ARM Assembly

Lecture 4

Microprocessor-based Systems (E155)

Prof. Josh Brake

Learning Objectives for Today

By the end of this lecture you will have…

- Refreshed your knowledge of how to translate C code into ARM assembly
- Applied your knowledge of ARM assembly to write several simple algorithms
- Learned how to mitigate the differences between ARM v7 and Thumb-2 code

Outline

- Review of ARM architecture
  - Register set
  - Memory
  - Instruction set architecture
  - Addressing modes
  - Conditional execution
  - Branching
- Translating C to ARM Assembly
- Lab 2

Warmup Quiz

https://docs.google.com/presentation/d/1ShVehgj6aX2Dr44j0UIh4JXsUJaZQ8HdsNg32T6HZcs/edit?usp=sharing

```
int a, b, c, d; // R0, R1, R2, R3
a = b + 4*c – d;
```

```
ADD R0, R1, R2,  LSL #2
SUB R0, R0, R3
```

Key questions when learning a new microprocessor

- What is the register set?
- What addressing modes are used?
- What types of instructions exist
- What does the memory map look like?
- What I/O functions are available?

Cortex-M4 Instruction Set

- Similar to ARMv4, but instructions are compressed to 16-bit Thumb instructions.
- Instructions are aligned on 2-byte (16-bit) boundaries.
- Thumb-2 instruction set architecture

ARM v7 Register Set

- 16, 32-bit registers R0-R15,
- R13: by convention, used as stack pointer
- R14: link register (holds return addresses)
- R15: serves as program counter (PC)
- Don’t use these for other things
- 4 more condition code bits N Z C V in the current program status register (CPSR)

