Xi'an Jiaotong-Liverpool University

西交利物浦大学

CPT210 - Microprocessor

Lecture 6 – ARM Functions and Stacks-Part 1

Overview

- Lecture 5 Review
  - ARM Assembler
  - Instruction Encoding
  - Rotation

- Lecture 6
  - LDR/STR
    - Notation
    - Instruction encoding
    - Offset
  - ADR
    - Definition
    - Syntax
    - How it works
  - LDR Pseudo-instruction

Before End: Final Exam & Assessment Explaination

Review of the Lecture 5

- *Assemblers*

➢ **Take-away information**

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

Review of the Lecture 5

- *Assemblers*

➢ **Take-away information**

🔧 **Assembler:**

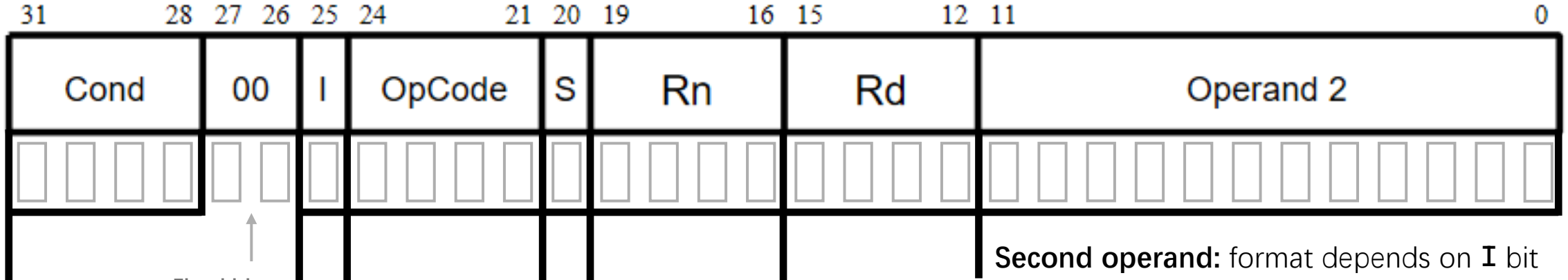
→ Translates human-readable **assembly code** directly into **machine code**.
→ Used when precise control of hardware and performance is critical.

💻 **Compiler:**

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

Review of the 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 **CSPR**); `0`=does not update

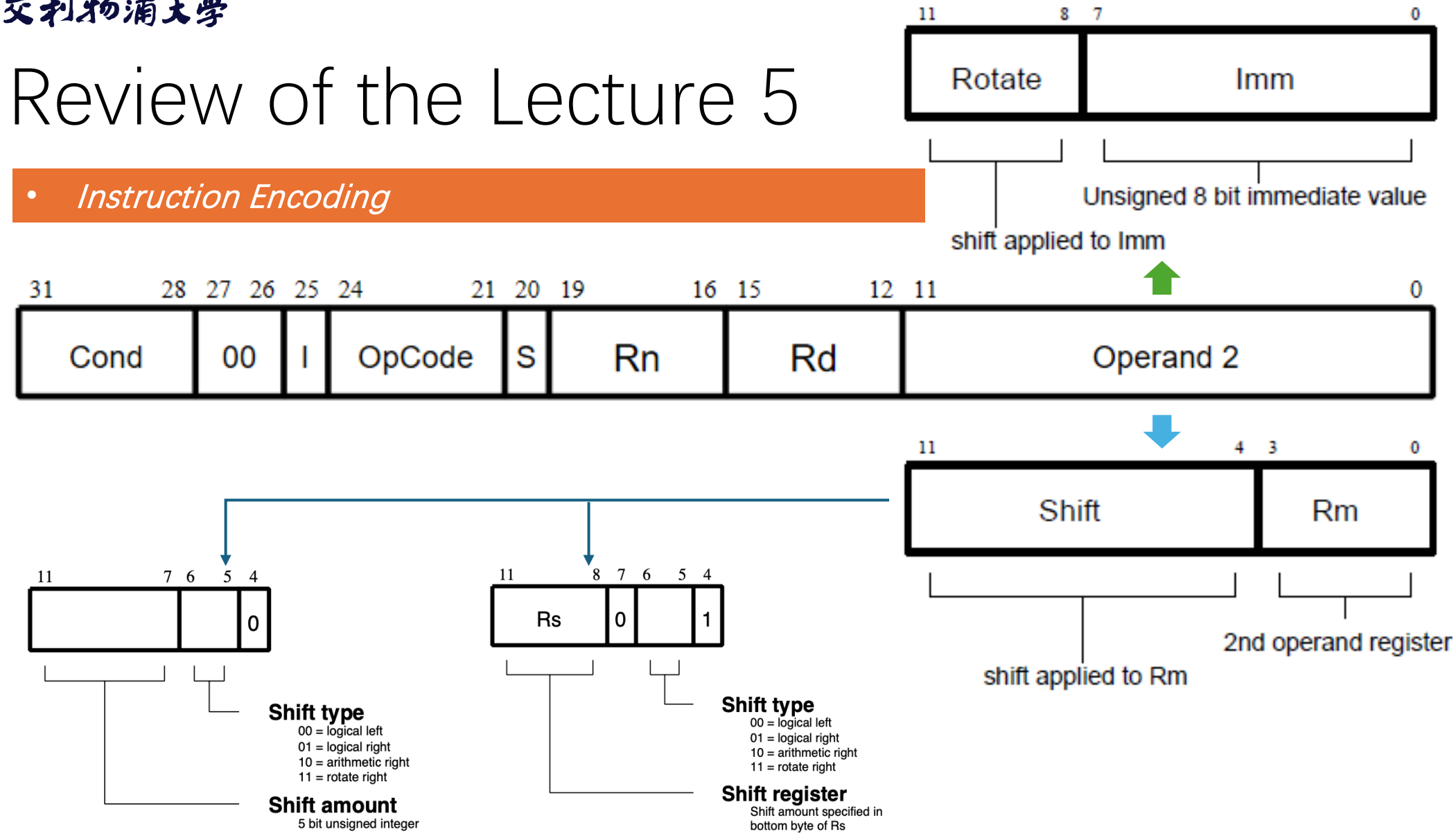
Operation code: Specifies the operation to perform (e.g., ADD, SUB, AND, MOV, etc.)

