

Timing Control

Timing Control

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

Instruction Timing

Instructions take time

- # cycles depend on Instruction & addr mode
- Instr. time = #cycles * oscillator cycle time
- Original 8051: 1 instr. cycle = 12 osc. cycles
 - e.g., 12 MHz osc. freq = 1 MHz instr. freq
=> $1\mu\text{s}$ instr. cycle
- CC2540: 1 instr. cycle \sim 1 oscillator cycle
 - Frequency = 16 MHz or 32 MHz

Where to find timing of instructions?

- ISA Manual for 8051, page 2-21 on
- http://lms.nthu.edu.tw/sys/read_attach.php?id=414788

•

intel.

MCS[®]-51 PROGRAMMER'S GUIDE AND INSTRUCTION SET

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			Instruction	Flag		
	C	OV	AC		C	OV	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						

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

Where to find instruction timing

- Also in the "instruction definitions"

intel.

MCS®-51 PROGRAMMER'S GUIDE AND INSTRUCTION SET

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.

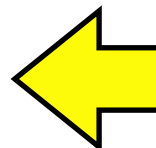
Example: Initially SP equals 07H. The label "SUBRTN" is at program memory location 0345 H. After executing the instruction,

ACALL SUBRTN

at location 0123H, SP will contain 09H, internal RAM locations 08H and 09H will contain 25H and 01H, respectively, and the PC will contain 0345H.

Bytes: 2

Cycles: 2



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

Instruction Timing on CC2540

- <http://www.ti.com/lit/ug/swru191e/swru191e.pdf>
- Page 37, Table 2-3 Instr. Set Summary
- However, the timing may depend on **flash prefetch**

Mnemonic	Description	Hex Opcode	Bytes	Cycles
ARITHMETIC OPERATIONS				
ADD A,Rn	Add register to accumulator	28–2F	1	1
ADD A,direct	Add direct byte to accumulator	25	2	2
ADD A,@Ri	Add indirect RAM to accumulator	26–27	1	2
ADD A,#data	Add immediate data to accumulator	24	2	2
ADDC A,Rn	Add register to accumulator with carry flag	38–3F	1	1
ADDC A,direct	Add direct byte to A with carry flag	35	2	2
ADDC A,@Ri	Add indirect RAM to A with carry flag	36–37	1	2
ADDC A,#data	Add immediate data to A with carry flag	34	2	2
SUBB A,Rn	Subtract register from A with borrow	98–9F	1	1
SUBB A,direct	Subtract direct byte from A with borrow	95	2	2
SUBB A,@Ri	Subtract indirect RAM from A with borrow	96–97	1	2
SUBB A,#data	Subtract immediate data from A with borrow	94	2	2

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

Mnemonic		Description	Byte	Oscillator Period
MOV	Rn, #data	Move immediate data to register	2	12
DJNZ	direct,rel	Decrement direct byte and Jump if Not Zero	3	24
RET		Return from Subroutine	1	24

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?



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

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 $1\mu s$ / instr. cycle
 $\Rightarrow 130,818\mu s$

