CSE 3103: Microprocessor and Microcontroller

Professor Upama Kabir

Computer Science and Engineering, University of Dhaka,

Lecture: ARM Instruction Encoding

March 20, 2024

Table of Contents

Moving Data within the processor

Table 5.4 Instructions for Transferring Data within the Processor

Instruction	Dest	Source	Operations
MOV	R4,	R0	; Copy value from R0 to R4
MOVS	R4,	R0	; Copy value from R0 to R4 with APSR (flags) update
MRS	R7,	PRIMASK	; Copy value of PRIMASK (special register) to R7
MSR	CONTROL,	R2	; Copy value of R2 into CONTROL (special register)
MOV	R3,	#0x34	; Set R3 value to 0x34
MOVS	R3,	#0x34	; Set R3 value to 0x34 with APSR update
MOVW	R6,	#0x1234	; Set R6 to a 16-bit constant 0x1234
MOVT	R6,	#0x8765	; Set the upper 16-bit of R6 to 0x8765
MVN	R3,	R7	; Move negative value of R7 into R3

Figure 1

Moving Data within the processor

- To set a register to a larger immediate value (between 9-bit and 16-bit), the MOVW instruction can be used.
- To set a register to a 32-bit immediate value, there are several ways of doing this.
  - Use a pseudo instruction called "LDR"; for example:
    LDR R0, =0x12345678 ; Set R0 to 0x12345678
    This is not a real instruction. The assembler converts this instruction into a memory transfer instruction and a literal data item stored in the program image:
    LDR R0, [PC, #offset]
    ..
    DCD 0x12345678
    The LDR instruction reads the memory at [PC+offset] and stores the value into R0.
  - Another way tis to use a combination of MOVW and MOVT instructions. For example:
    MOVW R0, #0x789A ; Set R0 to 0x0000789A
    MOVT R0, #0x3456 ; Set upper 16-bit of R0 to 0x3456,
    ; now R0 = 0x3456789A

Memory Access Instruction

Table 5.6 Memory Access Instructions for Various Data Sizes

Data Type	Load (Read from Memory)	Store (Write to Memory)
8-bit unsigned	LDRB	STRB
8-bit signed	LDRSB	STRB
16-bit unsigned	LDRH	STRH
16-bit signed	LDRSH	STRH
32-bit	LDR	STR
Multiple 32-bit	LDM	STM
Double-word (64-bit)	LDRD	STRD
Stack operations (32-bit)	POP	PUSH

Figure 2

Supports 3 primary addressing modes

- Preindex with writeback
- Preindex
- Postindex

Example

- Initial:
  r0= 0x00000000 r1= 0x00009000
  mem 32 [0x00009000]= 0x01010101
  mem 32 [0x00009004]= 0x02020202
- Preindexing with writeback: LDR r0, [r1, #4]!
  r0= 0x02020202
  r1= 0x00009004
- Preindexing with writeback: LDR r0, [r1, #4]
  r0= 0x02020202
  r1= 0x00009000
- Postindexing: LDR r0, [r1],#4
  r0 = 0x01010101
  r1 = 0x00009004
- A memory access can generate the address value from the current PC value and an offset value, needed for loading immediate values. into a register,

Stack PUSH and POP

- Push registers on and pop registers off a full-descending stack.
- Use the currently selected stack pointer (MSP or PSP) for address generation
- Syntax: PUSH {con} reglist
  POP{con} reglist
  cond: an optional condition code.
  reglist: a non-empty list of registers, enclosed in braces.
- Example:
  - PUSH {R0,R3-R6,R8} : Push R0,R3,R4,R5,R6,R8 into stack
  - POP {R0,R3} : Pop R0,R3 from stack
- Usually, a PUSH instruction will have a corresponding POP with the same register sets, but this is not always necessary.
- PUSH {R3-R6,LR} : Save R3 to R6 and LR at the beginning of the subroutine. LR contains the return address. processing in the subroutine
- POP {R3-R6,PC} : Pop R3 to R6 and return address from stack. return address is stored into PC directly, triggers a branch (subroutine return)
- Instead of POP LR and writing into PC, we can put return address directly to PC to save instruction count and cycle count.

