DE1-SOC Advanced RISC Machine v7 - Cortex A9 – Reduced Instruction Set Computer

所有 LABEL 等同于该 LABEL 对应行数的地址（32-bit Binary 或 8 位 HEX）

MOV

MOV RX,RY	复制 RY 至 RX
MOV RX,#imm12	写入最多 12-bitBinary 至 RX
MOV RX,RY,type <shift> **先 shift 再写入，不影响 RY**	type 可为 ASR,ROR,LSR,LSL <shift>可为 register 或 imm5

如果加 S 后缀，N 和 Z flag 更新
C flag 根据 shift 的结果变化，V flag 不变

MOVW R0,#0x8888 // R0=0x0000 8888
MOVT R0,#0xAAAA // R0=0xAAAA 8888

CMP

CMP RX,RY	对比 RY 与 RX，改变 flags
CMP RX,RY, type <shift>	type 可为 ASR,ROR,LSR,LSL <shift>可为 register 或 imm5

LDR

LDR RX,[RY]	以 RY 为地址的内容加载到 RX 中
LDR{S}{M}	S: 加载 2's complement 的数， 只有在 M 也定义的情况下才有效 M: H for halfword, B for byte
LDR RX, =imm32 **pseudo instruction**	加载最高 32-bit Binary 入 RX
LDR RX,#LABEL	加载(LABEL 等同的地址)入 RX
LDR RX,LABEL	以(LABEL 等同的地址)的内容加载入 RX ⇨ LDR RX, [pc, offset to literal pool]
LDR RX,[RY,#4]	以 RY+4 为地址，RY 值不变
LDR RX,[RY,#4]!	以 RY+4 为地址，RY= RY+4
LDR RX,[RY],#4	以 RY 为地址，RY= RY+4
LDR RX,[RY,-RZ]	以 RY-RZ 为地址，RY 值不变

STR

STR{M} RX,[RY]	将 RX 加载到地址为 RY 的内存, M 只在非 GPIO 设备地址时可用

Multiply

MUL R1,R2,R3	R1 <- R2xR3
MLA R1,R2,R3,R4	R1 <- R2xR3+R4
不存在 imm 的版本，不要乱用	

Bitwise OP{s} RX,RY,RZ

AND RX,RY
ORR RX,RY,RZ
EOR RX,RY,ASR #imm5

.align 1 for halfword	
.align 2 for word	
.space numOfBytes,fill	
e.g. .space 4, 0xFF	

Shift Instructions

LSR RX,RY,RZ	将 RY 的值向右移 RZ 位，MSB 补 0，赋值于 RX
LSR RX,RY,#imm5 **逻辑右移**	向右移 n 位等同于 unsigned num / 2^n
ASR 算数右移	向右移，根据正负 MSB 补 0 或 1
LSL 逻辑左移	向左移，LSB 补 0 向左移 n 位等同于 unsigned num * 2^n
ROR 旋转	向右移，用 LSB 补 MSB (Rotate) 向右旋转(32-n)位等同于向左旋转 n 位

Stack instructions

PUSH {R1,R2,R3}	将 R1, R2, R3 存入 stack，Stack Pointer is decremented
POP {R1-R3}	将 R1, R2, R3 取出 stack, Stack Pointer is incremented

Current Program Status Register

Bits	Name	Function
[31]	N	result is negative
[30]	Z	result is zero
[29]	C	result produced a carry-out
[28]	V	result overflowed for signed numbers
…		
[7]	I	IRQ disable bit
[6]	F	FIQ disable bit
[5]	T	ARM mode:0; Thumb mode: 1
[4:0]	M	Operating Mode

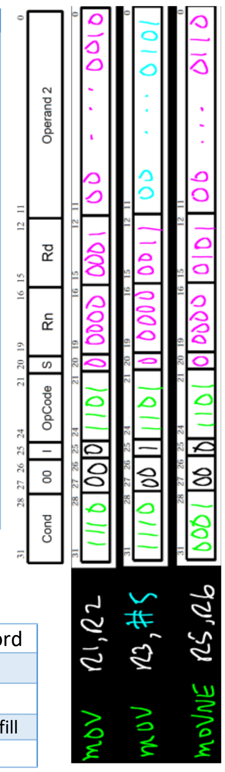
Code	Suffix	Description	Flags
0000	EQ	Equal / equals zero	Z
0001	NE	Not equal	!Z
1010	GE	Signed greater than or equal	N == V
1011	LT	Signed less than	N != V
1100	GT	Signed greater than	!Z and (N == V)
1101	LE	Signed less than or equal	Z or (N != V)
1110	AL	Always (default)	any