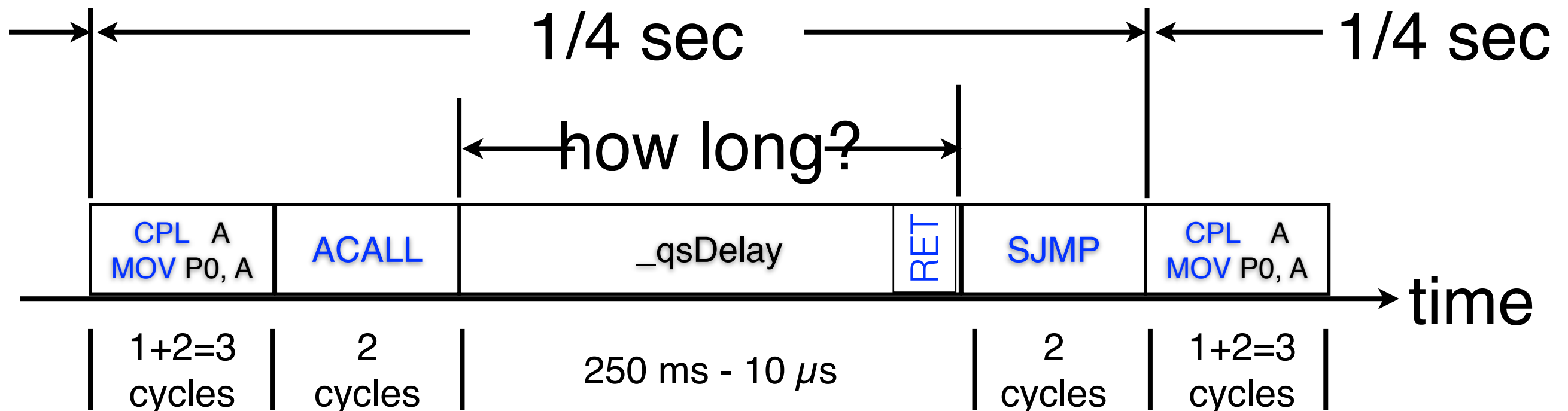
Ex: toggle bits every 1/4 sec -- will this work?

```
void Main(void) {  
    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  
  
_qsDelay: ???  
        ???  
        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

A function needs to save & restore registers

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

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!**

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

Oscillator cycle vs. Instruction cycle

- e.g., Oscillator Frequency = 16MHz
- Instruction cycle = $16/12$
= 1.33MHz instruction cycle frequency
=> cycle time = $1/1.33\text{MHz} = 0.75\mu\text{s}$

- Instruction timing

MOV	Rn, #data	Move immediate data to register	2	12
-----	-----------	---------------------------------------	---	----

- 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

- $\text{Cycles} = \text{ACALL} + \text{MOV} + 200 * \text{DJNZ} + \text{RET}$
 $= 2 + 1 + 200 * 2 + 2 = 405 \text{ cycles}$
- $\text{Delay} = 405 \text{ cycles} * \text{machine cycle time}$

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)

Comparison between traditional 8051 & CC2540

Instruction	8051	CC2540
ADD A, Rn	1	1
ADD A, dir	1	2
ADD A, @Ri	1	2
ADD A, #data	1	2
INC A	1	1
INC Rn	1	2
INC dir	1	3
INC DPTR	2	1
MUL AB	4	5
DIV AB	4	5

Instruction	8051	CC2540
ANL dir, A	1	3
ANL dir, #data	2	4
MOV A, Rn	1	1
MOV A, dir	1	2
MOV A, @Ri	1	2
MOV A, #data	1	2
MOV Rn, A	1	2
MOV Rn, dir	2	4
MOV Rn, #dat	1	2
MOV dir, A	1	3

(cont'd) Comparison between traditional 8051 & CC2540

Instruction	8051	CC2540
MOV dir, Rn	2	3
MOV dir, dir	2	4
MOV dir, @Ri	2	4