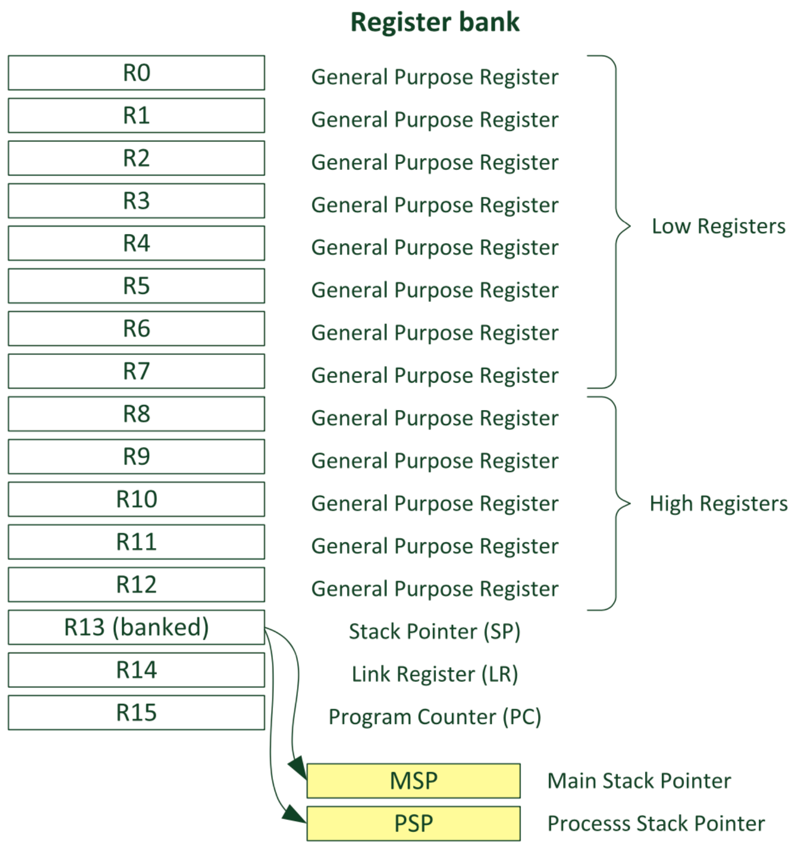


FIGURE 4.3

Registers in the register bank

Memory

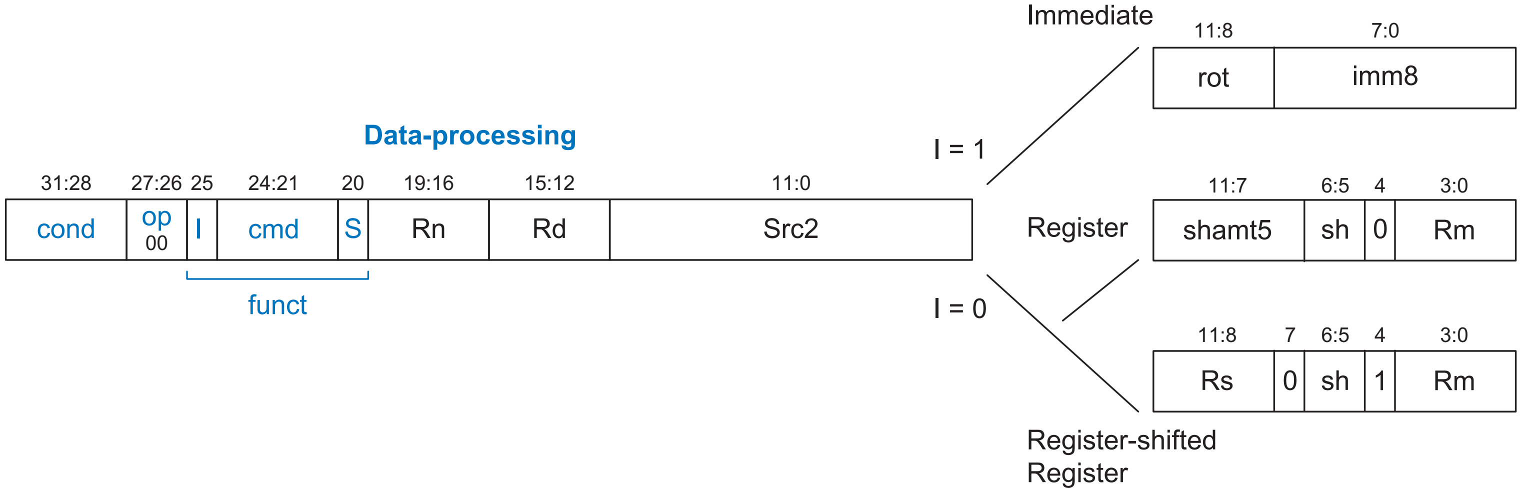
- Memory Map: 32-bit instruction set, byte accessible
- 2^{32} = 4GB of memory accessible
- Instructions are always aligned on word (4-byte) boundaries in standard ARM and halfword (2-byte) boundaries in Thumb mode

Assembly vs. Machine Language

- Assembler converts instructions from assembly mnemonics to machine code
- Line-by-line comments are critical!

```
// a = b & c
// a in R0, b in R1, C in R2
AND R2, R0, R1
```

cond = 1110 (unconditional)
I = 0 (register mode)
S = 0 (doesn't set condition codes)
Rn = R0 = 0000
Rm = R1 = 0001
Rd = R2 = 0010
Cmd = AND = 0000
Shamt5 = sh = 0
Assembles to 0xE0002001



Data processing Instructions

- ADD, SUB, ADDC, SUBC
- AND, ORR, EOR, BIC
- TST, TEQ, CMP
- MOV, MVN, LSL, LSR, ASR, ROR, RRX

cmd	Name	Description	Operation
0000	AND Rd, Rn, Src2	Bitwise AND	Rd ← Rn & Src2
0001	EOR Rd, Rn, Src2	Bitwise XOR	Rd ← Rn ^ Src2
0010	SUB Rd, Rn, Src2	Subtract	Rd ← Rn - Src2
0011	RSB Rd, Rn, Src2	Reverse Subtract	Rd ← Src2 - Rn
0100	ADD Rd, Rn, Src2	Add	Rd ← Rn+Src2
0101	ADC Rd, Rn, Src2	Add with Carry	Rd ← Rn+Src2+C
0110	SBC Rd, Rn, Src2	Subtract with Carry	Rd ← Rn - Src2 - $\bar{C}$
0111	RSC Rd, Rn, Src2	Reverse Sub w/ Carry	Rd ← Src2 - Rn - $\bar{C}$
1000 ($S = 1$)	TST Rd, Rn, Src2	Test	Set flags based on Rn & Src2
1001 ($S = 1$)	TEQ Rd, Rn, Src2	Test Equivalence	Set flags based on Rn ^ Src2
1010 ($S = 1$)	CMP Rn, Src2	Compare	Set flags based on Rn - Src2
1011 ($S = 1$)	CMN Rn, Src2	Compare Negative	Set flags based on Rn+Src2
1100	ORR Rd, Rn, Src2	Bitwise OR	Rd ← Rn | Src2
1101	Shifts:		
$I = 1$ OR ($instr_{11:4} = 0$)	MOV Rd, Src2	Move	Rd ← Src2
$I = 0$ AND ($sh = 00$; $instr_{11:4} \neq 0$)	LSL Rd, Rm, Rs/shamt5	Logical Shift Left	Rd ← Rm << Src2
$I = 0$ AND ($sh = 01$)	LSR Rd, Rm, Rs/shamt5	Logical Shift Right	Rd ← Rm >> Src2