Special Cases

1. 除 PUSH,POP,LDR,STR 所有 instruction 结尾（{Cond}前）都能加{S}
2. 所有 instruction 结尾都能加{Cond}
3. ADD R0,R1,R2,LSL #2// R0 <- R1+(R2<<2)
   ADD R0,R1,R2,ASR #4// R0 <- R1+(R2/16)
   MOV R4,R5,LSL #3*// R4 <- R5*8
   *This mov is actually the same as LSL!
   LSL R0,R1,#2 <-> MOV R0,R1,LSL #2
   LSL R0,R1,R2 <-> MOV R0,R1,LSL R2
4. While loop: B END 或 MOV R15,#END
5. 从 subroutine 返回: MOV PC,LR 或 BX LR

```
MOV R1,#0x5555 5555
MOV R2,#0xAAAA AAAA
AND R3,R1,R2 // R3 <- 00000000
ORR R3,R1,R2 // R3 <- FFFFFFFF
AND R3,R1,#1 // R3 <- 00000001
(used to isolate bit 0 of R1)
```

```
// Swaps r1<->r2
EOR R2,R1,R2 // R2<-FFFFFFFF
EOR R1,R2,R1 // R1<-AAAAAAAA
EOR R2,R2,R1 // R2<-55555555
```



SUB

SUB Rx, #imm12	Rx <- (Rx-imm12)
SUB Rx, Ry, Rz	Rx <- (Ry-Rz)
SUB Rx, Ry	Rx <- (Rx-Ry)
SUBS PC, LR, #4	从 exception 返回的特殊方式， {S} 将 SPSR 拷回 CPSR

MSR CPSR, Rx

Field: _c, _f, 默认_fc

更改 CPSR 专用，相当于 MOV

Operating Mode	CPSR[4:0]
Supervisor (SVC)	0b10011
Undefined	0b11011
User	0b10000
Abort	0b10111
Interrupt (IRQ)	0b10010
Fast Interrupt (FIQ)	0b10001

Common Binary

2^{-4}=	0.0625
2^{-3}=	0.125
2^{-2}=	0.25
2^{-1}=	0.5
2^{0}=	1
2^{1}=	2
2^{2}=	4
2^{3}=	8
2^{4}=	16
2^{5}=	32
2^{6}=	64
2^{7}=	128
2^{8}=	256
2^{9}=	512
2^{10}=	1024 (1K)
2^{11}=	2048
2^{12}=	4096
2^{13}=	8192
2^{20}=	1M
2^{30}=	1G

Common Hex

16^{0}=	1
16^{1}=	16
16^{2}=	256
16^{3}=	4096
16^{4}=	65536

4-Bit Binary

0000	0	
0001	1	
0010	2	
0011	3	
0100	4	
0101	5	
0110	6	
0111	7	
1000	8	
1001	9	
1010	10	A
1011	11	B
1100	12	C
1101	13	D
1110	14	E
1111	15	F

2's Complement

0000	0
1111	-1
1110	-2
1101	-3
1100	-4
1011	-5
1010	-6
1001	-7
1000	-8

HEX Display

0	0x3F
1	0x06
2	0x5B
3	0x4F
4	0x66
5	0x6D
6	0x7D
7	0x07
8	0x7F
9	0x67
A	0x77
B	0x7C
C	0x39
D	0x5E
E	0x79
F	0x71

Float

31	30	23	22	0
sign	exponent		mantissa	

type	Size
char c;	byte (8 bit)
short k;	halfword (16 bits)
int i;	word (32 bits)
float f;	word (32 bits, called single precision)
double d;	double-word (64 bits, called double precision)

From the debugger (or at instruction fetch)	PC points to the instruction that has not yet executed. This is what the debugger shows you.
Reading r15 as an operand of an instruction (e.g., mov r0, pc or ldr r0, [pc])	The observed value is the PC of the instruction itself + 8.
Writing r15 as the destination operand of an instruction (e.g., mov pc, lr)	PC points to the next instruction to execute (same as the first definition above)
Branch-and-link instructions (function call)	The observed value (the value written to LR) is the PC of the instruction itself + 4.
Interrupts	The observed value (the one written to LR) is the PC of the first unexecuted instruction + 4 (so that returning to LR-4 is correct). Note that this is not the same as "PC of the last completed instruction + 8"

Enable interrupt

1. Setup stack in both IRQ and SVC mode

```
LDR R1,= 0b11010010
MSR CPSR_c,R1
LDR SP,= 0xFFFFFFFF-3//A9_ONCHIP_END
LDR R1,= 0b11010011
MSR CPSR_c,R1
LDR SP,= 0x3FFFFFFF-3//DDR_END
```

