Week 3. ARM Assembly Programming

18-342: Fundamentals of Embedded Systems

Rajeev Gandhi
INI & ECE
Carnegie Mellon University

Carnegie Mellon

Electrical & Computer ENGINEERING

Overview of this Week

- Conditional execution of instructions
  - Improves code density
  - Can improve performance
- Load/Store instructions
  - Different addressing modes
- Multiple register load and store instructions
- Performing stack operations using load/store instructions
- Digging deeper
  - Limitations of branch instructions
  - Loading arbitrary 32-bit constants in registers

- Announcement:
  - Quiz onThursday

Value of Conditional Execution

- This improves code density *and* performance by reducing the number of forward branch instructions

```
if (x != 0)
  a = b+c;
else
   a = b-c;
```

```
   CMP   r3,#0
   BEQ   skip
   ADD   r0,r1,r2
   B afterskip
skip
   SUB   r0, r1, r2
afterskip
```

```
CMP   r3,#0
ADDNE r0,r1,r2
SUBEQ r0,r1,r2
```

- Can also be used to optimize a countdown loop (where the loop variable decrements from a positive number to zero)

```
loop
  . . .
 SUB r1,r1,#1
 CMP r1, #0
 BNE loop
```

```
loop
...
SUBS r1,r1,#1
BNE loop
```

3

Examples of Conditional Execution

- Use a sequence of several conditional instructions

```
if (a==0) x=1;
   CMP      r0,#0
   MOVEQ    r1,#1
```

- Set the flags, then use various condition codes

```
if (a==0) x=0;
if (a>0)  x=1;
   CMP      r0,#0
   MOVEQ    r1,#0
   MOVGT    r1,#1
```

- Use conditional compare instructions

```
if (a==4 || a==10) x=0;
   CMP      r0,#4
   CMPNE    r0,#10
   MOVEQ    r1,#0
```

4

Single Register Data Transfer

- ARM is based on a "load/store" architecture
  - All operands should be in registers
  - Load instructions are used to move data from memory into registers
  - Store instructions are used to move data from registers to memory
  - Flexible – allow transfer of a word or a half-word or a byte to and from memory

```
LDR/STR         Word
LDRB/STRB       Byte
LDRH/STRH       Halfword
LDRSB           Signed byte load
LDRSH           Signed halfword load
```

- Syntax:
  - `LDR{<cond>}{<size>} Rd, <address>`
  - `STR{<cond>}{<size>} Rd, <address>`

5

LDR and STR

- `LDR` and STR instructions can load and store data on a boundary alignment that is the same as the datatype size being loaded or stored
- `LDR` can only load 32-bit words on a memory address that is a multiple of 4 bytes – 0, 4, 8, and so on

- `LDR r0, [r1]`
  - Loads register r0 with the contents of the memory address pointed to by register r1

- `STR r0, [r1]`
  - Stores the contents of register r0 to the memory address pointed to by register r1

- Register r1 is called the base address register

6

Addressing Modes

- ARM provides three addressing modes
  - Preindex with writeback
  - Preindex
  - Postindex
- Preindex mode useful for accessing a single element in a data structure
- Postindex and preindex with writeback useful for traversing an array

7

Addressing Modes

- Preindex
  - Same as preindex with writeback, but does not update the base register
  - Example: `LDR r0, [r1, #4]`
- Preindex with writeback
  - Calculates address from a base register *plus* address offset
  - Updates the address in the base register with the new address
  - The *updated base register value* is the address used to access memory
  - Example: `LDR r0, [r1, #4]!`
- Postindex
  - Only updates the base register *after* the address is used
  - Example: `LDR r0, [r1], #4`

8

Examples

```
PRE   r0 = 0x00000000
      r1 = 0x00009000
      mem32[0x00009000] = 0x01010101
      mem32[0x00009004] = 0x02020202
```

Preindexing with writeback

```
LDR r0, [r1, #4]!

POST  r0 =
      r1 =
```

Preindexing

```
LDR r0, [r1, #4]

POST  r0 =
      r1 =
```

Postindexing

```
LDR r0, [r1], #4

POST  r0 =
      r1 =
```

More on Addressing Modes

- Address `<address>` accessed by LDR/STR is specified by
  - A base register plus an offset
- Offset takes one of the three formats
  1. Immediate: offset is a number that can be added to or subtracted from the base register

     Example: `LDR r0,[r1, #8];` r0 ⇦ mem[r1+8]

