Chapter 8: Instruction Encoding and ISA

Mohamed M. Sabry Aly
N4-02c-92

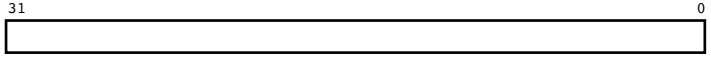
Learning Objectives

- Describe the instruction format of ARM instruction-set architecture (ISA)
- Describe differences between fixed and variable-length ISA

Instruction Encoding

1 instruction = 32 bits

- For CPU to identify each unique assembly instruction, they must be encoded with unique binary patterns.
  - ARM 32-bit machine encode instructions in a 32-bit word
- What is included?
  - Instruction type ➔ encoded
  - Operand(s) ➔ encoded
  - Condition for conditional execution ➔ encoded
  - Other flags?

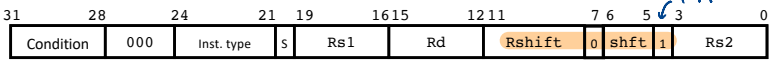


Overall Instruction Format

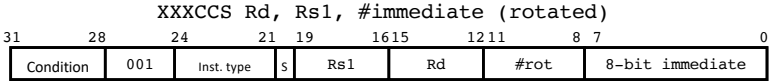
- Arithmetic & Logic instructions
  - ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC

XXXCCS Rd, Rs1, Rs2 , {shft #}

XXXCCS Rd, Rs1, Rs2 , {shft Rshift}



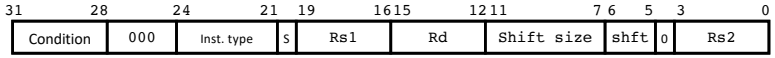
XXXCCS Rd, Rs1, #immediate (rotated)



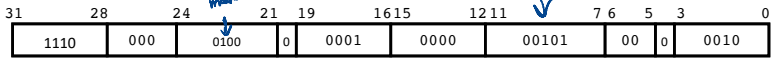
Overall Instruction Format

- Arithmetic & Logic instructions
  - ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC

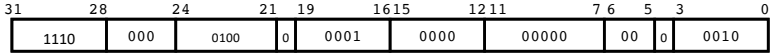
XXXCCS Rd, Rs1, Rs2 , {shft #}



ADD R0,R1,R2,LSL #5

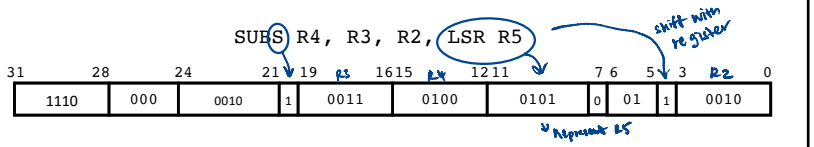


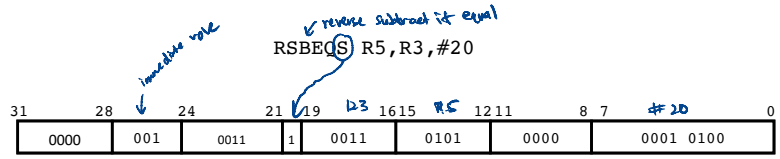
ADD R0,R1,R2



Overall Instruction Format

- Arithmetic & Logic instructions
  - ADD, ADC, SUB, SBC, RSB, AND, EOR, RSC, ORR, BIC

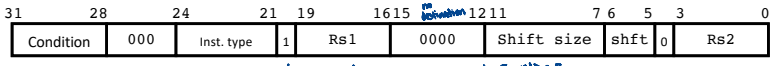




Overall instruction format

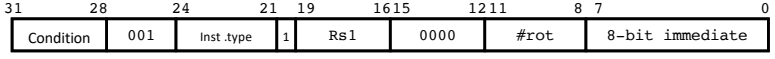
- Comparison instructions
  - TST, TEQ, CMP, CMN

