EECS 113
Lec. 12: Timing part 1

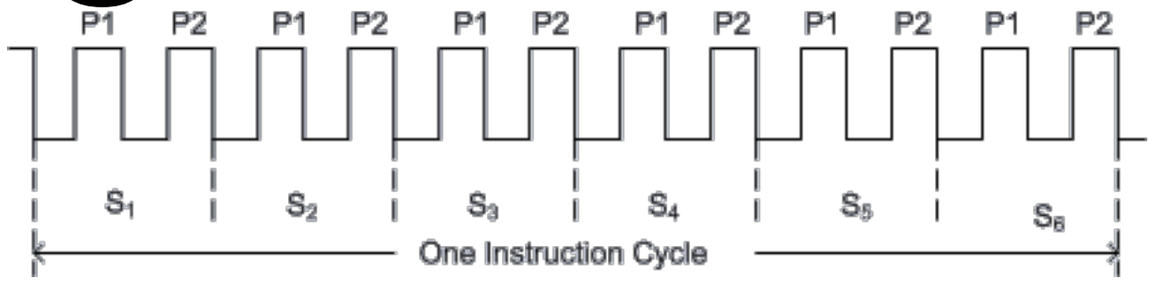
Dept. of EECS, UC Irvine

Timing Control

- Instruction timing
  - Relative delay
- Timer
  - 8051 Built-in timer
  - External timer hardware

Instruction Timing

Timing



- "Clock" (oscillator)
  - basic logical cycle of synchronous logic
  - based on crystal oscillator
- Intel version of 8051
  - one machine cycle = 12 oscillator cycles
  - machine cycle = unit of instruction timing
  - So 12 MHz means (12 Million oscillator cycles)/(12 oscillator cycles = 1 M machine cycles/sec), or 1us machine cycle time.

Instructions take time

- # cycles depend on
  - Instruction
    - Addressing mode
- Original 8051: 1 instr. cycle = 12 osc. cycles
  - e.g., 12 MHz osc. freq = 1 MHz instr. freq
    => 1µs instr. cycle

Where to find timing of instructions?

- Data sheet for 8051, page 2-21 on
- http://eee.uci.edu/10s/18065/27238302.pdf

MCS®-51 INSTRUCTION SET

Table 10. 8051 Instruction Set Summary

Interrupt Response Time: Refer to Hardware Description Chapter.

Instructions that Affect Flag Settings(1)

Instruction	Flag C	Flag OV	Flag AC	Instruction	Flag C	Flag OV	Flag AC
ADD	X	X	X	CLR C	0		
ADDC	X	X	X	CPL C	X		
SUBB	X	X	X	ANL C,bit	X		
MUL	0	X		ANL C,/bit	X		
DIV	0	X		ORL C,bit	X		
DA	X			ORL C,/bit	X		
RRC	X			MOV C,bit	X		
RLC	X			CJNE	X		
SETB C	1						

(1)Note that operations on SFR byte address 208 or bit addresses 209-215 (i.e., the PSW or bits in the PSW) will also affect flag settings.

Note on instruction set and addressing modes:

- **Rn** — Register R7–R0 of the currently selected Register Bank.
- **direct** — 8-bit internal data location's address. This could be an Internal Data RAM location (0–127) or a SFR [i.e., I/O port, control register, status register, etc. (128–255)].
- **@Ri** — 8-bit internal data RAM location (0–255) addressed indirectly through register R1 or R0.
- **#data** — 8-bit constant included in instruction.
- **#data 16** — 16-bit constant included in instruction.
- **addr 16** — 16-bit destination address. Used by LCALL & LJMP. A branch can be anywhere within the 64K-byte Program Memory address space.
- **addr 11** — 11-bit destination address. Used by ACALL & AJMP. The branch will be within the same 2K-byte page of program memory as the first byte of the following instruction.
- **rel** — Signed (two's complement) 8-bit offset byte. Used by SJMP and all conditional jumps. Range is −128 to +127 bytes relative to first byte of the following instruction.
- **bit** — Direct Addressed bit in Internal Data RAM or Special Function Register.

Mnemonic		Description	Byte	Oscillator Period
ARITHMETIC OPERATIONS				
ADD	A,Rn	Add register to Accumulator	1	12
ADD	A,direct	Add direct byte to Accumulator	2	12
ADD	A,@Ri	Add indirect RAM to Accumulator	1	12
ADD	A,#data	Add immediate data to Accumulator	2	12
ADDC	A,Rn	Add register to Accumulator with Carry	1	12
ADDC	A,direct	Add direct byte to Accumulator with Carry	2	12
ADDC	A,@Ri	Add indirect RAM to Accumulator with Carry	1	12
ADDC	A,#data	Add immediate data to Acc with Carry	2	12
SUBB	A,Rn	Subtract Register from Acc with borrow	1	12
SUBB	A,direct	Subtract direct byte from Acc with borrow	2	12
SUBB	A,@Ri	Subtract indirect RAM from ACC with borrow	1	12
SUBB	A,#data	Subtract immediate data from Acc with borrow	2	12
INC	A	Increment Accumulator	1	12
INC	Rn	Increment register	1	12
INC	direct	Increment direct byte	2	12
INC	@Ri	Increment direct RAM	1	12
DEC	A	Decrement Accumulator	1	12
DEC	Rn	Decrement Register	1	12
DEC	direct	Decrement direct byte	2	12
DEC	@Ri	Decrement indirect RAM	1	12