2. Setup GIC to enable

```
BL CONFIG_GIC
```

3. Enable interrupt on device

```
BL CONFIG_XXX
```

CONFIG_TIMER:

```
LDR R0, = 0xFF202000
LDR R1, = xxx // period
STR R1, [R0, #8] // lower 16 bits
LSR R1, #16
STR R1, [R0, #12] // higher 16 bits
MOV R1, #0b0111
STR R1, [R0, #4]
BX LR
```

CONFIG_PRIV_TIMER:

```
LDR R0, =0xFFFEC600
LDR R1, = xxx // period
STR R1,[R0]
LDR R1,=0b111
STR R1,[R0,#8]
BX      LR
```

CONFIG_KEYS:

```
LDR R0, =0xFF200050
MOV R1, #0xF
STR R1,[R0,#0x8]
BX      LR
```

4. Enable interrupt on ARM CPU

```
LDR R1, = 0b01010011
MSR CPSR_c,R1
```

```
void draw_line(int x0,int y0,int x1,int y1,short int line_color){
   bool is_steep = abs(y1-y0) >abs(x1-x0);
   if(is_steep){
      swap(&x0,&y0);
      swap(&x1,&y1);
   }
   if(x0>x1){
      swap(&x0,&x1);
      swap(&y0,&y1);
   }
   int delta_x = x1 - x0;
   int delta_y = abs(y1 - y0);
   int error = -(delta_x/2);
   int y = y0;
   int y_step;
   if (y0<y1){
      y_step = 1;
   } else{
      y_step = -1;
   }
   for(int x = x0; x<=x1; x++){
      if(is_steep){
         plot_pixel(y,x,line_color);
      } else{
         plot_pixel(x,y,line_color);
      }
      error += delta_y;
      if(error >= 0){
         y += y_step;
         error -= delta_x;
      }
   }
}
```

Interrupt Service Routine

1. Find out which device cause the interrupt

```
SERVICE_IRQ:
PUSH {R0-R7,LR}
LDR R4, = 0xFFFEC100 // GIC CPU interface base address
LDR R5, [R4, #0xC] // read from ICCIAR (who cause interrupt)
```

2. Resolve the device's interrupt request

```
CMP R5, #72
BEQ TIMER_ISR

UNEXPECTED: BNE UNEXPECTED
```

3. EXIT_IRQ:

```
STR R5,[R4, #0x10] // write to ICCEOIR to clear Interrupt
POP {R0-R7, LR}
SUBS PC,LR,#4 //with the S flag, the SPSR will be copied into CPSR when returning, restoring the NZCV flags and operating modes
```

TIMER_ISR:

```
LDR R0,=0xFF202000
MOV R1,#0
STR R1,[R0]
B EXIT_IRQ
```

Recursion in C and assembly

```
int FINDSUM(int N){
    if(N==0)
        return 0; *
    else
        return (N+FINDSUM(N-1));
}
```

```
          .text
          .global _start
_start:
          MOV SP, #0x20000 //init stack ptr
          MOV R4, #N        //R4<-N
          LDR R0,[R4]       //R0=N
          BL FINDSUM
POOP:STR R1,[R4],#4
END:  B    END
N:        .word 11
SUM: .space 4
/* Recursive Sum
   N is in R0, result returned in R1 */
FINDSUM:
          MOVS R1,R0      // N==0?
          MOVEQ PC,LR // if yes, return 0
RECURSIVE:
          PUSH {R0,LR} // save state (N, return address)
          SUB R0,#1        // N-1
          BL FINDSUM  // recurs
PEE:      POP {R0,LR}   // restore state
          ADD R1,R0       // FINDSUM (N-1) + N
          MOV PC,LR
```

```
int main(){
   int value;
   volatile int * LEDR_ptr = 0xFF200000;
   volatile int * SW_ptr = 0xFF200040;
   volatile int * HEX3_0_ptr = 0xFF200020;
   while(1){
      value = *SW_ptr; // read SW
      *LEDR_ptr = value; // write LEDR
      *HEX3_0_ptr =
seg7[value&0xF]|seg7[value>>4&0xF]<<8|seg7[value>>8]<<16; // write HEX3_0
   }
}
```

```
void wait_for_vsync(){
   volatile int * pixel_ctrl_ptr = (int *) 0xFF203020; // pixel controller
   register int status;

   *pixel_ctrl_ptr =1;
   status = *(pixel_ctrl_ptr+3);
   while ((status & 0x01)!= 0){
      status = *(pixel_ctrl_ptr+3);
   }
}
```

```
void plot_pixel(int x, int y,short int line_color){
    *(short *)(pixel_buffer_start + (y<<10)+(x<<1)) = line_color;
}
```

