8051 Instruction Set Architecture

Outline

- Architecture Overview
  - Processor, Memory, Interfaces
- Instructions and Assembly Language
  - Operate, Move, Control
- Addressing Modes

Useful Links

- Intel MCS-51 data sheets
  - http://lms.nthu.edu.tw/sys/read_attach.php?id=414787
- EdSim51: 8051+device simulator in Java
  - http://www.edsim51.com/ (can read .asm and .hex files)
- Keil 8051 Instruction Set Manual
  - http://www.keil.com/support/man/docs/is51/

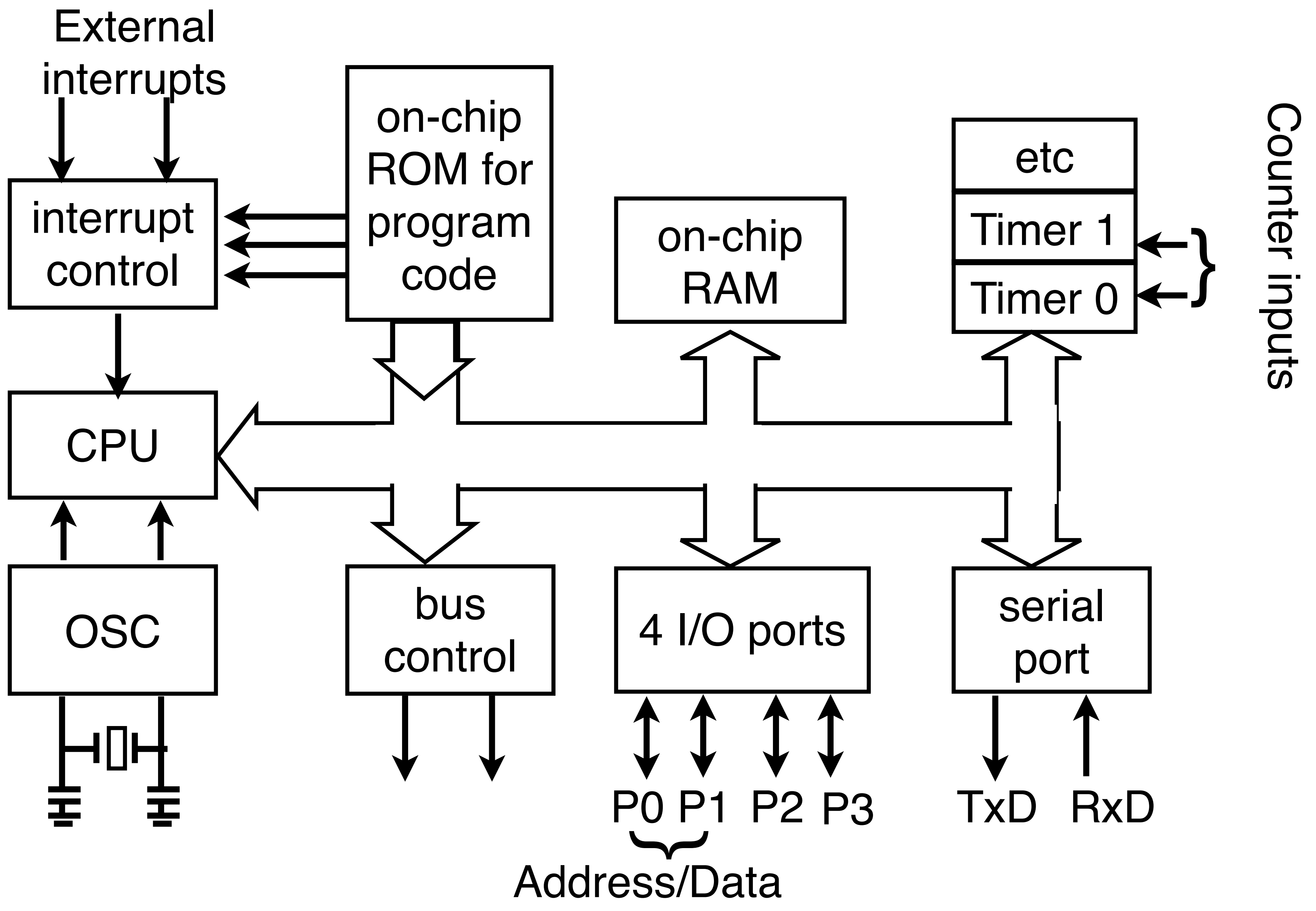
8-bit MCU instruction-set architectures (ISA)

- Older
  - 6811(formerly Motorola, now Freescale)
  - 8051(Intel), Z8(Zilog), PIC16 (Microchip)
- Newer
  - ATMega (Atmel)
  - Hitachi H8
- 8-bit registers (native data size = 8 bits)
  - address size may be 8 bit or 16 bit

Why 8051 still popular after 25 years

- Intel lets others make compatible MCUs
    - Atmel, Philips/Signetics, Siemens, Dallas Semiconductor
- New and improved
    - 100+MHz, low power, "high performance"
- Free cores available
- integrated RF (e.g., Nordic nRF24E1, TI CC2540, RadioPulse RG2400, Atmel...)