XXXCC Rs1, Rs2 , {shft #}



Always 1, as comparison instructions are meant to set the status flag by default

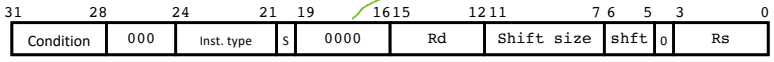
XXXCC Rs1, #immediate (rotated)



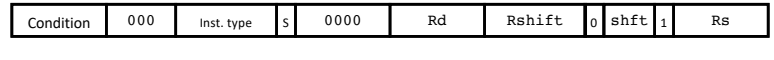
Overall instruction format

- MOV instructions
  - MOV, MOVN

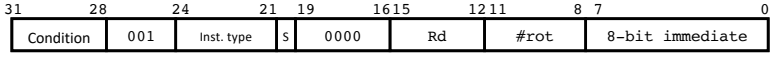
XXXCCS Rd, Rs , {shft #}



XXXCCS Rd, Rs , {shft #}



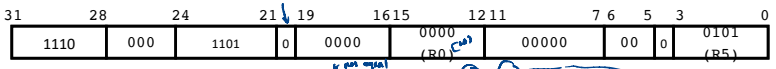
XXXCC Rd, #immediate (rotated)



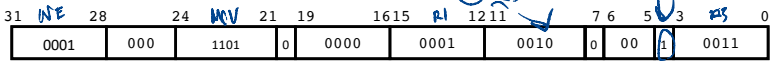
Overall instruction format

- MOV instructions
  - MOV, MOVN

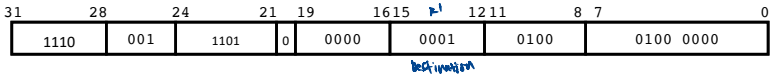
```
MOV   R0,R5
```




```
MOVNE R1, R3, LSL R2
```




```
MOV   R1,#0x40000000
```



Inst. Type field Field

- 4 bits for instructions ➔ up to 16 combinations
- 4 bits for source and destination registers ➔ 16 registers

Code	Instruction
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC

Code	Instruction
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

Condition Field

- 4 bits for condition ➔ up to 16 difference combinations

Code	Condition	Flags	Meaning
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS or HS	C = 1	Higher or same, unsigned
0011	CC or LO	C = 0	Lower, unsigned
0100	MI	N = 1	Negative
0101	PL	N = 0	Positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 and Z = 0	Higher, unsigned
1001	LS	C = 0 or Z = 1	Lower or same, unsigned
1010	GE	N = V	Greater than or equal, signed
1011	LT	N != V	Less than, signed
1100	GT	Z = 0 and N = V	Greater than, signed
1101	LE	Z = 1 and N != V	Less than or equal, signed
1110	AL		Always.
1111	NV		Reserved (unused)

What about shift instructions, what's there instruction format?

LSL, LSR, ASR, ROR, and RRX has no dedicated instructions

Shift/Rotate instructions

- Instructions are encoded as MOV instructions

  E.g. LSL R1, R2, #5 ≡ MOV R1, R2, LSL #5

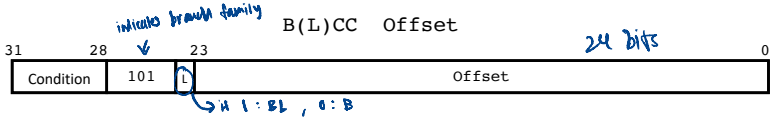
- Specify the shift type in bits 5 and 6

Code	Shift type
00	LSL
01	LSR
10	ASR
11	ROR/RRX

rely on use of MOV for shift

Branch Instructions

- For conditional branch and branch with link



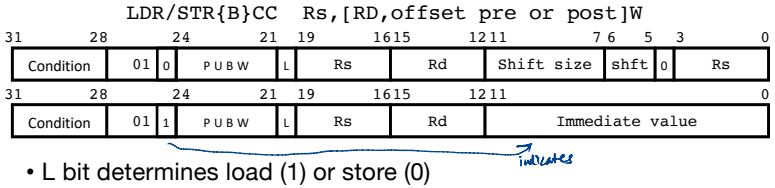
- 26 bits offset shifted to the right by 2 (branching to word addresses) ➔ calculated by the assembler