Comparison Instruction

- Compare and test instructions are used to update the flags in APSR, which later used by a conditional branch or conditional execution.
- CMP $< R_n >,< R_m >$: compare, calculate $R_n - R_m$, APSR is updated but the result is not stored
- CMN $< R_n >,< R_m >$: compare negative, calculate $R_n + R_m$, APSR is updated but the result is not stored
- TST $< R_n >,< R_m >$: test (bitwise AND), calculate $R_n ANDR_m$, N and Z bit in APSR is updated but the result is not stored
- TEQ $< R_n >,< R_m >$: test (bitwise XOR), calculate $R_n XORR_m$, N and Z bit in APSR is updated but the result is not stored

Flow control operations

Table 5.31 Unconditional Branch Instructions

Instruction	Operation
B <label> B.W <label>	Branch to label. If a branch range of over +/-2KB is needed, you might need to specify B.W to use 32-bit version of branch instruction for wider range.
BX <Rm>	Branch and exchange. Branch to an address value stored in *Rm*, and set the execution state of the processor (T-bit) based on bit 0 of *Rm* (bit 0 of *Rm* must be 1 because Cortex-M processor only supports Thumb state).

Figure 3

Flow control operations

Table 5.32 Instructions for Calling a Function

Instruction	Description
BL <label>	Branch to a labeled address and save the return address in LR
BLX <Rm>	Branch to an address specified by *Rm*, save the return address in LR, and update T-bit in EPSR with LSB of *Rm*

Figure 4

Conditional Branches

- Branch operation (with condition):
  B {condition}label: PC-relative expression (immediate)
  BL {condition} label : with link (immediate)
- All operations can be performed conditionally based on current value in APSR
  EQ, NE, CS, CC, MI, PL, VS, VC, HI, LS, GE, LT, GT, LE
- Conditional branch: BNE label
- Thumb2 additions (compare and branch if zero/nonzero):
  CBZ r0,label ;branch if r0 == 0: COMP r0, 0, BEQ label
  CBNZ r0,label ;branch if r0 != 0
- CBZ: Compare and Branch if Zero
- CBNZ: Compare and Branch if NonZero
- They only support forward branches and not backward branches

CBZ and CBNZ are very useful in loop structures such as while loops. For example:

```
i = 5;
while (i != 0 ){
  func1();     // call a function
  i--;
}
```

Figure 5

This can be compiled into:

```
      MOV R0, #5       ; Set loop counter
loop1 CBZ R0,looplexit ; if loop counter = 0 then exit the loop
      BL func1         ; call a function
      SUBS R0, #1      ; loop counter decrement
      B loop1          ; next loop
looplexit
```

Figure 6

The APSR value is not affected by the CBZ and CBNZ instructions.

Equivalent of a C program

- C: $if(a > b)\{x = 5; y = c + d;\}elsex = c - d;$
- Assembly: ; compute and test condition
  LDR r4,=a ; get address for a
  LDR r0,[r4] ; get value of a
  LDR r4,=b ; get address for b
  LDR r1,[r4] ; get value for b
  CMP r0,r1 ; compare $a < b$
  BLE fblock ; if $a <= b$, branch to false block

Equivalent of a C program

- ; true block
  MOV r0,#5 ; generate value for x
  LDR r4,=x ; get address for x
  STR r0,[r4] ; store x
  LDR r4,=c ; get address for c
  LDR r0,[r4] ; get value of c
  LDR r4,=d ; get address for d
  LDR r1,[r4] ; get value of d
  ADD r0,r0,r1 ; compute y
  LDR r4,=y ; get address for y
  STR r0,[r4] ; store y
  B after ; branch around false block

Equivalent of a C program

- ; false block
  fblock LDR r4,=c ; get address for c
  LDR r0,[r4] ; get value of c
  lDR r4,=d ; get address for d
  LDR r1,[r4] ; get value for d
  SUB r0,r0,r1 ; compute a-b
  LDR r4,=x ; get address for x
  STR r0,[r4] ; store value of x
  after ...