Block diagram of 8051



Memory Spaces

Space	**CODE**	**IDATA**	**XDATA**
Full name	Program Memory	Internal Data Memory	External Data Memory
Size	64 KB	256 Bytes	64 KB
Pur- pose	instruction and constant data	CPU registers, hardware stack, small variables	software stack and main memory
		Special Function Registers	

Registers in 8051

- General purpose, 8-bit
  - A: (Accumulator), B
  - R0, R1, ..., R7 (CPU registers, in 4 banks)
- 16-bit, specifically used as pointers
  - DPTR: (data pointer, <DPH:DPL>
  - PC: (program counter) not user visible
- PSW: program status word

CPU Registers

- Eight visible at a time
  - R0, R1, ... R7 => selected using 3 bits
- 4 Banks, mapped to IDATA space
  - bank 0: IDATA addresses 0x00 - 0x07
  - bank 1: IDATA addresses 0x08 - 0x0F
  - bank 2: IDATA addresses 0x10 - 0x17
  - bank 3: IDATA addresses 0x18 - 0x1F

Accumulator (A)

- An implicit register in many instructions
  - as both a source and the destination
    e.g, ADD A, #23
    meaning: A = A + 23
- Reason for using A
  - small code size, because there is just one!
  - All others require several bits for registers

Machine Instructions

- Opcode
  - Specifies the operation (~function)
- Operands
  - the "arguments" to an opcode
  - could be accumulator, register, constant value, value in memory, etc

Opcodes in 8051

- MOV, MOVX, MOVC, XCH, XCHD, PUSH, POP
- ADD, ADDC, SUBB, MUL, DIV, ANL, ORL, XRL
- RR, RL, RLC, RRC, SWAP
- INC, DEC, CLR, SETB, CPL, DA
- NOP
- AJMP, LJMP, ACALL, LCALL, RET
- JB, JNB, JC, JNC, JZ, JNZ, JMP, CJNE, DJNZ

Data Movement Instructions

- Within IDATA
  - MOV: move data (register, IDATA, A, ...)
  - PUSH, POP: stack operation (IDATA)
- With Code or XDATA
  - MOVX: move data (XDATA memory)
  - MOVC: move data (CODE memory)
- Exchange (accumulator, IDATA)
  - XCH, XCHD: exchange byte or "digit" (nibble)

MOV instruction

- syntax:
  MOV dest, src
  - Think assignment statement: dest := src;
- dest, src are called <u>Operands</u>
  - dest can be A, B, R0..R7, DPH, DPL
  - src can be A, B, R0...R7, or an <u>immediate</u>
- *Immediate* is aka a "constant", "literal" value, e.g., #12

Idiosyncrasy with immediate in Intel Assembly syntax

- Default base: decimal
  - #12 (assumed to be decimal)
  - Can be hex: #12**H**   (12 hex, = 18 dec.)
- However! the char after # must be 0..9
  - #FF**H**   is invalid! (since F is not in 0..9)
  - Solution: #**0**FF**H**   (add a useless 0 (zero) in front. It does not make it octal)

Immediate vs. direct (Addressing mode)

- MOV   A, #17H   ;; #17H is a *literal value*
  meaning: A = 0x17;
- MOV   A, 17   ;; 17 is IDATA address!
  meaning: A = *((char*)17);
- Big difference!
  - R0, ... R7   => register mode
  - #17 => immediate mode;
  - 17 => direct mode (IDATA address)

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 on-chip memory address 0,
  copy it into the Accumulator

code	binary	assembly	mode
E8	1110 1000	MOV A, R0	;; reg
74 00	0111 0100 0000 0000	MOV A, #0	;; imm
E5 00	1110 0101 0000 0000	MOV A, 0	;; 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

Allowed Combinations of byte-Addressing Modes

Opcode	Dest	Source
MOV	A,	reg or @reg
		#imm
		dir
	reg, @reg,	A
		#imm
		dir
	dir,	A
		#imm
		dir
		reg or @reg

Restricted combinations of Addressing Modes

- Disallowed: Register-to-register MOV
  - e.g., MOV ~~R1, R2~~
  - solution: go through A or use immediate
- Accumulator-to-accumulator ~~MOV A, A~~ (useless)
- anything-to-immediate MOV (nonsense)
  - e.g., MOV ~~#20, R3~~

ADD instruction

- ADD A, source
  - A := A + source; sets CarryBit as side effect
  - The first operand must be A ("accumulator")
- ADDC A, source
  - A := A + source + CarryBit
- source can be any R-value operand
  - immediate, direct, register

