CE/CZ 3001: Advanced Computer Architecture

(Module 2: Instruction Set Architecture Design)

Dr Smitha K. G.
School of Computer Science
And Engineering

Outline (Part 1/2)

- Introduction to ISA
- ARMv8 ISA: A design example named as LEGv8
  - Instruction format
    - R-format, D-format, I-format, B format and CB fomat
  - Addressing modes
    - Register addressing
    - Immediate addressing
    - Base or displacement addressing
    - PC-relative addressing

Outline (Part 2/2)

- ARMv8 ISA: A design example (continued)
  - Functionality category
    - Arithmetic and Logical instructions
    - Data transfer instructions
    - Conditional instructions
    - Unconditional instructions

- Design issues and relative advantages of RISC and CISC

Introduction to ISA (Part 1/2)

Instruction set architecture (ISA): A set of following specifications which a programmer must know to write a correct and efficient program for a specific machine.

- Instruction format
- Length of instruction and size of field
- Word size :16-bit , 32-bit or 64 bit etc.
- Set of all operations: opcodes/ machine language
- Register file specification: size, width, and its usage of registers in CPU

Introduction to ISA (Part 2/2)

Instruction set architecture (ISA): A set of following specifications which a programmer must know to write a correct and efficient program for a specific machine.

- Memory address space and addressability: no. of addressable locations & bits per location
- Addressing modes: ways of specifying and accessing operand(s): indicates how an address (memory or register) is determined
- Operand locations: all in registers, register and memory or all in memory

ARMv8 ISA(specifically LEGv8): A design example

Memory and register specification, Instruction format, addressing modes and instruction set.

ARM ISA (Part 1/2)

ARM (Advanced RISC Machine, originally Acorn RISC Machine)

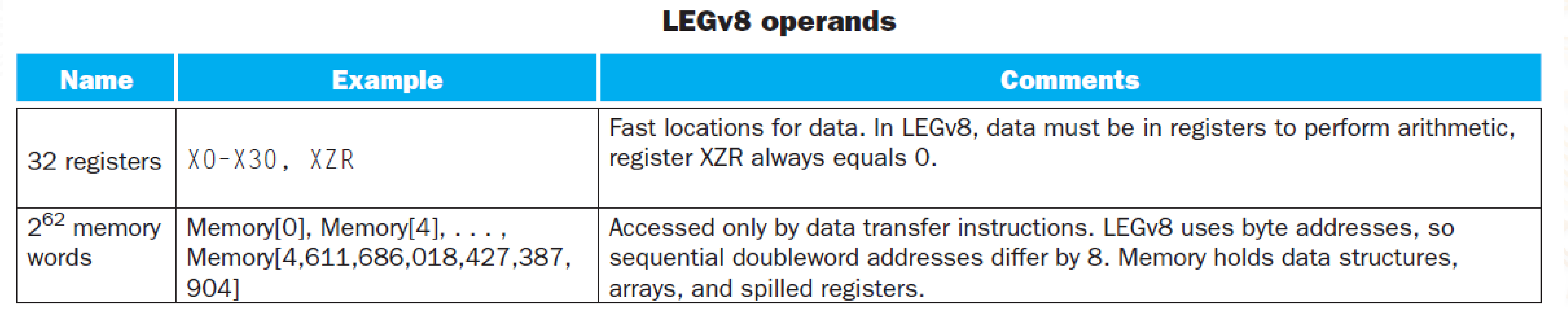
- Simple, sensible, regular, widely used RISC architecture
- Reduced Instruction Set Architecture (RISC) is widely used for its simpler implementation, easier pipelining, and superscalar computing
  - DEC alpha, PowerPC (Mac, IBM servers),SPARC (Sun)
  - ARM processors (smartphones and embedded systems)
    - iPhone 5S (64-bit Apple A7 processor) The A7 includes an Apple designed ARMv8 dual-core CPU, called Cyclone.
  - Many from Intel, AMD, and Atmel, etc.
- Two major revisions of ARM at present: ARMv7 for 32-bit address and ARMv8 for 64-bit address.
- We shall focus on ARMv8.

ARM ISA (Part 2/2)

ARMv8 (specifically LEG(Lessen Extrinsic Garrulity)v8): A design example

- 64 for bit address bus
- 64-bit data
- 32-bit instruction
- The size of a register in the LEGv8 architecture is 64 bits; groups of 64 bits (8 bytes- doubleword).
- 32 register files with each register storing a 64-bit data

groups of 32 bits (4 bytes-word).



LEGv8 operands

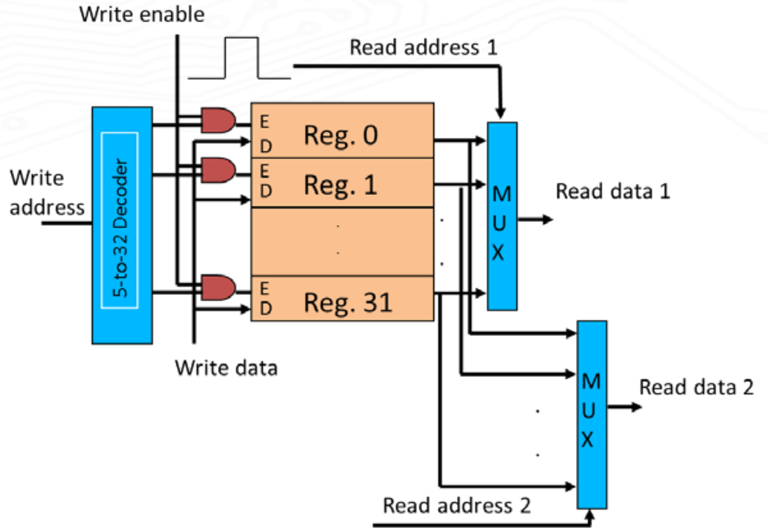
Name	Example	Comments
32 registers	X0–X30, XZR	Fast locations for data. In LEGv8, data must be in registers to perform arithmetic, register XZR always equals 0.
2^{62} memory words	Memory[0], Memory[4], . . . , Memory[4,611,686,018,427,387,904]	Accessed only by data transfer instructions. LEGv8 uses byte addresses, so sequential doubleword addresses differ by 8. Memory holds data structures, arrays, and spilled registers.

Register specification (Part 1/2)

CPU registers: Used for frequently accessed data

General purpose register (GPR) file

- Contains thirty-two 64-bit registers
- Contains 2 read ports and 1 write port



Q: Why only thirty-two registers only? Can it be more?

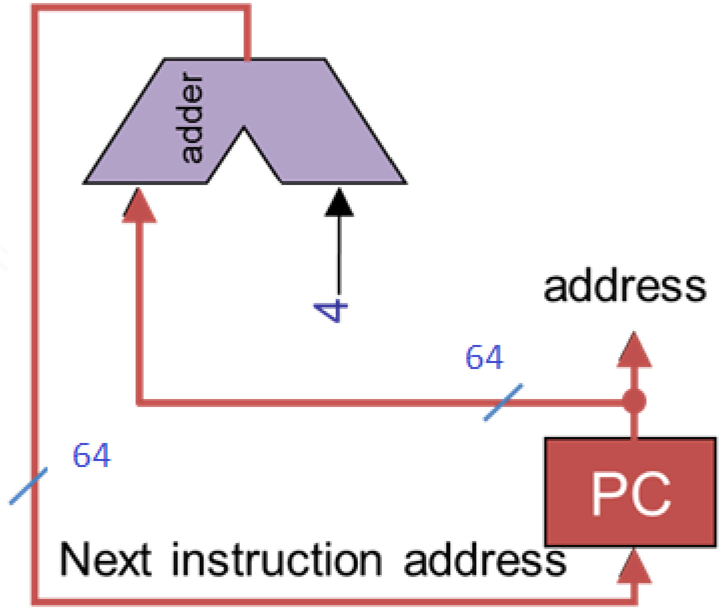
A1: More registers will make it significantly slower

A2: More registers will increase the operand size

Register specification (Part 2/2)

Program counter (PC):

64-bit register that holds the address of the next instruction

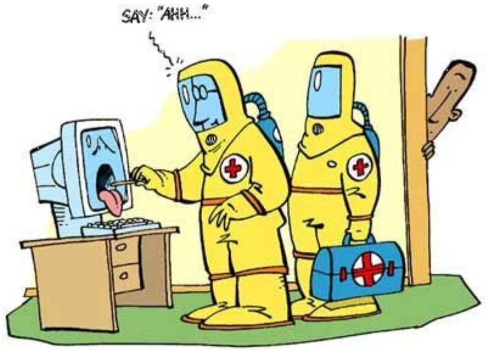


ARM register usage

Each register has 64-bit (2 word- 1 double word)- 8 Bytes (1 byte= 8 bits)

Name	Register number	Usage	Preserved on call?
`X0 – X7`	0-7	arguments/results	no
`X8`	8	Indirect result location register	no
`X9-X15`	9-15	Temporaries	no
`X16(IP0)`	16	May be used by linker as a scratch register; other times used as temporary register	no
`X17(IP1)`	17	May be used by linker as a scratch register; other times used as temporary register	no
`X18`	18	Platform register; other wise a temporary register	no
`X19-X27`	19-27	saved	yes
`X28(SP)`	28	Stack pointer	yes
`X29(FP)`	29	frame pointer	yes
`X30(LR)`	30	Link register (return address)	yes
`X31`	31	The constant value 0	yes

If you don't follow the convention, there will be troubles.



