Xi'an Jiaotong-Liverpool University
西交利物浦大学

CPT210 - Microprocessor

Lecture 5 - Assemblers and Instruction Encoding

Overview

- Lecture 1-4 Review
  - Computer Architecture
  - BUSes
  - Cache and Memory
  - Endianness
  - Data Representation

Break: SURF Introduction

- Lecture 5
  - ARM Assemblers
    - What assembler is
    - Why use assembler
    - Differences between assembler and compiler
  - Instruction encoding
    - Condition field
    - Immediate flag
    - Operation code
    - Status update flag
    - First operand register
    - Destination register
    - Second operand
  - Rotation and Shift
    - ROR
    - LSR
    - ASR
    - LSL

Before End: Assessment Explaination

Teaching Team & Office hours

Teaching staff (3)

Filbert JUWONO

Co-lecturer

- Monday 14:00 – 16:00
- Office at SC248

Lijie YAO

Module Leader

- Tuesday 16:00 – 18:00
- Office at SC564D

Kok Hoe WONG

Lab Instructor

- Wednesday 09:00 – 11:00
- Office at SD431

Review of the Lecture 1-4

- *Computer Architecture*

Feature	Von Neumann Architecture	Harvard Architecture
Memory Structure	Shared memory for data and instructions	Separate memory for data and instructions
Bus Structure	Single bus for both	Separate buses for each
Access to Instructions/Data	One at a time (either instruction or data)	Simultaneous access
Cost & Complexity	Simpler and cheaper	More complex and expensive
Speed	Slower due to **bottleneck**	Faster due to parallelism
Application	General computing (e.g., PCs)	Embedded systems, real-time systems

Review of the Lecture 1-4

- *Computer Architecture*

➢ **The Von Neumann Bottleneck**

Problem:
Because the CPU can only access either an instruction or data at a time through a single bus, it often has to wait for memory, limiting performance.

One Solution:
Modern CPUs often use **caches** or **modified Harvard architectures** to reduce this bottleneck.

out of the scope of this module, reverse for self-learning

Review of the Lecture 1-4

- *Computer Architecture*

➢ **Take-away information**

🎓 **"Von Neumann shares one wire, Harvard splits and runs faster."**

This means Von Neumann uses a single shared bus, while Harvard separates instruction and data paths for faster access.

Review of the Lecture 1-4

- *BUSes*

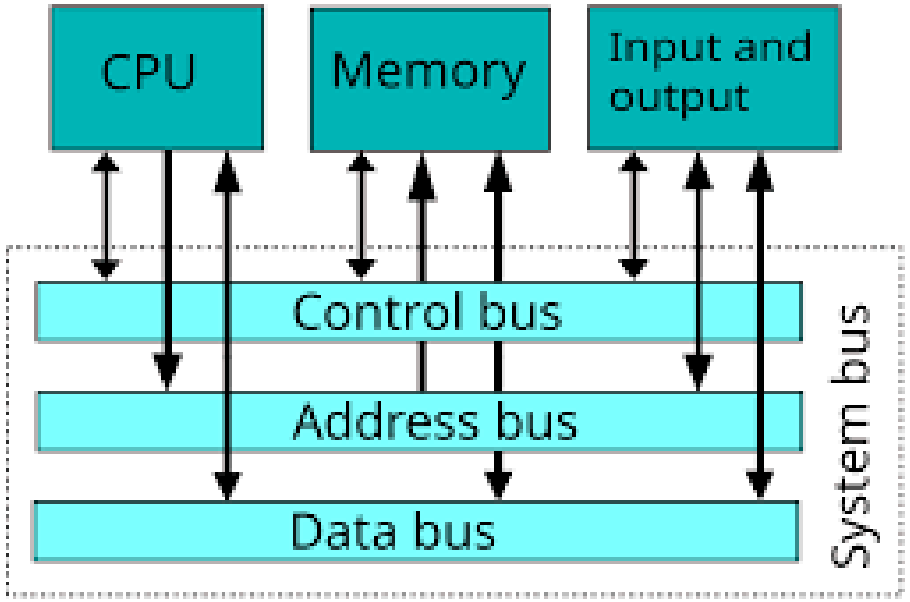
➢ **What is a BUS?**

📌 A **bus** is a communication pathway that connects different components of a computer system, such as the CPU, memory, and I/O devices. It consists of a set of parallel lines that carry **data, addresses, or control signals**.

Review of the Lecture 1-4

- *BUSes*

➢ **Three main types of BUSes**

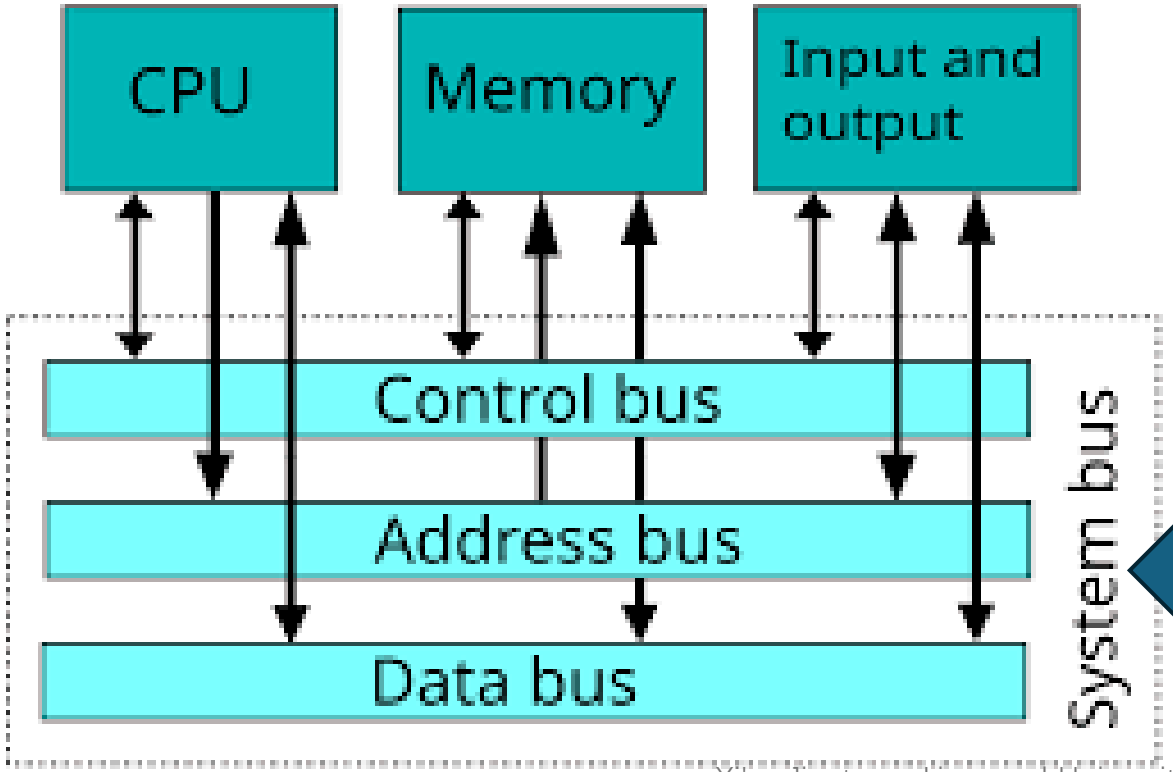


Bus Type	Description
🧭 **Address Bus**	Carries the **memory addresses** where data is located or needs to go. Usually **unidirectional (from CPU to memory).**
📦 **Data Bus**	Transfers **actual data** between the CPU, memory, and peripherals. Usually **bidirectional**.
🎮 **Control Bus**	Carries **control signals** (e.g., read/write signals, clock signals, interrupt requests). **Mostly unidirectional.**

Review of the Lecture 1-4

- *BUSes*

➢ **Exampe: How CPU reads Data from memory address `0x1000`**



1. CPU puts **`0x1000`** on the **address bus**

2. CPU sends a **READ** signal on the **control bus**

3. The memory module at that address sends the data back to the CPU via the **data bus**

*In practice, these three buses are often grouped together and called the **system bus**, which handles all communication between the CPU, memory, and I/O devices.*

Review of the Lecture 1-4

- *BUSes*

➢ **Key Feactures of BUSes**

Feature	Explanation
Parallel Transfer	A bus transfers multiple bits at once
Directionality	Some buses are unidirectional, others bidirectional
Bus Width	The number of bits in a bus determines how much data it can handle at once
Performance Impact	Wider and faster buses increase overall system performance

