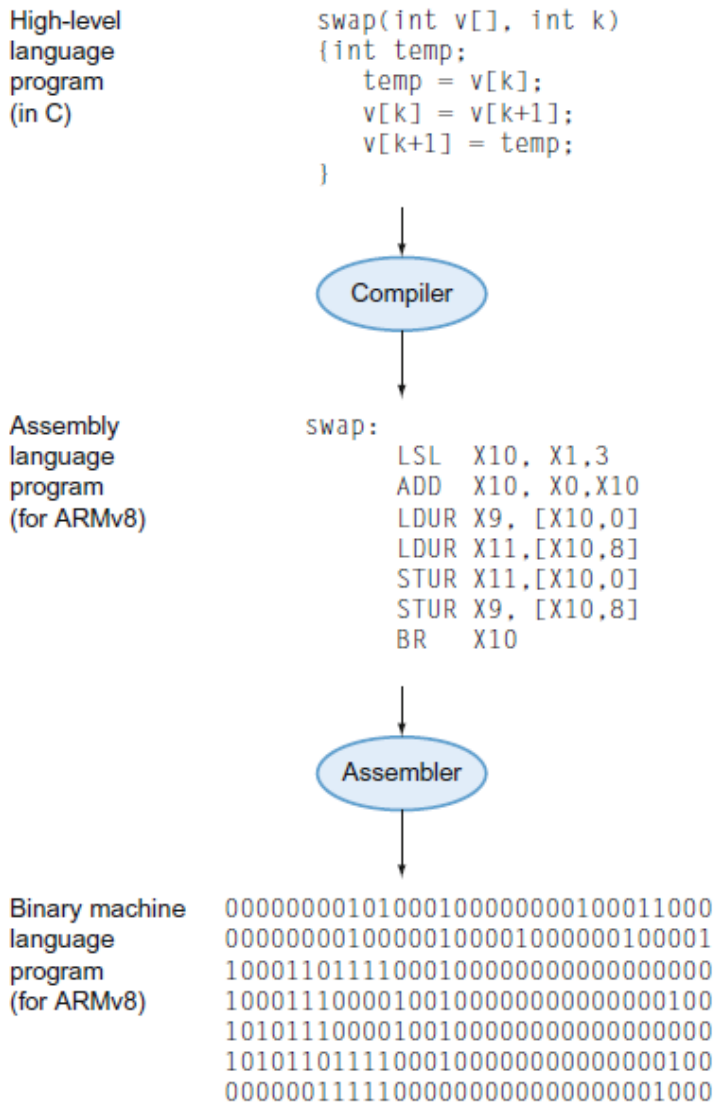
Chapter 2

Instructions: Language of the Computer

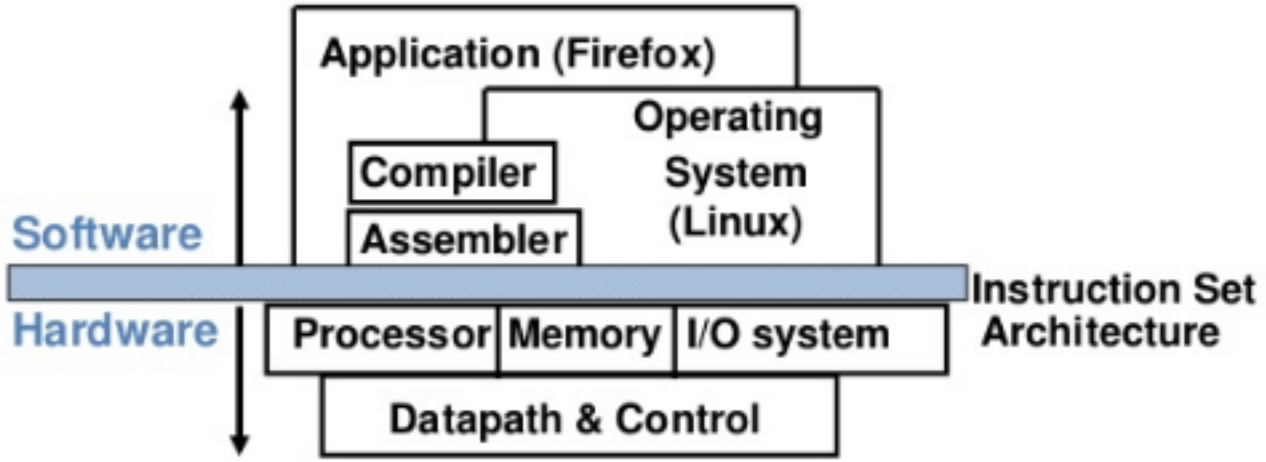
Instruction Sets

1. High-level language
   - Level of abstraction closer to problem domain
   - Provides for productivity and portability
2. Assembly language
   - Textual representation of instructions
3. Hardware representation
   - Binary digits (bits)
   - Encoded instructions and data
     - What is ARM instruction size in bits?



Instruction Sets

- Instruction Set is the full supply of instructions of a computer architecture
- Different computers have different instruction sets (different architectures)
  - But with many aspects in common



Instruction Sets

- The **instruction set**, also called **instruction set architecture** (**ISA**), is part of a computer architecture related to programming.
- The instruction set provides commands to the processor to tell it what it needs to do.
- CISC-based and RISC-based Architecture
  1. Complex Instruction Set Computing (CISC)
     - *X86 (*based on the Intel 8086 CPU)
  2. Reduced Instruction Set Computing (RISC)
     - *MIPS*
     - *ARM*
  3. RISC –V
     - *A free and open ISA based on RISC*
     - *Allowing anyone to design, manufacture and sell RISC-V chips or software*

Instruction Sets

- CISC
  - MULT mem[0] mem[1]
    - Multiply the content of memory location 0 by the content of memory location 1 and store the product in memory location 0
- RISC
  - LOAD R0, mem[0]
  - LOAD R1, mem[1]
  - PROD R0, R1
  - STORE R0, mem[0]
- RISC processors only use simple instructions that can be executed in fewer clock cycles.
  - Thus, the "MULT" command described above was divided into three separate commands: LOAD, PROD and STORE.

ARM Instruction Sets

- **ARM**, developed by British Co. **ARM Holdings** (originally, Acorn Risk Machine, aka Advanced Risk Machine).
  - ARM is a family of RISC-based instruction set architecture (ISA) for computer processors
  - Companies that make chips that implement an ARM architecture include:
    - *Analog Devices, Apple, AppliedMicro, Atmel, Broadcom, Cypress Semiconductor, Freescale Semiconductor, Nvidia, NXP, Qualcomm, Renesas, Samsung Electronics, ST Microelectronics* and *TI.*

ARM Instruction Sets

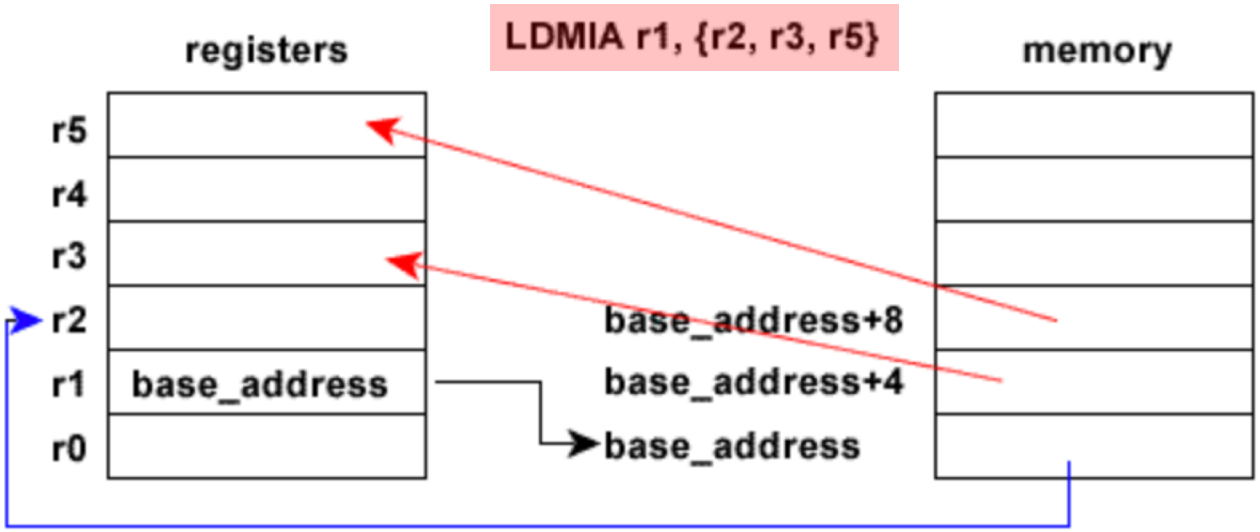
- **ARM Instruction Set (RISC-based)**
  - Require significantly fewer transistors than CISC x86 processors found in most personal computers (Intel, AMD, etc.).
    - Reduces costs, heat and power use.
    - Such reductions are desirable traits for embedded systems and smaller devices, **etc.**
  - Instructions are 32 bits wide
  - Other RISC-based ISA
    - MIPS version of RISC
  - “Load/Store” Architecture
    - ARM processor must first load data into one of the general-purpose registers before processing it

ARM Instruction Sets

- ARM version of RISC focuses not only on performance but also on
  - High code density (provides a 16-bit ISA as well)
  - Low power
  - Small die size
- Achieving it all through modifying RISC rules to also include
  1. Variable-cycle execution for certain instructions,
  2. An inline barrel shifter to preprocess one of the input registers,
  3. Conditional execution,
  4. A compressed 16-bit Thumb instruction set,
  5. And some enhanced DSP instructions.

ARM Instruction Sets

1. Variable-cycle execution for certain instructions

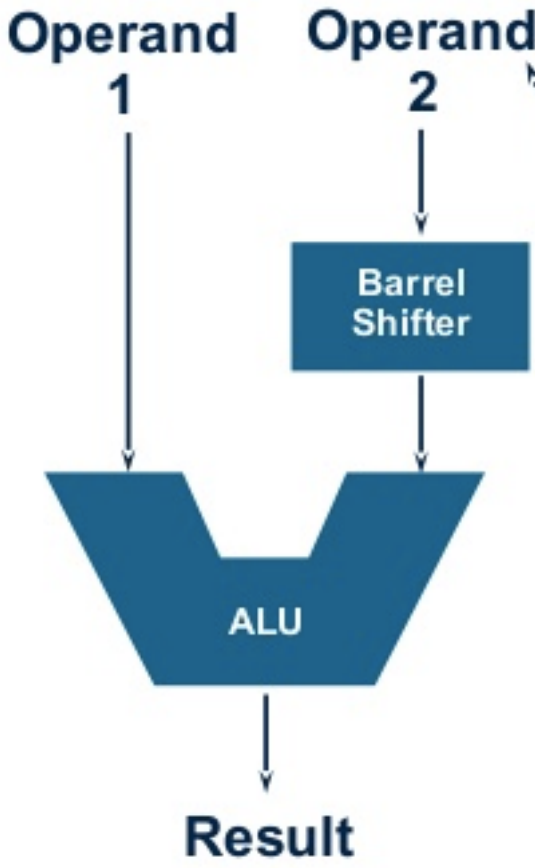


LDMIA r1, {r2, r3, r5} = LDR r2, [r1]
LDR r3, [r1, 4]
LDR r5, [r1, 8]

ARM Instruction Sets

2. An inline barrel shifter to preprocess one of the input registers (Operand2).

```
• MOV r0, r0, LSL #1
• MOV r1, r1, LSR #2
• MOV r2, r2, ASR #2
• MOV r3, r3, ROR #16
• ADD r4, r4, r4, LSL #4
• RSB r5, r5, r5, LSL #5
```



ARM Instruction Sets

3. Conditional execution

- An instruction with a condition code is only executed if the condition code flags in the xPSR meet the specified condition.

Normal	Conditional
CMP r3,#0	CMP r3,#0
BEQ skip	ADDNE r0,r1,r2
ADD r0,r1,r2	
skip	

Conditional execution improves code density and performance by reducing the number of forward branch instructions.

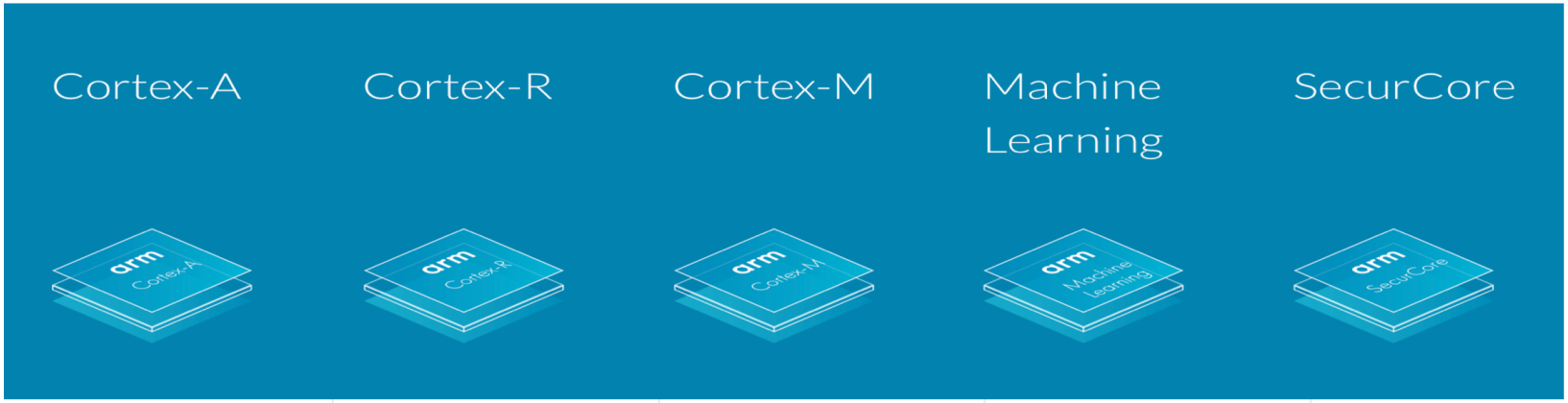
- Each data processing instruction prefixed by one of the 16 condition codes

EQ	equal	MI	negative	HI	unsigned higher	GT	signed greater than
NE	not equal	PL	positive or zero	LS	unsigned lower or same	LE	signed less than or equal
CS	unsigned higher or same	VS	overflow	GE	signed greater than or equal	AL	always
CC	unsigned lower	VC	no overflow	LT	signed less than	NV	special purpose

ARM Instruction Sets

4. Compressed 16-bit Thumb instruction set
   - All ARM instructions are 32 bits long for easy decoding and pipelining
     - at the cost of decreased code density
   - Additional modes allow better code density
     - Thumb (16 bit), Thumb 2 (16/ 32 bit), Jazelle (Byte code)

ARM Instruction Sets



Highest Performance

Supreme performance at optimal power

Real-Time Processing

Reliable mission-critical performance

Lowest Power, Lower Cost

Powering the most energy efficient embedded devices

Efficiency Uplift for All Devices

Project Trillium for unmatched versatility and scalability

Tamper Resistant

Powerful solutions for security applications

ARM Instruction Sets

Cortex-A	Cortex-R	Cortex-M	Machine Learning	SecurCore
				
Example use cases:	**Example use cases:**	**Example use cases:**	**Example use cases:**	**Example use cases:**
Automotive	Automotive	Automotive	Artificial intelligence	Advanced payment systems
Industrial	Cameras	Energy grid	Augmented reality	Electronic passports
Medical	Industrial	Medical	Edge computing	SIM
Modem	Medical	Secure embedded applications	Neural network frameworks	Smart cards
Storage		Smart cards	Object detection	
		Smart devices	Virtual reality	
		Sensor fusion		
		Wearables		

MK

The ARMv8 Instruction Set

A short ARM processors list:

Classic: With DSP, Floating Point, TrustZone and Jazelle extensions. **(2001)**

Cortex-M: ARM Thumb®-2 technology which provides excellent code density. With Thumb-2 technology, the Cortex-M processors support a fundamental base of 16-bit Thumb instructions, extended to include more powerful 32-bit instructions. First Multi-core. **(2004)**

Cortex-R: ARMv7 Deeply pipelined micro-architecture,Flexible Multi-Processor Core (MPCore) configurations:Symmetric Multi-Processing (SMP) & Asymmetric Multi-Processing (AMP), LPAE extension. . **(2011)**

Cortex-A50: ARMv8-A 64bit with load-acquire and store-release features , which are an excellent match for the C++11, C11 and Java memory models. **(2012)**

The ARMv8 Instruction Set

ARMv8 introduced a number of profiles or flavor of the architecture that targets certain classes of workloads.

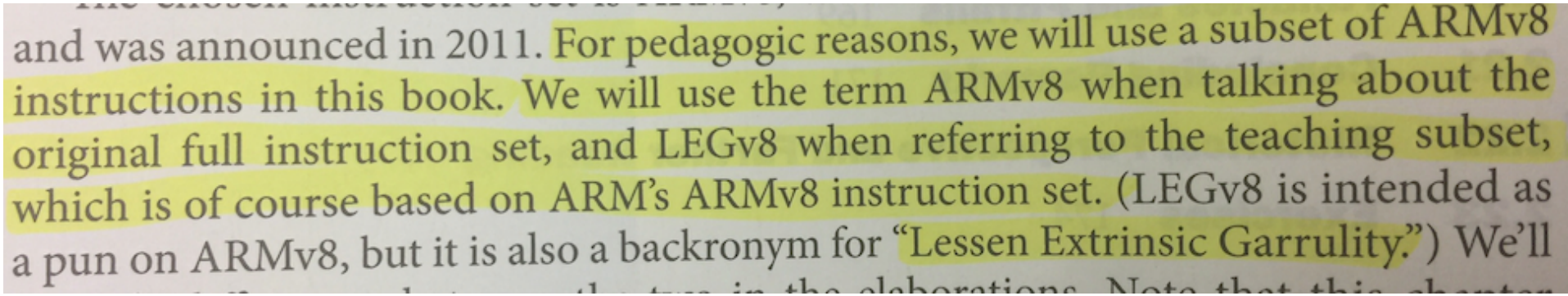
Profile	Name	Description
Application	ARMv8-A	Optimized for a large class of general applications for mobile, tablets, and servers.
Real-Time	ARMv8-R	Optimized for safety-critical environments.
Microcontroller	ARMv8-M	Optimized for embedded systems with a highly deterministic operation.