Memory organization (Part 1/3)

64-bit addresses bus of the LEGv8 processor

Address	instruction
0x0000000000000000	0xAC
0x0000000000000001	0x1C
0x0000000000000002	0xAB
0x0000000000000003	0x5E
0x0000000000000004	0x55
……	……
0xFFFFFFFFFFFFFFFC	0x62
0xFFFFFFFFFFFFFFFD	0x03
0xFFFFFFFFFFFFFFFE	0xD3
0xFFFFFFFFFFFFFFFF	0x5A

- LEGv8 memory is byte-addressable: each memory address references one byte
- Hence with 64-bit addresses the processor can use 2^{64} bytes = 16 Exabyte(EB)(1EB=2^60 BYTES)
- In 64-bit ARM processor each doubleword consists of 64 bits, i.e.
  - Word length = 64 bits = 8 bytes

Memory organization (Part 2/3)

Address	instruction
0x0000000000000000	0xAC
0x0000000000000001	0x1C
0x0000000000000002	0xAB
0x0000000000000003	0x5E
……	……
……	……
0xFFFFFFFFFFFFFFFC	0x62
0xFFFFFFFFFFFFFFFD	0x03
0xFFFFFFFFFFFFFFFE	0xD3
0xFFFFFFFFFFFFFFFF	0x5A

1 byte

Address	Instruction
0x0000000000000000	0x5EAB1CAC
0x0000000000000004	……
0x0000000000000008	……
0x000000000000000C	……
……	……
……	……
0xFFFFFFFFFFFFFFF0	……
0xFFFFFFFFFFFFFFF4	……
0xFFFFFFFFFFFFFFF8	……
0xFFFFFFFFFFFFFFFC	0x5AD30362

Little Endian method

- Instruction size (LEGv8) is 32 bits= 4 bytes
- Address of each instruction is a multiple of 4: the last two bits are 00
- Address of next instruction is 4 more than that of current instruction

Memory organization (Part 3/3)

Address	Data	Address	data
0x0000000000000000	0xAC	0x….00000	0x5AD303625EAB1CAC
0x0000000000000001	0x1C	0x….00008	……
0x0000000000000002	0xAB	0x….00010	……
0x0000000000000003	0x5E	0x….00018	……
0x0000000000000004	0x62	……	……
0x0000000000000005	0x03	……	……
0x0000000000000006	0xD3	……	……
0x0000000000000007	0x5A	……	……
……	……	……	……
……	……	……	……

- Data size (in reg file and Dmem)(LEGv8) is 64 bits= 8 bytes
- Address of next 64 bit data from the data memory is 8 more than that of current 64 bit data

Little Endian method

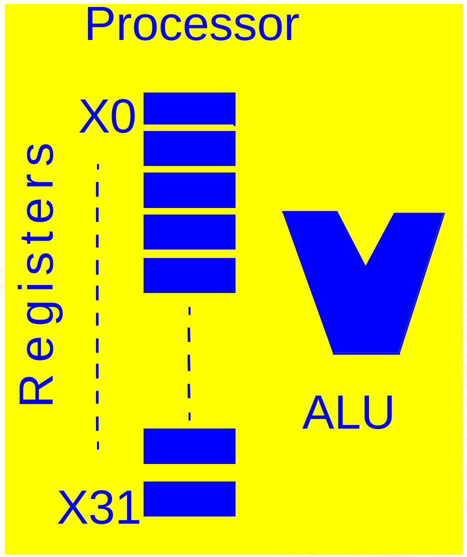
From C to machine language

- C statement → f = (g + h) – (i + j)
- Pseudo code instructions

```
add t0, g, h
add t1, i, j
sub f, t0, t1
```

- Opcode/mnemonic, operands, source/destination
- Operands must be **<u>registers, not variables</u>**

```
ADD X9, X20, X21
ADD X10, X22, X23
SUB X19, X9, X10
```



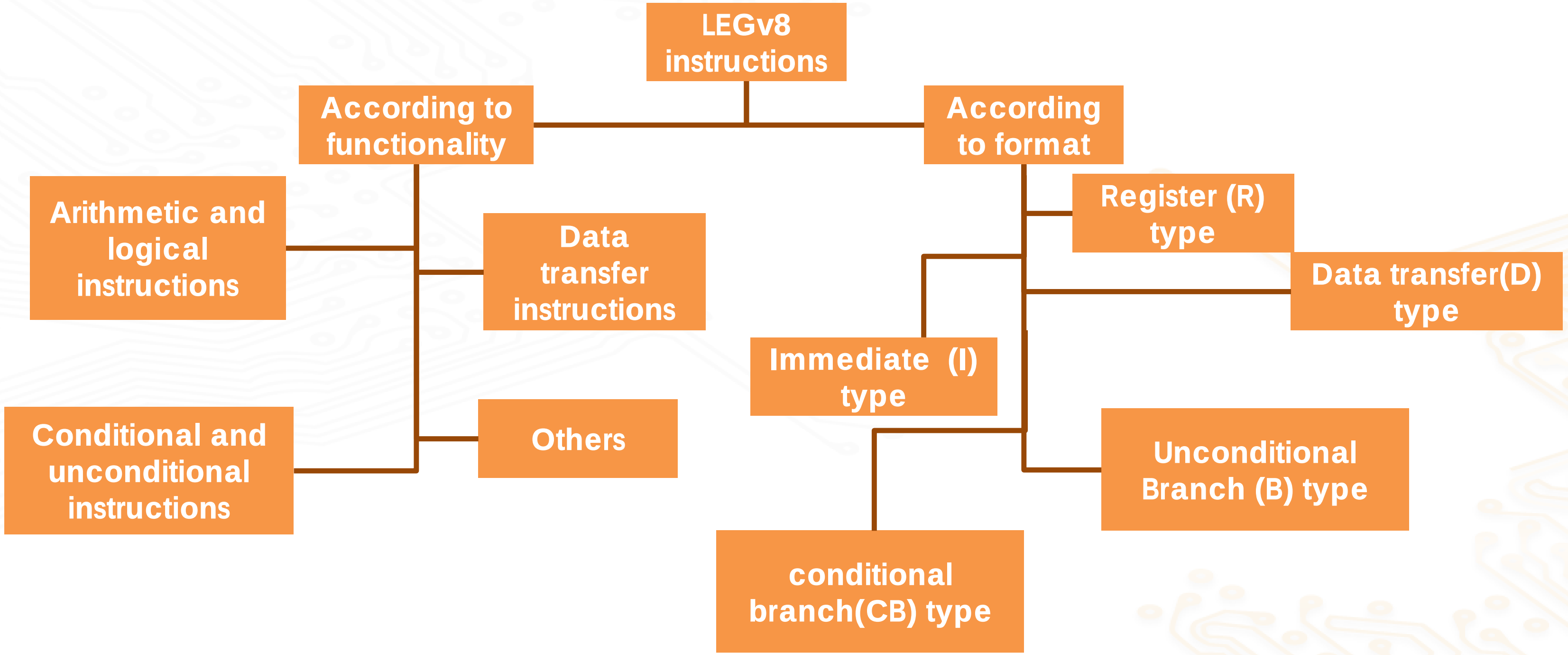
Each instruction need to be encoded in 32 bits:- More on instruction format

Classification of LEGv8 instructions

- Based on Instruction Format:
  - Each LEGv8 instruction is 32 bits and it has different fields: example: Opcode , Destination and Source registers , Address of destination etc
    - Opcode: Portion of a machine language instruction that specifies the operation to be performed. (add, sub, or load, and store etc.)