MOV dir, #data	2	3
MOV @Ri, A	1	3
MOV @Ri, dir	2	5
MOV @Ri, #da	1	3
MOV DPTR, #d16	2	3
MOVC A, @A+DPTR	2	3
MOVX A, @Ri	2	3

Instruction	8051	CC2540
MOVX A, @DPTR	2	3
PUSH dir	2	4
POP dir	2	3
XCH A, Rn	1	2
XCH A, dir	1	3
XCH A, @Ri	1	3
ACALL addr11	2	6
RET	2	4
AJMP addr11	2	3
LJMP addr16	2	4

(cont'd) Comparison between traditional 8051 & CC2540

Instruction	8051	CC2540
SJMP rel	2	3
JMP @A+DPT	2	2
JZ rel	2	3
CJNE	2	4
DJNZ Rn, rel	2	3
DJNZ dir, rel	2	4
NOP	1	1

- 8051:
 - 1 instr cycle = 12 osc cycles
 - osc Freq: 1-12 MHz
- CC2540:
 - 1 instr cycle = 1 osc cycle
 - osc Freq: 16 or 32 MHz
 - cycle count could be less if prefetch enabled!!

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 $\Rightarrow 1\mu\text{s}$ timer unit

Timer Hardware

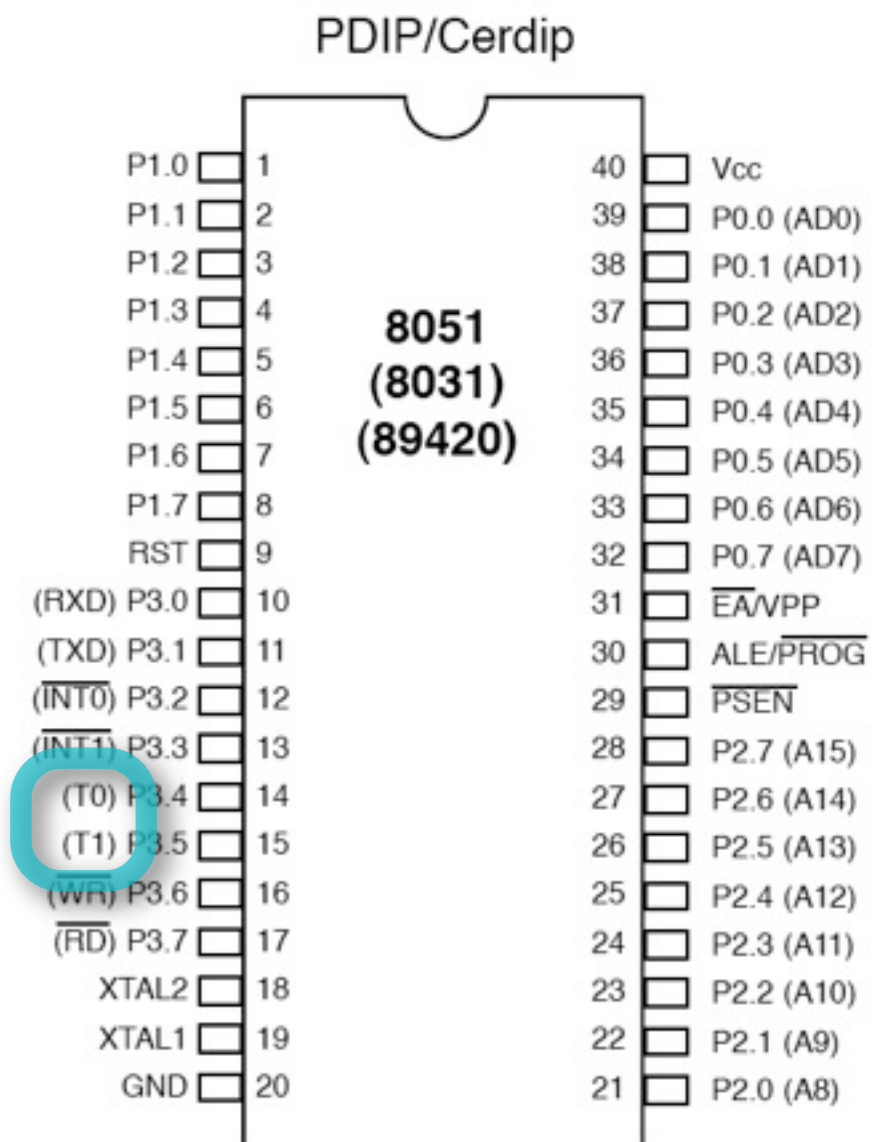
- Basically, a timer is a counter
 - +1 on each **rising-edge** of "clock" pulse
 - Raises flag on rollover (FFFF to 0000)
- Two sources of "clock"
 - Oscillator => timer
 - External digital input pin => counter

"Counter" vs. "Timer"

- Actually, they are the same hardware!
- **Difference:** 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
 - Timer: drive output or trigger sampling

Pins on 8051 (40-pin)

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



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

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")

TMOD: timer mode SFR

- TMOD is bit addressable:
 - 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

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
 - 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`

50% duty cycle using Timer0 in mode 1: works?

