



UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

Restoration and development of Arm's Java-based LEGv8 ISA simulator

Graduating student - Simone Deiana

Supervisor - Alberto Carini

Restoration and development of Arm's Java-based LEGv8 ISA simulator



Restoration and development of Arm's Java-based LEGv8 ISA simulator





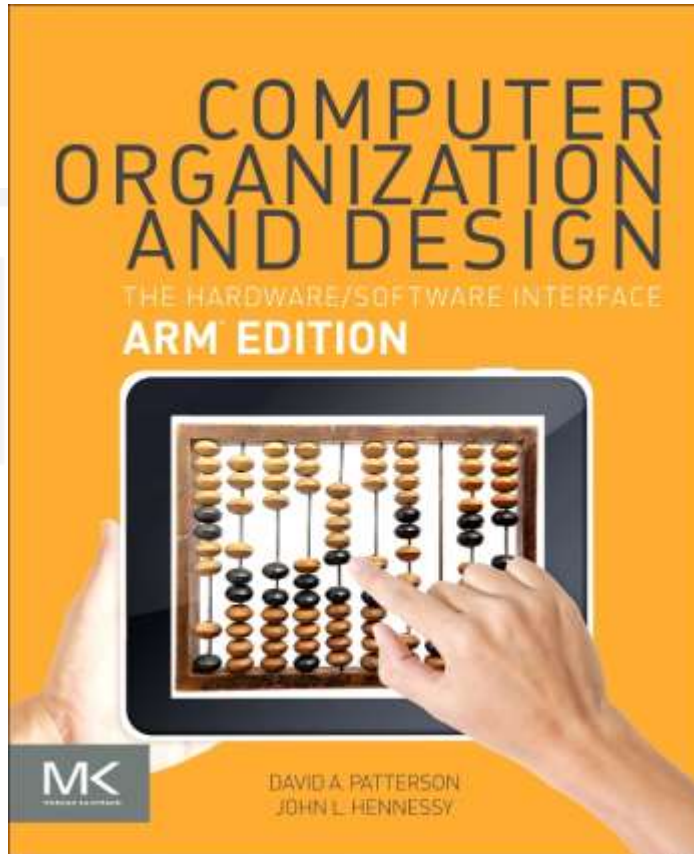
Restoration and development of Arm's Java-based LEGv8 ISA simulator

WHAT IS LEGv8?

Restoration and development of
Arm's Java-based **LEGv8 ISA** simulator

WHAT IS LEGv8?

AN ISA FOR LEARNING COMPUTER ARCHITECTURES



From Computer Organization and Design ARM Edition: The Hardware Software Interface - Patterson, D.A. and Hennessy, J.L.



David A.
Patterson

Peg Skorpinski, CC BY-SA
3.0
<<https://creativecommons.org/licenses/by-sa/3.0/>>, via
Wikimedia Commons



John L.
Hennessy

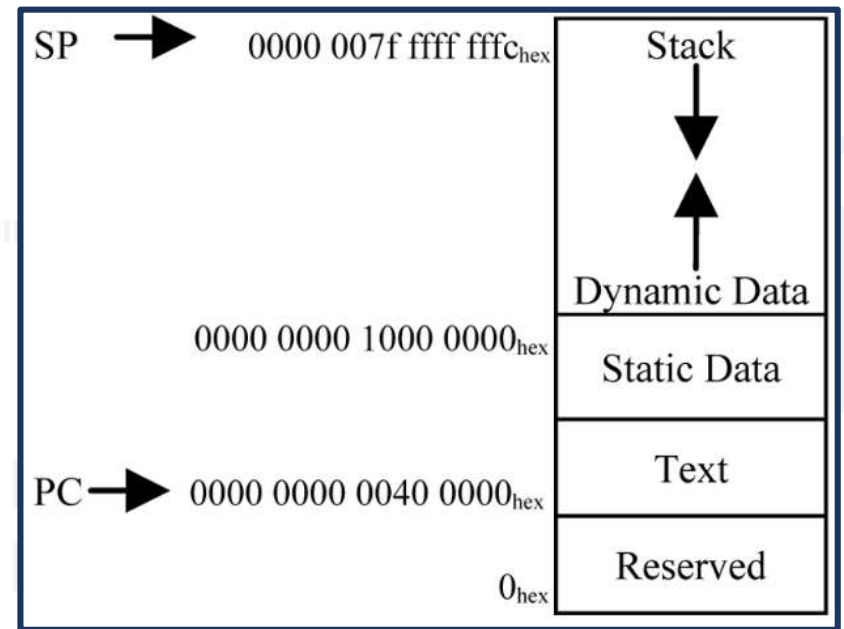
Eric Chan, CC BY 2.0
<<https://creativecommons.org/licenses/by/2.0/>>, via Wikimedia Commons

THE DESIGN PHILOSOPHY

- As simple as it can be...
- ... but with a modern design
- Heavily inspired by ARMv8, almost a "subset"

