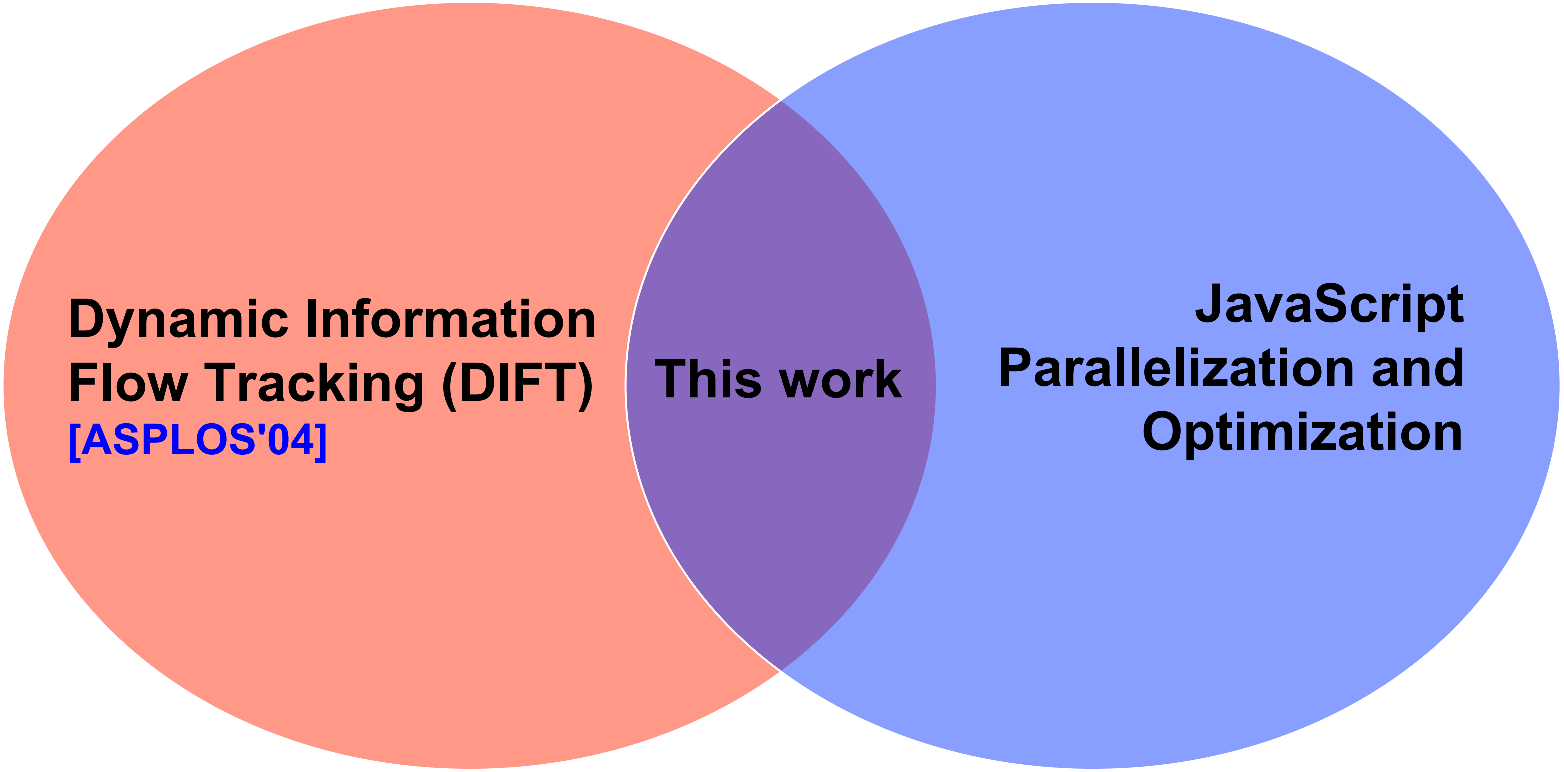
- **Dynamic type checking is one of the major sources of inefficiency for scripting languages**