Allowed Combinations for ADD & ADDC

Opcode	Dest	Source
ADD	A,	reg
		@reg
		#imm
		dir
ADDC	A,	reg
		@reg
		#imm
		dir

how to write R2 := 0x25 + 0x34;

- Key: need to involve the accumulator A

-
  ```
  MOV    A, #25H       ;;  A := 0x25;
  ADD    A, #34H       ;;  A := A + 0x34;
  MOV    R2, A         ;;  R2 := A;
  ```
- Would be incorrect (in 8051) to try
  ```
  MOV    R2, #25H      ;; this part is ok
  ADD    R2, #34H      ;; must use A, not R2
  ```
  (~~R2,~~ in the ADD line)
- Since the sum is constant, could optimize as
  ```
  MOV    R2, #59H      ;; eval. by assembler!
  ```

Assembly Language

- Directives
  - Commands to the assembler!
    e.g., starting address, allocate memory, ...
- Instructions
  - Correspond to machine instructions
- Labels
  - Symbolic names that mark addresses
- Comments -- started with ;; till end of line

Example program (in Intel Assembly syntax)

```
        ORG     0H          ; start address
        MOV     R5, #25H    ; put const into R5
        MOV     R7, #34H    ; put const into R7
        MOV     A, #0       ; clear accumulator
        ADD     A, R5       ; A := A + R5
        ADD     A, R7       ; A := A + R7
        ADD     A, #12H     ; A := A + 0x12
HERE:   SJMP    HERE        ; loop forever here
        END     ; some may be .END
```

Try this out (Intel assembly syntax)

- Option 1: EdSim51's built-in assembler
- Option 2: Asem51 http://plit.de/asem-51/
  - (Windows only) Save assembly program as plain text file named `file.a51`
  - `asem file.a51` ;; type as a command
  - `.hex` file: hex-formatted binary file
    `.lst` file: "listing" (before/after views)

.lst file by assembler

Addr	Machine code	Assembly Source		
0000	7D 25		MOV	R5,#25H
0002	7F 34		MOV	R7,#34H
0004	74 00		MOV	A, #0
0006	2D		ADD	A, R5
0007	2F		ADD	A, R7
0008	24 12		ADD	A, #12H
000A	80 FE	HERE:	SJMP	HERE

Option 3: sdcc assembler (sdas)

- Based on ASxxxx cross assembler
- Slightly different syntax
  - instead of #23H, write #0x23 (C-like)
  - need to specify segments, e.g., .area HOME (ABS, CODE)
  - no need for .END directive
- `sdas8051 -l -o -x file.asm`

Assembly syntax: Intel vs. ASxxxx

```
      ORG   0H
      MOV   R5, #25H
      MOV   R7, #34H
      MOV   A, #0
      ADD   A, R5
      ADD   A, R7
      ADD   A, #12H
HERE: SJMP  HERE
      END
```

```
      .area HOME (ABS, CODE)
      .org  0H
      mov   R5, #0x25
      mov   R7, #0x34
      mov   A, #0
      add   A, R5
      add   A, R7
      add   A, #0x12
HERE: sjmp  HERE
```

More Intel assembler features

- Literal: in other bases
  - Binary: add B after: 00001010B
- DB: directive for "define byte" (or string, whatever data)
  - store the data in program memory
  - label is optional, not required!
- EQU: directive for constant declaration
  - inlined when used. requires a macro name!

Example of directives: DB vs. EQU

- DATA1: DB "Hello world"
  DATA2: DB 25
  ;; both occupy space in code memory,
  ;; because DATA1, DATA2 are labels
- COUNT EQU 25 ;; occupies no space
  - MOV R3, #COUNT
    ;; macro expansion into MOV R3,#25

Stack

- Fundamental data structure in programming
  - Return address of a call
  - Space for local (auto) variables
- 8051 hardware supports stack in IDATA
  - PUSH (add element to stack)
  - POP (remove element from stack)
- Need: stack pointer (SP), an SFR in 8051

8051 architecture-supported stack

- 8-bit Stack Pointer SP
  - Initialized to 07H on power up
    => points to just before R0 of bank 1
- Grows *upward* from lower to higher address (opposite that of most CPUs!)
  - PUSH: pre-increment: stack[++SP]=d;
  - POP: post-decrement: d=stack[SP--];

Syntax of Push/Pop instructions

- PUSH 6
  - This doesn't mean push literal value #6
  - It means push content at IDATA address 6 to stack! (= R6 if bank 0 is selected)
- POP 2
  - Means pop the top element of stack to IDATA address 2

Range of value for SP

- [07 -1FH] (register banks 1,2,3)
  - On power-up, SP = 07H
  - Should not go lower; could go up to 1FH
- Avoid [20-2FH] => bit-addressable area
- OK to use [30 - 7FH] (scratchpad memory)
- On 8052: ok to use the range [80H - FFH]