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)

Review of the Lecture 5

- *Instruction Encoding*



Review of the Lecture 5

- *Rotation and Shift*

➢ **Take-away information**

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

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

Lecture 6

- *LDR/STR*

➢ **What are LDR and STR?**

Instrcution	Meaning	Basic Function	Syntax
LDR	Load Register	Load a value from memory into a register	`LDR Rt, [Rn, #offset]`
STR	Store Register	Store the value from a register into memory	`STR Rt, [Rn, #offset]`

Lecture 6

- *LDR/STR*

➢ **Why do we learn LDR and STR?** 🎓 **Because ARM is a Load/Store Architecture**

ARM follows a **Load/Store architecture**, which means:
👉 **All computations must happen in registers**,
👉 **Data in memory cannot be used directly in arithmetic or logic operations.**

Lecture 6

- *LDR/STR*

➢ **How to understand LDR and STR?**

Rt:
The target register to read from or write to

#offset:
Optional, the offset value (can be positive, negative, zero, or omitted)

```
LDR Rt, [Rn, #offset]
```
*Load memory → **Rt***

```
STR Rt, [Rn, #offset]
```
*Store **Rt** → memory*

Rn:
The base register (usually points to the starting memory address)

Lecture 6

- *LDR/STR*

➢ **Examples**

```
LDR R0, [R1, #4]
```

◆ **Meaning:**

Load a 4-byte word from the memory address at **R1+4** and store the value into register **R0**

If **R1** = **0x1000**:

- Take the address stored in **R1**, which is **0x1000**
- Add **#4** (which is 4 bytes) → result: **0x1004**
- Read a **word** starting from address **0x1004**
- Store the loaded word value into register **R0**

Lecture 6

- *LDR/STR*

➢ **Try it yourselves**

`R1 = 0x1008, R0 = ?`

```
LDR R0, [R1]
```

```
LDR R0, [R1, #8]
```

```
LDR R0, [R1, #-8]
```

Address	Stored Value
0x1000	0xAABBCCDD
0x1008	0x11223344
0x1010	0xDEADBEEF

Lecture 6

- *LDR/STR*

➢ **Examples**

```
STR R2, [R3]
```

🔸 **Meaning:**

Store the value in register **R2**
into the memory address pointed to by **R3**

one address, one byte

If
R2 = `0xCAFEBABE`
R3 = `0x2000`

- Value in register **R2** is `0xCAFEBABE`
- The memory address pointed to by **R3** is `0x2000`
- Store the value `0xCAFEBABE` at `0x2000`

Address	Byte (little-endian)
0x2000	0xBE
0x2001	0xBA
0x2002	0xFE
0x2003	0xCA

✅ *ARM uses **little-endian** format by default, so the lowest byte goes to the lowest address.*

Lecture 6

- *LDR/STR*

➢ **Try it yourselves**

`R3 = 0x1008, R2 = 0xCAFEBABE`

```
STR R2, [R3]
```

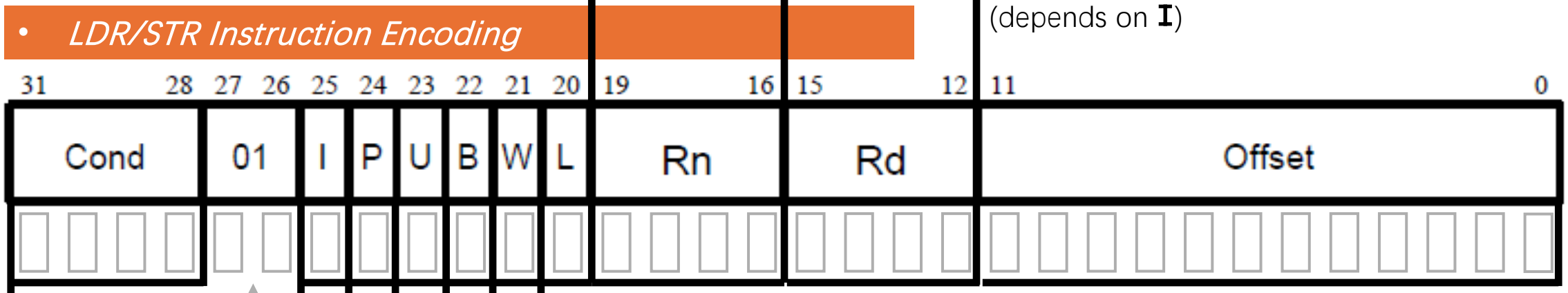
```
STR R2, [R3, #8]
```

```
STR R2, [R3, #-8]
```

Address	Original Stored Value	After overwritting
0x1008	0xAABBCCDD	0xCAFEBABE
0x1010	0x11223344	0xCAFEBABE
0x1000	0xDEADBEEF	0xCAFEBABE

Lecture 6

- *LDR/STR Instruction Encoding*



Base register: Register holding the base memory address

Destination/source register: Where data is stored or loaded from

Offset: 12-bit offset or shifted register value (depends on **`I`**)

Instruction class: Always **`01`** for Load/Store instructions

**different from data processing instructions

Load/Store: **`1`**=Load (LDR); **`0`**=Store (STR)

Write-back bit: **`1`**=Write result back to base register transfer; **`0`**=Don't write back

Byte/Word bit: **`1`**=Byte transfer; **`0`**=Word transfer

Up/Down bit: **`1`**=Add offset to base; **`0`**=Subtract offset from base

Pre/Post indexing: **`1`**=Pre-indexing (add offset before transfer; **`0`**=Post-indexing (after)

Immediate bit: **`0`**=**`offset`** is a 12-bit immediate value; **`1`**=**`offset`** is a register

**opposite of the data processing instructions

Condition field: Executes only if condition is met

Lecture 6

- *LDR/STR nstruction Encoding*

➢ **Pre/Post indexing (P)**

`P = 1:` pre-indexing, apply the **offset before** the memory access

```
LDR R0, [R1, #4]    Pre-index, no write-back
LDR R0, [R1, #4]!   Pre-index + write-back
```

The memory address = **`R1 + 4`**
If you use **`!`**, the result (new memory address) is written back to R1

➢ **Example**

```
R1 = 0x1000
LDR R0, [R1, #8]!
```

- Effective address = **`0x1000 + 8 = 0x1008`**
- Load memory at **`0x1008`** into **`R0`**
- Then update **`R1 = 0x1008`** (because of !)

Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Pre/Post indexing (P)**

`P = 0:` post-indexing, apply the **offset after** the memory access

`LDR R0, [R1], #4` Post-index (write-back implied)

First access memory at **R1**
Then update **R1 = R1 + 4**

➢ **Example**

```
R1 = 0x1000
LDR R0, [R1], #8
```

- Load memort from address **0x1000** into **R0**
- Then update **R1 = 0x1008** (write-back implied)

Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Pre/Post indexing (P)**

Code	Result
`LDR R0, [R1, #4]`	`R0 ← [R1+4], R1 unchanged`
`LDR R0, [R1, #4]!`	`R0 ← [R1+4], R1 ← R1+4`
`LDR R0, [R1], #4`	`R0 ← [R1], R1 ← R1+4`

Code	Result
`STR R0, [R1, #4]`	`Store R0 to [R1+4], R1 remains unchanged`
`STR R0, [R1, #4]!`	`Store R0 to [R1+4], then update R1 ← R1 + 4`
`STR R0, [R1], #4`	`Store R0 to [R1], then update R1 ← R1 + 4`

Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Byte/Word bit (B)**

📦 **What's a "Word" in ARM?**

- ARM that we teach is a **32-bit architecture**, so a **word = 4 bytes = 32 bits**
- Word access is **aligned** to addresses divisible by 4 (e.g., **`0x1000, 0x1004`**...)

Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Byte/Word bit (B)**

Instruction	Transfer Type	Size	Description
LDR	Word (default)	4 bytes	Load 32-bit word from memory
STR	Word (default)	4 bytes	Store 32-bit word to memory
LDRB	Byte	1 byte	Load 8-bit byte from memory
STRB	Byte	1 byte	Store 8-bit byte to memory

◆ The **"B" suffix** = **Byte access**

Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Byte/Word bit (B)**

`B = 0:` Transfer a **word** (32 bits)

➢ **Example**

```
LDR R0, [R1]
```

- Loads all **4 bytes** starting from **0x1000**
- In little-endian: **0xDDCCBBAA** → stored in **R0**

```
R1 = 0x1000
Memory[0x1000] = 0xAA
Memory[0x1001] = 0xBB
Memory[0x1002] = 0xCC
Memory[0x1003] = 0xDD
```

Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Byte/Word bit (B)**

`B = 0:` Transfer a **word** (32 bits)

➢ **Example**

```
STR R0, [R1]
```

```
R0 = 0xAABBCCDD
R1 = 0x2000
```

- Stores all **4 bytes** starting from **R0**
- To the memory address pointed to by **R1**

```
Memory[0x2000] = 0xDD
Memory[0x2001] = 0xCC
Memory[0x2002] = 0xBB
Memory[0x2003] = 0xAA
```

Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Byte/Word bit (B)**

`B = 1:` Transfer a **byte** (8 bits)

➢ **Example**

```
LDRB R0, [R1]
```

```
R1 = 0x1000
Memory[0x1000] = 0xAA
Memory[0x1001] = 0xBB
Memory[0x1002] = 0xCC
Memory[0x1003] = 0xDD
```

- Only loads **1 byte** at **0x1000** → **0xAA**
- The other 24 bits in **R0** are cleared

Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Byte/Word bit (B)**

`B = 1:` Transfer a **byte** (8 bits)

➢ **Example**

```
STRB R0, [R1]
```

- Store only the lowest 8 bits (1 byte) of **R0**
- To the memory address pointed to by **R1**

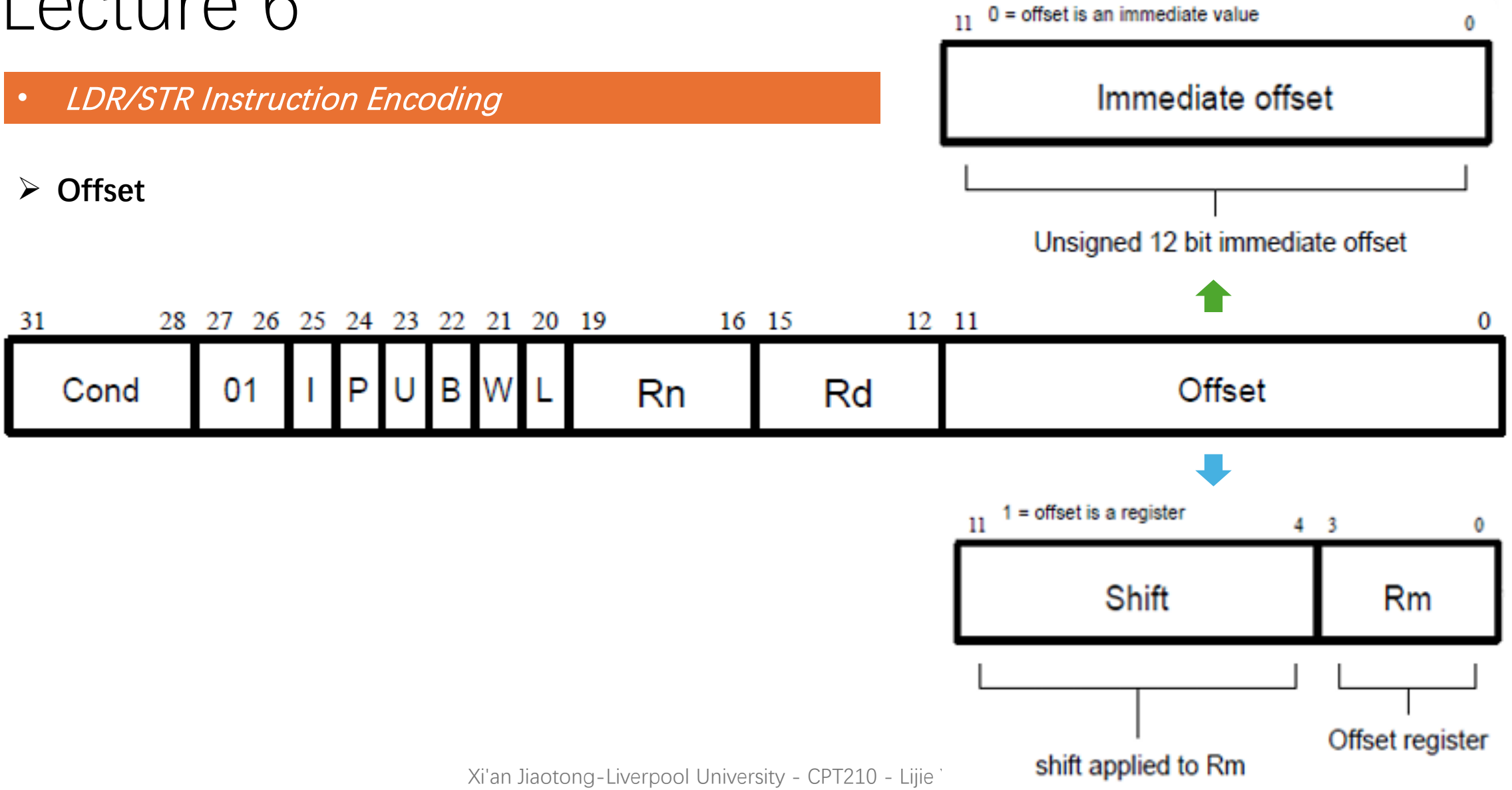
```
R0 = 0xAABBCCDD
R1 = 0x2000
```

```
Memory[0x2000] = 0xDD
The rest of R0 is ignored
```