- **Typed Architectures: Architectural support for efficient type checking**
  - Retaining high-level type information at an ISA level
  - Supporting polymorphic instructions depending on types of operands
  - Flexibly applied to multiple scripting languages and engines

- **Typed Architectures accelerate production-grade VM interpreters**
  - Geomean (Max.) speedups: **14.1% (46.0%)** for Lua, **11.7% (29.9%)** for JavaScript
  - **EDP** improved by **20.6%** for Lua and **17.1%** for JavaScript with only **1.6% area overhead** at 40nm technology node

One More Thing

- 어떤 계기로 연구를 시작하게 되었는가? 에피소드는?



Dynamic Information Flow Tracking (DIFT) [ASPLOS'04]

This work

JavaScript Parallelization and Optimization

Takeaway: 1. Nothing is new under the sun.

2. HW? SW? Both!

3. 연구와 시행착오에 대하여

Q & A

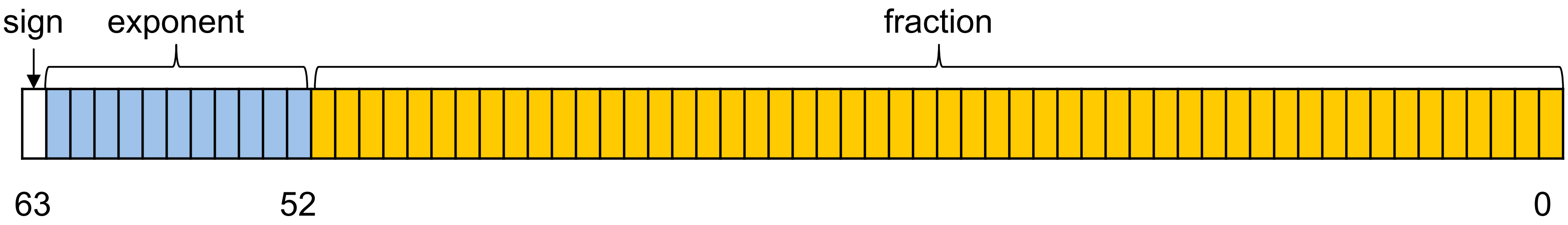
Q1: Comparison against Checked Load [HPCA'11]

- **Checked Load ported on RISC-V RocketCore to run on FPGA**
- **Limitations of Checked Load**
  - Fixing type comparison instruction at compile time (“**INT**” value)
  - No support for polymorphic instructions, tag extraction and insertion
  - Not suitable for fixed-length RISC instructions

```
// Checked load instruction (Variable length instruction)
chklb Rd ← Rs, imm, INT
```

* [HPCA ’11] Checked Load: Architectural support for JavaScript type-checking on mobile processors

Q2: Type Encoding in SpiderMonkey



- **Exploits NaNs (Not-a-Number) in IEEE 754 FP format to represent Ints**
  - sign: either 0 or 1 (1-bit)
  - exponent: all 1 bits (11-bits)
  - fraction: anything except all 0 bits (52-bits)

- **SpiderMonkey’s Type Encoding**

1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	INT (0x1fff1)
1	1	1	1	1	1	1	1	1	1	1	1	1	0	1	1	1	TAB (0x1fff7)
63											52					47	

Q3: Application to Higher Performance Core?