Review of the Lecture 1-4

- *BUSes*

➢ **Take-away information**

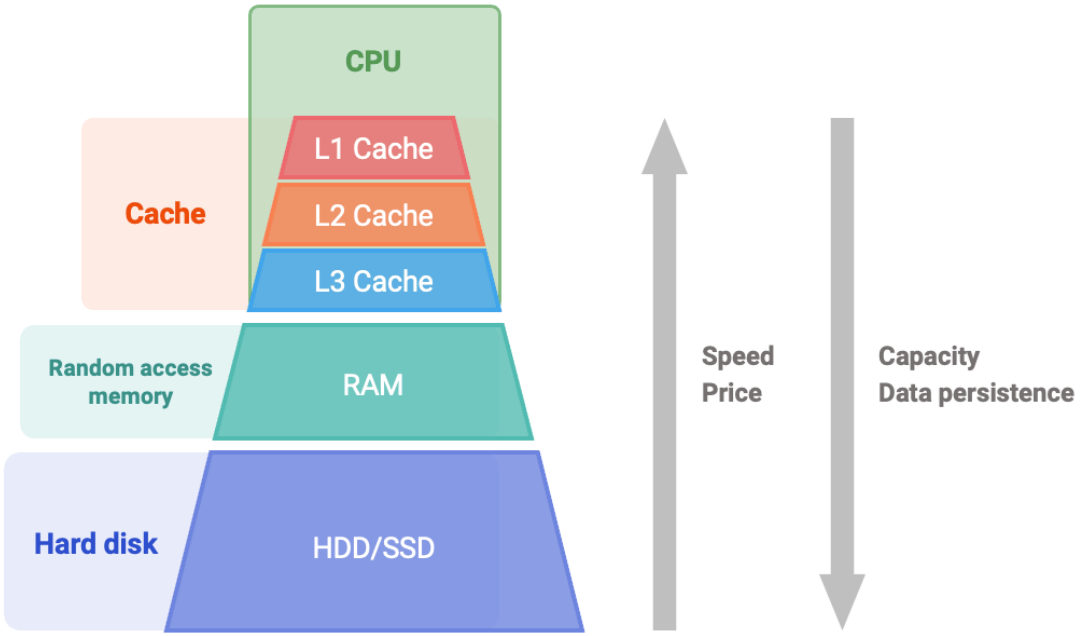
🎓 **"Address locates, data transfers, control directs."**

Don't confuse them!

Review of the Lecture 1-4

- *Cache and Memory*

➢ **CPU cache V.S. Memory**



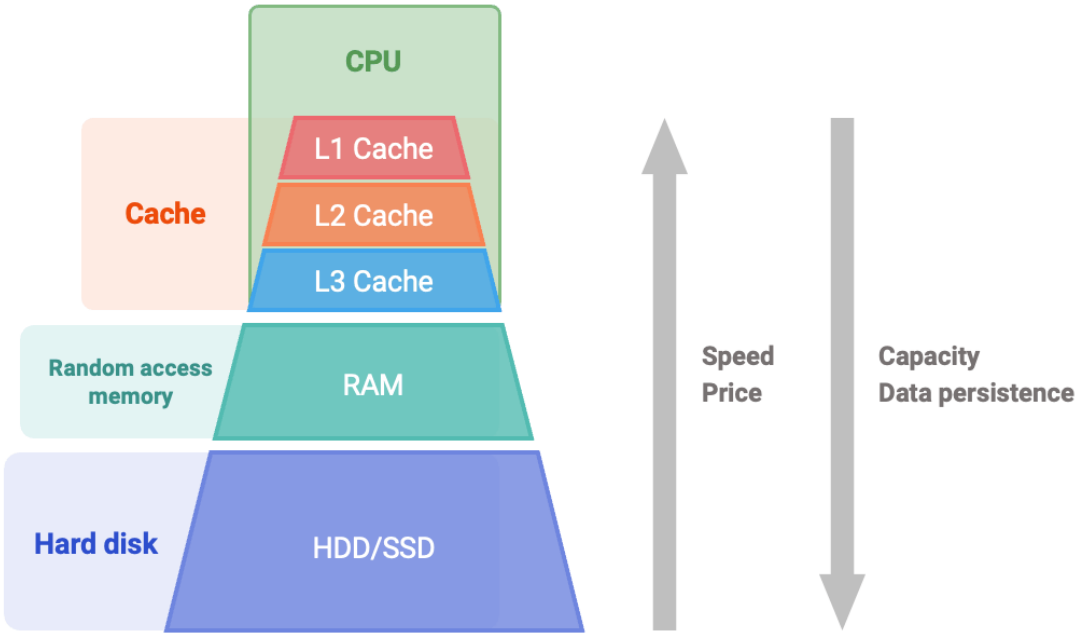
CPU Cache is a **small but extremely fast** type of memory located **inside or very close to the CPU**.
It temporarily stores frequently accessed data or instructions to speed up processing.

RAM (Random Access Memory) is the **main working memory** of the computer.
It stores **programs and data** that the CPU needs **while running**.

Review of the Lecture 1-4

- *Cache and Memory*

➢ **CPU cache V.S. Memory**



Feature	CPU Cache	RAM (main memory)
💨 Speed	Extremely fast	Fast (but slower than cache)
📏 Size	Small (KB to a few MB)	Large (GBs)
🧠 Location	Inside or near the CPU	On the motherboard
💰 Cost	Very high per byte	Moderate
🔋 Volatility	Volatile	Volatile
📦 Purpose	Temporary buffer for fast access	Stores all active programs/data

Review of the Lecture 1-4

- *Cache and Memory*

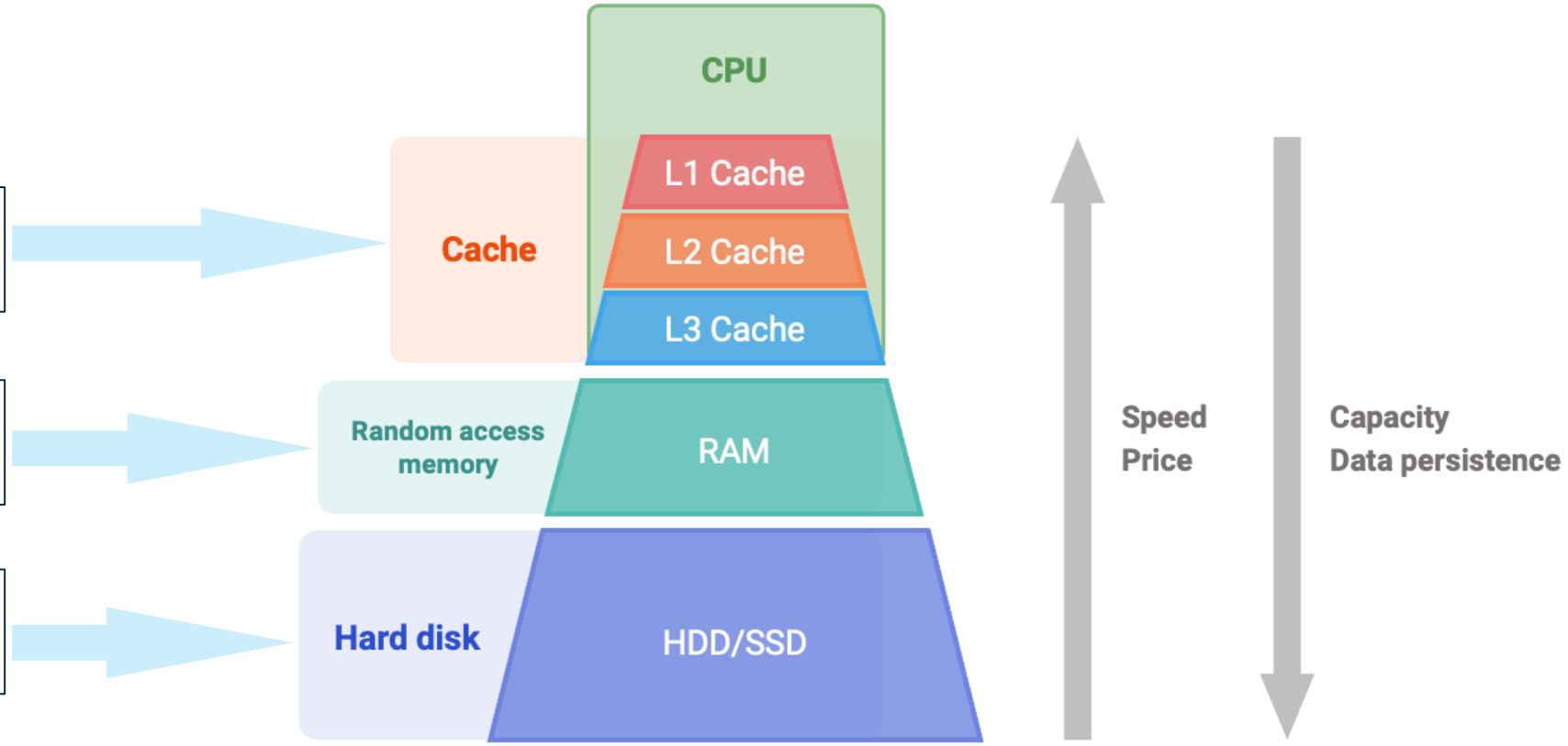
➢ **CPU cache V.S. Memory**

Analogy

Your brain's short-term memory (fast recall, small size)