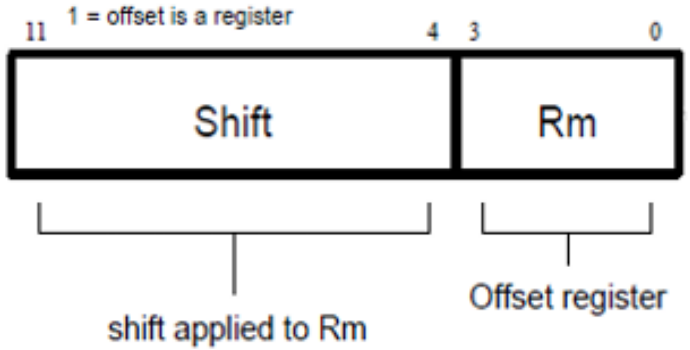
Lecture 6

- *LDR/STR Instruction Encoding*

➢ **Offset**



Lecture 6



- *LDR/STR Instruction Encoding*

➢ **Offset**

- `LDR` and `STR` support adding/subtracting offsets stored in another register:

```
LDR/STR{cond}   Rt, [Rn, ±Rm{, shift}]
LDR/STR{cond}   Rt, [Rn, ±Rm{, shift}]!
LDR/STR{cond}   Rt, [Rn], ±Rm{, shift}
```

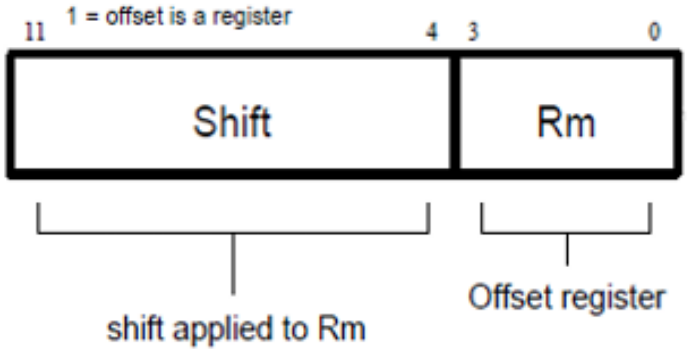
Symbols + and – are not supported in VisUAL.
They can be used in Keil IDE

➢ **Examples**

```
LDR r0, [r1, r2, LSL #2]
LDRB r0, [r1, r2, LSR #1]!
LDR r0, [r1], r2, LSL #3
```

Lecture 6

- *LDR/STR Instruction Encoding*



➢ **Examples**

```
LDR r0, [r1, r2, LSL #2]
```

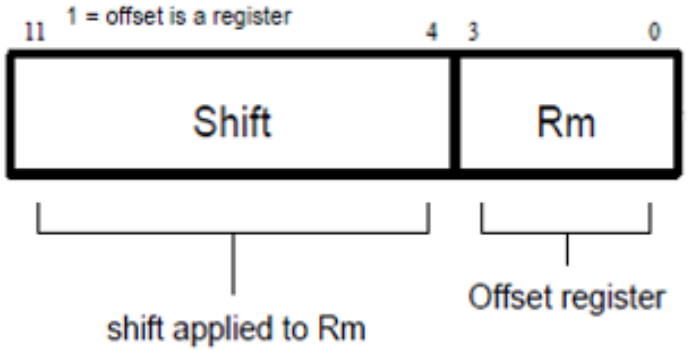
🔹 **Type: Pre-indexed with register-based shifted offset**

🔍 **Meaning:**

- **Base address** is in **R1**
- **Offset** = **R2** shifted left by 2 bits → **R2** × **4**
- **Effective address** = **R1** + (**R2** << **2**)
- Result is loaded into **R0**
- **R1** is **not** updated

Lecture 6

- *LDR/STR Instruction Encoding*



➢ **Examples**

```
LDRB r0, [r1, r2, LSR #1]!
```

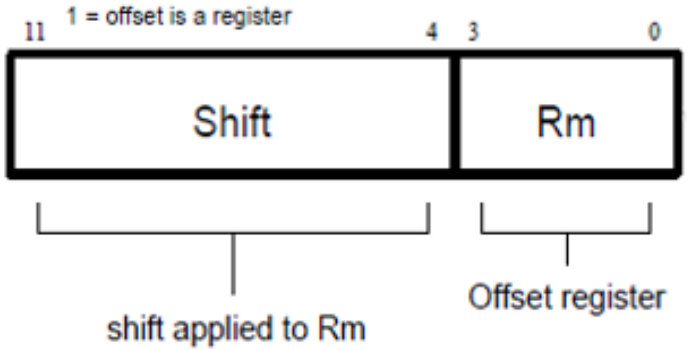
🔹 **Type: Pre-indexed with write-back, byte transfer**

🔍 **Meaning:**

- Load **1 byte** (because of LDR**B**)
- **Base address** is in **R1**
- **Offset** = **R2** shifted right by 1 bit → **R2 ÷ 2**
- **Effective address** = **R1 + (R2 >> 1)**
- Load byte from that address into **R0**
- Then store the new address back into **R1** (because of **!**)