- https://www.youtube.com/watch?v=IfHG7bj-CEI

Intel is in serious trouble. ARM is the Future.
Dec 21, 2018

The ARMv8 Instruction Set

- A subset ARMv8 instructions, called LEGv8, used as the example throughout the book.
  - ARM
    - Advanced RISC Machine (originally, Acorn RISC Machine)
  - LEG
    - Lessen Extrinsic Garrulity (less talk about non-essentials)


and was announced in 2011. For pedagogic reasons, we will use a subset of ARMv8 instructions in this book. We will use the term ARMv8 when talking about the original full instruction set, and LEGv8 when referring to the teaching subset, which is of course based on ARM's ARMv8 instruction set. (LEGv8 is intended as a pun on ARMv8, but it is also a backronym for "Lessen Extrinsic Garrulity.") We'll

Arithmetic Operations

- Add and subtract, three operands
  - Two sources and one destination

```
ADD a, b, c  // a gets b + c
```

- All arithmetic operations have this form
- When it comes to ISA design simplicity favors regularity (Design Principle 1)

Arithmetic Example

- C code:

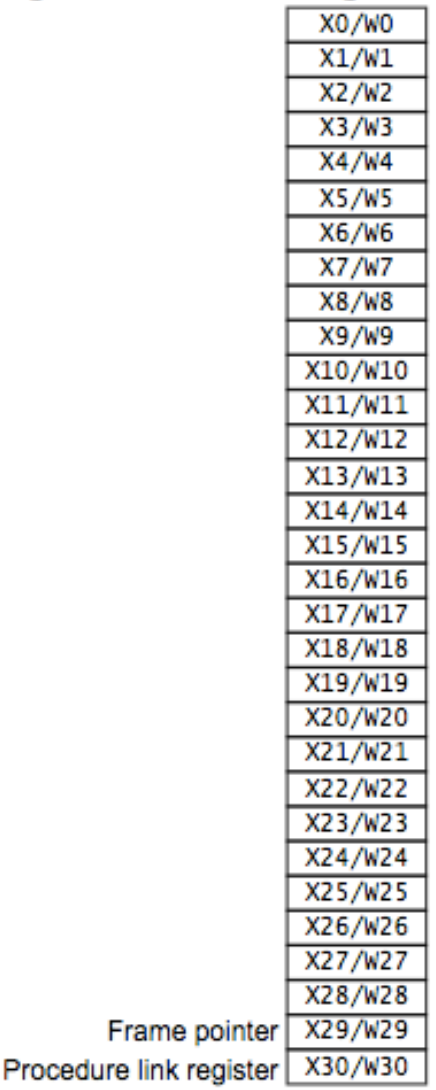
```
f = (g + h) - (i + j);
```

- Compiled LEGv8 code:
  - Assuming C variables in memory g, h, i, and j are already loaded in X0, X1, X2, and X3 registers:

```
ADD X9, X0, X1   // temp X9 = g + h

ADD X10, X2, X3   // temp X10 = i + j

SUB X19, X9, X10  // X19 = X9 - X10

STR X19, f      // mem[f] <== (g + h) - (i + j)
```

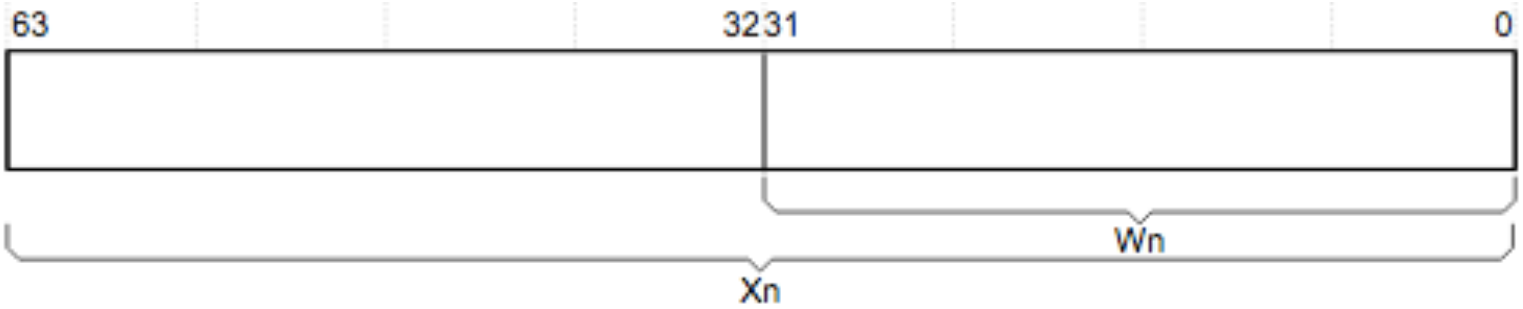
Register Operands

- Arithmetic instructions use register operands
- **LEGv8** has 32 registers each 64-bit long
  - A 32 x 64 bit register file
  - X registers are 64-bit long (extended/doubleword registers)
- **LEGv**8 has 32 registers each 32-bit long
  - W registers are 32-bit long (word registers)
- XZR and WZR (register 31)
  - Always contains the constant value 0 and can not be overwritten either



LEGv8 Registers

- The 32-bit W register forms the lower half of the corresponding 64-bit X register.
  - That is, W0 maps onto the lower word of X0, and W1 maps onto the lower word of X1.
- All reads from W registers disregard the higher 32 bits of the corresponding X register and leave them unchanged.
  - 0xFFEBCD00FFFFFABC will be read as 0xFFFFFABC
- All writes to W registers will set the higher 32 bits of the X register to zero.
  - Writing 0xFFFFFABC into W0 sets X0 to 0x00000000FFFFFABC.



LEGv8 Registers

- **X0 – X7: procedure arguments/results**
- X8: indirect result location register
- **X9 – X15: temporaries**
- X16 – X17 (IP0 – IP1): may be used by linker as a scratch register, other times as temporary register
- X18: platform register for platform independent code; otherwise a temporary register
- **X19 – X27: saved**
- **X28 (SP): stack pointer**
- X29 (FP): frame pointer
- **X30 (LR): link register (return address)**
- **XZR (register 31): the constant value 0**

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

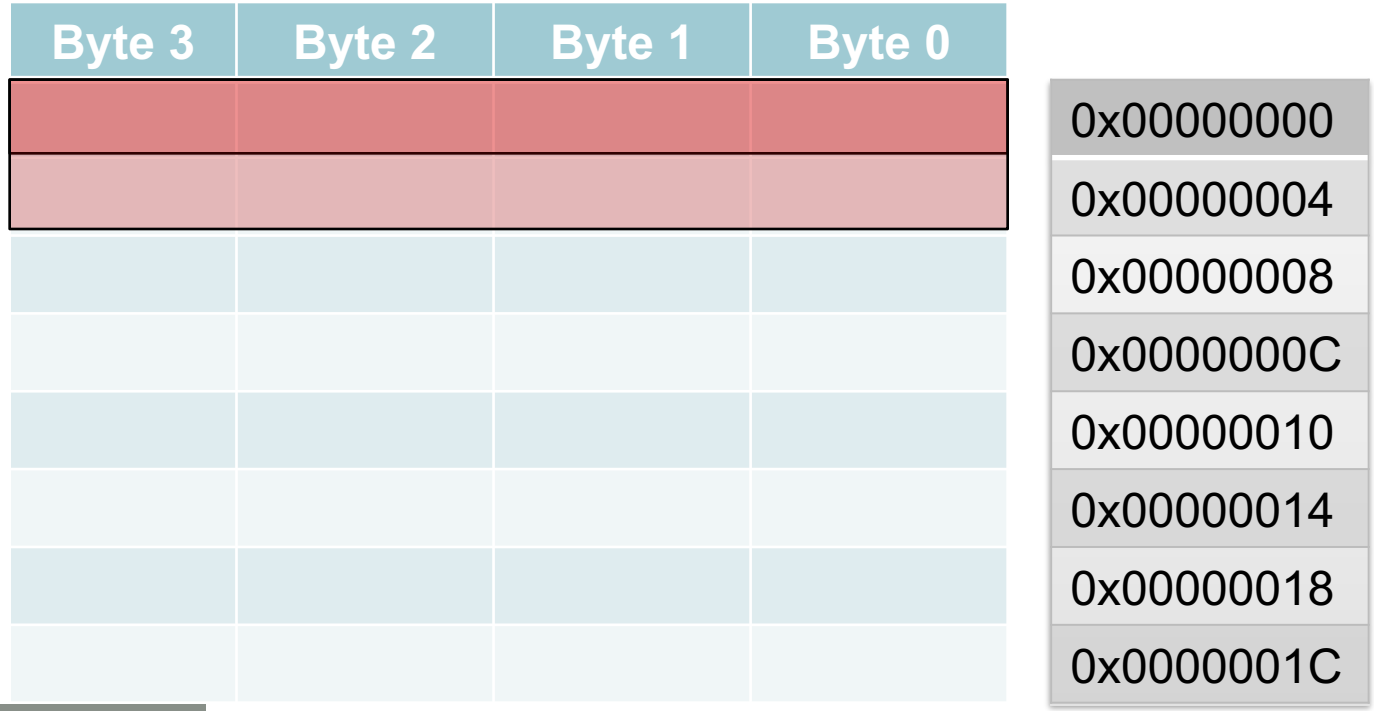
  - Assumptions:
    - g, h, i, and j in X0, X1, X2, X3 respectively.

- Compiled LEGv8 code:

```
ADD X9, X0, X1   // ADD result goes into temp X9
ADD X10, X2, X3  // ADD result goes into temp X10
SUB X19, X9, X10 // SUB result goes into saved X19
STR X19, f       // mem[f] <== (g + h) - (i + j)
```

Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- Memory is byte addressed
  - Each address identifies an 8-bit byte



Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- Memory is byte addressed
  - Each address identifies an 8-bit byte

32 bytes

Byte 7	Byte 6	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0

0x00000000
0x00000008
0x00000010
0x00000018

Memory Operands

- Before and after applying the arithmetic operations data transfer operations are used:
  - Load values from memory into registers
  - Store result from register to memory
    - LDR X0, [X1]
      - Unscaled no offset: *load contents of [X1] into X0*
    - LDR X0, [X1, 2] ;
      - Unscaled constant offset: *load contents of [X1 + 2] into X0*
    - LDR X0, [X1, X2] ;
      - Unscaled register offset: *load contents of [X1 + X2] into X0*
    - LDR X0, [X1, X2, LSL 3];
      - Scaled register offset: *load contents of [X1 + (X2 << 3)] into X0*
      - In scaled, you copy the content of X2 into barrel shifter to multiply it by 8 first before adding it to the address in X1 to calculate the memory address.

Memory Operands

- In LEGv8, we use **LDUR** to indicate "unscaled" operation of LDR is performed.
  - LDUR X0, [X1] ; or
  - LDUR X0, [X1, #0] ; *// Unscaled with no offset or offset 0*
  - LDUR X0, [X1, #2]: *// Unscaled with constant offset*
  - LDUR X0, [X1, X2]; *// Unscaled with register offset*
- In LEGv8, the scaled LDR instruction has been omitted and not used.
  - *Recall that LEGv8 only uses a subset of ARMv8 ISA*

Memory Operands

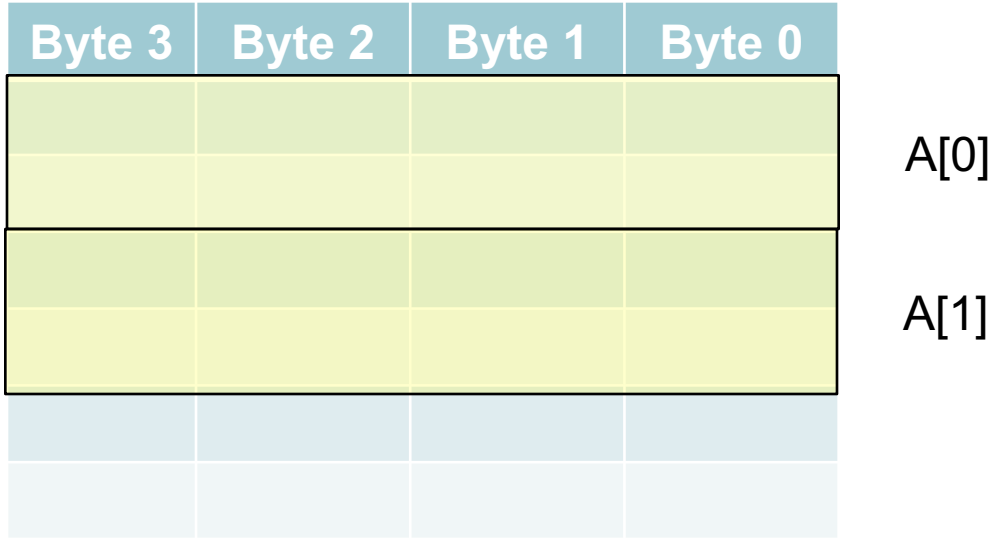
- In ARMv8 for scaled register offset LDR is used
  - **LDR X0, [X1, X2, LSL 3];** // load contents of [X1 + (X2 << 3)] into X0
- *Since LDR not available in LEGv8 then the scaling operation must be done outside of the load instruction as follows:*
- ***LSL X2, X2, 3***
- ***LDUR X0, [X1, X2]***

Memory Operand Example

- C code:

```
A[8] = h + A[4];
```

  - Assumptions:
    1. h in X0, and base address of A in X1
    2. A is an array of double words (each element in the A array is 8 bytes)

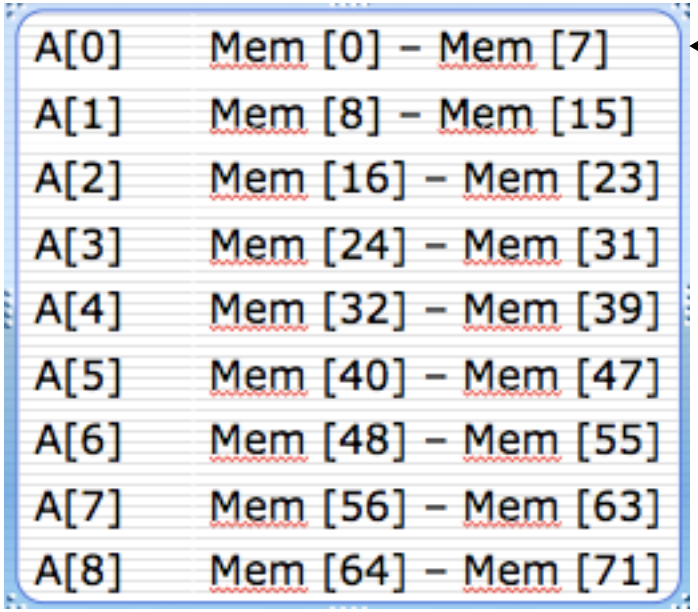


Memory Operand Example

A[8] = h + A[4]; // A is an array of doubleword (8 bytes each)
Note: Variable h is in X0, and base address of array A is in X1

```
LDUR    X9,[X1,#32] // U for "unscaled"
ADD     X9,X0,X9 // add the value found in A[4] to h
STUR    X9,[X1,#64] // store the result into A[8]
```

Base address of A



← A double word per entry

Registers vs. Memory

- Registers are faster to access than memory and require less energy
- Operating on data in memory requires loads and stores
- Compiler must use registers for variables as much as possible
  - Keep them in registers as long as possible
  - Only spill to memory for less frequently used variables or larger data sizes
- **Register optimization is important!**

Example of Register Optimization

- C code: newAge = age + years;

  - LDUR X0, age
  - LDUR X1, years
  - ADD X2, X0, X1
  - STUR X2, newAge

  - LDUR X0, age
  - LDUR X1, years
  - ADD X0, X0, X1
  - STUR X0, newAge

Immediate Operands

- Constant data specified in an instruction

  `ADDI X22, X22, 4`

  - Immediate operand avoids a load instruction
    - LDUR X21, 4
    - ADD X22, X21, X22
- Another ISA design principle:
  - Make the common case fast (***Design Principle 3***)
  - Small constants are common

The ARMv8 Instruction Set

ARMv8 introduced a number of profiles or flavor of the architecture that targets certain classes of workloads.

Profile	Name	Description
Application	ARMv8-A	Optimized for a large class of general applications for mobile, tablets, and servers.
Real-Time	ARMv8-R	Optimized for safety-critical environments.
Microcontroller	ARMv8-M	Optimized for embedded systems with a highly deterministic operation.

- https://www.youtube.com/watch?v=IfHG7bj-CEI

Intel is in serious trouble. ARM is the Future.
Dec 21, 2018

Kinds Of Data

- Numbers
  - Integers
    - Unsigned
    - Signed
  - Reals
    - Fixed-Point
    - Floating-Point
  - Binary-Coded Decimal
- Text
  - ASCII Characters
  - Strings
- Other
  - Graphics
  - Images
  - Video
  - Audio

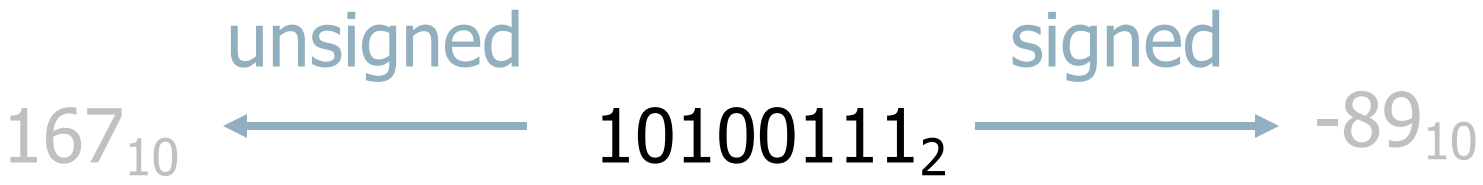
Numbers (recall)

1. Computers use binary numbers (0's and 1's).

2. Computers store and process numbers using a fixed number of digits (“fixed-precision”).

3. Computers represent signed numbers using 2's complement instead of the more natural “sign-plus-magnitude” representation.

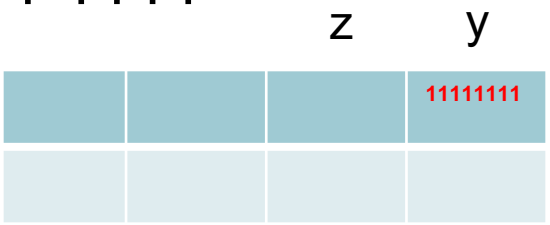
Signed vs. Unsigned (recall)



- Signed vs. unsigned is a matter of interpretation; thus a single bit pattern can represent two different values.
- Allowing both interpretations is useful
  - Some data (e.g., count, age) can never be negative, and having a greater range is useful.

Sign Extension (recall)

- Representing a number using more bits
  - But must preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit (1111 1111) to 16-bit
  - +255: 1111 1111 => 0000 0000 1111 1111
  - –1: 1111 1111 => 1111 1111 1111 1111
- In LEGv8 instruction set
  - LDURB: zero-extend loaded byte
  - LDURSB: sign-extend loaded byte



0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111

1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111 1111

Hexadecimal (recall)

- Base 16
  - Compact representation of bit strings
  - 4 bits per hex digit

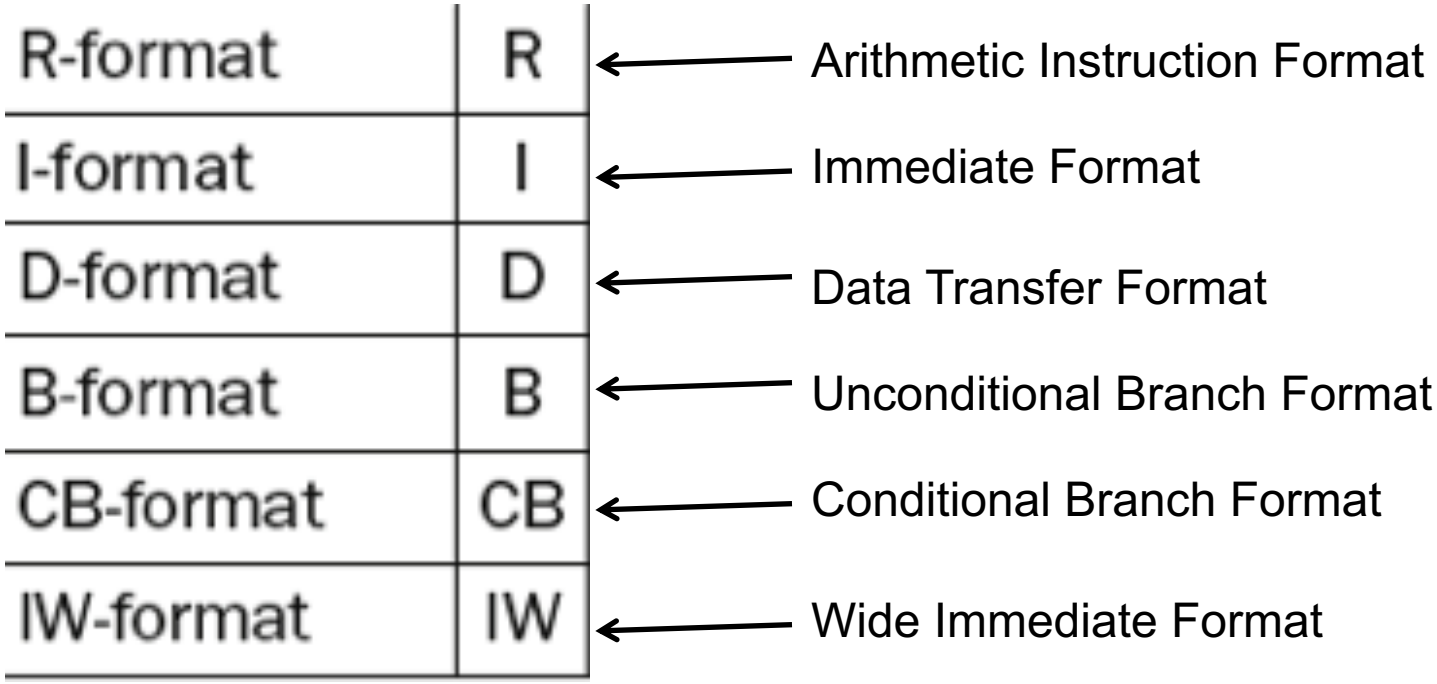
0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000
  - 1010000101001011010011101010101010

Representing Instructions

- Instructions are also encoded in binary
  - Called machine code
- LEGv8 instructions
  - Encoded as 32-bit instruction words
  - Small number of formats, encoding operation code (opcode), register numbers, …
  - Regularity!