#bytes - #cycles

	x0	x1	x2	x3	x4	x5	x6	x7
Ox	NOP 1-1	AJMP addr 2-2	LJMP code 3-2	RR A 1-1	INC A 1-1	INC dir 2-1	INC @R0 1-1	INC @R1 1-1
1x	JBC bit,rel 3-2	ACALL addr 2-2	LCALL code 3-2	RRC A 1-1.C	DEC A 1-1	DEC dir 2-1	DEC @R0 1-1	DEC @R1 1-1
2x	JB bit,rel 3-2	AJMP addr11 2-2	RET 1-2	RL A 1-1	ADD A,#imm 2-1,C,OV,AC	ADD A,dir 2-1,C,OV,AC	ADD A,@R0 1-1,C,OV,AC	ADD A,@R1 1-1,C,OV,AC
3x	JNB bit,rel 3-2	ACALL addr11 2-2	RETI 1-2	RLC A 1-1,C	ADDC A,#imm 2-1,C,OV,AC	ADDC A,dir 2-1,C,OV,AC	ADDC A,@R0 1-1,C,OV,AC	ADDC A,@R1 1-1,C,OV,AC
4x	JC rel 2-2	AJMP addr11 2-2	ORL dir,A 2-1	ORL dir,#imm 3-2	ORL A,#imm 2-1	ORL A,dir 2-1	ORL, A,@R0 1-1	ORL A,@R1 1-1
5x	JNC rel 2-2	ACALL addr11 2-2	ANL dir,A 2-1	ANL dir,#imm 3-2	ANL A,#imm 2-1	ANL A,dir 2-1	ANL A,@R0 1-1	ANL A,@R1 1-1
6x	JZ rel 2-2	AJMP addr11 2-2	XRL dir,A 2-1	XRL dir,#imm 3-2	XRL A,#imm 2-1	XRL A,dir 2-1	XRL A,@R0 1-1	XRL A,@R1 1-1
7x	JNZ rel 2-2	ACALL addr11 2-2	ORL C,bit 2-2,C	JMP @A+DPTR 1-2	MOV A,#imm 2-1	MOV dir,#imm 3-2	MOV @R0,#imm 2-1	MOV @R1,#imm 2-1
8x	SJMP rel 2-2	AJMP addr11 2-2	ANL C,bit 2-2,C	MOVC A,@A+PC 1-2	DIV AB 1-4,C=0,OV	MOV dir,dir 3-2	MOV dir,@R0 2-2	MOV dir,@R1 2-2
9x	MOV DPTR,#imm16 3-2	ACALL addr11 2-2	MOV bit,C 2-2	MOVC A,@A+DPTR 1-2	SUBB A,#imm 2-1,C,OV,AC	SUBB A,dir 2-1,C,OV,AC	SUBB A,@R0 1-1,C,OV,AC	SUBB A,@R1 1-1,C,OV,AC
Ax	ORL C,/bit 2-2,C	AJMP addr11 2-2	MOV C,bit 2-1,C	INC DPTR 1-2	MUL AB 1-4,C=0,OV		MOV @R0,dir 2-2	MOV @R1,dir 2-2
Bx	ANL C,/bit 2-2,C	ACALL addr11 2-2	CPL bit 2-1	CPL C 1-1,C	CJNE A,#imm,rel 3-2,C	CJNE A,dir,rel 3-2,C	CJNE @R0,#imm,rel 3-2,C	CJNE @R1,#imm,rel 3-2,C
Cx	PUSH dir 2-2	AJMP addr11 2-2	CLR bit 2-1	CLR C 1-1,C=0	SWAP A 1-1	XCH A,dir 2-1	XCH A,@R0 1-1	XCH A,@R1 1-1
Dx	POP dir 2-2	ACALL addr11 2-2	SETB bit 2-1	SETB C 1-1,c=1	DA A 1-1,C	DJNZ dir,rel 3-2	XCHD A,@R0 1-1	XCHD A,@R1 1-1
Ex	MOVX A,@DPTR 1-2	AJMP addr11 2-2	MOVX A,@R0 1-2	MOVX A,@R1 1-2	CLR A 1-1	MOV A,dir 2-1	MOV A,@R0 1-1	MOV A,@R1 1-1
Fx	MOVX @DPTR,A 1-2	ACALL addr11 2-2	MOVX @R0,A 1-2	MOVX @R1,A 1-2	CPL A 1-1	MOV dir,A 2-1	MOVX @R0,A 1-1	MOVX @R1,A 1-1

Where to find instruction timing

- Also in the "instruction definitions"

INSTRUCTION DEFINITIONS

ACALL addr11

Function: Absolute Call

Description: ACALL unconditionally calls a subroutine located at the indicated address. The instruction increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the Stack Pointer twice. The destination address is obtained by successively concatenating the five high-order bits of the incremented PC, opcode bits 7-5, and the second byte of the instruction. The subroutine called must therefore start within the same 2K block of the program memory as the first byte of the instruction following ACALL. No flags are affected.

Example: Initially SP equals 07H. The label "SUBRTN" is at program memory location 0345 H. After executing the instruction,

ACALL SUBRTN

at location 0123H, [illegible] locations 08H and 09H will contain 25H and 01H, resp[illegible]

Bytes: 2

Cycles: 2

This means "instruction cycles" not "oscillator" (on prev. slide)

Encoding:

a10 a9 a8 1	0 0 0 1	a7 a6 a5 a4	a3 a2 a1 a0

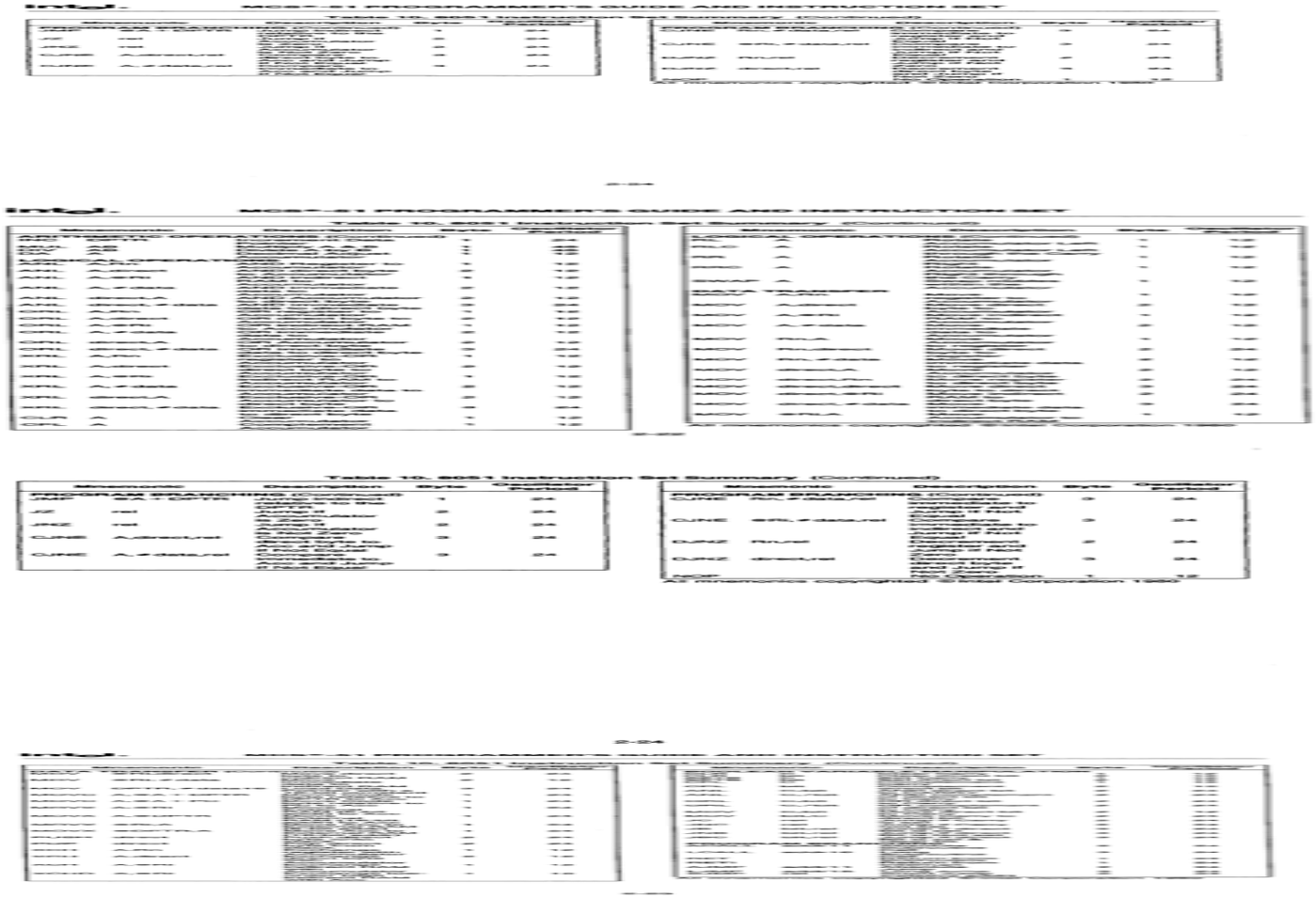
Operation:

ACALL
(PC) ← (PC) + 2
(SP) ← (SP) + 1
((SP)) ← (PC_{7-0})
(SP) ← (SP) + 1
((SP)) ← (PC_{15-8})
(PC_{10-0}) ← page address

Example: a fixed-delay subroutine (1st vers.)

- ```
  void delay() {
    unsigned char i;
    for (i = 255; --i != 0; );
  }
  ```

- But how precise is the delay?
  does sdcc generate something like this?

```
_delay:     MOV     R5, #0FFH       ;; R5=0xff
AGAIN:      DJNZ    R5, AGAIN       ;; while(--R5);
            RET                     ;; return
```
```

Exact instruction timing

_delay:	MOV	R5, #0FFH	;; R5=0xff	1µs (@12MHz)
AGAIN:	DJNZ	R5, AGAIN	;; while(--R5);	255 x 2µs
	RET		;; return	2µs



Total time assuming 12MHz osc. frequency: 513µs

a longer delay subroutine (2nd vers.)

- ```
  void delay() {
    unsigned char i;
    for (i = 255; --i; ) {
      unsigned char j;
      for (j = 255; --j; );
    }
  }
  ```

Does sdcc generate the following assembly? How long does it really take to execute?

```
_delay:   MOV   R4, #255    ;; R4=255;do{
outer:    MOV   R5, #255    ;; R5=255;
inner:    DJNZ  R5, inner   ;; do {}while(--R5);
          DJNZ  R4, outer   ;;}while(--R4);
          RET               ;; return
```
```

Exact timing for nested delay loops

				instr cycles	
_delay:	MOV	R4, #255	;; R4=255;do{	1	
outer:	MOV	R5, #255	;; R5=255;	1	x 255
inner:	DJNZ	R5, inner	;; do {}while(--R5);	2	x 255 x 255
	DJNZ	R4, outer	;;}while(--R4);	2	x 255
	RET		;; return	2	

Total: 130,818 instr. cycles
scaled by 1µs / instr. cycle
=> 130,818µs

A function needs to save & restore registers

```
_delay:   PUSH   4          ;; save registers
          PUSH   5          ;; R4, R5
          MOV    R4, #255   ;; R4=255;do{
outer:    MOV    R5, #255   ;; R5=255;
inner:    DJNZ   R5, inner  ;; do {}while(--R5);
          DJNZ   R4, outer  ;;}while(--R4);
          POP    5          ;; restore registers
          POP    4          ;; in reverse order
          RET               ;; return
```

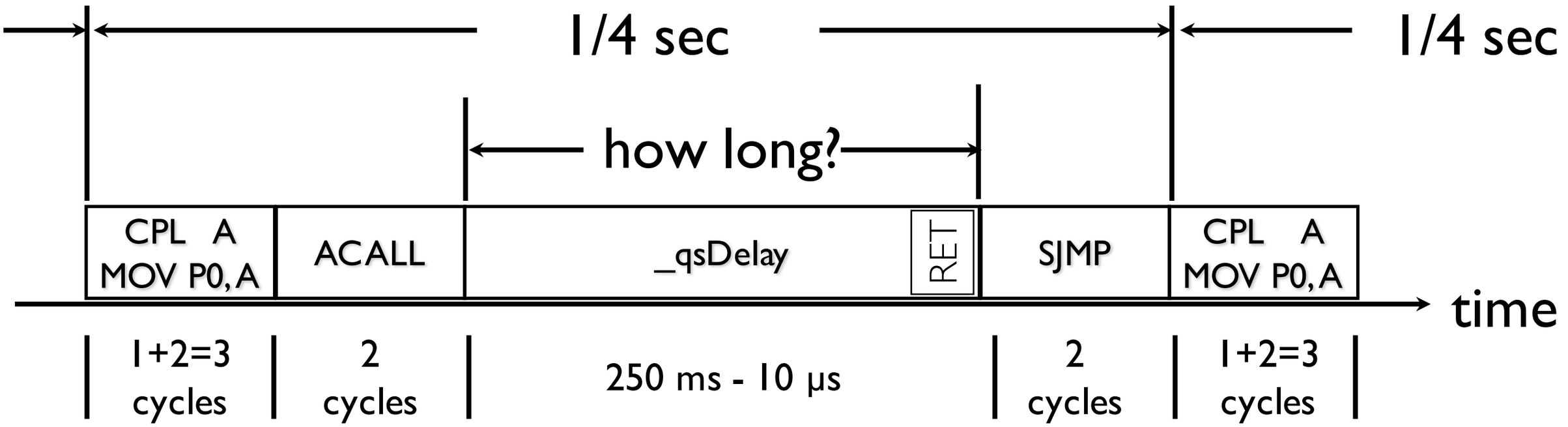
Ex: toggle bits every 1/4 sec -- will this work?

```
void Main(void)__naked {
P0 = 0xff;
while (1) {
    P0 = ~P0;
    qsDelay();
}
}
void qsDelay(void) {
???
}
```

```
        ORG     0
        MOV     A#0FFH
BACK:   CPL     A
        MOV     P0, A
        ACALL   _qsDelay
        SJMP    BACK

_qsDela
y:      ???

        ???

        RET
```

Timing control

- Q: How much to delay for ``every 1/4 sec"?
  - Delay takes effect relative to the ACALL
  - ACALL, SJMP, other overhead need to be accounted for



To call the delay() function

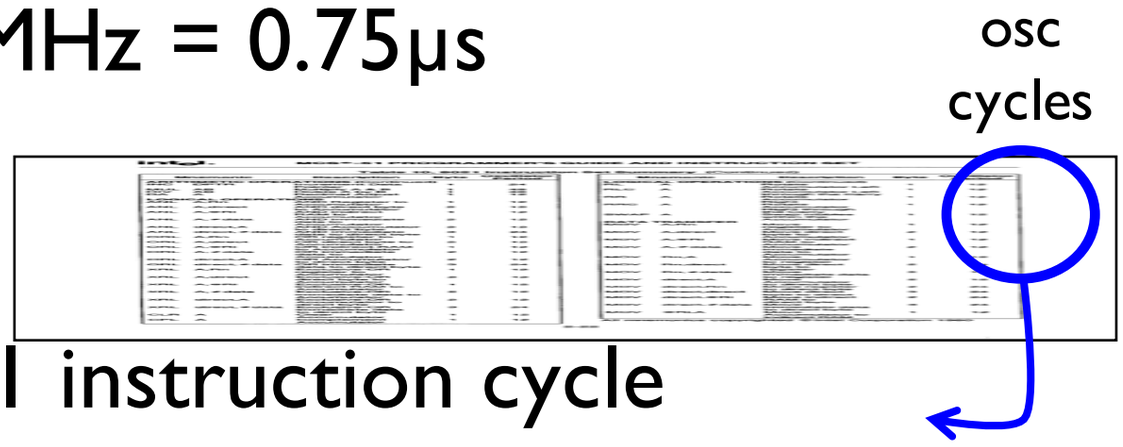
- Caller
  - LCALL _delay **or,** ACALL _delay
- Callee (the _delay **subroutine)**
  - Saves registers (push), and restores (pop)
- Parameter passing and return values
  - in registers or on the stack

Stack during Calls

- Intel: little-endian byte order
  - Low-order byte gets pushed first
  - High-order byte at a higher address
- PUSH/POP must match up
  - 8051 Stack grows from lower to higher address
  - By the time of RET, must be back to address pushed by call, or else trouble!

