COMP-462 Embedded Systems

Lecture 7: Phase-locked-loop, Data structures, Finite state machines, Interrupts

Read Sections 4.1, 5.2, 6.1, and 6.2

Agenda

- ❑ Recap
  - ❖ Indexed Addressing and Pointers
    - o In C: Address of (&), Pointer to (*)
  - ❖ Data Structures: Arrays, Strings
    - o Length: hardcoded vs. embedded vs. sentinel
    - o Array access: indexed vs. pointer arithmetic
  - ❖ Functional Debugging
  - ❖ SysTick Timer
- ❑ Outline
  - ❖ Phase Lock Loop (PLL) ← Called by TExaS_Init
  - ❖ Use **`struct`** to create Data structures
  - ❖ Finite State Machines, Linked data structures
  - ❖ Interrupts

Recap: Array access (assembly)

- Calculate address from Base and index

Byte	Base+index
Halfword	Base+2*index
Word	Base+4*index
Size_N	Base+N*index

- Access sequentially using pointers

Byte	pt = pt+1
Halfword	pt = pt+2
Word	pt = pt+4
Size_N	pt = pt+N

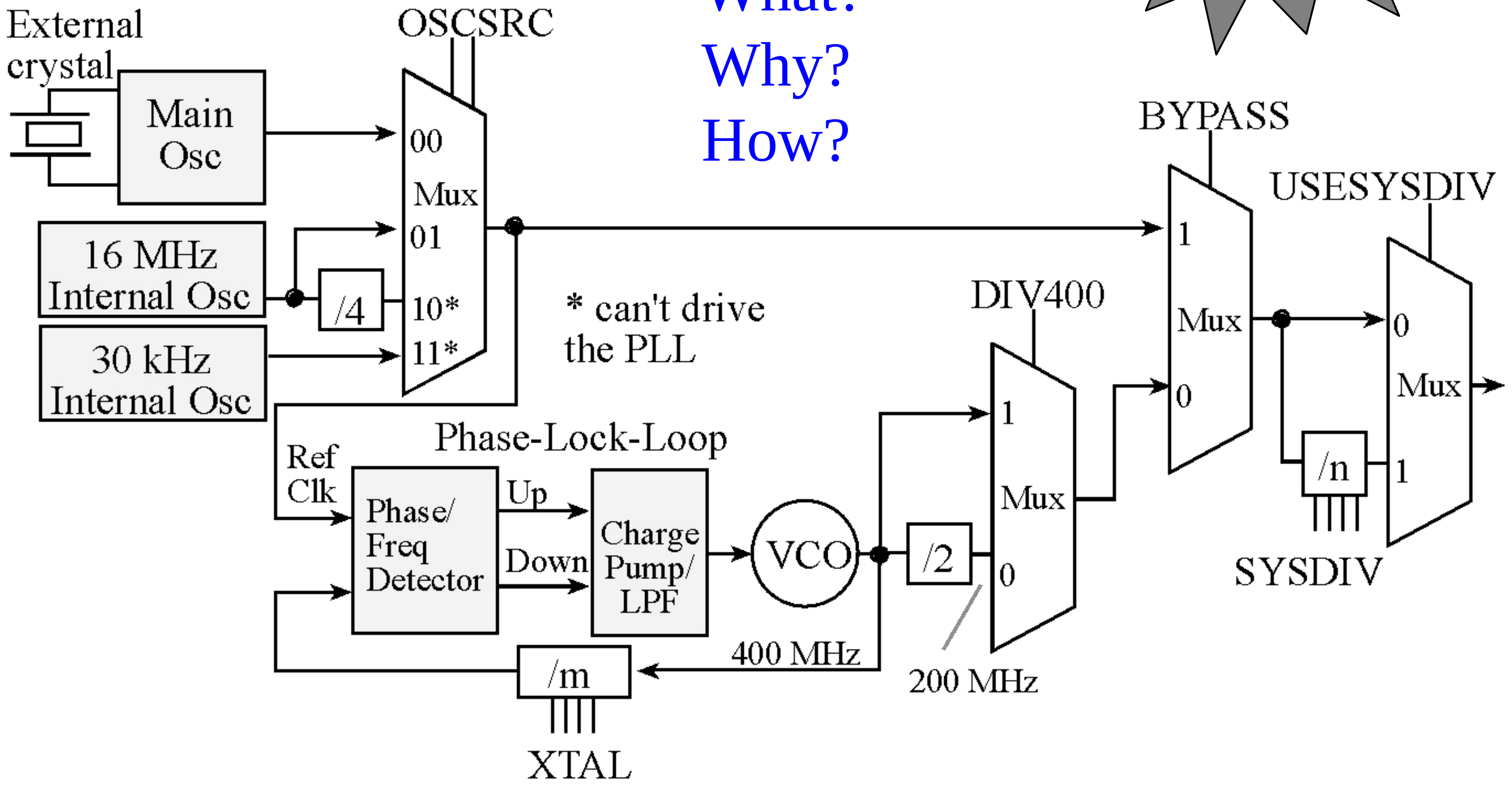
Recap: Array access (C)

- ❑Calculate address from Base and index
  - ❖Byte	array_name[index]
  - ❖Halfword	array_name[index]
  - ❖Word	array_name[index]
  - ❖Size_N	array_name[index]
- ❑Access sequentially using pointers
  - ❖Byte	pt = pt+1; *pt
  - ❖Halfword	pt = pt+1; *pt
  - ❖Word	pt = pt+1; *pt
  - ❖Size_N	pt = pt+1; *pt

Phase-Lock-Loop

Tolerance
$50*10^{-6}$

What?
Why?
How?



- ❑ **Internal oscillator requires minimal power but is imprecise**
- ❑ **External crystal provides stable bus clock**
- ❑ **TM4C123 is equipped with 16.000 MHz crystal and bus clock can be set to a maximum of 80 MHz**

Phase-Lock-Loop

XTAL	*Crystal Freq (MHz)*
0x0	Reserved
0x1	Reserved
0x2	Reserved
0x3	Reserved
0x4	3.579545 MHz
0x5	3.6864 MHz
0x6	4 MHz
0x7	4.096 MHz
0x8	4.9152 MHz
0x9	5 MHz
0xA	5.12 MHz
0xB	6 MHz (reset value)
0xC	6.144 MHz
0xD	7.3728 MHz
0xE	8 MHz
0xF	8.192 MHz

XTAL	*Crystal Freq (MHz)*
0x10	10.0 MHz
0x11	12.0 MHz
0x12	12.288 MHz
0x13	13.56 MHz
0x14	14.31818 MHz
0x15	16.0 MHz
0x16	16.384 MHz
0x17	18.0 MHz
0x18	20.0 MHz
0x19	24.0 MHz
0x1A	25.0 MHz
0x1B	Reserved
0x1C	Reserved
0x1D	Reserved
0x1E	Reserved
0x1F	Reserved

Address	26-23	22	13	11	10-6	5-4	Name
$400FE060	SYSDIV	USESYSDIV	PWRDN	BYPASS	XTAL	OSCSRC	SYSCTL_RCC_R
$400FE050	▨	▨	▨	▨	PLLRIS	▨	SYSCTL_RIS_R

	31	30	28-22	13	11	6-4	
$400FE070	USERCC2	DIV400	SYSDIV2	PWRDN2	BYPASS2	OSCSRC2	SYSCTL_RCC2_R

Phase-Lock-Loop

1) The first step is to set BYPASS2 (bit 11) (only for initialization).
2) specify the crystal frequency using four XTAL (0x15)
3) The third step is to clear PWRDN2 (bit 13) to activate the PLL.
4) The fourth step is to configure and enable the clock divider using the 7-bit SYSDIV2 field.

5) The fifth step is to wait for the PLL to stabilize by waiting for PLLRIS (bit 6) in the SYSCTL_RIS_R to become high.

6) The last step is to connect the PLL by clearing the BYPASS2 bit.