     `LDR r0,[r1, #-8];` r0 ⇦ mem[r1-8]

  2. Register: offset is a general-purpose register that can be added to or subtracted from the base register

     Example: `LDR r0,[r1, r2];` r0 ⇦ mem[r1+r2]

     `LDR r0,[r1, -r2];` r0 ⇦ mem[r1-r2]

  3. Scaled Register: offset is a general-purpose register shifted by an immediate value and then added to or subtracted from the base register

     Example: `LDR r0,[r1,r2, LSL #2];` r0 ⇦ mem[r1+4*r2]

10

Multiple-Register Transfer

- Load-store-multiple instructions can transfer multiple registers between memory and the processor in a single instruction
- Advantages
  - More efficient than single-register transfers for moving blocks of data around memory
  - More efficient for saving and restoring **context** and **stacks**
- Disadvantages
  - ARM does not interrupt instructions when executing ⇨ load-store multiple instructions can increase interrupt latency
- Compilers can limit interrupt latency by providing a switch to control the max number of registers that can be transferred on a load-store-multiple

`LDM`<cond><addrMode> `Rn{!}`, <registerList>`{^}`
`STM`<cond><addrMode> `Rn{!}`, <registerList>`{^}`

11

More on Load-Store-Multiple

- Transfer occurs from a base-address register `Rn` pointing into memory
- Transferred registers can be either
  - Any subset of the current bank of registers (default)
  - Any subset of the user mode bank of registers when in a privileged mode (postfix instruction with a '^')
    - Processor not in user mode or system mode
    - Writeback is not possible, i.e., `!` cannot be supported at the same time
    - If `pc` is in the list of registers, additionally copy `spsr` to `cpsr`
- Register `Rn` can be optionally updated following the transfer
  - If register `Rn` is followed by the `!` character
- Registers can be individually listed or lumped together as a range
  - Use a comma with "{" and "}" parentheses to list individual registers
  - Use a "-" to indicate a range of registers
  - Good practice to list the registers in the order of increasing register number (since this is the usual order of memory transfer)

12

Addressing Modes for Load-Store-Multiple

- Suppose that `N` is the number of registers in the list of registers

- `IA` (increment after)
  - Start reading at address `Rn`; ending address is `Rn + 4N – 4`
  - `Rn!` equals `Rn + 4N`
- `IB` (increment before)
  - Start reading at address `Rn+4`; ending address is `Rn + 4N`
  - `Rn!` equals `Rn + 4N`
- `DA` (decrement after)
  - Start reading at address `Rn – 4N + 4`; ending address is `Rn`
  - `Rn!` equals `Rn – 4N`
- `DB` (decrement before)
  - Start reading at address `Rn – 4N`; ending address is `Rn – 4`
  - `Rn!` equals `Rn – 4N`
- ARM convention: DB and DA are like loading the register list backwards from sequentially descending memory addresses

13

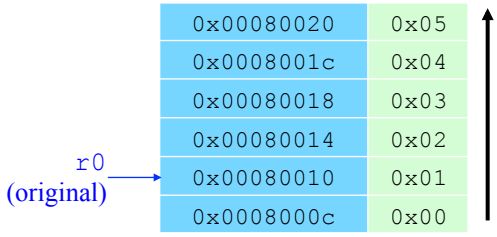
Things to Remember

- Any register can be used as the base register
- Any register can be in the register list
- Order of registers in the list does not matter
- The lowest register always uses the lowest memory address *regardless of the order in which registers are listed in the instruction*
- LDM and STM instructions only transfer words
  - Unlike LDR/STR instructions, they don't transfer bytes or half-words
- Can specify range instead of individual registers
  - Example: `LDMIA r10!, {r12, r2-r7}`
- If the base register is updated (using `!`) in the instruction, then it cannot be a part of the register set
  - Example: `LDMIA r10!, {r0, r1, r4, r10}` is not allowed

14

Examples

```
PRE     r0 = 0x00080010
        r1 = 0x00000000
        r2 = 0x00000000
        r3 = 0x00000000
        mem32[0x8001c] = 0x04
        mem32[0x80018] = 0x03
        mem32[0x80014] = 0x02
        mem32[0x80010] = 0x01
```




```
LDMIA r0!, {r1-r3}

POST    r0 =
        r1 =
        r2 =
        r3 =
```

```
LDMIB r0!, {r1-r3}

POST    r0 =
        r1 =
        r2 =
        r3 =
```