- Based on Functionality:
  - Arithmetic and Logical instructions
  - Data transfer instructions
  - Conditional instructions
  - Unconditional instructions

Classification of LEGv8 instructions



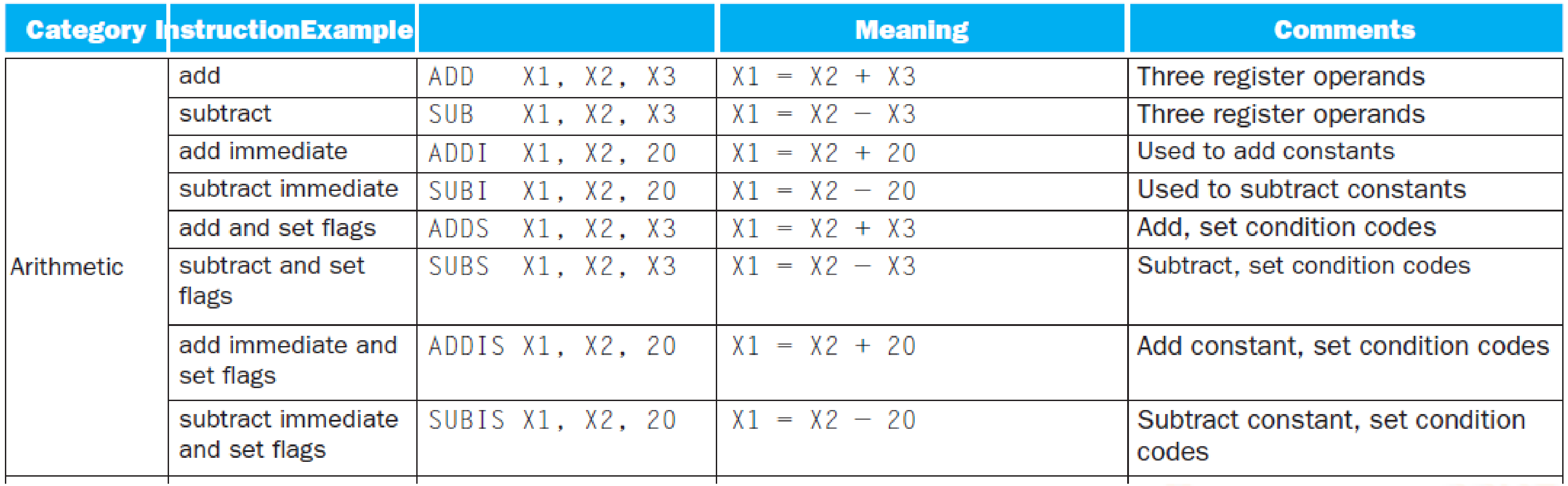
LEGv8 instructions
According to functionality
According to format
Arithmetic and logical instructions
Data transfer instructions
Conditional and unconditional instructions
Others
Register (R) type
Data transfer(D) type
Immediate (I) type
Unconditional Branch (B) type
conditional branch(CB) type

Classification of LEGv8 Instruction based on functionality

- ALU instructions
  - **arithmetic operations: ADD, SUB, ADDI, SUBI and their variants**
  - **logical operations: AND, ORR, EOR, ANDI,ORRI, EORI etc.**
  - **shift operations: LSL, LSR etc.**
- Data transfer: **LDUR, STUR, LDURB (load byte), STURB (store byte)** etc.
  - load and store: memory to register (load) and register to memory (store)
- Conditional branch and unconditional branch: changes the sequence of execution of instructions

ALU instructions – arithmetic operations

arithmetic operations: ADD, SUB, ADDI, SUBI and their variants



Category	InstructionExample		Meaning	Comments
Arithmetic	add	`ADD X1, X2, X3`	`X1 = X2 + X3`	Three register operands
	subtract	`SUB X1, X2, X3`	`X1 = X2 – X3`	Three register operands
	add immediate	`ADDI X1, X2, 20`	`X1 = X2 + 20`	Used to add constants
	subtract immediate	`SUBI X1, X2, 20`	`X1 = X2 – 20`	Used to subtract constants
	add and set flags	`ADDS X1, X2, X3`	`X1 = X2 + X3`	Add, set condition codes
	subtract and set flags	`SUBS X1, X2, X3`	`X1 = X2 – X3`	Subtract, set condition codes
	add immediate and set flags	`ADDIS X1, X2, 20`	`X1 = X2 + 20`	Add constant, set condition codes
	subtract immediate and set flags	`SUBIS X1, X2, 20`	`X1 = X2 – 20`	Subtract constant, set condition codes

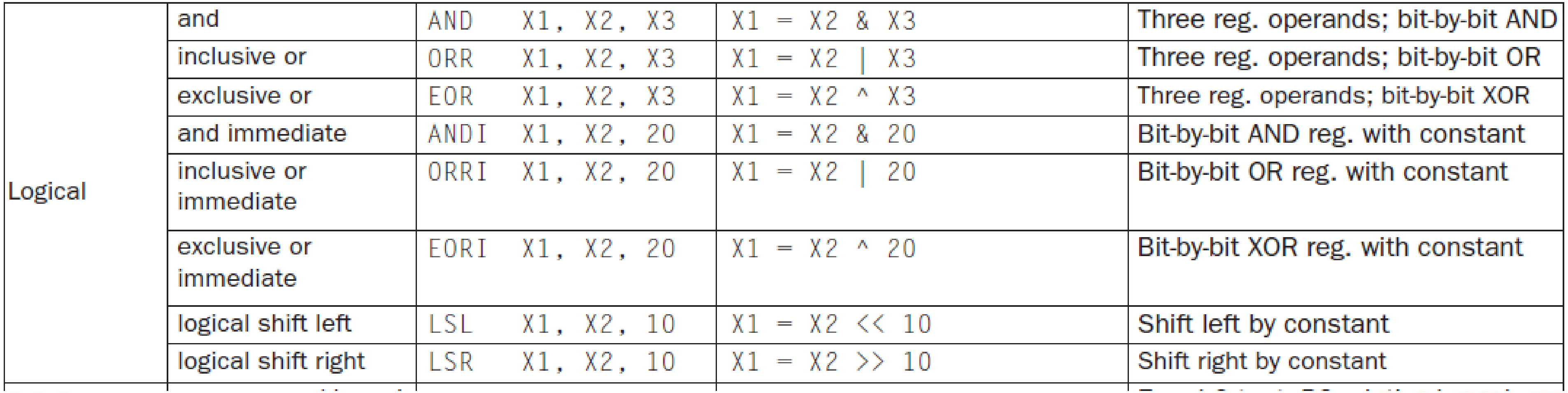
Condition codes, set from arithmetic instruction with S-suffix (ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS)

negative (N): result had 1 in MSB
zero (Z): result was 0

overlow (V): result overflowed
carry (C): result had carryout from MSB

ALU – logical operations

Logical operations: AND, ORR, EOR, ANDI and their variants

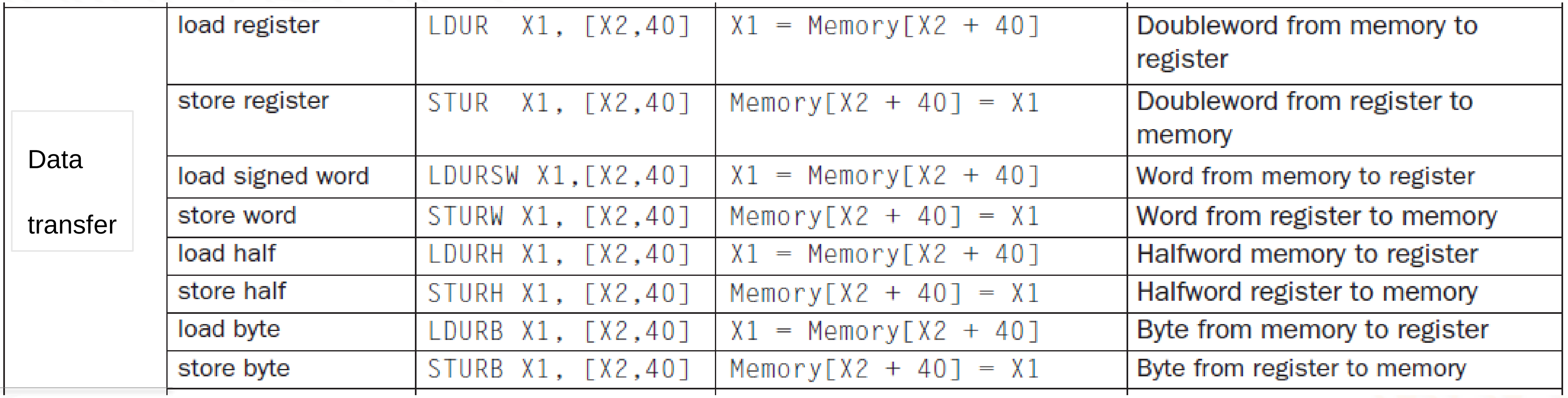


Logical	and	AND X1, X2, X3	X1 = X2 & X3	Three reg. operands; bit-by-bit AND
	inclusive or	ORR X1, X2, X3	X1 = X2 | X3	Three reg. operands; bit-by-bit OR
	exclusive or	EOR X1, X2, X3	X1 = X2 ^ X3	Three reg. operands; bit-by-bit XOR
	and immediate	ANDI X1, X2, 20	X1 = X2 & 20	Bit-by-bit AND reg. with constant
	inclusive or immediate	ORRI X1, X2, 20	X1 = X2 | 20	Bit-by-bit OR reg. with constant
	exclusive or immediate	EORI X1, X2, 20	X1 = X2 ^ 20	Bit-by-bit XOR reg. with constant
	logical shift left	LSL X1, X2, 10	X1 = X2 << 10	Shift left by constant
	logical shift right	LSR X1, X2, 10	X1 = X2 >> 10	Shift right by constant