Set Associative Cache Calculation

Num of Sets = Total Size / Size of Set

Size of a Set = Size of Block * N-Way

Bits used for Offset = $\log_2$(Size of Block)

Bits used for Sets = $\log_2$(Num of Sets)

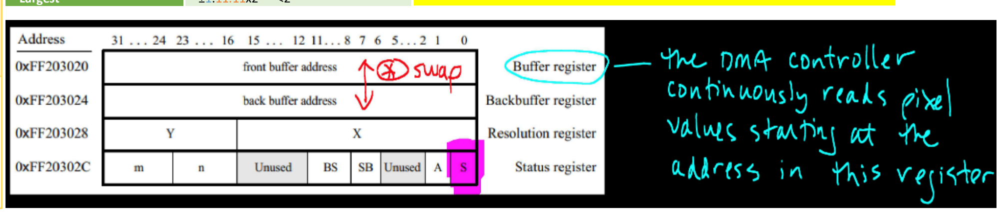
Bits used for (Tags + Sets + Offset) = 32

Special Case:

Directly-Mapped cache is One-Way SA;

Fully Associate Cache has 0 sets

Special Values	
±0	00000000 0000…00
±∞	11111111 0000…00
NaN	11111111 !=0
denormalized	00000000 !=0
Closet to 0	$\pm 1.0 \times 2^{-126}$
Largest	$\pm 1.11.11 \times 2^{127} \sim < 2^{128}$



When an exception occurs, the processor:
- Saves CPSR in the SPSR of the new mode
- Change CPSR to enter the new mode
- Saves PC into the banked LR of the new mode
- load into PC a unique address associated with the new mode. These addresses are called the Exception Vector Table.

```
.include "exceptions.s"
.include "config_GIC.s"
.section .vectors, "ax"
B      _start                 // reset vector
B      SERVICE_UND            // undefined instruction vector
B      SERVICE_SVC            // software interrupt vector
B      SERVICE_ABT_INST       // aborted prefetch vector
B      SERVICE_ABT_DATA       // aborted data vector
.word  0                      // unused vector
B      SERVICE_IRQ            // IRQ interrupt vector
B      SERVICE_FIQ            // FIQ interrupt vector

.text
.global _start
_start: ...
```

```
/* Program that counts consecutive 1's */
        .text                 // executable code follows
        .global _start
_start:MOV    R1, #TEST_NUM   // load the data word ...
        LDR    R1, [R1]       // into R1
        MOV    R0, #0         // R0 will hold the result
LOOP:CMP    R1, #0// loop until the data contains no more 1's
        BEQ    END
        LSR    R2, R1, #1 // perform SHIFT, followed by AND
        AND    R1, R1, R2
        ADD    R0, #1 // count the string length so far
        B      LOOP
END:  B      END
TEST_NUM: .word  0x103fe00f
        .end
```

```
/* Program that converts a binary number to decimal */
        .text              // executable code follows
        .global _start
_start: MOV    R9, #10      // MY DIVISOR
        MOV    R4, #N
        MOV    R5, #Digits  // R5 points to the decimal digits storage location
        LDR    R4, [R4]     // R4 holds N
        MOV    R0, R4 // parameter for DIVIDE goes in R0
        BL     DIVIDE
        STRB   R0, [R5]     // Ones digit is in R0

        MOV    R0, R2
        BL     DIVIDE
        STRB   R0, [R5, #1]     // Ones digit is in R0

        MOV    R0, R2
        BL     DIVIDE
        STRB   R0, [R5, #2]     // Ones digit is in R0

        MOV    R0, R2
        BL     DIVIDE
        STRB   R0, [R5, #3]     // Ones digit is in R0

END:     B    END
/* Subroutine to perform the integer division R0 / 10.
 * Returns: quotient in R1, and remainder in R0*/
DIVIDE:    MOV    R2, #0
CONT:      CMP    R0, R9
           BLT    DIV_END
           SUB    R0, R9
           ADD    R2, #1
           B      CONT
DIV_END: MOV R1, R2 // quotient in R1 (remainder in R0)
           MOV    PC, LR
N: .word  9876 // the decimal number to be converted
Digits: .space 4  // storage space for the decimal digits
           .end
```

在某 subroutine 中 BL 另 subroutine 前，记得 PUSH {LR}！！！！ 然后记得 POP！！！