LEGv8 Instructions Formats

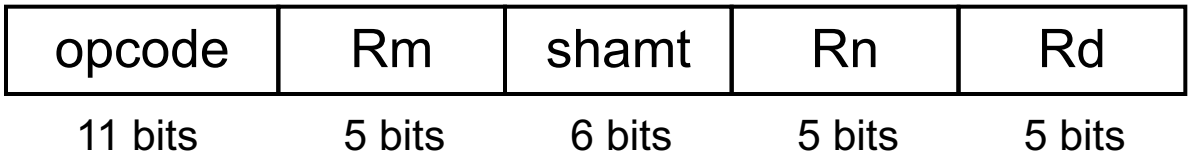


LEGv8 Instructions Formats

OPCODES IN NUMERICAL ORDER BY OPCODE ③

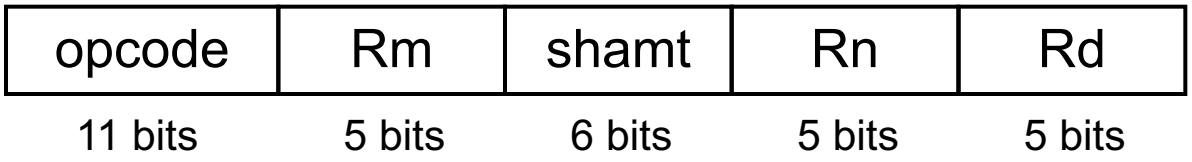
Instruction Mnemonic	Format	Opcode Width (bits)	Opcode Binary	Shamt Binary	11-bit Opcode Range (1) Start (Hex)	End (Hex)
B	B	6	000101 00000	- …11111	0A0	0BF
FMULS	R	11	00011110001	000010	0F1	
FDIVS	R	11	00011110001	000110	0F1	
FCMPS	R	11	00011110001	001000	0F1	
FADDS	R	11	00011110001	001010	0F1	
FSUBS	R	11	00011110001	001110	0F1	
FMULD	R	11	00011110011	000010	0F3	
FDIVD	R	11	00011110011	000110	0F3	
FCMPD	R	11	00011110011	001000	0F3	
FADDD	R	11	00011110011	001010	0F3	
FSUBD	R	11	00011110011	001110	0F3	
STURB	D	11	00111000000		1C0	
LDURB	D	11	00111000010		1C2	
B.cond	CB	8	01010100 000	-…111	2A0	2A7
STURH	D	11	01111000000		3C0	
LDURH	D	11	01111000010		3C2	
[illegible]	[illegible]	11	10001010000		450	

LEGv8 R-format Instructions



- Instruction fields
  - opcode: operation code
  - Rm: the second register source operand
  - shamt: shift amount (000000 for now)
  - Rn: the first register source operand
  - Rd: the register destination

R-format Example



ADD X9,X20,X21

1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}
10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_2$ =

$8B150289_{16}$

Logical Operations (recall)

Bitwise Operators

- Bitwise operators operate on individual bit positions within the operands.
- The result in any one bit position is entirely independent of all the other bit positions.

```
1  1 1
0  0 1 1   ADD
0  1 1 1
-----------
1  0 1 0
```

```
0  0 0 1   AND
0  1 1 1
-----------
0  0  0 1
```

Logical Operations (recall)

- Instructions for bitwise manipulation

Operation	C	Java	LEGv8
Shift left	<<	<<	LSL
Shift right	>>	>>>	LSR
Bit-by-bit AND	&	&	AND, ANDI
Bit-by-bit OR	|	|	OR, ORI
Bit-by-bit NOT	~	~	EOR, EORI

Logical Operations (recall)

- Example:

1011 0110 1010 0100 0011 1101 1001 1010

mask: 0000 0000 0000 0000 0000 1111 1111 1111

- The result of ANDing these:

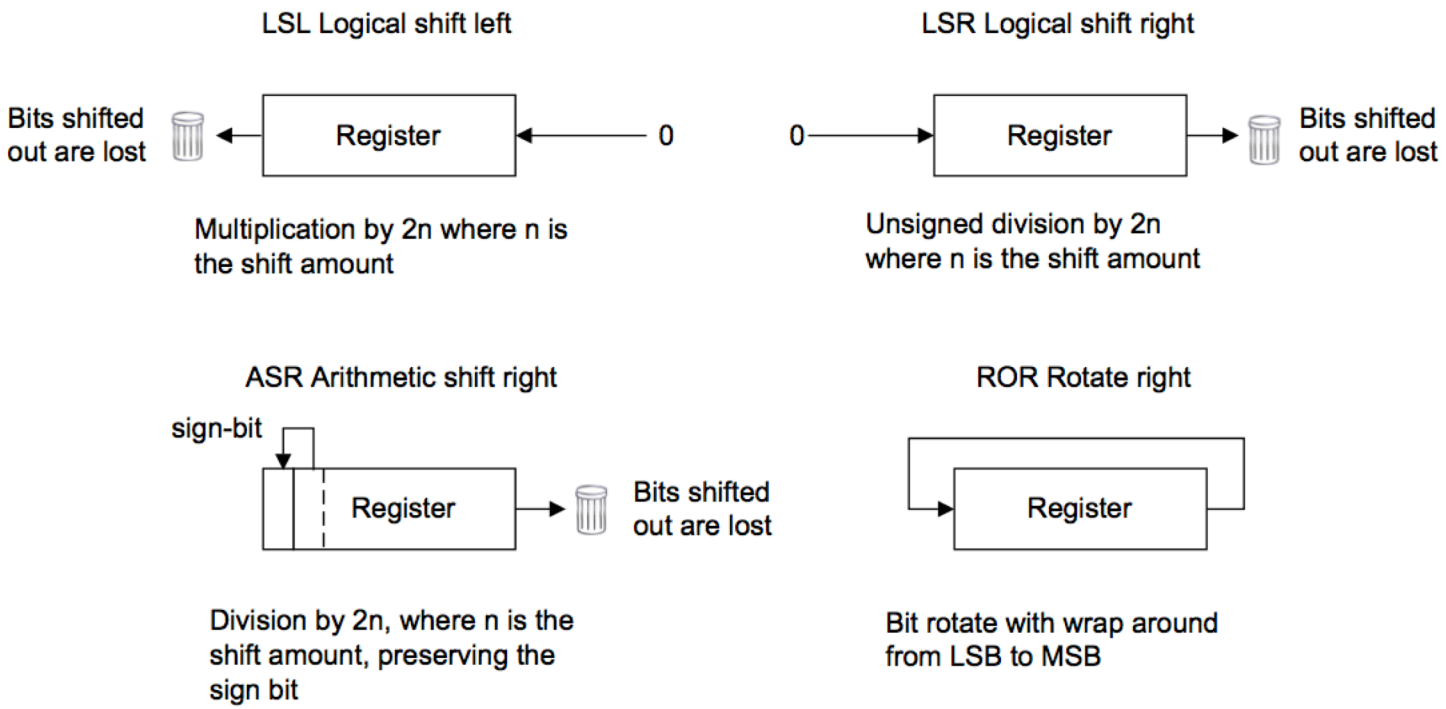
0000 0000 0000 0000 0000 1101 1001 1010

mask last 12 bits

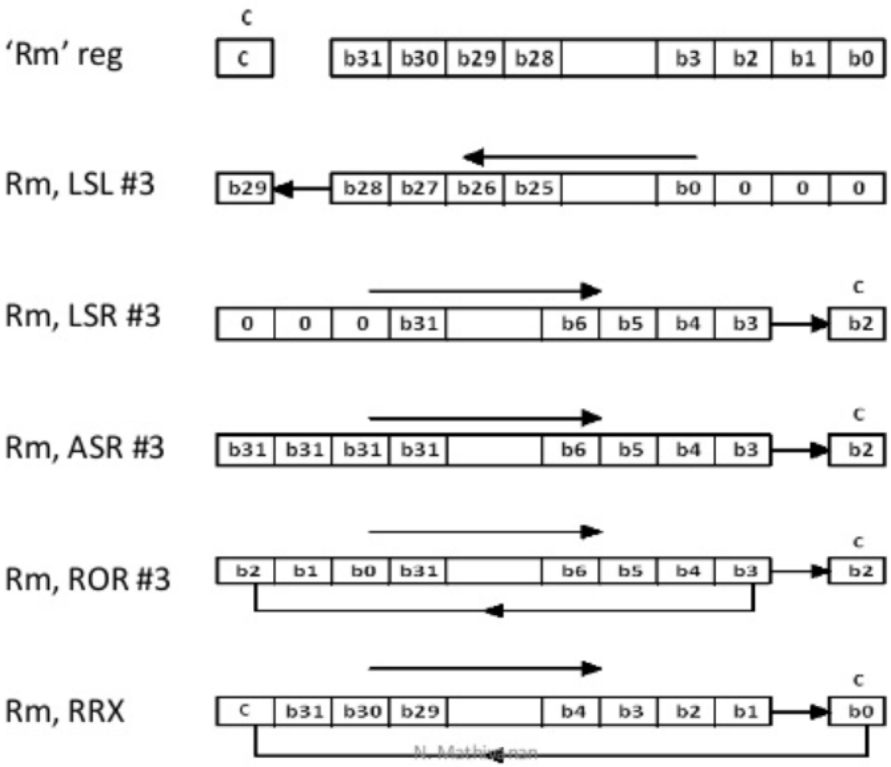
Using Bitwise OR to set the least significant half of a word:

(0x12345678 OR 0x000FFFF) results in 0x1234FFFF

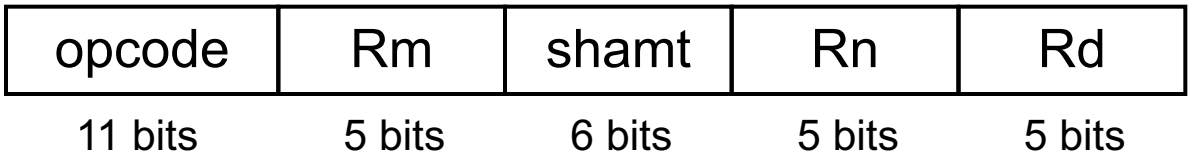
Shift Operations (recall)



Shift Operations (recall)

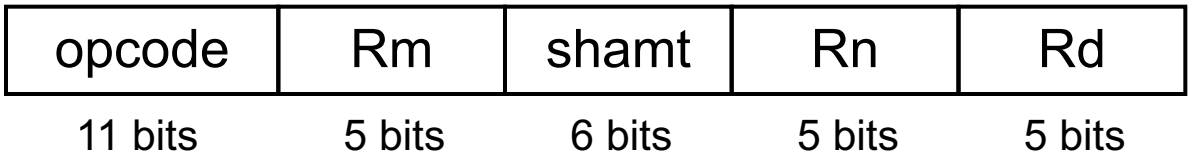


LEGv8 - Shift Instructions (R-Format)



- Shift machine language instructions use the R-format
- The *Rm* field is unused and is set to zero
- shamt: how many positions to shift
- The C language left-shift operator is “<<”
- The C language right-shift operator is “>>”

LEGv8 - Shift Instructions (R-Format)



- Shift left LEGv8 operator: LSL
- LSL X11,X19, 4 *// reg X11 = reg X19 << 4 bits*
  - Shift left and fill with 0 bits
  - LSL by *i* bits multiplies by 2^i

1691_{ten}	0_{ten}	4_{ten}	19_{ten}	11_{ten}
11010011011_{two}	00000_{two}	000100_{two}	10011_{two}	01001_{two}

LEGv8 - Shift Instructions (R-Format)

OPCODES IN NUMERICAL ORDER BY OPCODE ③

Instruction Mnemonic	Format	Opcode Width (bits)	Opcode Binary	Shamt Binary	11-bit Opcode Range (1) Start (Hex)	End (Hex)
B	B	6	000101		0A0	0BF
FMULS	R	11	00011110001	000010	0F1	
FDIVS	R	11	00011110001	000110	0F1	
FCMPS	R	11	00011110001	001000	0F1	
FADDS	R	11	00011110001	001010	0F1	
FSUBS	R	11	00011110001	001110	0F1	
FMULD	R	11	00011110011	000010	0F3	
FDIVD	R	11	00011110011	000110	0F3	
FCMPD	R	11	00011110011	001000	0F3	
FADDD	R	11	00011110011	001010	0F3	
FSUBD	R	11	00011110011	001110	0F3	
STURB	D	11	00111000000		1C0	
LDURB	D	11	00111000010		1C2	
B.cond	CB	8	01010100		2A0	2A7
STURH	D	11	01111000000		3C0	
LDURH	D	11	01111000010		3C2	
AND	R	11	10001010000		450	
ADD	R	11	10001011000		458	
ADDI	I	10	1001000100		488	489
ANDI	I	10	1001001000		490	491
BL	B	6	100101		4A0	4BF
SDIV	R	11	10011010110	000010	4D6	
UDIV	R	11	10011010110	000011	4D6	
MUL	R	11	10011011000	011111	4D8	
SMULH	R	11	10011011010		4DA	
UMULH	R	11	10011011110		4DE	
ORR	R	11	10101010000		550	
ADDS	R	11	10101011000		558	
ADDIS	I	10	1011000100		588	589
ORRI	I	10	1011001000		590	591
CBZ	CB	8	10110100		5A0	5A7
CBNZ	CB	8	10110101		5A8	5AF
STURW	D	11	10111000000		5C0	
LDURSW	D	11	10111000100		5C4	
STURS	R	11	10111100000		5E0	
LDURS	R	11	10111100010		5E2	
STXR	D	11	11001000000		640	
LDXR	D	11	11001000010		642	
EOR	R	11	11001010000		650	
SUB	R	11	11001011000		658	
SUBI	I	10	1101000100		688	689
EORI	I	10	1101001000		690	691
MOVZ	IM	9	110100101		694	697
LSR	R	11	11010011010		69A	
LSL	R	11	11010011011		69B	

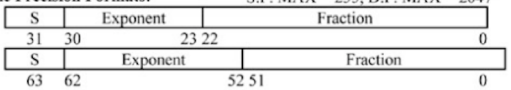
IEEE 754 FLOATING-POINT STANDARD ④

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$
where Single Precision Bias = 127,
Double Precision Bias = 1023

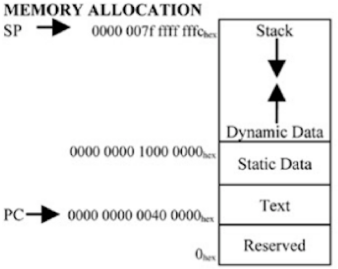
IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠ 0	± Denorm
1 to MAX - 1	anything	± F1. Pt. Num.
MAX	0	± ∞
MAX	≠ 0	NaN

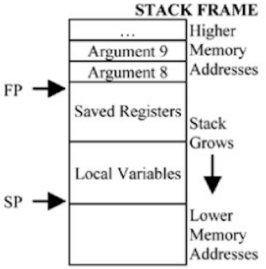
IEEE Single Precision and Double Precision Formats: S.P. MAX = 255, D.P. MAX = 2047



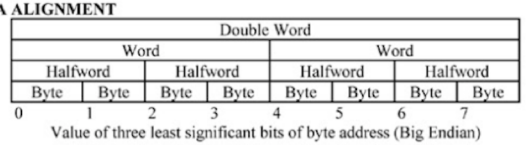
MEMORY ALLOCATION



STACK FRAME

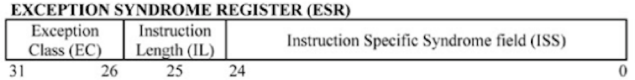


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

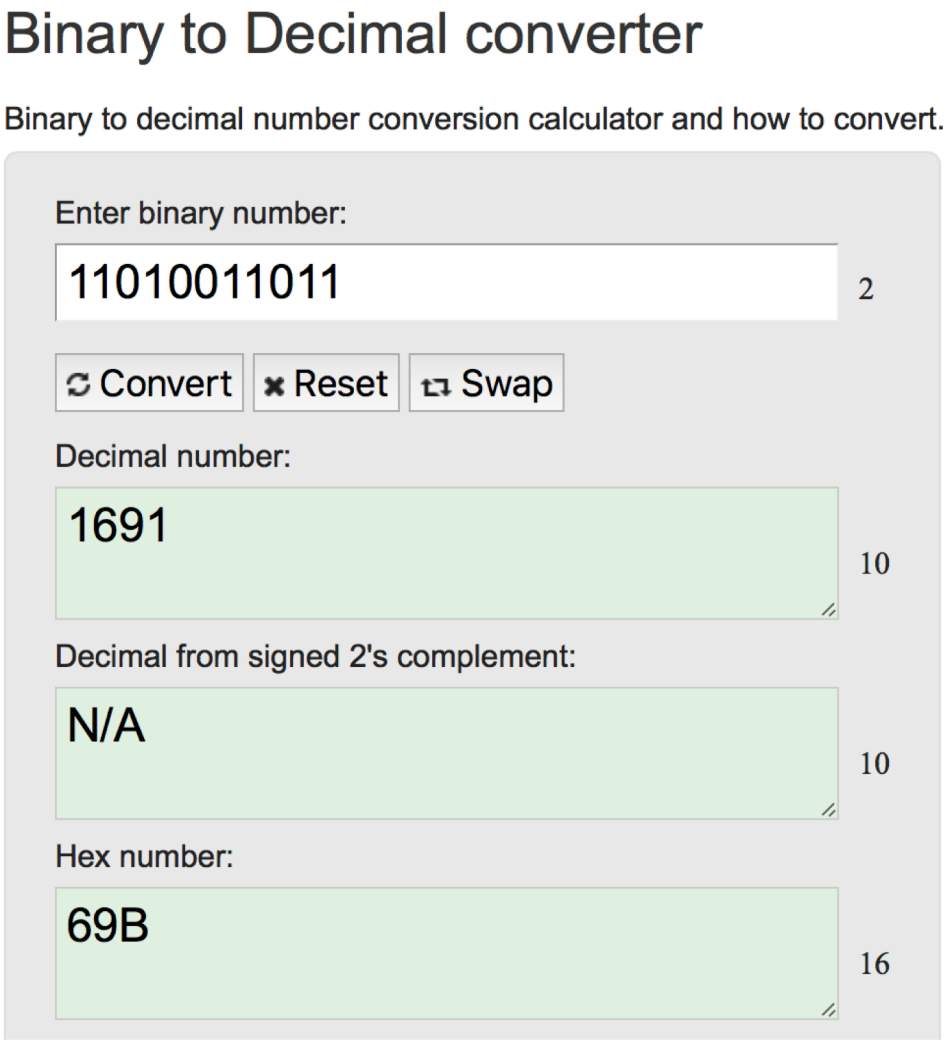
EXCEPTION SYNDROME REGISTER (ESR)



