CZ1106
CE1106

Chapter 4

Addressing Modes

1

CZ1106
CE1106

Addressing Modes

Introduction to Assembly Programming and Addressing Modes

Learning Objectives (4.1)

1. Identify why and when to use assembly language programming.
2. Describe what are addressing modes.

2

each mnemonics is represented by 4 bytes ; one to one correspondence ; every mnemonics has a series of machine codes ⇒ stored in memory

CZ1106 CE1106

What is an Assembly Program?

- Unlike high-level programming languages, assembly level statements:
  - Are known as **mnemonics**. Each has a one-to-one correspondence with a binary pattern (**machine code**) that is directly understood by CPU.
  - Are **hardware-dependent** and address the architecture of processor directly. (e.g. they are CPU register-aware and reference them by name).
  - Are converted to machine code by an **assembler**.

not complie, "assemble" is the correct word

can be anything

```
if (a > c)
  b = a;
else
  b = c;
```

C program example

standard name

```
      CMP R0,R2
      BLE Else
      MOV R1,R0 ;b=a
      B   Skip
Else  MOV R1,R2 ;b=c
Skip  :
```

ARM assembly program equivalent

3

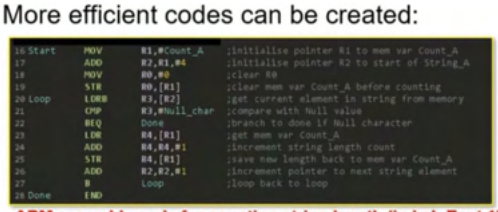
CZ1106 CE1106

Why Use Assembly Language?

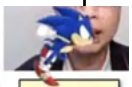
- More efficient codes can be created:
  - Codes with faster execution **speed**.
    e.g. Algorithms for **real-time** signal processing in handheld devices can be **computationally demanding**.
  - More compact program **size**.
    e.g. Low cost embedded devices may have **small memory** capacity but require many functionalities.
  - Exploit **optimized features** of processor's ISA.
    e.g. High-level language compiled codes may not **exploit optimized** instructions, addressing modes and features available in the processor instruction set architecture to produce efficient run-time code.
  - Many cybersecurity jobs needs good knowledge in assembly programming.

4

More efficient codes can be created:



ARM assembly code for counting string length (in Lab Expt 1)



167 cycles — Given code

83 cycles — My best effort

Exploit **optimized features** of processor's ISA.
e.g. High-level language compiled codes may not **exploit optimized** instructions, addressing modes and features available in the processor instruction set architecture to produce efficient run-time code.

Assembly used for critical portions only

not able to port to another processor

CZ1106 CE1106

When to Use Assembly Language?

- Critical parts of the operating system's software.
  Especially parts of system kernel that are constantly being executed (e.g. scheduler, interrupt handlers).
- Input/Output intensive codes.
  Device drivers and "loopy" segments of code that processes streaming data (e.g. video decoders, etc).
- Time-critical codes.
  Code that detect incoming sensor signals and respond rapidly, e.g. Anti-lock brake system (ABS) in cars.

Learn More: Google "Is Linux kernel written in assembly"

5

CZ1106 CE1106

Addressing Modes

- Addressing mode (AM) is concerned with how data is **accessed**, not the way data is processed.
  - The correct AM allows the CPU to identify the actual operand or the address location where operand is stored.

In memory or in the instruction itself

- The ARM processor instruction set architecture supports many different addressing modes.
  - Register direct
  - Immediate data
  - Register indirect
  - Register indirect with offset
  - Register indirect with index register
  - Pre and post auto-indexing

Learn More: Google "addressing modes"

6

CZ1106 CE1106

Addressing Mode Examples

Addressing Mode	ARM	Intel
Absolute (Direct)	None	`MOV AX,[1000h]`
Register Direct	`MOV R1,R0`	`MOV AX,DX`
Immediate	`MOV R1,#3`	`MOV AX,0003h`
Register Indirect	`LDR R1,[R0]`	`MOV AX,[BX]`
Register Indirect with Offset	`LDR R1,[R0,#4]`	`MOV AX,[BX+4]`
Register Indirect with Index	`LDR R1,[R0,R2]`	`MOV AH,[BX+DI]`
Implied	`BNE LOOP`	`JMP -8`

7

CZ1106 CE1106

Summary

- Codes written well in assembly language can usually execute **faster** and are smaller in **size**.
- Code for **low-level** OS kernels, **I/O** intensive and **time**-critical operations can benefit significantly from assembly-level coding.
- Understanding the **characteristics** and **application** of different **addressing modes** available in a processor's ISA allows programmers to write efficient codes.

8

CZ1106
CE1106

Chapter 4

Addressing Modes

Register Direct and Immediate Addressing

Learning Objectives (4.2)

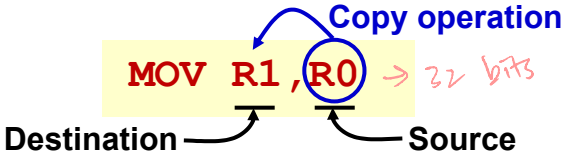
1. Describe what is register direct.
2. Describe what is immediate data and its application.

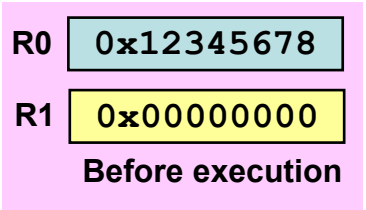
9

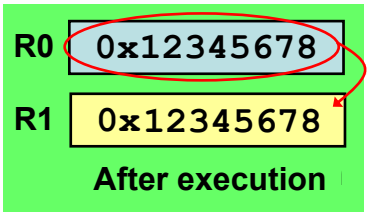
CZ1106
CE1106

Register Direct

- Operand is the content of the specified **register**.
  - Register direct can be used for both destination and source operand.
  - In the **MOV** instruction, the right operand is the source and left operand is the destination.







  - A **fast** addressing mode since no further memory access is involved during execution.
  - Should be used to optimise execution speed.

10

CZ1106 CE1106

Register Direct (cont)

- All ARM's 16 registers can be a register direct operand.
  - These registers can be either a source or destination operand.

```
MOV   R3,LR  ;make copy of LR in R3
MOVS  R0,R0  ;test for N or Z condition in R0
MOV   PC,R1  ;make a jump to address in R1
```

R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
SP
LR
PC

11

CZ1106 CE1106

Immediate Addresing

- **Operand** is directly specified **within** the **instruction** itself.
  - "**#**" symbol precedes the immediate value.
  - Example: `MOV R1,#3` ← Immediate Value
    Copy operation
  - After execution, the **immediate value** is **copied** into the destination register (left operand).
  - Immediate addressing can only be used as a **source operand**.
  - Used for loading **constant values** into registers. Values must be known at the time of coding (e.g. load loop count into a loop counter register).

R1	0x12345678

Before execution

R1	0x00000003

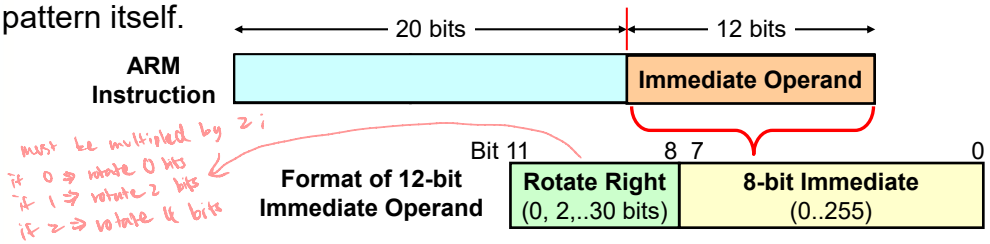
After execution

12

CZ1106
CE1106

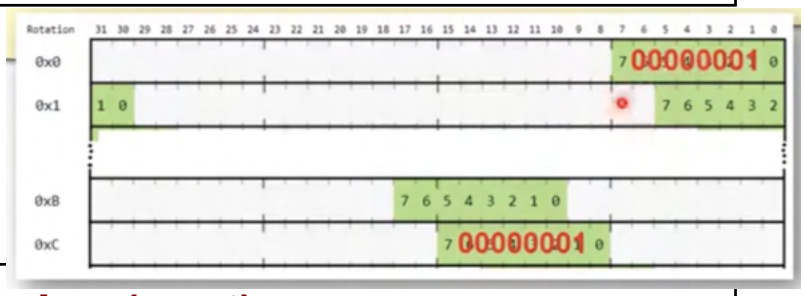
Immediate Addressing (cont)

- How is the 32-bit immediate value encoded?
  - The immediate value is specified **within the instruction** bit pattern itself.



- How can a 12-bit operand encode a 32-bit immediate value?
  - It can only describe a **subset** of all 2^{32} possible values.
  - Immediate value is a number between (**0..255**) rotated right by **2*n*** bits, where the value of ***n*** is given by 4 bits (**0** ≤ ***n*** ≤ **15**).

13



CZ1106
CE1106

Immediate Addressing (cont)

- Assembler does the necessary calculations and gives warning if requested immediate value cannot be encoded.

all is in 8 bit ; no rotation

```
MOV R3,#0xFF     ;immediate values within 8 bits always valid
MOV R0,#0x100    ;right rotate 8-bit value of 0x01 with n=12
MOV R1,#0x102    ;this is not a valid immediate value
```

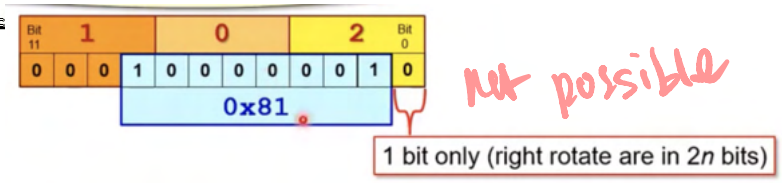
24 bit right rotate

not divisible by 4 ; hence cannot

only possible with this

```
MOV R1,#0x100 ; load 0x100 to R1
ADD R1,R1,#2  ; add 2 to R1
```

- A combination of instructions can be used to achieved the desired immediate values that is not valid.