Phase-Lock-Loop

```
void PLL_Init(void){
  // 0) Use RCC2
  SYSCTL_RCC2_R |=  0x80000000;  // USERCC2
  // 1) bypass PLL while initializing
  SYSCTL_RCC2_R |=  0x00000800;  // BYPASS2, PLL bypass
  // 2) select the crystal value and oscillator source
  SYSCTL_RCC_R = (SYSCTL_RCC_R &~0x000007C0)   // clear XTAL field, bits 10-6
                 + 0x00000540;   // 10101, configure for 16 MHz crystal
  SYSCTL_RCC2_R &= ~0x00000070;  // configure for main oscillator source
  // 3) activate PLL by clearing PWRDN
  SYSCTL_RCC2_R &= ~0x00002000;
  // 4) set the desired system divider
  SYSCTL_RCC2_R |= 0x40000000;   // use 400 MHz PLL
  SYSCTL_RCC2_R = (SYSCTL_RCC2_R&~ 0x1FC00000)  // clear system clock divider
                  + (4<<22);      // configure for 80 MHz clock
  // 5) wait for the PLL to lock by polling PLLLRIS
  while((SYSCTL_RIS_R&0x00000040)==0){};  // wait for PLLRIS bit
  // 6) enable use of PLL by clearing BYPASS
  SYSCTL_RCC2_R &= ~0x00000800;
}
```

Abstraction

- Software abstraction
  - Define a problem with a minimal set of basic, abstract principles / concepts
  - Separation of concerns via interface/policy mechanisms
  - Straightforward, mechanical path to implementation

- Three advantages of abstraction are
  1. it can be faster to develop
  2. it is easier to debug (prove correct) and
  3. it is easier to change

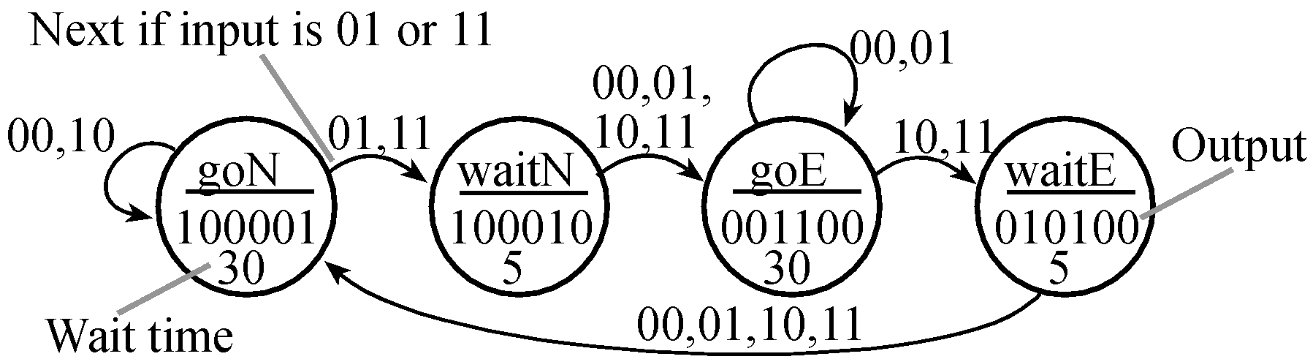
Finite State Machine (FSM)

- ❑ Finite State Machines (FSMs)
  - ❖ Set of inputs, outputs, states and transitions
  - ❖ State graph defines input/output relationship
- ❑ What is a state?
  - ❖ Description of current conditions
- ❑ What is a state graph?
  - ❖ Graphical interconnection between states
- ❑ What is a controller?
  - ❖ Software that inputs, outputs, changes state
  - ❖ Accesses the state graph

https://www.youtube.com/playlist?list=PLyg2vmIzGxXEle4_R2VA_J5uwTWktdWTu

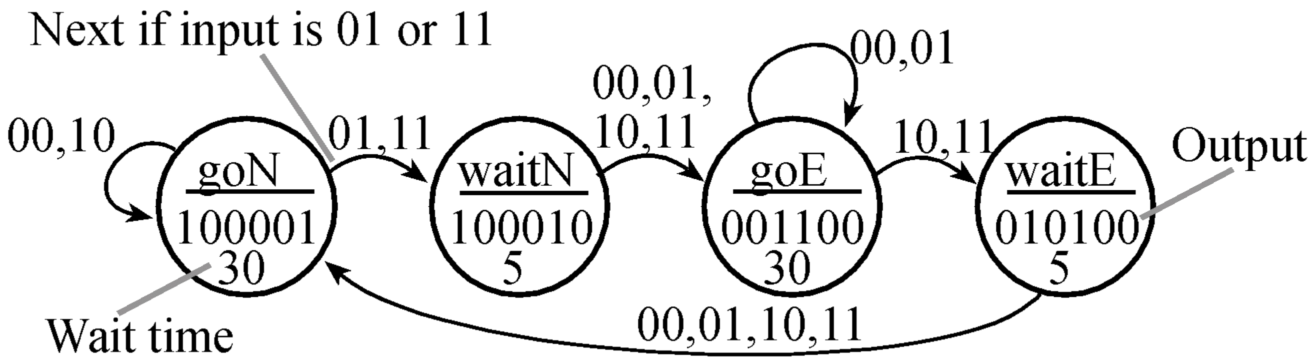
Finite State Machine (FSM)

- ❑What is a finite state machine?
  - ❖Inputs (sensors)
  - ❖Outputs (actuators)
  - ❖Controller
  - ❖State graph



Finite State Machine (FSM)

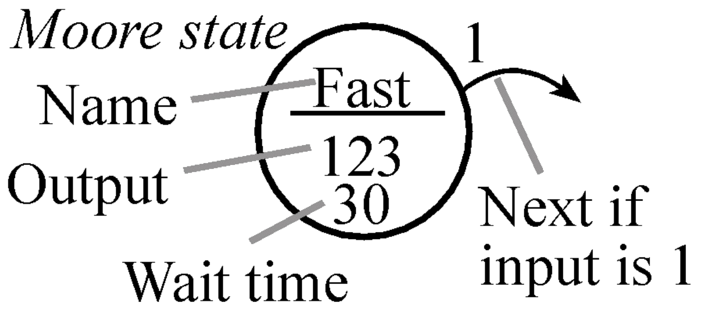
- ❑Moore FSM
  - ❖output value depends only on the current state,
  - ❖inputs affect the state transitions
  - ❖significance is being in a state
- ❑Input: when to change state
- ❑Output: definition of being in that state

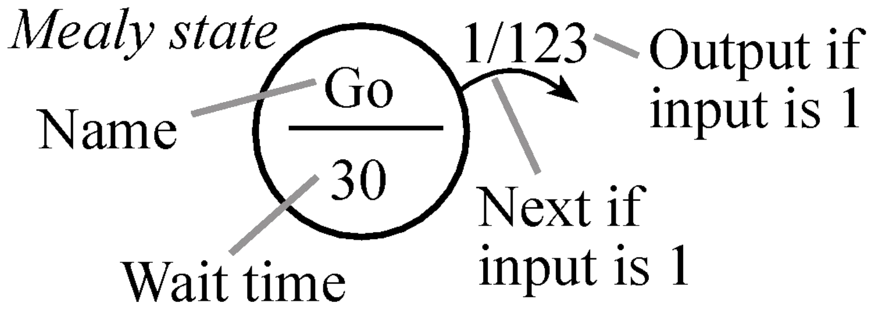


Finite State Machine (FSM)

- ❑ Moore FSM Execution Sequence
  1. Perform output corresponding to the current state
  2. Wait a prescribed amount of time (optional)
  3. Read inputs
  4. Change state, which depends on the input and the current state
  5. Go back to 1. and repeat

Finite State Machine (FSM)



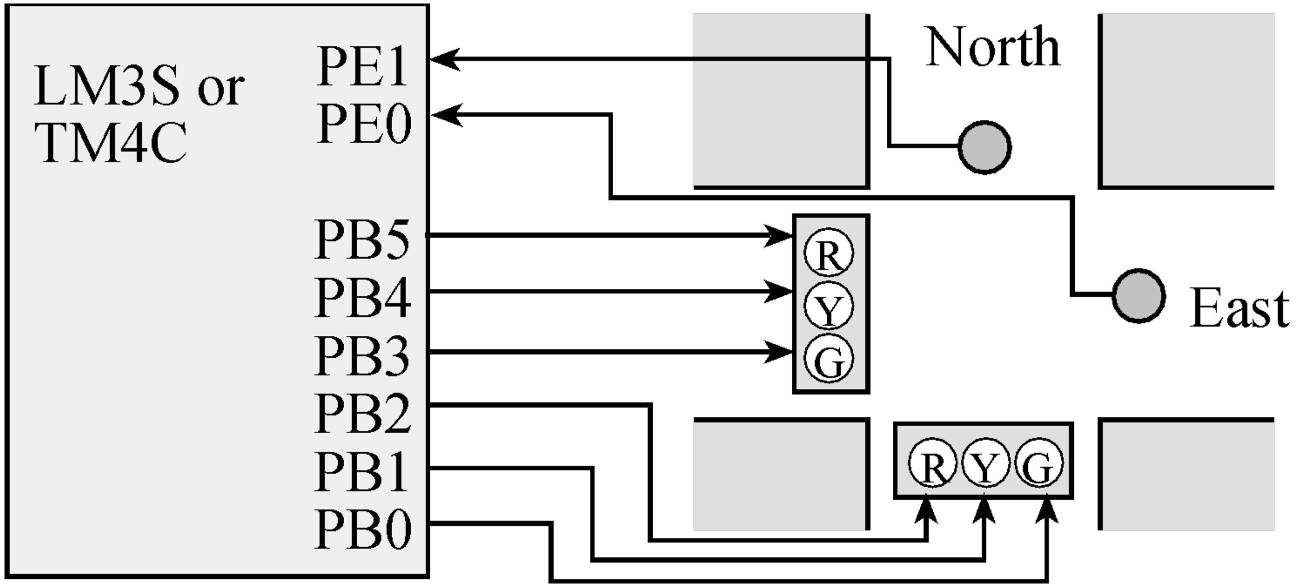


FSM Implementation

- **Data Structure** embodies the FSM
  - multiple identically-structured nodes
  - statically-allocated fixed-size linked structures
  - one-to-one mapping FSM state graph and linked structure
  - one structure for each state
- **Linked Structure**
  - pointer (or link) to other nodes (define next states)
- **Table structure**
  - indices to other nodes (define next states)