Oscillator vs Machine cycle

- e.g,. Oscillator Frequency = 16MHz
  - Machine cycle = 16/12
    = 1.33MHz instruction cycle frequency
    => cycle time = 1/1.33MHz = 0.75µs



- Instruction timing
  - MOV reg, #imm => 1 instruction cycle
    NOP => also 1 instruction cycle
  - DJNZ reg, target => 2 cycles

Cycle vs. Delay Calculation

label	instruction		Cycles
	ACALL	DELAY	**2**
	...		
DELAY:	MOV	R3, #200	**1**
HERE:	DJNZ	R3, HERE	**2**
	RET		**2**

- **Cycles =** ACALL**+**MOV**+ 200***DJNZ **+** RET
  = 2 + 1 + 200*2 + 2 = 405 cycles
- **Delay = 405 cycles * machine cycle time**

Implementation-dependent cycle time

- Clocks per instruction cycle
  - 12 (Atmel, Intel), 6 (Philips P89C54X2), 4 (Dallas Semi DS5000), 1 (DS89C)
- # instruction cycles per instruction
  - MOV *reg*, #imm: 1 (Intel), 2 (DS89C)
    DJNZ *reg*, *relTarget* 2 (Intel), 4 (DS89C)
    MUL 4 (Intel), 9 (DS89C)

Issue with Timing

- Instruction timing
  - Simple, but implementation-dependent
  - Relative timing (delay),
    not as good for absolute time control
- Timer
  - Independent hardware running in parallel
  - still dependent on oscillator

Timing Control using Timers

What is a timer

- A register whose value is auto-incremented
  - instruction to start/stop read/write reg.
  - counts the number of cycles elapsed

- 8051 has two timers: T0, T1
  - T0 accessed as TL0 (lower), TH0 (higher)
    T1 accessed as as TL1, TH1 (SFRs)
  - Resolution: 1/12 of XTAL oscillator freq.
    e.g., 12MHz XTAL => 1MHz => 1μs timer unit

Timer hardware

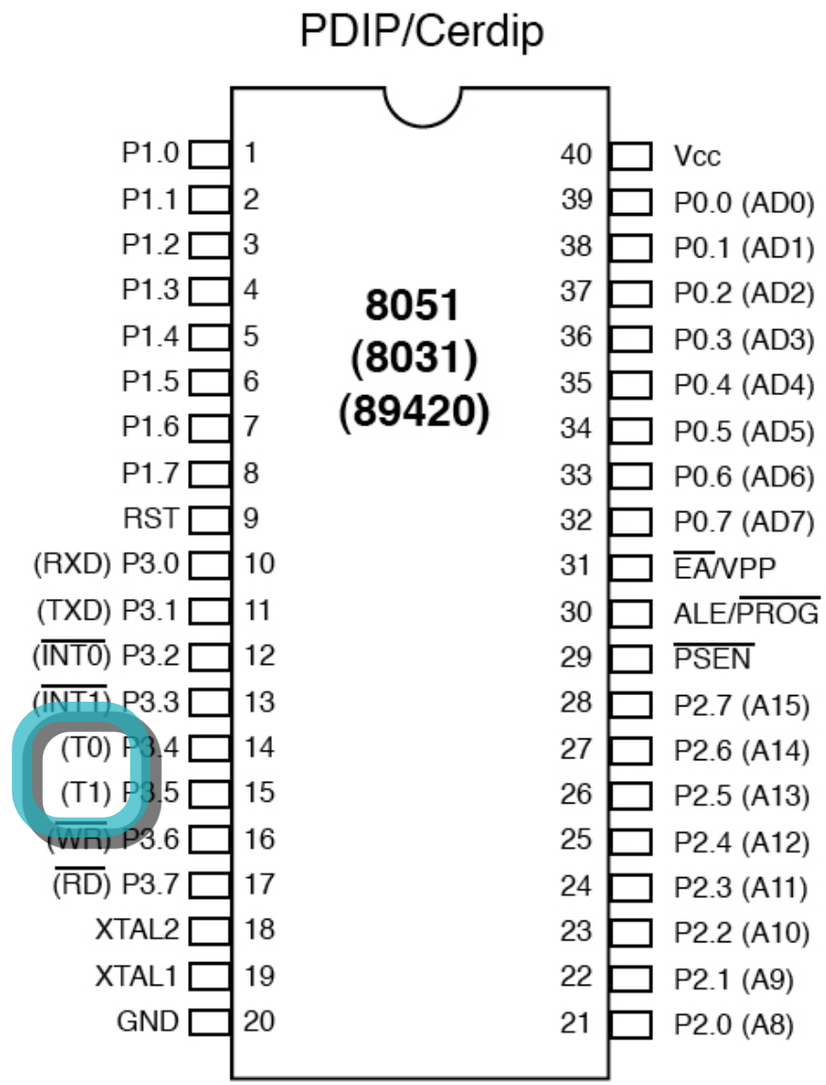
- Basically, counters
  - +1 on each rising-edge of "clock" pulse
    - Raises flag on rollover (FFFF to 0000)
- Two sources of "clock"
  - Oscillator => timer
  - External digital input pin => counter