Your working desk (holds what you're currently doing)

Your bookshelf (stores everything, but takes time to fetch)



Review of the Lecture 1-4

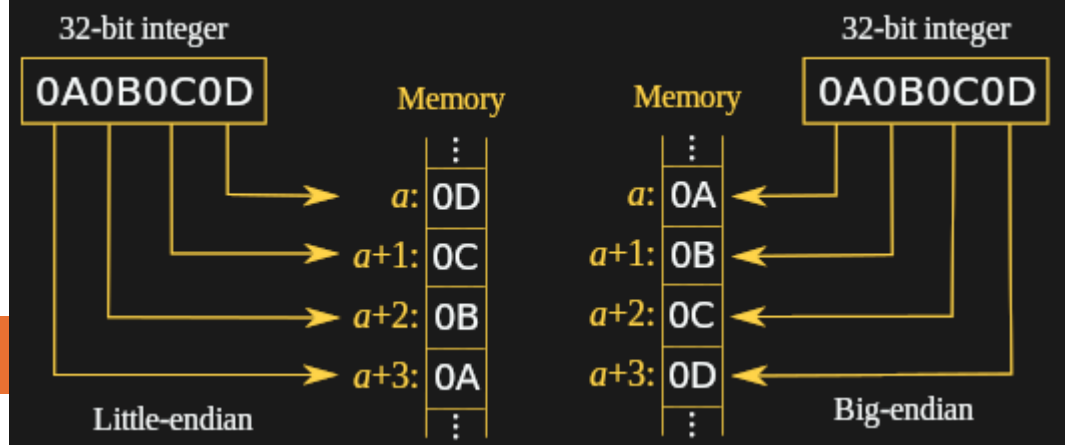
- *Cache and Memory*

➢ **Take-away information**

🎓 **"CPU cache is faster, smaller and more expensive than RAM."**

Review of the Lecture 1-4

- *Big and little Endianness*



➢ **Endianness**

Endianness refers to the **order in which bytes are stored in memory** for multi-byte data types like integers and floats.

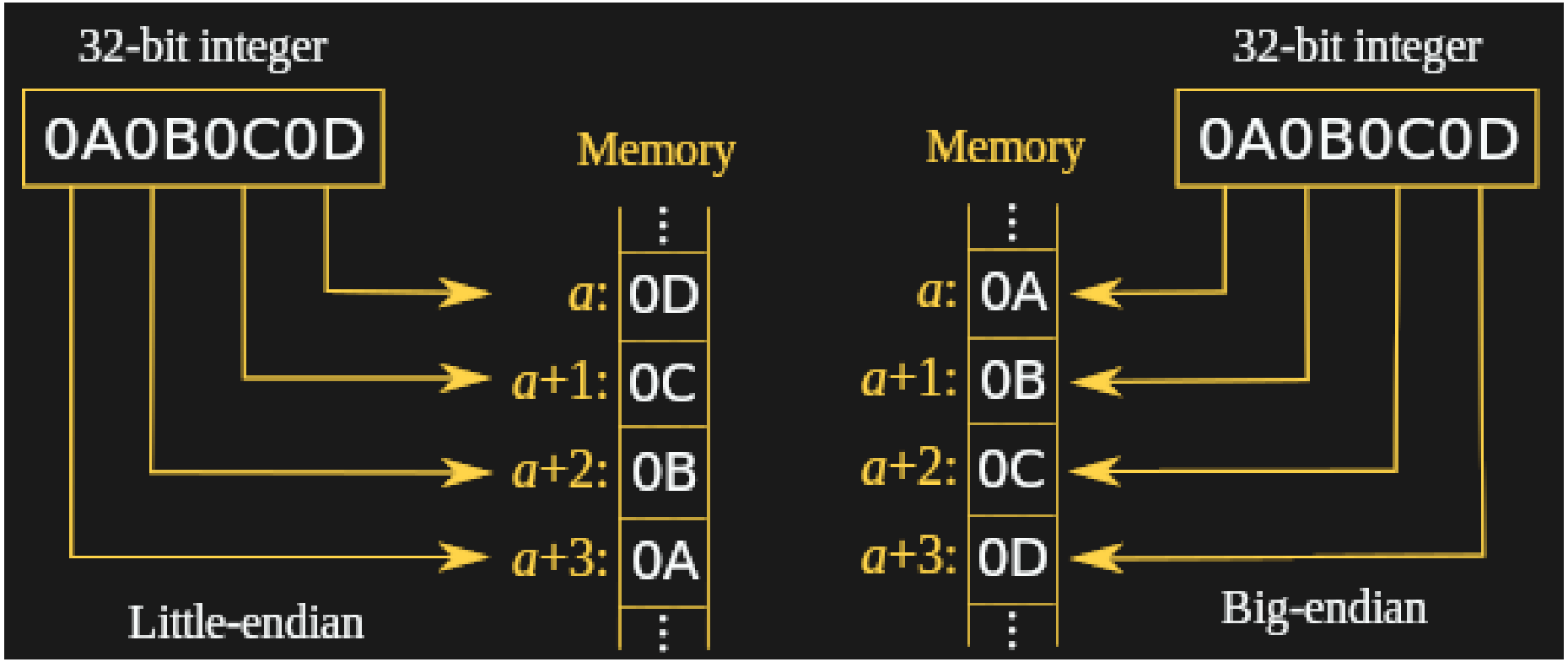
Big-endian: the **most significant byte (MSB)** is stored at the **lowest memory address;** storage order matches human reading order (left to right).

Little-Endian: the **least significant byte (LSB)** is stored at the **lowest memory address;** storage order is reversed from how we usually read numbers.

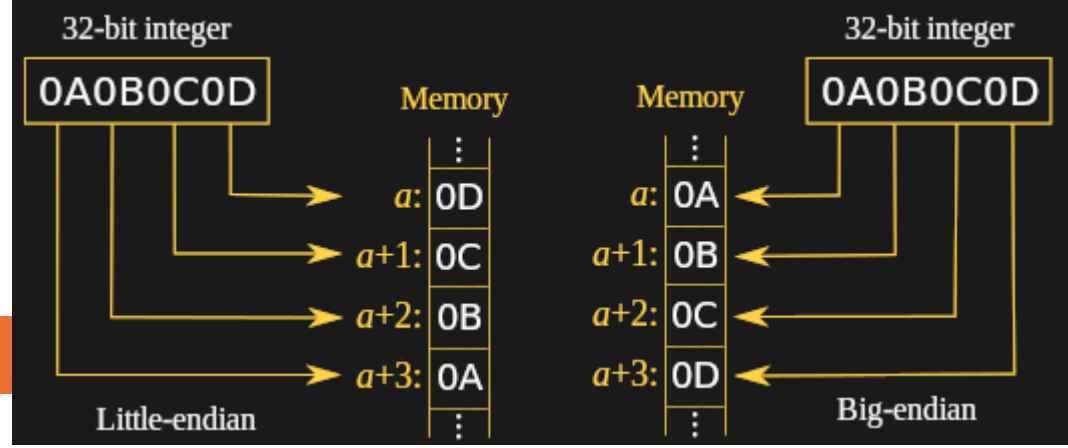
Review of the Lecture 1-4

- *Big and little Endianness*

➢ **Endianness**



Review of the Lecture 1-4



- *Big and little Endianness*

➢ **Endianness**

Aspect	Big-Endian	Little-Endian
Byte Order	MSB first	LSB first
Readability	Human-friendly	Needs reversing
Typical Architectures	SPARC, Motorola, Network protocols	x86, x86-64 (Intel/AMD)
Usage Example	Network transmission	Local memory, files

Why Important →

- *Cross-platform data exchange*
- *Debugging memory*
- *Affects how structures/unions are stored*
- *Networking uses big-endian as standard*

Review of the Lecture 1-4

- *Big and little Endianness*

➢ **Take-away information**

🎓 **"Big-endian MSB stores lowest, Little-endian LSB stores lowest."**

Review of the Lecture 1-4

- *Data Representation*

➢ **Base Conversion: Binary – Octal – Decimal - Hexadecimal**

Base	Prefix	Example
Binary (base = 2)	0b	0b1111 (=15)
Octal (base = 8)	0o/0	0o17 (=15)
Decimal (base = 10)	none	15
Hexadecimal (base = 16)	0x	0xF (=15)

Review of the Lecture 1-4

- *Data Representation*

➢ **Base Conversion: Binary – Octal – Decimal - Hexadecimal**

How to convert 13.125 in decimal into binary?

