


Xilinx Zynq FPGA,TI DSP, MCU 기반의 프로그래밍 전문가 과정


날 짜 : 2018 . 5. 1

강사 – Innova Lee(이상훈)
gcccompil3r@gmail.com
학생 – 정한별
hanbulkr@gmail.com

< Arm 명령어>

lotation_shift → (“mov r0, #0xff, 8”)

– arm 은 32bit 단위 이다. 0Xff → 0X000000ff 라고 볼수 있다.

– 위와 같이 mov 명령어를 사용할 경우, 8bit lotation 하겠다. 라는 의미이다.

– 방향은 항상 왼쪽인 것 같다.

lsl(logical shift left) → (“add r0, r1, r2, lsl #7”) ; (r0 = r1 +(r2 * 2^7))

– 왼쪽으로 비트를 밀어 버린다고 생각하면 된다. (r1 을 왼쪽으로 7 번 민다.)

asr(arithmetic shift right) → (“add r0, r1, asr #2”) ; (r0 = r1 + (r2 / 2^2))

– lsl 과 반대로 asr 은 오른쪽 방향으로 쉬프트 하는 연산이다.

– 위의 경우는 r1 을 오른쪽으로 2 번 민다.

mrs → (“mrs r0, cpsr”) ; (cpsr 를 r0 에 전달한다.)

– cprs, sprs 의 값을 arm 의 범용 레지스터로 읽어 온다.

– sprs 의 특정 비트의 접근을 가능하게 해주어 레지스트 설정 변경이 가능하다.

– cprs 의 특정 비트에 접근을 가능하게 해주어 레지스트 설정 변경이 가능하다.

– 주로 인터럽트를 껏다. 켰다 할 때 많이 사용한다.

msr → (“msr cpsr r0”) ; (r0 를 cpsr 에 전달한다.)

– arm 의 값을 cprs, sprs 의 레지스터에 쓰는 명령이다.

– sprs 의 특정 비트의 접근을 가능하게 해주어 레지스트 설정 변경이 가능하다.

– cprs 의 특정 비트에 접근을 가능하게 해주어 레지스트 설정 변경이 가능하다.

– 주로 인터럽트를 껏다. 켰다 할 때 많이 사용한다.

Mul (multiple) → (“mul r0 , r1, r2) ; (r0 = r1 *r2)

– 곱하는 연산을 한다.

mla (mul + add)→ (“mla r0, r1, r2, r3”) ; (r0 = r1 *r2 +3)

– 곱하고 더하는 연산을 한다.

mull(확장.) → (“umull r0, r1, r2, r3”) ; (r0(하위)=32bit 안으로 저장 , r1(상위)=32bit 초과 저장. r2*r3)

– 확장의 개념, 곱하기를 하지만 32bit 를 초과시에 상위 bit 에 저장을 한다.

– 예) 0x44 00 00 00

0x 2 00 → 이렇게 2 자리가 초과되면 상위큰 값을 저장.

0x88 00 00 00 00

umlal (mul + add) → (“umlal r0, r1, r2, r3”) ; (r0 = 곱계산된하위 + r0, r1= 곱 계산된 상위 + r1 , r2*r3)

– 확장의 개념 , 곱하기와 더하기를 하지만 32bit 를 초과시에 상위 bit 에 저장한다.

DATA

시스크와 리스크의 큰 차이점. → load store architecture
→ 인텔 (메모리 → 메모리)
→ ARM(메모리 → 레지스터 and 레지스터 → 메모리)

ldr (load register) → ("ldr r0, [r1, r2]") r0 = (r1 레지스터의 r2 바이트만큼 이동한 위치에 정보를 r0 에 저장)
– 메모리의 정보 값을 레지스터에 넣어줌.

** mov 로는 **메모리의 값을 읽지 못하기** 때문에

ldreqb (load register equal byte) → ("ldreqb r0, [r1, #0x5]") r0 = r1 주소에 5 번째 주소를 읽어옴)
– cpsr 의 z bit 를 읽고 셋이 되어 있으면 실행한다.

srt(store register) → ("strb r0, [r1,#5]") r1 의 주소의 5 번째 byte 에 r0 를 써줌.
– 레지스터에서 메모리로 저장한다.

ldr 과 ! → ("ldr r0, [r1, r2]!")
– r1 주소에의 r2 만큼 이동 후 그곳 부터 r0 에 읽어옴.

ldr 과 인자 3 개 → ("ldr r0, [r1], r2")
– r0 에 r1 의 주소 첫 값을 넣음 r1 에 r2 값을 넣어 줌.

stmia(store multiple increment after) → ("stmia r0, {r1, r2,r3}")
– 레지스터에서 메모리로 저장(스텍) 하고 index 증가시켜라.
– r0 주소에 인덱스 별 저장공간에 레지스터를 순서대로 저장한다.

ldmia(load multiple increment after) → ("ldmia r0, {r4,r5,r6}")
–r0 에 저장된 값들을 순서대로 "{}" 안에 있는 레지스터에 넣는다.

**** '!' 는 연산된 위치까지 이동되어 그 위치가 저장됨.**

Debugging

→ **bl(brench link)** 은 함수로 들어가는 녀석 . 함수로 들어갈 때 lr 레지스터에 복귀주소를 남김. (jmp+ call)
→ **bx** 은 함수를 나올 때 있었다. (returnq)
→ **r11 레지스터** bp 이다.
→ **str**(레지스터 → 메모리)이 나오면 **ldr**(메모리→ 레지스터)이 나온다.
→ **Inter** 은 함수의 인자를 stack 으로 전달. **ARM** 은 함수의 인자를 register 로 전달.
→ 함수의 매개 변수의 **인자가 4 개 이상**쓰면 register 에 저장되지 않고 stack 에 저장이 된다.(ARM 에서)
→ 함수의 거의 **return 값 r0** 사용.
→ r0 ~r3 까지는 함수관련해서 많이 사용함.
→ **r7 레지스터는 systemcall 역할**로 사용한다.
→ r4 ~r12 까지 그냥 사용해도 되지만 r7 조심.
→ 나머지는 쓰면 안됨.