# EECS 113
# Lec. 4: 8051 ISA cont'd, addressing modes

Dept. of EECS, UC Irvine

# Week 2

- Reading
  - (roughly) Chapter 2, Catsoulis's book
  - 8051 ISA reference
- Assignment
  - more assembly exercises
  - install SDCC

# CY flag (C-bit)

- Carry out of the highest-order adder
  - Does not mean overflow!!
    Example:   0x01 + 0xFF = 0x00 with C=1
    (corresponds to 1 + (-1) = 0.)
  - Carry or borrow, both have C=1
- Carry bit can be set, cleared, moved
  - CLR  C    ;; clear the C bit, means := 0
    SETB  C   ;; set the C bit, means := 1

# Bit Assignment: setb/clr

- Set/Clear the carry flag C
  - SETB C ;; carry=1
  - CLR C ;;carry=0
- Set/Clear a general bit (e.g., P1.2)
  - SETB bit ;; bit = 1
  - CLR bit ;; bit = 0

# Bit Assignment: MOV

- Between C and a bit register (e.g., P1.2)
  - MOV   C, *bit*
  - MOV   *bit*, C
- But cannot move between two explicitly addressed bit registers or literals!
  - MOV  P1.2,  P2.3  ;; this is illegal!
  - MOV  C, #1     ;; this is illegal! use SETB

# Conditional Jumps

- Jump based on carry flag
  - JC     target    ;; jump to target if C=1
  - JNC  target    ;; jump to target if C=0
- Jump based on general bit register (e.g. P1.2)
  - JB    *bit*, target
  - JNB  *bit*, target

# Instructions

## Single-bit

| Instruction | Function |
|---|---|
| SETB bit | Set the bit (bit = 1) |
| CLR bit | Clear the bit (bit = 0) |
| CPL bit | Complement the bit (bit = NOT bit) |
| JB bit,target | Jump to target if bit = 1 (jump if bit) |
| JNB bit,target | Jump to target if bit = 0 (jump if no bit) |
| JBC bit,target | Jump to target if bit = 1, clear bit (jump if bit, then clear) |

## Reading input port

| Mnemonic | Example | Description |
|---|---|---|
| `MOV A, PX` | `MOV A, P2` | Bring into A the data at P2 pins |
| `JNB PX.Y,..` | `JNB P2.1, TARGET` | Jump if pin P2.1 is low |
| `JB PX.Y,..` | `JB P1.3, TARGET` | Jump if pin P1.2 is high |
| `MOV C, PX.Y` | `MOV C, P2.4` | Copy status of pin P2.4 to CY |

# Example: polling

```
        SETB    P1.2
AGAIN:  JNB     P1.2, AGAIN
```

- Configure P.1 for input
- keep looping as long as P1.2 == 0
  => waits till rising edge of P1.2
- Polling: keep checking I/O in a loop
  Easy, fast, but a little wasteful
- more "efficient" ways
  - use interrupts or counters

# Use C as a "bit accumulator"

- Carry flag (C) can be used as a bit register
- Alternative to previous polling code
  Use MOV and JC or JNC instructions

```
        SETB  P1.2       ;; set it 1 for input
AGAIN:  MOV   C, P1.2    ;; read pin into C
        JNC   AGAIN      ;; if (!C) repeat
```

# PSW: program status word

- 8-bit register containing flags
  - indicating status of the processor

| | | |
|---|---|---|
| CY (C bit) | PSW.7 | Carry flag |
| AC | PSW.6 | Auxiliary carry, for BCD arithmetic |
| F0, -- | PSW.5, .1 | (user) |
| RS1 | PSW.4 | Register bank select |
| RS0 | PSW.3 | |
| OV | PSW.2 | Overflow |
| P | PSW.0 | Parity: even or odd# of 1's in A |

# Example: want to copy input bit to output

- Input P1.0, want to copy bit value to P2.7
- Cannot do
  MOV P2.7, P1.0
  => no such instruction!
- Solution: use C as temporary
  MOV C, P1.0
  MOV P2.7, C