Lecture 6

- *LDR/STR Instruction Encoding*



➢ **Examples**

```
LDR r0, [r1], r2, LSL #3
```

🔹 **Type: Post-indexed with register-based shifted offset**

🔍 **Meaning:**

- Load 4 bytes (word) from address in **R1** into **R0**
- After loading, update **R1 = R1 + (R2 << 3)** → **R2 × 8**
- The shift happens **after** the access (post-indexing)

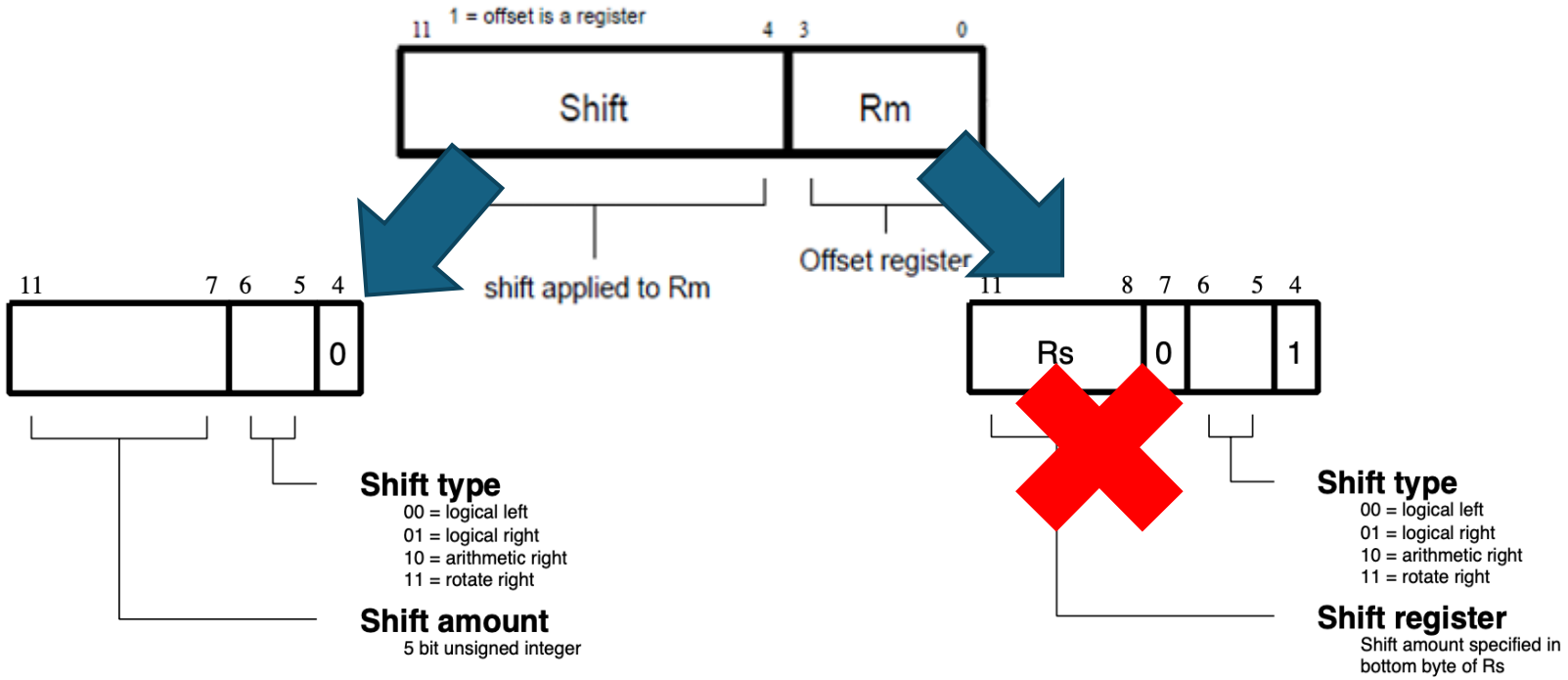
Lecture 6

- *LDR/STR Instruction Encoding*

Register-specified shift is **not** supported in LDR and STR.

Thus, you cannot write:

```
LDR r0, [r1, r2, LSL r3]
```



Lecture 6

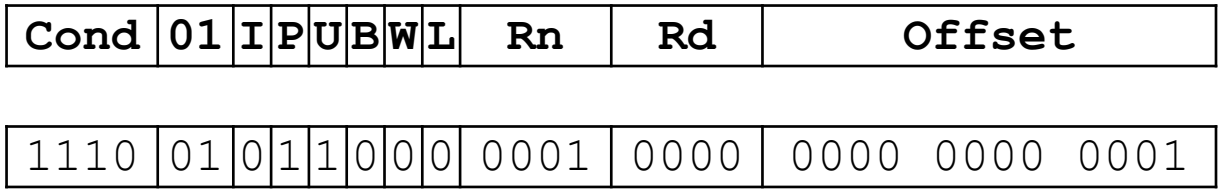
- *LDR/STR Instruction Encoding*

➢ **Examples**

What do the following machine code do?

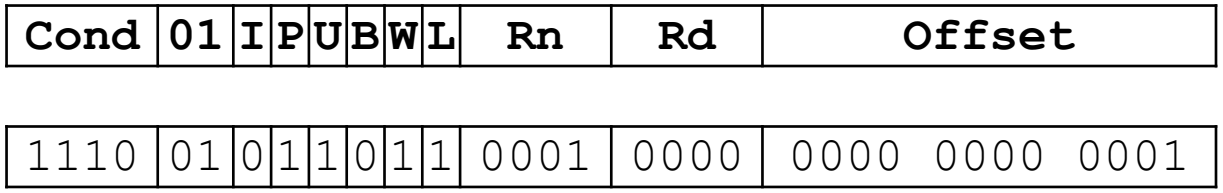
	Cond	01	I	P	U	B	W	L	Rn	Rd	Offset
1	1110	01	0	1	1	0	0	0	0001	0000	0000 0000 0001
2	1110	01	0	1	1	0	1	1	0001	0000	0000 0000 0001
3	1110	01	1	0	1	1	0	1	0001	0000	0011 0010 0010
4	1110	01	0	1	0	0	1	1	0001	0000	0000 0000 0001

