Branch Instructions

B: 무작정 점프

BL: R14에 돌아올 주소 넣고 점프

0또는 1이 체크가 돼서 -> 0일때는 B, 1일때는 BL

```
        mov r0, #1      ; 0x00
        add r1, r0, r0  ; 0x04
        bl  func        ; 0x08
        add r1, r0, r0  ; 0x0C
        sub r1, r0, r0  ; 0x10
loop        b   loop        ; 0x14
func        add r1, r0, r0  ; 0x18
        mov pc, lr      ; 0x1C
```

- 첫번째 줄 실행하면 PC값이 0x00이 아니라 0x08임

  왜? 2클락 앞에 가있음 fetch - decode - execute -

- bl 실행하면 link reg에 **PC - 4** = 0x0C가 들어감

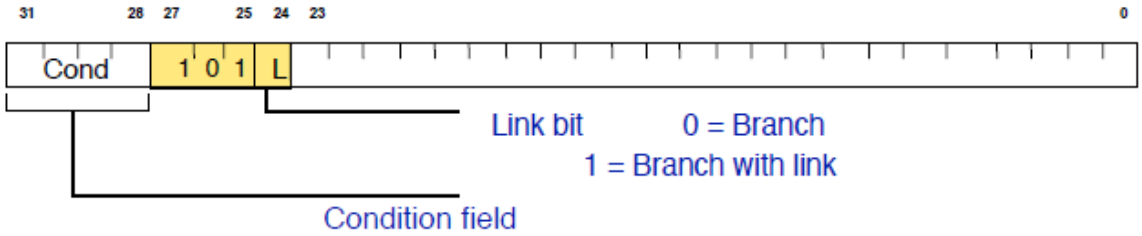
  `PC - 4인 이유는?`

  `PC base + 0 BL there EX base+0 base + 4 add R0, R1, R2 ID base+4 base + 8 STR R0, [R4] IF base+8 : LR <= base + 4 = (PC = base+8) - 4)`

  위 그림을 보았을 때, base+ 0이 실행되고 있는 시점에 base + 8주소의 명령어가 fetch되고, base + 4의 주소가 decode 된다. pc란 fetch되는 주소를 가리키므로, base + 0를 시작하고 있을 때 pc값은 base + 8이다. BL에서 돌아올 주소는 base + 8의 전 line인 base + 4이기 때문에 pc - 4를 link reg에 저장한다.

- func 실행하고 돌아오면 조금 느려지는 현상 발생 -> pipeline이 깨져서 fectch랑 decode를 채워넣는 추가 clock이 필요하기 때문에

Branch Instructions



- PC 값을 기준으로 비트 [23:0]에 해당하는 offset이 사용
- 24 비트의 offset
  - 맨 상위 비트는 +/- sign 비트
  - 나머지 23비트를 이용하여 2비트 왼쪽으로 shift한 값을 사용
  - 분기 가능한 주소영역: PC +/- 32MB

[31:18] 조건 필드

[27:25] 101: branch 구나!

[23:0] 상대주소임 - PC + offset

그래서 BL 은 lable(절대 주소)로 점프 할 수 있음

32bit -> 4GB (x), 최대 24bit -> 점프할 수 있는 길이 제한 = 위로 8MB 아래로 8MB(2의 20승 * 2의 4승 / 2)