Integer part:
$13 \div 2 = 6 \cdots 1$
$6 \div 2 = 3 \cdots 0$
$3 \div 2 = 1 \cdots 1$
$1 \div 2 = 0 \cdots 1$

↑ 0b1101

Decimal part:
$0.125 \times 2 = 0.25$
$0.25 \times 2 = 0.5$
$0.5 \times 2 = 1.0$

↓ 0.001 in binary

13.125 in decimal = 1101.001 in binary

Review of the Lecture 1-4

- *Data Representation*

➢ **Base Conversion: Binary – Octal – Decimal - Hexadecimal**

Decimal	Binary	Octal	Hexadecimal
0	0000	00	0
1	0001	01	1
2	0010	02	2
3	0011	03	3
4	0100	04	4
5	0101	05	5
6	0110	06	6
7	0111	07	7
8	1000	10	8
9	1001	11	9
10	1010	12	A
11	1011	13	B
12	1100	14	C
13	1101	15	D
14	1110	16	E
15	1111	17	F

Review of the Lecture 1-4

- *Data Representation*

➢ **IEEE 754**

📌 IEEE 754 is the most widely used standard for representing **floating-point numbers** in binary in computers.

Why Important →

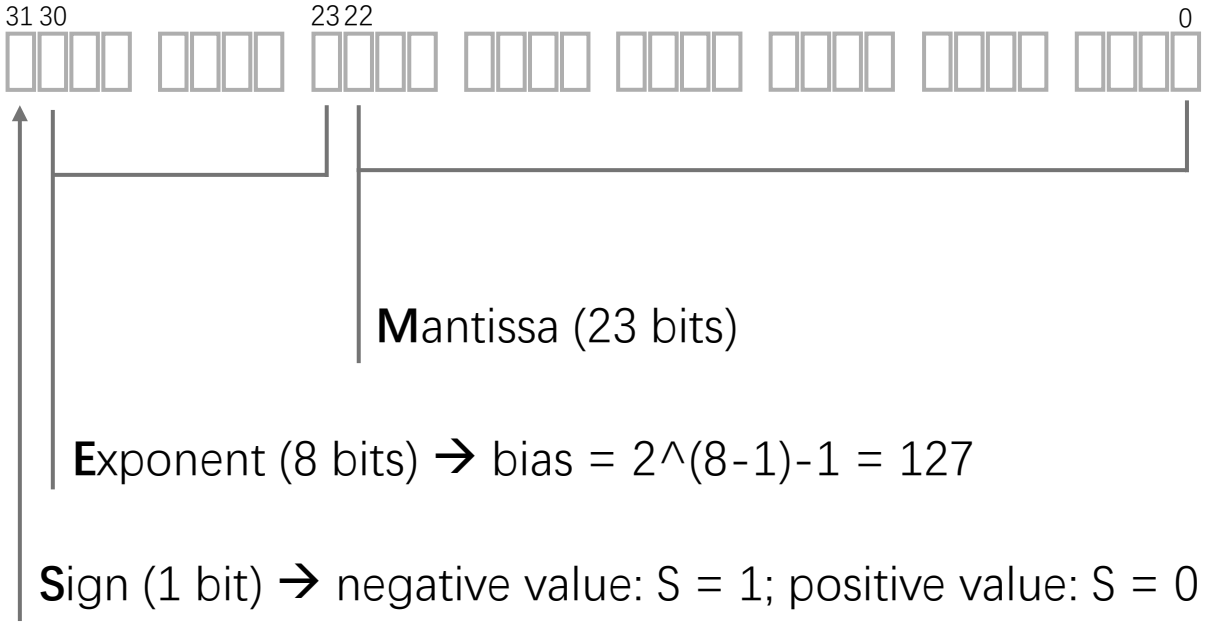
- *An universal standard for floating-point numbers*
- *Help to* understand why computers make rounding errors
- *How computers store real-world numbers using binary*

🧠 If you want to understand real-world numbers in computing, you need to understand IEEE 754.

Review of the Lecture 1-4

- *Data Representation*

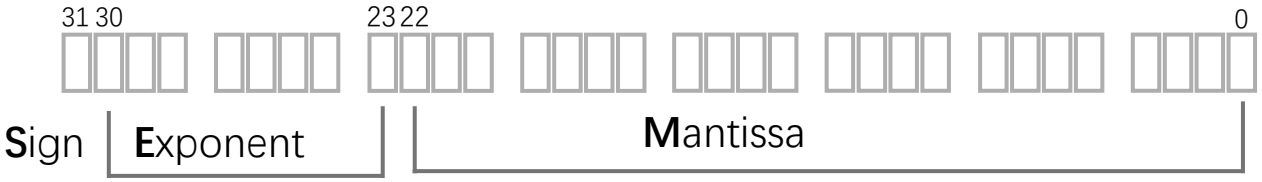
➢ **IEEE 754**



Review of the Lecture 1-4

- *Data Representation*

➢ **IEEE 754**



Example with $v = +13.125$ (1101.001 in binary):

- What the **S** is?
- What the **E** is?
- What the **M** is?
- What is the final representation in IEEE 754?

Break:
SURF Introduction

Lecture 5: learning target

📌 How ARM assembly code is translated into machine code and how different types of instructions are encoded and used.

- **ARM Assemblers**
  - What assembler is
  - Why use assembler
  - Differences between assembler and compiler
- **Instruction encoding**
  - Condition field
  - Immediate flag
  - Operation code
  - Status update flag
  - First operand register
  - Destination register
  - Second operand
- **Rotation and Shift**
  - ROR
  - LSR
  - ASR
  - LSL

Lecture 5

- *Assemblers*

➢ **What is Assembler?**

📌 An assembler is a tool that converts human-readable **assembly language** into **machine code** (binary instructions that the CPU can execute).

Lecture 5

- *Assemblers*

➢ **Why do we need and assembler?**

Reason	Explaination
CPUs **only understand binary machine code**	But writing 0s and 1s directly is too hard and error-prone.
Assembly is a **human-readable alternative**	Easier for programmers to write, debug, and understand.
Assemblers **automate translation** to exact opcodes and binary format	Ensures correctness and efficiency.
Assemblers allow the use of **labels, constants, macros, and directives**	These features simplify and organize code.
Enables writing **low-level, efficient code** close to hardware	Critical for embedded systems, performance tuning, etc.

Lecture 5

- *Assemblers*

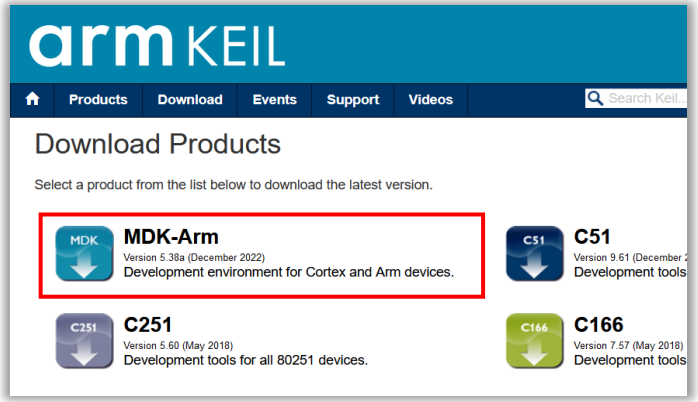
➢ **Common tools**

Tool Name	Description
Keil MDK	Includes official ARM assembler
GNU ARM Toolchain	Open-source ARM toolchain
VisUAL emulator	For simulation, not real execution (good for teaching)

Lecture 5

- *Assemblers*

- The content of this lecture can be verified using the Keil MDK: https://www.keil.com/download/product/



You can compile and run small programs for free.
Full version is quite expensive for personal use.