  ↓ bytes not set, multiply by 4

- Branch range: ±32 MBytes

Load/Store Instructions

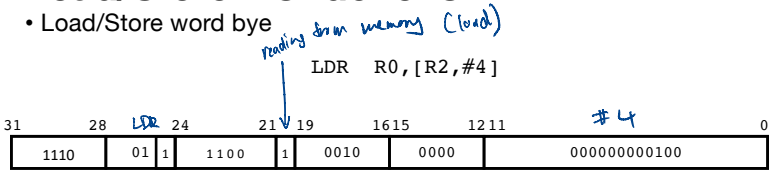
- Load/Store word bye



- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset

Load/Store Instructions

- Load/Store word bye



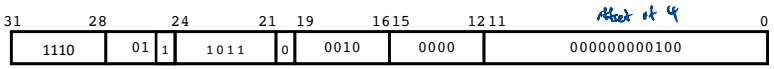
- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset (0, no write back)

LDR / STR : 011

Load/Store Instructions

- Load/Store word bye

```
STRB R0,[R2,#-4]!
```



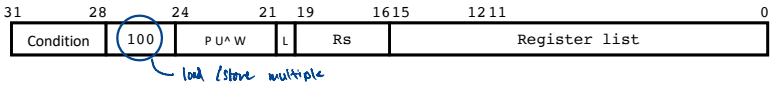
offset of 4

- L bit determines load (1) or store (0)
- P bit determines indexing pre (1) or post (0)
- U determines offset direction add (1) or subtract (0)offset
- B determines byte (1) or word (0) access
- W is for write back (!) of the offset (1, auto-indexing)

Load/Store Instructions

- Load/Store Multiple instructions

```
LDM/STMFXCC  RsW,{register list}
```



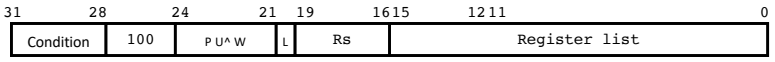
load /store multiple

- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (0)
- U determines offset direction add (1) or subtract (0)
- ^ is “don’t care” → don't care if its 1 or 0, as no bit transfer only words
- W is for write back (!) after operation

Load/Store Instructions

- Load/Store Multiple instructions

```
LDM/STMFXCC  RsW,{register list}
```



- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (
- U determines offset direction add (1) or subtr
- ^ is “don’t care”
- W is for write back (!) after operation

PU	Instruction
10	STMFD
01	LDMFD
00	STMFA
11	LDMFA

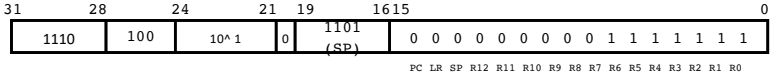
- fully calculable offset and do subtraction
- move down and do addition

Load/Store Instructions

- Load/Store Multiple instructions

order does not matter

```
STMFD SP!,{R0-R6}
```



- L bit determines load (1) or store (0)
- P bit determines indexing before (1) or after (0)
- U determines offset direction add (1) or subtract (0)
- ^ is “don’t care”
- W is for write back (!) after operation

Not in Pre-lec Vids (Mentioned in Lec)

Fixed vs. Variable-Length instruction

- ARM is fixed-length
  - Operands had to be registers or immediate values
- Variable length instructions
  - Intel 80x86: Instructions vary from 1 to 17 bytes long.
  - Digital VAX: Instructions vary from 1 to 54 bytes long.
  - Require multi-step fetch and decode.
  - Allow for a more flexible (but complex) and compact instruction set.