8MB 이상 가고싶다면? 레지스터에다가 32bit값(가고싶은 주소의 절대 주소)을 작성

= 메모리의 어떤 영역에 4 byte만큼의 원하는 주소를 적어놓은 다음에 메모리로부터 이것을 r0에 로드

- 실제상으로는 32bit를 사용할 수 있도록 구현

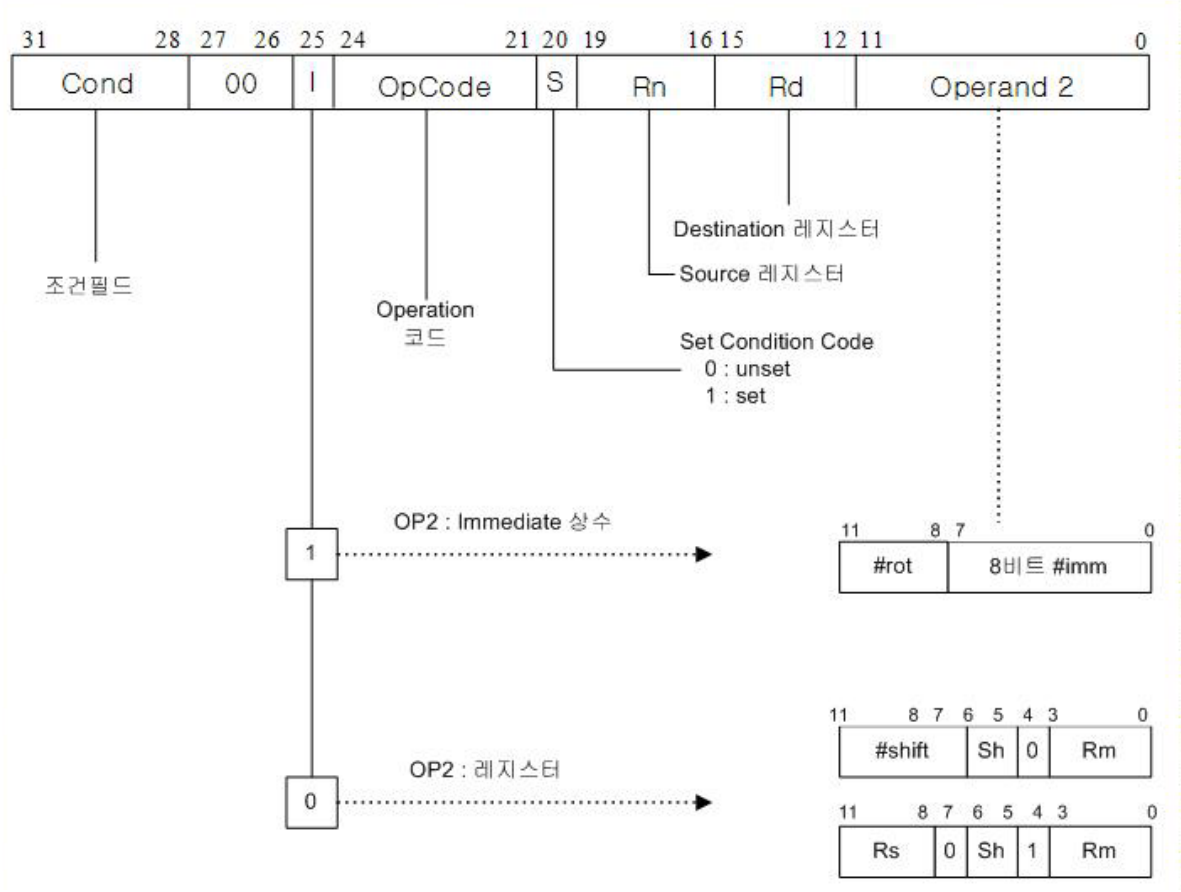
  하나의 명령어 = 4byte (32bit), word adress로 쓴다면 하위 두비트는 항상 0일 테니깐 8 * 4 = 32 MB씩 위아래로 쓸 수 있음

Subroutines

- 다음에 수행할 명령의 위치를 LR에 저장한다.
- LR = PC - 4 <- 파이프라인 단계에서 F - D - Ex - ... 즉, BL이 실행중일때는 다음 PC는 BL로부터 2번째 명령어를 인출중이다. 따라서 돌아갈 번지는 BL로 부터 첫번째 명령어를 저장

Data Processing Instructions

하드웨어에서는 한 블록 내에 여러개의 클럭이 구현 가능




```
ADD r0, r1, r2
COND I OP S Rn Rd #shift Sh Rm
1110 00 0 0010 0 0001 0000 00000 00 0 0010
e0810002
```

1110 -> ALWAYS, 0010 -> ADD, 0001 -> Rn, 0000 -> Rd

Arithmetic Operations

opcode	inst	behavios	explanation
0100	ADD	Rd := Rn + Op2	Add
0101	ADC	Rd := Rn + Op2 + Carry	Add with Carry
0010	SUB	Rd := Rn – Op2	Subtract
0110	SBC	Rd := Rn – Op2 – 1 + Carry	Subtract with Carry
0011	RSB	Rd := Op2 - Rn	Reverse subtract
0111	RSC	Rd := Op2 – Rn – 1 + Carry	Reverse Subtract with Carry

```
    ;       FF FFFFFFFF
    ;       +         1
    ;       -------------
    ;       100 00000000
    ldr     r1, =0x000000ff  ; MSB of OP1
    ldr     r0, =0xffffffff  ; LSB of OP1
    ldr     r3, =0x00000000  ; MSB of OP2
    ldr     r2, =0x00000001  ; LSB of OP2
    ldr     r5, =0x0 ; MSB of destination register
    ldr     r4, =0x0 ; LSB of destination register
      adds      r4, r0, r2              // 더하고 정보를 cpsr에 저장 -> z와 c가 set
                                        // z는 왜?? 어쨌건 올림되서라도 0이니깐
      adc       r5, r1, r3              // carry와 함께 더하기
```

