ARMv7 (DE1-SoC / CPUlator) চিট-শিট — লাইনে লাইন সহজ উদাহরণ (বাংলায়)

চিট-শিটটি ARMv7 (Cortex-A9) জন্য সাধারণ ভাবে সবচেয়ে বেশি ব্যবহার হওয়া ইনস্ট্রুকশন, অ্যাড্রেসিং মুড ও ব্যবহারের টিপস নিয়ে। প্রতিটি লাইনের পাশে ছোট উদাহরণ ও টীকা আছে— CPUlator/DE1-SoC এ কাজ করার সময় যে ছোট নিয়মগুলো মনে রাখবেন সেগুলোও দিলাম।

সাধারণ নোট (CPUlator-specific)

- বড় 32-bit constants সরাসরি MOV Rn, #0xC8000000 করা যাবে না — 대신 LDR Rn, =0xC8000000 ব্যবহার করুন।
- কিন্তু LDR =symbol নিয়ে সরাসরি symbol-Arithmetic করলে assembler *ABS* and *UND* error দিতে পারে। ঠিকানা গণনা করা হলে চেষ্টা করুন **register math only** (LDR + register ops)।
- Memory-mapped peripherals সাধারণত 32-bit বা 16-bit; VGA=16-bit(pixel), HEX/LED=32-bit 등 ডেটাশিট/DE1 manual দেখুন।

Data processing (আরও দরকারী ইনস্ট্রাকশন)

MOV — রেজিস্টারে immediate বা register কপি

```
MOV R0, #5       @ R0 = 5
MOV R1, R0       @ R1 = R0
```

MVN — bitwise NOT of operand

```
MVN R0, R1       @ R0 = ~R1
```

ADD, SUB — যোগ-বিয়োগ

```
ADD R2, R0, R1   @ R2 = R0 + R1
SUB R2, R0, #1   @ R2 = R0 - 1
```

ADC, SBC — carry-aware add / sub (flags use)

```
ADC R2, R0, R1
SBC R2, R0, #1
```

CMP — subtract and set flags (no result register)

```
CMP R0, #10       @ sets N,Z,C flags for branch decisions
BEQ equal_label
```

RSB / RSC — reverse subtract

```
RSB R1, R0, #0    @ R1 = 0 - R0  (negate)
```

AND, ORR, EOR, BIC — bitwise logic

```
AND R0, R0, #0xFF   @ mask low byte
ORR R1, R1, R2
EOR R3, R3, #1
BIC R4, R4, #0x10   @ clear bit 4
```

শিফটের সঙ্গে ব্যবহার (immediate shift):

```
LSL R0, R1, #2     @ R0 = R1 << 2
LSR R0, R1, #8
ASR R0, R1, #1
ROR R0, R1, #4
```

MOV/ORR shifting shorthand:

```
ORR R2, R2, R1, LSL #8   @ R2 |= (R1 << 8)
```

Memory access (load/store)

Byte / Halfword / Word:

```
LDR R0, [R1]          @ word load from [R1]
LDRB R2, [R1, #3]     @ byte load from [R1+3]
LDRH R3, [R1, #4]     @ halfword load
STR R0, [R1]          @ store word
STRB R2, [R1, #3]     @ store byte
STRH R3, [R1, #0]     @ store halfword
```

Auto-increment (post-index):

```
STRH R4, [R5], #2     @ store halfword then R5 += 2
LDRB R0, [R6], #1     @ read byte then increment
```

Pre-index:

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

Multiple register load/store (stack like ops):

```
STMFD SP!, {R4-R7, LR}  @ push registers
LDMFD SP!, {R4-R7, PC}  @ pop and return (PC pop for branch)
PUSH {R4,R5}            @ assembler macro for STMFD
POP {R4,R5}             @ assembler macro for LDMFD
```

Load literal / address:

```
LDR R0, =0xFF200000     @ load address or 32-bit imm (assembler places literal)
```

> টিপ: `LDR =label` ভাল; কিন্তু পরে `ADD R1, R0, label2` করলে assembler error দিতে পারে — তাই address math register-only রাখুন।

Control flow (branching & subroutines)

B — unconditional branch

```
B loop
```

BL — branch with link (call function)

```
BL my_function
```

BX LR — return from function (branch to link register)

```
BX LR
```

Conditioned branches (flag ও CMP ব্যবহার করে):

```
CMP R0, #0
BEQ zero_label    @ equal (Z=1)
BNE not_zero
BGT greater       @ greater (signed)
BLT less
BGE >=
BLE <=
```

SVC (supervisor call / semihosting) — debugging/semihosting I/O (CPUlator may support semihosting)

```
SVC #0
```

NOP — no operation

```
NOP
```

Multiply & multiply-accumulate

MUL — multiply (32-bit result low)

```
MUL R2, R0, R1     @ R2 = R0 * R1 (low 32 bits)
```

MLA — multiply and accumulate

```
MLA R2, R0, R1, R3 @ R2 = R0*R1 + R3
```

UMULL / SMULL — full 64-bit product

```
UMULL R1, R2, R3, R4   @ R1:high, R2:low = R3 * R4 (unsigned)
```

Flags and status

- S suffix sets flags (N Z C V) — e.g., ADDS R0, R1, R2
- TST, TEQ, CMN, CMP — compare/test (set flags)

```
ADDS R0, R1, R2   @ set flags
TST R0, #1        @ test bit
TEQ R0, R1
CMN R0, #1
```

Useful patterns (practical)

Write pattern to 7-segment HEX (example):

```
LDR R0, =0xFF200020   @ HEX0..HEX3 base
LDR R1, =digits_table
LDR R2, [R1, R3, LSL #2]  @ load pattern
ORR R4, R2, R2, LSL #8
ORR R4, R4, R2, LSL #16
ORR R4, R4, R2, LSL #24
STR R4, [R0]
```

UART transmit byte (polling LSR bit):

```
LDR R0, =UART0_BASE
wait:
  LDR R1, [R0, #LSR]
  TST R1, #0x20
  BEQ wait
  STRB R2, [R0, #THR]
```

VGA pixel write (16-bit RGB565):

```
LDR R0, =VGA_BASE
MOV R1, #320
@ compute offset in registers only then:
MUL R2, RY, R1      @ R2 = row * width
ADD R2, R2, RX
LSL R2, R2, #1      @ byte offset
ADD R2, R0, R2
STRH Rcolor, [R2]
```

Assembler pitfalls & tips (CPUlator / arm-eabi-as)

- LDR Rn, =const sometimes assembles via literal pool — OK.
- কিন্তু ADD Rd, Rn, =const → **not allowed**. Do LDR Rtmp, =const then ADD Rd, Rn, Rtmp.
- 32-bit immediates in MOV are limited (rotatable immediates). Use LDR =value for full 32-bit.
- When linking, ensure .global _start and proper linker script (CPUlator provides default).
- Use STRB/STRH when peripheral expects byte/halfword.

Quick cheat table (one-liner examples)

- Move immediate: MOV R0,#10
- Load word: LDR R1,[R2]
- Store byte: STRB R3,[R4,#1]
- Load byte: LDRB R5,[R6,#2]
- Add: ADD R0,R1,R2

- Compare & branch: CMP R0,#0 BEQ done
- Branch link: BL func
- Return: BX LR
- Push regs: PUSH {R4-R7,LR}
- Pop regs: POP {R4-R7,PC}
- Logical OR with shift: ORR R0,R0,R1,LSL #8
- Multiply: MUL R0,R1,R2
- Load immediate 32bit: LDR R0,=0xC8000000
- Write 16-bit pixel: STRH Rcolor,[Raddr]