```
HERE:      MOV      TMOD, #01      ;; set timer0 mode 1
           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 12 MHz?
- Timer period is $1\mu s$
- Counter range is FFF2, FFF3, ... 0000
 - rolls over every 14 times=> $14\mu s$
high time, low time
 - Entire period = $x2 = 28\mu s$. or is it??

	SETB TR0	JNB TF0		JNB TF0		JNB TF0		JNB TF0		JNB TF0		JNB TF0		JNB TF0	
Counter	FFF2	FFF3	FFF4	FFF5	FFF6	FFF7	FFF8	FFF9	FFFA	FFFB	FFFC	FFFD	FFFE	FFFF	0000

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 2 x 1μs = 56μs period

Review: Timer 0, Timer1

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

TMOD.7	Timer 1				Timer 0				TMOD.0
	gate	c/t	M1	M0	gate	c/t	M1	M0	

Timer modes	Mode 0	Mode 1	Mode 2	Mode 3
	13-bit	16-bit	8-bit auto	8-bit

Longest possible delay of 16-bit timer?

- Start from 0000H, to FFFFH and roll over
= 65536 cycles
- At timer resolution of $1.085\mu s$,
 - $65536 \times 1.085\mu s = 71.1065ms$
- Can get more delay with software
- Actually, overhead should be included
(start/stop, load timer register, ...)

How precise can you make timing?

- e.g., 5ms at 12MHz oscillator freq
 - $5\text{ms} / 1\mu\text{s} = 5000$ clocks (exactly)
 - timer value = $\text{hex}(2^{16} - 5000) = \text{EC78}$
- but what about the overhead of turning on/off timer? checking flag?

Mode 0 vs Mode 1

- Mode 0: 13-bit timer
 - Range: 0000 to 1FFF hex
 - max steps = (8192 decimal)
- Mode 1: 16-bit timer
 - Range: 0000 to FFFF hex
- Manual reload required

Timer Mode 2: 8-bit, auto-reload

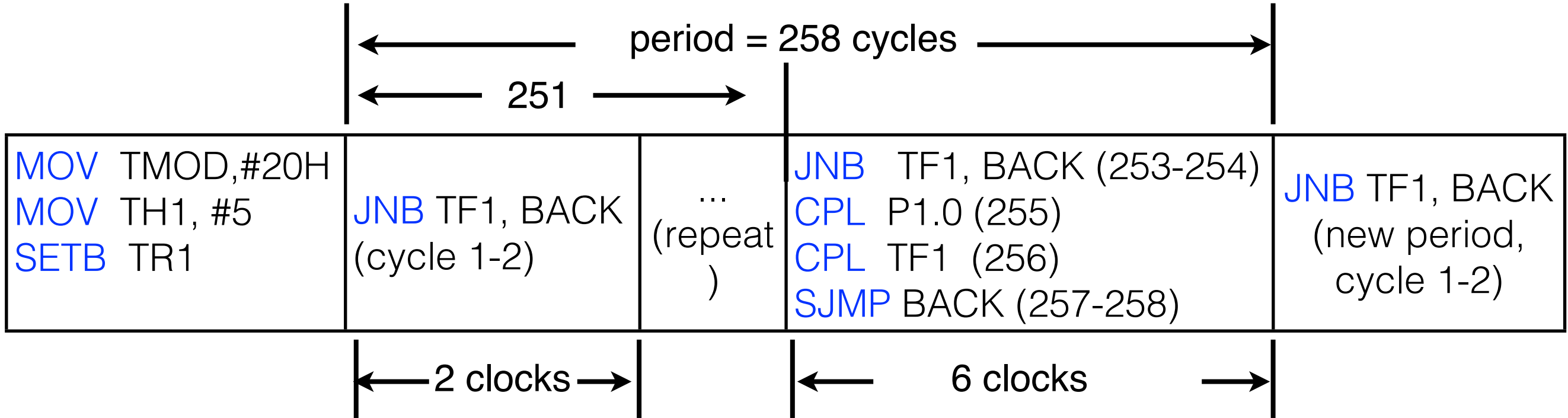
- Load 00-FFH into TH^* (* = 0 or 1)
 - in mode 2, TL^* gets a copy of TH^*
- Start by **SETB** TR^* , stop by **CLR** TR^*
- Rollover \Rightarrow sets TF^* just like before
- Difference: TL^* gets auto-reloaded w/ TH^*