- A tutorial from past is also uploaded to LMO to guide you to setup the project.
  - Replace the **`main.c`** with an assembly file **`main.s`**
  - Make sure **`__main`** is defined and exported.

```
AREA        MY_PROG,CODE,READONLY
            EXPORT __main
__main
            …
```

Lecture 5

- *Assemblers*

- **GNU toolchain for ARM**: https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads.
- **ARM tools assembler**: https://developer.arm.com/downloads/-/arm-compiler-for-embedded
  - This assembler is included in Keil MDK 5: https://www2.keil.com/mdk5

Lecture 5

- *Assemblers*

➢ **Take-away information**

🎓 **Assembler turns readable instructions into real CPU actions.**

Lecture 5

- *Assemblers*

➢ **Any differences between Assembler and Compiler?**

Feature	Assembler	Compiler
🔤 **Input Language**	Assembly language (e.g., ARM, x86 assembly)	High-level language (e.g., C, C++, Java)
💾 **Output**	Machine code (binary) or object file	Assembly code or object file (then linked to binary)
🎛️ **Level of Control**	Very low-level; programmer controls every instruction	High-level; compiler handles optimizations and structure
🛠️ **Purpose**	Direct hardware manipulation, performance tuning	General-purpose programming, software development
🧠 **Abstraction Level**	Minimal abstraction; close to CPU instructions	High abstraction; programmer writes logic, not hardware
🧩 **Use Case Examples**	Embedded systems, device drivers, bootloaders	Applications, operating systems, web services
🔄 **Translation Process**	1-to-1 translation: each instruction → one machine code	Many-to-many: one line of code → many instructions
🗂️ **Symbols & Labels**	Supports labels, constants, macros (simplified)	Supports variables, functions, types, objects
🧪 **Debugging Granularity**	Very detailed, instruction-level	Abstracted, source-code level

Lecture 5

- *Assemblers*

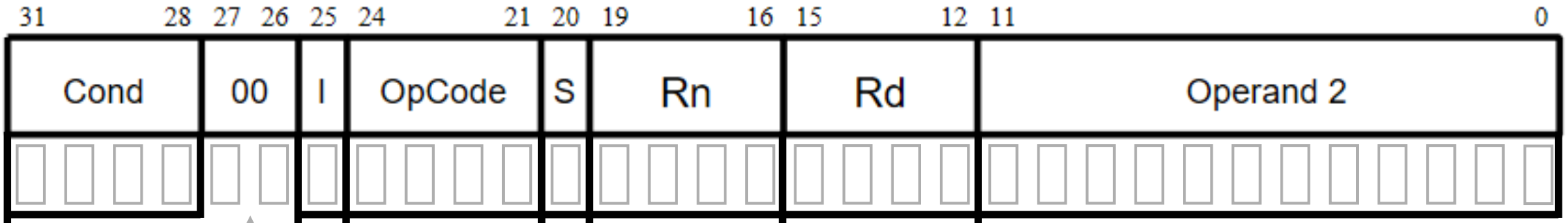
➢ **Take-away information**

🔧 **Assembler:**
→ Translates human-readable **assembly code** directly into **machine code**.
→ Used when precise control of hardware and performance is critical.

💻 **Compiler:**
→ Translates **high-level code** into assembly or machine code.
→ Optimizes code for readability, performance, and maintainability.

Lecture 5

- *Instruction Encoding*



Fixed bits: Always `00` for data processing instructions

Second operand: format depends on `I` bit

Destination register: Register to store the result of the operation

First operand register: Register of the first operand (source)

Status update flag: `1`=update condition flags (*NZCV* in **CPSR**); `0`=does not update

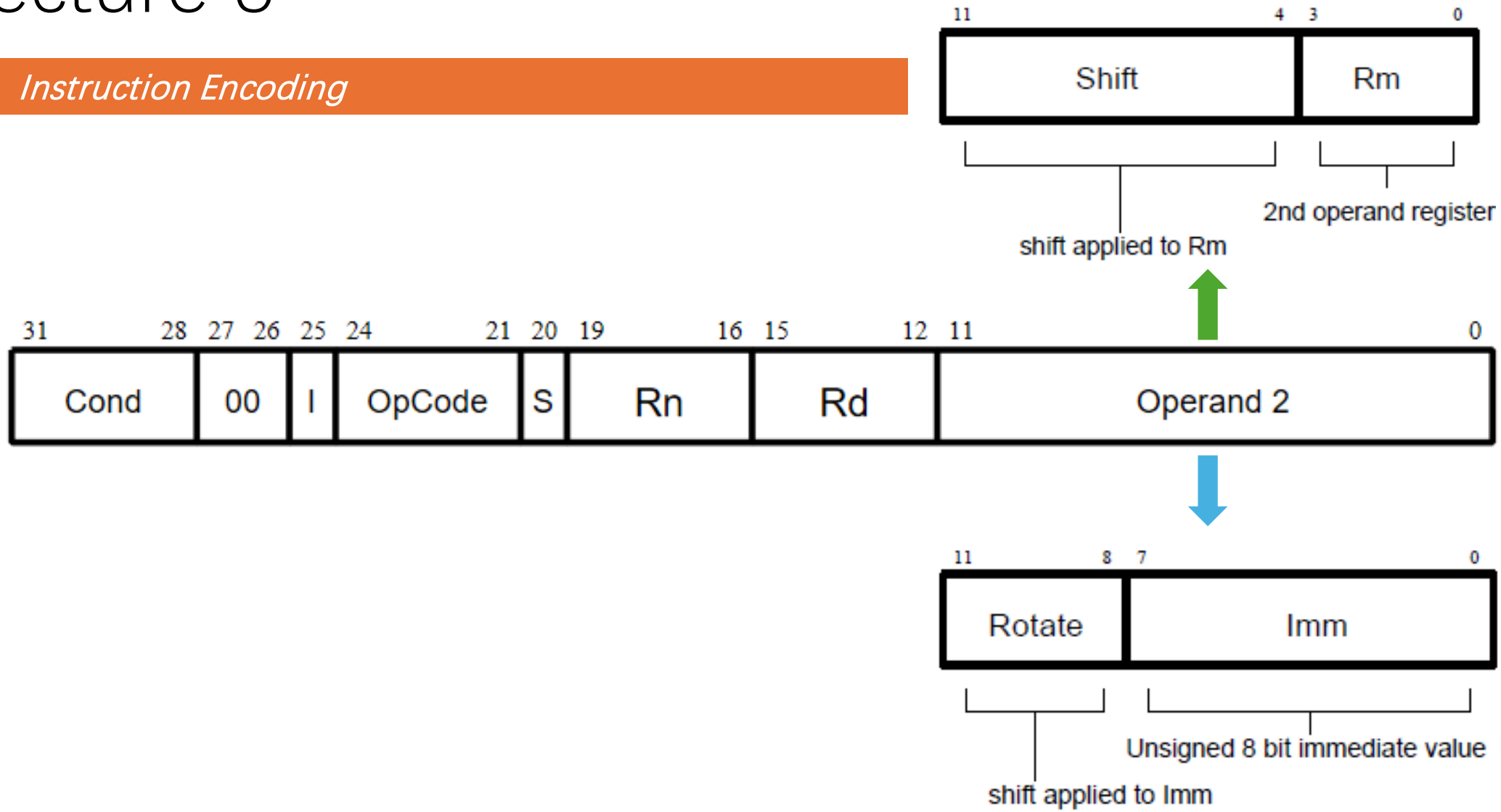
Operation code: Specifies the operation to perform (e.g., ADD, SUB, AND, MOV, etc.)