Logical shift left : Shift left and fill with 0 bits
LSL by i bits multiplies by 2^i
Logical shift right: Shift right and fill with 0 bits
LSR by i bits divides by 2^i (unsigned only)

Data transfer operations

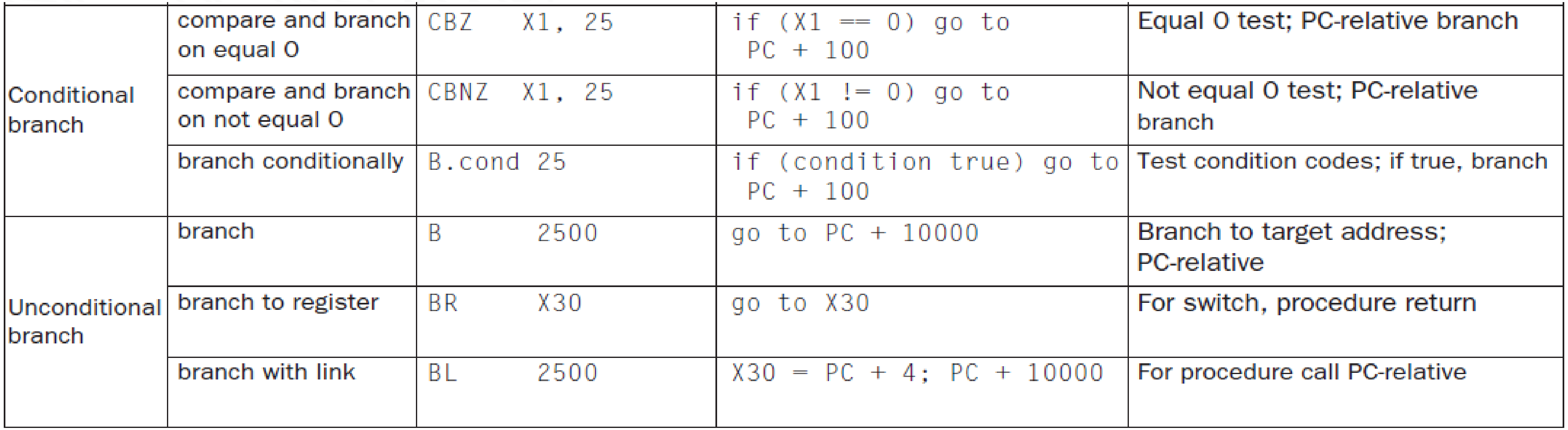
Data transfer operations: LDUR, STUR, LDURB, STURB



Data transfer	load register	`LDUR X1, [X2,40]`	`X1 = Memory[X2 + 40]`	Doubleword from memory to register
	store register	`STUR X1, [X2,40]`	`Memory[X2 + 40] = X1`	Doubleword from register to memory
	load signed word	`LDURSW X1,[X2,40]`	`X1 = Memory[X2 + 40]`	Word from memory to register
	store word	`STURW X1, [X2,40]`	`Memory[X2 + 40] = X1`	Word from register to memory
	load half	`LDURH X1, [X2,40]`	`X1 = Memory[X2 + 40]`	Halfword memory to register
	store half	`STURH X1, [X2,40]`	`Memory[X2 + 40] = X1`	Halfword register to memory
	load byte	`LDURB X1, [X2,40]`	`X1 = Memory[X2 + 40]`	Byte from memory to register
	store byte	`STURB X1, [X2,40]`	`Memory[X2 + 40] = X1`	Byte from register to memory

Conditional and unconditional operations

Control Flow operations: **B, B. cond, CBZ, CBNZ, BR, BL**



Conditional branch	compare and branch on equal 0	`CBZ X1, 25`	`if (X1 == 0) go to PC + 100`	Equal 0 test; PC-relative branch
	compare and branch on not equal 0	`CBNZ X1, 25`	`if (X1 != 0) go to PC + 100`	Not equal 0 test; PC-relative branch
	branch conditionally	`B.cond 25`	`if (condition true) go to PC + 100`	Test condition codes; if true, branch
Unconditional branch	branch	`B 2500`	`go to PC + 10000`	Branch to target address; PC-relative
	branch to register	`BR X30`	`go to X30`	For switch, procedure return
	branch with link	`BL 2500`	`X30 = PC + 4; PC + 10000`	For procedure call PC-relative

More Conditional operators

- Condition codes, set from arithmetic instruction with S-suffix (ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS)
  - negative (N): result had 1 in MSB
  - zero (Z): result was 0
  - overlow (V): result overflowed
  - carry (C): result had carryout from MSB
- Use subtract to set flags, then conditionally branch:
  - **B.EQ**
  - **B.NE**
  - **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)

Classification of LEGv8 Instruction based on Instruction format -Register (R) type

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

- All data values (operands/result) are located in registers.
- **Rn** and **Rm** : specify the first and the second source registers
- **Rd:** specifies the destination register
- **shamt** stands for shift amount: specifies the number of bit positions to be shifted (used in shift instructions.)
- **Opcode**: Operation Code: specifies the type of instruction.

Register(R)-type example

opcode	Rm	shamt	Rn	Rd
11 bits	5 bits	6 bits	5 bits	5 bits

Register (R) type

ADD X9,X20,X21

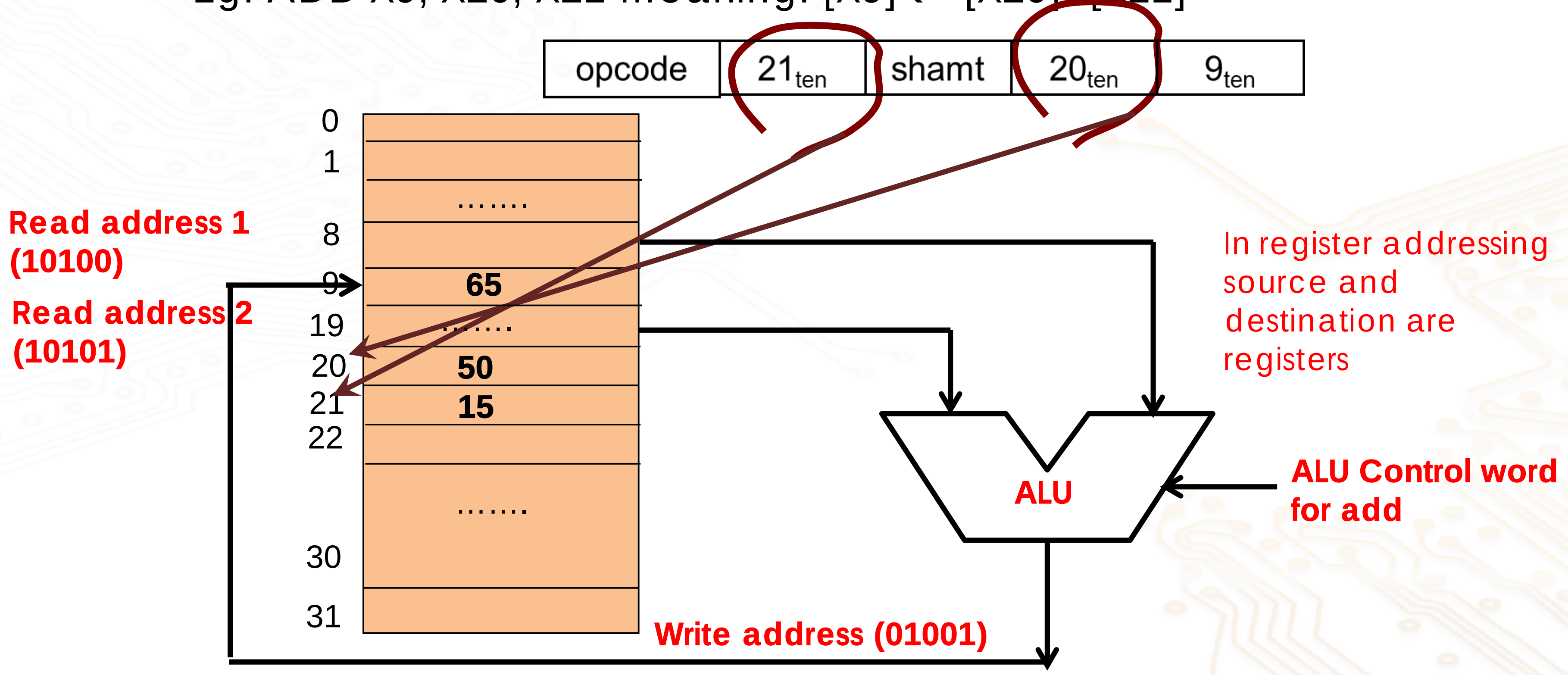
1112_{ten}	21_{ten}	0_{ten}	20_{ten}	9_{ten}