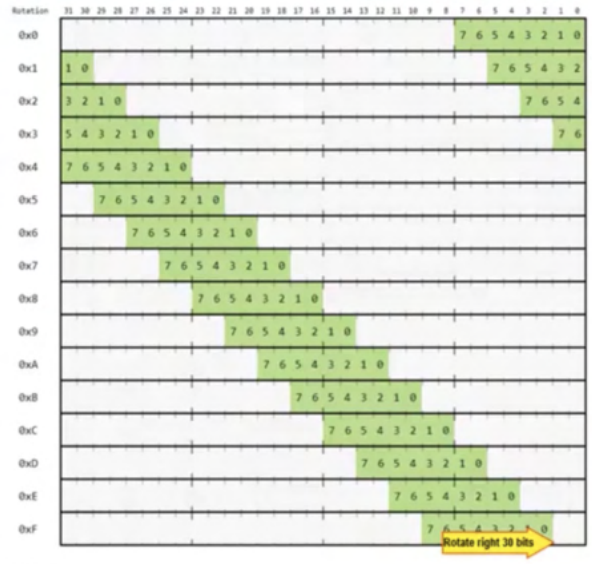


not possible

14

Slide 13 (cont)

Immediate Addressing (cont)



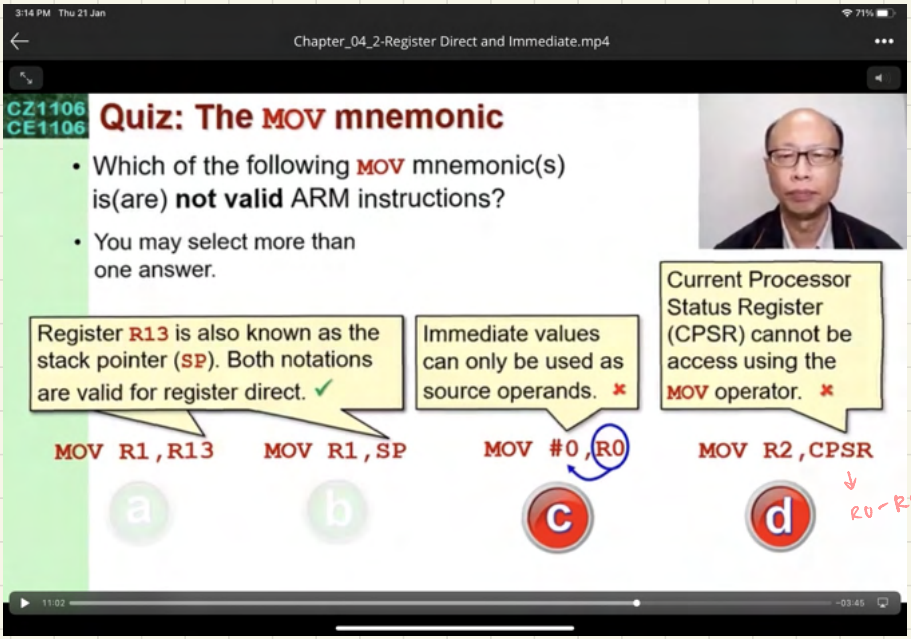
$2n = 0$

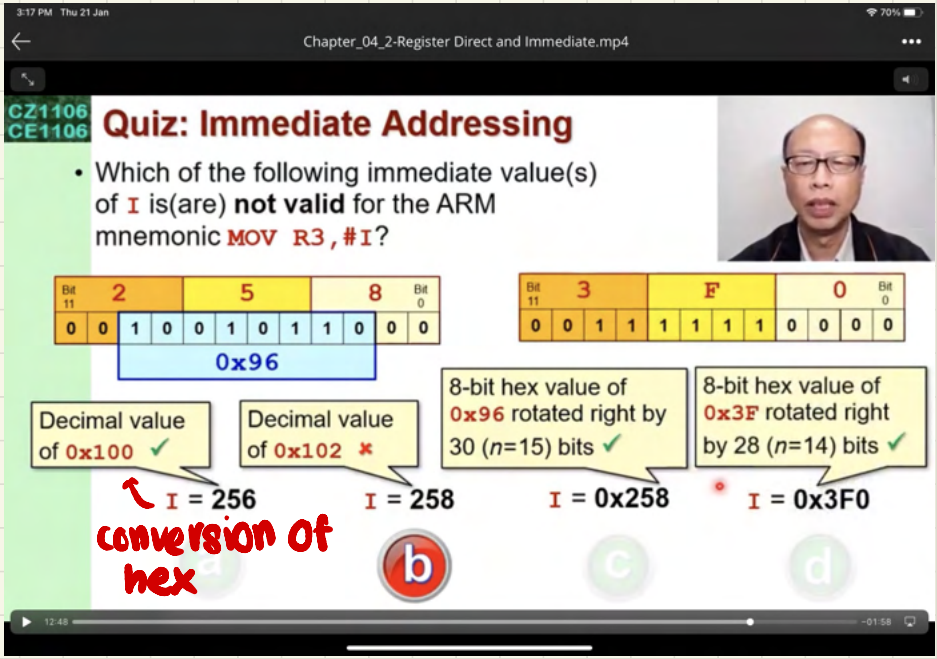
$2n = 2$

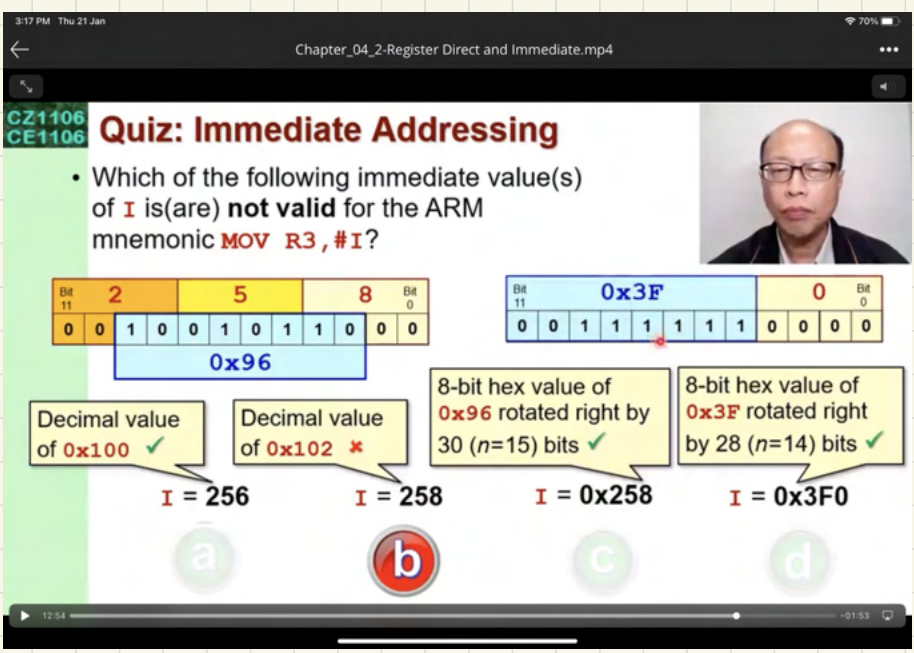
$2n = 30$

Right rotating an 8-bit base value by $2n$ bits to get a better 32-bit range

with 4 bits, you can have the value 0 to 15


3:14 PM Thu 21 Jan
71%
Chapter_04_2-Register Direct and Immediate.mp4
CZ1106
CE1106
Quiz: The MOV mnemonic
• Which of the following MOV mnemonic(s) is(are) not valid ARM instructions?
• You may select more than one answer.
Register R13 is also known as the stack pointer (SP). Both notations are valid for register direct. ✓
Immediate values can only be used as source operands. ✗
Current Processor Status Register (CPSR) cannot be access using the MOV operator. ✗
MOV R1,R13
MOV R1,SP
MOV #0,R0
MOV R2,CPSR
a
b
c
d
R0 - R15 only
11:02
-03:45


3:17 PM Thu 21 Jan
70%
Chapter_04_2-Register Direct and Immediate.mp4
CZ1106
CE1106
Quiz: Immediate Addressing
• Which of the following immediate value(s) of I is(are) not valid for the ARM mnemonic MOV R3,#I?
Bit 11 2 5 8 Bit 0
0 0 1 0 0 1 0 1 1 0 0 0
0x96
Bit 11 3 F 0 Bit 0
0 0 1 1 1 1 1 1 0 0 0 0
Decimal value of 0x100 ✓
Decimal value of 0x102 ✗
8-bit hex value of 0x96 rotated right by 30 (n=15) bits ✓
8-bit hex value of 0x3F rotated right by 28 (n=14) bits ✓
I = 256
I = 258
I = 0x258
I = 0x3F0
conversion of hex
a
b
c
d
12:48
-01:58


3:17 PM Thu 21 Jan
70%
Chapter_04_2-Register Direct and Immediate.mp4
CZ1106
CE1106
Quiz: Immediate Addressing
• Which of the following immediate value(s) of I is(are) not valid for the ARM mnemonic MOV R3,#I?
Bit 11 2 5 8 Bit 0
0 0 1 0 0 1 0 1 1 0 0 0
0x96
Bit 11 0x3F 0 Bit 0
0 0 1 1 1 1 1 1 0 0 0 0
Decimal value of 0x100 ✓
Decimal value of 0x102 ✗
8-bit hex value of 0x96 rotated right by 30 (n=15) bits ✓
8-bit hex value of 0x3F rotated right by 28 (n=14) bits ✓
I = 256
I = 258
I = 0x258
I = 0x3F0
a
b
c
d
12:54
-01:53

CZ1106
CE1106

Summary

- Register direct is **efficient** as its execution involves no access to memory.