Immediate flag: `1`=`Operand 2` is an immediate value; `0`=`Operand 2` is a register + optional shift

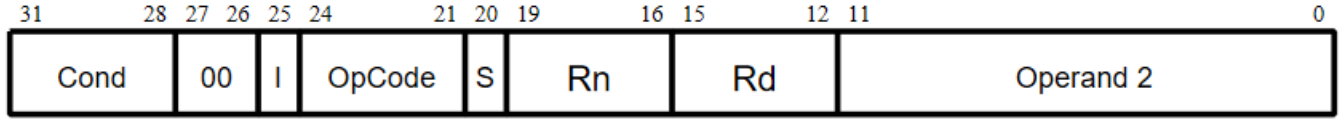
Condition field: Specifies whether the instruction executes based on condition flags (e.g., EQ, NE, AL)

Lecture 5

- *Instruction Encoding*



Lecture 5



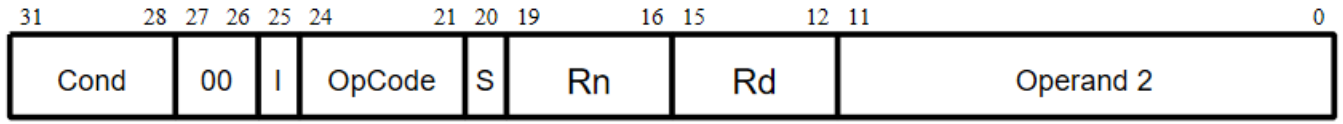
- *Instruction Encoding – Cond codes*

➢ **Condition field (Cond)**

Cond [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

Reference;
https://cas.ee.ic.ac.uk/people/gac1/Architecture/Lecture8.pdf

Lecture 5



- *Instruction Encoding – I flag*

➢ **Immediate flag (I)**

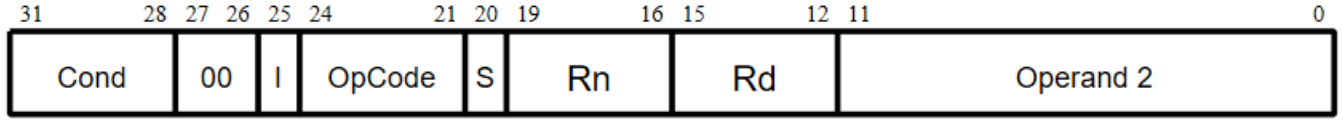
`I = 1: Operand 2` is an immediate value
`I = 0: Operand 2` is a register + optional shift

Example:

`ADD r1, r2, #1` → `1110 00 1 0100 0 0010 0001 0000 00000001`

`ADD r0, r1, r2` → `1110 00 0 0100 0 0001 0000 0000 00000010`

Lecture 5



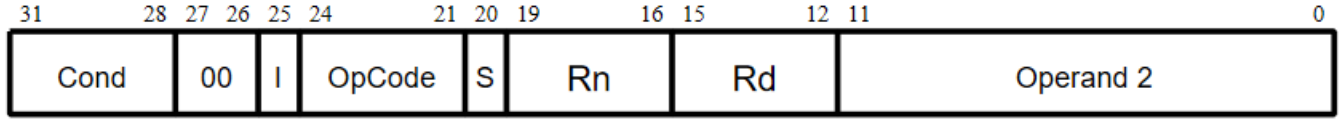
- *Instruction Encoding – OpCode codes*

➢ **Operation code**

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

Reference;
ARM7TDMI-S Instruction Set Encoding.pdf (uploaded on LM)

Lecture 5

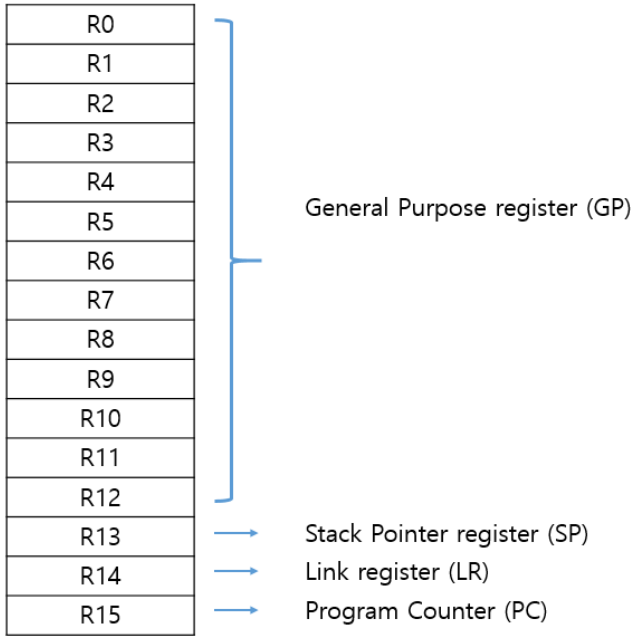


• *Instruction Encoding – S flag*

➢ **Set condition codes (S)**

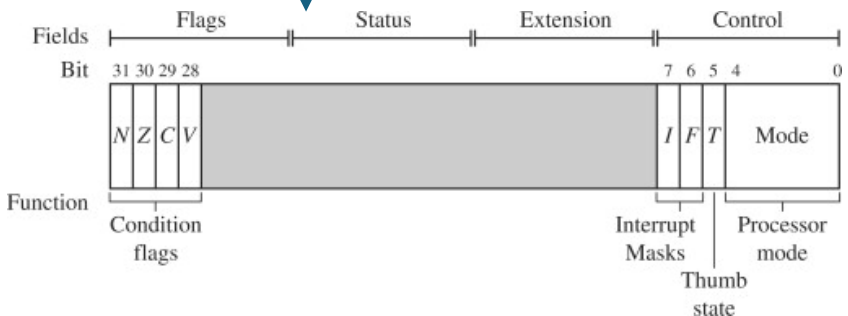
`S = 1:` update **the condition flags** stored in the **CPSR**
`S = 0:` do not alert

Condition flags	Meaning	When it is set to 1
N (Negative)	The result is negative	When the result's most significant bit is 1
Z (Zero)	The result is zero	When the result is exactly 0
C (Carry)	Carry out of an unsigned operation	In addition: if there's an extra bit In subtraction: if there's **no borrow**
V (Overflow)	Signed overflow occurred	When the result overflows the signed range (e.g. + + = −)

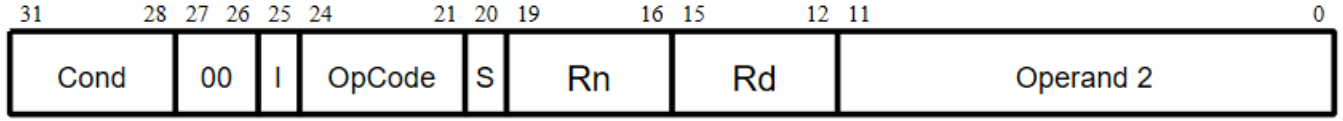


CPSR

Current Program Status Register



Lecture 5



• *Instruction Encoding – S flag*

Condition flags	Meaning	When it is set to 1
N (Negative)	The result is negative	When the result's most significant bit is 1
Z (Zero)	The result is zero	When the result is exactly 0
C (Carry)	Carry out of an unsigned operation	In addition: if there's an extra bit In subtraction: if there's **no borrow**
V (Overflow)	Signed overflow occurred	When the result overflows the signed range (e.g. + + = −)

Example:

positive

R1 = 0x7FFFFFFF = **0**111 1111 1111 1111 1111 1111 1111 1111

R2 = 0x1 = **0**000 0000 0000 0000 0000 0000 0000 0001

positive

ADD**S** R0, R1, R2

update S (= update NZCV)

---

negative

R1 + R2 = 0x80000000 = **1**000 0000 0000 0000 0000 0000 0000 0000

- Result is negative → N = 1
- Result is not zero → Z = 0
- No unsigned carry → C = 0
- Signed overflow → V = 1

Lecture 5

31–28	27–26	25	24–21	20	19–16	15–12	11–0
Cond	00	I	OpCode	S	Rn	Rd	Operand 2

• *Instruction Encoding – S flag*

Condition flags	Meaning	When it is set to 1
N (Negative)	The result is negative	When the result's most significant bit is 1
Z (Zero)	The result is zero	When the result is exactly 0
C (Carry)	Carry out of an unsigned operation	In addition: if there's an extra bit In subtraction: if there's **no borrow**
V (Overflow)	Signed overflow occurred	When the result overflows the signed range (e.g. + + = –)

➢ **When does overflow (V) happen?**

→ Only apply for calculation with **signed** values
→ Does not apply for unsigned calculation

	Cases	V = ?
Overflow	(+) + (+) = (-)	1
	(-) + (-) = (+)	1
	(+) - (-) = (-)	1
	(-) - (+) = (+)	1

Lecture 5

31–28	27–26	25	24–21	20	19–16	15–12	11–0
Cond	00	I	OpCode	S	Rn	Rd	Operand 2

- *Instruction Encoding – Rn & Rd*

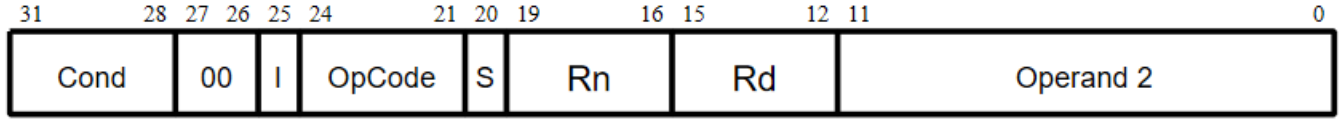
➢ **Registers**

`Rn = First operand register:` Register number of the first operand (**source**)
`Rd = Destination register:` Register number to store the **result** of the operation