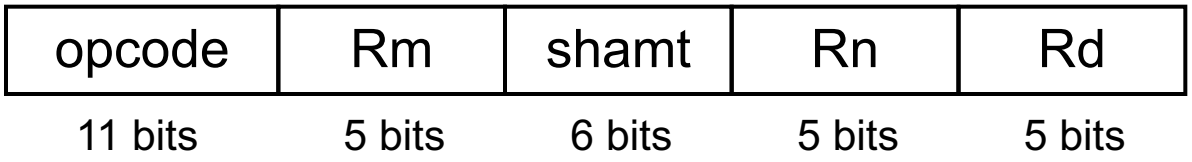
EXCEPTION CLASS

EC	Class	Cause of Exception	Number	Name	Cause of Exception
0	Unknown	Unknown	34	PC	Misaligned PC exception
7	SIMD	SIMD/FP registers disabled	36	Data	Data Abort
14	FPE	Illegal Execution State	40	FPE	Floating-point exception
17	Sys	Supervisor Call Exception	52	WPT	Data Breakpoint exception
32	Instr	Instruction Abort	56	BKPT	SW Breakpoint Exception

LEGv8 - Shift Instructions (R-Format)



LEGv8 - Shift Instructions (R-Format)

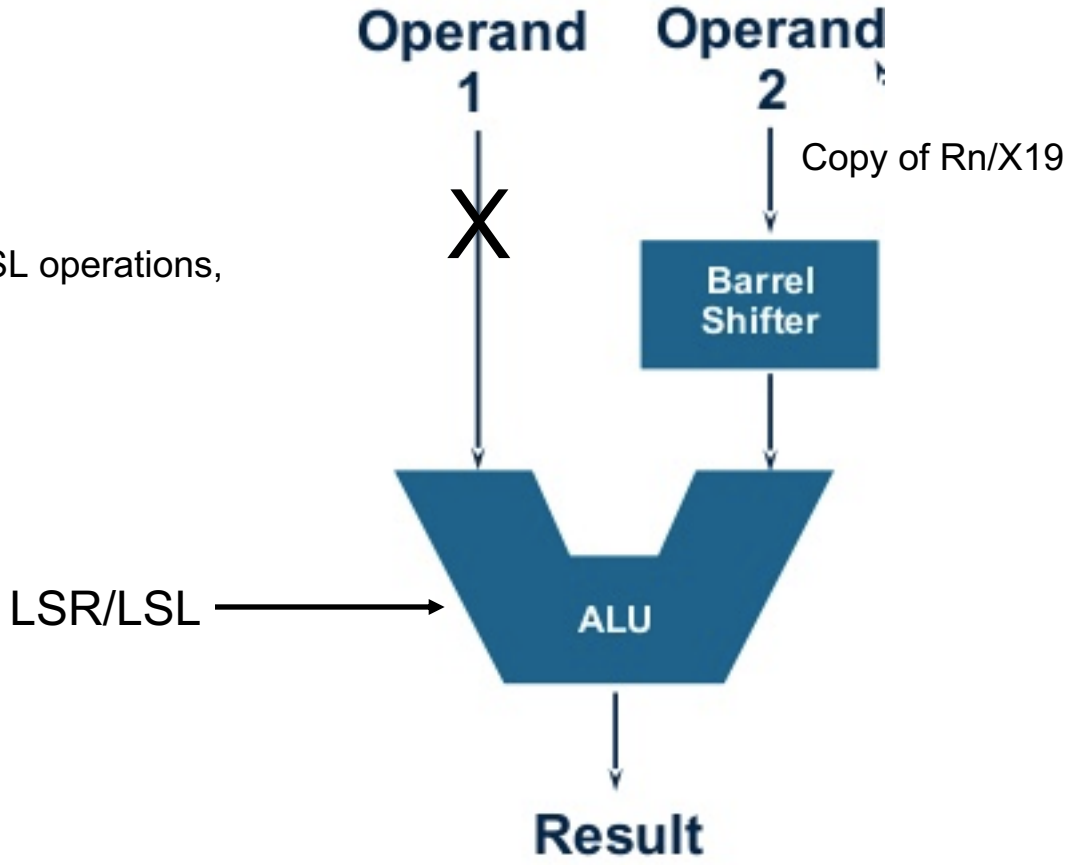


- Shift right LEGv8 operator: LSR
- LSR X11,X19, 4 *// reg X11 = reg X19 >> 4 bits*
- Shift right and fill with 0 bits
  - `LSR` by *i* bits divides by 2^i (unsigned only)
- *NOTE:* in both LSR and LSL operations, X19 is not altered

LEGv8 - Shift Instructions (R-Format)

LSR/LSL	0	4	19	11
11 bits	5 bits	6 bits	5 bits	5 bits

- LSR X11,X19, 4
- LSL X11,X19, 4
  - *NOTE:* in both LSR and LSL operations, X19 is not altered

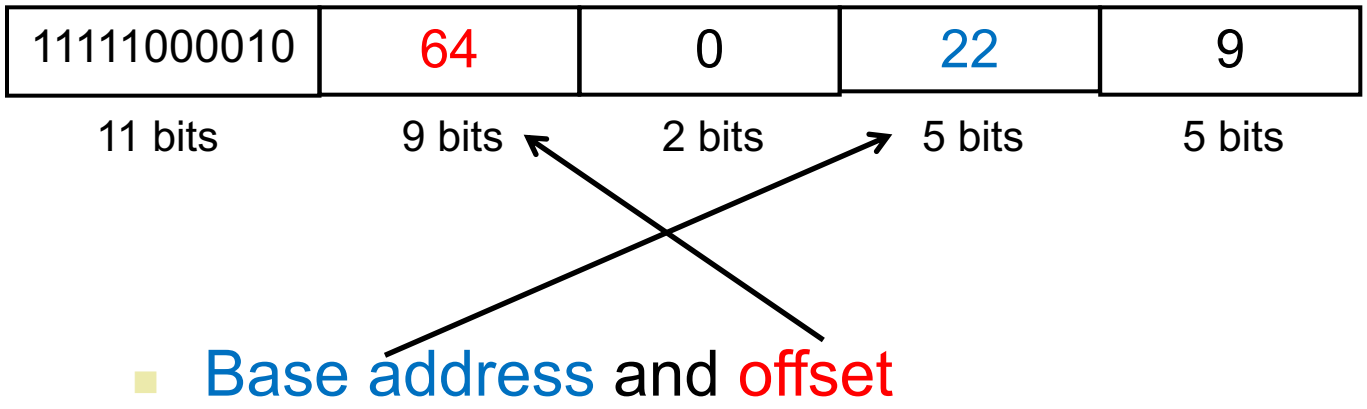


LEGv8 D-format Instructions

- Load/store instructions
  - Rn: the register number that contains the base address
  - address: constant offset from the base address in Rn (+/- 256 bytes)
  - Rt: destination (for load) or source (for store) register number
  - The 2-bit op2 is just an extension of the opcode field
    - Will be 00 in LEGv8
- ***Design Principle 3**:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

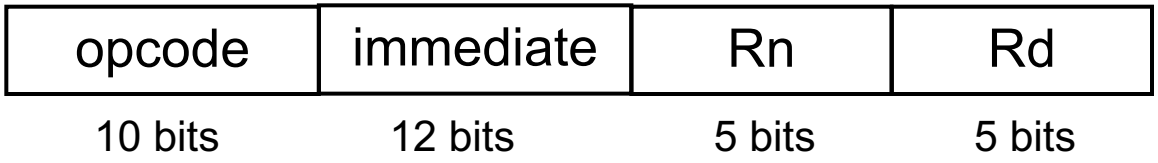
LEGv8 D-format Instructions

- LDUR X9, [X22, #64] // Temporary reg X9 gets A[8]



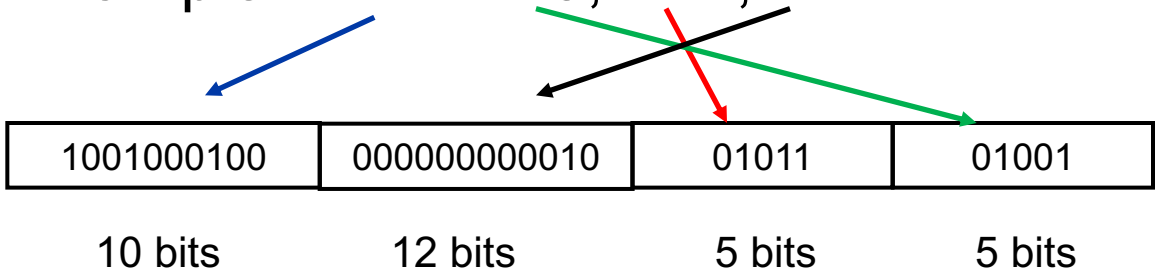
- Base address and offset

LEGv8 I-format Instructions

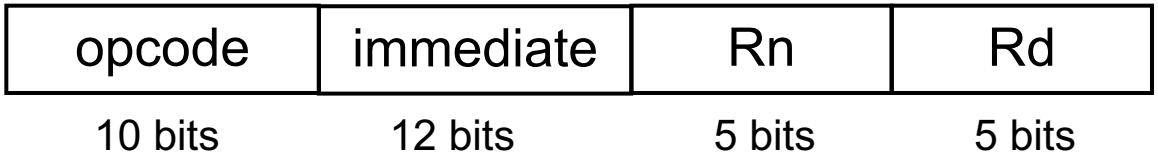


- Immediate instructions
  - Rn: source register
  - Rd: destination register

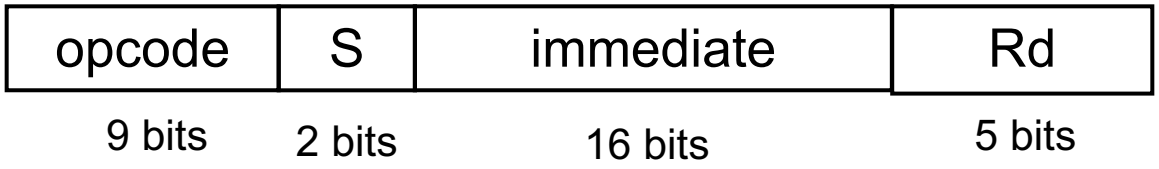
- Immediate field is zero-extended
- Example: ADDI X9, X11, 2



LEGv8 IW-format Instructions

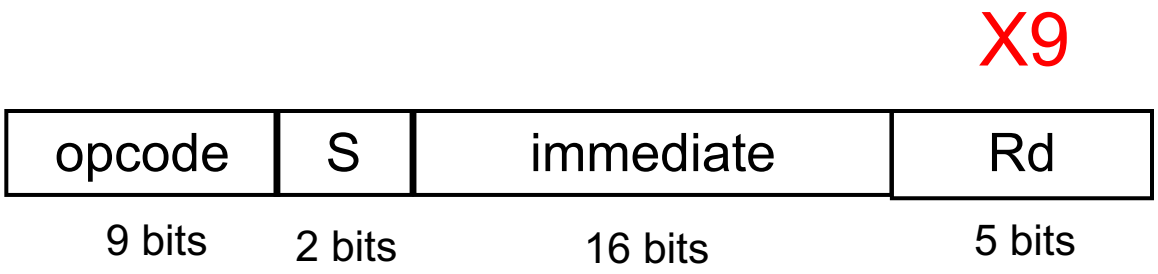


- MOVZ – Move Wide with Zero
- MOVK – Move Wide with Keep



LEGv8 IW-format Instructions

- MOVZ – Move Wide with Zero

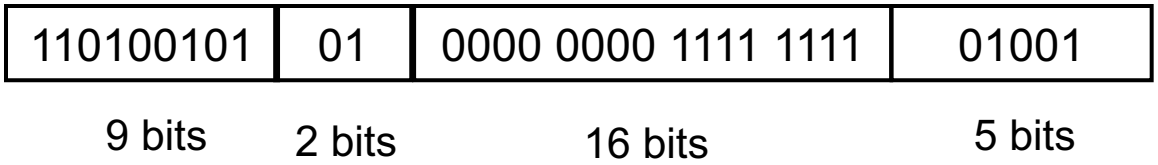


0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1100 1100

Original X9 value = 460_{10}

- MOVZ X9, 255, LSL 16

Move wide with 0's



0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 0000 0000 0000 0000

New X9 value = 255×2^{16}

LEGv8 IW-format Instructions

Original X9 value = 460_{10}

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0001 1100 1100

- MOVZ X9, 255, LSL 16

Move wide with 0's

110100101	01	0000 0000 1111 1111	01001
9 bits	2 bits	16 bits	5 bits

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 0000 0000 0000 0000

New X9 value = $255_{10} \times 2^{16}$

- MOVK X9, 255, LSL 16

111100101	01	0000 0000 1111 1111	01001
9 bits	2 bits	16 bits	5 bits

Move wide with keep

0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1111 1111 0000 0001 1100 1100

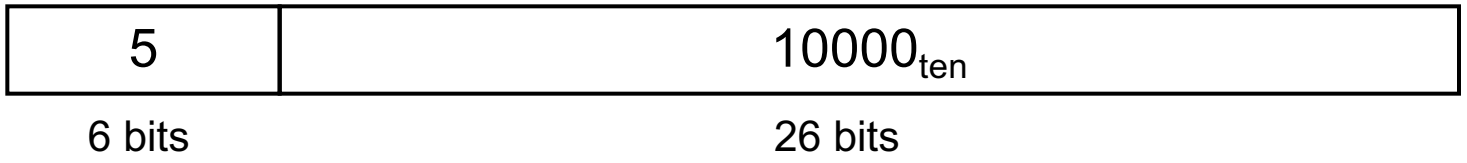
New X9 value = $255_{10} \times 2^{16} + 460_{10}$

Conditional Operations

- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `CBZ register, L1`
  - if (register == 0) branch to instruction labeled L1;
- `CBNZ register, L1`
  - if (register != 0) branch to instruction labeled L1;
- `B L1`
  - branch unconditionally to instruction labeled L1;

Branch Addressing

- B-type
  - `B 10000 // go to location 10000`$_{ten}$



- CB-type
  - `CBNZ X19, Exit // go to Exit if X19 != 0`

- Both addresses are PC-relative, and not absolute
  - Address = PC + offset (from instruction)

Branch Addressing

OPCODES IN NUMERICAL ORDER BY OPCODE ③

Instruction Mnemonic	Format	Opcode Width (bits)	Opcode Binary	Shamt Binary	11-bit Opcode Range (1) Start (Hex)	End (Hex)
B	B	6	000101		0A0	0BF
FMULS	R	11	00011110001	000010	0F1	
FDIVS	R	11	00011110001	000110	0F1	
FCMPS	R	11	00011110001	001000	0F1	
FADDS	R	11	00011110001	001010	0F1	
FSUBS	R	11	00011110001	001110	0F1	
FMULD	R	11	00011110011	000010	0F3	
FDIVD	R	11	00011110011	000110	0F3	
FCMPD	R	11	00011110011	001000	0F3	
FADDD	R	11	00011110011	001010	0F3	
FSUBD	R	11	00011110011	001110	0F3	
STURB	D	11	00111000000		1C0	
LDURB	D	11	00111000010		1C2	
B.cond	CB	8	01010100		2A0	2A7
STURH	D	11	01111000000		3C0	
LDURH	D	11	01111000010		3C2	
AND	R	11	10001010000		450	
ADD	R	11	10001011000		458	
ADDI	I	10	1001000100		488	489
ANDI	I	10	1001001000		490	491
BL	B	6	100101		4A0	4BF
SDIV	R	11	10011010110	000010	4D6	
UDIV	R	11	10011010110	000011	4D6	
MUL	R	11	10011011000	011111	4D8	
SMULH	R	11	10011011010		4DA	
UMULH	R	11	10011011110		4DE	
ORR	R	11	10101010000		550	
ADDS	R	11	10101011000		558	
ADDIS	I	10	1011000100		588	589
ORRI	I	10	1011001000		590	591
CBZ	CB	8	10110100		5A0	5A7
CBNZ	CB	8	10110101		5A8	5AF
STURW	D	11	10111000000		5C0	
LDURSW	D	11	10111000100		5C4	
STURS	R	11	10111100000		5E0	
LDURS	R	11	10111100010		5E2	
STXR	D	11	11001000000		640	
LDXR	D	11	11001000010		642	
EOR	R	11	11001010000		650	
SUB	R	11	11001011000		658	
SUBI	I	10	1101000100		688	689
EORI	I	10	1101001000		690	691
MOVZ	IM	9	110100101		694	697
LSR	R	11	11010011010		69A	
LSL	R	11	11010011011		69B	

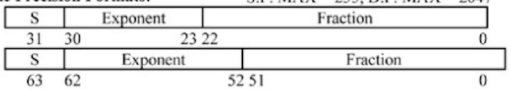
IEEE 754 FLOATING-POINT STANDARD ④

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$
where Single Precision Bias = 127,
Double Precision Bias = 1023

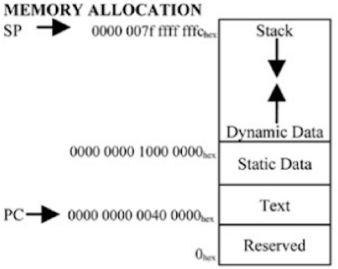
IEEE 754 Symbols

Exponent	Fraction	Object
0	0	± 0
0	≠ 0	± Denorm
1 to MAX - 1	anything	± F1. Pt. Num.
MAX	0	± ∞
MAX	≠ 0	NaN

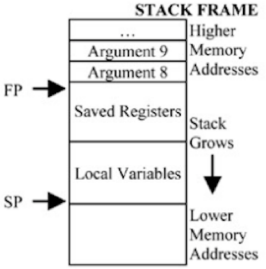
IEEE Single Precision and Double Precision Formats: S.P. MAX = 255, D.P. MAX = 2047



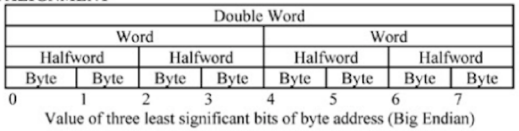
MEMORY ALLOCATION



STACK FRAME

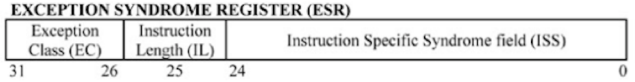


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

EXCEPTION SYNDROME REGISTER (ESR)