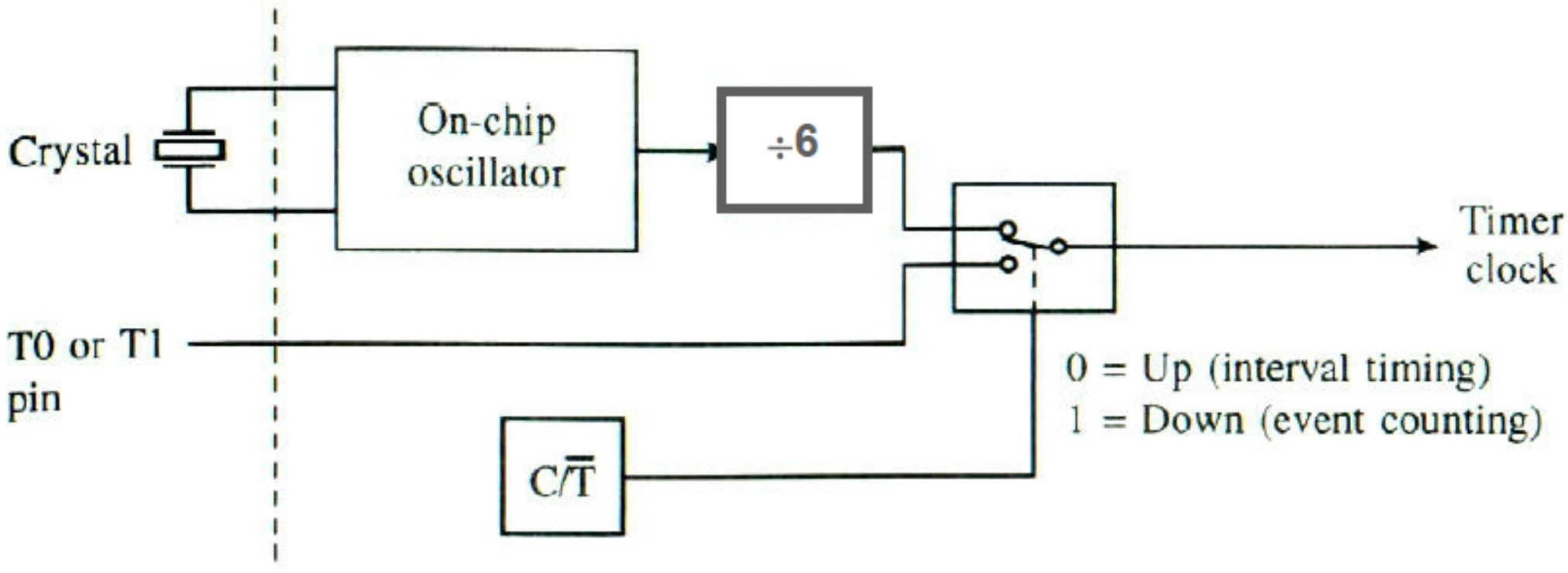
"Counter" vs. "Timer"

- Both use the same hardware!
- Difference is the source of pulses to count
  - Timer: counts crystal oscillator pulses
  - Counter: counts pulses from T0 / T1 pins
- Example Usage:
  - Counter: triggered by input or ext. clock (event counting e.g. how many times has a user pressed a button)
  - Timer: drive output or trigger sampling (interval timing e.g. delay)

Pins on 8051 (40-pin)

- Two timer/counter pins
  - T0, T1
  - Input pins
- Actually, used for counter, not timer

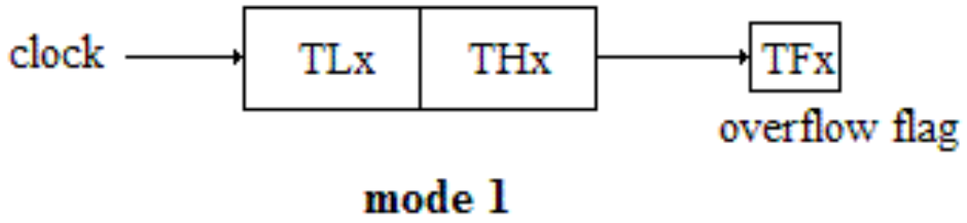



Crystal
On-chip oscillator
÷6
Timer clock
T0 or T1 pin
C/T
0 = Up (interval timing)
1 = Down (event counting)

Timer Registers

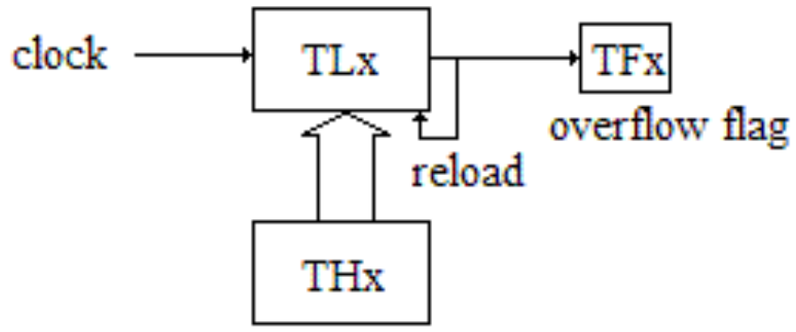
- TCON: Timer Control
- TMOD: Timer Mode
- TH0/TL0: Timer 0 16-bit register
- TH1/TL1: Timer 1 16-bit register

Timer Modes Overview



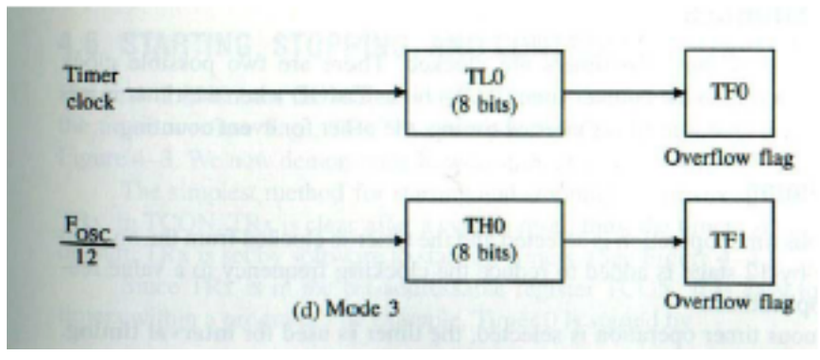
mode 1

- Mode 1: 16-bit mode.
  - Counts 0000H to FFFFH
  - TFx flag set when rollback to 0000H



mode 2

- Mode 2: 8-bit autoreload mode.
  - TLx Counts up to FFH
  - When rollback, TLx is reloaded with THx contents
  - TFx flag set when rollback to 00H



- Mode 3: split timer mode.
  - Counts 00H to FFH
  - TL0 sets TF0 flag
  - TH0 sets TF1 flag

How to use Timer T0

- Configure Timer Mode (0, 1, 2, or 3)
  - Whether to start by sw or hw trigger
- Load starting values into timer registers (TH0, TL0). **To delay *x* cycles, load *-x***
- **Start: (from software),** SETB TR0
- **Check flag** TF0 **for roll-over**
- **Stop: (from software):** CLR TR0

