SPRING 2016

CMPE 364

Microprocessor Based Design

Dr. Mohamed Al-Meer

Memory Access Instructions

ARM Architecture

Lecture Expectations

- Expected to achieve:
  - Know about **Memory model** and **Endians**
  - Know about **Addressing Modes**
  - Know **Single Register Load-Store** Instructions.
  - Know about **Loading Constants**
  - Know about **Multiple Registers Load-Store** Instructions
  - Know about The **Stack**.
  - Test Yourself and **solve problems**

Memory Access Instructions

- Each memory storage location has an address that identify uniquely that location.
- Load instruction copies data from memory location into a register
- Store instruction copies data from a register to memory location
- Load/Store instructions in ARM can move a byte (8-bit), half word (16-bit), or a word (32-bit)

Memory Model

- Each memory location is specified in 32-bit address
- An address in ARM ranges from 0x00000000 to 0xFFFFFFFF
- Provides 2^{32} memory locations or 4G Bytes, each can store 1 byte only.
- Words and half words are multi-byte structures (span multiple continuous bytes in memory)
- When access half-words and words, there are some rules.
  - Half-word address should be divisible by 2.
  - Word address should be divisible by 4.

Endians

- One word saved in memory address occupies 4 addresses
- There are 2 ways for a byte in a multi byte structure may be ordered in memory (endians)
  - **Big Endian**
  - **Little Endian**
- **Big endian**
  - When the most significant byte of a word is stored at the word address
  - Big end of the word is stored at the word address

Endians

- **Little endian**
  - Store Least significant byte of the word at the word address.
  - Little End of the word is stored at the word address.
- Same concepts applies to Half-Words
  - Big Endian: Most significant byte of the half-word is stored at the half word address
  - Little Endian: Least significant byte of the half-word is stored at the half-word address.
- ARM can be configured to operate on both systems
- But **Little Endian** is the **default** unless explicitly stated.

Endians (Examples)

- What word is stored at memory address 0x08000114 when **big endian** is used?
  - 0x08000114: CA
  - 0x08000115: 50
  - 0x08000116: 02
  - 0x08000117: 00
- **Answer**
  - Word = 0xCA500200

Endians (Examples)

- What word is stored at memory address 0x08000114 when **little endian** is used?
  - 0x08000114: CA
  - 0x08000115: 50
  - 0x08000116: 02
  - 0x08000117: 00
- **Answer**
  - Word = 0x000250CA

Addressing Modes

- Every Load-Store instruction must specify the address of memory location being accessed
- ARM cannot include 32-bit hardcoded address within the 32-bit instruction op-code
- So to solve this problem, **addressing modes**, will be used.
- Processor uses information supplied with addressing modes to calculate the **Effective Address**.

```
Syntax: <LDR|STR>{<cond>}{B} Rd,addressing[1]
        LDR{<cond>}SB|H|SH Rd, addressing[2]
        STR{<cond>}H Rd, addressing[2]
```

Addressing Modes: Syntax

LDR	load word into a register	*Rd <- mem32[address]*
STR	save byte or word from a register	*Rd -> mem32[address]*
LDRB	load byte into a register	*Rd <- mem8[address]*
STRB	save byte from a register	*Rd -> mem8[address]*
LDRH	load halfword into a register	*Rd <- mem16[address]*
STRH	save halfword into a register	*Rd -> mem16[address]*
LDRSB	load signed byte into a register	*Rd <- SignExtend (mem8[address])*
LDRSH	load signed halfword into a register	*Rd <- SignExtend (mem16[address])*

AM: Register Indirect

- Uses a register to hold the 32-bit address of the operand.
- Syntax is **[rm]** where rm is the register storing the operand address.
- **EA = Base Address = (rm)**
- Example
  - Get the Effective Address for next Instruction, where R1=0x004000A4
  - LDR R2, [R1]
- Solution
  - EA = 0x004000A4

AM: Pre Indexed Addressing

- **EA = Base address + Offset**
- Utilizes a register to store the base address and adds or subtracts an offset to obtain effective address.
- Offset could be
  - Immediate syntax: **[rn, #+/- immediate]**
  - Register syntax: **[rn, +/- rm]**
  - Shifted value syntax: **[rn, +/- rm, shift # shift_amount]**
- Shift types
  - LSL, LSR, ASR, ROR, RRX