# OV-flag (overflow)

- Overflow: too big/too small to represent
- ADD:
  - Both operands same sign, sum different
    e.g., both +, sum -; or both -, sum +
  - Cannot overflow when operands mix + -
    Because the sum is between the two
- SUBB:
  - opposite condition of ADD

# SUB vs. SUBB

- SUB (subtract): doesn't exist on 8051
- SUBB: subtract with borrow (like ADDC)
- Actually, CLR C, then SUBB is same as SUB
- To implement SUB A, arg  in 8051:
  - take 2's complement of arg
  - A + (-arg) by ADDC
  - Complement CY

# Multiplication

- AB := A * B
  - 8 bits each, together as 16-bit product
  - A = lower order, B = higher order
- Assumption: unsigned numbers!
  - MOV  A, #25H
    MOV  B, #65H
    MUL   AB

# Division

- A, B := A / B,  A % B  (python syntax)
  - input: A = numerator, B = denominator
  - output: A = quotient, B = remainder
- Example
  - ```
    MOV A, #95
    MOV B, #10
    DIV   AB
    ```
    ```

# 2's complement

- To negate a number
- Invert all bits and then add 1
- Assume number is in A, easiest

```
CPL   A
INC  A
```

- If number is in register e.g., R1

```
CLR    A        ;; A = 0
CLR    C        ;; CY = 0
SUBB  A, R1     ;; A = -R1
```

# Bitwise vs Bytewise logical instructions

- Same mnemonics for ANL, ORL, CPL
- Bit version not available for XRL C, *bit*

| operator | Bit version | Byte version |
|---|---|---|
| AND | ANL C, *bit* | ANL A, *byte* |
| OR | ORL C, *bit* | ORL A, *byte* |
| complement | CPL C | CPL A |
| | CPL *bit* | |
| XOR | N/A!! | XRL A, *byte* |

# CJNE

- Compare and Jump if not Equal
- Syntax: CJNE *arg1*, *arg2*, *offset*
  - *arg1*: A or Reg
  - *arg2*: reg, dir, #imm
  - *offset*: +127 to -128
- Side effect: set CY if arg1 $<$ arg2!!!

# Rotate instruction

- Shift by 1 position with wrap-around
- Four versions, all use the accumulator
  - RL  A  (rotate left)