- **Possible.**
- **However, in higher performance cores, other software techniques (like JIT) are also viable.**

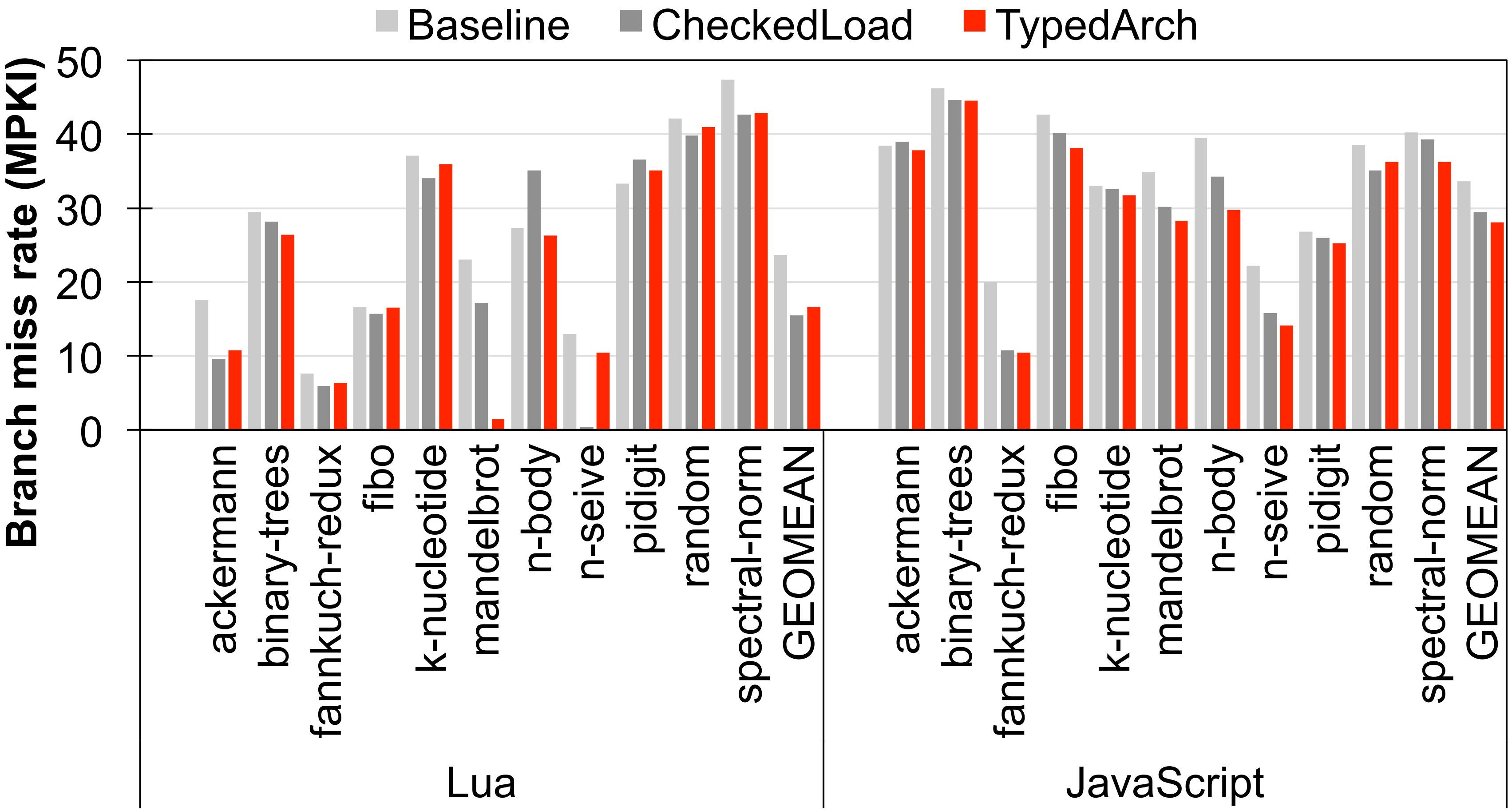
Q4: Why not JIT compilation?