THE MEMORY

- 64-bit addresses
- Harvard model



THE REGISTERS

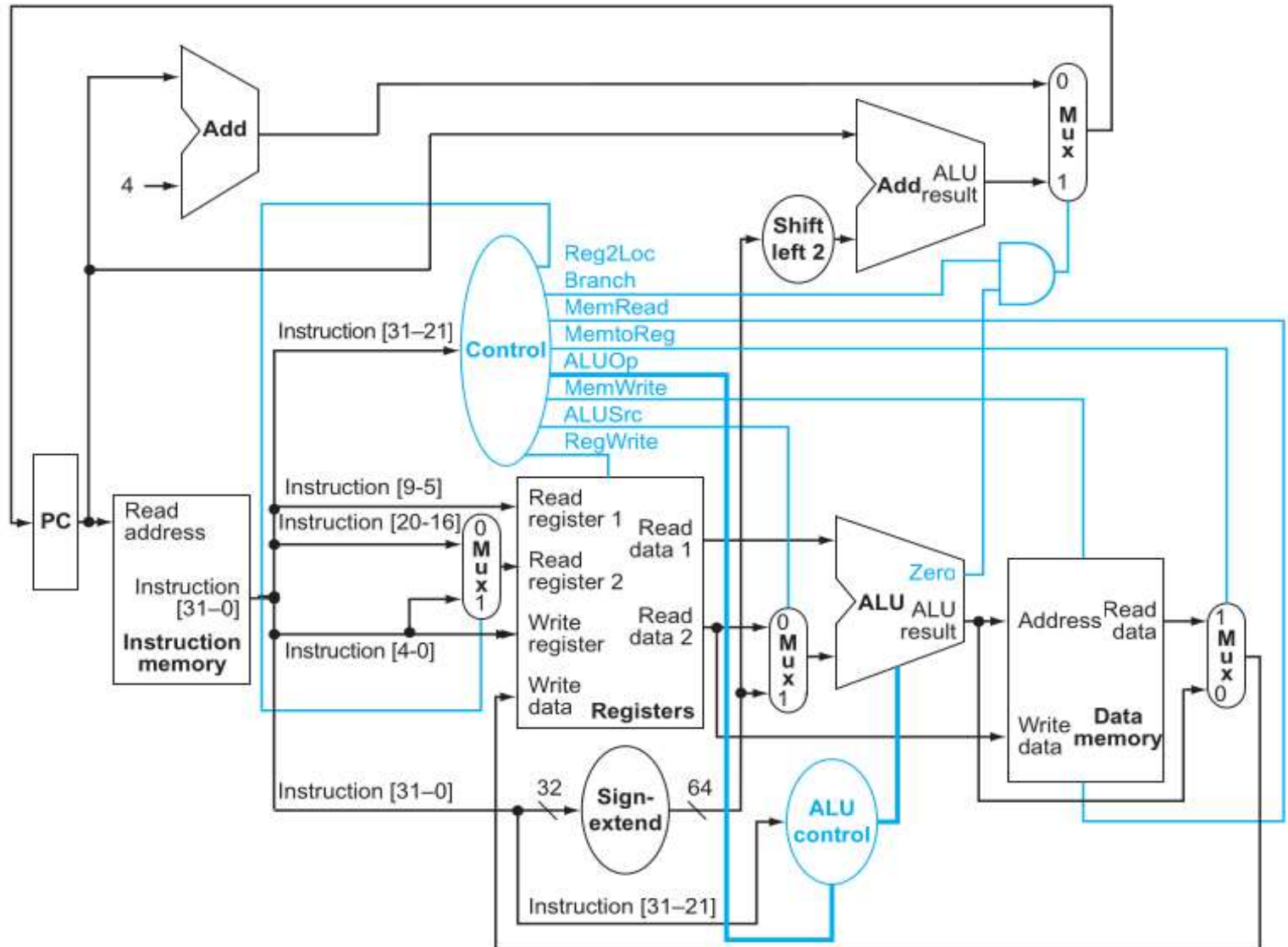
- 32 64-bit "X" integer registers
- 32 64-bit "D" floating-point registers
- 32 32-bit "S" floating-point "registers"

THE INSTRUCTIONS

- 64-bit integer and IEEE-754 floating-point arithmetic
- Designed and optimized for pipelined execution

WHAT IS LEGv8?

THE DATAPATH

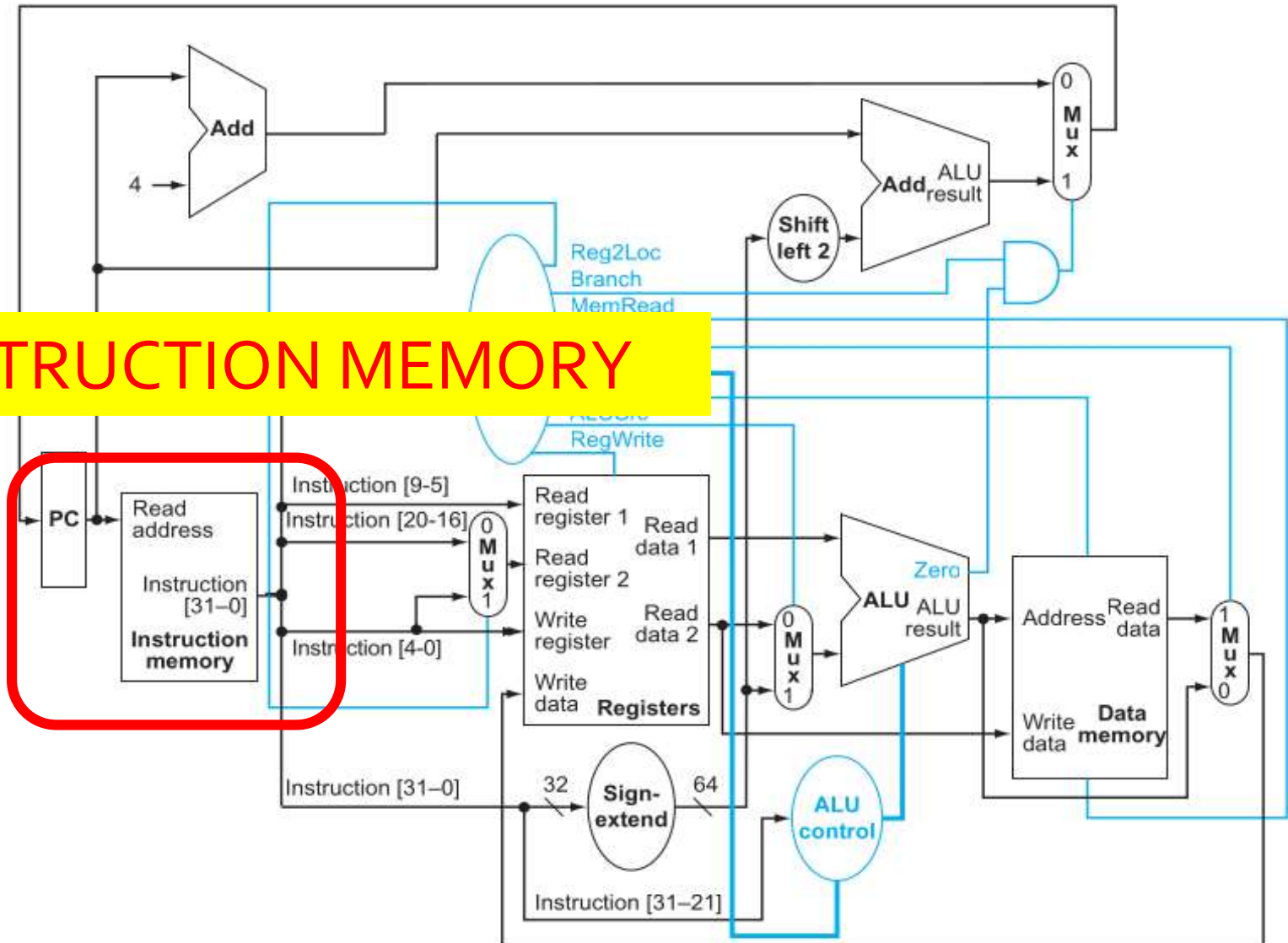


From Computer Organization and Design
ARM Edition: The Hardware Software Interface -
Patterson, D.A. and Hennessy, J.L.