cmd	Name	Description	Operation
$I = 0$ AND ($sh = 10$)	ASR Rd, Rm, Rs/shamt5	Arithmetic Shift Right	Rd ← Rm>>>Src2
$I = 0$ AND ($sh = 11$; $instr_{11:7, 4} = 0$)	RRX Rd, Rm, Rs/shamt5	Rotate Right Extend	{Rd, C} ← {C, Rd}
$I = 0$ AND ($sh = 11$; $instr_{11:7} \neq 0$)	ROR Rd, Rm, Rs/shamt5	Rotate Right	Rd ← Rn ror Src2
1110	BIC Rd, Rn, Src2	Bitwise Clear	Rd ← Rn & ~Src2
1111	MVN Rd, Rn, Src2	Bitwise NOT	Rd ← ~Rn

Data Processing Addressing Modes

`Source1 (Rn)` and destination are always a register

`Src2` can be register or immediate. Registers can be shifted by a constant or another register.

- Immediates are 8 bits, optionally rotated right by any multiple of two
- Registers can be shifted 4 ways (LSL, LSR, ASR, ROR) by 5-bit constant or a register

Operation		**§**	**Assembler**	**S updates**	**Action**
Add	Add		ADD{S} Rd, Rn, <Operand2>	N Z C V	Rd := Rn + Operand2

Data Processing Addressing Modes: Examples

Operation		§	Assembler	S updates	Action
Add	Add		ADD{S} Rd, Rn, <Operand2>	N Z C V	Rd := Rn + Operand2

Flexible Operand 2	
Immediate value	#<imm8m>
Register, optionally shifted by constant (see below)	Rm {, <opsh>}
Register, logical shift left by register	Rm, LSL Rs
Register, logical shift right by register	Rm, LSR Rs
Register, arithmetic shift right by register	Rm, ASR Rs
Register, rotate right by register	Rm, ROR Rs

Register, optionally shifted by constant		
(No shift)	Rm	Same as Rm, LSL #0
Logical shift left	Rm, LSL #<shift>	Allowed shifts 0-31
Logical shift right	Rm, LSR #<shift>	Allowed shifts 1-32
Arithmetic shift right	Rm, ASR #<shift>	Allowed shifts 1-32
Rotate right	Rm, ROR #<shift>	Allowed shifts 1-31
Rotate right with extend	Rm, RRX	

Ex: `int a in R0, b in R1, c in R2, d in R3`

```
a = b + c;          ADD R0, R1, R2
a = b + 5;          ADD R0, R1, #5
a = b + 4*c;        ADD R0, R1, R2, LSL #2
a = b + c >> d;     ADD R0, R1, R2, ASR R3
```

ARM® and Thumb®-2 Instruction Set Quick Reference Card - QRC0001_UAL (on course website)

Condition Codes

- Z: result is zero
- N: result is negative (msb = 1)
- C: adder produces a carry out
- V: adder overflows

- Data processing instructions come with S variant to set the condition codes based on the result. Not used much.
- CMP, CMN, TST, TEQ all need S bit set (but we don't write it in the name)

- Stored in Current Program Status Register (CPSR)

CMP, CMN, TST, TEQ

- CMP is SUBS but the result is not written
- CMN is ADDS but the result is not written
- TEQ is EORS but the result is not written
- TST is ANDS but the result is not written

Conditional Execution in ARM v7

Do a TST or CMP, then make the next instruction conditional

```
C Snippet
if (a == b) c = d + 3;

ARM Assembly
TEQ R0, R1
ADDEQ R2, R3, #3
```