  - RR  A  (rotate right)
  - RLC  A  (rotate left thru CY)
  - RRC  A  (rotate right thru CY)

# Application of RLC/RRC

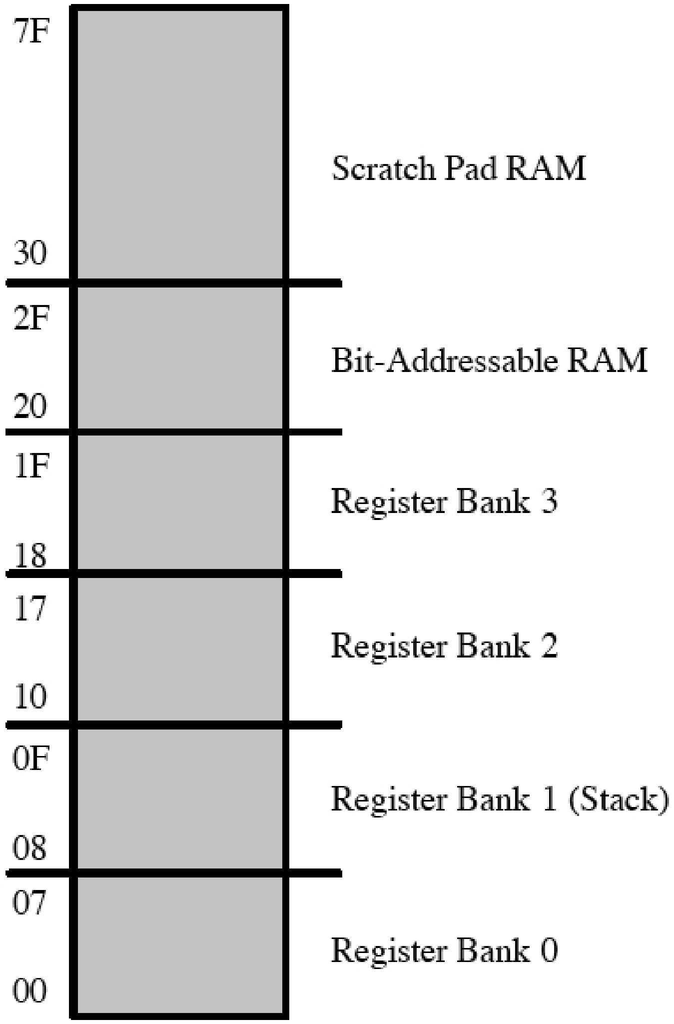
- Count number of bits in a byte
  - Put byte into A
  - RLC A 8 times
  - JNC to test C and increment count

# SWAP instruction

- Syntax:  SWAP A
- Meaning: swap two nibbles in A
  - Like rotate w/out going thru C 4 times!
- Use of SWAP
  - Quick extraction of nibbles

# Register banks and Scratchpad memory

- Four register banks [00-1FH]
  - 8 registers per bank (8 bytes/bank)
  - Bank 1 is the stack
- Bit-addressable (16 bytes) [20-2FH]
- Scratchpad (Byte-addressable) [30-7FH]
- Total:   128 bytes [00-7FH]



# Register bank selection

- Registers can be from 1 of 4 banks
  - Depending on PSW.4, PSW.3
- Example
  - MOV  R0,  #12H
    Where is R0?

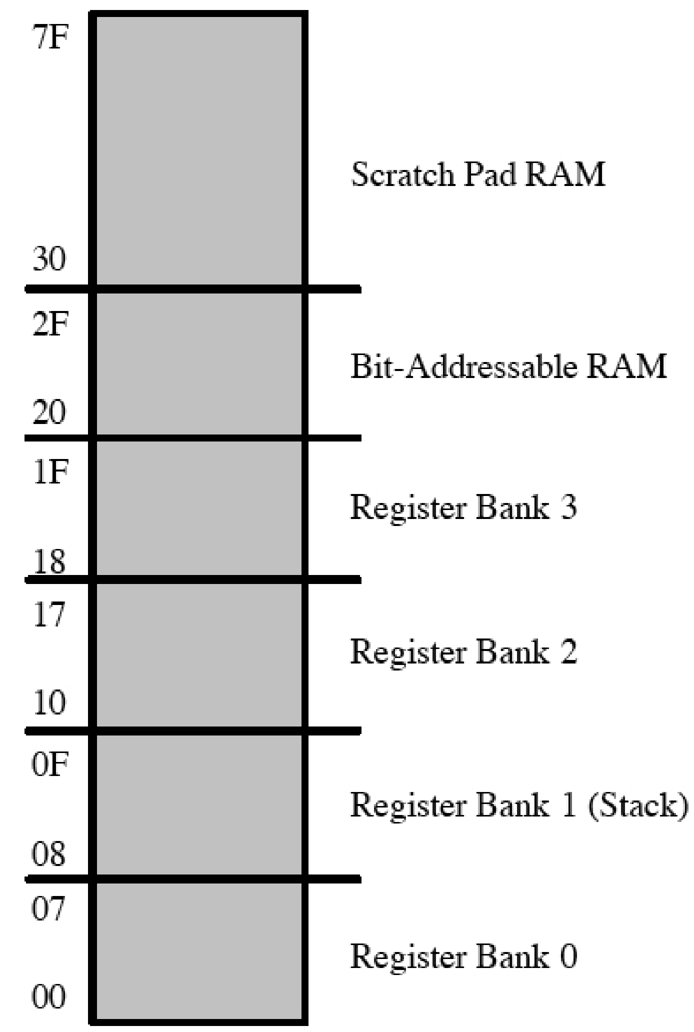
| | RS1 (PSW.4) | RS0 (PSW.3) |
|---|---|---|
| Bank 0 | 0 | 0 |
| Bank 1 | 0 | 1 |
| Bank 2 | 1 | 0 |
| Bank 3 | 1 | 1 |

- Answer: depends on which bank!
  - Could be address 00H, 08H, 10H, or 18H

# Instruction for setting banks

- "Set a bit"  (means assign the bit to 1)
  - SETB    PSW.4
- "Clear a bit"  (means assign the bit to 0)
  - CLR      PSW.3
- So, together, PSW.4=0, PSW.3=1
  => selects Bank 1

# Two ways of accessing the same registers




- Register mode:
  - Use R0, R1, ... name
  - Current bank
  - Need bank switching
- Direct mode:
  - Use the RAM address of the register!
  - 00 for *R0 of bank 0*
    12 for *R2 of bank 3*... etc

# Examples

## Register addressing:

SETB PSW.4

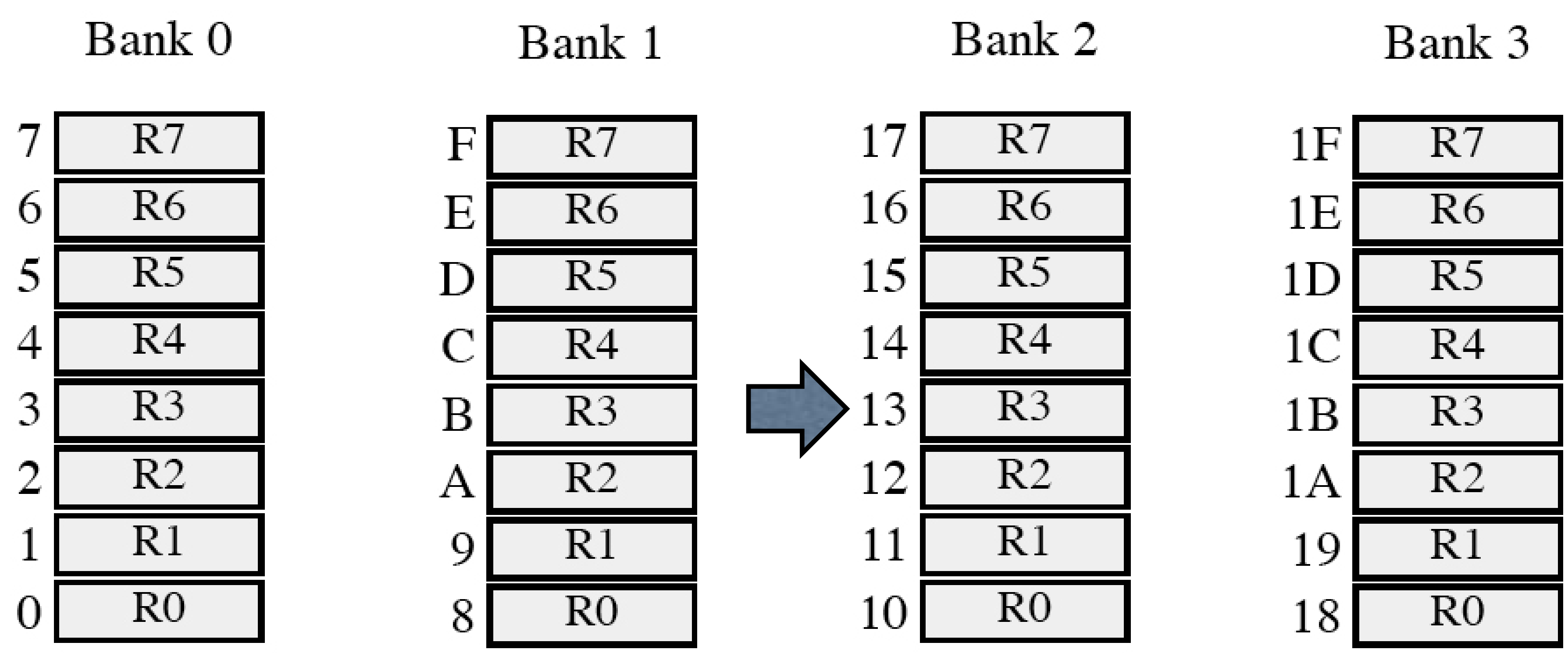
CLR PSW.3

MOV R3, #99H

## Direct addressing:

MOV 13, #99H

| PSW.4, PSW.3 | 00 | 01 | 10 | 11 |
|---|---|---|---|---|



# 8051 Memory spaces

- On-Chip: 1-byte pointer
  - 00-7FH: Register, bit-memory, scratchpad
  - 80-FFH: special function registers (direct)