↗ 1 memory cycle instruction

- Immediate addressing encode the operand **within the instruction**.
  - Like register direct, immediate addressing in the ARM is **efficient** as memory access is not incurred during execution, only when fetching the instruction.
  - Because data is encoded within fixed-length instruction, **only a subset** of immediate values are available.
- Immediate addressing is used when the operand **value is known** during the time of coding (e.g. loading known constants into registers).

15

CZ1106
CE1106

16

CZ1106
CE1106

Chapter 4

Addressing Modes

Register Indirect with Base Register

Learning Objectives (4.3)

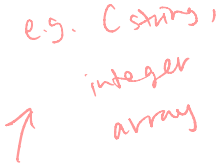
1. Describe what is register indirect and the ARM instructions that support this addressing mode.
2. Describe the variants and application of register indirect that uses base plus offset and index register.
3. Compare the relative pros and cons of register direct and register indirect addressing modes.

17

CZ1106
CE1106

Limitation of Register Direct and Immediate Addressing

- Register direct and immediate addressing **do not** allow CPU to access operands stored in **memory**.
  - C variables are usually allocated memory for storage (especially large arrays).
- How do you specific a 32-bit address in memory using a 32-bit long instruction?
  - The ARM specifies the 32-bit address of the operand in a 32-bit **register**.
  - The register with the memory address **points to the memory** location where the operand is stored.
  - Memory operand is fetched during instruction execution using **register indirect** addressing.
  - The ARM uses the `LDR` and `STR` mnemonics to access memory operands.

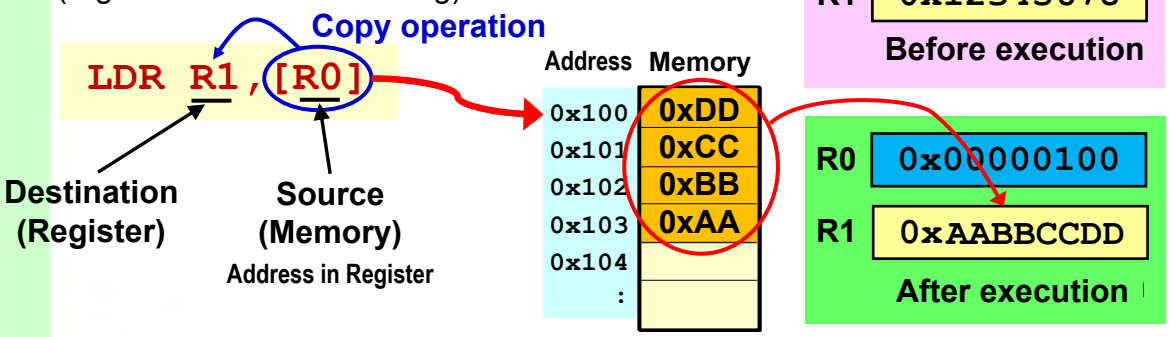


18

CZ1106 CE1106

The LDR Instruction

- The **LDR** operator is used to copy **memory** content to a register.
  - The left operand the destination register.
  - The right source operand is a memory location whose address is contained in a register (register indirect addressing).

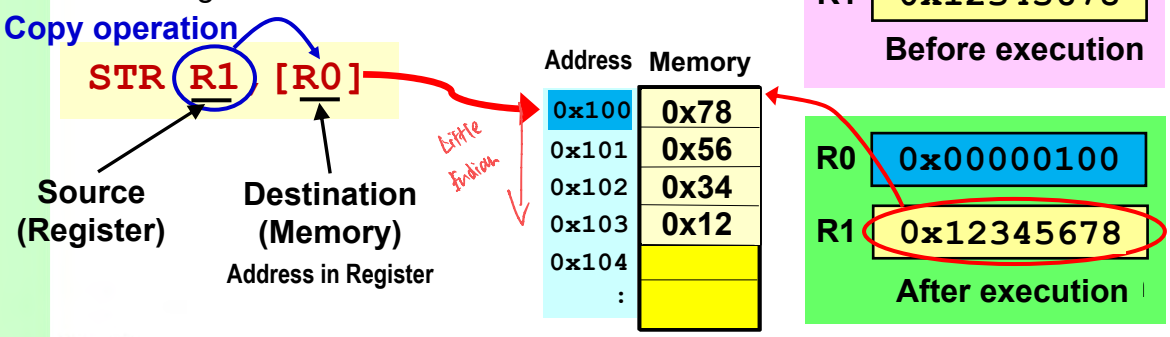


19

CZ1106 CE1106

The STR Instruction

- The **STR** operator is used to copy register content to **memory**.
  - The left operand is always a source register.
  - The right destination operand is a memory location whose address is contained in the indirect register.



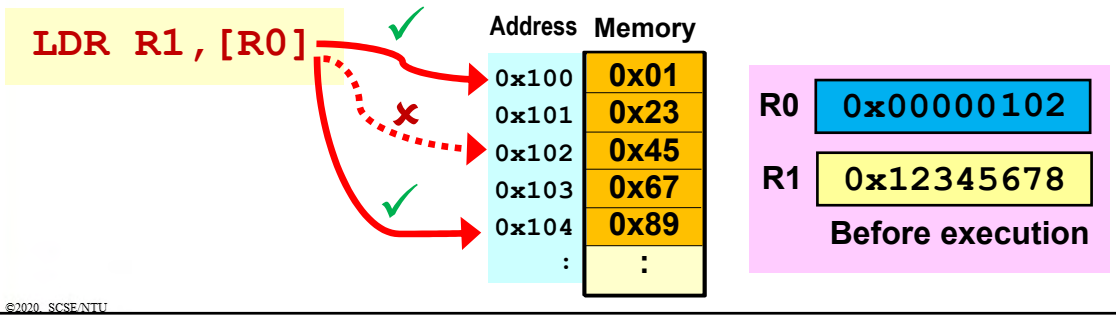
20

little endian
big endian

CZ1106 CE1106

Data Alignment Constraints

- Access of 32-bit operand from memory must follow data alignment constraints.
  - The **4-byte data** read or written to memory must start at an address that is a **multiple of 4**.
  - The effects of an unaligned memory access depend on the ARM architecture but they invariably result in **performance degradation**.

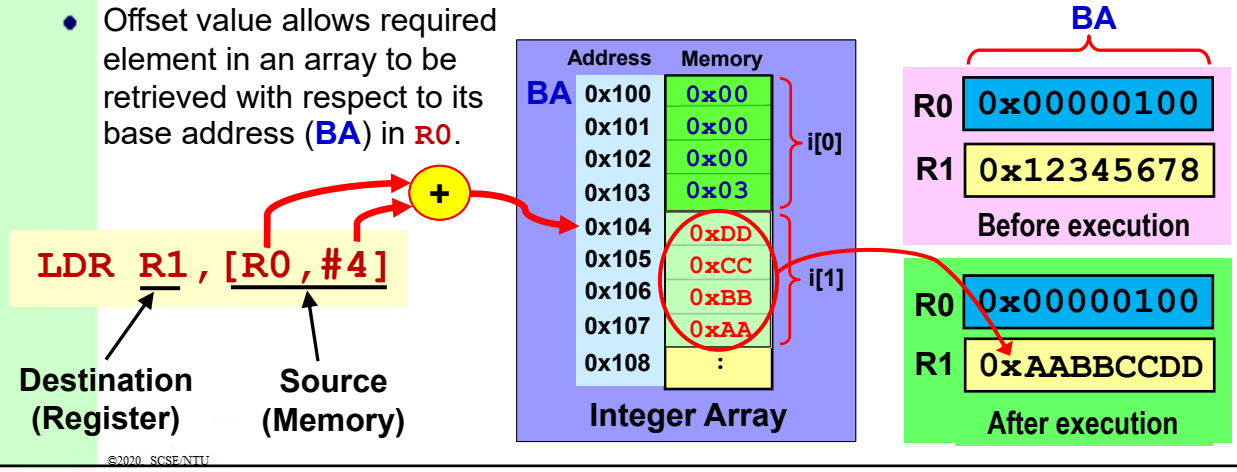


21

CZ1106 CE1106

Register Indirect with Offset

- **Adds** a specific **offset value** to the indirect register to compute the effective address (EA) in memory.
  - **Base Plus Offset** addressing does not change indirect register's content.
  - Offset value allows required element in an array to be retrieved with respect to its base address (**BA**) in `R0`.