Variable-Length: Intel Instruction Format

- Complex instructions = complex hardware
- Higher variety for smaller code and faster execution

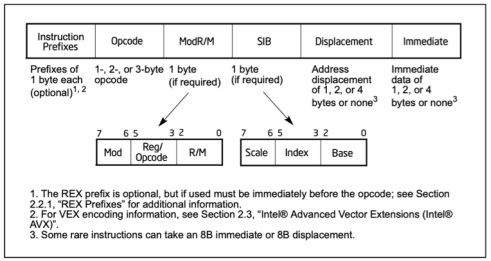
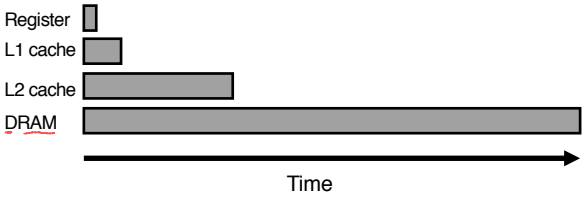


Figure 2-1. Intel 64 and IA-32 Architectures Instruction Format

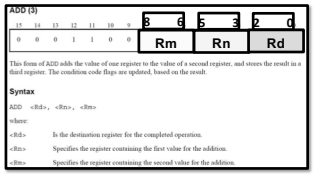
Using Registers

- Operations using registers are **faster** than those involving memory.
  - Data transfer between registers and ALU is faster due to it physically proximity.
  - To speed up code execution, keep as much of your computations within **registers**.

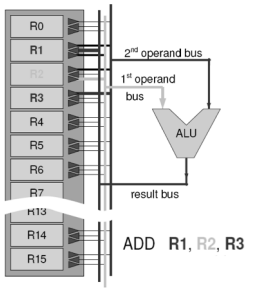


Implication of Register Count

- Implementing many registers within the CPU will incurs increasing overheads.
  - They take up precious **space** in the silicon die.
  - Connections to various busses increase the **routing** and **multiplexing complexity**.
  - More registers increases the **operand size** during instruction encoding.



Instruction Encoding for ADD in Thumb-2



No. of pins relate to memory size not no. of registers

Orthogonality of ISA

- In a truly orthogonal ISA, every instruction is able to use every available addressing modes.
- A truly orthogonal ISA does not restrict certain instruction from using only specific registers.
- Difficult to achieve complete orthogonality due to the **large number of bit patterns** required to express all combinations of operations and addressing modes.
- Increase instruction length will increases the memory size required by the program.

25

Summary - Why is Good Instruction Set Architecture (ISA) Design Important?

- Good ISA design can yield benefits such as:
  - Increases the execution performance of the processor.
  - Increases the code density (i.e. how much memory a piece of code will occupy).
  - Reduces the power consumption of the CPU.
  - Reduces manufacturing cost of processor.
  - Easy for programmers to write efficient programs.
  - Efficient mapping of high-level programming requirements into low-level instruction sets.

26

Why Should I Care About Processor Designs and ISA?

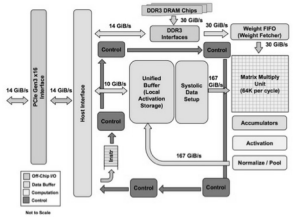
- Designing a processor (or a co-processor) is no longer a hardware company job

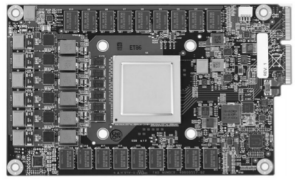
Why Should I Care About Processor Designs and ISA?

- Designing a processor (or a co-processor) is no longer a hardware company job

The Google TPU

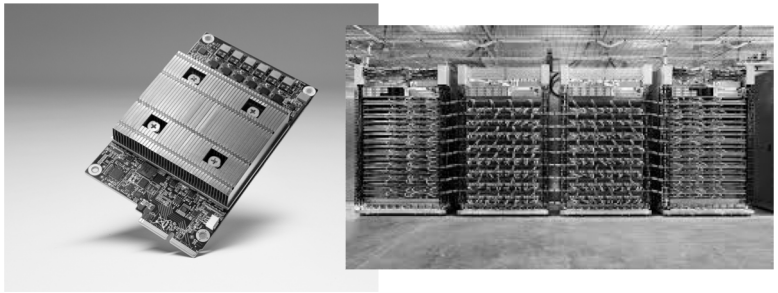
- Tensor Processing Unit
- Designed by Engineers in Google
- Tailored to make AI applications run faster





