Math and bit instructions, indirect memory use

Overview

- Math operations
  - Basic 8-bit math instructions
- Bit operations
  - Bitwise AND, OR, XOR, shifting
- Indirect memory tricks
  - Read from RAM
    - Array-like functionality
  - Read from code memory
    - Storing fixed data
    - Switch-case-like functionality

Math operations

- 8052 is an 8-bit microcontroller
  - Most math operations 8-bit
  - Multiplication the exception
- Supports only basic math operations
  - Add, subtract, multiply and divide

Adding numbers

- ADD A, *operand*
  - Add value in operand to the accumulator
  - Leave result in the accumulator, operand not effected
  - Ignores incoming Carry bit (C)
  - 1-2 bytes, 1 cycle

- ADDC A, *operand*
  - Add value in operand to the accumulator
  - Leave result in the accumulator, operand not effected
  - Uses incoming Carry bit (C)
  - 1-2 bytes, 1 cycle

Adding numbers

- ADD and ADDC details
  - If carry-out, Carry bit C set to 1
  - Carry out occurs if unsigned sum of A, operand and any incoming carry is > 255
  - Overflow bit (OV) set if sum is out of ranged of a signed byte (-128 through +127)

Subtraction

- SUBB A, *operand*
  - Subtract the value of operand from A
  - Leave the result in accumulator, operand not effected
  - Carry bit C set if borrow required (i.e. the unsigned operand being subtracted > A)
  - 1-2 bytes, 1 cycle

Division

- DIV *AB*
  - Divide unsigned value of accumulator (A) by the B register.
  - Resulting quotient placed in A
  - Remainder placed in B
  - Operand always "AB", no other choice
  - 1 byte, 4 cycles

Multiplication

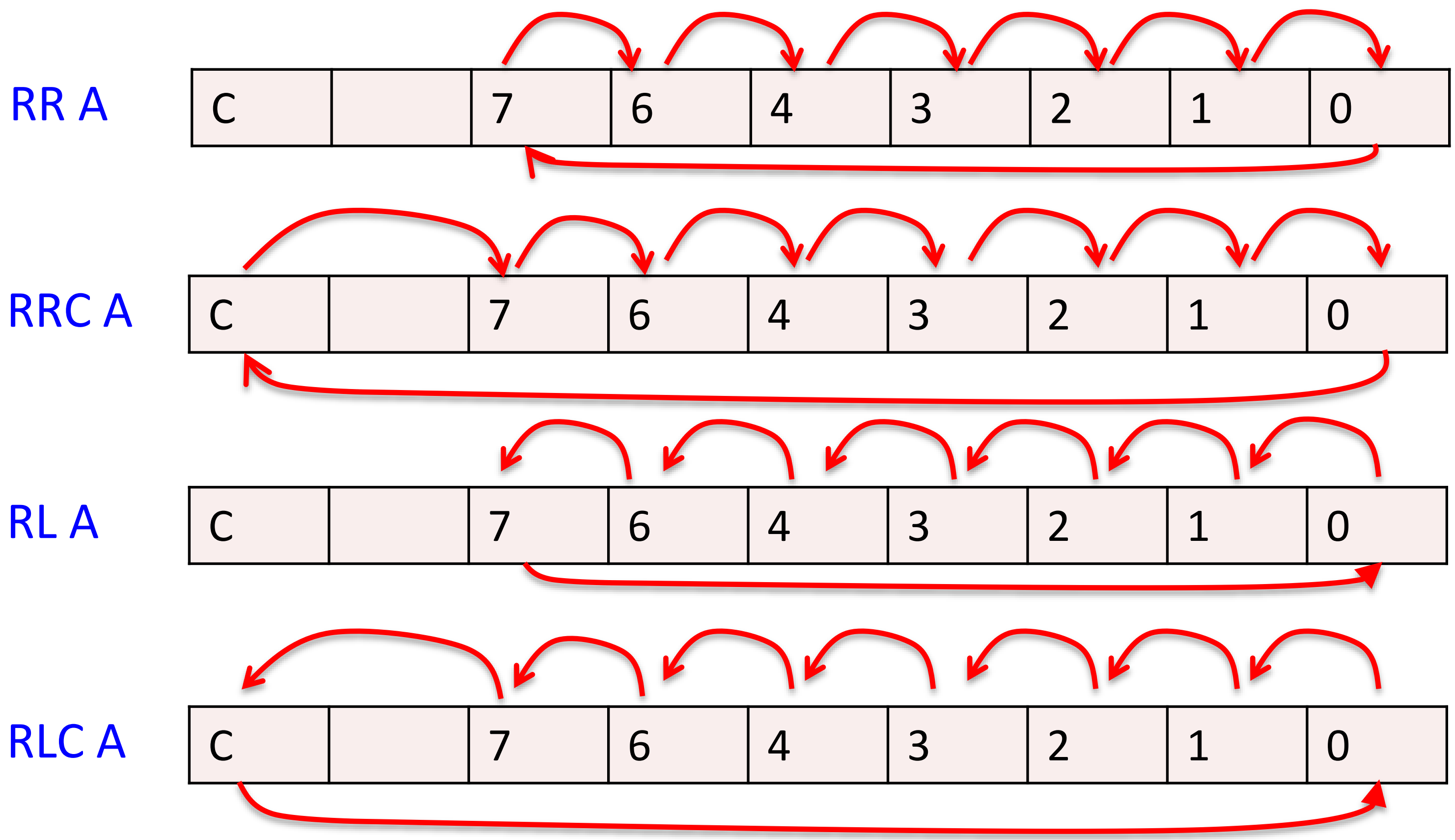
- MUL *AB*
  - Multiplies unsigned value in A by the B register
  - 16-bit result
    - Least significant byte in A
    - Most significant byte in B
  - Operand always "AB", no other choice
  - 1 byte, 4 cycles