Other use of stack: Call/Return instructions

- Call instruction (ACALL, LCALL)
  - pushes return address onto stack then go to target
- Return instruction (RET)
  - pops return address then go to target
- Interrupt: hardware pushes return address
- Return-from-interrupt instruction (RETI)

Control Instructions (jumps, call/return)

- Unconditional jump
  - AJMP, SJMP
- Conditional jump (branch)
  - JC, JNC, JB, JNB, JZ, JNZ, JBC
  - CJNE, DJNE
- Subroutine Call and Return
  - ACALL, LCALL, RET

Looping example

```
        MOV    A, #0
        MOV    R2, #10       ;; init
AGAIN:  ADD    A, #3         ;; loop body
        DJNZ   R2, AGAIN     ;; while test
        MOV    R5, A         ;; after loop
```

- init: A=0; R2=10;
- loop: do { A += 3; } while (--R2!=0);
  DJNZ R2 means "**d**ecrement R2, **j**ump if **n**ot **z**ero"
- after: R5 = A;

Allowed Combinations of DJNZ

Opcode	Operand 1	target
DJNZ	dir,	label (PC-relative)
	reg,	

* dir is direct byte address in IDATA

** reg is register R0..R7

Nested Loops

```
        MOV   A, #55H      ;; outer loop
        MOV   R3, #10      ;; init
NEXT:   MOV   R2, #70      ;; inner loop init
AGAIN:  CPL   A            ;; complement A
        DJNZ  R2, AGAIN    ;; inner loop test
        DJNZ  R3, NEXT     ;; outer loop test
```

- init: A = 0x55; R3 = 10;

-
  ```
  do {  R2 = 70;
        do { A = ~A; } while (--R2);
  } while (--R3);
  ```

8051 Conditional Jump

Opcode	Meaning
JZ	Jump if A == 0
JNZ	Jump if A != 0
DJNZ	Decrement byte, jump if byte != 0
CJNE	Compare byte w/#data, jump if !=
JC	Jump if Carry bit set (C== 1)
JNC	Jump if not Carry bit (C==0)
JB	Jump if bit at bit-address == 1
JNB	Jump if bit at bit-address == 0
JBC	Jump if bit == 1 and clear bit

Conditional Jump Example

```
        MOV     A, R5
        JNZ     NEXT      ;; not zero=> skip
        MOV     R5, #55
NEXT:
```

-
  ```
  if (R5 == 0) {
      R5 = 0x55;
  }
  ```
- JNZ relies on A; that's why A=R5 on line 1.

Terminology: Jump vs. Branch

- Conventionally,
  - Jump = unconditional
  - Branch = conditional
- Intel uses "conditional jumps" for branches (CJNE, JNZ, ...)
- Almost universally,
  - branches use offsets (relative to next PC)

Short-Jump vs. Long-Jump Instructions

- All conditional jumps are *short jumps*
  - Target address within -128 to +127 of PC
- LJMP (long jump): 3-byte instruction
  - 2-byte target address: 0000 to FFFFH
  - Original 8051 has only 4KB on-chip ROM
- SJMP (short jump): 2-byte instruction
  - 1-byte offset (relative): -128 to +127

Calculating Short Jump Addresses

- Relative to the next instruction's address
- Why relative?
  - Location independence
  - Small offset
- NEXT is 3 bytes from next instruction

binary	assembly		
60 03		JZ	NEXT
8		INC	R0
4		INC	A
4		INC	A
24 77	NEXT:	ADD	#77H

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!!!

Call Instructions

- LCALL (long call): 3-byte instruction
  - 2-byte address within 64K-byte range
- ACALL (absolute call): 2-byte instruction
  - 11-bit target address within 2K-byte range
- Both will push return address on stack!!
  Both LCALL, ACALL specify *absolute addr.*
- Use RET instruction to return

Example Use of Subroutine: 16-bit add

- Adding two 16-bit numbers
  - R1:R0 = R1:R0 + R3:R2; // think X=X+Y
- Structure it as a subroutine.
  e.g., to do R1:R0 = 1234H + 5678H, do

  -
    ```
    MOV    R1, #12H  ;; X[upper] = 12H
    MOV    R0, #34H  ;; X[lower] = 34H
    MOV    R3, #56H  ;; Y[upper] = 56H
    MOV    R2, #78H  ;; Y[lower] = 78H
    LCALL  ADD16     ;; make the call
    ;; expect the sum 68ACH in R1:R0
    ```

Code for 16-bit add