Conditions are often used with branches, but also allow a short bit of code to be executed without branch

Can't do this in Thumb-2

cond	Mnemonic	Name	CondEx
0000	EQ	Equal	Z
0001	NE	Not equal	$\overline{Z}$
0010	CS/HS	Carry set / unsigned higher or same	C
0011	CC/LO	Carry clear / unsigned lower	$\overline{C}$
0100	MI	Minus / negative	N
0101	PL	Plus / positive or zero	$\overline{N}$
0110	VS	Overflow / overflow set	V
0111	VC	No overflow / overflow clear	$\overline{V}$
1000	HI	Unsigned higher	$\overline{Z}C$
1001	LS	Unsigned lower or same	Z OR $\overline{C}$
1010	GE	Signed greater than or equal	$\overline{N \oplus V}$
1011	LT	Signed less than	$N \oplus V$
1100	GT	Signed greater than	$\overline{Z}(\overline{N \oplus V})$
1101	LE	Signed less than or equal	Z OR $(N \oplus V)$
1110	AL (or none)	Always / unconditional	Ignored

Conditional Execution on Thumb-2

```
if (a) b = 1;
// ARM v7
CMP R0, #0
MOVNE R2, #1
```

But we can’t use MOVNE in ARM Thumb-2. So what do we do?

- Branch
- Use IT statement. Only in Thumb-2.

```
// ARM Thumb-2
CMP R0, #0
IT NE
MOVNE R2, #1
```

Conditional Execution on Thumb-2

Summary

- Can either use an "if-then" (IT) instruction or a conditional branch
- IT blocks
  - Can handle up to 4 instructions
  - Condition codes are not set by instructions in the IT block
  - Why do this instead of branching? Avoid branching penalties.

Conditional Execution on Thumb-2: IT Statements

<u>Syntax of IT Statements</u>

IT<x><y><z><cond>

- <x>, <y>, and <z> are optional and must be either T (then) or E (else).
- <cond> is required and must reflect one of the condition codes that are related to the bits in the Application Program Status Register (APSR).
- Else conditions must be the opposite of the if conditions.

IT Statement Example

C Pseudocode

```
if (R4 == R5)
{
  R7 = R8 + R9;
  R7 /= 2;
}
else
{
  R7 = R10 + R11;
  R7 *= 2;
}
```

```
CMP R4, R5
ITTEE EQ
ADDEQ R7, R8, R9    // if R4 = R5, R7 = R8 + R9
ASREQ R7, R7, #1    // if R4 = R5, R7 /= 2
ADDNE R7, R10, R11  // if R4 != R5, R7 = R10 + R11
LSLNE R7, R7, #1    // if R4 != R5, R7 *=2
```

Branches

- B - branch
- BL – branch and link, saves PC+2/4 in link register LR
- BX/BLX – branch/branch and link + exchange instruction set (from ARM to Thumb mode or vice versa)

Memory Addressing

LDR, STR, LDRB, STRB, LDRSB

Rd is destination for loads, source for stores

Rn is base address

Src2 is offset. Can be 12-bit immediate or register with optional constant shift

Let a hold the base address of an array of unsigned bytes:

C Snippet

```
unsigned char a[32];        // a in R0
unsigned char b;            // b in R1

b = a[6];
```

ARM Assembly

```
LDRB R1, [R0, #6]
```

C Snippet

```
int a[40];    // a in R0
int b, c;     // b in R1, C in R2

b = a[c];
```

ARM Assembly

```
LDR R1, [R0, R2, LSL #2]
```

Translating C snippets to assembly: Arithmetic

```
int a, b, c, d; // R0, R1, R2, R3
```

C Snippet	ARM Assembly
`a = b + c;`	`ADD R0, R1, R2`

C Snippet	ARM Assembly
`a = b + 2*c – d;`	`ADD R0, R1, R2,  LSL #1` `SUB R0, R0, R3`

C Snippet	ARM Assembly
`a = d / 4;`	`ASR R0, R3, #2  // would be LSR if D were unsigned`

Translating C snippets to assembly: Logical

```
int a, b, c, d; // R0, R1, R2, R3
```