Bit operations

- ORL *operand1, operand2*
  - Bitwise OR of 8-bit values
- ANL *operand1, operand2*
  - Bitwise AND of 8-bit values
- XRL *operand1, operand2*
  - Bitwise XOR of 8-bit values
- CPL *operand*
  - Bitwise complement, 1-bit address or Carry (C) bit
  - Or 8-bit value in Accumulator (A)

Bit shifting

- Rotate bit values in Accumulator (A)
  - Optionally rotate through Carry bit (C)



Rotating for fun and profit

- Rotates can be used to quickly:
  - Multiply by 2
  - Divide by 2 (dropping remainder)
  - Must take care to clear Carry bit (C)

```
; Multiply the accumulator by 2
; 4 bytes code, 6 cycles

MOV B, #2
MUL AB
```

```
; Multiply the accumulator by 2
; 2 bytes code, 2 cycles

CLR C
RLC A
```

Indirect addressing

- Indirect addressing
  - e.g. MOV A, @R0
  - Read the value of R0, obtain value at memory pointed to by R0
  - Allows us to get to second 128 bytes of RAM
  - Example:

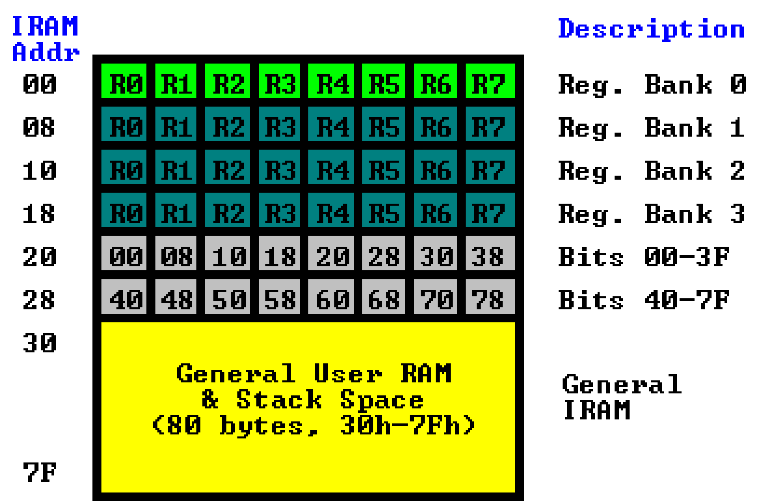
    MOV R0, #40h

    MOV A, @R0

    Register R0 holds value 40h, load accumulator with whatever is stored at RAM address 40h

Array-like maneuvers

- Use indirect addressing
  - Put memory address in register
  - Increment / decrement register
    - Moves around the block of memory representing the "array"
  - Get/set values using MOV and indirect addressing



Initializing an array

```
; Parameters to our ArrayInit subroutine
ArrayNum        EQU 30h    ; Where our array starts
ArrayMemStart   EQU 31h    ; 1st memory location in the array
ArrayInitVal    EQU 32h    ; What value to load into array

Start:
                MOV ArrayNum, #10
                MOV ArrayMemStart, #40h
                MOV ArrayInitVal, #0ABh
                CALL ArrayInit
                JMP Start

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Init the memory in an array to a value, uses R0 and R1
ArrayInit:
                MOV R0, ArrayNum
                MOV R1, ArrayMemStart
ArrayInitLoop:
                MOV @R1, ArrayInitVal
                INC R1
                DJNZ R0, ArrayInitLoop
                RET
```

Implementing switch-case logic

- Goal: run different code for a fixed set of values currently stored in A (0, 1, or 2)
- Option 1: use multiple CJNE instructions

```
; Run code based on whether the accumulator is 0, 1 or 2
        CJNE A, #0, Check1
        JMP A_IS0
Check1:
        CJNE A, #1, Check2
        JMP A_IS1
Check2:
        JMP A_IS2

A_IS0:  ...
A_IS1:  ...
A_IS2:  ...
```

Jump lists

- Option 2: use a jump list
  - Jump to a location in code based on value in A
  - Use DPTR since we need 2-bytes for code address

```
; Run code based on whether the accumulator is 0, 1 or 2
Start:
    MOV A, #2              ; Load the value we are testing
    RL A                   ; Double A, code addr = 2 bytes
    MOV DPTR, #JumpTable   ; Starting code address
    JMP @A+DPTR            ; Go go gadget jump
JumpTable:
    JMP A_IS0
    JMP A_IS1
    JMP A_IS2

A_IS0:  ...
A_IS1:  ...
A_IS2:  ...
```

Jump lists

- Why?
  - Saves code memory for 2+ case "switches"
  - Deterministic runtime
    - Same # of cycles regardless of value being tested
    - Not true for a repeated CJNE approach

Code indirect addressing

- MOVC A, @A+DPTR
- MOVC A, @A+PC
  - Moves byte from code memory into accumulator
  - Code memory address is:
    - Value in accumulator
    - Plus Data Pointer (DPTR) or Program Counter (PC).
  - In case of @A+PC form, PC is incremented by one before adding

Code indirect addressing

- Put table of fixed values in memory
- Read in programmatically MOVC A, @A + PC

```
; Copy a sequence stored in code memory to the LEDs
Start:
   MOV R0, #5

Loop:
        MOV A, R0
        MOV DPTR, #Values
        MOVC A,@A+DPTR
        MOV P0,A
        DJNZ R0, Loop
        JMP Start
Values:
        DB 00h,01h,02h,03h,04h,05h
```

Summary

- Math operations
  - We can add, subtract, multiple and divide
  - 8-bit numbers anyway
- Bit operations
  - Bitwise AND, OR, XOR, rotating bits
- Indirect memory tricks
    - Array-like functionality
    - Switch-case-like functionality
    - Storing fixed data in code memory