10001011000_{two}	10101_{two}	000000_{two}	10100_{two}	01001_{two}

$1000\ 1011\ 0001\ 0101\ 0000\ 0010\ 1000\ 1001_{two} = 8B150289_{16}$

Register (R) type – Register Addressing

- Instruction structure: <operation> <Rd>, <Rn>, <Rm>
- Eg: ADD X9, X20, X21 meaning: [X9]← [X20]+[X21]



In register addressing source and destination are registers

LEGv8 Instruction format – Data transfer (D) type

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- Load/Store instructions,
- **Rn**: specify the base registers
- "address": constant **offset** from contents of base register (+/- 32 doublewords)
- **Rt**: specifies the destination register (load) or source (store) register number (why Rt and not Rd?)
  - ALU calculates the address (address+[Rn])
  - Data at memory location (address+[Rn]) is read
- The read data is saved in the destination register (Rt).

LEGv8 Instruction format – Data transfer (D) type- example

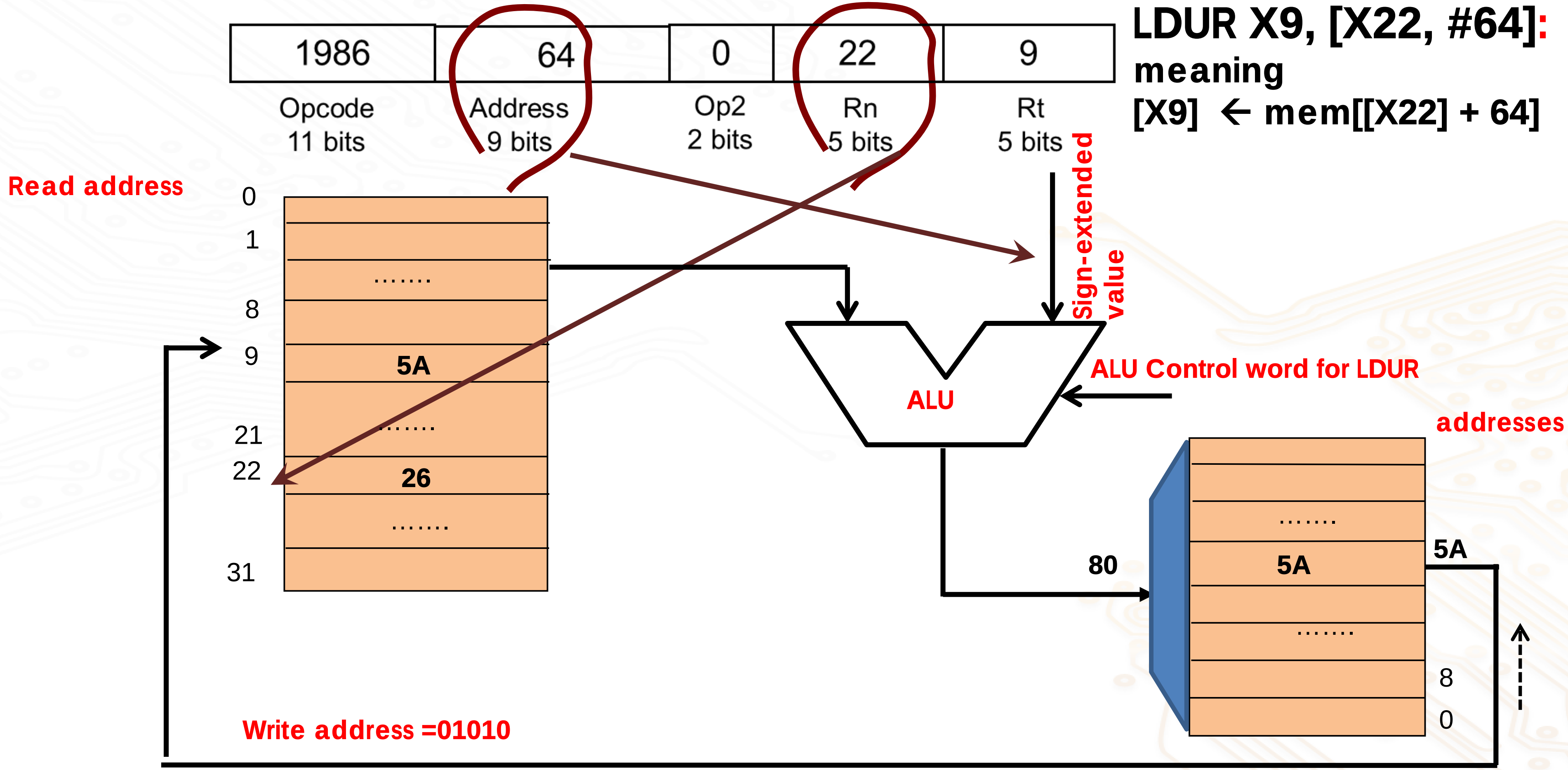
opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- LDUR (load register)
- LDUR X9, [X22, #64] //Temporary REG X9 is loaded with value from memory whose address is =content of X22 +64

1986	64	0	22	9
Opcode 11 bits	Address 9 bits	Op2 2 bits	Rn 5 bits	Rt 5 bits

Base/ Displacement Addressing (load word)

LDUR X9, [X22, #64]:
meaning
[X9] ← mem[[X22] + 64]



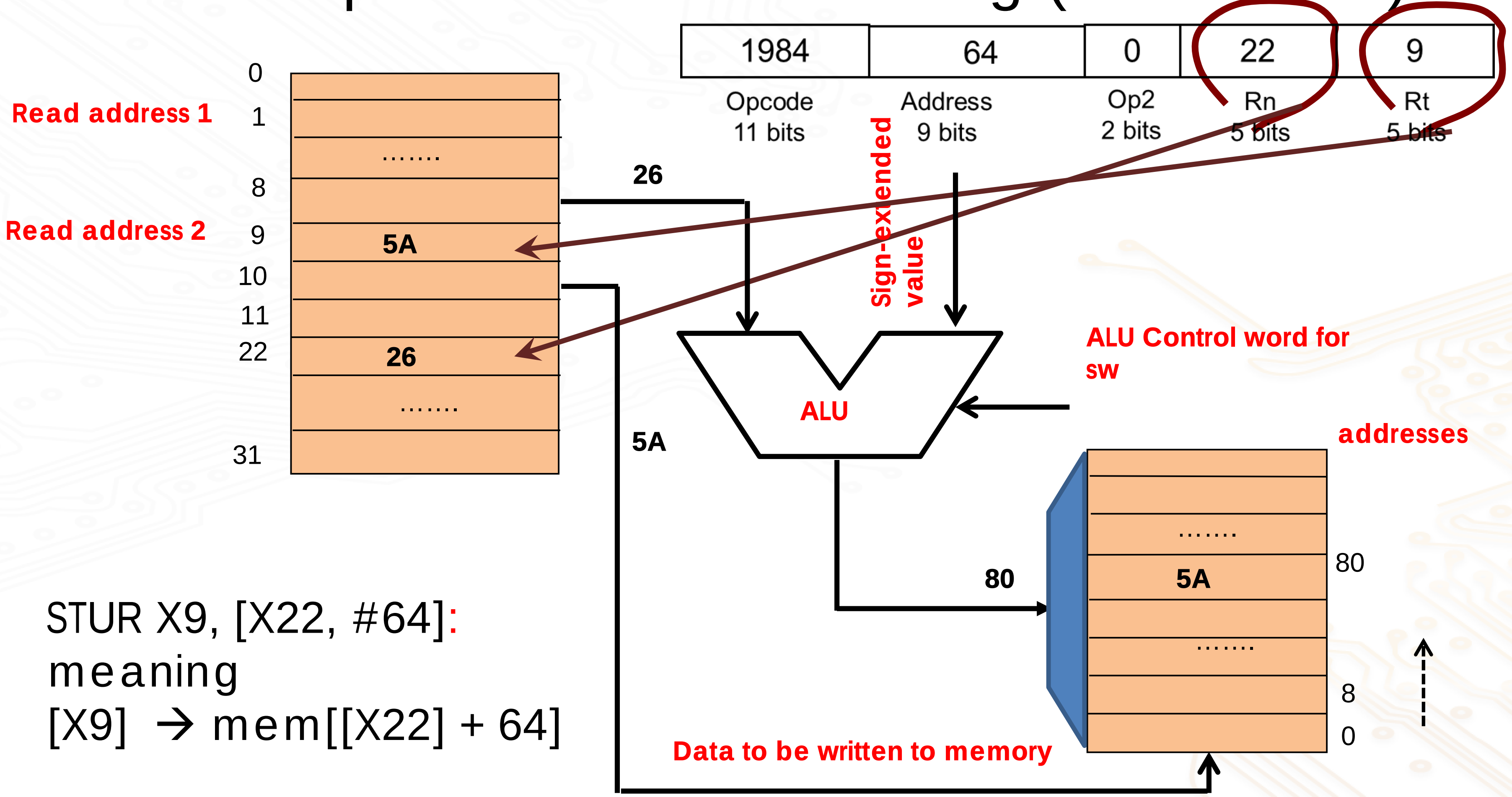
LEGv8 Instruction format – Data transfer (D) type- STUR