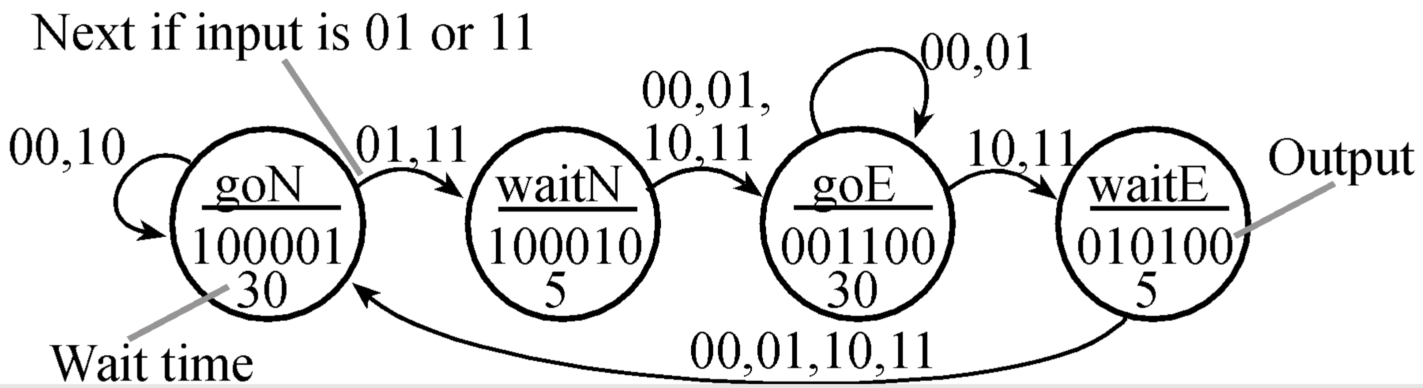
Traffic Light Control

PE1=0, PE0=0 means no cars exist on either road
PE1=0, PE0=1 means there are cars on the East road
PE1=1, PE0=0 means there are cars on the North road
PE1=1, PE0=1 means there are cars on both roads



goN,	PB5-0 = 100001 makes it green on North and red on East
waitN,	PB5-0 = 100010 makes it yellow on North and red on East
goE,	PB5-0 = 001100 makes it red on North and green on East
waitE,	PB5-0 = 010100 makes it red on North and yellow on East

Traffic Light Control



State \ Input	00	01	10	11
goN (100001,30)	goN	waitN	goN	waitN
waitN (100010,5)	goE	goE	goE	goE
goE (001100,30)	goE	goE	waitE	waitE
waitE (010100,5)	goN	goN	goN	goN

FSM Data Structure in C (Indexes)

```
const struct State {
  uint32_t Out;
  uint32_t Time;  // 10 ms units
  uint32_t Next[4]; // list of next states
};
typedef const struct State STyp;

#define goN   0
#define waitN 1
#define goE   2
#define waitE 3
STyp FSM[4] = {
 {0x21,3000,{goN,waitN,goN,waitN}},
 {0x22, 500,{goE,goE,goE,goE}},
 {0x0C,3000,{goE,goE,waitE,waitE}},
 {0x14, 500,{goN,goN,goN,goN}}
};
```

FSM Engine in C (Indexes)

```
void main(void) {
  uint32 CS;  // index of current state
  uint32_t Input;

  // initialize ports and timer
  …

  CS = goN; // start state
  while(1) {
    LIGHT = FSM[CS].Out;  // set lights
    SysTick_Wait10ms(FSM[CS].Time);
    Input = SENSOR;  // read sensors
    CS = FSM[CS].Next[Input];
  }
}
```

FSM Data Structure in C (Pointers)

```
const struct State {
  uint32_t Out;
  uint32_t Time;  // 10 ms units
  const struct State *Next[4];
};
typedef const struct State STyp;

#define goN   &FSM[0]
#define waitN &FSM[1]
#define goE   &FSM[2]
#define waitE &FSM[3]
STyp FSM[4] = {
 {0x21,3000,{goN,waitN,goN,waitN}},
 {0x22, 500,{goE,goE,goE,goE}},
 {0x0C,3000,{goE,goE,waitE,waitE}},
 {0x14, 500,{goN,goN,goN,goN}}
};
```

FSM Engine in C (Pointers)

```
void main(void) {
  STyp *Pt;  // state pointer
  uint32_t Input;

  // initialize ports and timer
  …

  Pt = goN; // start state
  while(1) {
    LIGHT = Pt->Out;  // set lights
    SysTick_Wait10ms(Pt->Time);
    Input = SENSOR;  // read sensors
    Pt = Pt->Next[Input];
  }
}
```

Linked Data Structure

```
OUT   EQU 0    ;offset for output
WAIT  EQU 4    ;offset for time (10ms)
NEXT  EQU 8    ;offset for next
goN   DCD 0x21 ;North green, East red
      DCD 3000 ;30 sec
      DCD goN,waitN,goN,waitN
waitN DCD 0x22 ;North yellow, East red
      DCD 500  ;5 sec
      DCD goE,goE,goE,goE
goE   DCD 0x0C ;North red, East green
      DCD 3000 ;30 sec
      DCD goE,goE,waitE,waitE
waitE DCD 0x14 ;North red, East yellow
      DCD 500  ;5 sec
      DCD goN,goN,goN,goN
```

In ROM

FSM Engine (Moore)

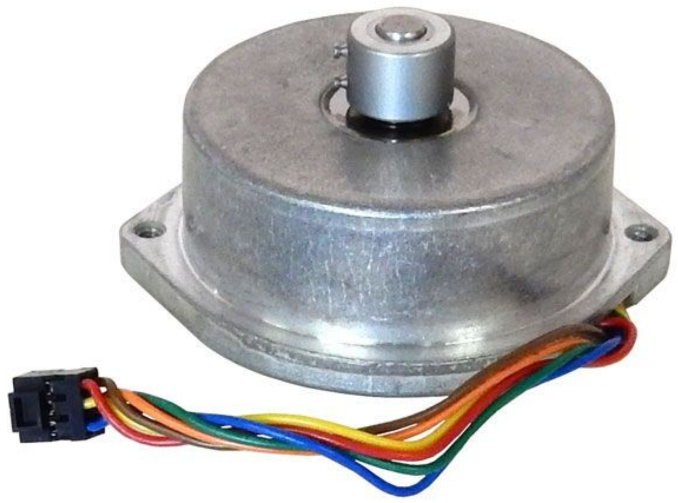
```
    ; Port and timer initialization
    LDR R4,=goN          ; state pointer
    LDR R5,=SENSOR       ; PortE
    LDR R6,=LIGHT        ; PortB
FSM LDR R0,[R4,#OUT]     ; 1.output value
    STR R0,[R6]          ;   set lights
    LDR R0,[R4,#WAIT]    ; 2. time delay
    BL  SysTick_Wait10ms
    LDR R0,[R5]          ; 3. read input
    LSL R0,R0,#2         ;    offset(index):
                         ;    4 bytes/address
    ADD R0,R0,#NEXT      ;    8,12,16,20
    LDR R4,[R4,R0]       ; 4. go to next state
    B   FSM
```

Thought Exercise: Implement Lab 3 as an FSM

- 1) The system starts with 20% duty-cyle
- 2) Turn the LED on/off at current duty-cycle
- 2) If the switch is pressed, then switch duty-cycle
  - Cycle: 40->60->80->100->0->20->40…
- 3) Steps 1 and 2 are repeated over and over