22

CZ1106 CE1106

Program Example

Accessing Array Elements

- Use base plus offset to access array element whose index is known during coding time.

```
main()
{
// assume base address
// of array i is 0x100
int i[5];

i[0]=7;
i[4]=7;
}
```

C program example
Assign first & last elements of array `i` with value of 7.

```
MOV   R2,#0x100      1
MOV   R1,#7          2
STR   R1,[R2,#0]     3
STR   R1,[R2,#16]
```

Using register indirect with base plus offset

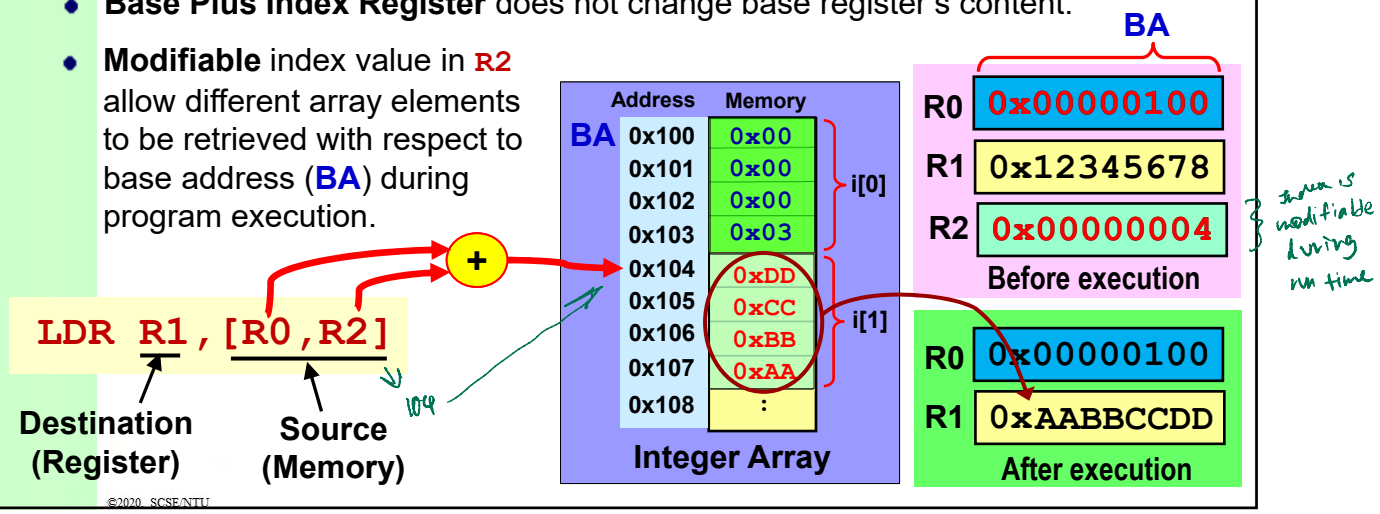
1. Initialize base address of array `i` into register `R2`.
2. Load value of 7 into register `R1`.
3. Store the value of 7 into **i[0]** and **i[4]** using offsets of 0 and 16 of register **R2** respectively .
   - In computing offset, note that each integer element occupies 4 bytes in memory.

23

CZ1106 CE1106

Register Indirect with Index Register

- This variant **adds** the content of the **index register** to the indirect register to compute EA.
  - **Base Plus Index Register** does not change base register's content.
  - **Modifiable** index value in `R2` allow different array elements to be retrieved with respect to base address (**BA**) during program execution.



24

CZ1106 CE1106

Program Example

Clearing All Array Elements

- Use base plus index register to access each array element in turn.

```
main() {
// assume base address
// of array i is 0x100
int i[400];
int n=0;

while (n < 400) {
    i[n] = 0;
    n = n + 1;
    }
}
```

C program example
Initialise all 400 elements in array `i` with zero.

```
MOV   R2,#0x100   (1)
MOV   R0,#0       (2)
MOV   R1,#0       (2)
  :
STR   R0,[R2,R1]  (3)
ADD   R1,R1,#4    (4)
```

loop back 399 times

3 ×400 = 1200 cycles

Using register indirect with base plus index

1. Initialize base address of array `i` into register `R2`.
2. Load value of 0 into register `R0` and `R1`(index register).
3. Store 0 in `R0` into `i[n]` using current index value in `R1` plus base address in `R2`.
4. Increment index by 4, the size of each integer element in array.

25

CZ1106 CE1106

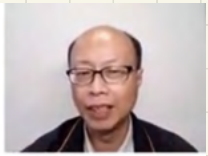
Summary

- Register indirect (with the `LDR` and `STR` operators) allows memory operands to be accessed.
- There are two variants of register indirect using base register.
  - Register indirect with **offset** (base plus offset)
  - Register indirect with **index** (base plus index register)
  - Contents in base register **do not change** after execution.
- Given the **base address** of an array, register indirect with base addressing is useful for accessing the contents of the array.
  - Use base plus offset if position of array element is known during coding time
  - Use base plus index if array element position is computed during run time.

26

CZ1106
CE1106

Quiz: Register Indirect



- Given the initial states shown, which are the correct states of `R0` and `R2` after the execution of the instruction `STR R2,[R0,#4]`?

base don't change

Initial States

R0	0x00000100
R2	0x12345678

Address	Memory
0x100	0x00
0x101	0x00
0x102	0x00
0x103	0x00
0x104	0x88
0x105	0x88
0x106	0x88
0x107	0x88
:	:

©2020, SCSE/NTU

a

R0 0x00000100
R2 0x12345678

R0 0x00000100
R2 0x88888888 ✗

Adding `R0` with offset `4` to get the effective address of memory operand does not alter content of indirect register `R0` after execution.

b

R0 0x00000104
R2 0x1245678 ✗

R0 0x00000100
R2 0x00000000 ✗

`STR` operator copies register content to memory. So source register is not altered after execution. Only destination memory location is modified.

↳ register itself does not get altered

(unless maybe LDR)

CZ1106
CE1106

Quiz: Execution Speed

- With the initial states shown, which instruction will be **able** to load the value `0x102` into register `R2` with the **least** number of clock cycles?

Initial States

R0	0x00000102
R2	0x12345678

Address	Memory
0x100	0x02
0x101	0x01
0x102	0x00
0x103	0x00
0x104	0x00
0x105	0x00
:	:

`MOV R2,#0x102` ✗

The immediate value #0x102 is not valid

`MOV R2,R0` ✓

0x102 in R0 is copied into R2. only 1 memory cycle

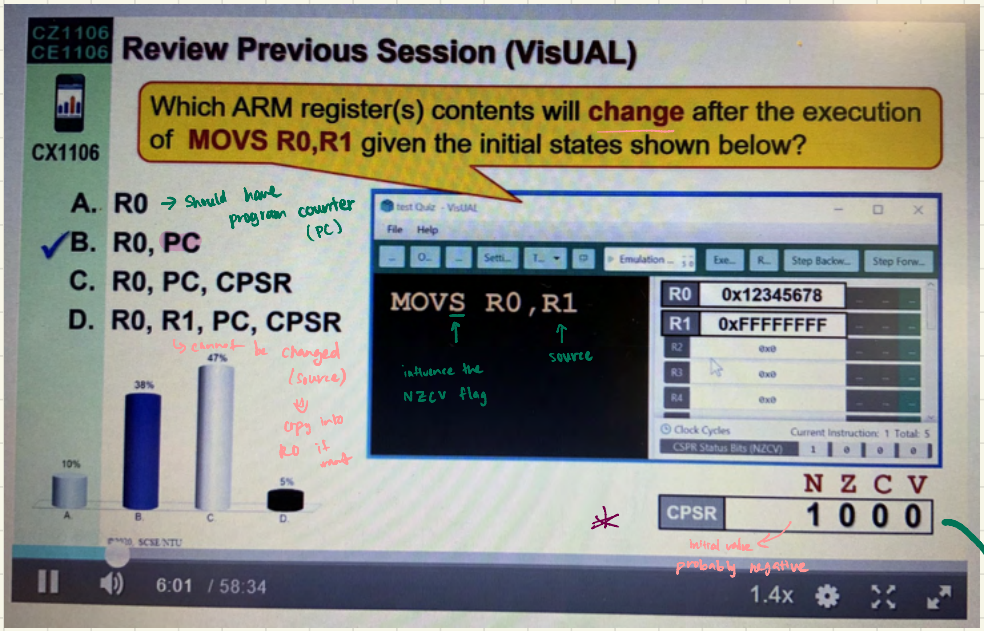
`LDR R2,[R0]` ✗

Address 0x102 in R0 does not meet data alignment constraints for 4-bytes memory access.

`LDR R2,[R0,#-2]` ✗

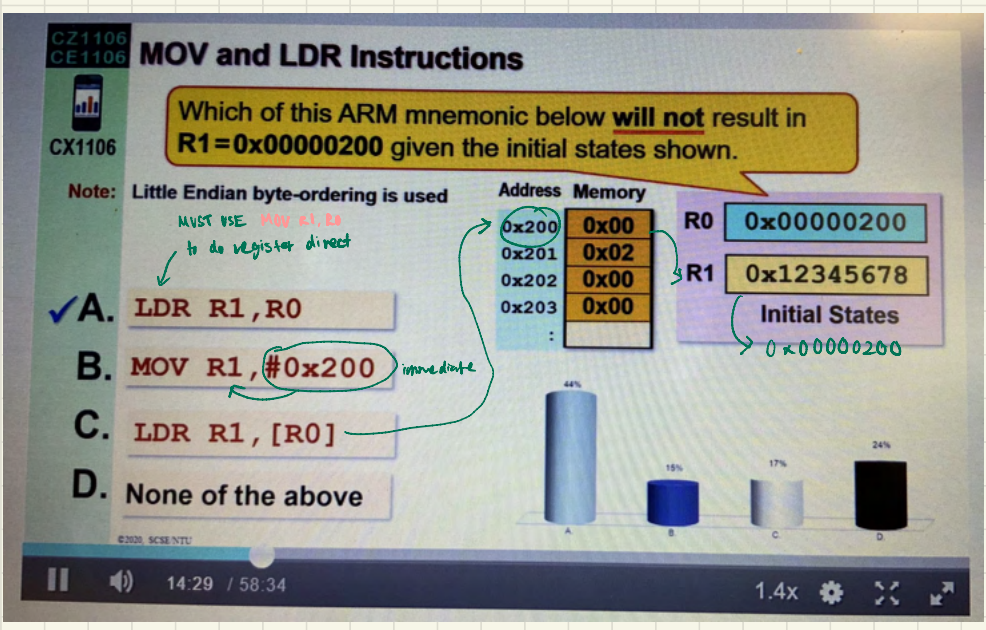
Effective address 0x100 meets data alignment constraints but LDR takes 2 memory cycles (not the least no. of clk cycles)