opcode	address	op2	Rn	Rt
11 bits	9 bits	2 bits	5 bits	5 bits

- STUR (store register)
- STUR X9, [X22, #64] //Value from REG X9 is loaded to memory whose address is =content of X22 +64

1984	64	0	22	9
Opcode 11 bits	Address 9 bits	Op2 2 bits	Rn 5 bits	Rt 5 bits

Base/displacement Addressing (store word)



STUR X9, [X22, #64]:
meaning
[X9] → mem[[X22] + 64]

LEGv8 Instruction format – Immediate(I) type

opcode	immediate	Rn	Rd
10 bits	12 bits	5 bits	5 bits

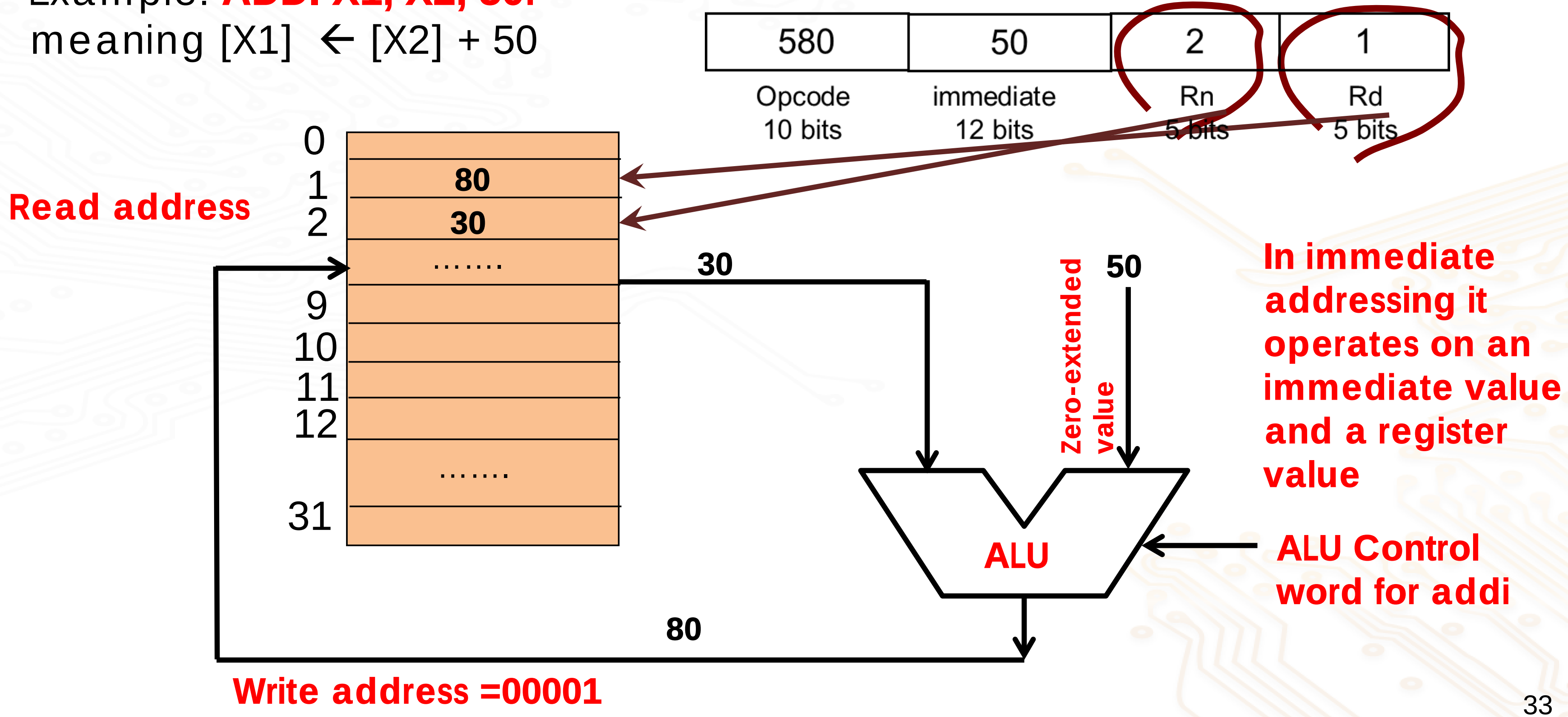
– Rn: source register
– Rd: destination register

580	50	2	1
Opcode 10 bits	immediate 12 bits	Rn 5 bits	Rd 5 bits

- Immediate field is zero-extended
- Opcode is reduced to 10 bits to have more range for immediate field.
  – Example: ADDI X1, X2, #50 ([X1] ← [X2]+ 50)

Immediate (I) type – Immediate Addressing

- Example: **ADDI X1, X2, 50:** meaning [X1] ← [X2] + 50



In immediate addressing it operates on an immediate value and a register value

Conditional instructions

- 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;

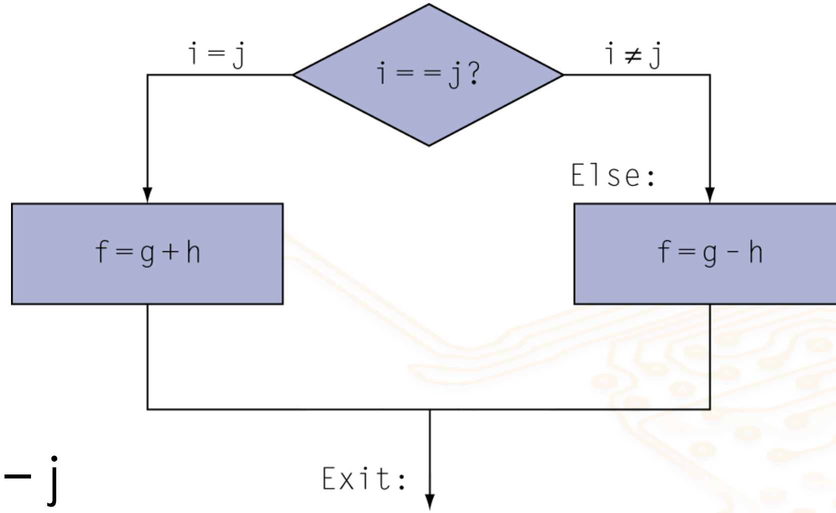
- `B L1`
  - **branch unconditionally** to instruction labeled L1;

Compiling “IF” statements

- C code:

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

  – f, g, h, i, j … in X19 to X23



- Compiled LEGv8 code:

```
      SUB X9,X22,X23     //X9 = i – j
      CBNZ X9,Else       //go to Else if i ≠ j (X9 ≠ 0)
      ADD X19,X20,X21    //f = g + h (skipped if i ≠ j)
      B Exit
Else:   SUB X9,X22,x23   //// f = g – h (skipped if i = j)
Exit: …
```