Stepper Motor Interface

Input=0	Stop
Input=1	Spin CW (1,2,4,8,16,…)
Input=2	Spin CCW (16,8,4,2,1,…)
Input=3	Stop



```
// Spin at constant speed
while(1){
  GPIO_PORTB_DATA_R=1;
  Wait1ms(T);
  GPIO_PORTB_DATA_R=2;
  Wait1ms(T);
  GPIO_PORTB_DATA_R=4;
  Wait1ms(T);
 GPIO_PORTB_DATA_R=8;
  Wait1ms(T);
 GPIO_PORTB_DATA_R=16;
  Wait1ms(T);
}
```

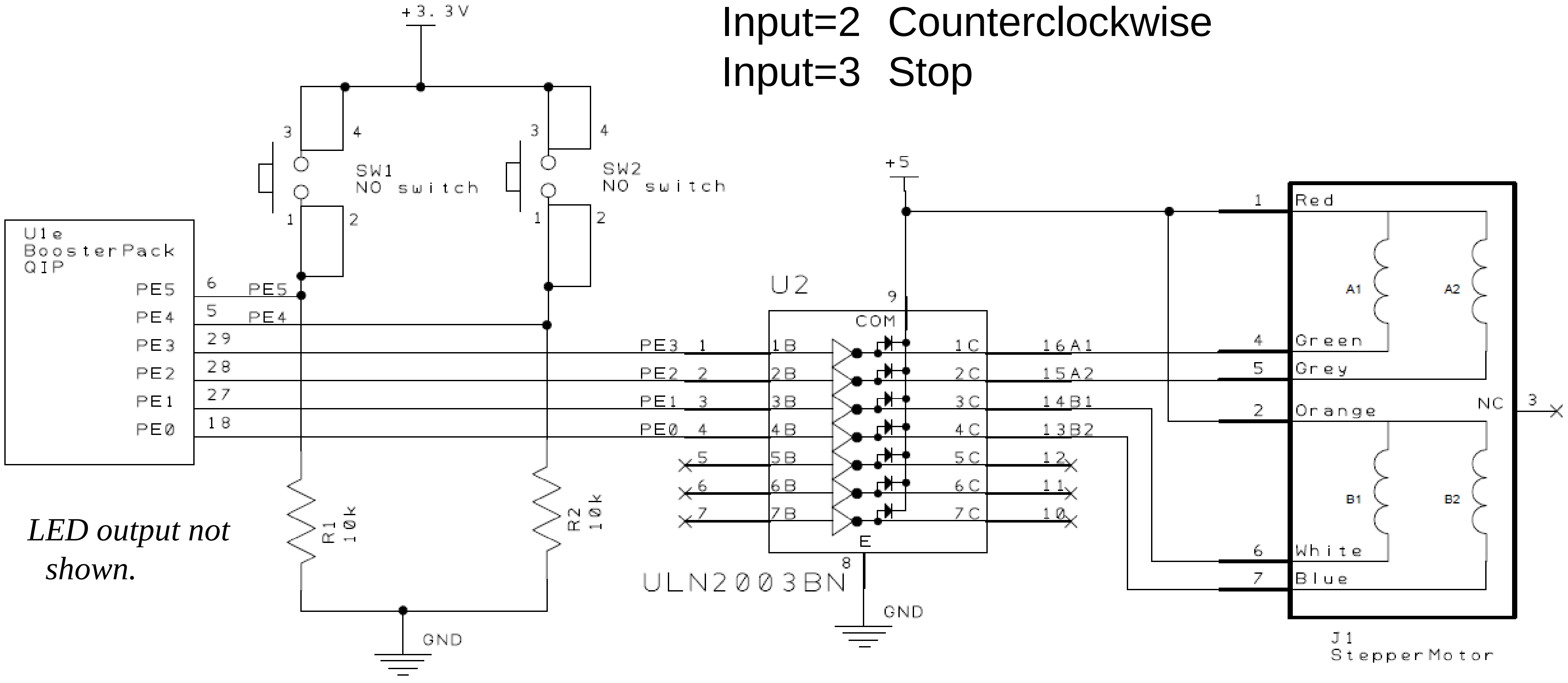
Each output is one step of 4 degrees
90 steps/rotation
T is in ms

Speed = 4(deg/step)*1000(ms/s)/T(ms)/360(deg/rot) = 11.11/T (rps)

Stepper Motor Interface

TAs recommend Port B

Input=0	Stop
Input=1	Clockwise
Input=2	Counterclockwise
Input=3	Stop

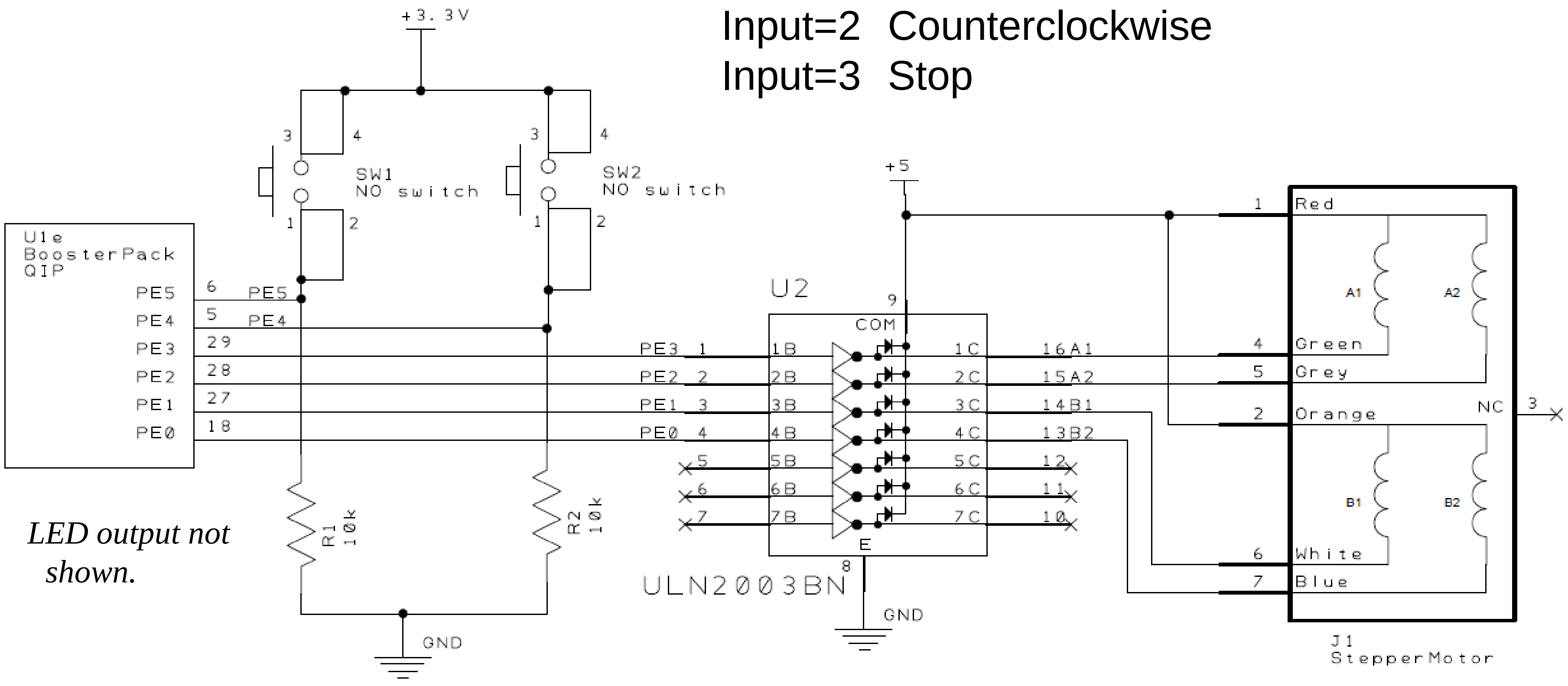


LED output not shown.

Stepper Motor Interface

TAs recommend Port B

Input=0	Stop
Input=1	Clockwise
Input=2	Counterclockwise
Input=3	Stop

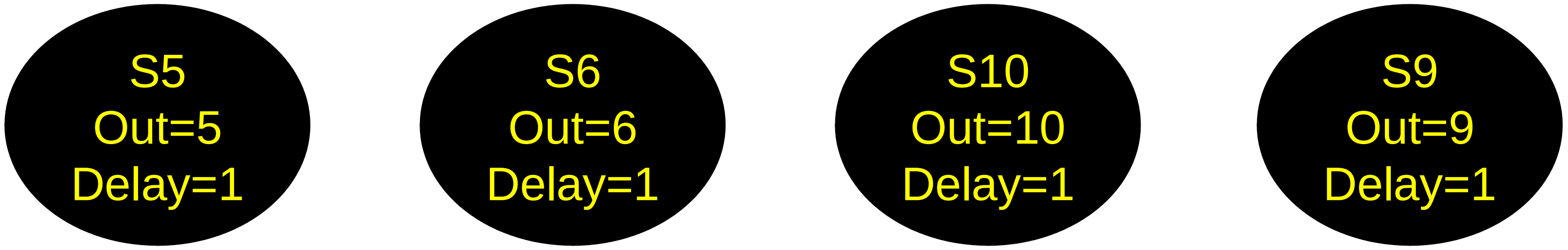


LED output not shown.