```
ADD16:   MOV    A, R0    ;; put X[lower] in A
         ADD    A, R2    ;; X[lower]+Y[lower]
         MOV    R0, A    ;; R0 := sum[lower]
         MOV    A, R1    ;; put X[upper] in A
         ADDC   A, R3    ;; X[upper]+Y[upper]
         MOV    R1, A    ;; R1 := sum[upper]
         RET             ;; return to caller
```

Subroutine call with string parameter

- e.g., function call DISPLAY("MESSAGE")
- Similar to a printf("message")
  - can pass pointer of string to DPTR
  - then do LCALL to DISPLAY routine

```
        MOV     DPTR, #str      ;; pass parameter
        LCALL   DISPLAY         ;; make the call
        ...
str:    DB      "message"       ;; string data
        DB      0               ;; null termination
```

Bit Access

- Bit operands
  - Carry flag (C, aka CY)
  - Bit-addressable IDATA regions
- Bit instructions
  - SETB, CLR, MOV, CPL, ANL, ORL
  - JC, JNC, JB, JNB,

Bit Addressing Mode

- Ability to specify a bit
  - Normally, an address refers to a byte
- Bit-addressable areas
  - IDATA bytes 20H--2FH
  - SFR bytes at 80H - FFH that are divisible by 8

Bit addressable data (IDATA) region

Byte address	Bit address
20H	00H .. 07H
21H	08H .. 0FH
22H	10H .. 17H
23H	18H .. 1FH
24H	20H .. 27H
25H	28H .. 2FH
26H	30H .. 37H
27H	38H .. 3FH

Byte address	Bit address
28H	40H .. 47H
29H	48H .. 4FH
2AH	50H .. 57H
2BH	58H .. 5FH
2CH	60H .. 67H
2DH	68H .. 6FH
2EH	70H .. 77H
2FH	78H .. 7FH

Bit address calculation

```
SETB        42H        ;; set bit at bit-address 42H
```

- Which byte contains bit address 42H?
  - Top 5 bits: offset from 20H
  - Bottom 3 bits: bit position within the byte
- Answer:
  - Byte addr = 20H + 8 = 28H
  - Bit 2 within byte 28H

4	2
0 1 0 0	0 0 1 0
0 1 0 0 0	0 1 0
8	2

More Bit-address Calculation

```
CLR        67H        ;; clr bit at bit-address 67H
```

- What is bit address 67H?
  - Top 5 bits: offset from 20H
  - Bottom 3 bits: bit position within byte
- Answer:
  - Byte addr = 20H + CH = 2CH
  - Bit 7 within byte 2CH

6	7
0 1 1 0	0 1 1 1
0 1 1 0 0	1 1 1
C	7

Bit addressable SFR region

Byte	SFR	Bit address	Byte	SFR	Bit address
80H	P0	80H .. 87H	C0H		C0H .. C7H
88H	TCON	88H .. 8FH	C8H		C8H .. CFH
90H	P1	90H .. 97H	D0H	PSW	D0H .. D7H
98H	SCON	98H .. 9FH	D8H		D8H .. DFH
A0H	P2	A0H .. A7H	E0H	ACC	E0H .. E7H
A8H	IE	A8H .. AFH	E8H		E8H .. EFH
B0H	P3	B0H .. B7H	F0H	B	F0H .. F7H
B8H	IP	B8H .. BFH	F8H		F8H .. FFH

Bit address of SFRs

- SFRs have address 80H - FFH
- Bit addressable SFRs are
  - P0, P1, P2, P3, TCON, SCON, IE, IP, PSW, ACC, B
  - They all have address divisible by 8!!! 80H, 88H, 90H, 98H, A0H, A8H, B0H, ...
- Bit address = byte address+bit position

Example Bit-address calculation for SFR

- What is bit address of P1.2?
  - P1 has byte address 90H
  - therefore, P1.2 has bit address 92H
- What is bit address of IE.5?
  - IE has byte address A8H
  - therefore, IE.5 has bit address A8H+5 = ADH

PSW: program status word

- Contains flags indicating status of CPU

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

Symbolic names for bit registers

- May be built-in definition of assembler
  - e.g., OV is PSW.2, CY is PSW.7
    most assemblers know these names
- If not, definable using the BIT directive
  - Like a macro definition for bits!

```
OV          BIT  PSW.2  ;; #def OV PSW.2
OVEN_HOT    BIT  P2.3
BUZZER      BIT  P1.5
```

BIT vs EQU directives

- EQU is general for constant declaration
- BIT is specific for bits

```
BUZZER      BIT    P1.5
```

can also be written as

```
BUZZER      EQU    P1.5
```

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

Difference between C and CY

- Both refer to PSW.7 physically
- C: when *implicitly* addressed as part of instruction
- CY: when *explicitly* addressed as a bit-address

Assembly		byte1	byte2	Addr. mode
SETB	CY	D2	D7	explicit PSW.7
SETB	C	D3		implicit C
CLR	CY	C2	D7	explicit PSW.7
CLR	C	C3		implicit C

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

Implicit vs Explicit bit

- MOV _, C and MOV C, _ are the only two bit-MOV instructions
    - C is implicit (hardwired) in these instructions
    - _ is an explicit bit address
- MOV bit1, bit2 => does not exist
    - No 8051 instruction takes two bit addresses explicitly; all go through C (similar to A for 8-bit instructions)

Use C as a "bit accumulator"

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