  - 80-FFH: more scratchpad (indirect only) 8052-only extra 128 bytes; not on 8051
- Off-Chip: 2-byte pointer
  - 0000-FFFFH: data memory (MOVX)
  - 0000-FFFFH: code memory (MOVC)

# Addressing mode: way of specifying operand

- An instruction consists of
  - Opcode    (e.g., ADD, MOV, ...)
  - Operand(s)(e.g., R3, #23H, ...)
- Where is the (value of) operand located?
  - part of the instruction (immediate)
  - in data memory (direct, indirect, indexed)
  - in register;   also, in a bit of a register/pin

# Review: Registers

- 8-bit registers
  - General purpose: R0, R1, ... R7
  - Special function (SFR): A, B, PSW, SP, I/O ports...
- 16-bit registers: DPTR (=DPH, DPL),PC
- However! "Register addressing" refers to R0..R7 ONLY!
  - SFR ones are "Direct addressing"

# Register addressing mode in 8051

- Encoded as part of the instruction byte
- A is implicitly addressed; Rxxx is explicit

| Machine code | binary | Assembly |
|---|---|---|
| E8 | 11101000 | MOV A, R0 |
| EF | 11101111 | MOV A, R7 |
| F8 | 11111000 | MOV R0, A |
| FF | 11111111 | MOV R7, A |
| | 11111xxx | MOV Rxxx, A |

# Immediate Addressing

- Immediate comes from "data value immediately follows the opcode byte"
  - Meaning: constant value in an instruction
  - Example:
    Green part: immediate;     Blue: register

| code | binary | Assembly |
|---|---|---|
| 74 25 | 0111 0100 0010 0101 | MOV   A, #25H |
| 78 25 | 0111 1000 0010 0101 | MOV   R0, #25H |
| 7F 25 | 0111 1111 0010 0101 | MOV   R7, #25H |

# Immediate may be multiple bytes

- Example: MOV *imm* to the 16-bit DPTR
  - MOV   DPTR, #2550H ;; 2-byte immed.
  - code is 3-bytes:  90 25 50 (hex)
  - DPTR is implicit
- Assembler checks constant range
  - MOV  DPTR, #68975  ;; causes asm error
  - #68975 is too large to fit in 2 bytes

# Assembler label may be an immediate value

