ARM Microprocessors 2

Data transfer instructions

- Single register load and store instructions
  - Transfer unit: byte, half-word, word
- Multiple register load and store instructions
  - Multiple transfer using a single instruction
  - Useful for function entries/exits & bulk memory copy
- Single register swap instructions
  - An atomic operation
  - Useful when implementing semaphores and mutual exclusion in OS

Data Transfer Instructions (cont'd)

Single Register Transfer

Register-indirect addressing

LDR r0, [r1]	r0 := mem_{32}[r1]
STR r0, [r1]	mem_{32}[r1] := r0

LDRB r0, [r1]	r0 := mem_{8}[r1]

Note: r1에 제한 없음.

Base+offset addressing
(offset of up to 4Kbytes)

LDR r0, [r1, #4]	r0 := mem_{32}[r1 +4]

Auto-indexing addressing

LDR r0, [r1, #4]!	r0 := mem_{32}[r1 + 4]
	r1 := r1 + 4

Post-indexed addressing

LDR r0, [r1], #4	r0 := mem_{32}[r1]
	r1 := r1 + 4

Data Transfer Instructions (cont'd)

```
COPY:    LDR r1, .TABLE1    ; r1 points to TABLE1
         LDR r2, .TABLE2    ; r2 points to TABLE2
LOOP:    LDR r0, [r1]
         STR r0, [r2]
         ADD r1, r1, #4
         ADD r2, r2, #4
         ...
.TABLE1: ...
.TABLE2:...
```

```
COPY:    LDR r1, .TABLE1    ; r1 points to TABLE1
         LDR r2, .TABLE2    ; r2 points to TABLE2
LOOP:    LDR r0, [r1], #4
         STR r0, [r2], #4
         ...
.TABLE1: ...
.TABLE2:...
```

Data Transfer Instructions (cont'd)

- Example :
  - C code: `A[8] = h + A[8]`
    - A : an array of 100 words
    - r1 : base address of the array A
    - r3 : h
  - ARM code:

```
ldr    r2, [r1, #32]
add    r2, r3, r2
Str    r2, [r1, #32]
```

Data Transfer Instructions (cont'd)

- C code: `g = h + A[i]`
  - r1 : base register for A
  - g, h, i : r2, r3, r4
- ARM code:

```
ldr     r5, [r1, r4, asl #2]
add     r2, r3, r5
```

Data Transfer Instructions (cont'd)

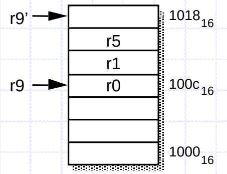
Multiple Register data Transfers

LDMIA r1, {r0, r2, r5}	r0 := mem_{32}[r1]
	r2 := mem_{32}[r1 + 4]
	r5 := mem_{32}[r1 + 8]

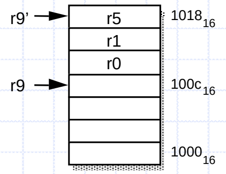
- Block copy view
  - Up or down from the base register
  - Address chagne before or after load/store

- Block Load/Store
  - IA – Increment After
  - IB – Increment Before
  - DA – Decrement After
  - DB – Decrement **Before**
- Stack Pop/Push
  - FA – full ascending
  - EA – empty ascending
  - FD – full descending
  - ED – empty descending

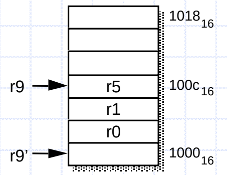
Multiple register transfer addressing modes



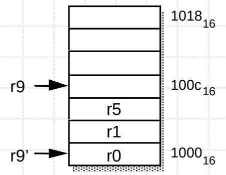
STMIA r9!, {r0,r1,r5}



STMIB r9!, {r0,r1,r5}



STMDA r9!, {r0,r1,r5}



STMDB r9!, {r0,r1,r5}

Block Memory Copy Example

```
loop
 LDMIA r9!, {r0-r7}
 STMIA r10!, {r0-r7}

 CMP   r9, r11
 BNE   loop
```

The mapping between the stack and block copy views

Addr Mode		Pop	== LDM	Push	== STM
FA	Full Ascending	LDMFA	LDMDA	STMFA	STMIB
FD	Full Descending	LDMFD	LDMIA	STMFD	STMDB
EA	Empty Ascending	LDMEA	LDMDB	STMEA	STMIA
ED	Empty Descending	LDMED	LDMIB	STMED	STMDA

Push & Pop using LDM & STM
Full Stack (last full) vs. Empty Stack (next empty)
Ascending vs. Descending

Control flow instructions

Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Control flow instructions

- Example:
  - C code: `if (i==j) h = i+j;`
  - ARM code:

```
        cmp   r1, r2
        bne   .L2
        add   r3, r1, r2
   .L2:
        ...
```

Control flow instructions

- Example:
  - C code:
```
if (i!=j) h = i+j;
else      h = i-j;
```
  - ARM code:
```
          cmp     r1, r2
          beq     .L2
          add     r3, r1, r2
          b       .L3
.L2:
          sub     r3, r1, r2
.L3:
          ...
```