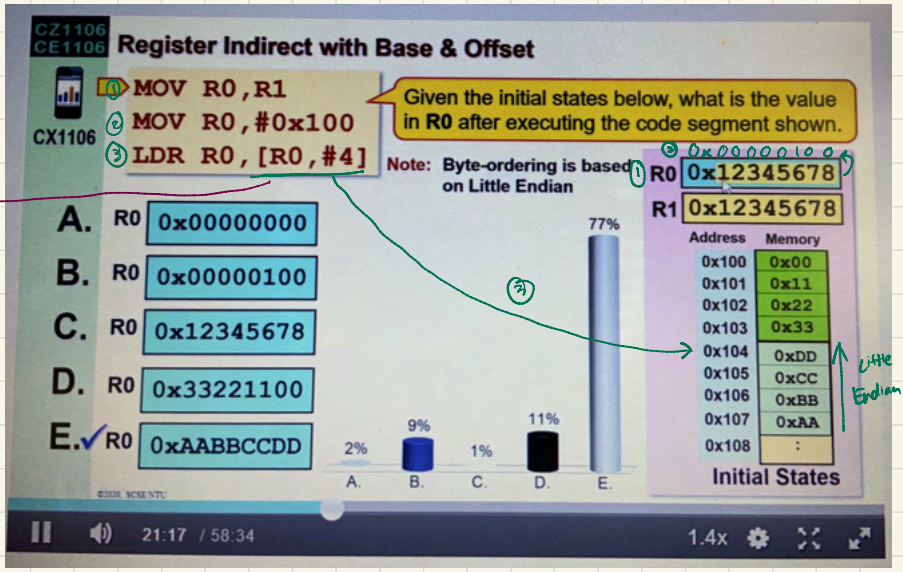
Lecture (22/1)



register to register:
LDR won't work

if its:
0 0 0 0
⇒ Answer : C

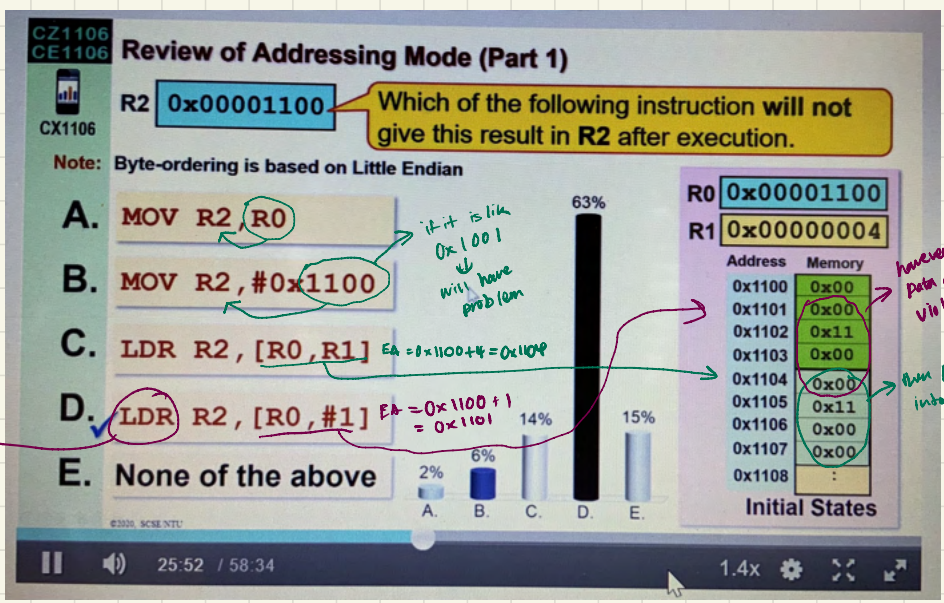




Register indirect,

EA = 0 x100 + 4 (0x...100)

= 0x104



if it is like 0x1001 will have problem

EA = 0x1100+4 = 0x1104

EA = 0x1100+1 = 0x1101

however, Data alignment violation

then placed into R2

So that no data alignment violation, use LDRB

Bit size handler

CZ1106
CE1106

Chapter 4

Addressing Modes

Register Indirect with Autoindexing and Stacks

Learning Objectives (4.4)

1. Describe what is autoindexing feature of ARM's register indirect addressing mode.
2. Describe the differences between pre-index and post-index addressing modes.
3. Describe the various stack implementations and operations using the ARM addressing modes.

27

CZ1106
CE1106

Register Indirect with Autoindexing

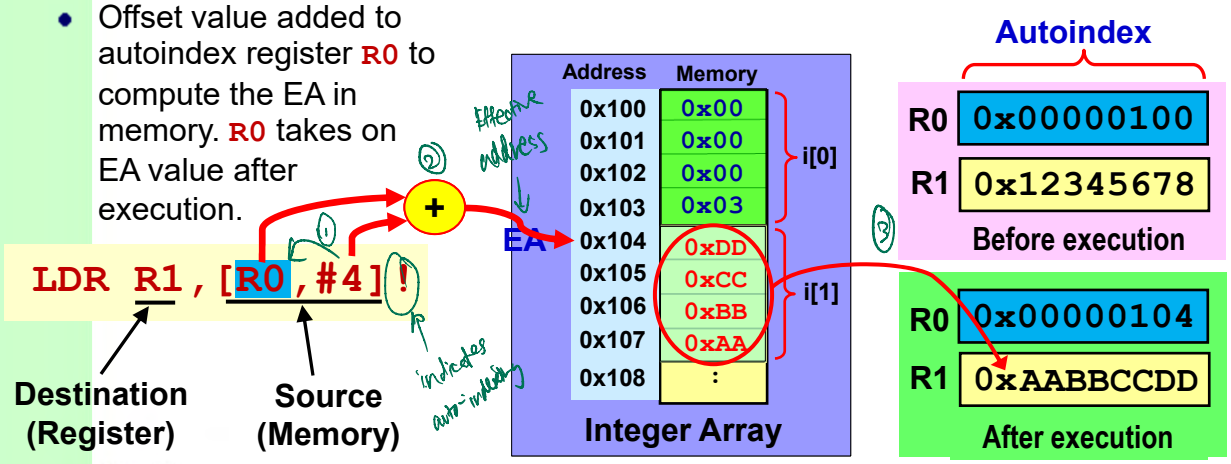
- Recap – Register indirect with base register:
  - The base address in the indirect register can be added with an offset or the contents of index register to compute effective address of operand in memory.
  - Base address is **never modified** after execution as it is assumed to be the sole reference to the start of the array.
- What if we allow the indirect register to be modified before or after computing the effective address?
  - Keep a copy of the array's base address elsewhere.
  - **Autoindexing** allows the indirect register's content to be **modified** during execution.
  - Autoindexing provides an efficient way to access **consecutive** array elements.

28

CZ1106 CE1106

Offset with Autoindexing

- Adds offset value to the autoindex register (AR) to compute effective address (EA) and AR gets modified.
  - "!" in mnemonic causes autoindex register to be modified with the EA.
  - Offset value added to autoindex register R0 to compute the EA in memory. R0 takes on EA value after execution.



29

CZ1106 CE1106

Program Example (Optimised Version)

Clearing All Array Elements

- Use offset with autoindex to efficiently access each array element in turn.

```
main() {
// assume base address
// of array i is 0x100
int i[400];
int n=0;

while (n < 400) {
    i[n] = 0;
    n = n + 1;
    }
}
```

C program example
Initialise all 400 elements in array `i` with zero.

```
MOV R2,#0x100     1
MOV R1,#0         2
STR R1,[R2]       3
  :
STR R1,[R2,#4]!   4
```

loop back 398 times

2 ×400 = 800 cycles

Using register indirect plus offset with autoindex

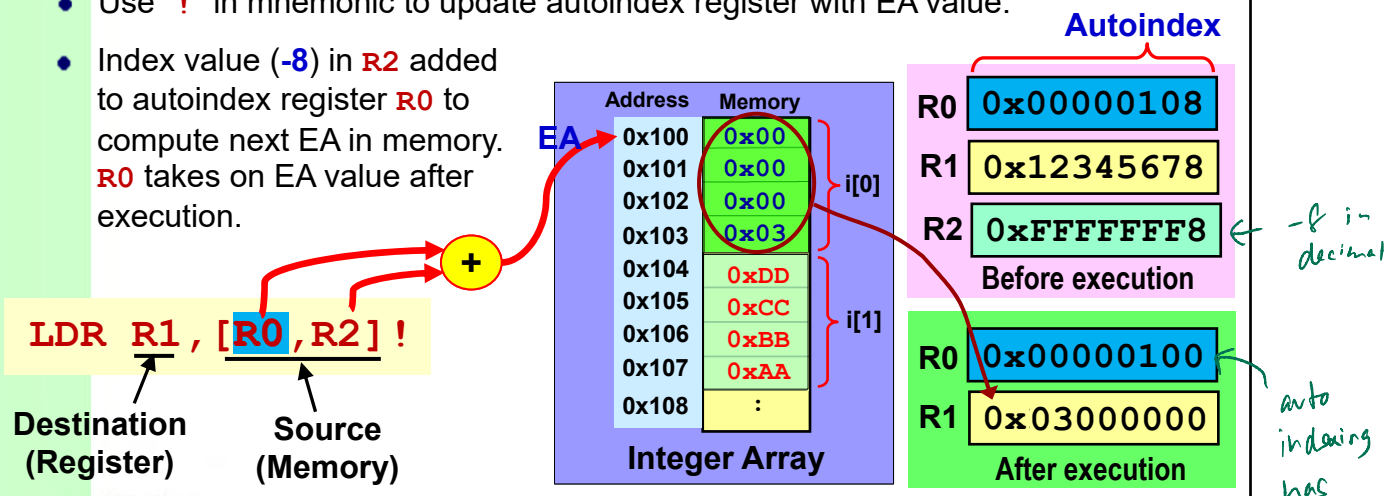
1. Initialize base address of array `i` into register `R2`.
2. Load value of 0 into source register `R1`.
3. Store 0 in `R1` into first element of array `i[0]`.
4. Store 0 in `R1` into `i[n]` using current effective address (EA) of autoindex register `R2` plus offset `4`. Then put this EA into `R2`.