Clockwise 5,6,10,9,5,6,10,9,5,6,10,9,5,6,10,9,5,6,10,9,5,6,10,9,5,6,10,9,…
Counterclockwise 9,10,6,5,9,10,6,5,9,10,6,5,9,10,6,5,9,10,6,5,9,10,6,5,9,10,6,5, …
36 steps/revolution means each step changes angle by 10 degrees

Stepper Motor Example

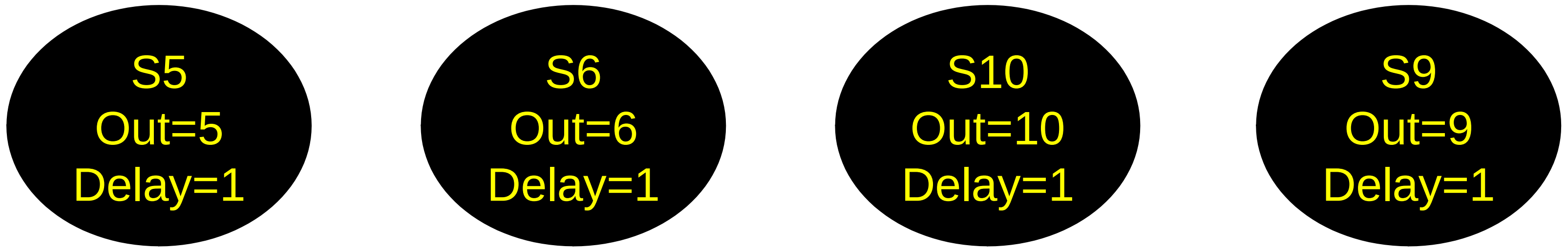
Input=0	Stop
Input=1	Clockwise
Input=2	Counterclockwise
Input=3	Stop



1. Output (depends on state)
2. Wait
3. Input
4. Next (depends on state and input)

Stepper Motor Example

Input=0	Stop
Input=1	Clockwise
Input=2	Counterclockwise
Input=3	Stop

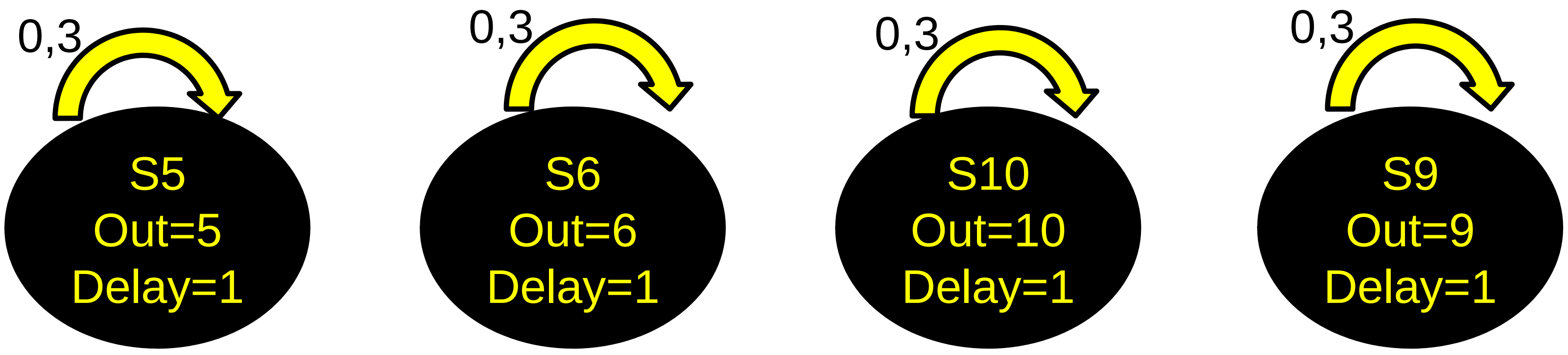


```
#define T1sec 100
struct State{
  uint32_t Out;      // output to PortB

  uint32_t Delay;    // time in 10ms
  uint32_t Next[4]; // for each input
};
typedef const struct State State_t;
```