- **Typed Architectures aim to complement, but not replace JIT.**
  - JIT compilation can be applied on Typed Architectures.
  - But, in resource-constrained IoT platforms JIT may not be viable.
  - Also, effectiveness of JIT depends highly on a small number of hot methods dominating total execution time
- **Typed Architectures augment JIT in terms of the applicability.**

Q5: Performance of fannkuch-redux, n-sieve, and spectral-norm (1)

- **Significant reduction in dynamic instruction count**
  - Table access-heavy workloads (using GETELEM and SETELEM bytecodes)
  - High type hit rates → bypasses complex if-chains in table access bytecodes

Q5: Performance of fannkuch-redux, n-sieve, and spectral-norm (2)



- **Reduction in branch miss rates (in MPKI)**
  - Lua: 23.68 → 16.61
  - JavaScript: 33.67 → 28.03

Q6: Post-camera-ready Updates

- **Fixed two performance bugs in our RTL implementation**
  - Removes unnecessary bubbles for floating point calculation
  - Increases type hit rates (eliminating false type misses)
- **No correctness issues**

Q7: Primitive Types of Lua and JavaScript

- **Primitive types**
  - Lua: *nil*, *boolean*, ***number***, *string*, *function*, *userdata*, *thread*, and ***table***
  - JavaScript: *boolean*, *null*, *undefined*, ***number***, *string*, symbol, and ***object***
  - ***number*** type is internally magaged as either ***Integer*** and ***Float***.

- **May be applicable to other types (such as *boolean* and *string*)**
  - But difficult for *function*, *userdata*, *thread*, etc.

Q8: What if tag field requires more than 8 bits?

- **8-bit type tag support 256 distinct types**
  - In observation, it can accommodate most engines.
- **If necessary we can re-encode type values to fit in 8 bits**

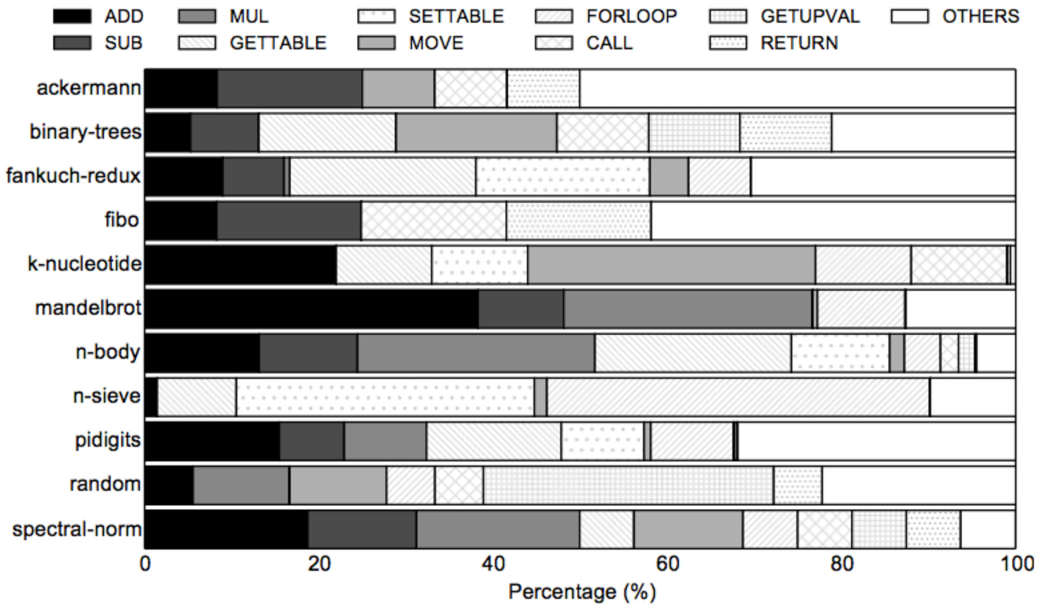
Q9: Application to Other Scripting Languages

- **Typed architecture is flexible enough to be applicable to other scripting languages and implementations.**

Q10: Polymorphic Instructions with Integer and FP Source Operands

- **Currently, handled in the slow path**
  - Type conversion is needed (Integer to FP) before the operation

Q11: Breakdown of dynamic bytecodes in Lua



- **5 most frequently used bytecodes account for a majority of total bytecode count. (ADD, SUB, MUL, GETTABLE, SETTABLE)**

Q13: Why Not Python Instead of Lua?