Machine code for Branch Instruction

- use a single 24 bit immediate operand (imm24)
- op: 10_2
- funct:
  - upper bit always 1 for branches
  - lower bit L: (i) 1 (BL), (ii) 0 (B)
- imm24 is used to specify an instruction address relative to PC+8

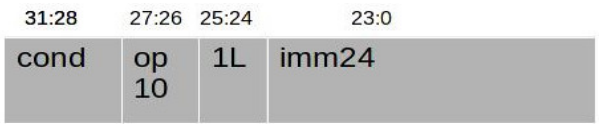

Figure 7

BLT THERE

- 0x80A0 BLT THERE
  0x80A4 ADD R0, R1,R2
  0x80A8 SUB R0, R0,R9
  0x80AC ADD SP,SP,#8
  0x80B0 MOV PC, LR
  0x80B4 THERE SUB R0, R0, #1
  0x80B8 ADD R3,R3,#0x5
- BTA (Branch Target Address): address of the next instruction to execute if the Branch is considered
- BTA for BLT: 0x80B4
- imm24: Total number of instruction between BTA and PC+8 (two instruction past the branch)
- Here,
  BTA= 0x80B4
  PC+8= 0x80A8
  imm24=3

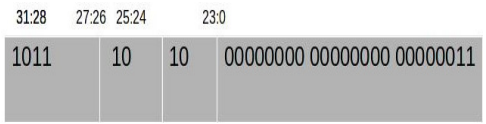

Figure 8

Conditional Executions

- (IF-THEN) instruction, IT, supports conditional execution in Thumb2 of up to 4 instructions in a "block"
- After an IT instruction is executed, up to four of the subsequent instructions can be conditionally executed based on the condition specified by the IT instruction and the APSR value.
- Format: IT $< x >< y >< z >< condition >$, where x,y,z are T/E/blank

Table 1: IT instruction Block of Various Sizes

Number of Conditions	IT block	Examples
One cond instr	IT $<$ *cond* $>$; instr1 $<$ *cond* $>$	IT EQ;ADDEQ R0, R0, R1
Two cond instr	IT $< x > <$ *cond* $>$; instr1$<$ *cond* $>$, instr2$<$ *cond* $>$	ITE GE ; ADDGE R0, R0, R1;ADDLT R

- $< x >$: specifies the execution condition for the second instruction
- $< y >$: specifies the execution condition for the third instruction
- $< z >$: specifies the execution condition for the fourth instruction
- $<$ *cond* $>$: specifies the base condition of the instruction block: the first instruction following IT executes if $<$ *cond* $>$ is true

Subroutine/Function

- Branch and link instruction:
  BL foo ;copies current PC to R14.
- To return from subroutine:
  BX R14 ; branch to address in R14
- Unconditional subroutine call
  - BL subRoutine_A; branch to subRoutine_A with link save return address in R14
- Conditional subroutine call
  - CMP R1,R2; branch conditionally
    BLLT subRoutine_A; branch to subRoutine_A if $R1 < R2$
    BX LR; return to the calling function

Subroutine Example

- Nested function calls in C:
  void f1(int a)
  f2(a);
  void f2 (int r)
  int g;
  g = r+5;
  main ()
  f1(xyz);
- Nesting/recursion requires a "coding convention" to save/pass parameters:
  AREA Code1,CODE
  Main LDR r13,=StackEnd ;r13 points to last element on stack
  MOV r1,#5 ;pass value 5 to func1
  STR r1,[r13,#-4]! ; push argument onto stack
  BL func1 ; call func1()
  here B here

Subroutine Example: contd

- ; void f1(int a) ; f2(a);
  Func1 LDR r0,[r13] ; load arg a into r0 from stack
  ; call func2()
  STR r14,[r13,#-4]! ; store func1 return address
  STR r0,[r13,#-4]! ; store arg to f2 on stack
  BL func2 ; branch and link to f2
  ; return from func1()
  ADD r13,#4 ; "pop" func2's arg off stack
  LDR r15, [r13],#4 ; restore stack and return

Subroutine Example: contd

```
; void f2 (int r)
; int g;
; g = r+5;
Func2 ldr r4,[r13] ;get argument r from stack
add r5,r4,#5 ;r5 = argument g
BX r14 ;preferred return instruction
; Stack area
AREA Data1,DATA
Stack SPACE 20 ;allocate stack space
StackEnd
END
```

Bit-field processing instruction

- New Thumb 2 instructions
- Bit field insert or clear (to pack or unpack data within a register)
  - BFC $R_0, 5, 4$;Clear 4 bits of r0, starting with bit 5
  - BFI $R_0, R_1, 5, 4$;Insert 4 bits of r1 into r0, start at bit 5
  - RBIT R_0, R_1 ; Reverse bit order of value in R1 and write the result to R0.
    - lets $R_1 = 10110100111000010000110000100011$
    - after executing the instruction:
      $R_0 = 11000100001100001000011100101101$
  - REV R_3, R_7 ; Reverse byte order of value in R7 and write it to R3