```
        SETB   P1.2
AGAIN:  MOV    C, P1.2
        JNC    AGAIN
```

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

equivalent!

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*

Bit-jump instructions

assembly	encoding (hex)			addressing mode
JC *target*	40	*offset*		C is implicit
JNC *target*	50	*offset*		
JB *bit*, *target*	20	*bit*	*offset*	***bit*** (e.g., P0.3) is explicit
JNB *bit*, *target*	30	*bit*	*offset*	

* *target* label is represented as PC-relative *offset*

** *bit* is bit address

Bit-moving instructions

assembly	encoding (hex)		addressing mode
CPL bit	B2	*bit*	*bit* (e.g., P0.3) is explicit
CLR *bit*	C2	*bit*	
SETB *bit*	D2	*bit*	
CPL C	B3		C is implicit
CLR C	C3		
SETB C	D3		
MOV C, P0.3	A2	83	C is implicit, P0.3 is explicit
MOV P0.3, C	92	83	

Implicit vs Explicit references to Accumulator

Unit	Implicit	Explicit	Address
byte	A	ACC	E0H (byte)
bit		ACC.0	E0H (bit)
		ACC.1	E1H
		...	...
		ACC.7	E7H

Use explicit addressing ACC when implicit addressing is not available
(e.g., PUSH ACC, POP ACC)

Example: bit addressing in Accumulator

- Check if a number is even or odd
  - Can look at bit ACC.0 (bit address E0H) bit 0 of the accumulator: 0=even, 1=odd
  - Use JNB ACC.0, *Label* or JB ACC.0, *Label*
- Check if a number is negative
  - Can look at bit ACC.7 (bit address E7H) sign bit: 1 means negative, 0 nonnegative.

Bit vs Byte address

- Same number can mean very different things!!

MOV 05, A	;; 05 is direct IDATA (byte) address
MOV A, #05	;; #05 is immediate (constant) value
MOV 05, C	;; 05 is (explicit) bit addressing ;; = bit 5 of IDATA byte 20H

OV-flag (PSW.2)

- Overflow: too big/too small to represent
- ADD:Both operands same sign, sum different

  $\left.\begin{array}{l}\text{positive + positive => get negative}\\ \text{negative – negative => get positive}\end{array}\right\}$ overflow!

  - Cannot overflow when adding a positive and a negative, because the sum is in between
- SUBB:
  - opposite condition of ADD

A	B	A–B
> 0	< 0	< 0
< 0	> 0	> 0

overflow!

Arithmetic, Logical, and Bit Operations

- Arithmetic:
  - ADD, ADDC, SUBB, MUL, DIV, INC, DEC
- Logical bit
  - ANL, ORL, XRL, CPL
- Rotate
  - RR, RL, RRC, RLC
- Swap

SUBB: Subtract with Borrow

- SUB (subtract without borrow): doesn't exist on 8051!
- SUBB: subtract with borrow (like ADDC)
- You can simulate SUB by
  - CLR C
  - SUBB A, arg

MUL: Multiplication

- AB := A * B
  - 8 bits each: A (accumulator), B (SFR)
  - 16-bit Product:
    A = lower order, B = higher order
- Assumption: unsigned numbers!

  -
    ```
    MOV  A, #25H
    MOV  B, #65H
    MUL   AB
    ```

0x25 * 0x65 = 0x0e99
=> A = 0x99, B = 0x0e

DIV: 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
    ```

note: EdSim may have
the quotient & remainder backwards

CPL: Complement

- CPL A
  - Flip all bits in Accumulator
- CPL C
  - Complement C
- CPL bit
  - Complement bit

INC, DEC

- Increment (+1) or Decrement (-1) a byte
  - INC A DEC A
  - INC Reg DEC Reg
  - etc
- INC also works with DPTR (16-bit), but not DEC!

2's complement

- To negate a number:
  invert all bits + 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

Logical vs. Arithmetic Instructions

- Logical ones have more combinations!
  - Arithmetic ones require A to be LHS
  - Logical ones allow *direct* to be LHS!