- **Lua**
  - Having a simpler type system and easier to understand
  - Easier to build for RISC-V on FPGA

- **Stay tuned!**

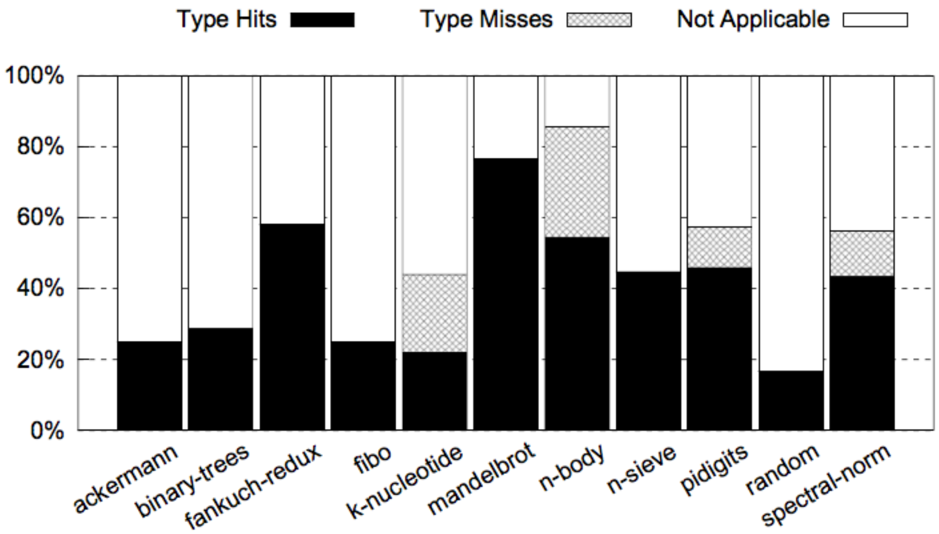
Q14: Controling Garbage Collection (GC)

- **We wanted to measure performance of the mutator (main code).**
  - GC performance depends on many factors (heap size, GC algorithm, etc.) that are orthogonal to our work.
  - Lua: GC turned off
  - SpiderMonkey (JavaScript): GC turned on (No easy way to turn off)

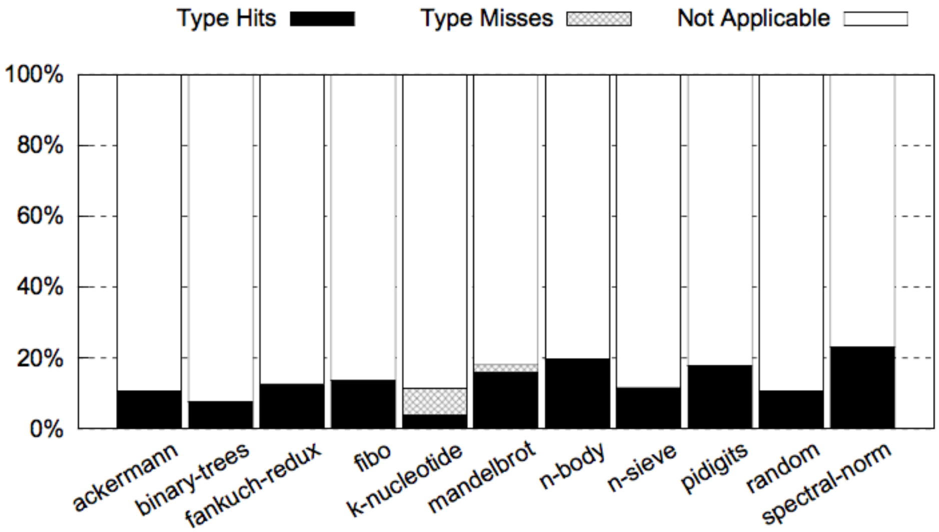
- **GC does NOT occur frequently in our benchmarks, so only marginal impact.**