```
                MOV     DPTR, #Label
                ...
Label:          DB      "Hello world"
```

- The #Label part represents the address of the Label after the assembler determines its value
- fits the size of DPTR

# Direct addressing

- Direct = address of operand
  - on-chip memory
  - Also mapped to GPIO & SFR
  - pointer in a separate byte, like immed.'s
- Usage: when naming anything addressable
  - e.g., PSW, SP, P0..P3, DPH, DPL, address constant (as a label or constant)

# Reg. vs. immediate vs. direct addressing

- meaning of MOV  A, 0
  take content at <u>on-chip memory address</u> 0,
  copy it into the Accumulator

| code | binary | assembly | mode |
|---|---|---|---|
| E8 | 1110 1<u>000</u> | MOV    A, R<u>0</u> | ;; reg |
| 74 00 | 0111 0100 <u>0000 0000</u> | MOV    A, #<u>0</u> | ;; imm |
| E5 00 | 1110 0101 <u>0000 0000</u> | MOV    A, <u>0</u> | ;; dir |

# Subtle difference betw. Reg & Direct mode

- MOV A, 0 ;; direct mode => 2 bytes
  - at on-chip address 0,
  - mapped to R0 of bank 0 (4 banks total)
- MOV A, R0 ;; register mode => 1 byte!
  - register R0 of current bank
  - does not have to be bank zero! Depends on PSW.3 and PSW.4