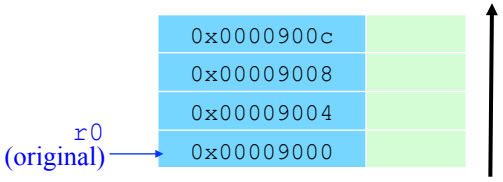
15

Example 1: Saving & Restoring Registers

- Here's what we want to accomplish
  - Save the contents of registers `r1`, `r2` and `r3` to memory
  - Mess with the contents of registers `r1`, `r2` and `r3`
  - Restore the original contents of `r1`, `r2` and `r3` from memory & restore `r0`

```
PRE     r0 = 0x00009000
        r1 = 0x09
        r2 = 0x08
        r3 = 0x07

; store contents to memory
STMIB r0!, {r1-r3}
; mess with registers r1, r2, r3
MOV r1, #1
MOV r2, #2
MOV r3, #3
; restore original r1, r2, r3
LDMDA r0!, {r1-r3}
```



ARM convention: Highest memory location maps to highest numbered register

16

Example 1: Block Copying

- Here's what we want to accomplish
  - Copy blocks of 32 bytes from a source address to a destination address
  - `r9` points to the start of the source data
  - `r10` points to the start of the destination data
  - `r11` points to the end of the source data

```
loop
    ; load 32 bytes from source address and update r9 pointer
    LDMIA r9!, {r0-r7}
    ; store 32 bytes to destination address and update r10 pointer
    STMIA r10!, {r0-r7}
    ; check if we are done with the entire block copy
    CMP r9, r11
    ; continue until done
    BNE loop
```

17

Stack Operations

- ARM uses load-store-multiple instructions to accomplish stack operations
- Pop (removing data from a stack) uses load-multiple
- Push (placing data on a stack) uses store-multiple
- Stacks are ascending or descending
  - Ascending (A): Grow towards higher memory addresses
  - Descending (D): Grow towards lower memory addresses
- Stacks can be full or empty
  - Full (F): Stack pointer `sp` points to the last used or full location
  - Empty (E): Stack pointer `sp` points to the first unused or empty location
- Four possible variants
  - Full ascending (FA) – `LDMFA` & `STMFA`
  - Full descending (FD) – `LDMFD` & `STMFD`
  - Empty ascending (EA) – `LDMEA` & `STMEA`
  - Empty descending (ED) – `LDMED` & `STMED`

18

Stacks on the ARM

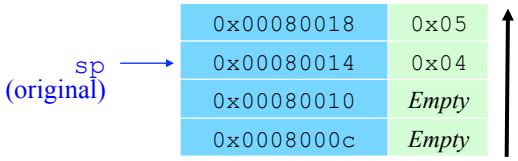
- ARM has an ARM-Thumb Procedure Call Standard (ATPCS)
  - Specifies how routines are called and how registers are allocated
- Stacks according to ATPCS
  - Full descending
- What does this mean for you?
  - Use `STMFD` to store registers on stack at procedure entry
  - Use `LDMFD` to restore registers from stack at procedure exit
- What do these handy aliases actually represent?
  - `STMFD` = `STMDB` (store-multiple-decrement-before)
  - `LDMFD` = `LDMIA` (load-multiple-increment-after)

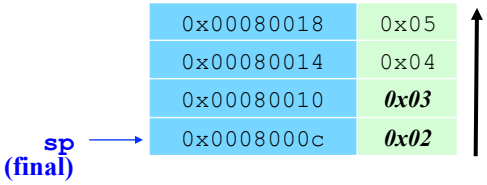
19

Example

```
PRE     r1 = 0x00000002
        r4 = 0x00000003
        sp = 0x00080014


STMFD   sp!, {r1, r4}
```





20

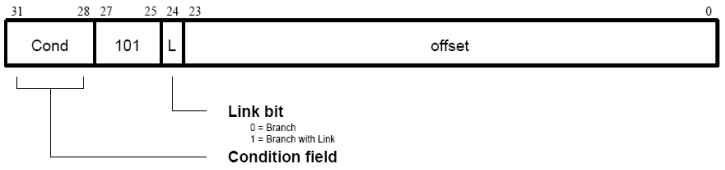
Stack Checking

- Three stack attributes to be preserved
- Stack base
  - Starting address of the stack in memory
  - If `sp` goes past the stack base, stack underflow error occurs
- Stack pointer (`sp`)
  - Initially points to the stack base
  - As data is inserted when a program executes, `sp` descends memory and points to top of the stack
- Stack limit (`sl`)
  - If `sp` passes the stack limit, a stack overflow error occurs
  - ATPCS: `r10` is defined as `sl`
    - If `sp` is less than `r10` after items are pushed on the stack, stack overflow occurs