EXCEPTION CLASS

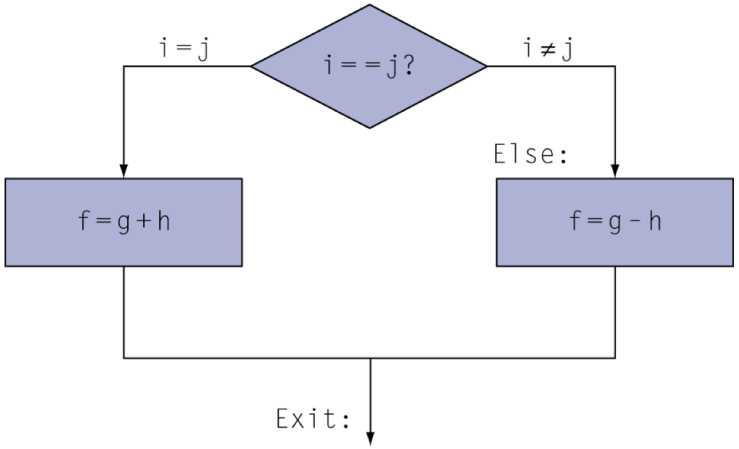
EC	Class	Cause of Exception	Number	Name	Cause of Exception
0	Unknown	Unknown	34	PC	Misaligned PC exception
7	SIMD	SIMD/FP registers disabled	36	Data	Data Abort
14	FPE	Illegal Execution State	40	FPE	Floating-point exception
17	Sys	Supervisor Call Exception	52	WPT	Data Breakpoint exception
32	Instr	Instruction Abort	56	BKPT	SW Breakpoint Exception

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - i in X20, j = X21, g in X22, h in X23



- Compiled LEGv8 code:

```
        SUB X9,X20,X21 // i-j, set Z flag
        CBNZ X9,Else
        ADD  X19,X22,X23 // g+h
        B Exit

Else:   SUB  X19,X22,x23 // g-h
Exit: …
```

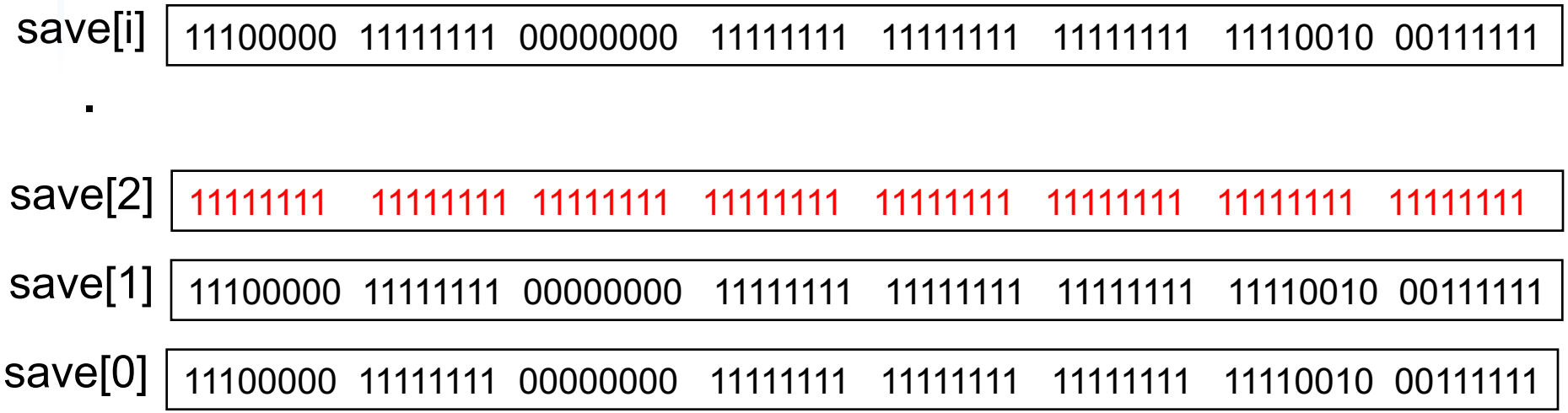
Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - The save array, an array of 8 bytes long elements
  - k = 11100000 11111111 00000000 11111111 11111111 11111111 11110010 00111111



Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in x22, 64-bit k in x24, address of save in x25
- Compiled LEGv8 code:

// get to starting address of the ith element in save keeping in mind that each element of the save array is 8 bytes long or 64 bits.

```
Loop: LSL    X10,X22,3    //Calculate the offset for ith element
      ADD    X10,X10,X25  //X10 gets (starting address of save +
                            offset) = address of ith element
      LDUR   X9,[X10,#0]  //X9 gets the content of ith element
      SUB    X11,X9,X24   //is save[i] == k?
      CBNZ   X11,Exit     //if not, we are done (anomaly found)
      ADDI   X22,X22,1    //if equal increment i(check next item)
      B      Loop        //go back to check the (i+1)th element
Exit: …
```

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

  - i in x22, 64-bit k in x24, address of save in x25
- Compiled ARMv8 code:

```
Loop: // LSL   X10,X22,3
      // ADD   X10,X10,X25
      LDR    X9,[X25,X22,LSL 3]
      SUB    X11,X9,X24 // is save[i] == k?
      CBNZ   X11,Exit   // if not, we are done, was found
      ADDI   X22,X22,1 // if equal increment i, continue
      B      Loop
Exit: …
```

Conditional Codes and Flags

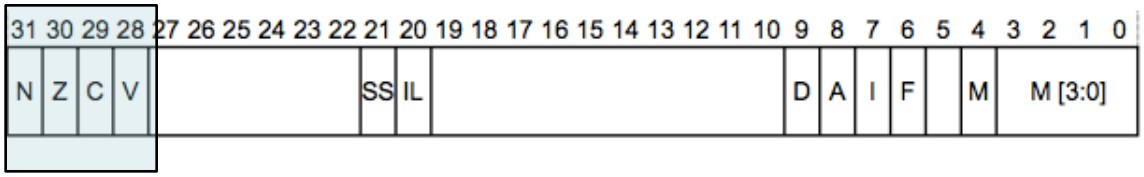


Figure 4-4 SPSR

The individual bits represent the following values for AArch64:

N	Negative result (N flag).
Z	Zero result (Z) flag.
C	Carry out (C flag).
V	Overflow (V flag).

SS	Software Step. Indicates whether software step was enabled when an exception was taken.
IL	Illegal Execution State bit. Shows the value of PSTATE.IL immediately before the exception was taken.
D	Process state Debug mask. Indicates whether debug exceptions from watchpoint, breakpoint, and software step debug events that are targeted at the Exception level the exception occurred in were masked or not.
A	SError (System Error) mask bit.
I	IRQ mask bit.
F	FIQ mask bit.
M[4]	Execution state that the exception was taken from. A value of 0 indicates

Conditional Codes and Flags

- Flags are set after the execution of the following LEGv8 instructions:
- ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS

- negative (N): result had 1 in MSB
- zero (Z): result was 0
- overflow (V): result overflowed
- carry (C): result had carryout from MSB, or borrow into MSB

Signed vs. Unsigned

- Example
  - X22 = 1111 1111 1111 1111 1111 1111 1111 1111
  - X23 = 0000 0000 0000 0000 0000 0000 0000 0001
  - Is X22 > X23?
    - `X22 < X23 # signed`
      - –1 < +1
    - `X22 > X23 # unsigned`
      - +4,294,967,295 > +1

Conditional Codes and Flags

```
SUB   X11,X9,X24
CBNZ  X11,Exit   or   CBZ X11, Done
```

- Can equally use the S-suffix to set the flags used by the condition codes such as **SUBS X11, X9, X24**
  - Negative (N=1): result had 1 in MSB
  - Zero (Z=1): result was 0
  - Both C and V flags will also be updated
- Use **subtract** to set flags, then conditionally branch:
  - **B.EQ** Done (will branch to Done label if Z = 1)
  - **B.NE.** Exit. (will branch to Exit label if Z = 0)
  - **B.LT** (less than, signed), **B.LO** (less than, unsigned)
  - **B.LE** (less than or equal, signed), **B.LS** (less than or equal, unsigned)
  - **B.GT** (greater than, signed), **B.HI** (greater than, unsigned)
  - **B.GE** (greater than or equal, signed),
  - **B.HS** (greater than or equal, unsigned)

Conditional Codes and Flags

	Signed numbers		Unsigned numbers	
Comparison	Instruction	CC Test	Instruction	CC Test
=	B.EQ	Z=1	B.EQ	Z=1
≠	B.NE	Z=0	B.NE	Z=0
<	B.LT	N!=V	B.LO	C=0
≤	B.LE	~(Z=0 & N=V)	B.LS	~(Z=0 & C=1)
>	B.GT	(Z=0 & N=V)	B.HI	(Z=0 & C=1)
≥	B.GE	N=V	B.HS	C=1

GT (s greater than), GE (s greater than or equal), LT (s less than), LE (s less than or =)
HI (u higher), HS (u higher or same), LO (u lower), LS (u lower or same).

Conditional Codes Using Flags

Condition Code			Opposite	
Code	Description		Code	Description
eq	Equal.		ne	Not equal.
hs (or cs)	Unsigned higher or same (or carry set).		lo (or cc)	Unsigned lower (or carry clear).
mi	Negative.		pl	Positive or zero.
vs	Signed overflow.		vc	No signed overflow.
hi	Unsigned higher.		ls	Unsigned lower or same.
ge	Signed greater than or equal.		lt	Signed less than.
gt	Signed greater than.		le	Signed less than or equal.
al (or omitted)	Always executed.		*There is no opposite to al.*	

Conditional and Unconditional Summary

`B.cond label`

Branch: conditionally jumps to program-relative label if cond is true.

`CBNZ Xn, label`

Compare and Branch Not Zero (extended): conditionally jumps to `label` if `Xn` is not equal to zero.

`CBZ Xn, label`

Compare and Branch Zero (extended): conditionally jumps to `label` if `Xn` is equal to zero.

`B label`

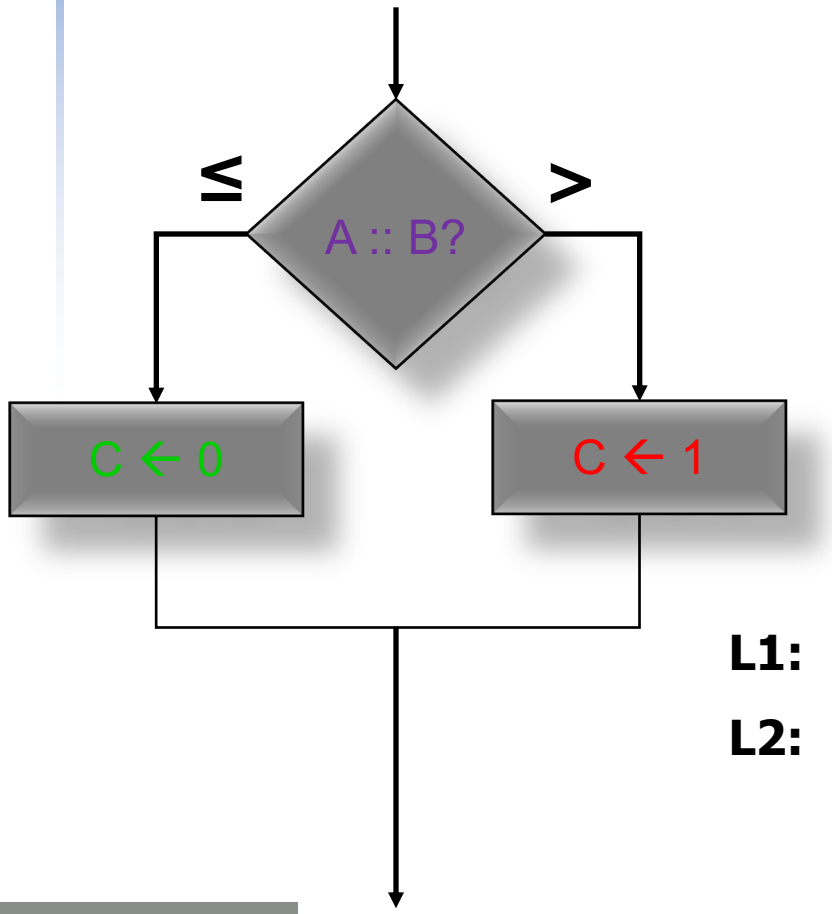
Branch: unconditionally jumps to pc-relative `label`.

`BL label`

Branch and Link: unconditionally jumps to pc-relative `label`, writing the address of the next sequential instruction to register `X30`.

Conditional Code Example

if-then-else statement




```
        LDUR    X0,A
        LDUR    X1,B
        SUBS    X2,X0,X1 // X0-X1 & set flags
        //CMP   X0, X1 //(A-B) is A LE B
        B.LE    L1  //if Z=1 or N !=V
        LDUR    X0,=1   ( MOV X0, 1 )
        B       L2
L1:     LDUR    X0,=0   ( MOV X0, 0 )
L2:     STUR    X0,C
        …
```

Opposite Conditional Code Example

- if (a > b) a += 1;
  - Assuming a is already loaded in X22, and b in X23

```
LDUR X22, a
LDUR X23, b
SUBS X9,X22,X23  // (a-b) use subtract to make comparison

  B.LE Exit      // if a <= b (if opposite condition true) then
                 //           exit (conditional branch)

  ADDI X22, X22,1

Exit:
```

Function Call and Return

Function Call: "BL function"

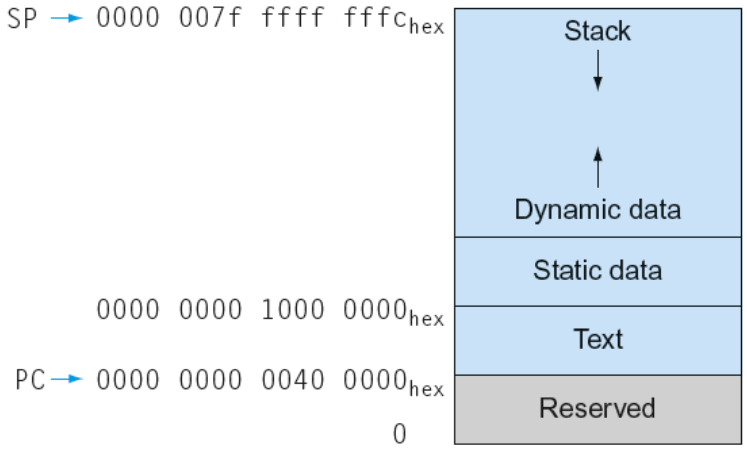
- Loads program counter (pc) with entry point address of function.
- Saves return address in the link register (LR aka X30).

Function Return: "BR LR"

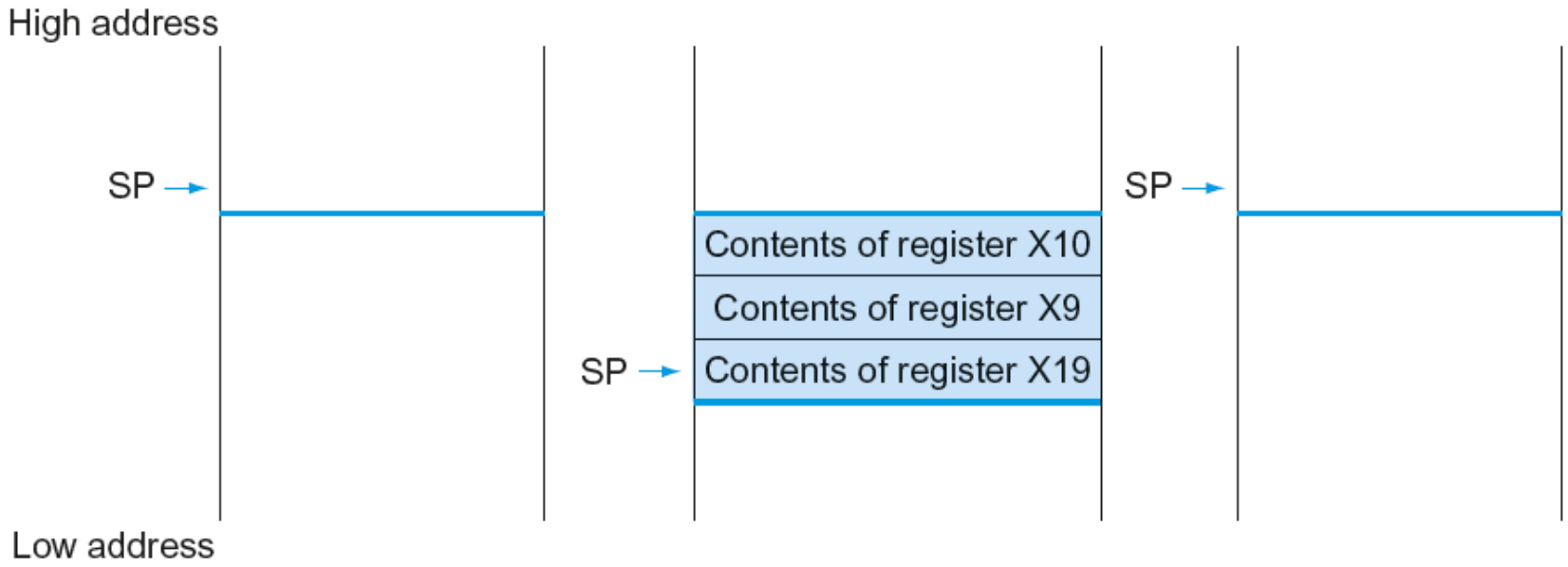
- Copies link register (X30 aka LR) back into program counter.

Memory Layout (recall)

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Storage on the Stack (recall)



Push {X10, X9, X19}

Pop {X19, X9, X10}

LIFO

X19, X9, X10 and SP are restored with their original values

Register Usage (Recall)

- X0 – X7: procedure arguments/results
- X9 to X15: temporary registers
  - Not preserved by the called function
- X19 to X27: saved registers
  - If used, the called function saves and restores them
- X30 aka LR contains the return address in the calling function.