30

CZ1106 CE1106

Index Register with Autoindex

- Adds index register value to autoindex register (AR) to compute effective address (EA) and AR gets modified.
  - Use "**!**" in mnemonic to update autoindex register with EA value.
  - Index value (**-8**) in `R2` added to autoindex register `R0` to compute next EA in memory. `R0` takes on EA value after execution.



©2020 SCSE/NTU

31

CZ1106 CE1106

Pre-index and Post-index

- In **pre-index**, the indirect register is **autoindex before** being used to compute effective address.

```
LDR R1,[R0,#4]!   ; R0 = R0+4
                  ; R1 = mem[R0]
```

Offset with Autoindexing (pre-index)

```
LDR R1,[R0,R2]!   ; R0 = R0+R2
                  ; R1 = mem[R0]
```

Index with Autoindexing (pre-index)

- In **post-index**, the indirect register is used to compute the effective address **after** it is **autoindexed**.

```
LDR R1,[R0],#4    ; R1 = mem[R0]
                  ; R0 = R0+4
```

Offset with Autoindexing (post-index)

```
LDR R1,[R0],R2    ; R1 = mem[R0]
                  ; R0 = R0+R2
```

Index with Autoindexing (post-index)

©2020 SCSE/NTU

32

CZ1106 CE1106

Program Example (Alternative Version)

Clearing All Array Elements

- Use offset with **post-index autoindex** to keep all array access within loop.

```
main() {
// assume base address
// of array i is 0x100
int i[400];
int n=0;

while (n < 400) {
    i[n] = 0;
    n = n + 1;
    }
}
```

C program example
Initialise all 400 elements in array `i` with zero.

```
MOV R2,#0x100      (1)
MOV R1,#0          (2)
   :
STR R1,[R2],#4     (3)   2 ×400 = 800 cycles
```

loop back 399 times

neater than pre index (slide 30)

Using offset with post-index autoindexing

1. Initialize base address of array `i` into register `R2`.
2. Load value of 0 into source register `R1`.
3. Store 0 in `R1` into **i[n]** using current effective address (EA) in indirect register `R2`. Then add offset `4` to the current EA in `R2` so that `R2` is now pointing to the next array element.

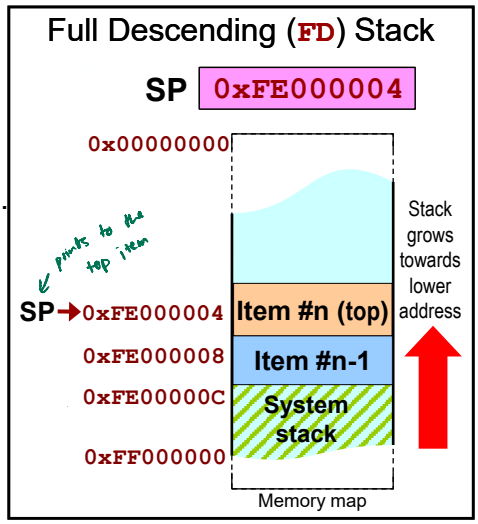
33

CZ1106 CE1106

The System Stack

- A stack is a first-in, last-out linear data structure that is maintained in the memory's data area.
  - The system stack in the ARM is maintained by a dedicated stack pointer (**SP**) or **R13**.
  - The `FD` stack grows towards **lower memory** addresses.
    (e.g. by default, **SP** starts at `0xFF000000` in VisUAL ARM simulator).
  - In the `FD` stack, the **SP** points to the **top item (full)** on the stack (but SP can also point to the next **empty** space on the stack).
  - The 3 basic stack operations are **push**, **pop** and **access** items on the stack.

Learn More: Google "ARM stack implementation"



34

push addition

empty stack from the top

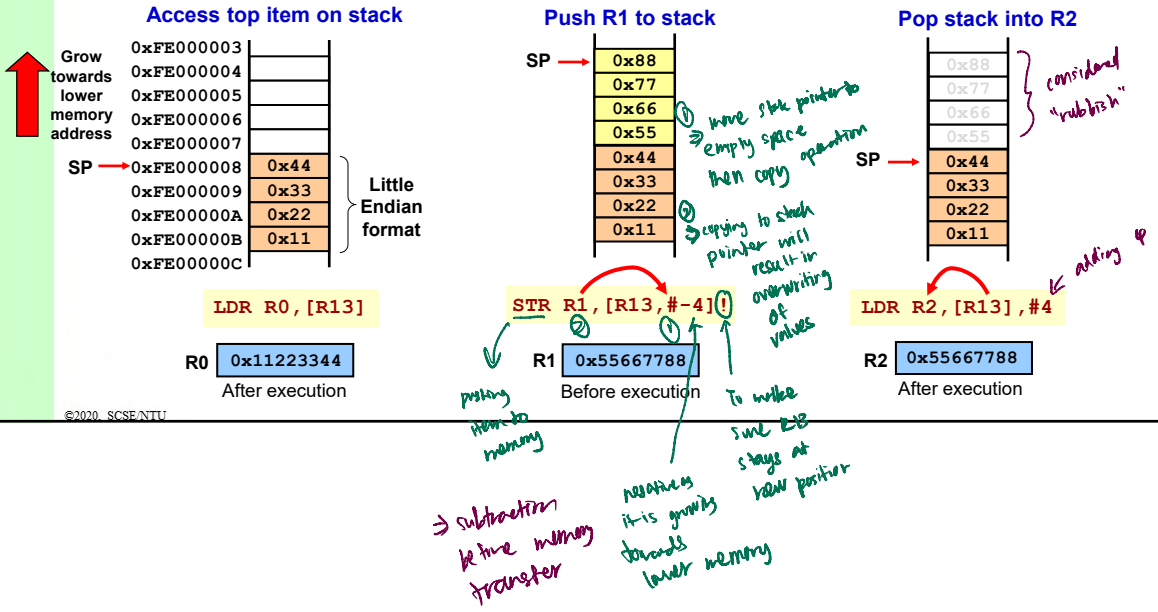
items on the stack from the top

CZ1106 CE1106

ARM Stack Implementation (FD)

- The are 4 possible stack implementations supported by the ARM instruction set.
  - **F**ull **D**escending, **F**ull **A**scending, **E**mpty **D**escending and **E**mpty **A**scending

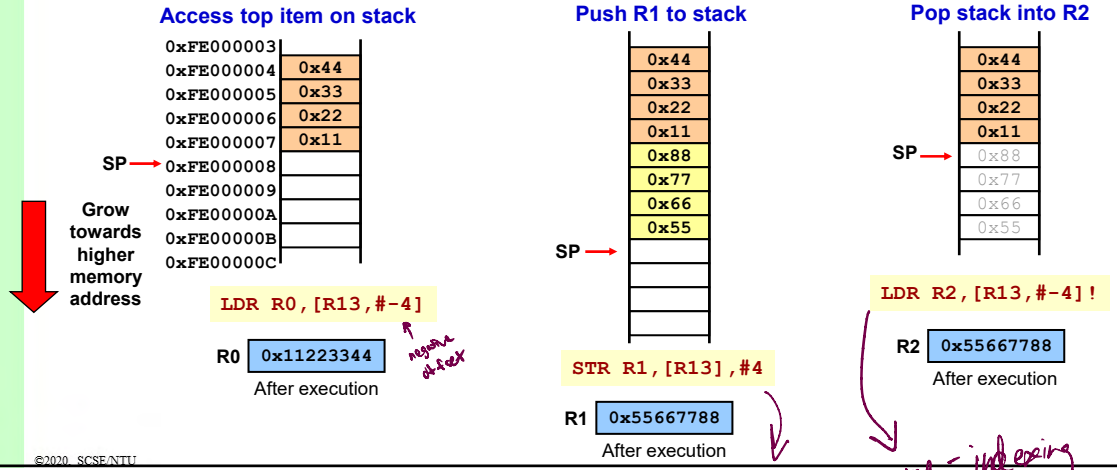
Example of **Full Descending** (FD) stack implementation:



CZ1106 CE1106

Empty Ascending Implementation (EA)

- EA is an alternative stack implementation.
  - **Empty** means **SP** points to an available **unoccupied** stack space.
  - **Ascending** means stack grows toward **higher** memory address.



⇒ offsets are in values of 4, cause items on stack is 32 bits

CZ1106
CE1106

Quiz: Autoindexing

- Given the initial states shown, which are the correct states of `R0` and `R2` after the execution of the instruction `LDR R2,[R0],#4`?

Initial States

R0	0x00000100
R2	0x12345678

Address	Memory
0x100	0x00
0x101	0x00
0x102	0x00
0x103	0x00
0x104	0x88
0x105	0x88
0x106	0x88
0x107	0x88
:	:

✗ R0 0x00000100, R2 0x00000000

Autoindex will modify the indirect register `R0`.