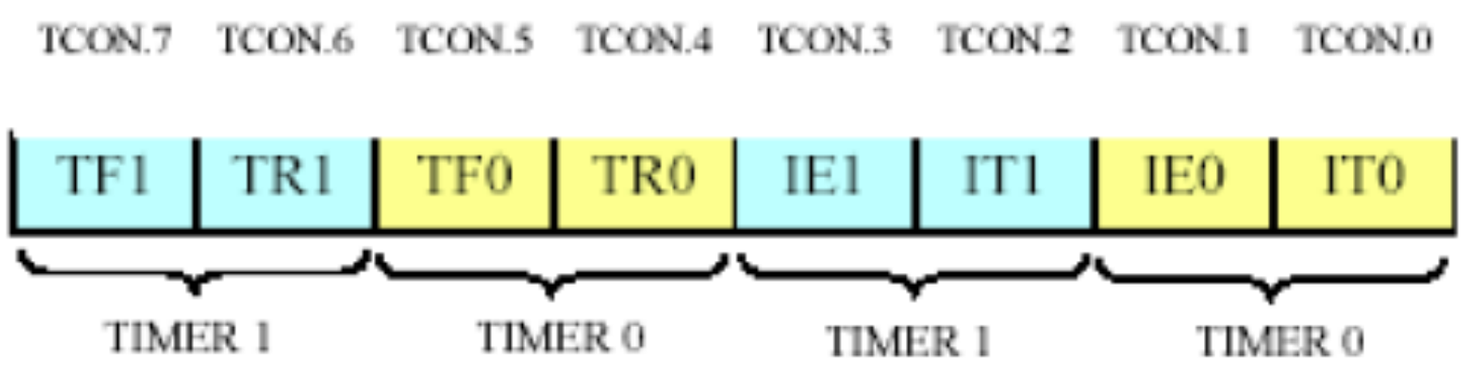
SFRs involved

Programmer sets this

Timer 1	Timer 0	purpose
TMOD<7:4>	TMOD<3:0>	timer mode
TH1, TL1	TH0, TL0	high/low bytes for timer value
TR1 (=TCON.6)	TR0 (=TCON.4)	start(1), stop(0) ('R' => "run")
TF1 (=TCON.7)	TF0 (=TCON.5)	rollover flag ('F' => "flag")

Programmer checks this

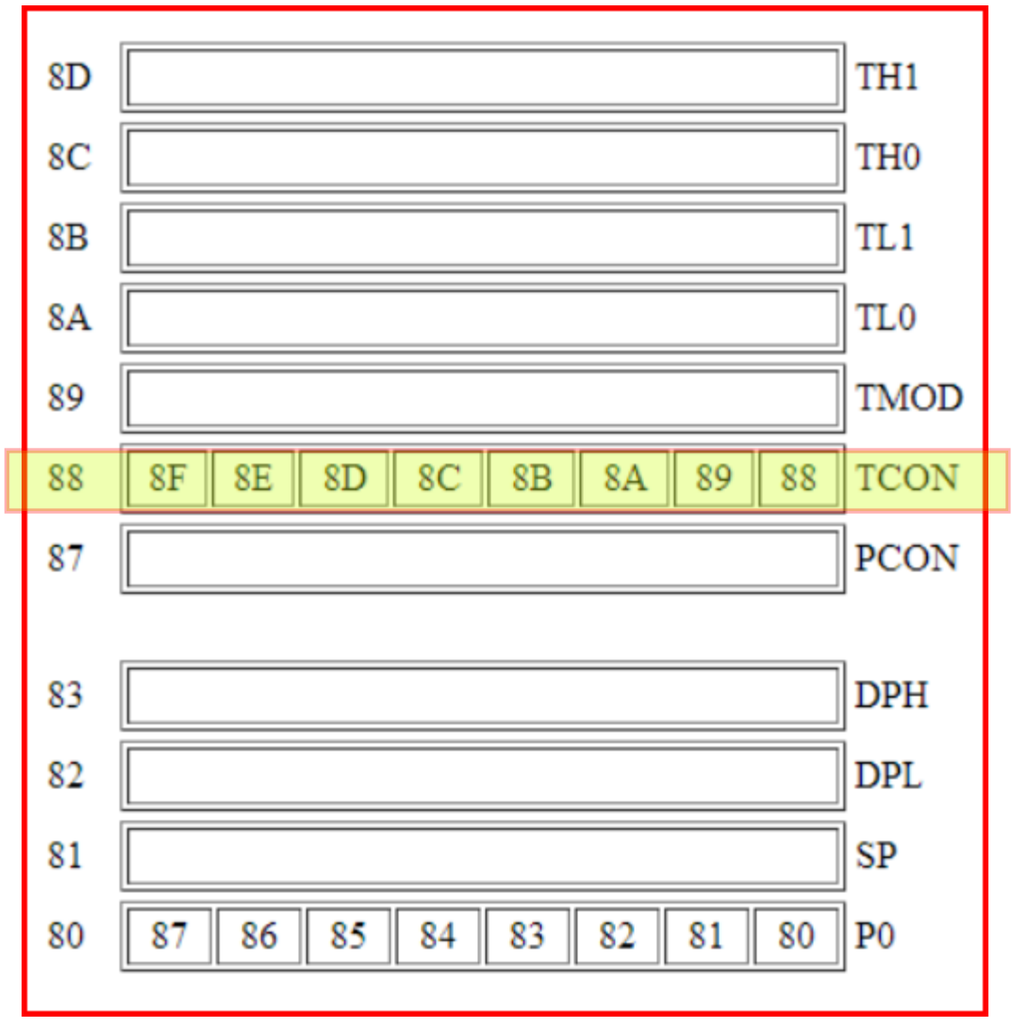
TCON



TCON SFR and its individual bits

- IT0/IT1: Used for timer Interrupts
- IE0/IE1: Used for external Interrupts
- TR0/TR1: Timer 0/1 run control flag
    - 1 = Run
- TF0/TF1: Timer 0/1 overflow flag
    - 1 = Overflow