Q15: Type Hit and Miss Rates

- **Lua**

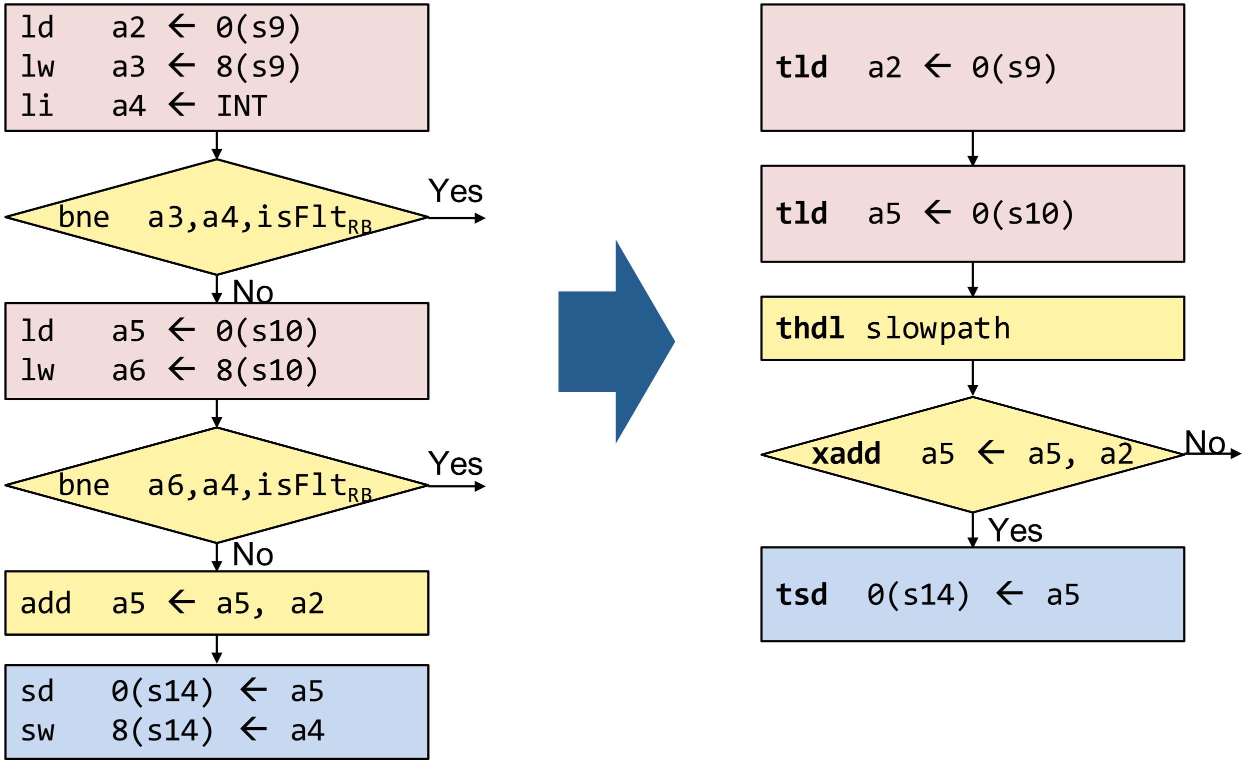


- **JavaScript**



Q16: Three Major Sources of Performance Improvement

- **Reduction in dynamic instruction count**
  - For integer addition, dynamic instruction count is reduced from 10 to 5.
- **Reduction in instruction cache miss**
- **Reduction in branch miss**



Q17: Using Newest Version of SpiderMonkey

- Failed to cross-compile using RISC-V toolchain

Q18: 40nm Technology Node? Why Not 10nm?

- **Good enough to evaluate the area and power overhead**

Q20: IoT Benchmarks