✗ R0 0x00000100, R2 0x88888888

This is offset with **post-index** autoindexing. This mean `EA` is given by `0x100` in `R0` before offset `4` is added to `R0`.

R0 0x00000104, R2 0x88888888 ✗

Offset with **pre-index** autoindexing given by `LDR R2,[R0,#4]!` gives this answer.

R0 0x00000104, R2 0x00000000 ✓

d

LDR R2, [R0], #4

① ②

CZ1106
CE1106

Quiz: The ED Stack

- Give the mnemonic to **push** the register `R1` on to an **Empty Descending** (ED) stack.

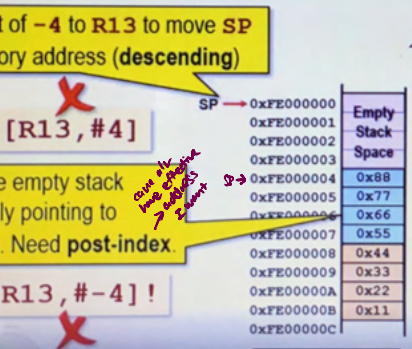
Then, add **-ve** offset of `-4` to `R13` to move `SP` towards lower memory address (**descending**)

✗ `STR R1,[R13,#4]`

pre-index

First, copy `R1` to the empty stack space `SP` is currently pointing to before autoindexing. Need **post-index**.

✗ `STR R1,[R13,#-4]!`



`STR R1,[R13],#-4`

b

✗ `STR R1,[R13],#4`

ascending stack

24:33 / 26:22

CZ1106
CE1106

Summary

- Autoindexing modifies the indirect register besides just computing the effective address.
  - The autoindexing can use either **offset** or **index register**.
  - **Pre-index** does the autoindexing first before computing the effective address.
  - **Post-index** computes the effective address first, then does the autoindexing.
- ARM's autoindexing feature can be used to implement stacks.
  - There are **4** possible stack implementations (FD, FA, ED, EA).
  - For a given stack implementation, the Push and Pop operation must complement each other to ensure the stack grows and collapses correctly.

37

CZ1106
CE1106

38

CZ1106
CE1106

Chapter 4

Addressing Modes

PC-related Addressing Modes

Learning Objectives (4.5)

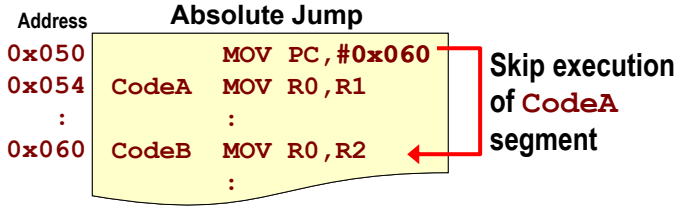
1. Describe difference between absolute & relative jump.
2. Describe the concept of position-independent code and how it is achieved.
3. Describe how data can be accessed using PC relative addressing

39

CZ1106
CE1106

Absolute Jump

- A **new address** can be loaded into the **PC** to alter the sequential order of program execution.
  - An **absolute jump** to a new code position is done by loading the address to jump to into the **PC**.
  - Example: `MOV PC,#0x060 ;Jump to CodeB`



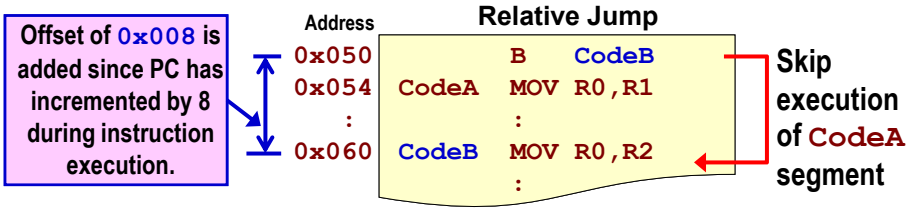
  - Absolute jump is not **position-independent**. This code can only execute correctly in this specific area of code memory.

40

CZ1106 CE1106

Relative Jump

- An **offset** can be added to the **PC** to alter the sequential order of program execution.
- A **relative jump** is done using the branch instruction (e.g. `B`) with an appropriate **signed offset.** (Note: the range of this offset in ARM is **+/- 32 Mbytes**).
- Example: `B CodeB` ; **Jump to CodeB**



- Relative jump supports **position-independent** code.

Note: In the ARM processor the PC points 8 bytes ahead of the current executed instruction due to its pipeline architecture.

41

⇒ end address unknown / unlikely absolute jump

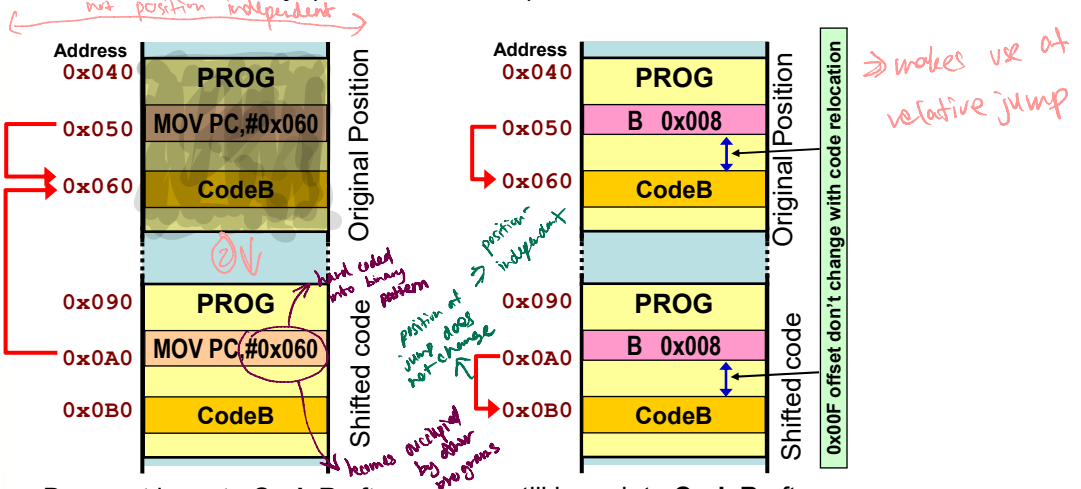
Absolute jump : "Move to 3"

Relative Jump : "Move 2 step forward"

CZ1106 CE1106

Position-Independent Code

- Such programs can be loaded anywhere in memory and still execute correctly (i.e. relocatable).



Does not jump to **CodeB** after **PROG** is shifted to `0x090`

`B` still branch to **CodeB** after **PROG** is shifted to `0x090`

42

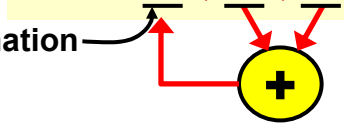
CZ1106 CE1106

ADD Instruction (Introduction)

- This instruction does the **addition** operation.
  - The 3-operand `ADD` instruction adds 2 source operands and puts result into a 3rd destination operand.
  - The right and middle operands are added and the result is placed in the destination register.

```
ADD R2,R0,R1
```

Destination



R0	0x00000003
R1	0x00000006
R2	0x00000000

Before execution

R0	0x00000003
R1	0x00000006
R2	0x00000009

After execution

  - Destination and middle operands must be registers but rightmost operand can be a **register** or an **immediate** value.

43

Key idea to achieve position independent code:
⇒ make sure that all access to your memory variables are relative to current program counter value.

CZ1106 CE1106

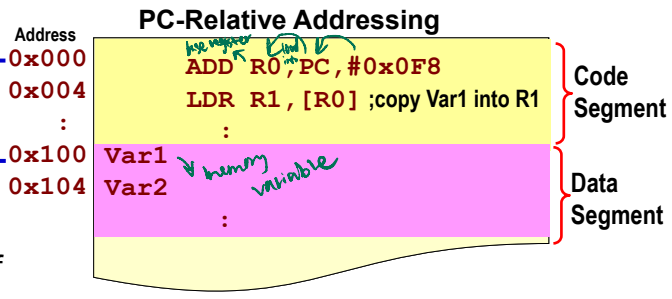
Accessing Data

- Position-independent (P-I) programs require data to be accessed relative to the **PC**.
  - **PC-relative addressing** is used to access variables in the data segment of program in memory.
  - E.g.: `ADD R0,PC,#0x0F8` **; Get P-I address of Var1 in Data Seg into R0**

PC-relative offset of `0x0F8` is added since PC has incremented by `8` when executing `ADD` instruction.

PC-relative Offset:
Var address – (PC value + 8)

0x100 – (0 + 8) = 0x0F8



  - Referencing absolute address of variable in whatever ways will violate **P-I** requirements.

Note: In the ARM processor the **PC** points **8** bytes ahead of the current executed instruction.

44

E.g. MOV R0,#0x100
MOV R1,[R0]
Using Absolute address of Memory Variable
Register indirect will not be position independent
(MOV R1,#0x100 ⇒ Immediate addressing

CZ1106
CE1106

Summary

- Non-sequential execution of code can be achieved by modifying the PC contents directly.
- The Branch instruction does this by **adding** a **signed offset**.
- Such **relative** jumps create **position-independent** code.
- **PC-relative addressing** with **appropriate offsets** allows memory data to be accessed in a position-independent manner.

Y offset needs to take into account the additional 8 bits that PC would have moved ahead.

45

CZ1106
CE1106

Quiz: Position-Independent

- Select the instructions below that can be used at label Q1, do what is in the comments and are **position-independent (P-I)**.