Why auto-reload?

- Very convenient for periodic timing
 - same value each time, saves instructions
- No software overhead in the loop!
 - need instructions to setup and start/stop
 - Once running, works on its own
easy to compute period,
independent of instruction timing

Example: square wave

			<u>cycles</u>	
	MOV	TMOD, #20H	:: timer 1 mode 2	2
	MOV	TH1, #5	:: 251 clocks (=256-5)	2
	SETB	TR1	:: start timer	1
BACK:	JNB	TF1, BACK	:: poll till roll over	2
	CPL	P1.0	:: complement output bit	1
	CLR	TF1	:: clear flag; auto-reloaded	1
	SJMP	BACK	:: loop	2

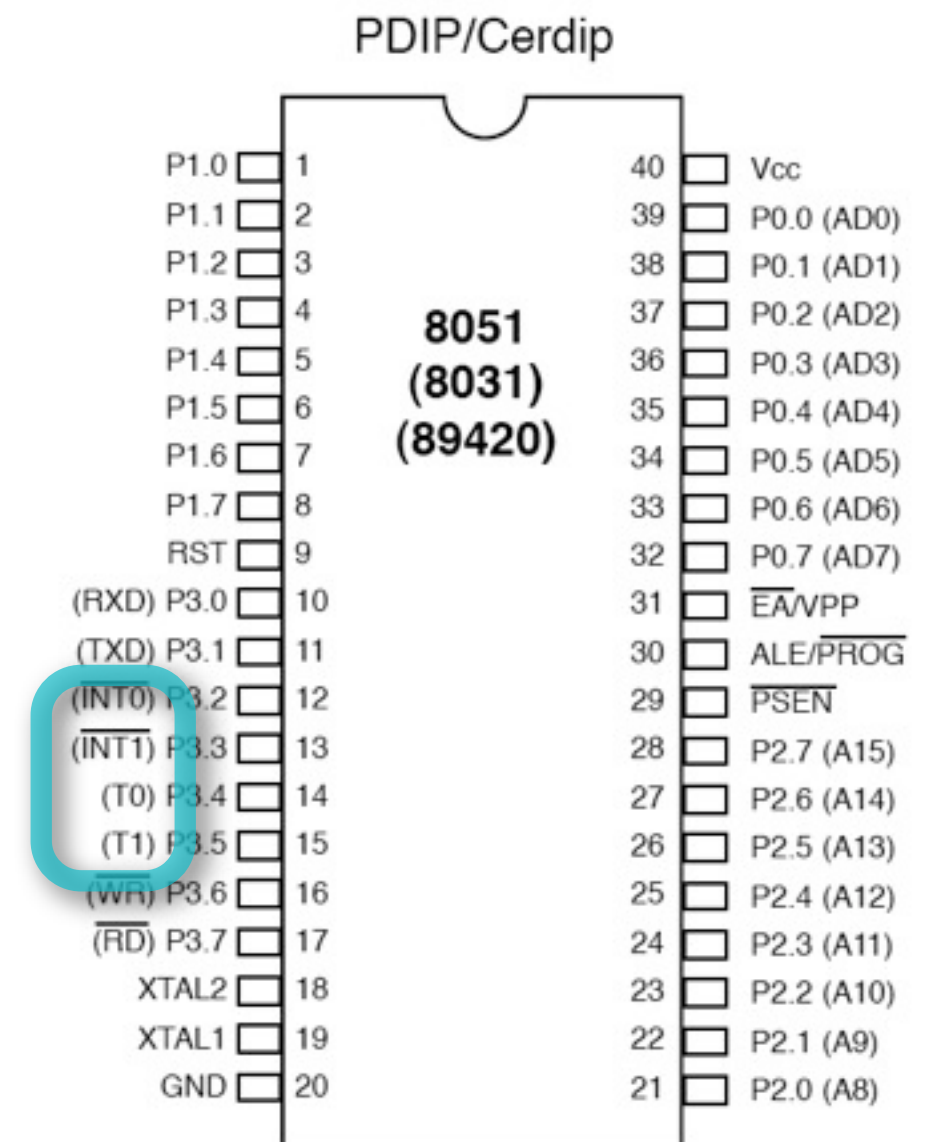


Counter vs. Timer, gate bit

- Counter & hardware use the same, diff src