WHAT IS LEGv8?

THE DATAPATH

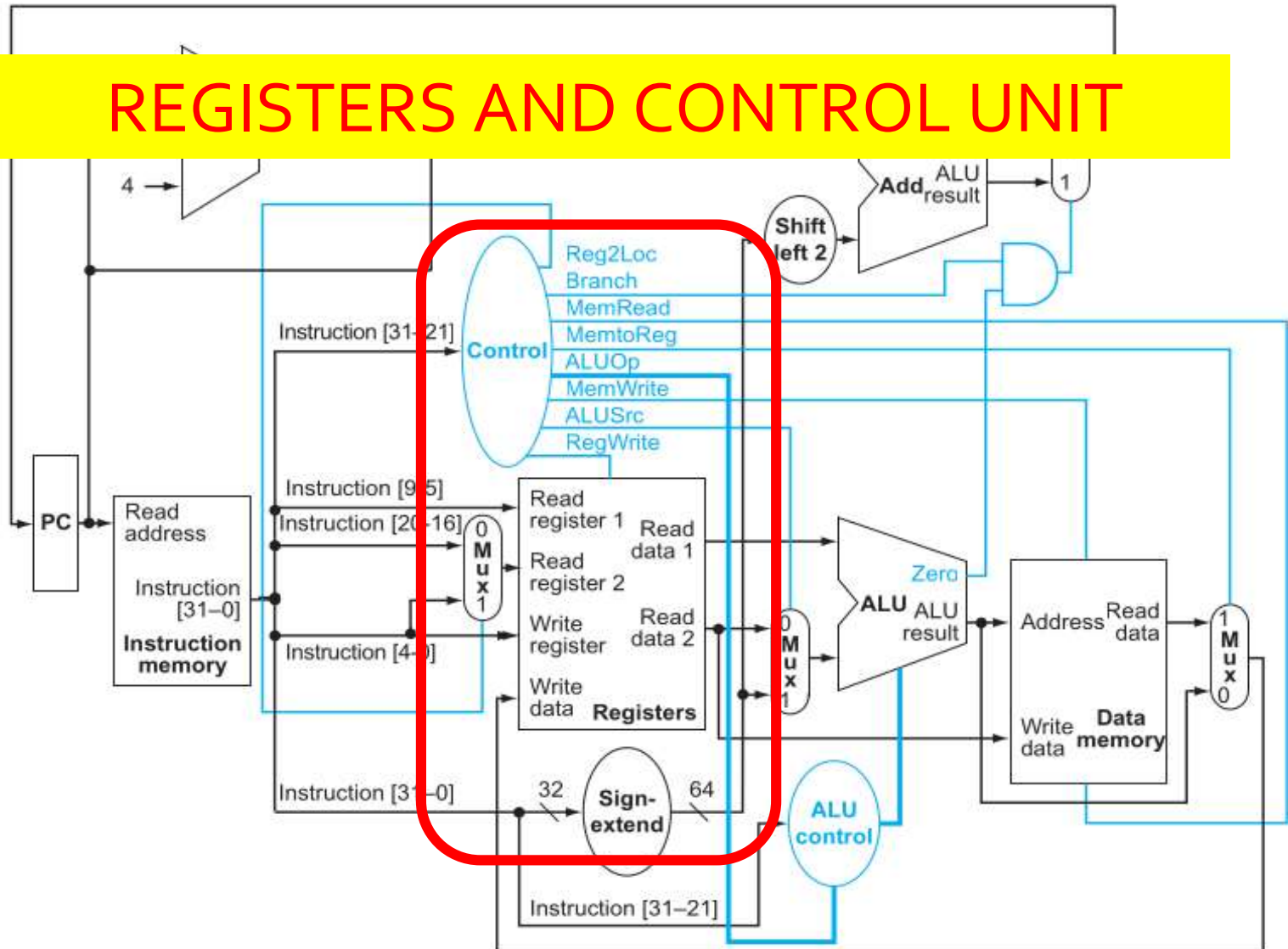
INSTRUCTION MEMORY



WHAT IS LEGv8?