21

Digging Deeper

- Branch instructions

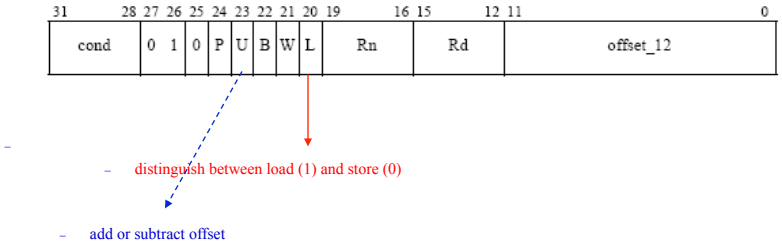


  - When executing the instruction, the processor
    - Shifts the offset left two bits, sign extends it to 32 bits, and adds it to the `pc`
  - This gives a 26-bit offset from the current instruction
  - What is the maximum distance (in bytes) that we can jump to, in a branch?

22

LDR Instruction

- `LDR rd, [rn, #offset]` `rd ← mem[rn+offset]`
  - Decimal numbers prefixed by #
- `rd, rn` can be any register (r0 – r15)
- Binary encoding of `LDR/STR` instruction (with immediate offset addressing mode)



LDR (contd.)

- `LDR` can be used for branches beyond the range of `BL` instruction
  - Example: `LDR pc, [pc, #offset]` `pc ← mem[pc+offset]`
  - The address of the branch should be stored in memory location `curraddr +offset+8`
- Example

```
    0x10000000      add r0, r1, r2
    0x10000004      ldr pc, [pc, #4];
    0x10000008      sub r1, r2, r3
    0x1000000c      cmp r0, r1
    0x10000010      0x20000000
    …
    …
Branch_target
    0x20000000      str r5, [r13, -#4]!
    …
    …
```

- `STR/LDR` instructions can be used to save/restore registers on function entry/exit

Digging Deeper: Loading Constants

- There is no single instruction which will load a 32 bit immediate constant into a register without performing a data load from memory.
  - All ARM instructions are 32 bits long
  - ARM instructions do not use the instruction stream as data
- The data processing instruction format has 12 bits available for operand2
  - If used directly, this would only give a range of 4096 bytes
- Instead it is used to store 8-bit constants, giving a range of 0 - 255
- These 8 bits can then be rotated right through an even number of positions
  - This gives a much larger range of constants that can be directly loaded, though some constants will still need to be loaded from memory

25

Specific Example: MOV

- MOV or MVN data processing instruction can be used to load 8-bit numbers into registers
  - `MOV r0, #0x07 ;r0=0x00000007`
- Use MOV with the barrel shifter to load more than 8-bit numbers into registers
  - `MOV r0, #0x0F, #2 ; r0=0xC0000003`
- Remember operand2 in MOV instruction takes 12 bits
  - 8 bits are used for the immediate value
  - 4 bit rotate value (which should be an even number between 0 and 30)
- Rule to remember is "8-bits rotated right by an even number of bit positions".

26

Loading Constants: Pseudo-Instruction LDR

- To allow larger constants to be loaded, the assembler offers a pseudo-instruction:
  - `LDR Rd, =const`
- This will either:
  - Produce a `MOV` or `MVN` instruction to generate the value (if possible)

  or
  - Generate a `LDR` instruction with a `pc`-relative address to read the constant from a *literal pool* (constant data area embedded in the code)
  - What is a literal pool?
    - Portion of memory used by the assembly code to store constants
- This is the recommended way of loading constants into a register

27

Loading Constants – Example

Original assembly code

```
LDR r0, =0x55555555
LDR r1, =0x4867
ADD r2, r1, r0
SWI 0x123456
```

Machine code generated by assembler

```
0x00000000:    e59f0008
0x00000004:    e59f1008
0x00000008:    e0812000
0x0000000c:    ef123456
0x00000010:    55555555
0x00000014:    00004867
```

- Assembler places the constant `0x55555555` at location `0x00000010`
- and converts the pseudo-instruction to `LDR r0, [pc, #8]`
- Assembler places the constant `0x4867` at location `0x00000014`
- and converts the pseudo-instruction to `LDR r0, [pc, #8]`

28