Arithmetic	Logical
ADD A, Reg	ANL A, Reg
ADD A, #imm	ANL A, #imm
ADD A, dir	ANL A, dir
~~ADD dir, A~~	ANL dir, A
~~ADD dir, #imm~~	ANL dir, #imm

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
  - test C and increment count

```
        MOV     R0, #0      ;; R0 is count
        MOV     A, #23H     ;; bit pattern 23H
        MOV     R1, #8      ;; loop 8 times using R1
Loop:   RLC     A           ;; rotate thru CY
        JNC     Skip        ;; skip if CY=0
        INC     R0          ;; CY=1 => incr count
Skip:   DJNZ    R1, Loop    ;; while R1 > 0
```

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 bank selection

- 4 possible banks depending on PSW<4:3>
- Example
  - MOV R0, #12H
    Where is R0?
- Answer:
  depends on which bank!
  - Could be address 00H, 08H, 10H, or 18H

PSW<4:3>	Bank Selected
00	bank 0
01	bank 1
10	bank 2
11	bank 3

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 registers

- Register mode:
  - Use R0, R1, ... name
  - Limited to the current bank
- Direct mode:
  - Use the IDATA address of the register!
  - 00 for *R0 of bank 0*
    12 for *R2 of bank 3*... etc

Why register mode vs. direct mode?

- Register mode
  - Smaller code size. e.g.,
    INC R1 (1 byte) vs. INC 01 (2 bytes)
- Direct mode
  - Instruction with limited combinations
    e.g., PUSH R2 is not allowed
    => must say PUSH 02

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 through DPTR
  - 0000-FFFFH: data memory (MOVX)
  - 0000-FFFFH: code memory (MOVC)

Addressing Modes in 8051

- Register
- Immediate
- Direct
- Register-indirect
- Indexed
- Implicit
- Bit

Special function registers (SFR)

- Located at address ≥ 80H, up to FFH
  - Also for Timer, Interrupt control, Serial port, Power control (not shown in table)
  - Not all addresses are used

Register	Address
P0, P1, P2, P3	80H, 90H, A0H, B0H
PSW, A, B	D0H, E0H, F0H
SP, DPL, DPH	81H, 82H, 83H

Implicit vs. explicit direct addressing

- Accumulator has explicit address (E0H)
- Two ways: same functionality, but different encoding!

code	assembly	mode
E8	MOV A, R0	implicit A
88 E0	MOV 0E0H, R0	explicit A, direct mode

Explicit A can be written as ACC

More example of implicit vs. explicit

- MOV DPTR, #2550H ;; 3 bytes
  - this is implicit addressing for DPTR (DPTR does not have a direct address!)
- MOV DPL, #50H ;; 3 bytes
  MOV DPH, #25H ;; 3 bytes
  - DPL, DPH are explicitly direct addressing => each requires 1 byte
  - #50H, #25H each requires 1 byte

implicit vs. direct mode example cont'd

code	assembly	#bytes
90 25 50	MOV DPTR, #2550H	3 bytes
75 *82* 50 75 *83* 25	MOV DPL, #50H MOV DPH, #25H	6 bytes!

Limited combinations of addressing modes

- The opcode dictates operand's addr. modes
  - Assembly mnemonic may look the same, but they may be different opcodes

#bytes	Assembly format			machine code bytes		
1	MOV	A	reg	5-bit opcode (E8), 3-bit reg		
2			imm	8-bit opcode (74)	imm	
2			dir	8-bit opcode (95)	dir	
1		reg	A	5-bit opcode (F8), 3-bit reg		
2			imm	5-bit opcode (78), 3-bit reg	imm	
2			dir	5-bit opcode (A8), 3-bit reg	dir	
2		dir	A	8-bit opcode (F5)	dir	
3			imm	8-bit opcode (75)	dir	imm
3			dir	8-bit opcode (85)	dir1	dir2
2			reg	5-bit opcode (88), 3-bit reg	dir	

PUSH/POP: direct addressing

- Syntax:
  - PUSH *dir* ;; push val at *dir* to stack
    POP *dir* ;; pop value to mem at *dir*
  - Explicit: *dir* *;;* on-chip mem location
  - Implicit: SP (stack pointer)
- Example
  - PUSH 05 ;; on-chip addr = 05
    Cannot say PUSH R5 ;; register mode

Restrictions and Workarounds

Cannot say	Instead, say	assume
PUSH R0	PUSH 0	Bank 0
POP A	POP 0E0H	
PUSH B	PUSH 0F0H	

- PUSH, POP are available only in direct mode
- ACC => 0E0H, B => 0F0H

Pointers in 8051

- Indirect addressing:
  - Address is in a register; variable
  - IDATA: use MOV with @R0, @R1
  - XDATA: use MOVX with DPTR
  - CODE: use MOVC with DPTR
- Compare to direct addressing
  - fixed address inside instruction! IDATA only

Direct addressing vs. indirect addressing

- Direct addressing (e.g, MOV A, 02)
  - 02 is the hardwired address of the operand (i.e., constant pointer, single-byte)
  - here, address = 02, in IDATA
- Register-Indirect (e.g., MOV A, @R0)
  - R0 or R1 contains the pointer to the operand in IDATA
  - R0 or R1 can be computed!