 - Timer: clock from crystal oscillator/clock
 - Counter: input from T0, T1
- Gate bit:
 - 0: software `SETB/CLR` TR0 or TR1
 - 1: external pins `/INT0, /INT1`

Pins on 8051 (40-pin)

- Two timer/counter pins
 - T0, T1
- Input pins
- Gate bit
 - /INT0, /INT1



TCON register (SFR)

- timer control
 - the TF* and TR* flags
- interrupt control
 - IE1, IT1, IE0, IT0

TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0
-----	-----	-----	-----	-----	-----	-----	-----

more on GATE bit in TMOD

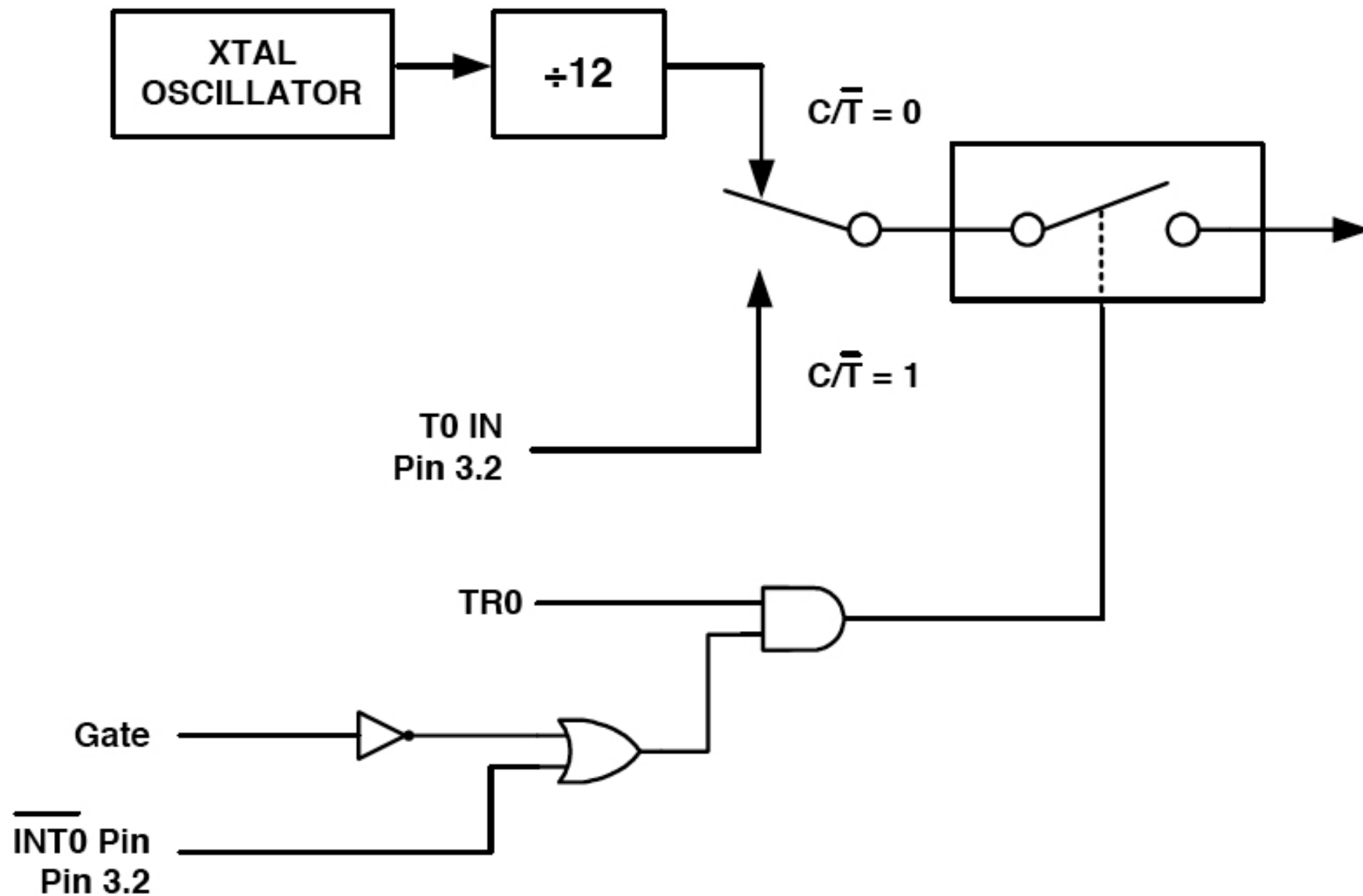
- 0: use internal (software) to start/stop
- 1: enables /INT0 or /INT1 pin to start/stop, while TR0 or TR1 is enabled.