Control flow instructions

- Example:
  - C code:
```
while (A[i] == k)     i = i+ j;
```
  - ARM code:
```
.L2:
          ldr     r3, .L5             ; r3 is the base register A[]
          mov     r4, r4, asl #2      ; r4 = i
          ldr     r3, [r3, r4]        ; r3 = A[i]
          cmp     r3, r5              ; compare A[i] and k
          beq     .L4
          b       .L3                 ; exit loop
.L4:
          ldr     r4, [fp,#-16]
          add     r4, r4, r1
          str     r4, [fp,#-16]
          b       .L2
.L3:
     ...
.L5:
          .word   A
```

Conditional execution

- Conditional execution can replace ‘branch’ instructions
- Example

```
          CMP r0, #5          ;
          BEQ BYPASS          ; if (r0!=5) {
          ADD r1, r1, r0      ;    r1:=r1+r0-r2
          SUB r1, r1, r2      ; }
BYPASS: ...
```

With conditional execution

```
          CMP r0, #5          ;
          ADDNE r1, r1, r0    ;
          SUBNE r1, r1, r2    ;
          ...
```

```
; if ((a==b) && (c==d)) e++;

CMP r0, r1
CMPEQ r2, r3
ADDEQ r4, r4, #1
```

Note: add 2 –letter condition after the 3-letter opcode

Branch and link instructions

- Branch to subroutine (link register == r14)

```
          BL SUBR ; branch to SUBR
          ..          ; return here

SUBR:     ..          ; SUBR entry point
          MOV pc, r14 ; return
```

- Nested subroutines

```
          BL SUB1
          ..
SUB1:     ; save work and link register
          STMFD r13!, {r0-r2,r14}
          BL SUB2
          ..
          LDMFD r13!, {r0-r2,pc}
SUB2:     ..
          MOV pc, r14 ; copy r14 into r15
```

Software Interrupt Instruction

- Provide a mechanism to call O/S routines

```
; output r0[7:0]
SWI SWI_WriteC
; return from a user program back to monitor
SWI SWI_Exit
```

```
SWI{<cond>} SWI_number
        lr_svc = addr. of inst following the SWI
        spsr_svc = cpsr
        pc = vectors + 0x8
        cpsr mode = SVC
        cpsr I = 1 (mask IRQ interrupts)
```

SWI Handler Example

```
SWI_handler:
  STMFD sp!, {r0-r12, lr}

  LDR   r10, [lr, #-4]
  BIC   r10, r10,#0xff000000

  ; the number in r10 used to call the service routine
  BL    service_routine

  LDMFD  sp!, {r0-r12, pc}^
```

Jump tables

- Useful for 'switch' statement:

```
        BL JTAB
        ...
JTAB:   CMP r0, #0
        BEQ SUB0
        CMP r0, #1
        BEQ SUB1
        CMP r0, #2
        BEQ SUB2
```

```
        BL JTAB
        ...
JTAB:   ADR r1, SUBTAB
        CMP r0, #SUBMAX ; overrun?
        LDRLS pc, [r1, r0, LSL #2]
        B ERROR
SUBTAB: DCD SUB0
        DCD SUB1
        DCD SUB2
        ...
```

Note: slow when the list is long, and all subroutines are equally frequent

Hello ARM World!

```
          AREA HelloW, CODE, READONLY ; declare code area
SWI_WriteC      EQU     &0          ; output character in r0
SWI_Exit        EQU     &11         ; finish program
          ENTRY                     ; code entry point
START:    ADR r1, TEXT              ; r1 <- Hello ARM World!
LOOP:     LDRB r0, [r1], #1         ; get the next byte
          CMP r0, #0                ; check for text end
          SWINE SWI_WriteC          ; if not end of string, print
          BNE LOOP
          SWI SWI_Exit              ; end of execution
TEXT      = "Hello ARM World!", &0a, &0d, 0
          END
```

PSR Instruction

- Two instructions for controlling PSR
  - MRS Rd, <cpsr|spsr>
  - MSR <cpsr | spsr>_<fields>, Rm
  - MSR <cpsr | spsr>_<fields>, #immediate
- Fields: byte regions of PSR
  - control (c): psr[0:7]
  - Extension (x): psr[8:15]
  - status (s): psr[16:23]
  - flags (f): psr[24:31]

MRS & MSR Example

```
MRS  r1, cpsr

BIC  r1, r1, #0x80
                 ; clear bit 7

MSR  cpsr_c, r1
```

Coprocessor Instruction

- MRC: move from coprocessor register to register
- MCR: move from register to coprocessor register

CP15: ARM System Control Coprocessor

- CP15
  - on-chip cache or caches, memory management or protection unit, write buffer, prefetch buffer, branch target cache and system configuration
  - In Supervisor mode, MRC,MCR used to access registers

Register	Purpose
0	ID Register
1	Control
2	Translation Table Base
3	Domain Access Control
5	Fault Status
6	Fault Address
7	Cache Operations
8	TLB Operations
9	Read Buffer Operations
10	TLB Lockdown
13	Process ID Mapping
14	Debug Support
15	Test and Clock Control
4, 11-12 UNUSED	

Loading Constant Instruction

- Two pseudoinstructions to move a 32-bit value into a register
  - LDR Rd, =constant
    - MOV Rd, #constant

```
LDR r0, =0xff      ==   MOV r0, #0xff
```

  - ADR Rd, label
    - Rd = 32-bit relative address (PC relative add or subtract)