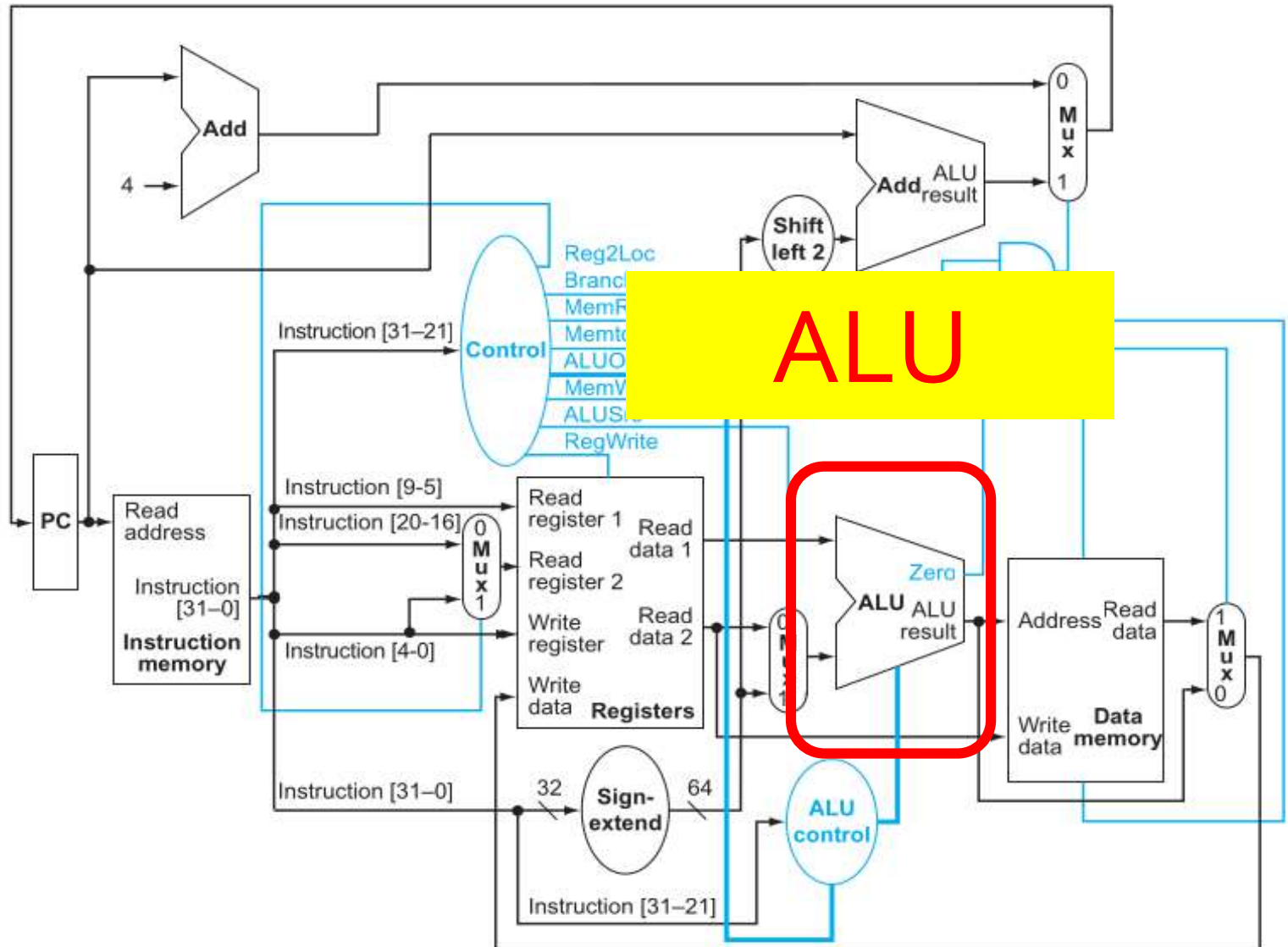
THE DATAPATH

REGISTERS AND CONTROL UNIT



WHAT IS LEGv8?

THE DATAPATH



THE DATAPATH



15

WHAT SIMULATOR, AND WHY?

Restoration and development of
Arm's Java-based LEGv8 ISA simulator

WHAT SIMULATOR, AND WHY?

NO HARDWARE FOR LEGv8 => NEED A SIMULATOR



Michael H. („Laserlicht“) / Wikimedia Commons

WHAT SIMULATOR, AND WHY?

BUT WHICH ONE?



LEGv8

197 results

THE PROBLEM:

NO SOFTWARE CAN YET
SIMULATE THE ENTIRE
LEGv8 ISA!

THE SOLUTION:

- Write one from scratch
- OR (BETTER)
- Improve one that already exists

ARM HAS OFFICIALLY MADE A LEGv8 SIMULATOR

GOOD!

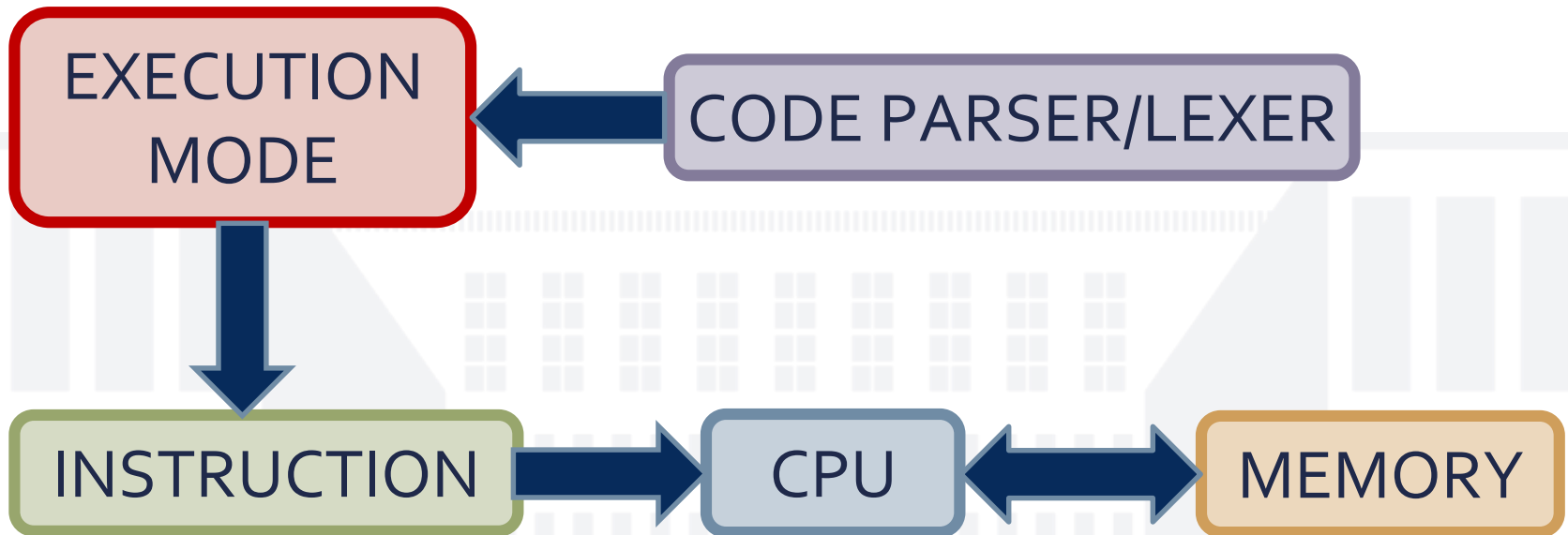
IT'S INCOMPLETE
AND BROKEN

BAD!