C Snippet	ARM Assembly
`a = b & c;`	`AND R0, R1, R2`

C Snippet	ARM Assembly
`a = b | c;`	`ORR R0, R1, R2`

C Snippet	ARM Assembly
`a = b ^ c;`	`EOR R0, R1, R2`

Translating C snippets to assembly: Shift

```
int a, b, c, d; // R0, R1, R2, R3
```

C Snippet	ARM Assembly
`a = b << 4;`	`LSL R0, R1, #4`

C Snippet	ARM Assembly
`a = b >> c;`	`ASR R0, R1, R2`

Translating C snippets to assembly: If

`int a, b, c, d; // R0, R1, R2, R3`

C Snippet	ARM Assembly

`if (a != b) c = d;`	`TEQ R0, R1`
	`IT NE`
	`MOVNE R3, R4`

C Snippet	ARM Assembly

`if (a) c = 3;`	`CMP R0, #0`
	`IT NE`
	`MOVNE R3, #3`

C Snippet	ARM Assembly

`If (a <= b) { do stuff 1}`	`CMP R0, R1`
	`BGT around`
	`// stuff 1 goes here`
	`around:`

Translating C snippets to assembly: If

```
int a, b, c, d; // R0, R1, R2, R3
```

C Snippet	ARM Assembly

if (a > b) { do stuff 1}
else { do stuff 2}
	CMP R0, R1
	  BLE else
	  // stuff1 goes here
	  B done
	else:
	  // stuff2 goes here
	done:

Translating C snippets to assembly: If

`int a, b, c, d; // R0, R1, R2, R3`

C Snippet	ARM Assembly

`if ( a > b) c = 1;`
`else c = 0;`
	`// ARM Thumb-2`
	`  CMP R0, R1`
	`  ITE GT`
	`  MOVGT R2, #1`
	`  MOVLE R2, #0`

Translating C snippets to assembly: For

```
int sum; // R0
int i; // R1
```

C Snippet	ARM Assembly

`sum = 0;`
`for (i=0; i<10; i++)  sum = sum + i;`
	`  MOV R0, #0`
	`  MOV R1, #0`
	`loop: CMP R1, #10`
	`  BGE done`
	`  ADD R0, R0, R1`
	`  ADD R1, R1, #1`
	`  B LOOP`
	`done:`

Translating C snippets to assembly: For

```
int i, j; // R1, R2
int q;  // R3
```

<table>
<tr><th>C Snippet</th><th>ARM Assembly</th></tr>
<tr><td><pre>for (i=2; i<8; i++)
  for (j=1; j<i; j++)
    q = q + i – j;</pre></td><td><pre> MOV R1, #2
loopi:
  CMP R1, #8
  BGE donei
  MOV R2, #1
loopj:
  CMP R2, R1
  BGE donej
  ADD R3, R3, R1
  SUB R3, R3, R2
  ADD R2, R2, #1
  B loopj
donej:
  ADD R1, R1, #1
  B loopi
donei:</pre></td></tr>
</table>

Translating C snippets to assembly: For

`unsigned int a1[20], a2[20]; // in R4, R5`

C Snippet	ARM Assembly

`for (i=0; i<20; i++) a1[i] = a2[i] / 2;`	`  MOV R1, #0`
	`loop:`
	`  CMP I, #20`
	`  BGE done`
	`  LDR R6, [R4, R1, LSL #2]`
	`  LSR R6, R6, #1`
	`  STR R6, [R5, R1, LSL #2]`
	`  ADD R1, R1, #1`
	`  B loop`
	`done:`

Translating C snippets to assembly: While

```
unsigned int a1[20], a2[20]; // in R4, R5
```

C Snippet	**ARM Assembly**

`int i = 1;`
`int j = 0;`
 	
`while (i <= 2048) {`
`  a1[j++] = i;`
`  i = i * 2;`
`}`
	`  MOV R1, #1`
	`  MOV R2, #0`
	`while:`
	`  CMP R1, #2048 // legal because this can be represented`
	`  BGT done`
	`  STR R1, [R4, R2]`
	`  ADD R2, R2, #1`
	`  LSL R1, R1, #1`
	`  B while`

Translating C snippets to assembly: String Copy

```
char str1[64], str2[64]; // R4, R5
```