The Google TPU

- Design by Engineers in Google to make AI applications run faster



Summary

- ARM is fixed-length load/store architecture
  - Operands reside in registers
- Other architectures can use variable-length format
  - E.g, Intel architecture
- Use of registers is key towards higher performance

26/2 Lecture

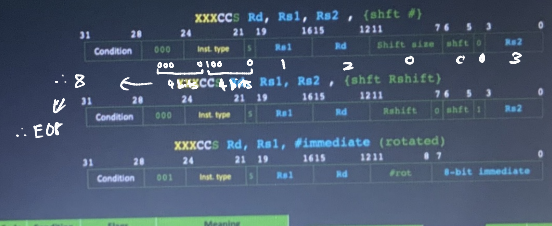
Q1 which of the following hexadecimal encoding of ADD R2,R1,R3

1. 0xE0812003

2. 0xF1021003

3. 0XE0931200

4. 0x00012003



Code	Condition	Flags	Meaning
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS or HS	C = 1	Higher or same, unsigned
0011	CC or LO	C = 0	Lower, unsigned
0100	MI	N = 1	Negative
0101	PL	N = 0	Positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 and Z = 0	Higher, unsigned
1001	LS	C = 0 or Z = 1	Lower or same, unsigned
1010	GE	N = V	Greater than or equal, signed
1011	LT	N != V	Less than, signed
1100	GT	Z = 0 and N = V	Greater than, signed
1101	LE	Z = 1 and N != V	Less than or equal, signed
1110	AL		Always
1111	NV		Reserved (unused)

Code	Instruction
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC

Code	Instruction
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

Addresses of registers:

R0 – 0000

R1 – 0001

R2 – 0010

R3 – 0011

⋮

ADD – no condition

⇒ does not mean condition = 0000

↳ cause 0000 indicates EQ

⇒ So must be condition = 1110

↳ aka (E), indicates "Always"

XXXCCS Rd, Rs1, Rs2 , {shft #}

31–28	27–24	23–21	20	19–16	15–12	11–7	6–5	4	3–0
Condition	000	Inst. type	S	Rs1	Rd	Shift size	shft	0	Rs2
1110	000	0100	0	1	2	0	0	0	3

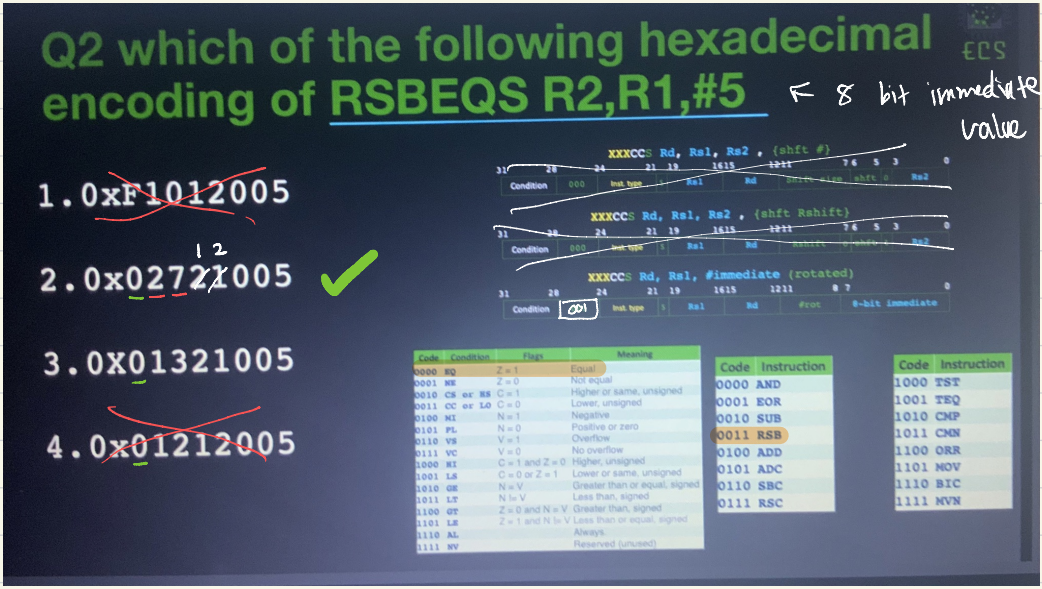
1110 aka E, "always"