WHAT STANDS OUT:

- Written in Java (high level, extensible)
- Distributed as a web application
- Nice, functional UI
- Closely follows the textbook

STRUCTURE OF THE SIMULATOR



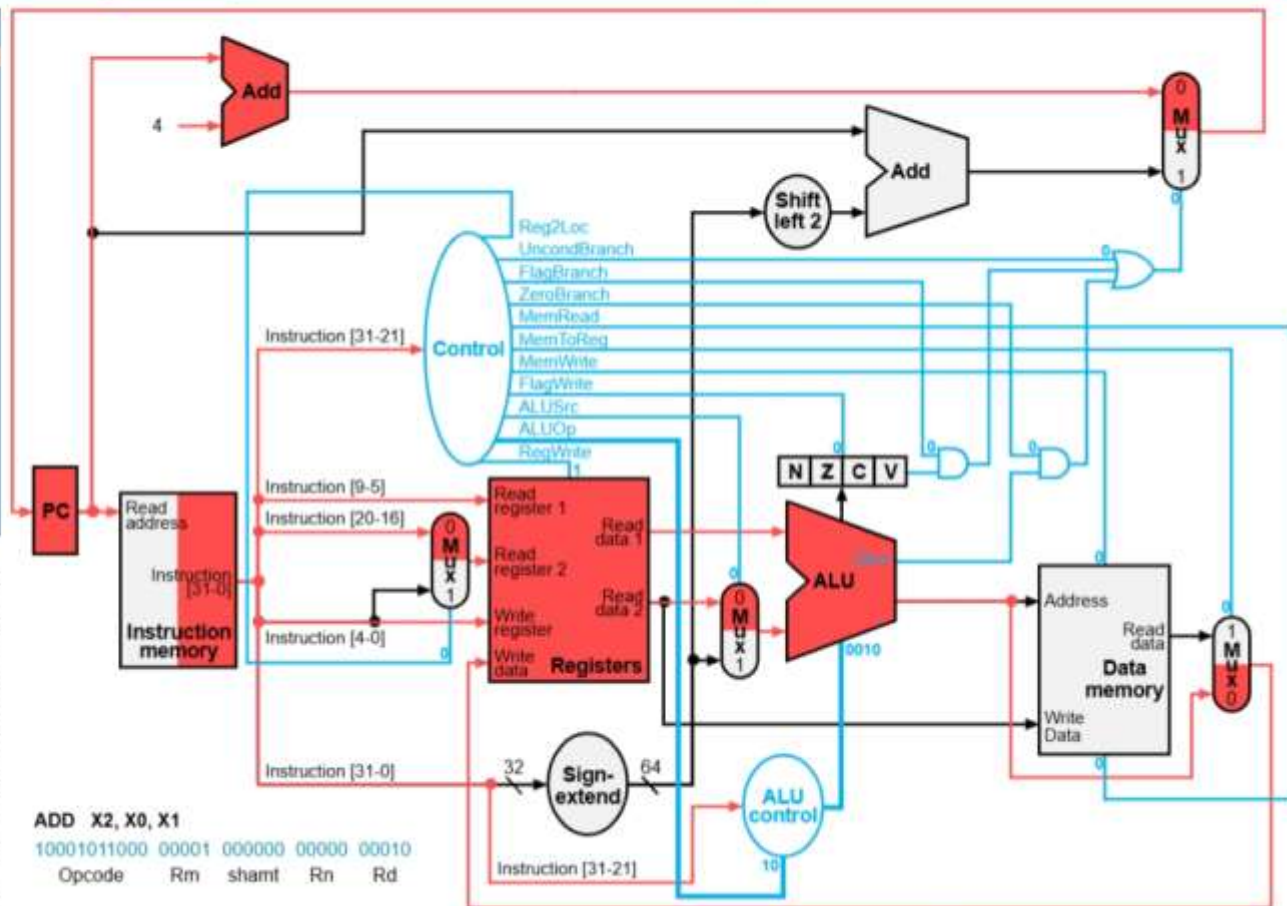
WHAT SIMULATOR, AND WHY?

LEGv8 Simulator

Execution Mode: **Single Cycle** Assemble Execute Instruction Help

1		ADDI	X0, XZR, #53
2		ADDI	X1, XZR, #75
3		ADD	X2, X0, X1
4		CMP	X2, X0
5		B.LE	1F
6	1f:	SUBIS	X3, X2, #9
7		MOV	X4, X3
8			

PC	0x40000c	Hex	Dec	Z	0	N	0	C	0	V	0
X0	0x35	Hex	Dec	X16	0x0					Hex	Dec
X1	0x4b	Hex	Dec	X17	0x0					Hex	Dec
X2	0x80	Hex	Dec	X18	0x0					Hex	Dec
X3	0x0	Hex	Dec	X19	0x0					Hex	Dec
X4	0x0	Hex	Dec	X20	0x0					Hex	Dec
X5	0x0	Hex	Dec	X21	0x0					Hex	Dec
X6	0x0	Hex	Dec	X22	0x0					Hex	Dec
X7	0x0	Hex	Dec	X23	0x0					Hex	Dec
X8	0x0	Hex	Dec	X24	0x0					Hex	Dec
X9	0x0	Hex	Dec	X25	0x0					Hex	Dec
X10	0x0	Hex	Dec	X26	0x0					Hex	Dec
X11	0x0	Hex	Dec	X27	0x0					Hex	Dec
X12	0x0	Hex	Dec	SP	0x7ffffffc					Hex	Dec
X13	0x0	Hex	Dec	FP	0x0					Hex	Dec
X14	0x0	Hex	Dec	LR	0x0					Hex	Dec
X15	0x0	Hex	Dec	XZR	0x0					Hex	Dec



FIXING AND RESTORING THE SIMULATOR

Restoration and development of
Arm's Java-based LEGv8 ISA simulator

COMPARISONS DON'T WORK!

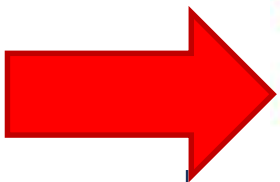
- No "if-else" conditionals
- No "switch-case" conditionals
- No "while" loops
- No "for" loops

THE COMPARISON BUG

```
1      ADDI      X0, XZR, #1
2      ADDI      X1, XZR, #2
3      CMP       X0, X1
4      B.LE      should_jump
5      ADD       X2, X0, X1
6  should_jump:  ADDI      X2, XZR, #16
7
```

X0 = 1, X1 = 2

PC	0x400008	Hex	Dec	Z	0	N	0	C	0	V	0
X0	0x1	Hex	Dec	X16				0x0		Hex	Dec
X1	0x2	Hex	Dec	X17				0x0		Hex	Dec
X2	0x0	Hex	Dec	X18				0x0		Hex	Dec

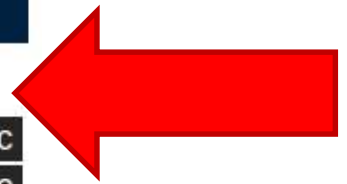


THE COMPARISON BUG

```
1      ADDI    X0, XZR, #1
2      ADDI    X1, XZR, #2
3      CMP     X0, X1
4      B.LE    should_jump
5      ADD     X2, X0, X1
6  should_jump:  ADDI    X2, XZR, #16
7
```

COMPARE X0 WITH X1

PC	0x40000c	Hex	Dec	Z	0	N	1	C	0	V	1
X0	0x1	Hex	Dec	X16		Hex	Dec			Hex	Dec
X1	0x2	Hex	Dec	X17		Hex	Dec			Hex	Dec
X2	0x0	Hex	Dec	X18		Hex	Dec			Hex	Dec



THE COMPARISON BUG

```
1      ADDI    X0, XZR, #1
2      ADDI    X1, XZR, #2
3      CMP     X0, X1
4      B.LE    should_jump
5      ADD     X2, X0, X1
6  should_jump:  ADDI    X2, XZR, #16
7
```

1 <= 2 ?