Function Call and Return

- Steps required
  1. Place parameters in registers X0 to X7
  2. Transfer control to function (BL func)
  3. Acquire storage for function (on stack)
  4. Perform function's operations
  5. Place result in register for caller (X0…X7)
  6. Return to place of call (address in X30 aka LR)

Function Call and Return

```
void enable(void) ;          //defining enable()
●●●
enable() ;                   //calling enable()
●●●
```

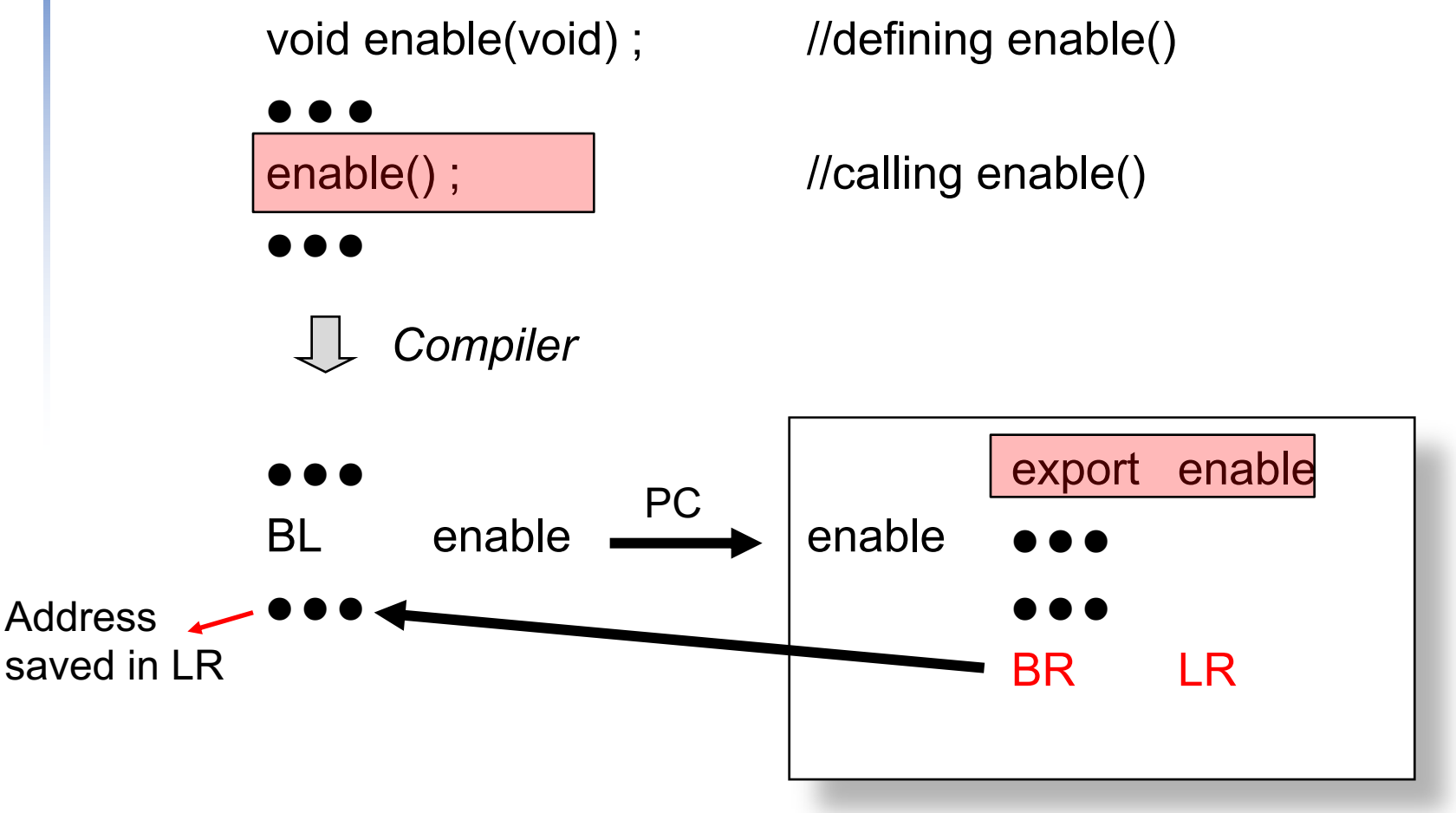
Function Call and Return

```
void enable(void) ;          //defining enable()
● ● ●
enable() ;                   //calling enable()
● ● ●
```

Compiler

```
● ● ●
BL        enable
● ● ●
```

Function Call and Return



Leaf Procedure Example

- C code:

```
long long int leaf_example (long long int
g, long long int h, long long int i, long
long int j)
{
    long long int f;
    f = (g + h) - (i + j);
    return f;
}
```

X0 X1 X2 X3

  - Arguments g, …, j in X0, …, X3

Leaf Procedure Example

- LEGv8 code:

```
{long long int f;
 f = (g + h) - (i + j);
 return f;}
```

```
leaf_example:
    ADD X9,X0,X1        // X9 = g + h
    ADD X10,X2,X3       // X10 = i + j
    SUB X11,X9,X10      // f = X9 – X10
    ADD X0,X11,XZR      // copy f to X0 to return it
    BR  LR              // Return to caller
```

Leaf Procedure Example

- LEGv8 code:

```
{long long int f;
 f = (g + h) - (i + j);
 return f;}
```

```
leaf_example:
    SUBI SP,SP,#24      // Make room to save X20, X19, X21 on stack
    STUR X20,[SP,#16]   // Save X20 on the stack, SP [#16-#23]
    STUR X19,[SP,#8]    // Save X19 on the stack, SP [#8-#15]
    STUR X21,[SP,#0]    // Save X21 on the stack, SP [#0-#7]
    ADD X19,X0,X1       // X19 = g + h
    ADD X20,X2,X3       // X20 = i + j
    SUB X21,X19,X20     // f = X19 – X20
    ADD X0,X21,XZR      // copy f to X0 to return it
    LDUR X20,[SP,#16]   // Restore X20 from stack
    LDUR X19,[SP,#8]    // Restore X19 from stack
    LDUR X21,[SP,#0]    // Restore X21 from stack
    ADDI SP,SP,#24      // Update the stack pointer
    BR LR               // Return to caller
```

Leaf Procedure Example

- In earlier versions of ARM:

```
leaf_example:
    PUSH {X19, X20, X21}   // Save X19, X20, and X21 on the stack
    ADD X19,X0,X1          // X19 = g + h
    ADD X20,X2,X3          // X20 = i + j
    SUB X21,X19,X20        // f = X19 – X20
    ADD X0,X21,XZR         // copy f to X0 to return it
    POP {X19, X20, X21}    // Restore X10, X9, X19 from stack
    BR LR                  // Return to caller
```

C Sort Example

- Illustrates use of assembly instructions for a C sort function
- Swap procedure (leaf)

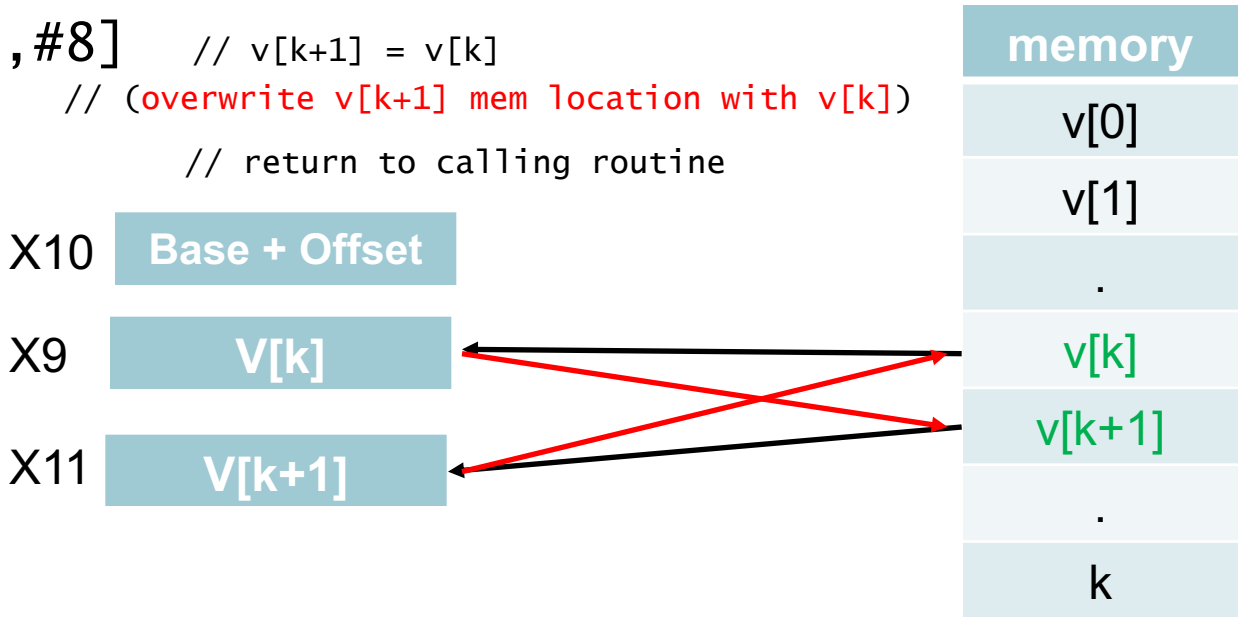
```
void swap(long long int v[], long long
int k)
{
  long long int temp;
  temp = v[k];//save the k_th element
  v[k] = v[k+1];//copy (k+1)_th to k_th
  v[k+1] = temp;//copy temp to (k+1)_th
}
```

  - Starting address of v in X0, and k in X1

The Procedure Swap

***Assumption: Starting address of the array of double words v in X0, and k in X1

```
swap:
    LSL X10,X1,#3       // X10 = k * 8 (find byte offset of the v[k])
    ADD X10,X0,X10      // X10 = address of v[k] = address of v[0] + offset of v[k]
    LDUR X9,[X10,#0]    // X9 = v[k] (save content of v[k] into X9 temp register)
    LDUR X11,[X10,#8]   // X11 = v[k+1] (load content of v[k+1] into X11)
    STUR X11,[X10,#0]   // v[k] = v[k+1] (overwrite v[k] mem location with v[k+1])
    STUR X9,[X10,#8]    // v[k+1] = v[k]
                        // (overwrite v[k+1] mem location with v[k])
    BR LR                       // return to calling routine
```



Passing Parameters to a Function

```
void display(uint8_t  *p, int32_t) ;
●●●
display(ptrToBuffer, 5) ;
●●●
```

Compiler

```
●●●
```

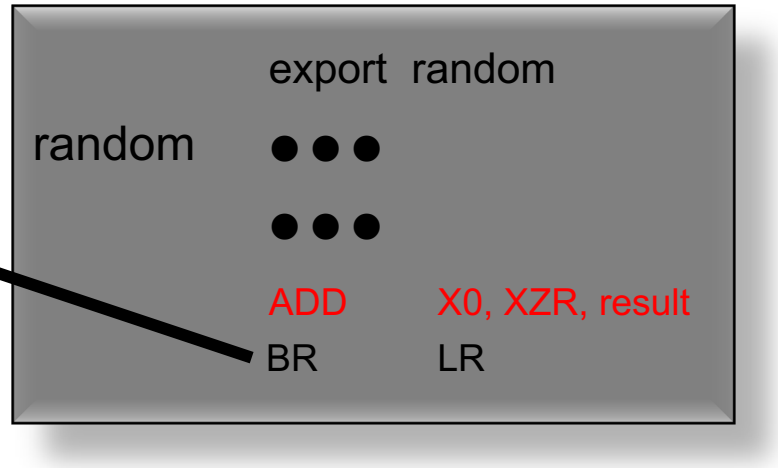
Registers X0-X7 used for first 8 parameters; any more have to be pushed on stack

```
●●●
```

Returning a Value from Functions

```
int32_t  random(void) ;
●●●
numb = random() ;
●●●
```

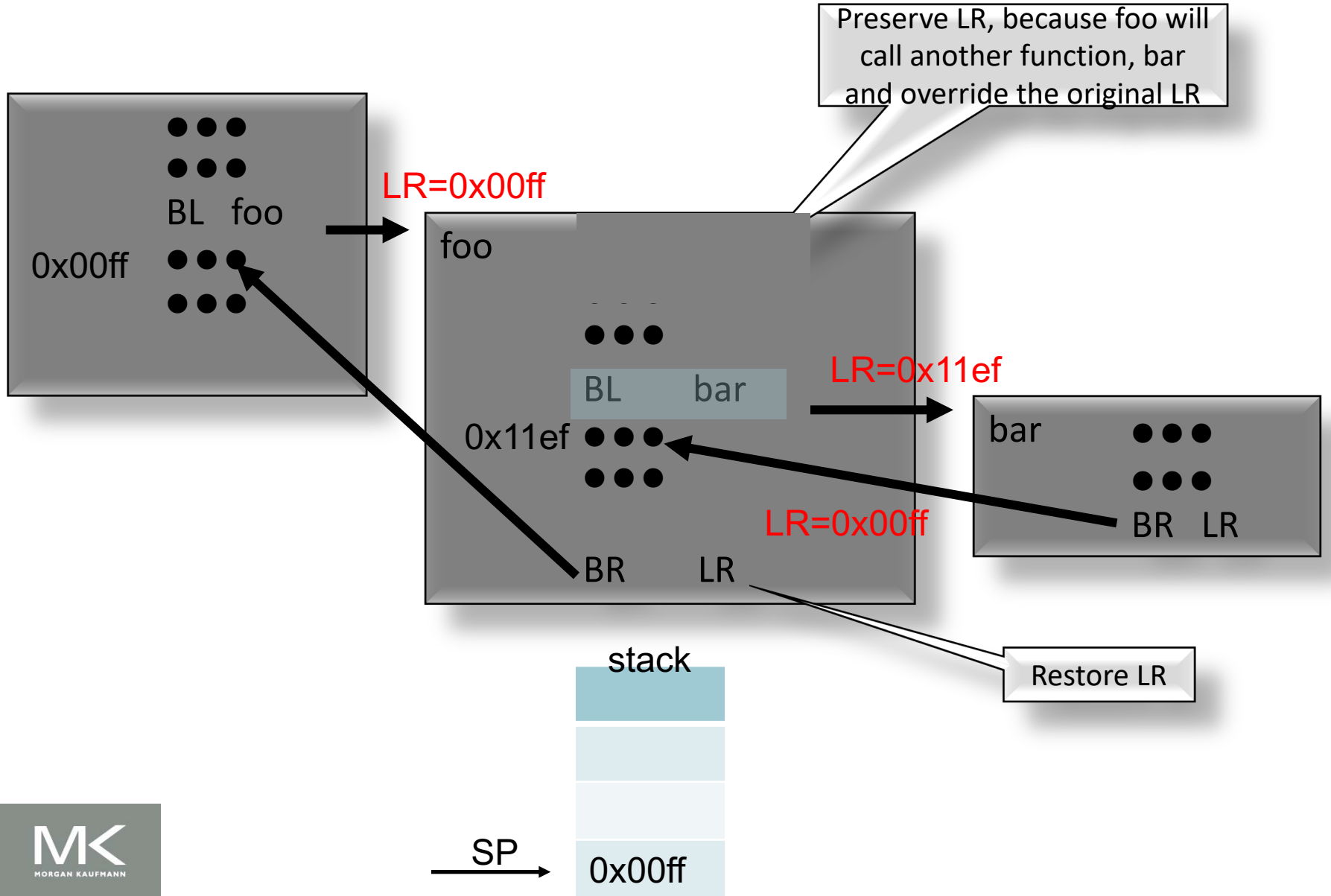

```
●●●
BL      random
STUR    X0, numb
●●●
```



Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example



Non-Leaf Procedure Example

```
Main()
{
.
  y = f1(2);
.
}
```

Two functions with parameters and return values, one calling the other.

C Version	ARMv8 Assembly Version

`int32_t f1(int32_t i)`
`{`
`return f2(4) + i ;`
`}`
 	
***assume i in X0
	`f1:`
	`PUSH {X4,LR} // Preserve X4 & LR`
	`ADD X4, XZR, X0 //Keep i safe in X4`
	`ADDI X0,XZR,#4 //X0 <-- f2's arg`
	`BL f2 // X0 <-- f2(4)`
	`ADD X0,X0,X4 // X0 <-- f2(4) + i`
	`POP {X4,LR} // Restore X4 & LR`
	`BR LR`

1999 ISO (C99) New Data Types

#include <stdint.h>

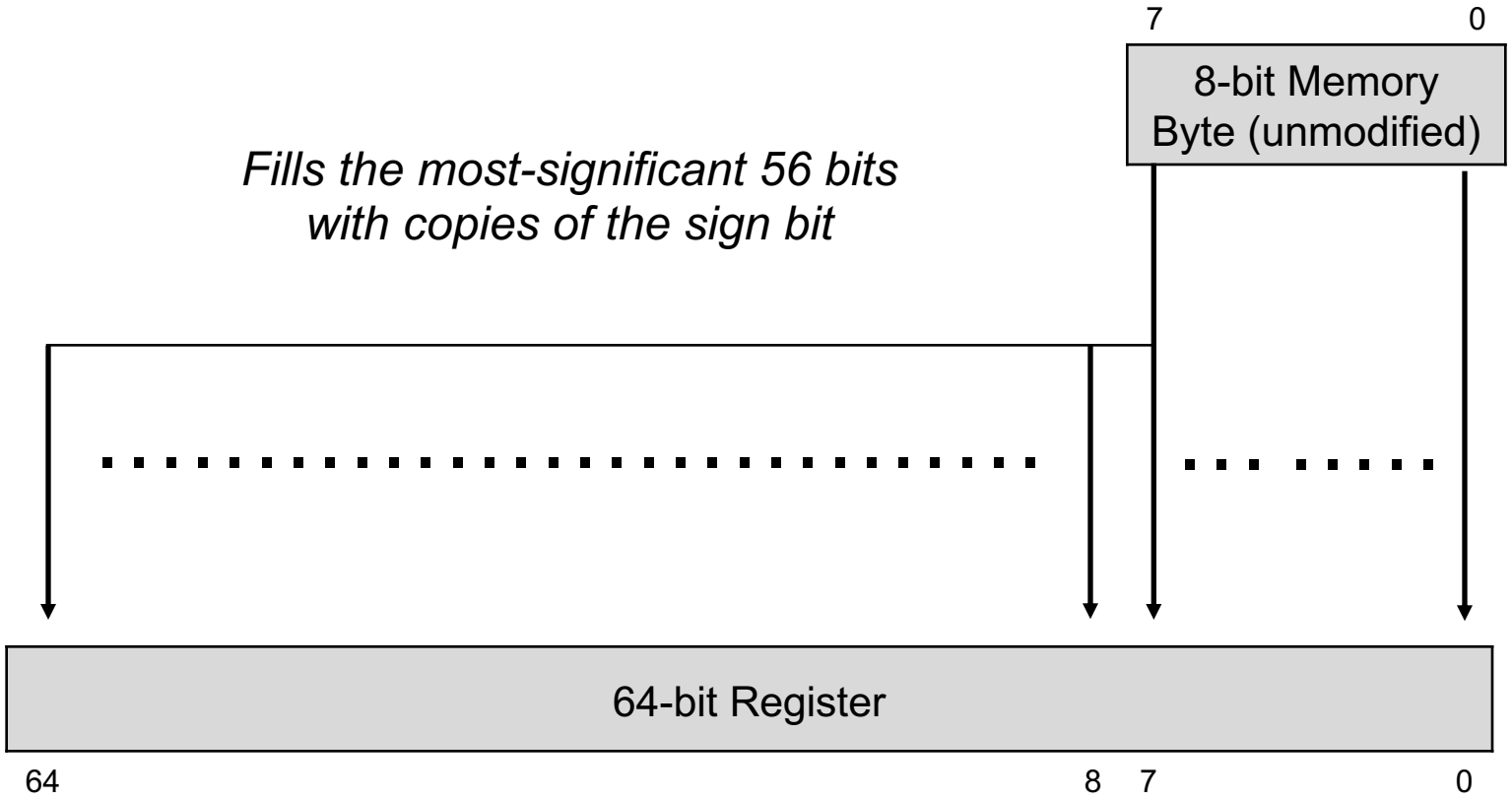
	Signed	Unsigned
8-bits	int8_t	uint8_t
16-bits	int16_t	uint16_t
32-bits	int32_t	uint32_t
64-bits	int64_t	uint64_t

DATA TYPE	SIZE (IN BYTES)	RANGE
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	