LEGv8 Instruction format – conditional branch (CB) type

- – Based on a condition
  - CBZ (if register value is zero, then branch)
  - CBNZ (if register value is nonzero, then branch)

opcode	address	Rt
8 bits	19 bits	5 bits

- – Example: CBZ X19, Exit // go to Exit if X19 == 0

180	Exit	19
8 bits	19 bits	5 bits

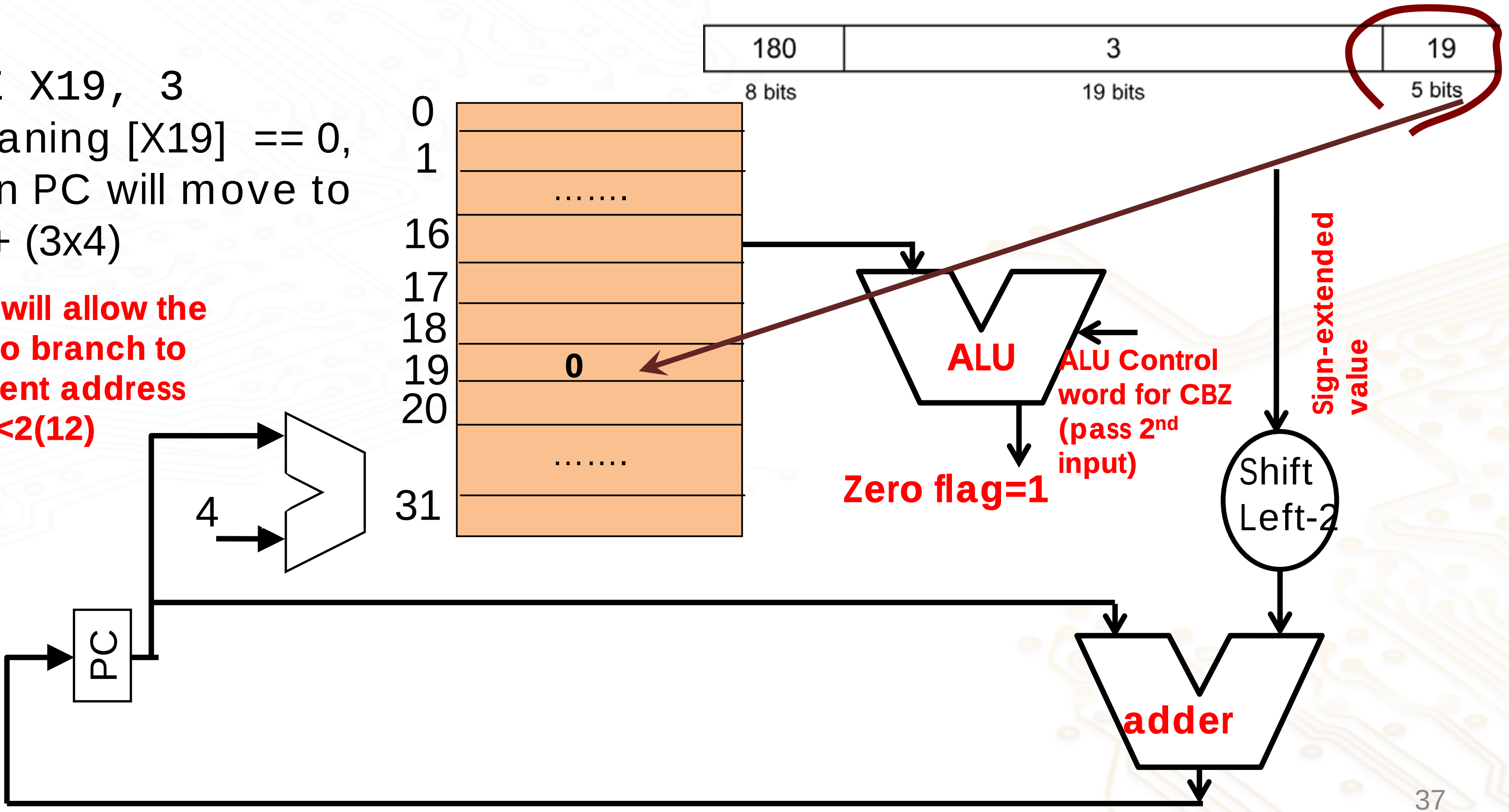
- Addressing mode is (Program Counter) PC-relative
  - – Address = PC + address (from instruction)

PC relative addressing (Part 1/2)

```
CBZ X19, 3
```

meaning [X19] == 0, then PC will move to PC+ (3x4)

This will allow the PC to branch to current address +3<<2(12)



PC relative addressing (Part 2/2)

```
SUB X9,X22,X23
CBNZ X9,Else
ADD X19,X20,X21
B Exit
Else:SUB X9,X22,x23
Exit: …
```

`CBNZ X9,3` **meaning [X9] ≠ 0, then PC will move to PC+ (3x4)**

	PC Address	Instruction mem
	0x0000000000000000	SUB X9,X22,X23
	0x0000000000000004	CBNZ X9,3
	0x0000000000000008	ADD X19,X20,X21
	0x000000000000000C	B Exit
Else:	0x0000000000000010	SUB X9,X22,x23
Exit:	0x0000000000000014	……
	……	……
	0xFFFFFFFFFFFFFFF4	……
	0xFFFFFFFFFFFFFFF8	……
	0xFFFFFFFFFFFFFFFC	……

Step 1 → [X9] ≠ 0, then nPC=PC+(3x4)

Skip instructions

newPC (nPC) points to this location

LEGv8 Instruction format – Unconditional branch (B) type

opcode	address
6 bits	26 bits

- B-type instructions: are used when a unconditional branch in instruction sequence is to be performed.
- For branch, we need to change the content of program counter (PC),which stores the address of the next instruction to be executed.
- Syntax: B offset. (example B 8 (branch to address 8)