OF COURSE, LET'S JUMP!

PC	0x400010	Hex	Dec	Z	0	N	1	C	0	V	1
X0	0x1	Hex	Dec	X16			0x0	Hex	Dec		
X1	0x2	Hex	Dec	X17			0x0	Hex	Dec		
X2	0x0	Hex	Dec	X18			0x0	Hex	Dec		

THE COMPARISON BUG

```
1 |      ADDI      X0, XZR, #1
2 |      ADDI      X1, XZR, #2
3 |      CMP       X0, X1
4 |      B.LE      should_jump
5 |      ADD       X2, X0, X1
6 | should_jump:   ADDI      X2, XZR, #16
7 |
```

...OR NOT

PC	0x400014	Hex	Dec	Z	0	N	1	C	0	V	1
X0	0x1	Hex	Dec	X16			0x0	Hex	Dec		
X1	0x2	Hex	Dec	X17			0x0	Hex	Dec		
X2	0x3	Hex	Dec	X18			0x0	Hex	Dec		

THE FLAGS ARE SET WRONG!

HOW THEY SHOULD BE

\leq	B.LE	$\sim(Z=0 \ \& \ N=V)$
--------	------	------------------------

HOW THEY ARE

Z	0	N	1	C	0	V	1
---	---	---	---	---	---	---	---

$\neg(Z=0 \ \wedge \ N=V) = \neg(\text{TRUE} \ \wedge \ \text{TRUE}) = \text{FALSE}$

BRANCH AND LINKS DON'T WORK!

~~*void subroutine(arg1, ...)*~~
~~*float function(arg1, ...)*~~

CAN'T REUSE CODE

THE BRANCH AND LINK BUG

```
1      ADDI      X0, XZR, #1
2      BL       subroutine
3      B        exit
4
5  subroutine:
6      ADDI      X0, X0, #16
7      BR       LR
8  exit:
9
```

PROGRAM COUNTER: 0x0

RETURN ADDRESS: 0x0

THE BRANCH AND LINK BUG

```
1      ADDI      X0, XZR, #1
2      BL        subroutine
3      B         exit
4
5  subroutine:
6      ADDI      X0, X0, #16
7      BR        LR
8  exit:
9
```

PROGRAM COUNTER: 0x4

RETURN ADDRESS: 0x0

THE BRANCH AND LINK BUG

```
1      ADDI      X0, XZR, #1
2      BL       subroutine
3      B        exit
4
5  subroutine:
6      ADDI      X0, X0, #16
7      BR       LR
8  exit:
9
```

PROGRAM COUNTER: 0xC

RETURN ADDRESS: 0xC

THE BRANCH AND LINK BUG

```
1      ADDI      X0, XZR, #1
2      BL        subroutine
3      B         exit
4
5  subroutine:
6      ADDI      X0, X0, #16
7      BR        LR
8  exit:
9
```

PROGRAM COUNTER: 0x10

RETURN ADDRESS: 0xC

THE BRANCH AND LINK BUG

```
1      ADDI      X0, XZR, #1
2      BL        subroutine
3      B         exit
4
5  subroutine:
6      ADDI      X0, X0, #16
7      BR        LR
8  exit:
9
```

IT DOESN'T GO BACK!

IT SHOULD GO HERE!

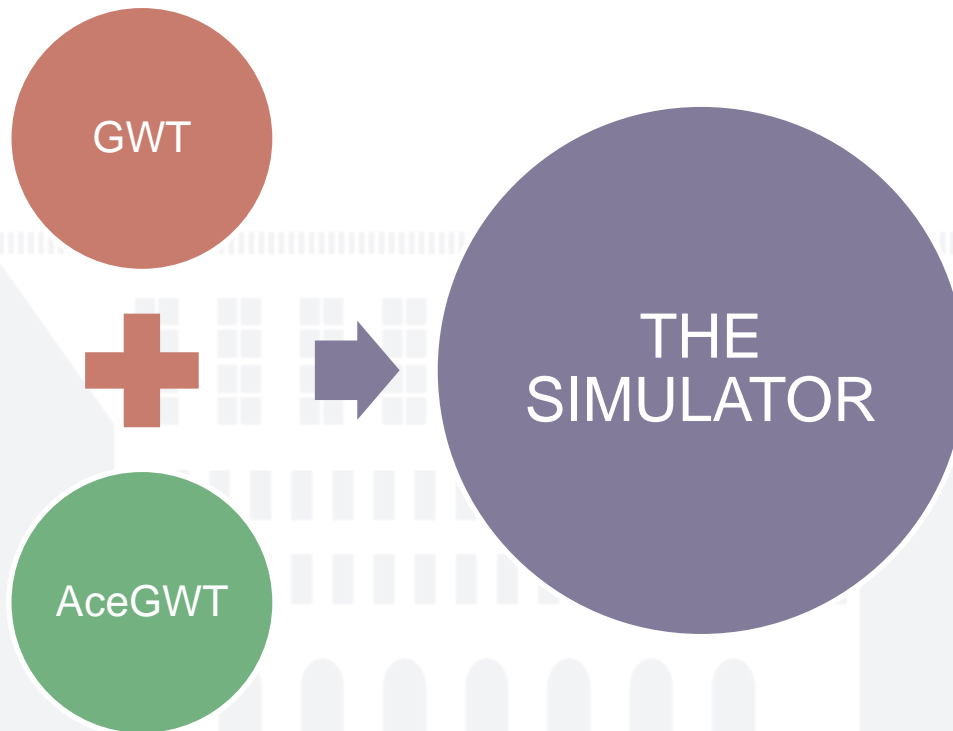
1	ADDI	X0, XZR, #1	
2	BL	subroutine	
3	B	exit	ADDRESS: 0x8
4			
5	subroutine:		
6	ADDI	X0, X0, #16	
7	BR	LR	
8	exit:		
9			

PROGRAM COUNTER: 0xC

RETURN ADDRESS: 0xC

ALL FIXED, BUT...
NOBODY KNOWS HOW IT WORKS!