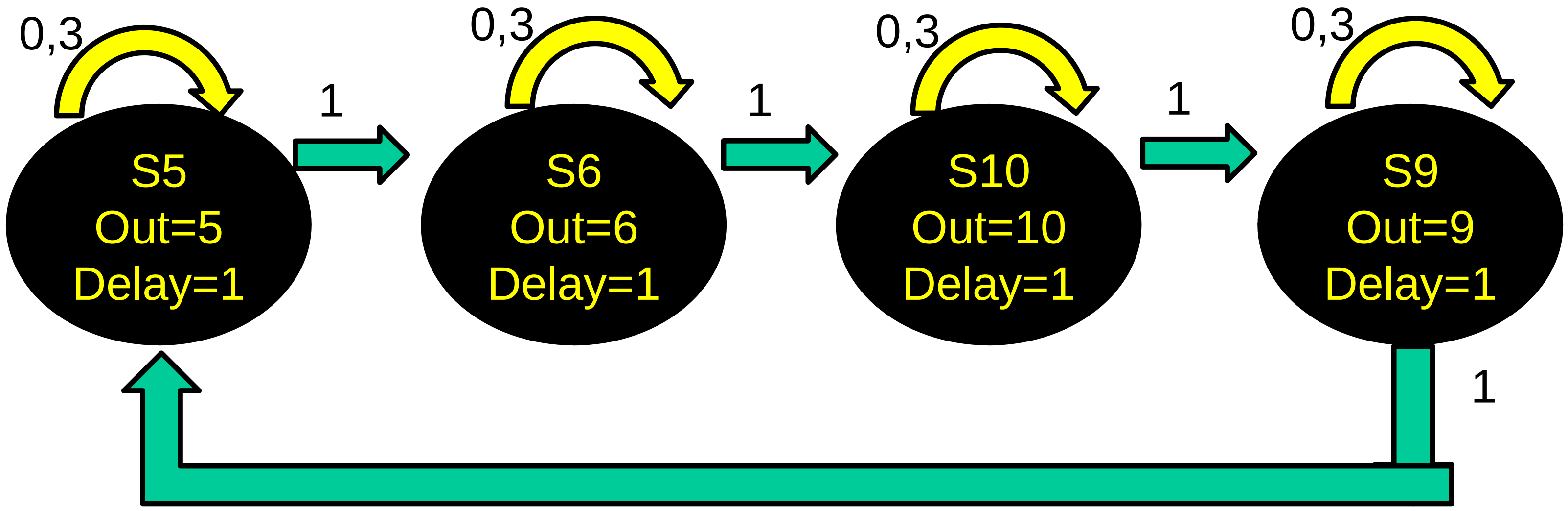
Stepper Motor Example

Input=0	**Stop**
Input=1	Clockwise
Input=2	Counterclockwise
Input=3	**Stop**



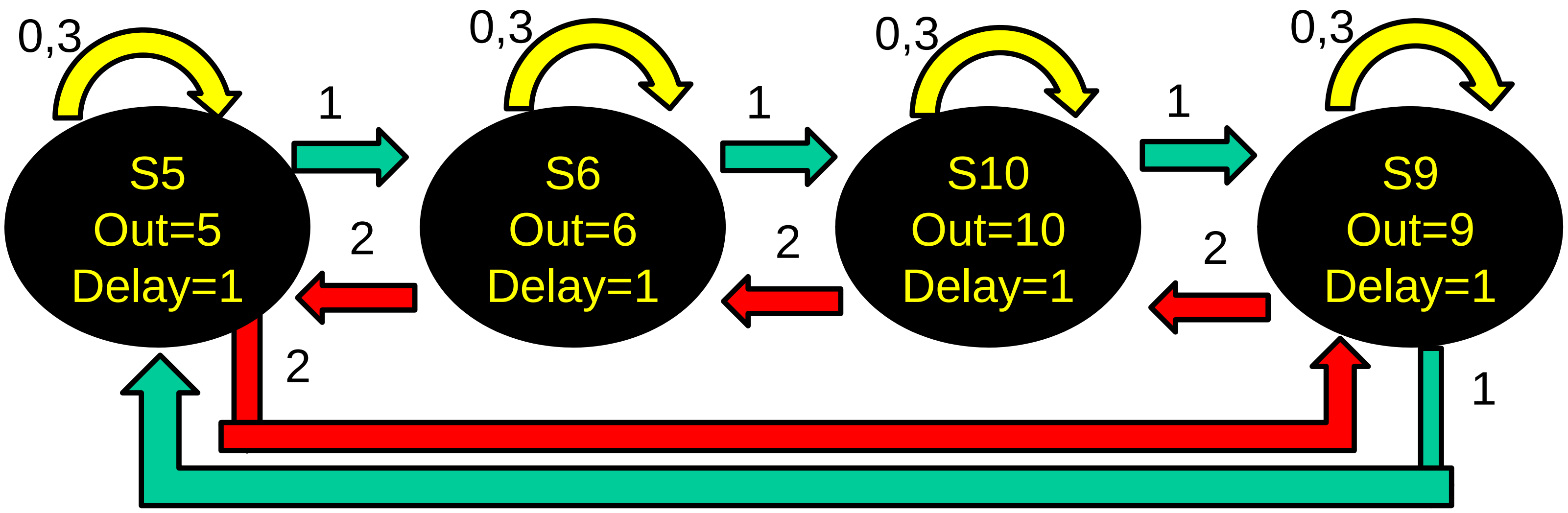
Stepper Motor Example

Input=0 Stop
Input=1 Clockwise
Input=2 Counterclockwise
Input=3 Stop

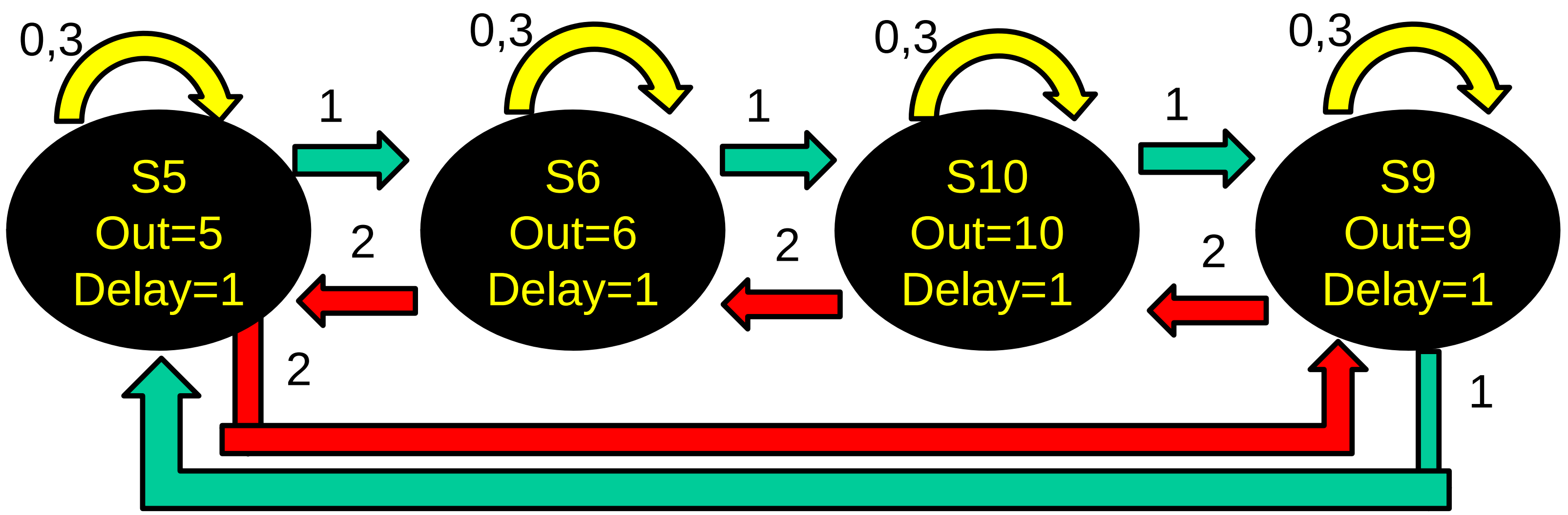


Stepper Motor Example

Input=0 Stop
Input=1 Clockwise
Input=2 Counterclockwise
Input=3 Stop



Stepper Motor Example



```
#define S5  0
#define S6  1
#define S10 2
#define S9  3
```

```
StateType fsm[4]={
  {5, T1sec, S5, S6, S9, S5},
  {6, T1sec, S6,S10, S5, S6},
  {10,T1sec,S10, S9, S6,S10},
  {9, T1sec, S9, S5,S10, S9}
};
```

FSM Engine in C (Indexes)

```
uint32_t N,Input;
int main(void){  // Logic analyzer to Port E
  TExaS_Init(4); // bus to 80MHz
  SysTick_Init();
  Port_Init();
  N = 0;
  while(1){
    GPIO_PORTE_DATA_R = fsm[N].Out;  // Output
    SysTick_Wait10ms(fsm[N].Delay); // Wait
    Input = (GPIO_PORTE_DATA_R&0x60)>>4;
    N = fsm[N].Next[Input];        // Next
  }
}
```

Thought Exercise: Implement Lab 3 as an FSM

- 1) The system starts with 20% duty-cyle
- 2) Turn the LED on/off at current duty-cycle
- 2) If the switch is pressed, then switch duty-cycle
  - Cycle: 40->60->80->100->0->20->40...
- 3) Steps 1 and 2 are repeated over and over

FSM Summary

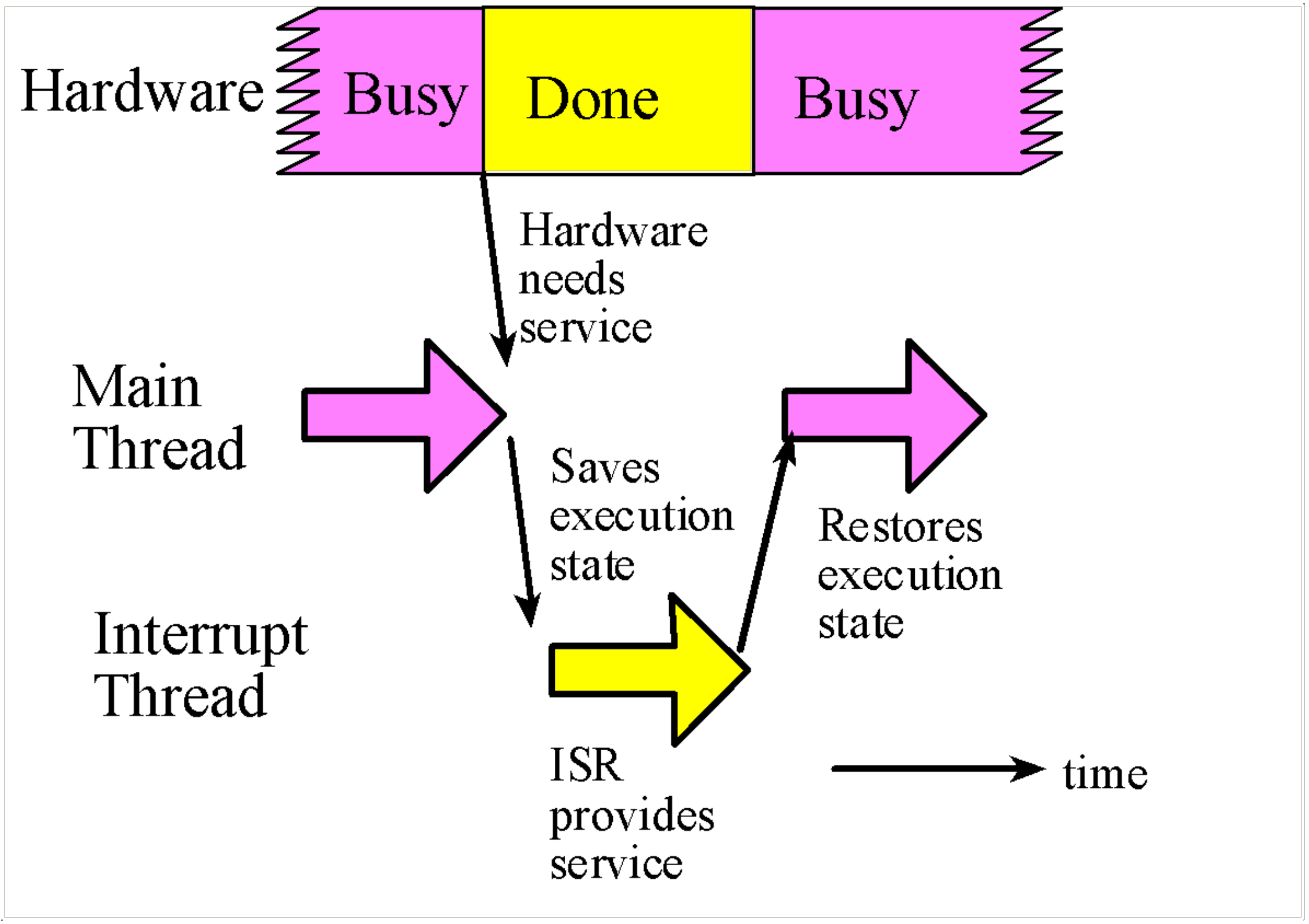
- Run Stepper_5phase_FSM_4C123
- Run Lab 5 solution