Answers



Step	Instruction
`L` is 0	`STR` …
`B` is 0, transfer a whole word.	`STR` …
`Cond` is 0b1110, which means "always".	`STR` …
`P` is 1, pre-indexing	`STR Rd, [Rn, ??]`
`Rn` is 0b0001, which means `R1`.	`STR Rd, [R1, ??]`
`Rd` is 0b0000, which means `R0`.	`STR R0, [R1, ??]`
`W` is 0, do not update the base address (`Rn`)	`STR R0, [R1, ??]`
`I` is 0, the offset will be an immediate number.	`STR R0, [R1, #?]`
`Offset` is 1.	`STR R0, [R1, #1]`
`U` is 1, adding offset to base (offset is a positive immediate).	`STR R0, [R1, #1]`

Answers



Step	Instruction
`L` is 1.	`LDR …`
`B` is 0, transfer a whole word.	`LDR …`
`Cond` is 0b1110, which means "always".	`LDR …`
`P` is 1, pre-indexing	`LDR Rd, [Rn, ??]`
`Rn` is 0b0001, which means `R1`.	`LDR Rd, [R1, ??]`
`Rd` is 0b0000, which means `R0`.	`LDR R0, [R1, ??]`
`W` is 1, update the base address (`Rn`)	`LDR R0, [R1, ??]!`
`I` is 0, the offset will be an immediate number.	`LDR R0, [R1, #??]!`
`Offset` is 1.	`LDR R0, [R1, #1]!`
`U` is 1, adding offset to base (offset is a positive immediate).	`LDR R0, [R1, #1]!`

Answers

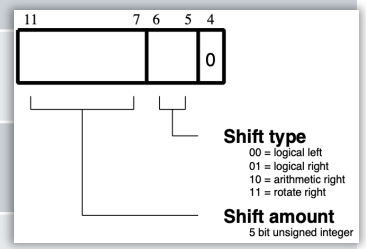
Step	Instruction
`L` is 1.	`LDR …`
`B` is 0, transfer a whole word.	`LDR …`
`Cond` is 0b1110, which means "always".	`LDR …`
`P` is 1, pre-indexing	`LDR Rd, [Rn, ??]`
`Rn` is 0b0001, which means `R1`.	`LDR Rd, [R1, ??]`
`Rd` is 0b0000, which means `R0`.	`LDR R0, [R1, ??]`
`W` is 1, update the base address (`Rn`)	`LDR R0, [R1, ??]!`
`I` is 0, the offset will be an immediate number.	`LDR R0, [R1, #??]!`
`Offset` is 1.	`LDR R0, [R1, #1]!`
`U` is 0, subtracting offset to base (offset is a negative immediate).	`LDR R0, [R1, #-1]!`

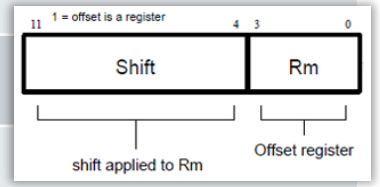
Cond	01	I	P	U	B	W	L	Rn	Rd	Offset

1110	01	1	0	1	1	0	1	0001	0000	0011 0010 0010

Answers

Step	Instruction
`L` is 1.	`LDR` …
`B` is 1, transfer a byte.	`LDRB` …
`Cond` is 0b1110, which means "always".	`LDRB` …
`P` is 0, post-indexing	`LDRB Rd, [Rn], ??`
`Rn` is 0b0001, which means `R1`.	`LDRB Rd, [R1], ??`
`Rd` is 0b0000, which means `R0`.	`LDRB R0, [R1], ??`
`W` is 0, do not update the base address. (ignored as Post-indexing always updates the base address)	`LDRB R0, [R1], ??`
`I` is 1, the offset will be a register.	`LDRB R0, [R1], Rm, ?????`
`Offset` register is 0b0010 (`R2`), shift type is LSR (01), shift amount is 0b00110 (=6)	`LDRB R0, [R1], R2, LSR #6`
`U` is 1, adding offset to base (offset is a positive immediate).	`LDRB R0, [R1], R2, LSR #6`

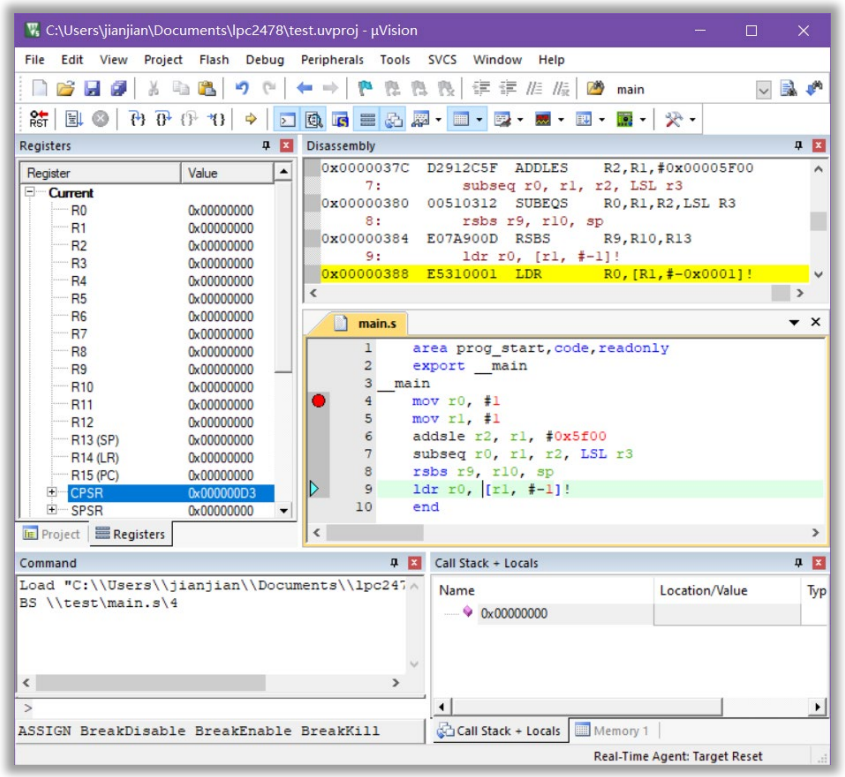




If U is 0: `LDRB R0, [R1], -R2, LSR #6`
This syntax is only available on Keil.