THE PROJECT'S DEPENDENCIES



GWT

- Framework (formerly) from Google
- Generates web applications (client-server, client only) from Java
- Emulates Java's JVM with JavaScript

GWT

- Old, outdated, barely supported
- Convoluted custom build tools
- Limited emulation of JVM
- Basically needs Eclipse plug-in for real development

AceGWT

- Provides GWT bindings for the Ace editor
- Can be used like normal GWT component
- Also old, outdated, and unsupported

WORKING IT OUT:

- Need old version of Eclipse, and Eclipse GWT plug-in
- Reverse engineer the dependencies and where they are needed
- Configure the project to stop failing

FILLING THE GAPS

Restoration and development of
Arm's Java-based LEGv8 ISA simulator

WHAT IS THE SIMULATOR MISSING?

- Incomplete integer arithmetic
- No IEEE-754 arithmetic and data instructions
- No visualization for the stack memory

THE MISSING INTEGER-BASED INSTRUCTIONS

- **MUL** — LOWER 64 BITS OF THE MULTIPLICATION
- **SMULH** — HIGHER 64 BITS OF THE SIGNED MULTIPLICATION
- **UMULH** — HIGHER 64 BITS OF THE UNSIGNED MULTIPLICATION
- **SDIV** — SIGNED DIVISION
- **UDIV** — UNSIGNED DIVISION
- **LDA** — LOAD ADDRESS OF A LABEL IN A REGISTER

UMULH: A CASE STUDY

- Takes two 64-bit unsigned integer values
- Extends them to 128 bits unsigned
- Performs 128-bit product
- Saves higher 64 bits to destination

PROBLEM 1

- Java does not have primitive 128-bit integer types
- Product of 64-bit integers truncated
- The BigInteger library exists
- GWT 2.7 doesn't emulate it (2.8 does)

PROBLEM 2

- Primitive integers are signed
- BigInteger also signed
- Bitmask converts 64-bit unsigned integers to 65-bit signed, perform signed multiplication, take the higher bits

CAN'T SEE THE STACK

- Fundamental for testing and debugging complex programs (now we can write them)
- Useful to understanding LEGv8 and stack management
- Visible in most simulators

THE INTEGER REGISTERS VIEW

X0	0x0	Hex	Dec	X16	0x0	Hex	Dec
X1	0x0	Hex	Dec	X17	0x0	Hex	Dec
X2	0x0	Hex	Dec	X18	0x0	Hex	Dec
X3	0x0	Hex	Dec	X19	0x0	Hex	Dec
X4	0x0	Hex	Dec	X20	0x0	Hex	Dec
X5	0x0	Hex	Dec	X21	0x0	Hex	Dec
X6	0x0	Hex	Dec	X22	0x0	Hex	Dec
X7	0x0	Hex	Dec	X23	0x0	Hex	Dec
X8	0x0	Hex	Dec	X24	0x0	Hex	Dec
X9	0x0	Hex	Dec	X25	0x0	Hex	Dec
X10	0x0	Hex	Dec	X26	0x0	Hex	Dec
X11	0x0	Hex	Dec	X27	0x0	Hex	Dec
X12	0x0	Hex	Dec	SP	0x7fffffff	Hex	Dec
X13	0x0	Hex	Dec	FP	0x0	Hex	Dec
X14	0x0	Hex	Dec	LR	0x0	Hex	Dec
X15	0x0	Hex	Dec	XZR	0x0	Hex	Dec

THE NEW STACK VIEW

0x8000000000:	0x0	Hex	0x7fffffff80:	0x0	Hex
0x7fffffff8:	0x0	Hex	0x7fffffff78:	0x0	Hex
0x7fffffff0:	0x0	Hex	0x7fffffff70:	0x0	Hex
0x7fffffff8:	0x0	Hex	0x7fffffff68:	0x0	Hex
0x7fffffff80:	0x0	Hex	0x7fffffff60:	0x0	Hex
0x7fffffff8:	0x0	Hex	0x7fffffff58:	0x0	Hex
0x7fffffff80:	0x0	Hex	0x7fffffff50:	0x0	Hex
0x7fffffff8:	0x0	Hex	0x7fffffff48:	0x0	Hex
0x7fffffff80:	0x0	Hex	0x7fffffff40:	0x0	Hex
0x7fffffff8:	0x0	Hex	0x7fffffff38:	0x0	Hex
0x7fffffff80:	0x0	Hex	0x7fffffff30:	0x0	Hex
0x7fffffff8:	0x0	Hex	0x7fffffff28:	0x0	Hex
0x7fffffff80:	0x0	Hex	0x7fffffff20:	0x0	Hex
0x7fffffff8:	0x0	Hex	0x7fffffff18:	0x0	Hex
0x7fffffff80:	0x0	Hex	0x7fffffff10:	0x0	Hex
0x7fffffff88:	0x0	Hex	0x7fffffff08:	0x0	Hex

ADDING FLOATING-POINT SUPPORT

- **FADDS, FADDD** — ADD TWO IEEE-754 VALUES
- **FSUBS, FSUBD** - SUBTRACT TWO IEEE-754 VALUES
- **FMULS, FMULD** - MULTIPLY TWO IEEE-754 VALUES
- **FDIVS, FDIVD** — DIVIDE TWO IEEE-754 VALUES
- **LDURS, LDURD** — LOAD IEEE-754 VALUE FROM MEMORY
- **STURS, STURD** - STORE IEEE-754 VALUE TO MEMORY
- **FCMPS, FCMPPD** — COMPARE TWO IEEE-754 VALUES

ARITHMETICAL INSTRUCTIONS (FADDD, FDIVS, ...)

- Native Java support for IEEE-754 with float and double types
- Native Java support for IEEE-754 arithmetical operations
- Straight forward implementation

MEMORY ACCESS INSTRUCTIONS (LDURS, STURD, ...)