Example:

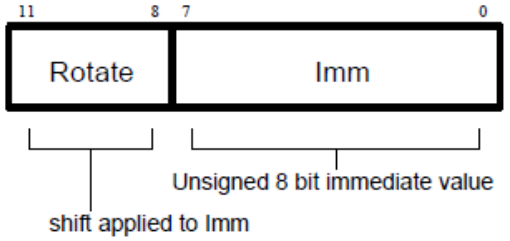
	Rd Rn		Rn Rd
`ADD r1, r2, #1`	→	`1110 00 1 0100 0 0010 0001 0000 00000001`	
`ADD r0, r1, r2`	→	`1110 00 0 0100 0 0001 0000 0000 00000010`	

Lecture 5



- *Instruction Encoding – Operand 2*

➢ **Second operand**



`I = 1: Operand 2` is a (rotate) immediate (rotate_imm X 2 + imm8)

Example:

```
ADDSLE r2, r1, #0x5F00
```

➡

```
1101 00 1 0100 1 0001 0010 1100 01011111
```

Lecture 5

- *Instruction Encoding – Operand 2*

➢ **Example**

Match the following machine code to the instructions below:

```
ADDSLE R2, R1, #0x5F00
```

	Cond	00	I	OpCode	S	Rn	Rd	Operand 2
1	1101	00	1	0100	1	0001	0010	1100 0101 1111
2	0000	00	0	0010	1	0001	0000	0011 0001 0010
3	1110	00	0	0011	1	1010	1001	0000 0000 1101

Lecture 5

ADDSLE R2, R1, #0x5F00

• *Instruction Encoding – Operand 2*

➢ **Example answer:**

- **Cond** is **LE** (code: **1101**); Fixed bits = **00**.
- **I** is **1**, the **Operand 2** will be an immediate number.
- **OpCode** is **ADD** (code: **0100**).
- **S** is **1**, so it will set flags.
- **Rn** is **R1** (code: **0001**) and **Rd** is **R2** (code: **0010**).
- **Operand 2**:

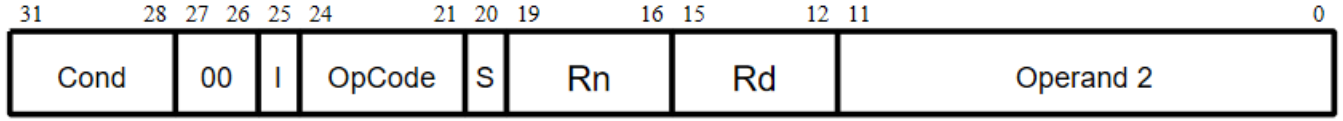
```
0x5F00 = 0000 0000 0000 0000 0011 1111 0000 0000
       = 0000 0000 0000 0000 0000 0000 0011 1111
```
rotated right 24 bits (ROR)

Rotate_imm = 24/2 = 12 bits, in binary = `1100; Imm = 0101 1111`

`Operand 2 = Rotate_imm + Imm = 1100 0101 1111`

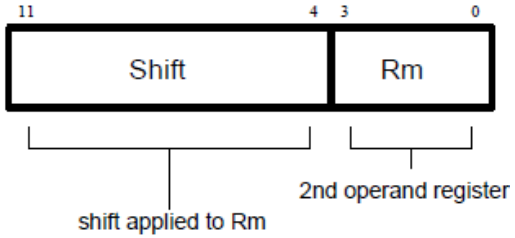
Cond	00	I	OpCode	S	Rn	Rd	Operand 2
1101	00	1	0100	1	0001	0010	1100 0101 1111

Lecture 5

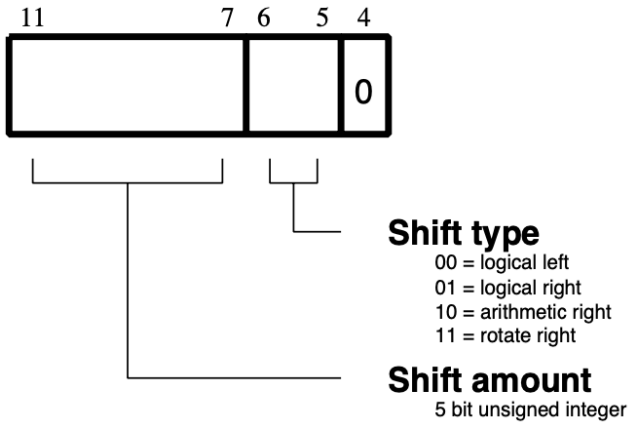


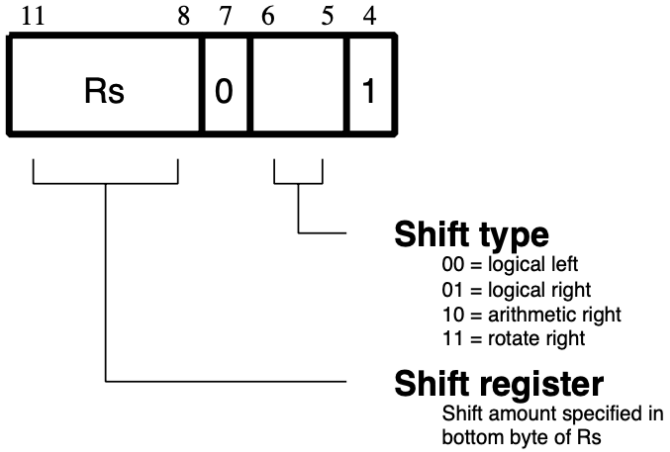
• *Instruction Encoding – Operand 2*

➢ **Second operand**



`I = 0: Operand 2` is a register + optional shift (shift8 + register4)





Example:

```
SUBSEQ r0, r1, r2, LSL r3
```

applied to r2

```
0000 00 0 0010 1 0001 0000 00110001 0010
```

Lecture 5

- *Instruction Encoding – Operand 2*

➢ **Example**

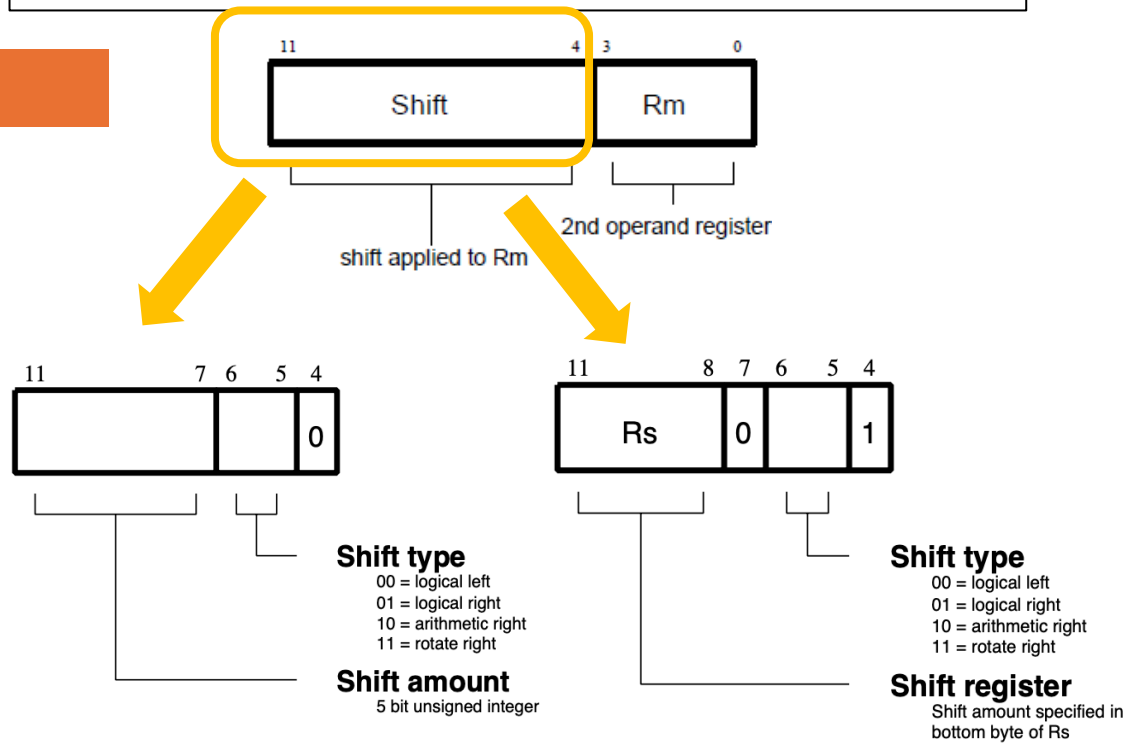
Match the following machine code to the instructions below:

```
SUBSEQ R0, R1, R2, LSL R3
```