More Examples

- The encodings on the right are obtained in Keil IDE.
- VisUAL might give errors.



str r0, [r1, #1]	0xE5810001
str r0, [r1, #1]!	0xE5A10001
str r0, [r1], #1	0xE4810001
ldr r0, [r1, #1]	0xE5910001
ldr r0, [r1, #1]!	0xE5B10001
ldr r0, [r1], #1	0xE4910001

str r0, [r1, r2, LSL #1]	0xE7810082
strb r0, [r1, r2, LSR #2]!	0xE7E10122
str r0, [r1], r2, ASR #3	0xE68101C2
ldrb r0, [r1, r2, ROR #4]	0xE7D10262
ldr r0, [r1, r2, LSL #5]!	0xE7B10282
ldrb r0, [r1], r2, LSR #6	0xE6D10322

Lecture 6

- *ADR*

➢ **What is a label in ARM?**

In assembly language, a **label** is a **name you give to a memory location**.

It can refer to:

- A block of **data** (e.g., constants, arrays)
- A **line of code** (e.g., function entry or a jump target)

```
num1    DCD 0x1, 0x2, 0x3, 0x4
```

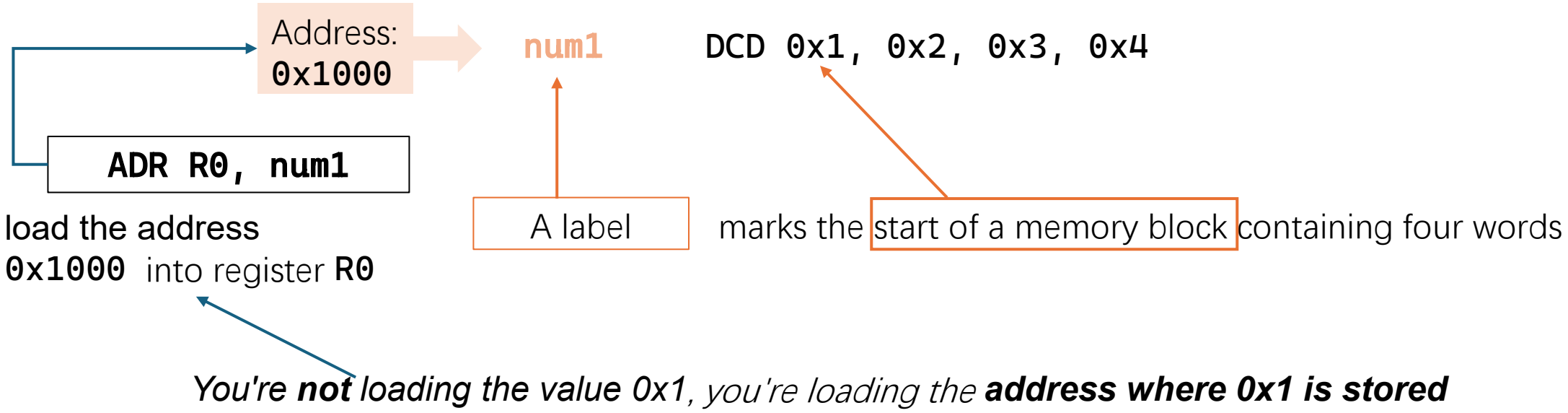
A label marks the start of a memory block containing four words

Lecture 6

- *ADR*

➢ **What is ADR?**

ADR stands for **Address Register**.
It is an instruction to load the **address of a label** (memory location) into a register.



Lecture 6

- *ADR*

➢ **Syntax**

```
ADR Rd, label
```

- **Rd:** Destination register
- **label:** A named memory location (usually defined in .data or .rodata section)

🔍 What does ADR actually do?

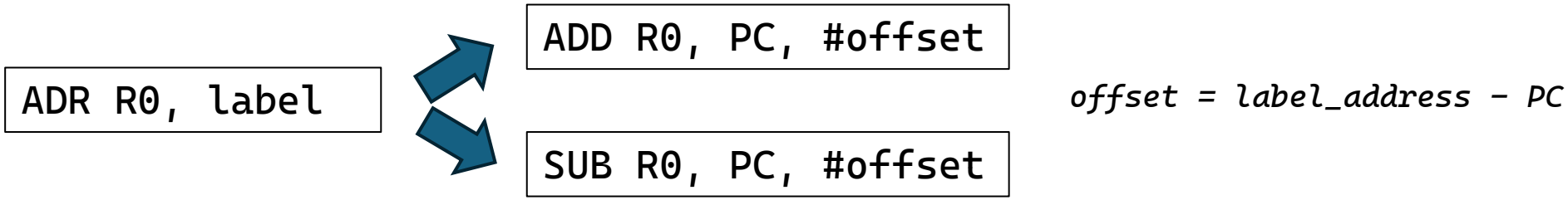
- It does **not** load the value at the label,
  but rather loads **the address of the label** into the register.

Lecture 6

- *ADR*

➢ **How does ADR work interbally?**

- **ADR** is implemented as either an **ADD** or **SUB** instruction under the hood.



- It uses **PC-relative addressing** (because it adds/subtracts an offset to/from the PC)
- The assembler calculates the distance between current **PC** and **`label`** and encodes it

A basic rule in ARM:
PC = current instruction address + 0x00000008

Lecture 6

- *ADR*

Symbols will be converted into PC-relative values by assemblers

➢ **Example**

```
        AREA    MY_PROG,CODE,READONLY
        EXPORT  __main
num1    DCD     0x1, 0x2, 0x3, 0x4
__main
        ADR     R6, num1
        ADR     R7, num2
        ADD     R5, R6, R7
        ADD     R5, R5, #0x45
num2    DCD     0x7, 0x8, 0x9, 0xA
        END
```

Memory address

value

```
0x00000550  00000001  ANDEQ      R0,R0,R1
0x00000554  00000002  ANDEQ      R0,R0,R2
0x00000558  00000003  ANDEQ      R0,R0,R3
0x0000055C  00000004  ANDEQ      R0,R0,R4
     5:         ADR     R6, num1
0x00000560  E24F6018  SUB        R6,PC,#0x00000018
     6:         ADR     R7, num2
0x00000564  E28F7004  ADD        R7,PC,#0x00000004
     7:         ADD           R5, R6, R7
0x00000568  E0865007  ADD        R5,R6,R7
     8:         ADD           R5, R5, #0x45
0x0000056C  E2855045  ADD        R5,R5,#0x00000045
0x00000570  00000007  ANDEQ      R0,R0,R7
0x00000574  00000008  ANDEQ      R0,R0,R8
0x00000578  00000009  ANDEQ      R0,R0,R9
0x0000057C  0000000A  ANDEQ      R0,R0,R10
```

If converted into instruction

Lecture 6

- *ADR*

➢ **Try by yourselves**

What do the following instructions mean?