Register addressing vs Register-indirect addr.

- Register addressing (e.g., MOV A, R1)
  - R1 contains the operand value
- Register-indirect addressing (MOV A, @R1)
  - R1 contains the (1-byte) *pointer* to operand value. e.g., if R1 contains 20H, then operand is at on-chip memory address 20H
- R2, ... R7 cannot be used with @
- OK to use SP (stack pointer) for indirect!

Example: Register vs Indirect mode

MOV A, R0

before	A	xx
	R0	3
after	A	3
	R0	3

copy

MOV A, @R0

before	A	xx
	R0	3
	R1	2
	R2	5
	R3	1
after	A	1
	R0	3
	R1	2
	R2	5
	R3	1

copy

address of R3 = 3(assume bank 0)

copying constant to array: (1) direct

```
MOV   A, #55H   ;; set up A
MOV   40H, A    ;; copy from A
MOV   41H, A    ;; 2-byte instr.
MOV   42H, A    ;; could also MOV 42H,#55H
MOV   43H, A    ;; but instr. would be 3 bytes
MOV   44H, A    ;; which would be wasteful
```

copying constant to array: (2) indirect

```
MOV   A, #55H   ;; set up A
MOV   R0,#40H   ;; set up pointer
MOV   @R0,A     ;; copy A to mem at addr=40
INC   R0        ;; R0++
MOV   @R0, A    ;; copy to addr=41
INC   R0        ;; next R0
...   ;; ... continued for addr=42, 43, 44
```

copying constant to array: (3) indirect+loop

```
        MOV   A, #55H      ;; A = 0x55,
        MOV   R0,#40H      ;; array = 0x40,
        MOV   R2, #5       ;; R2 = 5;
Again:  MOV   @R0,A        ;; do { *array=0x55;
        INC   R0           ;;      array++;
        DJNZ  R2, Again    ;; } while (--R2);
```

Example: clear 16-byte array at address 60H

```
       CLR   A
       MOV   R1, #60H  ;; ptr = &Array[0];
       MOV   R7, #16   ;; count=16;
Again: MOV   @R1, A    ;; do { (*ptr) = 0;
       INC   R1        ;;        ptr++;
       DJNZ  R7, Again ;; } while (--count);
```

Example: loop 10 bytes *p++ = *q++;

```
        MOV   R0, #35H   ;; source ptr, called q
        MOV   R1, #60H   ;; dest ptr, called p
        MOV   R3, #10    ;; count = 10;
Back:   MOV   A, @R0     ;; do { A = *q;
        MOV   @R1, A     ;;        *p = A;
        INC   R0         ;;        q++;
        INC   R1         ;;        p++;
        DJNZ  R3, Back   ;; } while (--count);
```

Indexed Addressing Mode

- Index: array access
  - Base address is in DPTR
  - Index (offset) is in A
- Instruction in the form ;; implicit register
  - MOVC A, @A+DPTR
  - C of MOVC means "code memory"
- Meaning: A = DPTR[A];

e.g., Lookup Table: x^2

- **const int** Table[ ]={0, 1, 4, 9, 16, 25, 36, 49, 64, 81};
  for(;;) { P2 = Table[P1]; }
- Assembly: Statically initialized array:
  ```
  Table:  DB  0, 1, 4, 9, 16, 25, 36, 49, 64, 81
  ```
- Use DPTR for base pointer to Table:
  ```
          MOV   DPTR, #Table
  ```
- Lookup is
  ```
          MOV    A, P1            ;; P1 as index
          MOVC  A, @A+DPTR  ;; base + index
  ```

Program: write P2 = P1^2 continuously

```
        ORG     0
        MOV     DPTR, #Table    ;;base addr of array
        MOV     P1, #0FFH       ;;config P1 for input
BACK:   MOV     A, P1           ;;read P1
        MOV     A, @A+DPTR      ;; lookup square
        MOV     P2, A           ;; write square to P2
        SJMP    BACK            ;; loop forever
Table:  DB      0,1,4,9,16,25,36,49,64,81
        END
```

MOVC vs. MOVX

- Both use 16-bit pointer DPTR
- MOVC:
  C = "code" mem (read-only data)
- MOVX:
  X = "external" data mem (read/write)
- Harvard architecture:
  - separate code/data. e.g., 8051

8051 vs 8052: Upper IDATA

- 8051 IDATA space:
  - 00 - 7FH: data, direct or indirect addressing
  - 80 - FFH: SFR, by direct addressing only
- 8052 IDATA space:
  - 00 - 7FH: data, direct or indirect (=8051)
  - 80 - FFH: SFR, if by direct addressing
  - 80 - FFH: data, by indirect addressing (8052 only!!)

Common use of Upper IDATA

- For stack
  - Always indirectly accessed!
  - Return address, local variables, parameters, ...
- Separate from registers, bit memory, globals