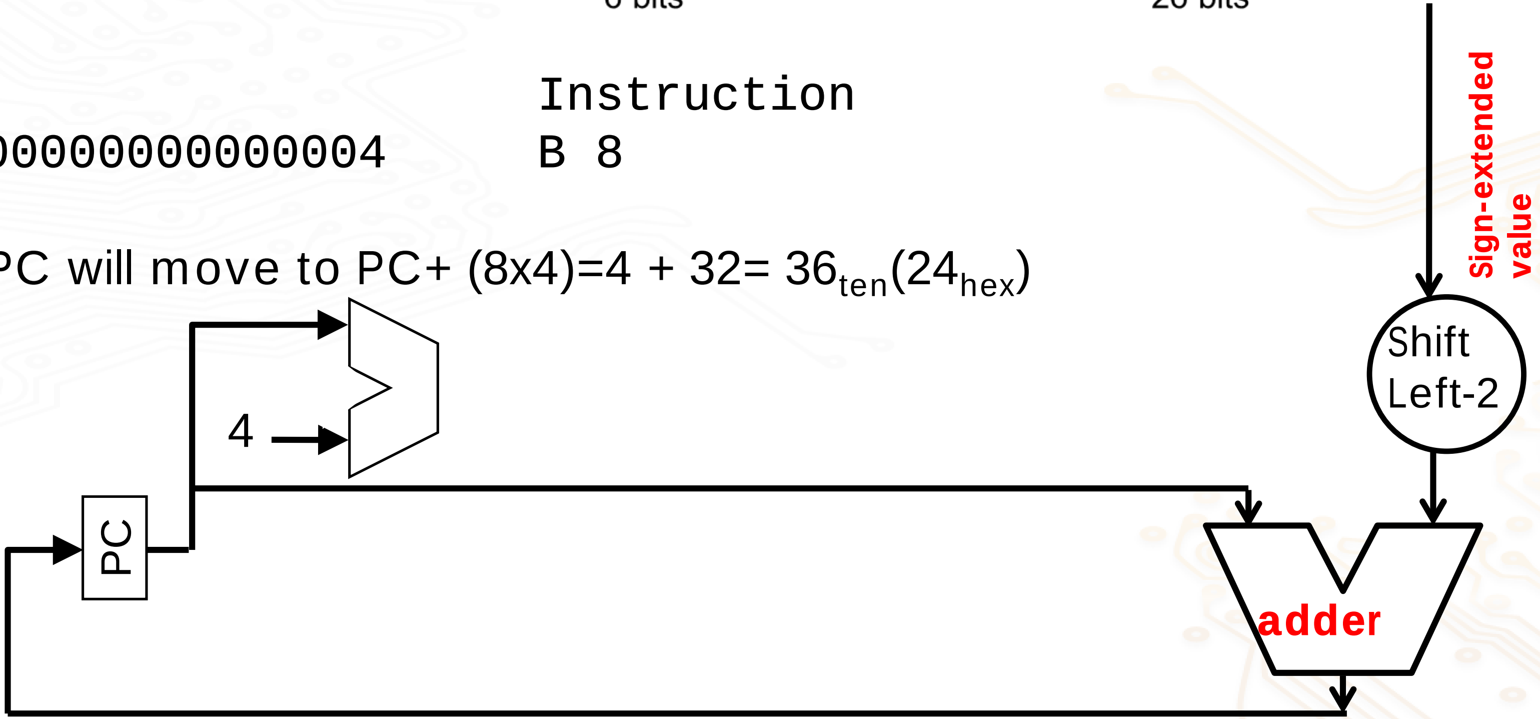
5	8_{ten}
6 bits	26 bits

PC relative addressing mode:- Unconditional branch (B) type

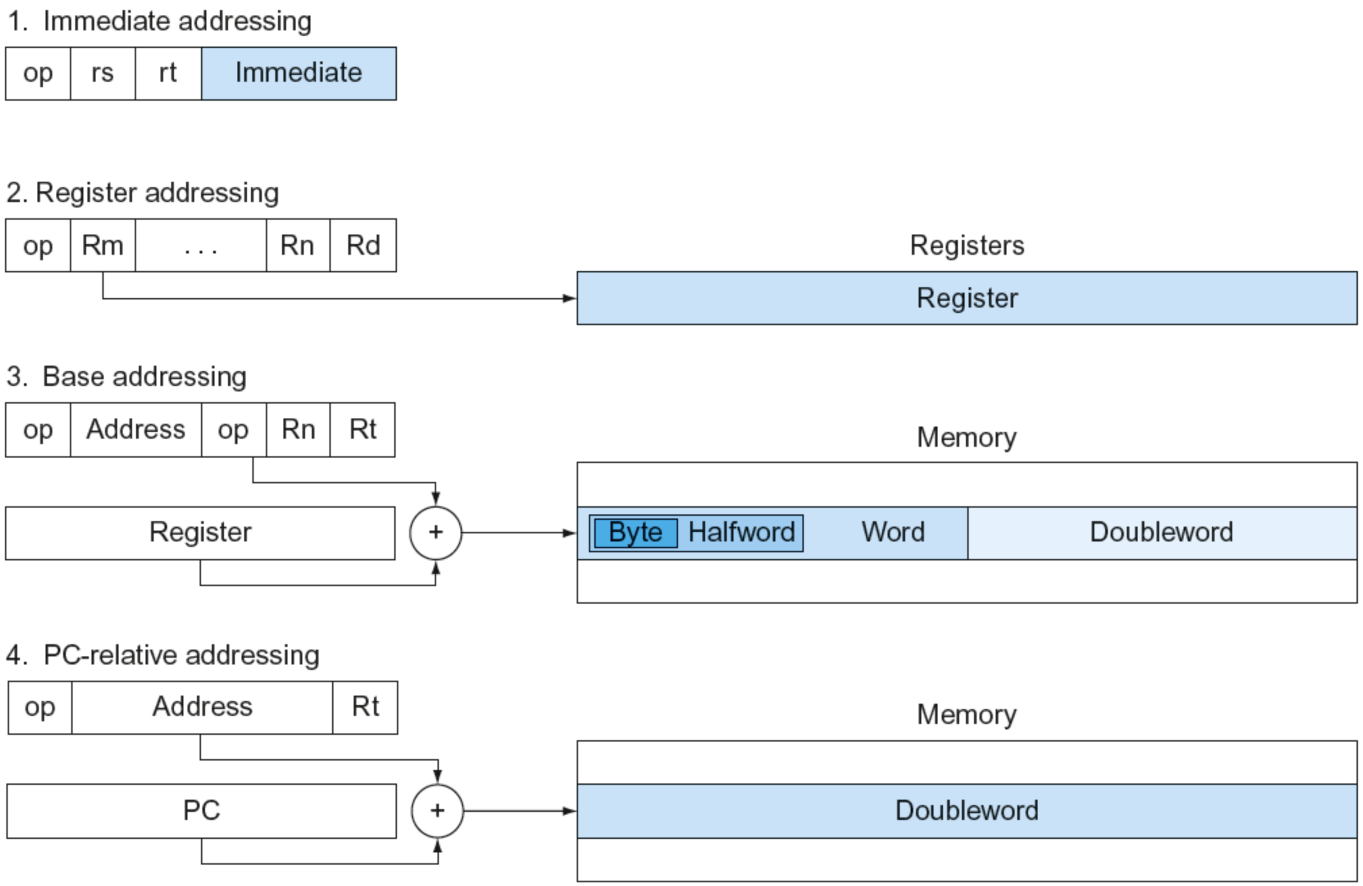
5	8_{ten}
6 bits	26 bits

```
PC                                  Instruction
0X0000000000000004                  B 8
```

then PC will move to PC+ (8x4)=4 + 32= 36_{ten}(24_{hex})



Addressing mode summary



ISA Design Issues

ISA design issues and goals

- Design goals
  - Simple microarchitecture implementation: low hardware complexity
  - High performance: execution time = IC x CPI x clock period → balanced optimization
  - Low Power: less data movement, scope for parallel and pipeline implementation
  - Programmability: easy to express the high-level programs by machine language instructions
  - Compatible: programmability across generation
- Design issues
  - Selection of operations to be executed by the instructions → size of instruction set and the size of opcode
  - Operand location: registers, or memory or in both
  - Addressing mode and instruction formats
  - Register size, word size, memory address space and address size

CISC and RISC design policy (Part 1/2)

- CISC and RISC are two basic philosophies of ISA design.
- CISC stands for Complex Instruction Set Computer (developed in early 60s: examples PDP-11, DEC system, Intel 80x86, and Motorola 68K series).
  - Aims at a small program memory and less compiler workload.
  - Involves large instruction set comprised of complex and specialized instructions,e.g., transcendental functions (exponentiation, logarithm) and string manipulation, to have fewer instructions per task.

CISC and RISC design policy (Part 2/2)

- RISC stands for Reduced Instruction Set Computer (relatively new: started in late 1970: 1980 ARM developed by British manufacture Acorn Computers: early 1990s IBM POWER (Performance Optimization With Enhanced RISC) architecture.
  - Based on the 80/20 rule: 80% of instructions use only 20% of the instruction set.
  - All computing tasks can be performed ay a small instruction set comprised of simple instructions.
  - Each of those instructions can be executed in a single clock cycle of short duration.

RISC design features

Small and simple instruction set:

- Fixed opcode-width and fewer addressing modes, fixed instruction-length, fewer instruction formats with common fields.
  - Advantages: simple and fast hardwired decoding and control generation, shorter clock period, load/store architecture.
- Only two instructions (load & store) and only one addressing mode for memory access: all operands and destination are located in registers.
  - Advantages: memory access and operand processing are performed by separate instructions: register access is faster) shorter clock period single-cycle implementation.
- All instructions will be executed in a single cycle.
  - Advantages: makes the superscalar/instruction level pipelining simpler.

CISC design features

- Source and destination could be in registers/memory/ both.
  - relatively longer clock period involves less registers.
- Several addressing modes and multiple displacement (offset) sizes.
  - instruction length varies according to the addressing mode.
  - complex instruction-decoding logic.
- Instructions are microcoded, executed in multiple clock cycles.
- Program code size is relatively small.
- CISC implementations translate the instructions to RISC like microinstructions to realize pipelining.
- Less compilation effort: complexity lies in micro-program level.

CISC vs RISC (Part 1/3)

CISC	RISC
Many complex instructions in the instruction set	Fewer simple instructions in the instruction set
Many formats, & several addressing modes to support complex instructions	Fewer instruction formats, & a few addressing modes
Instruction length varies according to the addressing mode	Fixed instruction length) simpler implementation

CISC vs RISC (Part 2/3)

CISC	RISC
Instructions are microcoded and executed in multiple clock cycles	Hardwired decoding: single cycle instruction execution
Memory can be referenced by many different modes	Only load/store instructions can reference memory
Operands could be memory: higher clock period : less number of registers	Operands in register for faster clocking: more registers : less memory access

CISC vs RISC (Part 3/3)

CISC

- Difficult to pipeline and super-scalar implementation
- Program code size is relatively small: complexity is in micro-program level
- Higher complexity of instruction implementation:
  CPI more than 1.

RISC

- Easy to pipeline: and super-scalar implementation
- Program code size is usually large: complexity is in the compiler
- More compile time: higher register and more cache area