	Cond	00	I	OpCode	S	Rn	Rd	Operand 2
1	1101	00	1	0100	1	0001	0010	1100 0101 1111
2	0000	00	0	0010	1	0001	0000	0011 0001 0010
3	1110	00	0	0011	1	1010	1001	0000 0000 1101

Lecture 5

SUBSEQ R0, R1, R2, **LSL** R3



• *Instruction Encoding – Operand 2*

➢ **Example answer:**

- **Cond** is **EQ** (code: **0000**); Fixed bits = **00**.
- **I** is **0**, the **Operand 2** will be a register.
- **OpCode** is **SUB** (code: **0010**).
- **S** is **1**, so it will set flags.
- **Rn** is **R1** (code: **0001**) and **Rd** is **R0** (code: **0000**)
- **Operand 2**:

```
Second operand register (Rm)is R2 (code: 0010)
Shift register: the 4th bit = 1 and the 7th bit = 0, relevant register is R3 (code: 0011)
Shift type is LSL (code: 00)
Operand 2 = Shift register + 7th + Shift type + 4th + Second operand register
          = 0011 0 00 1 0010 = 0011 0001 0010
```

Cond	00	I	OpCode	S	Rn	Rd	Operand 2
0000	00	0	0010	1	0001	0000	0011 0001 0010

Lecture 5

- *Instruction Encoding – Operand 2*

➢ **Question:**

What instruction the following machine code represents?

Cond	00	I	OpCode	S	Rn	Rd	Operand 2
1110	00	0	0011	1	1010	1001	0000 0000 1101

🧠 Try it yourself ☺

Lecture 5

- *Rotation and Shift*

➢ **Definition and example**

Type	Name	What it does	Example
LSL	Logical Shift Left	Shifts bits left, fills 0s	0010 << 1 = 0100
LSR	Logical Shift Right	Shifts bits right, fills 0s	1000 >> 2 = 0010
ASR	Arithmetic Shift Right	Shifts right, fills with sign bit	1000 >> 2 = 1110 (if signed)
ROR	Rotate Right	Rotates bits around to front	1001 ROR 1 = 1100

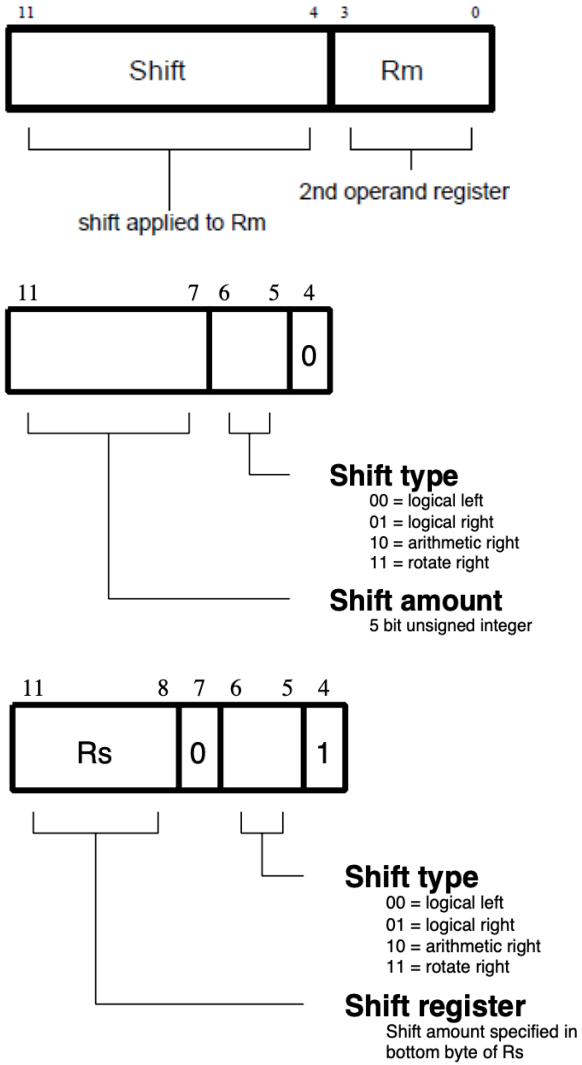
Lecture 5

- *Rotation and Shift*

➢ **Binary Encoding (in Operand 2)**

Type	Meaning	Bits 6-5
LSL	Logical Shift Left	00
LSR	Logical Shift Right	01
ASR	Arithmetic Shift Right	10
ROR	Rotate Right	11

- Bits 11-7: shift amout (0-31)

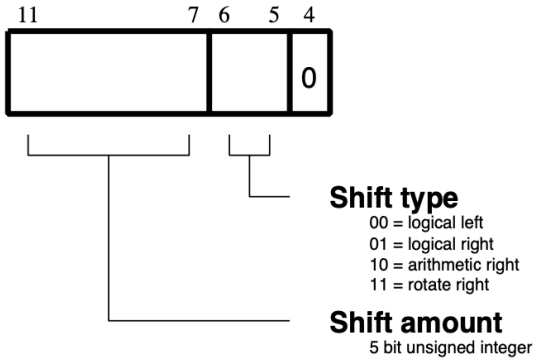


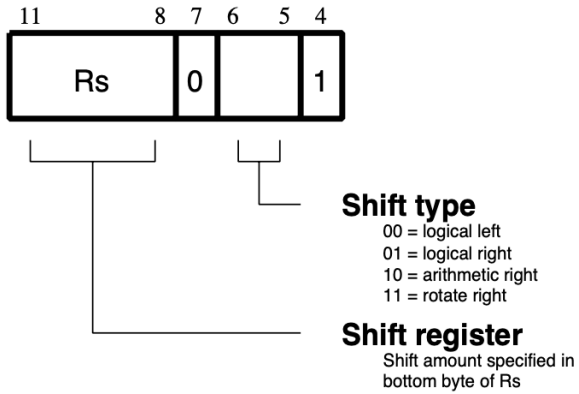
Lecture 5

- *Rotation and Shift*

➢ **Binary Encoding (in Operand 2)**

- Bit 4 = 0:
  - shift amout is immediate: 5-bit unsigned integer
- Bit 4 = 1:
  - shift amout is in a register: shift amout secified in bottom byte of Rs





- When I = 1 (immediate):
  - Oeprand 2 = {rotate_imm (4 bits), imm8 (8 bits)}
  - The real value = ROR(imm8, rotate_imm x 2)

✅ Allows many 32-bit constants to be encoded with just 12 bits!

Lecture 5

- *Rotation and Shift*

➢ **Take-away information**

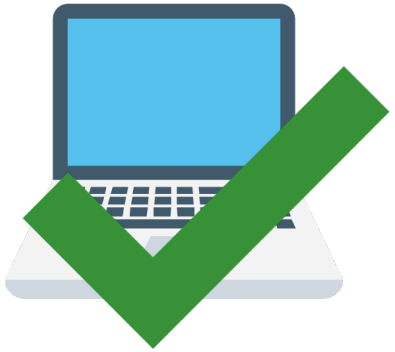
🎓 **LSL and LSR fill with 0s, ASR fills with sign, ROR moves bits around.**

Before End:
Assessment Explaination

Assessment 1&2

- Assessment 1 & 2 will be combined into one and take **30% of the module mark**
- **On-site** online assessment that happens during Tutorial → **You MUST come to the tutorial in person**
- You need to complete this assignment individually
- Photography/AI agencies are not allowed


- Questions are exacted from lectures and labs (about 6 questions)
- **Scheduled during a specific time of your tutorial sessions** (about 40 minutes)
- **Bring your own LAPTOPS with FULL batteries (not tablet, not cell phone)**
- **Missing assessment = missing 30% of the module mark**
- **NO RESIT OF ASSESSMENT**




Formal assessment on Week 13

Assessment trial-run on Week 12

We are here

Week 2	Week 3	Week 4	Week 5	Week 6	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13
Lecture 1	Lecture 2	Lecture 3	Lecture 4	Lab 1	Lab 2	Lecture 5	Lecture 6	Lecture 7	Lecture 8	Tutorial 2 Formal Assessment
									Tutorial 1 Assessment Trial-run	

Assessment 1&2 will be combined into one and should be completed at **a specific time in the Tutorial**

Assessment 1 (CW 1) 15%	Assessment 2 (CW 2) 15%
Final Exam 70%	

Come in person and don't miss!!