Addressing TCON



- TCON has byte address 88H
- bit address of TCON.3: 88H + 3 = 8BH

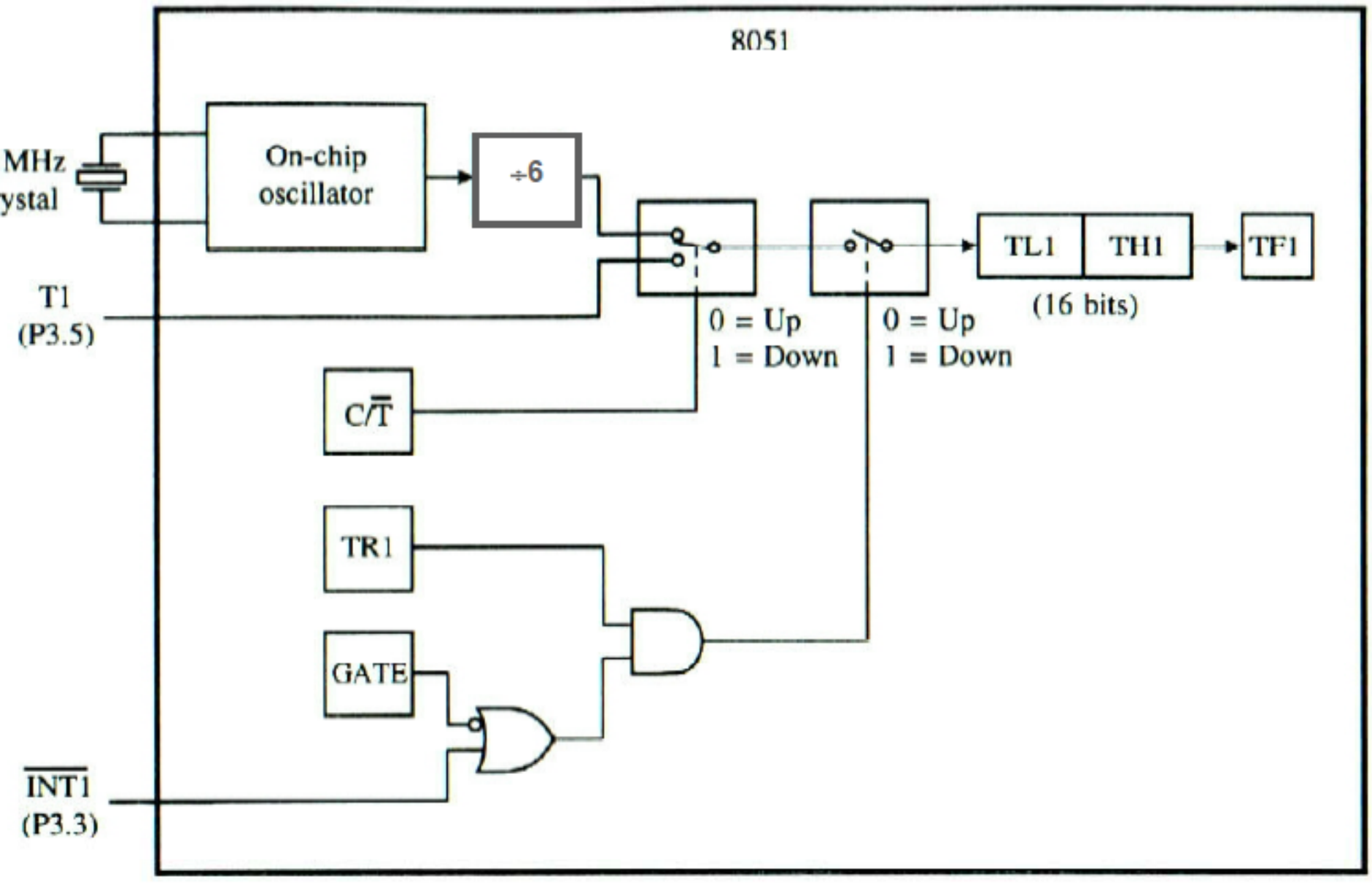
TMOD register ("timer mode")

- TMOD register:
  - TMOD<7:4> for Timer1
  - TMOD<3:0> for Timer0
- Fields:

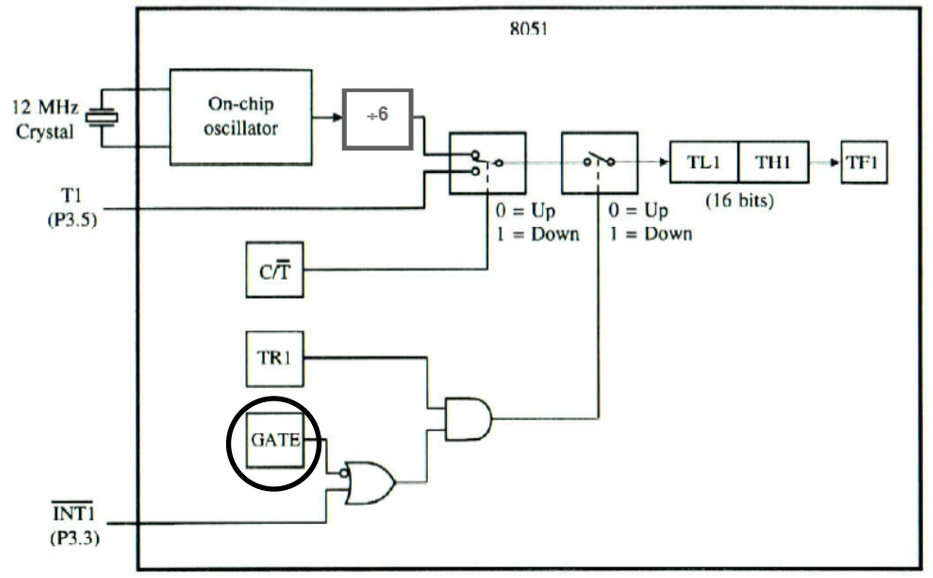
Timer 1				Timer 0			
gate	c/t	M1	M0	gate	c/t	M1	M0

TMOD.7 TMOD.0

Timer Architecture



GATE bit in TMOD

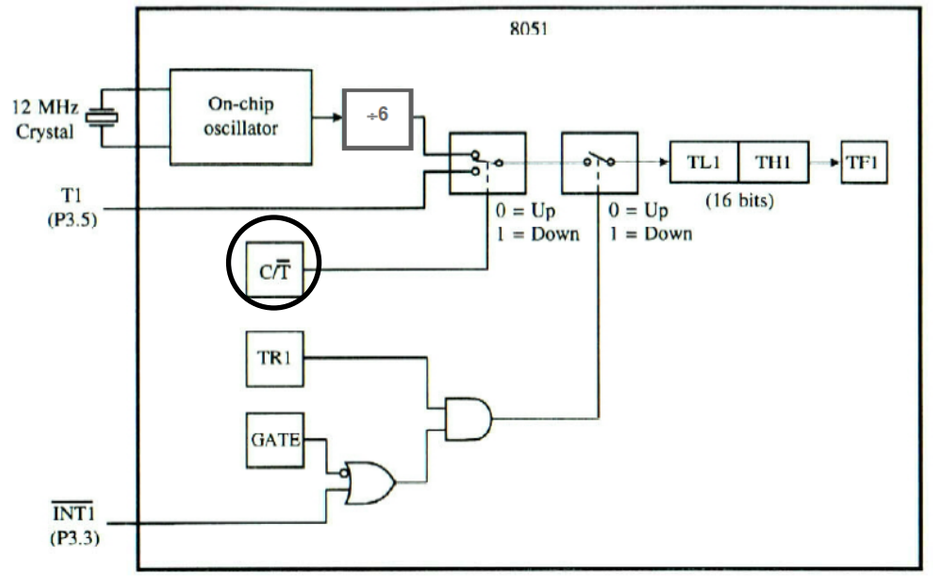


- 0: use internal (software) to start/stop
  - **Use** SETB/CLR **of** TR0 **or** TR1 **where** TR0=TCON.4, TR1=TCON.6
- 1: use external (hardware pin) to start/stop

Timer 1				Timer 0			
gate	c/t	M1	M0	gate	c/t	M1	M0

TMOD.7 TMOD.0

C/T bit in TMOD

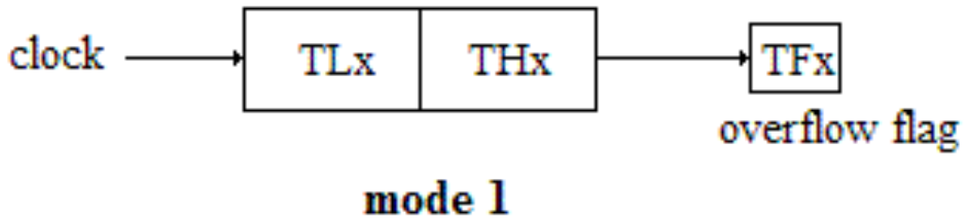


- 0: timer mode
  - Counts crystal cycles
- 1: counter mode
  - Counts number of pulses on T0 or T1 pin