C Snippet	ARM Assembly

`int i = 0;`
`do {`
`  str2[i] = str1[i];`
`} while (str1[i++]);`
	`  MOV R1, #0`
	`do:`
	`  LDRB R6, [R4, R1]`
	`  STRB R6, [R5, R1]`
	`  CMP R6, #0`
	`  ADD R1, R1, #1`
	`  BNE do`

Lab 2 Starter Code

Lab 2 Linker Script

```
SECTIONS
{
. = 0x08000000; /* From 0x08000000, the start of flash memory */
/* This address is mapped to 0x00000000 because */
/* BOOT0 = 0 (p. 41 RM0368) */

.text :
{
*(vectors) /* Vector table */
*(.text) /* Program code */
}
.rodata :
{
*(.rodata) /* Read only data */
}
_DATA_ROM_START = .; /* Store the current location as _DATA_ROM_START */
```

sort_linker.ld

Lab 2 Linker Script continued

```
. = 0x20000000; /* From 0x20000000 */

_DATA_RAM_START = .; /* Store the current location as _DATA_RAM_START */
.data : AT(_DATA_ROM_START)
{
*(.data) /* Data memory */
}
_DATA_RAM_END = .;

_BSS_START = .; /* Indicates where BSS section starts in RAM */
.bss :
{
*(.bss) /* Zero-filled run time allocate data memory */
}
_BSS_END = .; /* Indicates where BSS section ends in RAM */
}
```

sort_linker.ld

Lab 2 Startup Code

```
extern unsigned int _DATA_ROM_START;
extern unsigned int _DATA_RAM_START;
extern unsigned int _DATA_RAM_END;
extern unsigned int _BSS_START;
extern unsigned int _BSS_END;

#define STACK_TOP 0x20018000 // 96 KB of
void startup();

/* Define minimal vector table. First entry is the address of the top of the
* stack and the second one is the address of the "reset handler" function
*/
unsigned int * myvectors[2]
/* __attribute (section("section-name")) makes sure that this gets assembled
* into a section with the name "vectors". This section label is used later
* in our linker script to make sure these get put in the right spot.
*/
__attribute__ ((section("vectors")))= {
(unsigned int *) STACK_TOP, // stack pointer
(unsigned int *) startup // code entry point
};

void main(); // Function prototype declaration for sort function
```

startup.c

Lab 2 Startup Code

```c
void startup() {
/* Copy data belonging to the `.data` section from its
* load time position on flash (ROM) to its run time position
* in SRAM. */
unsigned int * data_rom_start_p = &_DATA_ROM_START;
unsigned int * data_ram_start_p = &_DATA_RAM_START;
unsigned int * data_ram_end_p = &_DATA_RAM_END;

while(data_ram_start_p != data_ram_end_p) {
  *data_ram_start_p = *data_rom_start_p;
  data_ram_start_p++;
  data_rom_start_p++;
}

/* Initialize data in the `.bss` section to zeros. */
unsigned int * bss_start_p = &_BSS_START;
unsigned int * bss_end_p = &_BSS_END;

while(bss_start_p != bss_end_p) {
 *bss_start_p = 0;
bss_start_p++;
}
/* Call the `main()` function defined in `sort.s`.*/
main();
}
```

startup.c

Lab 2 Starter Code

```
// sort.s
// Main sort function template
// jbrake@hmc.edu
// 6/23/20

// Directives
.syntax unified // Specify the syntax for the file
.cpu cortex-m4 // Target CPU is Cortex-M4
.fpu softvfp // Use software libraries for floating-point operations
.thumb // Instructions should be encoded as Thumb instructions

// Define main globally for other files to call
.global main
```

Lab 2 Starter Code

```
// Create test array of bytes. Change this for different test cases.
// This will get loaded to the RAM by the startup code (address 0x20000000)
.data
arr:
    .byte 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1
.size arr, .-arr

.text
// The main function
.type main, %function
main:
    ldr r3, =arr // Load the base address of RAM where the array is stored
    // YOUR CODE HERE

done:
     b done
.size main, .-main
```

Summary

By the end of this lecture you will have…

- Refreshed your knowledge of how to translate C code into ARM assembly
- Applied your knowledge of ARM assembly to write several simple algorithms
- Learned how to mitigate the differences between ARM v7 and Thumb-2 code

Lecture Feedback

- What is the most important thing you learned in class today?
- What point was most unclear from lecture today?

https://forms.gle/9K9dAhdf61iunoEu7

Feedback