Logical Operations

opcode	inst	behavior	explanation
0000	AND	Rd := Rn AND Op2	AND
1111	ORR	Rd := Rn OR Op2	OR
0001	EOR	Rd := Rn XOR Op2	Exclusive OR
1110	BIC	Rd := Rn AND (NOT Op2)	Bit Clear

Comparision Operations

명령어 인위적으로 외우지 x

opcode	inst	behavior	expanation
1010	CMN	CPSR flags := Rn + Op2	Compare Negative
1011	CMP	CPSR flags := Rn-Op2	Compare
1000	TEQ	CPSR flags := Rn EOR Op2	Test bitwise equality
1001	TST	CPSR flags := Rn AND Op2	Test bits

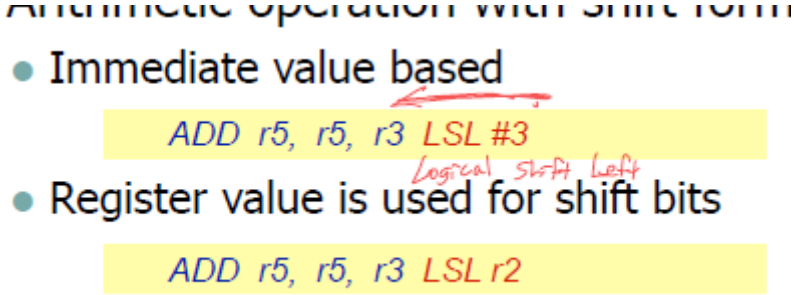
Data Move Operations

MOVS, MVNEQ 가볍게 읽어보시구여

opcode	inst	behavior	explanation
1101	MOV	Rd := Op2	Move register or constant
1111	MVN	Rd := 0xFFFFFFFF EOR Op2	Move Negative register

Arithmetic Operation with Shift

- 이점
  - 단일 명령으로 2차 피연산자로의 시프트연산을 적용하는 피연산자의 산술 연산
  - 배럴 시프터 사용



Arithmetic Operation with Shift

- LSL : 왼쪽으로 쉬프트 후 LSB(1의 자리)의 빈자리를 0으로 채움
- LSR : 오른쪽으로 쉬프트 후 MSB(부호비트)의 빈자리를 0으로 채움
- ASL : LSL과 같음
- ASR : 오른쪽으로 쉬프트 한 후 MSB(부호비트)의 빈자리가 양수인 경우에는 0 음수인 경우에는 1로 채운다.
- ROR : 오른쪽으로 쉬프트 한 후 LSB에서 밖으로 나온 비트는 다시 MSB로 들어간다

Operand 2 and Immediate Value

- 제한된 32bit 공간안에 큰 immediate value를 어떻게 넣을까?

  2^12 = 4095

  양보하는김에 더해 : 8bit 쓰세요 -> 2^8

  //3강

Operand 2 and Immediate Value

PSR Transfer Instruction

MRS: PSR의 값을 일반 레지스터로 가져올때, 통째로 지정한 레지스터에 copy, copy해온 사본은 우리가 편집할 수도 있고 마음대로 읽을 수 있음

하지만

특정 bit을 수정하여 st 하는것은

MSR : 레지스터에서 PSR로 쓸때, 쓰기권한은 user모드에서 지원하지 않음

MRS, MSR이란 명령어가 있었다 라는 것만 확인

arm에서 메모리를 접근한다 -> 오늘 어제 배운 몇개 명령어

- PSR(Program Status Register)의 경우 특별 관리가 필요
  - 범용 레지스터와 PSR 사이에서 레지스터 컨텐츠를 유일한 이동 가능 접근 방법
  - MRS : PSR에서 Reg로
  - MSR : Reg에서 PSR로
- CPSR (Current Program Status Register)
  - 모든 operation mode에서 CPSR를 접근할 수 있슴
  - USER mode에서 조건 필드(condition fields)만 수정할 수 있음
- SPSR (Saved Program Status Register)
  - 각 operatio mode는 전용 SPSR를 가지고 있음
    - USER mode는 SPSR를 가지고 있지 않음

PSR Transfer Instruction (MRS)

- MRS : PSR에서 Reg로 이동
- ex) MRS r0, CPSR

PSR Transfer Instruction (MSR)

- MSR : Reg에서 PSR로 이동
- ex) MSR CPSR, r0

  MSR CPSR_f, r0 CPSR의 condition flags 비트 = r0

  MSR CPSR_fc, r0 CPSR의 flags, control 비트 = r0

  MSR CPSR_c #0x80 CPSR의 control 비트 = 0x80