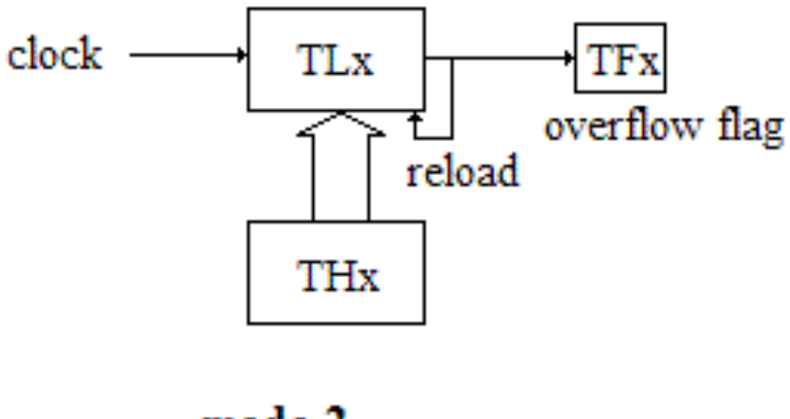
Timer 1				Timer 0			
gate	c/t	M1	M0	gate	c/t	M1	M0

TMOD.7 TMOD.0

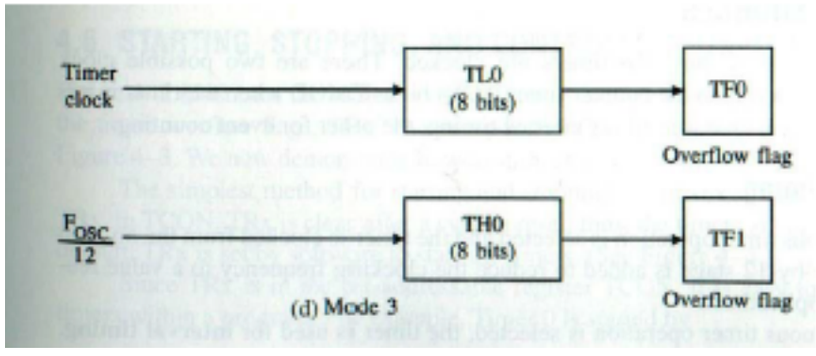
M1,M0 bits in TMOD



- 00: Mode 0: 13-bit timer
- 01: Mode 1: 16-bit timer
- 10: Mode 2: 8-bit auto-reload



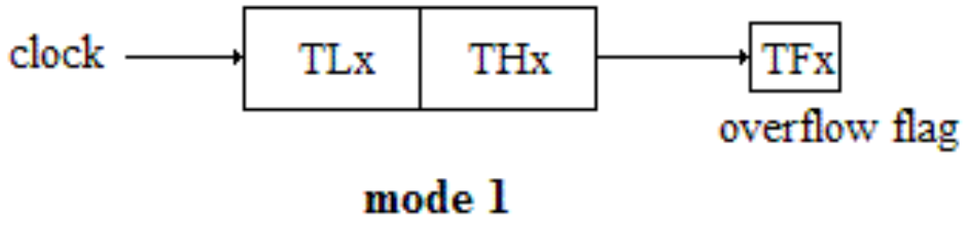
- 11: Mode 3: split timer, for timer/counter0 (two 8-bit timers or one 8-bit counter)



Timer 1				Timer 0			
gate	c/t	M1	M0	gate	c/t	M1	M0

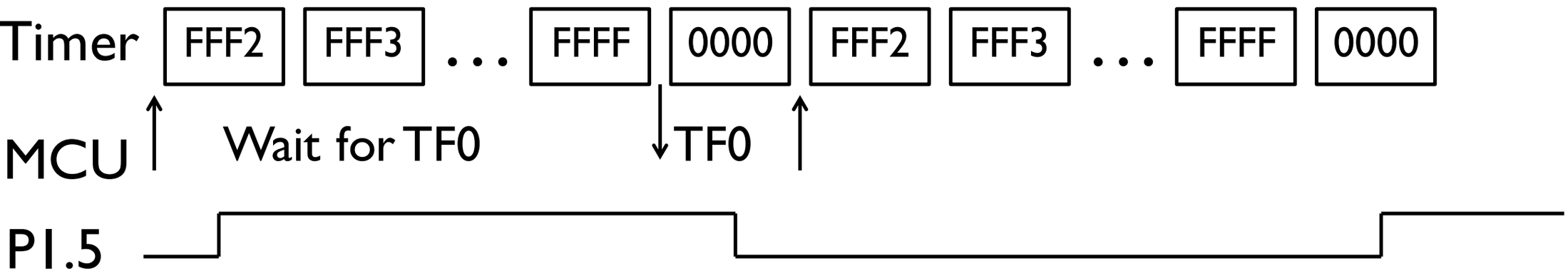
TMOD.7 TMOD.0

Timer mode 1



- 16-bit timer
  - (e.g. Timer 0, mode 1 -> TCON = #01H)
  - Start time value loaded into TL0,TH0 or TL1,TH1
    - SETB TR0 or SETB TR1 to start timer
- Count-up timer
  - when rollover (from FFFF to 0000), sets the TF0 or TF1 flag (Timer Flag)
  - Stop the timer by CLR TR0 or CLR TR1

Example code for 50% duty cycle using Timer0




```
        MOV     TMOD, #01       ;; set timer0 mode 1
HERE:   MOV     TL0,#-14        ;; 14 times (F2...00)
        MOV     TH0, #-1        ;; 1 time (FF...00)
        CPL     P1.5            ;; toggle output bit
        ACALL   DELAY
        SJMP    HERE
DELAY:  SETB    TR0             ;; start timer0
AGAIN:  JNB     TF0, AGAIN      ;; poll till rollover 00
        CLR     TR0             ;; stop timer0
        CLR     TF0             ;; clear timer0 flag
        RET
```

Timer rolls over every 14 instr cycles

but, period of loop includes additional overhead!

How long is the timer loop in DELAY?

- Given oscillator frequency 11.0592MHz?
- Timer period is 12/11.0592MHz = 1.085μs
- Counter range is FFF2, FFF3, ... 0000
  - rolls over every 14 times=> 15.19μs
    high time, low time
  - Entire period = x2 = 30.38μs (28us @12MHz)

Precise timing of code?

				#cycles
	MOV	TMOD, #01	;; set timer0 mode1	2 (once)
HERE:	MOV	TL0,#-14	;; from F2 to 00	2/loop
	MOV	TH0, #-1	;; from FF to 00	2/loop
	CPL	P1.5	;; toggle output bit	1/loop
	ACALL	DELAY		2/loop
	SJMP	HERE		2/loop
DELAY:	SETB	TR0	;; start timer0	1/call
AGAIN:	JNB	TF0, AGAIN	;; poll till rollover 00	14/call
	CLR	TR0	;; stop timer0	1/call
	CLR	TF0	;; clear timer0 flag	1/call
	RET			2/call

9+19 = 28 cycles

19 cycles !

28 cycles x 1.085µs = 60.76µs period (54@ 12MHz)