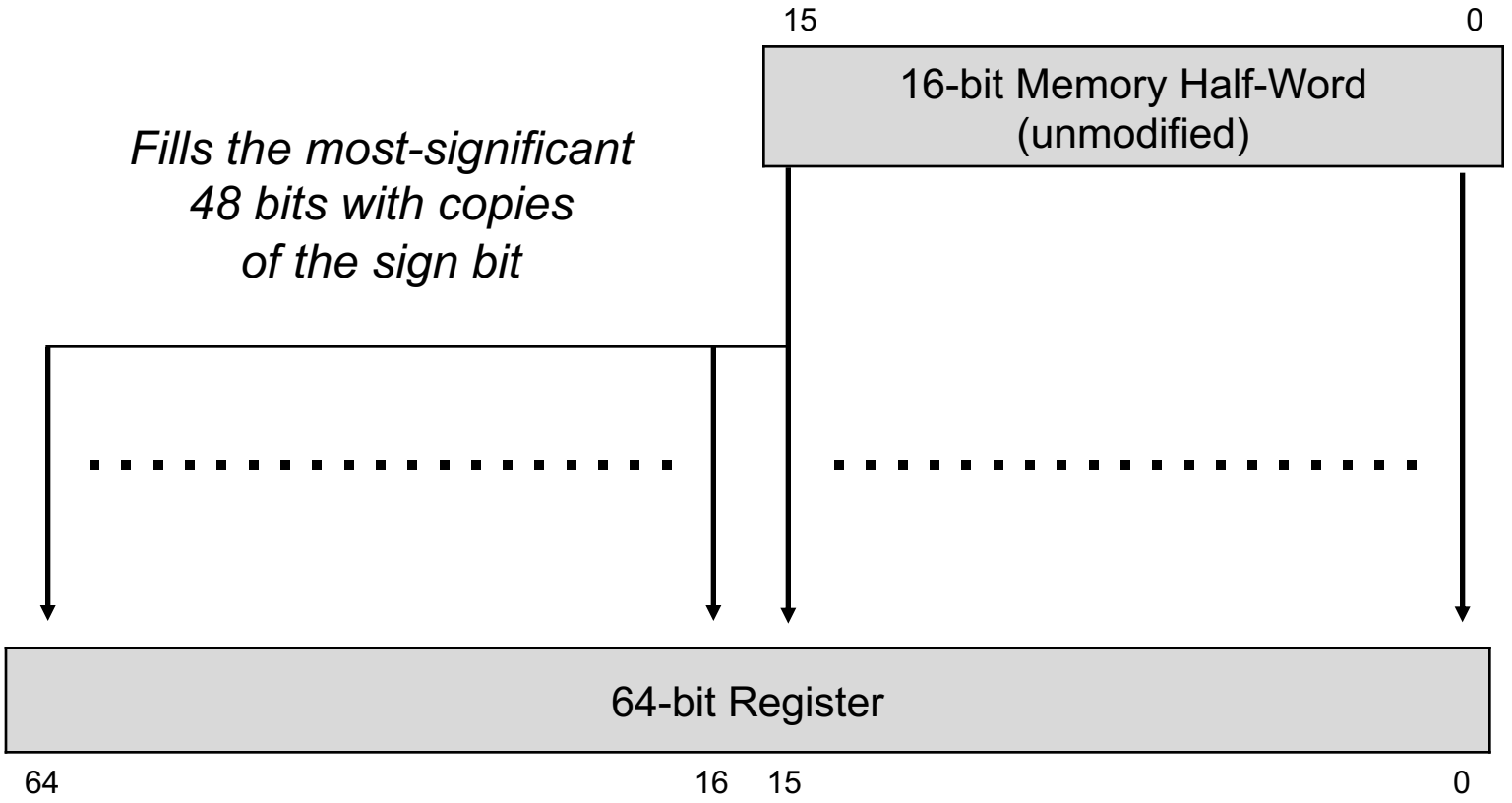
Byte/Halfword/Word Operations

- LEGv8 byte/halfword/word load/store
  - Load/Store byte:
    - `LDURB Rt, [Rn, offset]` //Zero extend to 64 bits in rt
    - `LDURSB Rt, [Rn, offset]` //Sign extend to 64 bits in rt
    - `STURB Rt, [Rn, offset]` //Store just rightmost byte, same size as dest.
  - Load/Store halfword:
    - `LDURH Rt, [Rn, offset]` //Zero extend to 64 bits in rt
    - `LDURSH Rt, [Rn, offset]` //Sign extend to 64 bits in rt
    - `STURH Rt, [Rn, offset]` //Store just rightmost h-word, same size as dest
  - Load/Store signed word:
    - `LDURSW Rt, [Rn, offset]` //Sign extend to 64 bits in rt
    - `STURW Rt, [Rn, offset]` //Store just rightmost word, same size as dest.
  - Where both Rt and Rn are 64-bit X registers

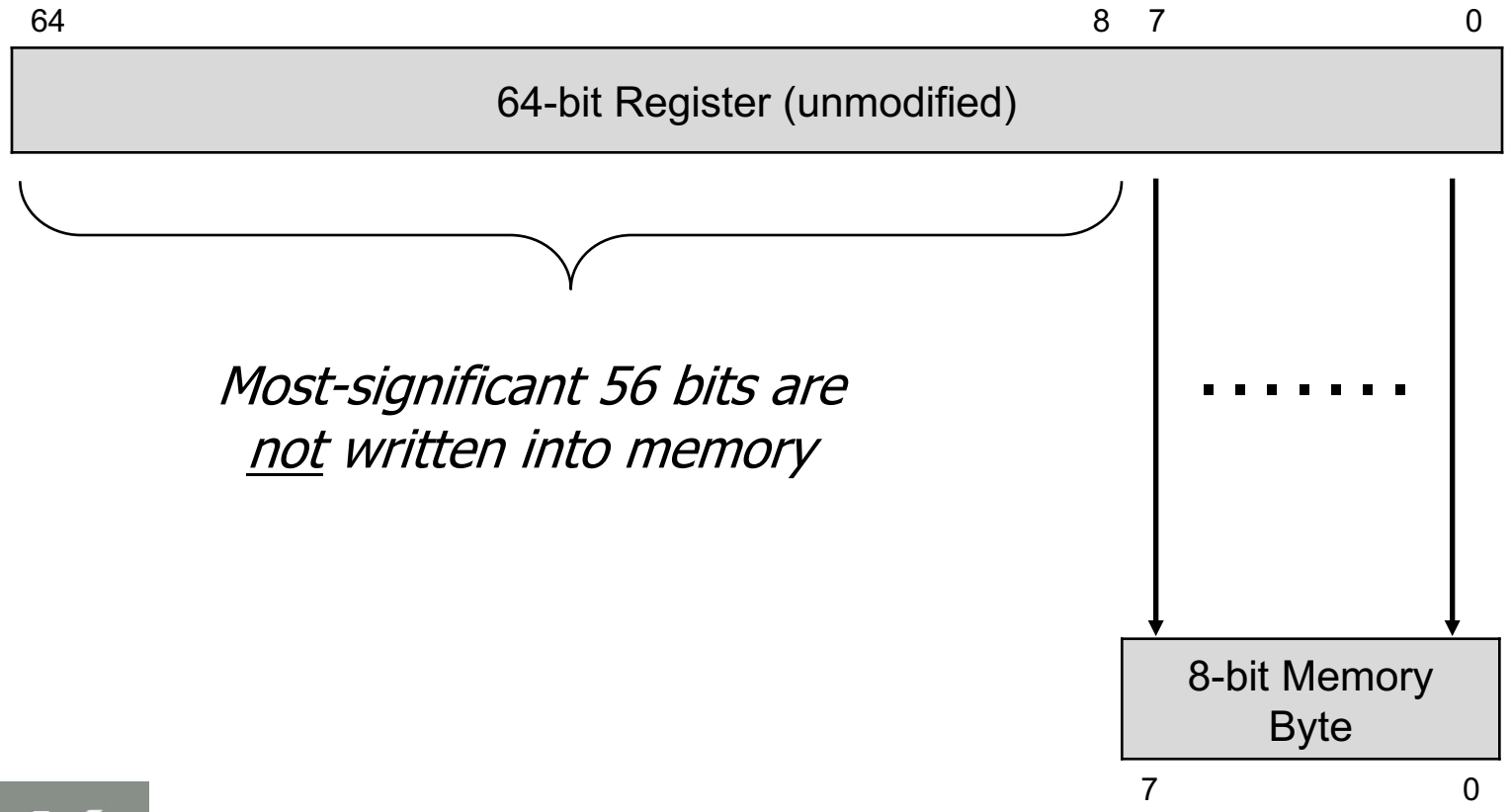
LDURSB: Load Register with Signed Byte



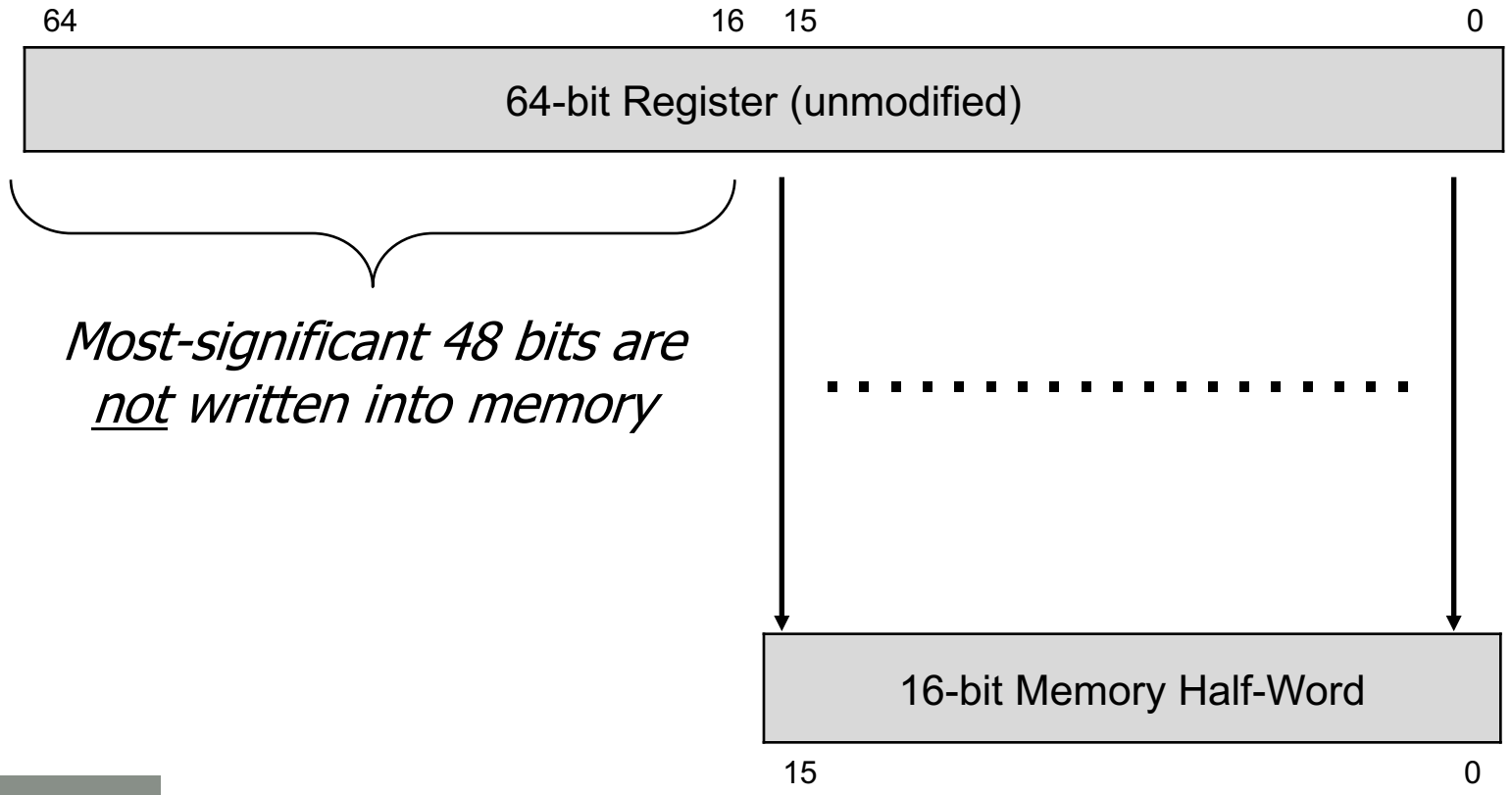
LDURSH: Load Register with Signed Half-Word



STURB: Store Register to Byte

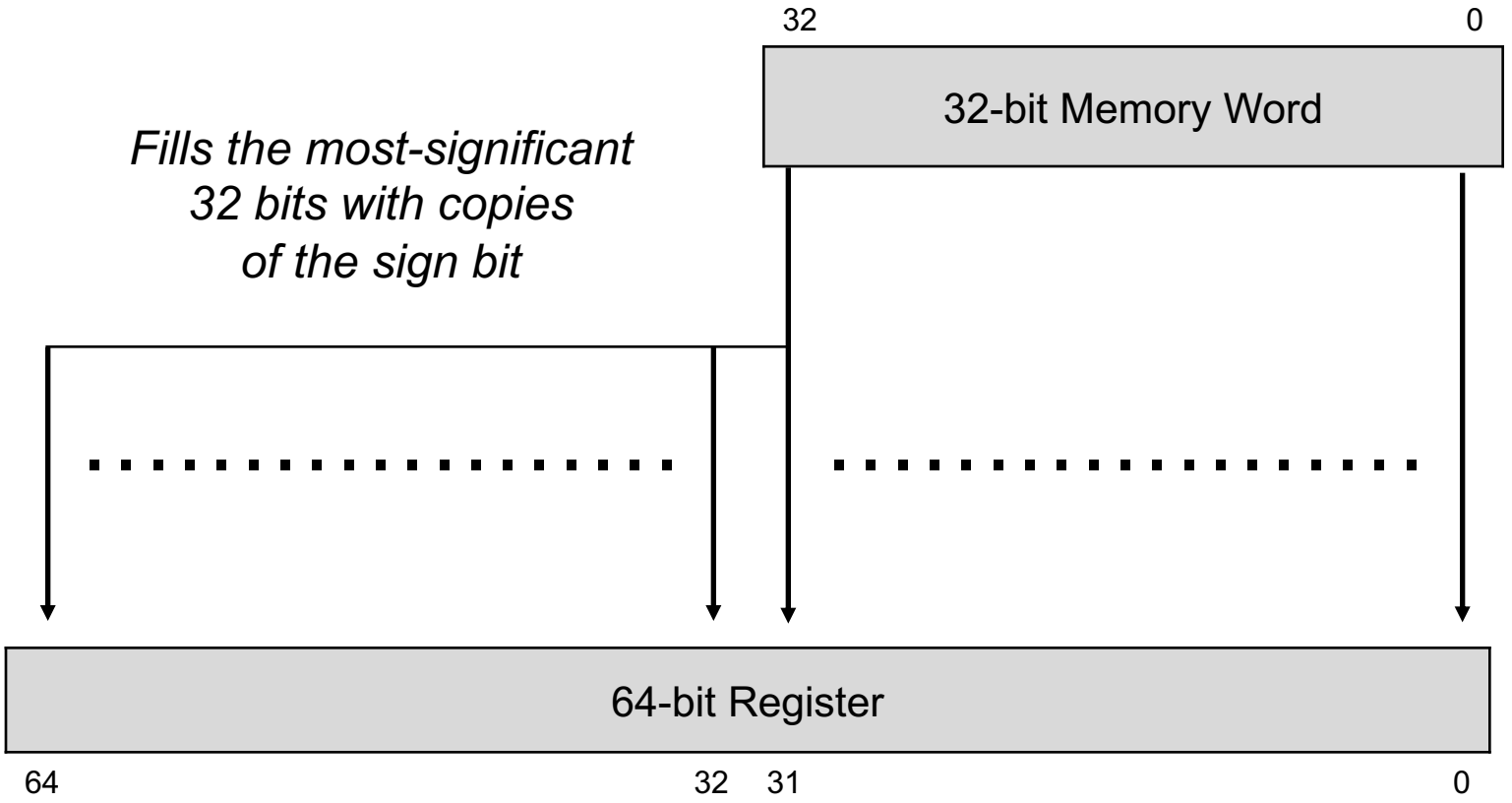


STURH: Store Register to Half-Word



LDURSW: Load Register with Signed Word

LDURW W0, a

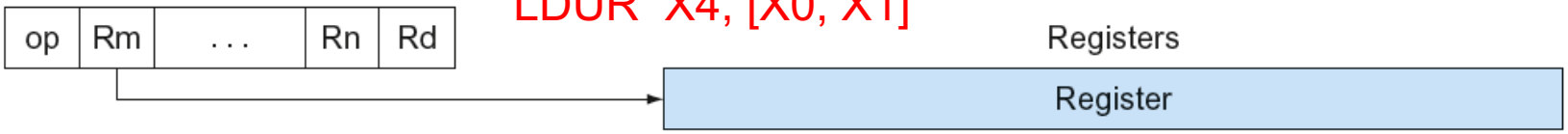


LEGv8 Addressing Summary

1. Immediate addressing

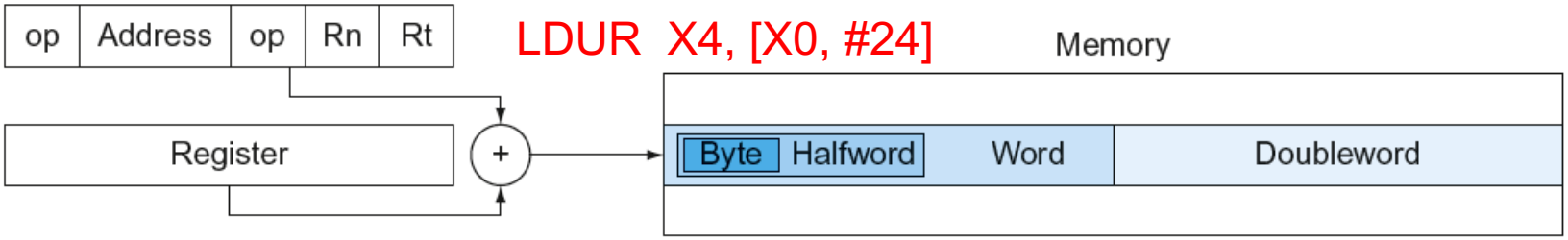
ADDI X0, X1, #5

2. Register addressing



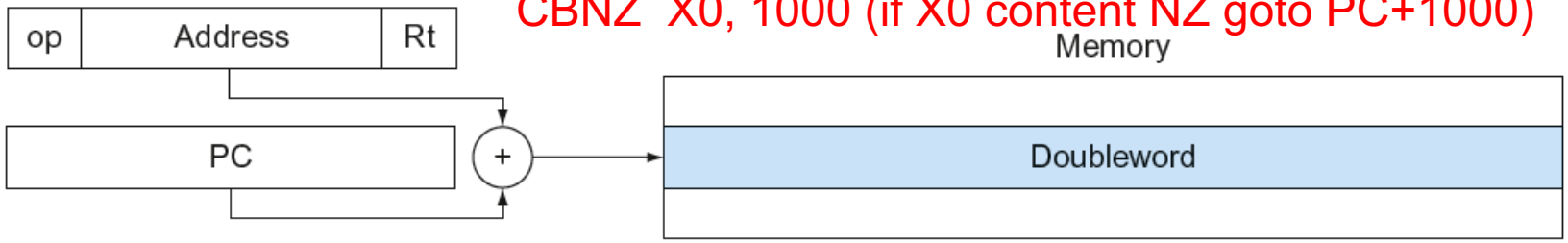
LDUR X4, [X0, X1]

3. Base addressing



LDUR X4, [X0, #24]

4. PC-relative addressing



CBNZ X0, 1000 (if X0 content NZ goto PC+1000)