Address	ARM code segment
0x00	MOV R0,#0x20
0x04	Q1 ???
:	:
0x20	Jmp MOV R0,R2
:	:

`MOV R1,R0 ;copy R0 to R1`

Changing any register other than the PC is always position independent

a

Register - Register do not involve memory

X

`MOV PC,#0x20 ;jump to Jmp`

`B Jmp ;jump to Jmp`

The ARM branch instruction B used a PC relative offset to carry out the jump and is therefore position-independent

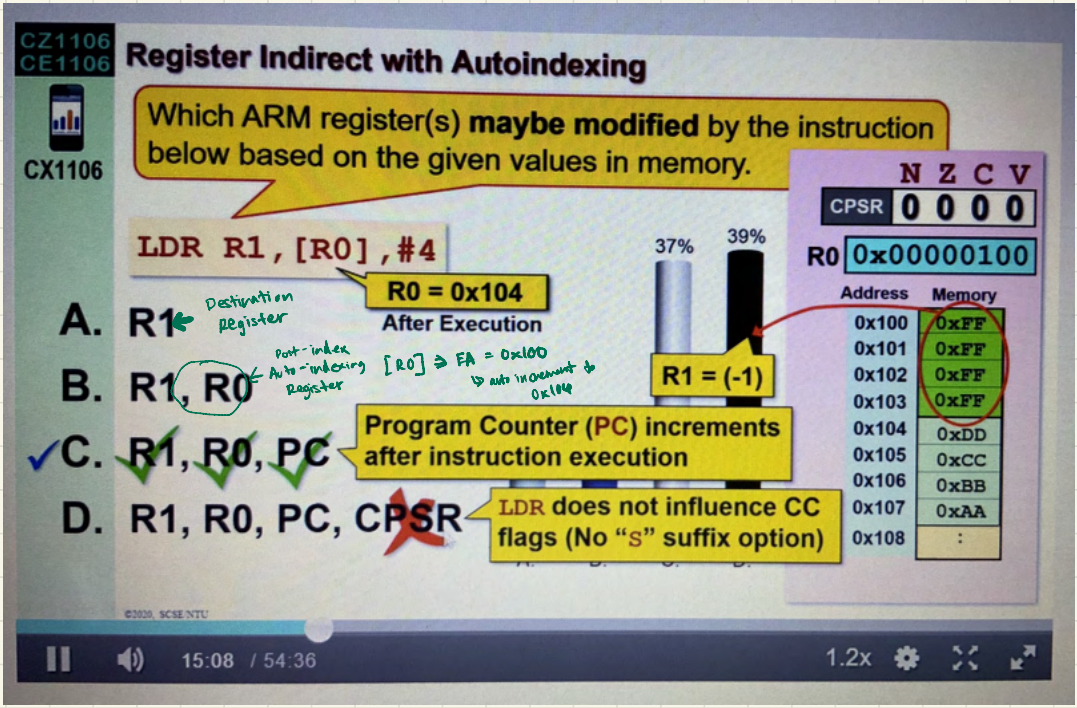
b

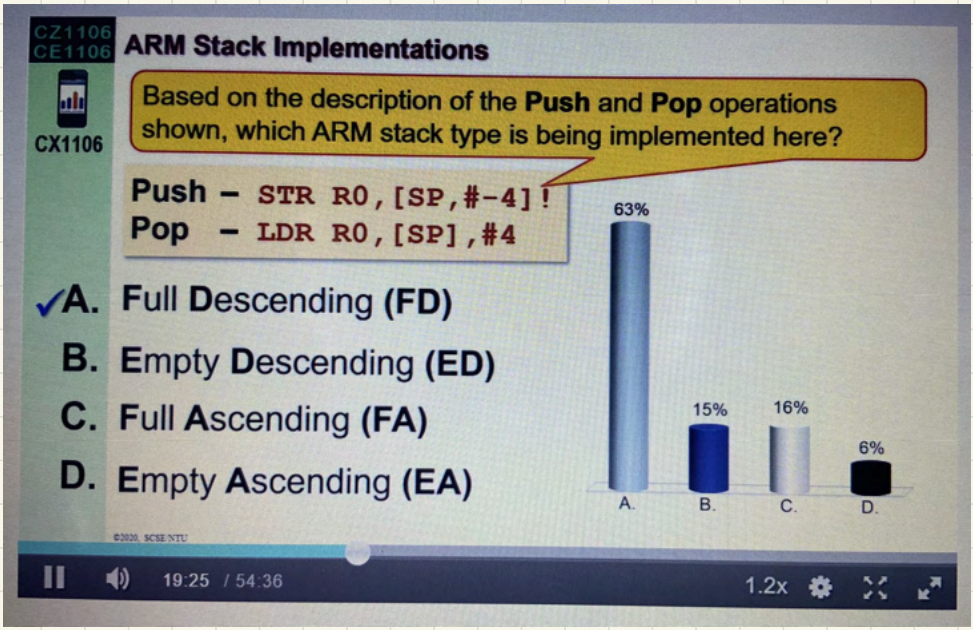
X

`MOV PC,R0 ;jump to Jmp`

Loading **PC** with the absolute start address (i.e. 0x20) of the instruction to jump to violates the P-I requirement that jump must be relative to the current **PC** value. It does not matter whether immediate or register direct is used to modify the **PC**.

Lecture (25/1)





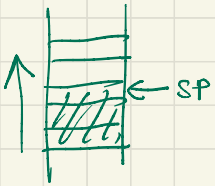
Solution Explanation next page →

(A) Full Descending:

⇒ Grows towards lower memory address

⇒ Stack Pointer pointing at very top
↳ points to a valid item

⇒ Full means you are pointing to occupied space



Push – STR R0,[SP,#-4]!
Pop – LDR R0,[SP],#4

⇒ a clue that it is descending; would be ++ if ascending

← Pre-indexing to decrement SP
⇒ to point to empty space

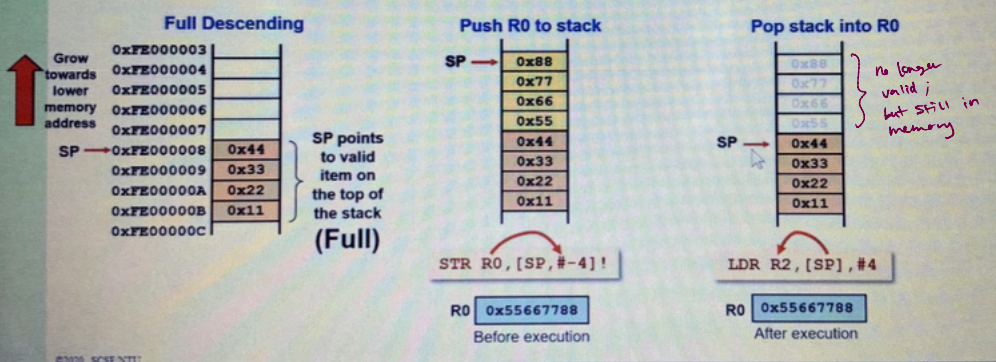
CZ1106
CE1106

ARM Stack Implementation

Push – STR R0,[SP,#-4]!
Pop – LDR R0,[SP],#4

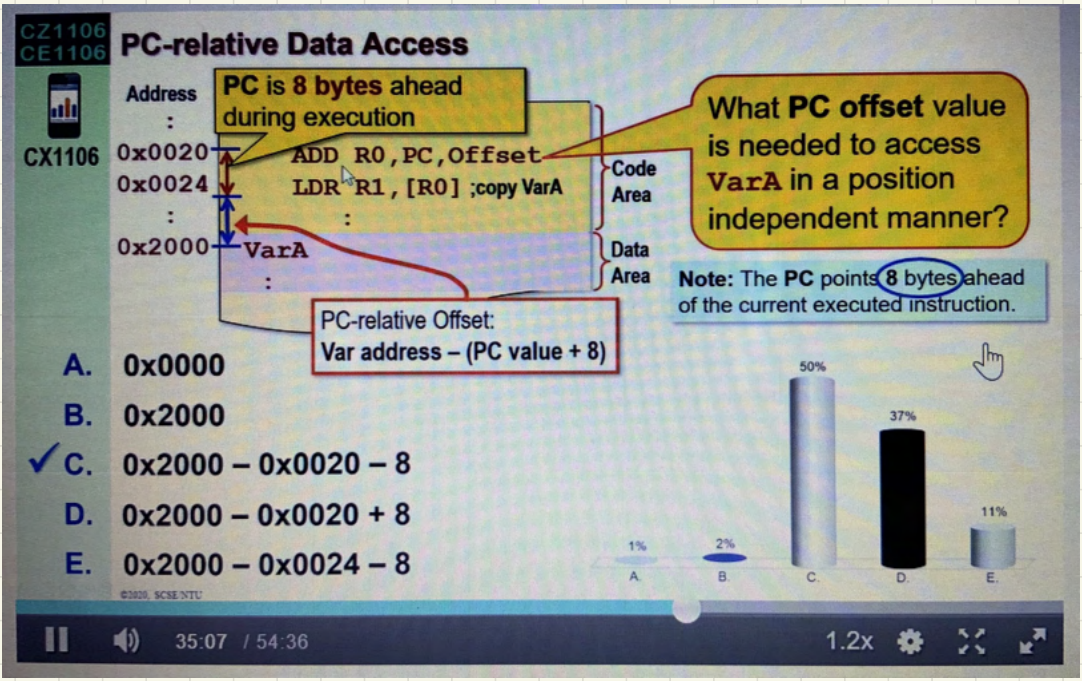
Note: the Procedure Call Standard for the ARM Architecture (**AAPCS**), and ***armcc*** always use a **full descending (FD)** stack.

A. Full Descending (FD) stack implementation:



©2020, SCSE NTU

28:33 / 54:36 1.2x



what you will be computing if 8 is ADDED

⇒ excess ∴ should SUBTRACT