- ❑ Abstraction separates
  - ❖ What it does (STG)
  - ❖ How it works (C code)
- ❑ Finite State Machines
  - ❖ States define what you know/believe
  - ❖ 1-1 mapping STG ⇔ STT ⇔ C structure
  - ❖ No conditional statements

Interrupts

- An **interrupt** is the automatic transfer of software execution in response to a hardware event (trigger) that is asynchronous with current software execution.
  - external I/O device (like a keyboard or printer) or
  - an internal event (like an op code fault, or a periodic timer.)
- Occurs when the hardware needs or can service (busy to done state transition)

Interrupt Processing



ARM Cortex-M Interrupts

- Each potential interrupt source has a separate **arm** bit
  - Set for those devices from which it wishes to accept interrupts,
  - Deactivate in those devices from which interrupts are not allowed
- Each potential interrupt source has a separate **flag** bit
  - hardware sets the flag when it wishes to request an interrupt
  - software clears the flag in ISR to signify it is processing the request
- Interrupt **enable** conditions in processor
  - Global interrupt enable bit, I, in PRIMASK register
  - Priority level, BASEPRI, of allowed interrupts (0 = all)

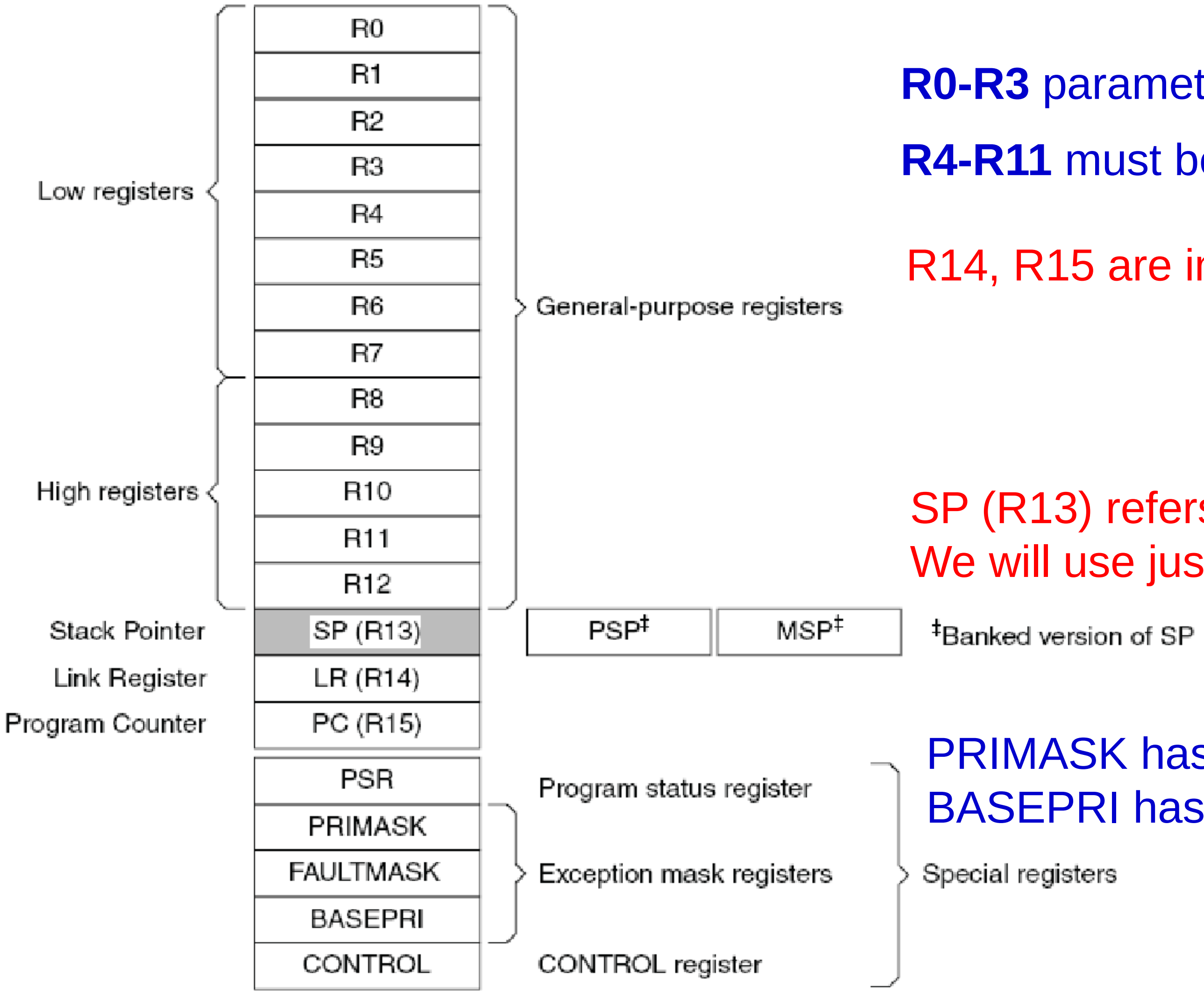
Interrupt Conditions

- Five conditions must be true simultaneously for an interrupt to occur:
  1. Arm: control bit for each possible source is set
  2. Enable bit of the device in NVIC must be set
  3. Enable: interrupts globally enabled (I=0 in PRIMASK)
  4. Level: interrupt level must be less than BASEPRI
  5. Trigger: hardware action sets source-specific flag
- Interrupt remains **pending** if trigger is set but any other condition is not true
  - Interrupt serviced once all conditions become true
- Need to **acknowledge** interrupt
  - Clear trigger flag or will get endless interrupts!

Interrupt Processing

1. The execution of the main program is suspended
   1. the current instruction is finished,
   2. suspend execution and push 8 registers (R0-R3, R12, LR, PC, PSR) on the stack
   3. LR set to 0xFFFFFFF9 (indicates interrupt return)
   4. IPSR set to interrupt number
   5. sets PC to ISR address
2. The interrupt service routine (ISR) is executed
   - ❖ clears the flag that requested the interrupt
   - ❖ performs necessary operations
   - ❖ communicates using global variables
3. The main program is resumed when ISR executes **BX LR**
   - ❖ pulls the 8 registers from the stack

Registers



R0-R3 parameters

R4-R11 must be saved

R14, R15 are important

SP (R13) refers to PSP or MSP
We will use just the MSP

PRIMASK has intr. enable (I) bit
BASEPRI has allowed intr. priority

Priority Mask Register

Priority mask register

The PRIMASK register prevents activation of all exceptions with configurable priority. See the register summary in *Table 2 on page 13* for its attributes. *Figure 5* shows the bit assignments.

Figure 5. PRIMASK bit assignments

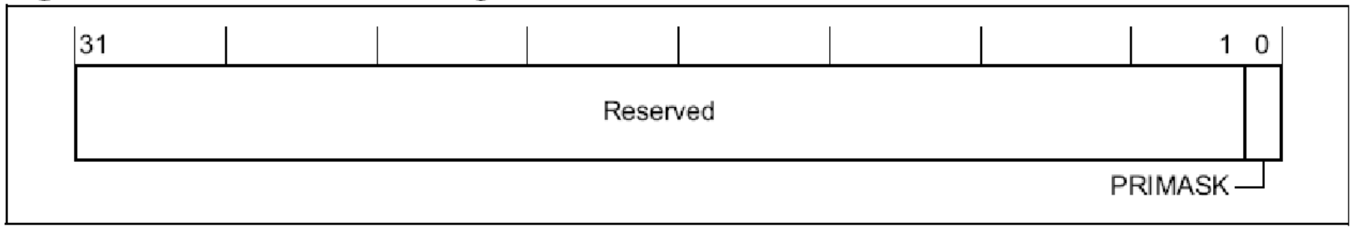


Table 7. PRIMASK register bit definitions

Bits	Description
Bits 31:1	Reserved
Bit 0	**PRIMASK:** 0: No effect 1: Prevents the activation of all exceptions with configurable priority.