0100 ADD code

S = 0 cause it is ADD, not ADDS

shift size = 0, shft = 0 → no shifts

3



0000 001 0011 1 1 2 5

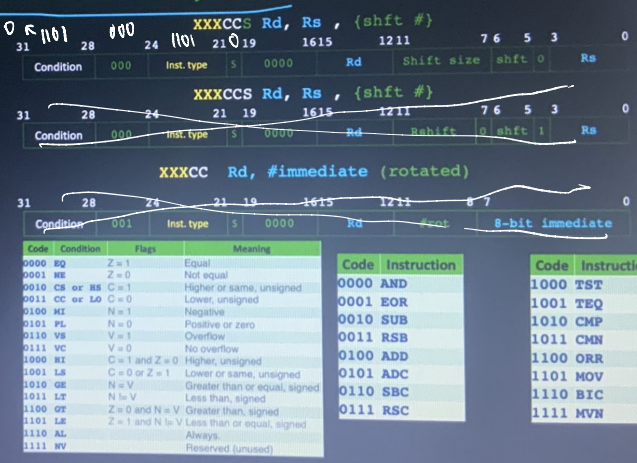
XXXCCS Rd, Rs1, #immediate (rotated)

31–28	27–24	23–21	20	19–16	15–12	11–8	7–0
Condition	001	Inst. type	S	Rs1	Rd	#rot	8-bit immediate

000 if 3rd operand is a register

Q3 which of the following hexadecimal encoding of MOVLE R7,R4

1. 0xD0D74000
2. 0xD0D70004
3. 0XD1A70004 ✓
4. 0xD1A74000

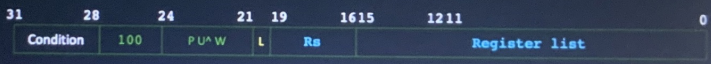


Code	Condition	Flags	Meaning
0000	EQ	Z = 1	Equal
0001	NE	Z = 0	Not equal
0010	CS or HS	C = 1	Higher or same, unsigned
0011	CC or LO	C = 0	Lower, unsigned
0100	MI	N = 1	Negative
0101	PL	N = 0	Positive or zero
0110	VS	V = 1	Overflow
0111	VC	V = 0	No overflow
1000	HI	C = 1 and Z = 0	Higher, unsigned
1001	LS	C = 0 or Z = 1	Lower or same, unsigned
1010	GE	N = V	Greater than or equal, signed
1011	LT	N != V	Less than, signed
1100	GT	Z = 0 and N = V	Greater than, signed
1101	LE	Z = 1 and N != V	Less than or equal, signed
1110	AL		Always
1111	NV		Reserved (unused)

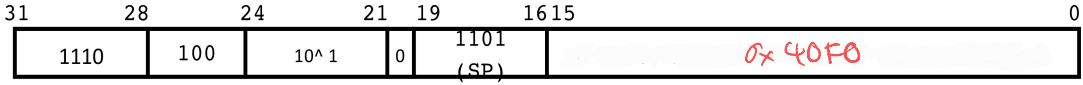
Code	Instruction
0000	AND
0001	EOR
0010	SUB
0011	RSB
0100	ADD
0101	ADC
0110	SBC
0111	RSC

Code	Instruction
1000	TST
1001	TEQ
1010	CMP
1011	CMN
1100	ORR
1101	MOV
1110	BIC
1111	MVN

Q4 If the registers list field is encoded with 0X40F0, what register list this represents?



1. {R4,R15}
2. {R4-R7,LR}
3. {R1,R8-R11}
4. Don't know!



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

0 1 0 0 0 0 0 0 1 1 1 1 0 0 0 0

R4 - R7 , LR