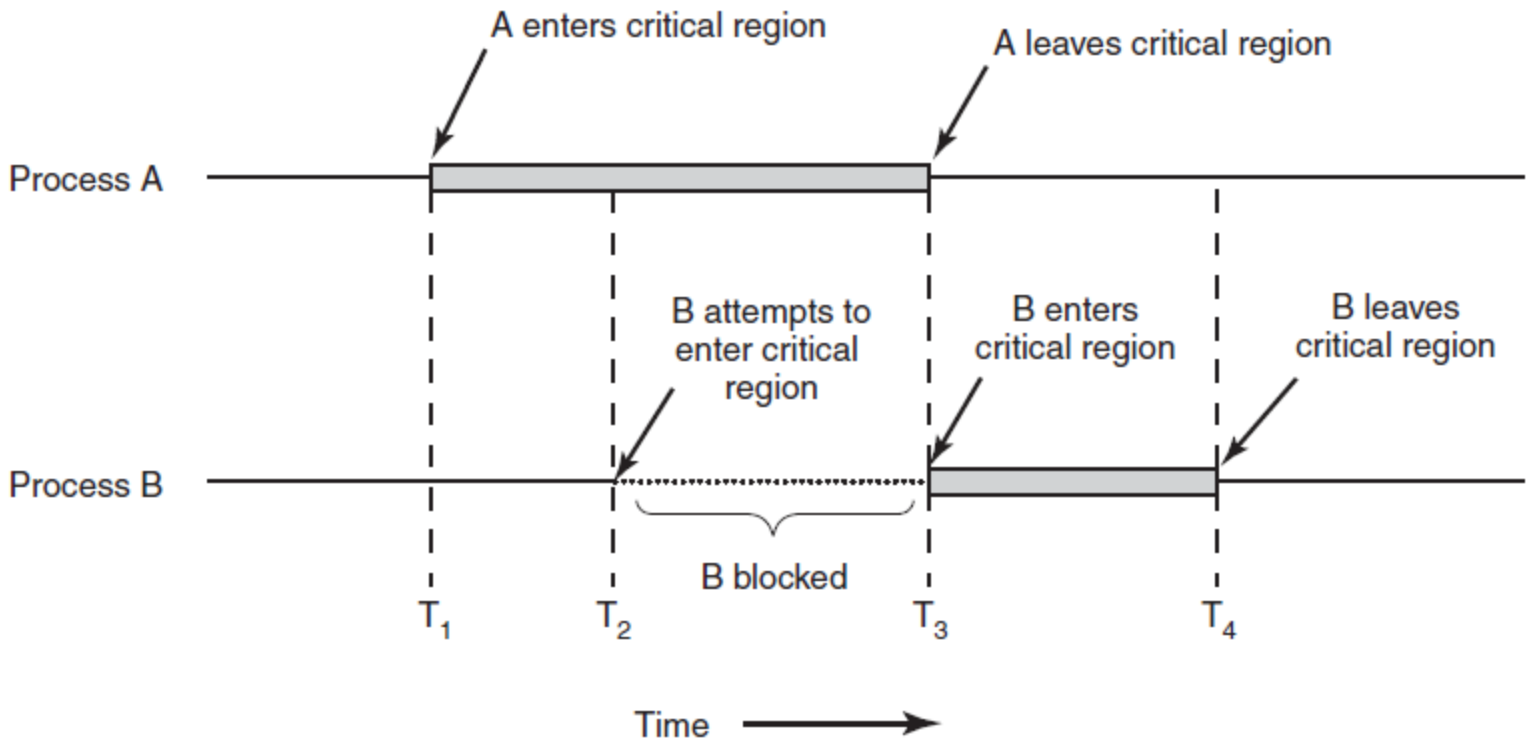
Inter-Process Communication (IPC)

- Race Conditions
- Critical Regions
- Mutual Exclusion with Busy Waiting
- Sleep and Wakeup
- The Producer-Consumer Problem
- Semaphores
- Mutexes
- Monitors

Critical Regions (recall)

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

Critical Regions (recall)



Synchronization (recall)

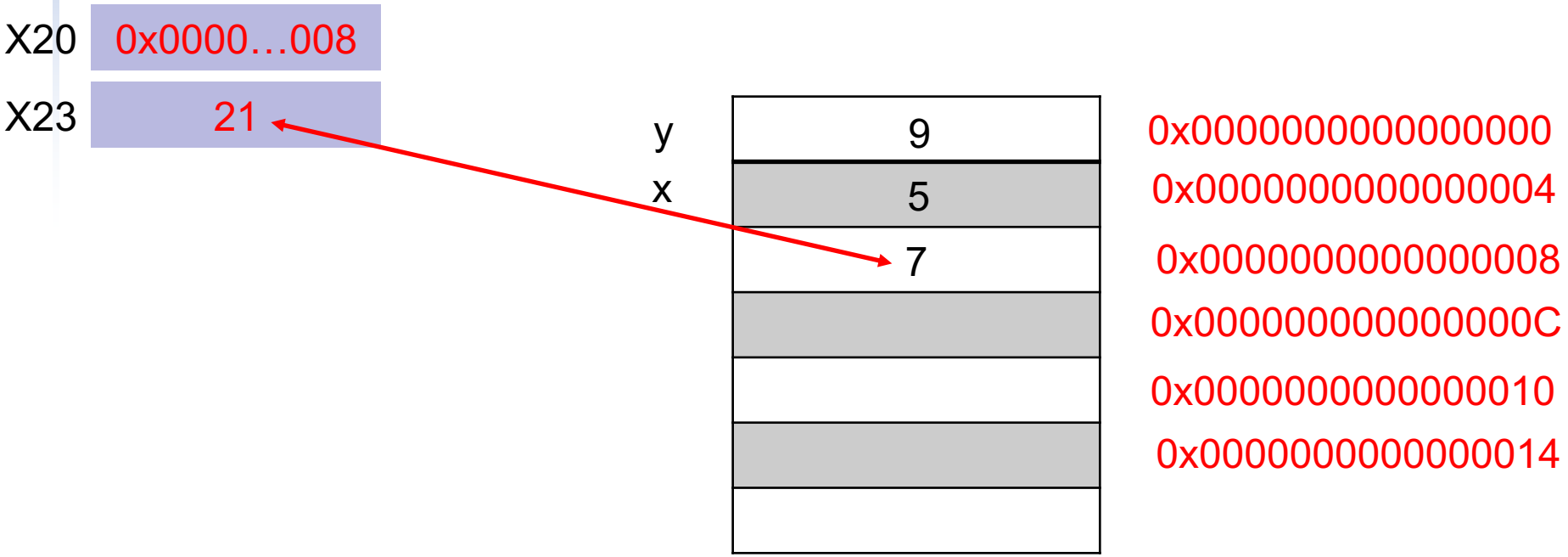
- Two processors (or processes, threads, etc. for that matter) sharing an area of memory
  - P1 writes, then P2 reads
  - Data race if P1 and P2 don't synchronize
    - Result depends of order of accesses
- Hardware support required
  - Atomic read/write memory operation
  - No other access (processor, process, or thread) to the location allowed between the read and write
- An atomic pair of instructions
  - E.g., atomic swap of register ↔ memory

Synchronization in LEGv8

- Load exclusive register: LDXR
- Store exclusive register: STXR
- To use:
  - Execute LDXR then STXR with same address
  - If there is an intervening change to the content of address, store fails
    - STXR fails with a return value of 1 stored in a register specified by the STXR instruction
  - Only register instructions allowed (not memory/data transfer instructions) on the loaded register in between LDXR and STXR.
    - ADD, SUB, LSL, etc.

Synchronization in LEGv8

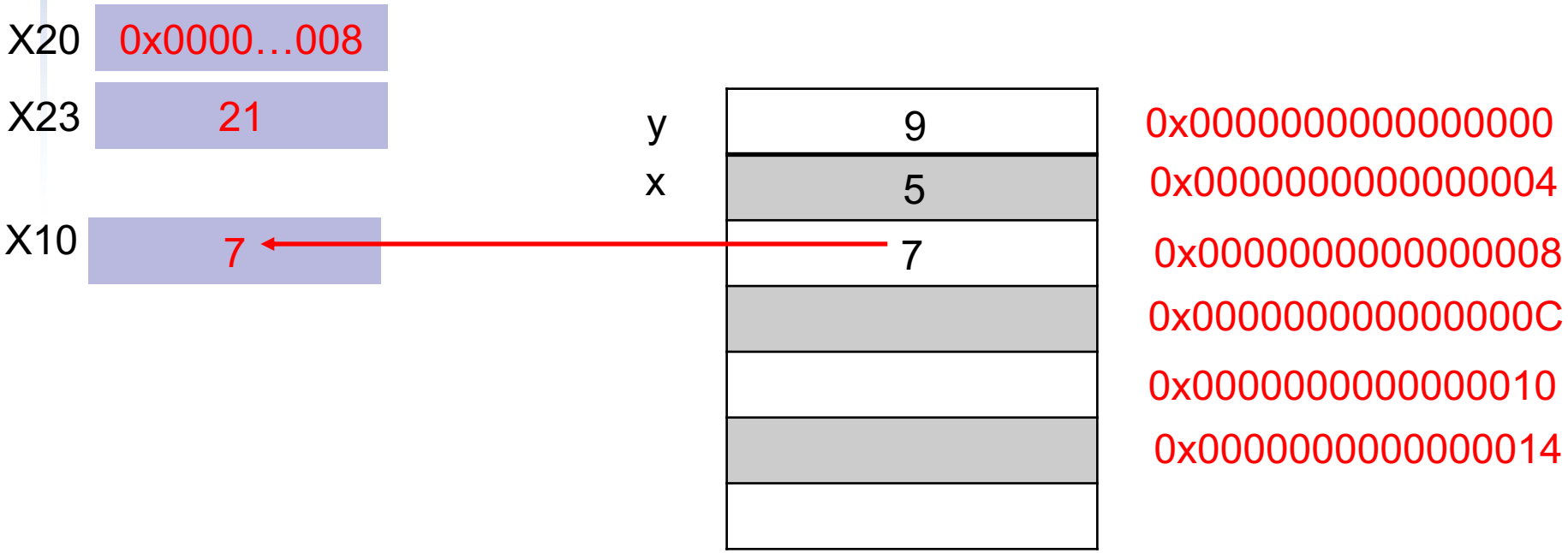
- Example 1: atomic swap



Synchronization in LEGv8

- Example 1: atomic swap

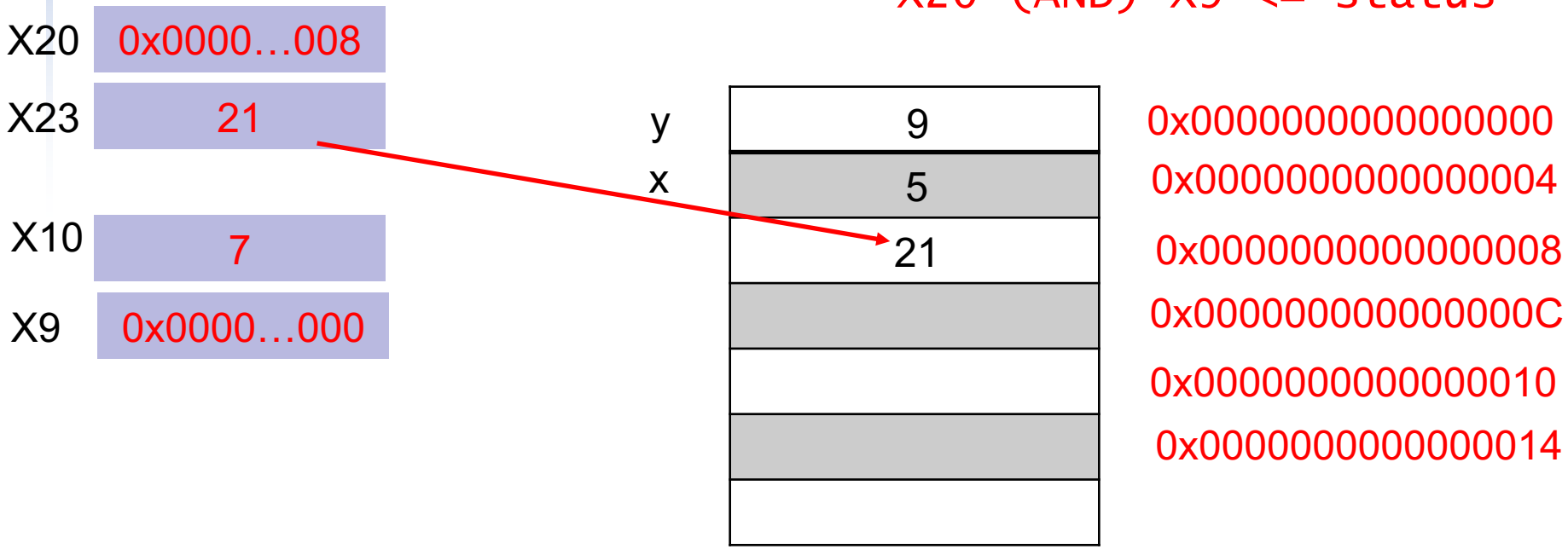
```
again:  LDXR X10,[X20,#0] //load mem content pointed
                          to by [X20, #0] into X10
```



Synchronization in LEGv8

- Example 1: atomic swap

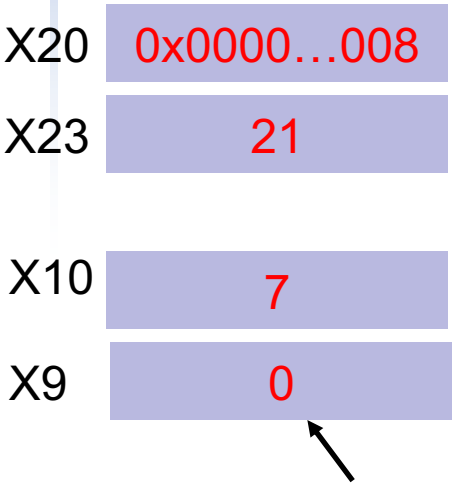
```
again:  LDXR X10,[X20,#0]
        STXR X23,X9,[X20] //Store X23 content into
                            mem location pointed by
                            X20 (AND) X9 <= status
```



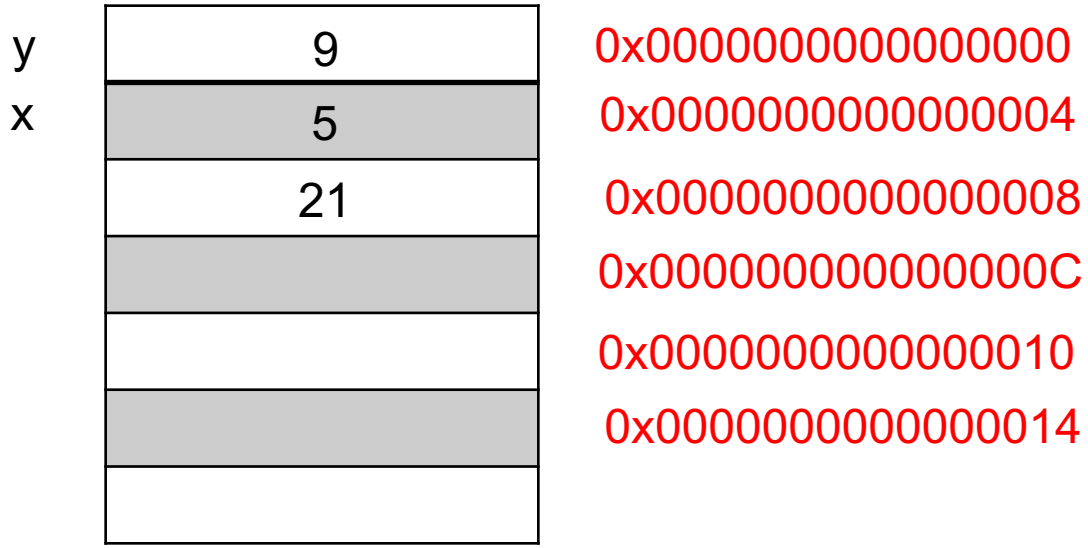
Synchronization in LEGv8

- Example 1: atomic swap

```
again:  LDXR X10,[X20,#0]
        STXR X23,X9,[X20] //X9 = status
        CBNZ X9, again    //was it successful?
```



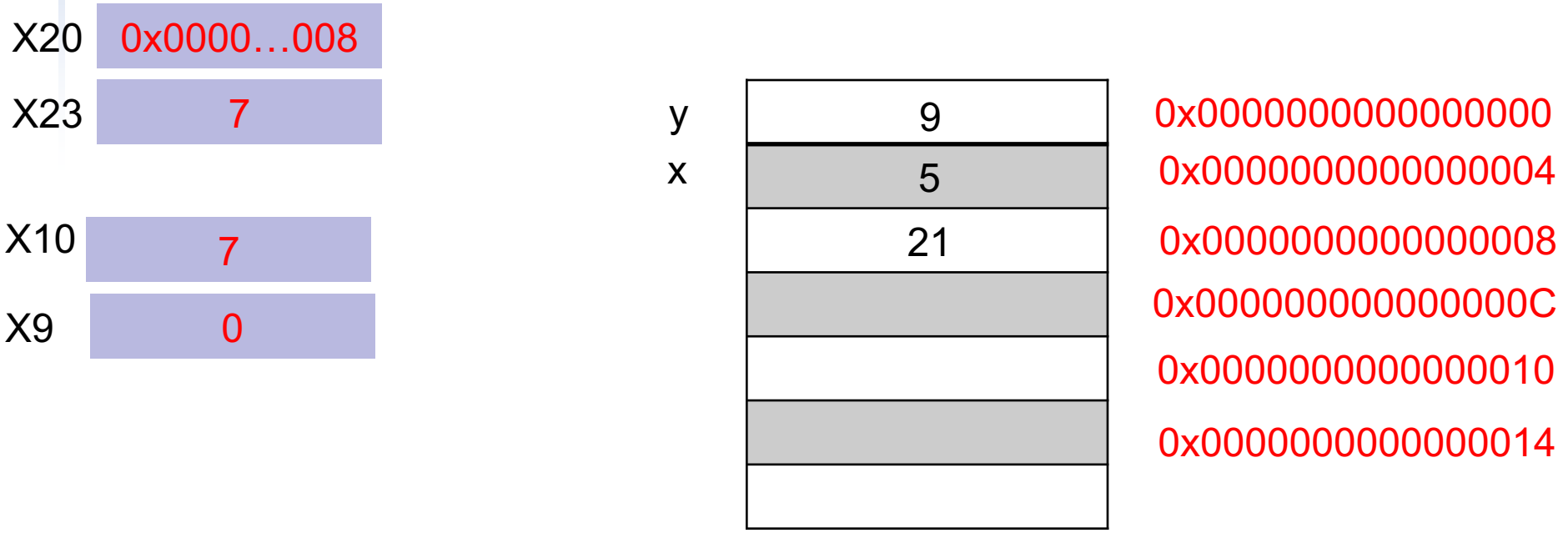
HW reports that during the LDXR and STXR, nothing changed in that mem location (no access).



Synchronization in LEGv8

- Example 1: atomic swap

```
again:  LDXR X10,[X20,#0]
        STXR X23,X9,[X20,#0] // X9 = status
        CBNZ X9, again       // was it successful?
        ADD X23,XZR,X10      // X23 <= loaded value
```



Synchronization in LEGv8

- Example 2: lock (1 = locked , 0 = unlocked)

```
        ADDI X11,XZR,#1      // get ready to lock
again:  LDXR X10,[X20,#0]    // read lock
        CBNZ X10, again      // check if it is 0 yet
        STXR X11, X9, [X20] // attempt to lock
        CBNZ X9,again       // try again if fails
          . . .             // can access/update
        STUR XZR, [X20,#0] // free lock
```

X20	0x0000…014
X11	1
X10	0 = locked
X9	0= success