Disable interrupts (I=1)

```
CPSID I
```

Enable interrupts (I=0)

```
CPSIE I
```

Program Status Register

- Accessed separately or all at once

Figure 3. APSR, IPSR and EPSR bit assignments

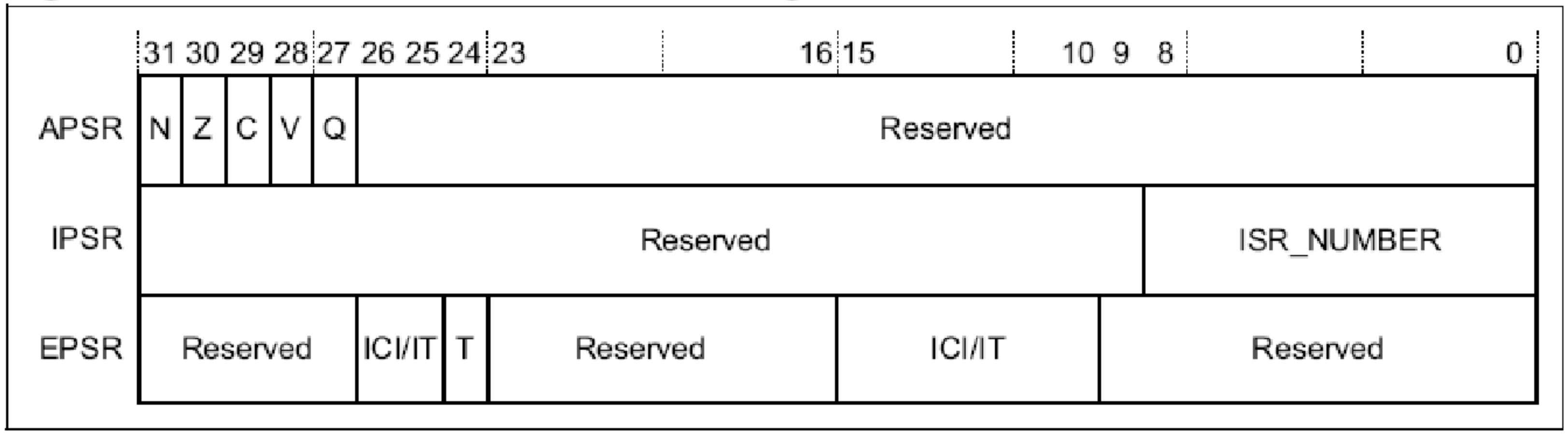
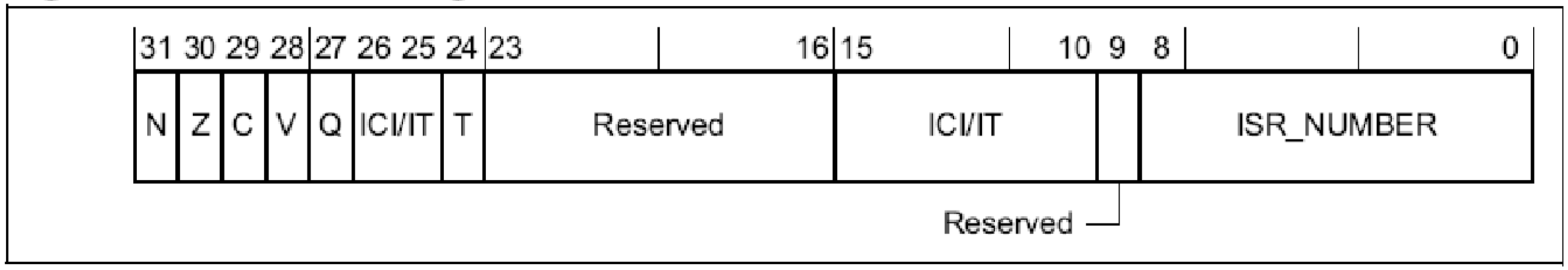


Figure 4. PSR bit assignments



Q = Saturation, T = Thumb bit

Interrupt Program Status Register (IPSR)

Bits	Description
Bits 31:9	Reserved
Bits 8:0	**ISR_NUMBER**: This is the number of the current exception: 0: Thread mode 1: Reserved 2: NMI 3: Hard fault 4: Memory management fault 5: Bus fault 6: Usage fault 7: Reserved .... 10: Reserved 11: SVCall 12: Reserved for Debug 13: Reserved 14: PendSV 15: SysTick 16: IRQ0[1] ....

Run debugger:
- stop in ISR and
- look at IPSR

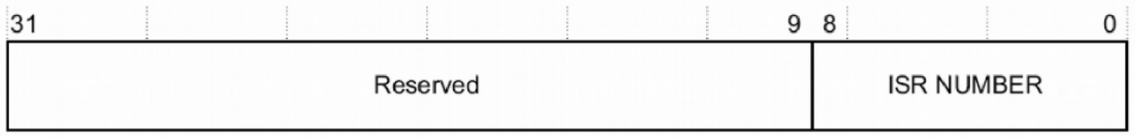
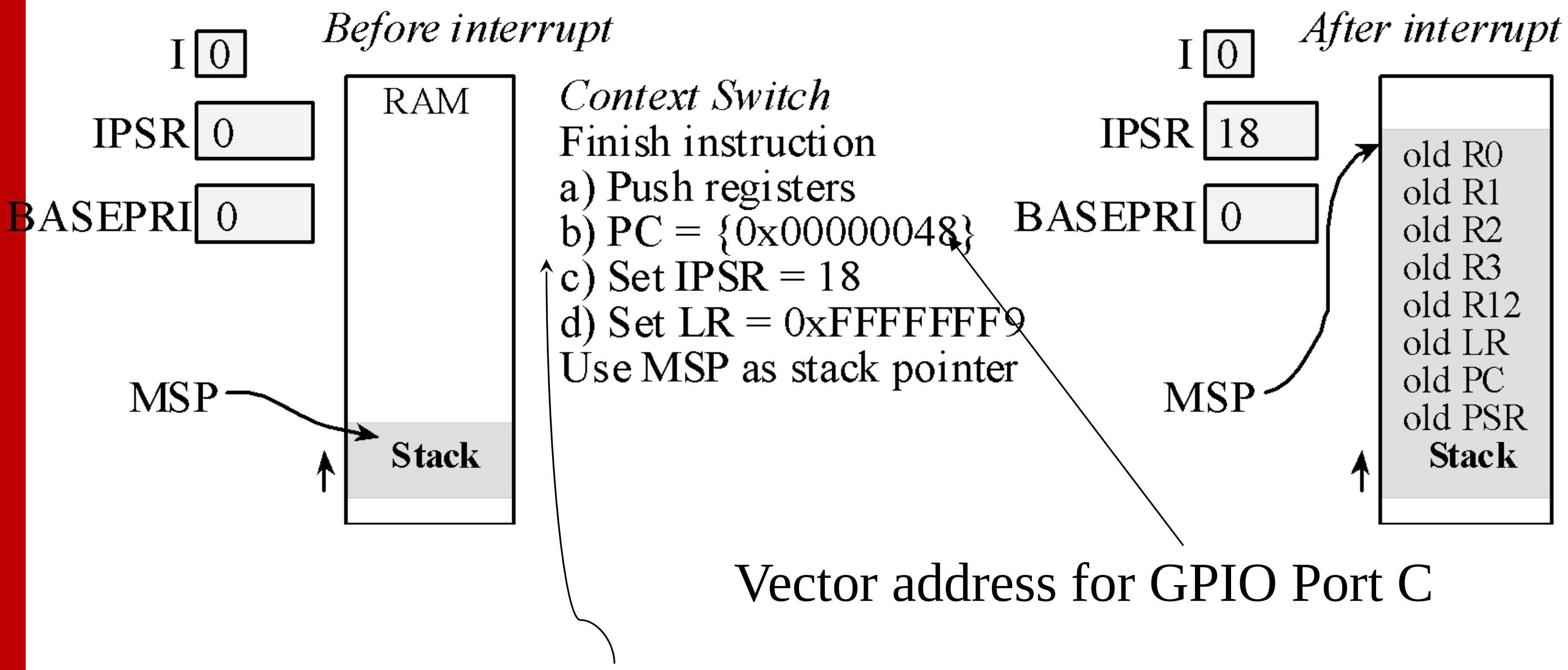


Figure 2-3, The IPSR Register.

Interrupt Context Switch



Interrupt Number 18 corresponds to GPIO Port C