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

TMOD.7

TMOD.0

Schematic for the GATE, /INT0, TR0



C programs for Timer

- Fundamentally not too different from asm
- assignment statement instead of MOV
- while (TF0==0) ; to poll TF0 flag.
- TR0 = 1 to start, TR0 = 0 to stop
- Delay accomplished by count-up roll-over

Example 1: delay1.c

(1/2)

```
#include <8051.h>
void T0Delay(void);
void main(void) {
    P1 = 0x55;
    while(1) {
        T0Delay();
        P1 = ~P1;
    }
}
```

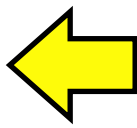
```
void T0Delay(void) {
    TMOD = 0x01;
    TL0 = 0x00;
    TH0 = 0x35;
    TR0 = 1;
    while (!TF0) ;
    TR0 = 0;
    TF0 = 0;
}
```

Periodic execution

- Delay: relative to a point in time
 - Mode 0, Mode 1 (13-bit, 16-bit delay)
 - not as good to use Delay for periodic
- Periodic (e.g., "do task every 500ms")
 - Mode 2 (with auto reload) may be better
 - however, need to be careful with overhead, since timer reg is only 8 bits

Ex. 2: Auto-Reload

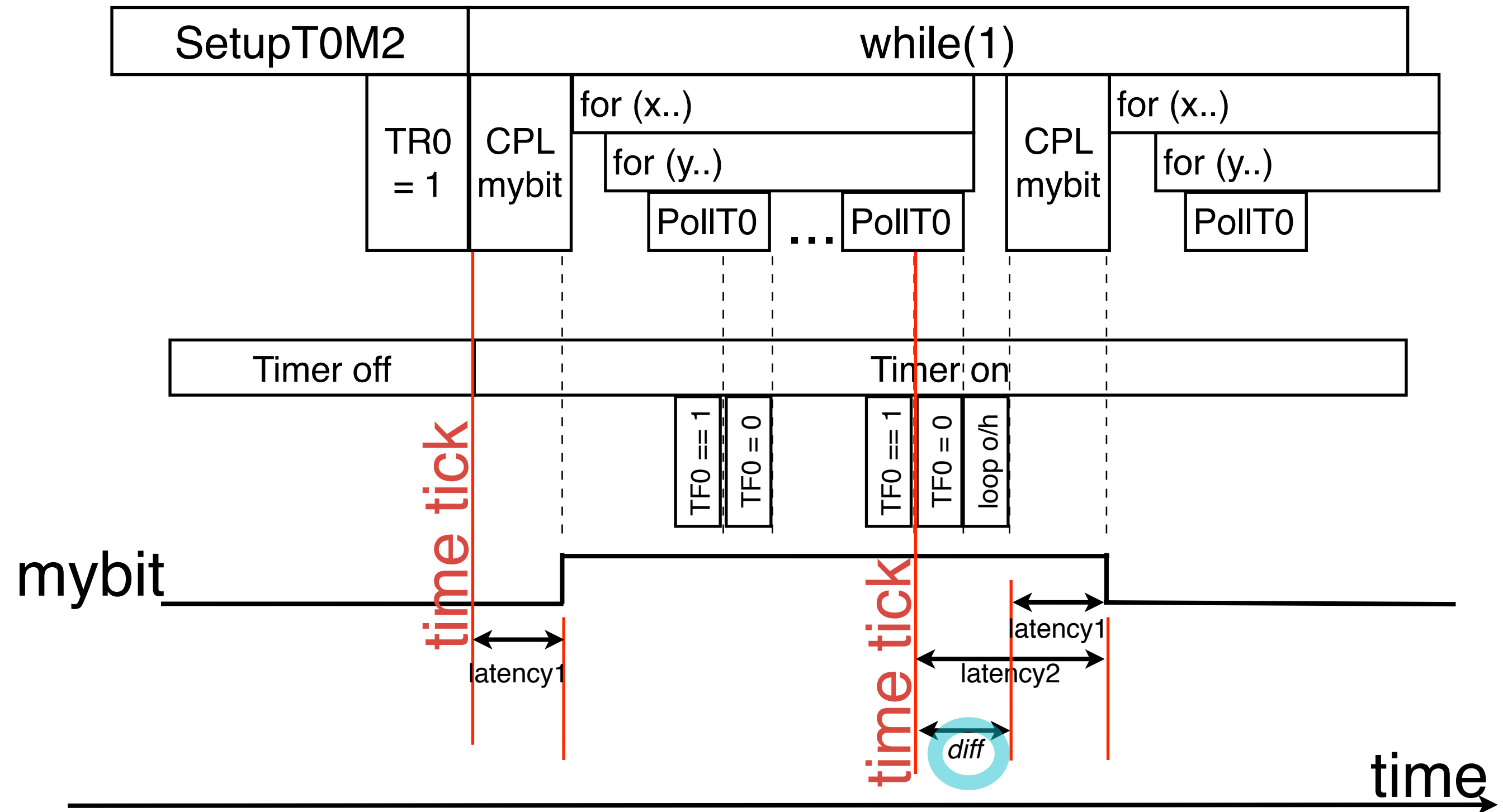
```
#include <8051.h>
void SetupT0M2(void), PollT0(void);
void main(void) {unsigned char x, y;
    SetupT0M2();
    while (1) {
        P1 = ~P1;
        for (x = 0; x < 250; x++) {
            for (y = 0; y < 36; y++) {
                PollT0();
            }
        }
    }
}
```



```
void SetupT0M2(void) {
    TMOD = 0x02;
    TH0 = -23;
    TR0 = 1;
}

void PollT0(void) {
    while (TF0 == 0) ;
    TF0 = 0;
}
```

Timing for better Ex 2



Summary of Ex 2

- Use auto-reload to absorb overhead
 - don't disable/re-enable timer on reloads!
- Line up with timer as precisely as possible
 - Immediately after start running timer
 - Immediately after polling TF going high
- 1st high interval longer by *diff*; others exact
 - easy to fix by padding nops