Instruction	Meaning
`ADR R0, label`	?
`LDR R0, =label`	?

Lecture 6

- *LDR Pseudo-Instruction*

```
LDR R0, =0xAABBCCDD
```

pseudo-instruction

- ARM instructions are **32-bit wide**
- But you want to load a **full 32-bit constant**
- There isn't enough room in one instruction to include a full 32-bit number

Lecture 6

- *LDR Pseudo-Instruction*

```
LDR R0, =0xAABBCCDD
```

pseudo-instruction

🔧 **What does the assembler actually do?**

1. It places the constant (0xAABBCCDD) somewhere **near your instruction**, in a region called the **literal pool**
2. It then converts the pseudo-instruction into a real instruction like:

```
LDR R0, [PC, #offset]
```

This loads the value from **memory near the PC**, not directly from the instruction.

Lecture 6

- *LDR Pseudo-Instruction*

LDR =something is a pseudo-instruction

➢ **Example**

```
        AREA prog,CODE,READONLY
        EXPORT __main
num1    DCD  0x1, 0x2, 0x3, 0x4
__main
        LDR  R0, =num1
        LDR  R1, =0xaabbccdd
        END
```

Memory address	Value	If converted into instruction	
0x00000374	00000001	ANDEQ	R0,R0,R1
0x00000378	00000002	ANDEQ	R0,R0,R2
0x0000037C	00000003	ANDEQ	R0,R0,R3
0x00000380	00000004	ANDEQ	R0,R0,R4
5:		LDR R0,	=num1
0x00000384	E59F0000	LDR	R0,[PC]
6:		LDR R1,	=0xaabbccdd
0x00000388	E59F1000	LDR	R1,[PC]
0x0000038C	00000374	ANDEQ	R0,R0,R4,ROR R3
0x00000390	AABBCCDD	BGE	0xFEEF370C

- When the PC register is involved in instructions like `LDR` and `MOV`, its value is the address of the current instruction + 8.
- For instruction: LDR R0, [PC]
  - The address of the next instruction: 0x388
  - 0x388 + 0x4 = 0x38C, which points to the value 0x374 (address of num1)
- This design is to preserve compatibility for programs written for early ARM processors.

Lecture 6

- *LDR Pseudo-Instruction*

➢ ❓ **Why does LDR R0, =num1 give R0 = 0x00000374 (the address of num1) instead of loading the value stored at that address (e.g., 0x00000001)?**

✅ **The short answer:**
Because **`LDR R0, =num1`** loads the **address of the label** num1, not the value stored at that label.
It's a design behavior of the **LDR** pseudo-instruction with a label.

Lecture 6

- *LDR Pseudo-Instruction*

```
0x00000374  00000001  ANDEQ      R0,R0,R1
0x00000378  00000002  ANDEQ      R0,R0,R2
0x0000037C  00000003  ANDEQ      R0,R0,R3
0x00000380  00000004  ANDEQ      R0,R0,R4
     5:                  LDR R0, =num1
0x00000384  E59F0000  LDR        R0,[PC]
     6:                  LDR R1, =0xaabbccdd
0x00000388  E59F1000  LDR        R1,[PC]
0x0000038C  00000374  ANDEQ      R0,R0,R4,ROR R3
0x00000390  AABBCCDD  BGE        0xFEEF370C
```

➢ ❓ **Why does LDR R0, =num1 give R0 = 0x00000374 (the address of num1) instead of loading the value stored at that address (e.g., 0x00000001)?**

🔍 **What actually happens:**

When you write **`LDR R0, =num1`**, the assembler creates an entry in the **literal pool**

Address	Value
0x0000038C	0x00000374

the CPU reads the value store at **`0x0000038C`** (= **`0x00000374`** = address of **`num1`**) and load it to **`R0`**

Lecture 6

- *LDR Pseudo-Instruction*

➢ **Take-away information**

- **`LDR =something`** is a **pseudo-instruction**
- The assembler creates a **literal pool** after the instruction section
- The **PC** is always **current address + 8** when executing an instruction
- The offset is calculated so that the **LDR** accesses the right constant

Before End:
Final Exam & Assessment Explaination

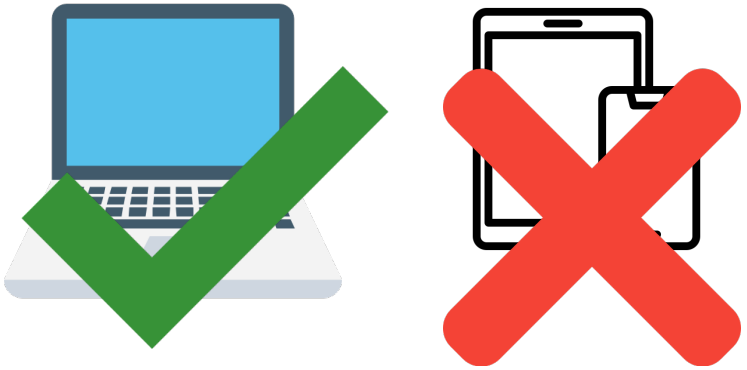
Final Exam

- Final Exam will take **70% of the module mark.**
- **Close-book exam: 0 materials will be provided, 0 materials can be brought in.**
- **On-site exam** in Computer Lab Rooms from 10:00 am - 12:00 pm (should arrive before 9:30 am) on **May 26th**
- **2 hours exam, conduct on LMO, with SEB** (single-way exam, cannot go back to the previous questions, cannot leave the current page, otherwise the machine will be locked automatically).
- Questions are exacted from lectures and labs (about 9 big questions)
- **Missing final exam = missing 70% of the module mark**

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!!