- Simulator uses long values to store bits in memory
- Use existing *longs* and *ints* as raw bits
- Use Java *Double.longBitsToDouble* and *Double.doubleToLongBits* to convert before memory

COMPARISON INSTRUCTIONS (FCMPS, FCMPD)

- LEGv8 does not specify flag-setting conditions for IEEE-754 comparisons
- Use ARMv8's ones

IEEE-754 Relationship	ARM APSR Flags			
	N	Z	C	V
Equal	0	1	1	0
Less Than	1	0	0	0
Greater Than	0	0	1	0
Unordered (<i>At least one argument was NaN.</i>)	0	0	1	1

SHOWING THE REGISTERS

S0

0xc00a0000

Hex

Dec

S0

-2.15625

Hex

Dec

THE FINAL VIEW

1	MOVK	X0, #192	0xc0000000	0x0	Hex	0xc7ffffff00	0x0	Hex	
2	LSL	X0, X0, #8	0xc7ffffff80	0x0	Hex	0xc7ffffff78	0x0	Hex	
3	MOVK	X0, #10	0xc7ffffff00	0xc0000000	Hex	0xc7ffffff70	0x0	Hex	
4	LSL	X0, X0, #16	0xc7ffffffe0	0x0	Hex	0xc7ffffff68	0x0	Hex	
5	SUBI	SP, SP, #16	0xc7ffffffd0	0x0	Hex	0xc7ffffff60	0x0	Hex	
6	STUR	X0, [SP, #0]	0xc7ffffffc0	0x0	Hex	0xc7ffffff58	0x0	Hex	
7	LDURW	S0, [SP, #0]	0xc7ffffffb0	0x0	Hex	0xc7ffffff50	0x0	Hex	
8			0xc7ffffffa0	0x0	Hex	0xc7ffffff48	0x0	Hex	
			0xc7ffffff90	0x0	Hex	0xc7ffffff40	0x0	Hex	
			0xc7ffffff80	0x0	Hex	0xc7ffffff38	0x0	Hex	
			0xc7ffffff70	0x0	Hex	0xc7ffffff30	0x0	Hex	
			0xc7ffffff60	0x0	Hex	0xc7ffffff28	0x0	Hex	
			0xc7ffffff50	0x0	Hex	0xc7ffffff20	0x0	Hex	
			0xc7ffffff40	0x0	Hex	0xc7ffffff18	0x0	Hex	
			0xc7ffffff30	0x0	Hex	0xc7ffffff10	0x0	Hex	
			0xc7ffffff20	0x0	Hex	0xc7ffffff00	0x0	Hex	

Dec	Z	N	C	V	DO	0xc0000000	Hex	Dec	D16	0x0	Hex	Dec	S0	-2,15625	Hex	Dec	S16	0x0	Hex	Dec	S17	0x0	Hex	Dec
Dec	X16	0x0	Hex	Dec	D01	0x0	Hex	Dec	D17	0x0	Hex	Dec	S1	0x0	Hex	Dec	S17	0x0	Hex	Dec	S17	0x0	Hex	Dec
Dec	X17	0x0	Hex	Dec	D02	0x0	Hex	Dec	D18	0x0	Hex	Dec	S2	0x0	Hex	Dec	S18	0x0	Hex	Dec	S18	0x0	Hex	Dec
Dec	X18	0x0	Hex	Dec	D03	0x0	Hex	Dec	D19	0x0	Hex	Dec	S3	0x0	Hex	Dec	S19	0x0	Hex	Dec	S19	0x0	Hex	Dec
Dec	X19	0x0	Hex	Dec	D04	0x0	Hex	Dec	D20	0x0	Hex	Dec	S4	0x0	Hex	Dec	S20	0x0	Hex	Dec	S20	0x0	Hex	Dec
Dec	X20	0x0	Hex	Dec	D05	0x0	Hex	Dec	D21	0x0	Hex	Dec	S5	0x0	Hex	Dec	S21	0x0	Hex	Dec	S21	0x0	Hex	Dec
Dec	X21	0x0	Hex	Dec	D06	0x0	Hex	Dec	D22	0x0	Hex	Dec	S6	0x0	Hex	Dec	S22	0x0	Hex	Dec	S22	0x0	Hex	Dec
Dec	X22	0x0	Hex	Dec	D07	0x0	Hex	Dec	D23	0x0	Hex	Dec	S7	0x0	Hex	Dec	S23	0x0	Hex	Dec	S23	0x0	Hex	Dec
Dec	X23	0x0	Hex	Dec	D08	0x0	Hex	Dec	D24	0x0	Hex	Dec	S8	0x0	Hex	Dec	S24	0x0	Hex	Dec	S24	0x0	Hex	Dec
Dec	X24	0x0	Hex	Dec	D09	0x0	Hex	Dec	D25	0x0	Hex	Dec	S9	0x0	Hex	Dec	S25	0x0	Hex	Dec	S25	0x0	Hex	Dec
Dec	X25	0x0	Hex	Dec	D10	0x0	Hex	Dec	D26	0x0	Hex	Dec	S10	0x0	Hex	Dec	S26	0x0	Hex	Dec	S26	0x0	Hex	Dec
Dec	X26	0x0	Hex	Dec	D11	0x0	Hex	Dec	D27	0x0	Hex	Dec	S11	0x0	Hex	Dec	S27	0x0	Hex	Dec	S27	0x0	Hex	Dec
Dec	X27	0x0	Hex	Dec	D12	0x0	Hex	Dec	D28	0x0	Hex	Dec	S12	0x0	Hex	Dec	S28	0x0	Hex	Dec	S28	0x0	Hex	Dec
Dec	SP	0xc7ffffff80	Hex	Dec	D13	0x0	Hex	Dec	D29	0x0	Hex	Dec	S13	0x0	Hex	Dec	S29	0x0	Hex	Dec	S29	0x0	Hex	Dec
Dec	FP	0x0	Hex	Dec	D14	0x0	Hex	Dec	D30	0x0	Hex	Dec	S14	0x0	Hex	Dec	S30	0x0	Hex	Dec	S30	0x0	Hex	Dec
Dec	LR	0x0	Hex	Dec	D15	0x0	Hex	Dec	D31	0x0	Hex	Dec	S15	0x0	Hex	Dec	S31	0x0	Hex	Dec	S31	0x0	Hex	Dec
Dec	X28	0x0	Hex	Dec																				

THE CHERRY ON TOP: MODERNIZING THE BUILD SYSTEM

INTEGRATING MAVEN

- Latest GWT and AceGWT support Maven
- Integrated Maven into the simulator
- Can now use other IDEs, Java 21, GWT 2.11
- To develop, download the code and run *mvn package*. That's it.

CONCLUSIONS

- Arm's LEGv8 simulator finally working
- Only one to implement every LEGv8 instruction
- Can now be developed with modern tools, set-up and build in seconds

THANK YOU FOR
YOUR ATTENTION

[THESIS AVAILABLE HERE](#)

[SIMULATOR AVAILABLE HERE](#)