AM: Pre Indexed Addressing

- Example
  - Get the operand's effective address from the next instruction if R1 = 0x00400004?
  - **LDR R2, [R1, +#0xAC]**
- Solution
  - EA= 00400004 + 000000AC = 004000B0

AM: Pre Indexed Addressing

- Example
  - Get the EA for the next instruction if R0 = 0x008C0080, and R4 = 0x00000160?
  - **LDR R5, [R0, -R4]**
- Solution
  - 008C0080 – 00000160 = 0x008BFF20

AM: Pre Indexed Addressing

- Example
  - Get the EA if R2 =0x400AC000, and R9 = 0X0C010500?
  - **LDR R1, [R9, +R2 ASR #12**]
- Solution
  - Shifting **R2** to right 12 bit positions yields **000400AC**, and

0x0C010500 +
0x000400AC

0x0C0505AC

AM: Pre Indexed with Auto Indexing Addressing

- Effective Address is identical to pre-indexed mode but with a difference that: the **base address** is **updated before the memory access** takes place.
- EA = Base address + Offset
- ! (exclamation symbol) denotes that mode.
- Offset could be
  - Immediate syntax: **[rn, #+/- immediate]!**
  - Register syntax: **[rn, +/- rm]!**
  - Shifted value syntax: **[rn, +/- rm, shift # shift_amount]!**

AM: Pre Indexed with Auto Indexing Addressing

- Example
  - Get the EA if R5 = 0x0000400C
  - **LDR R11, [R5, - #8]!**
- Solution
  - R11 = Word in location **0x00004004**. EA = R5 – 8.
  - **R5** updated = **0x00004004**.

AM: Pre Indexed with Auto Indexing Addressing

- Example
  - Get EA used by the next instruction and any changes in register values if R0 = 00000018, R4 = 7EF5C004?
  - **LDR R9, [R4, +R0]!**
- Solution
  - EA = 7EF5C004 + 018 = 7EF5C01C
  - R0 Unchanged
  - R4 changed with EA.

AM: Pre Indexed with Auto Indexing Addressing

- Example
  - Get EA and content of R3 and R8 after the next instruction if R3 = 0x21 and R8 = 0x000EF080?
  - **LDR R4, [R8, +R3, LSL #5]!**
- Solution
  - EA = R8 + R3*32 = 000EF080 + 00000021 << 5
  - EA = 0x000EF4A0
  - R3 Unchanged
  - **R8** = EA = 0x000EF4A0

AM: Post Indexed with Auto Indexing Addressing

- Uses base register to store effective address
- **After** memory access an offset is added or subtracted to or from the base register
- Base register is updated
- EA = Base address + Offset
- Offset could be
  - Immediate syntax: **[rn], #+/- immediate**
  - Register syntax: **[rn], +/- rm**
  - Shifted value syntax: **[rn] +/- rm, shift # shift_amount**

AM: Post Indexed with Auto Indexing Addressing

- Example
  - Get the EA for next instruction if R2 = **0x035C0170**?
  - **LDR R0, [R2], #+48**
- Solution
  - EA = **0x35C0170**
  - After the execution R2 = 0x035C0170 + 48 = 0x035C01A0
  - R2 Changed = EA = 0x35C0170

AM: Post Indexed with Auto Indexing Addressing

- Example
  - Get the EA for next instruction and if R$ and R5 will change if R4 = **0x00050048**, and R5 = 0x00000024?
  - **LDR R8, [R4], -R5**
- Solution
  - EA = R4 = **0x00050048**
  - After execution **R4 =** 0x00050048 - 0x24 = **0x00050024**
  - R5 unchanged.

AM: Post Indexed with Auto Indexing Addressing

- Example
  - Get the EA for next instruction if R2 = **0x40000018**, R3 = 0x000000CE?
  - **LDR R10, [R2], R3, LSR #4**
- Solution
  - **EA** = R2 = **0x40000018**
  - After the execution **R2** = 0x40000018 + 0x000000CE >> 4
  - **R2 = 0x40000024**.
  - R3 Unchanged