

# HANDS-ON

# Training

## **103 ASP** **Getting Started with the** **16-bit Architecture,** **Instruction Set and** **Assembly Programming**





# Key Objectives

- Familiarity of 16-bit Architecture
- Knowledge of 16-bit Instruction Set
- Learning how to program using the 16-bit Assembly Language



# Session Agenda

- 16-bit Architecture Basics
- Some 16-bit Assembly Instructions
- I/O Port Handling
  - **Hands-on Lab #1 Follow along**
    - Basic Assembly language setup
    - Light LED on I/O Port
- Complete 16-bit Assembly Instructions
- Addressing Modes
  - **Hands-on Lab #2**
    - Use Loop Instructions to Blink LED
- Interrupts and Timers
  - **Hands-on Lab #3**
    - Use Interrupt to Blink LED
- Program Memory Access
  - **Optional Hands-on Lab #4**
    - Use PSV to transmit Serial String “Hello World”



# 16-bit Product Families Overview

**PIC24F**

**PIC24H**

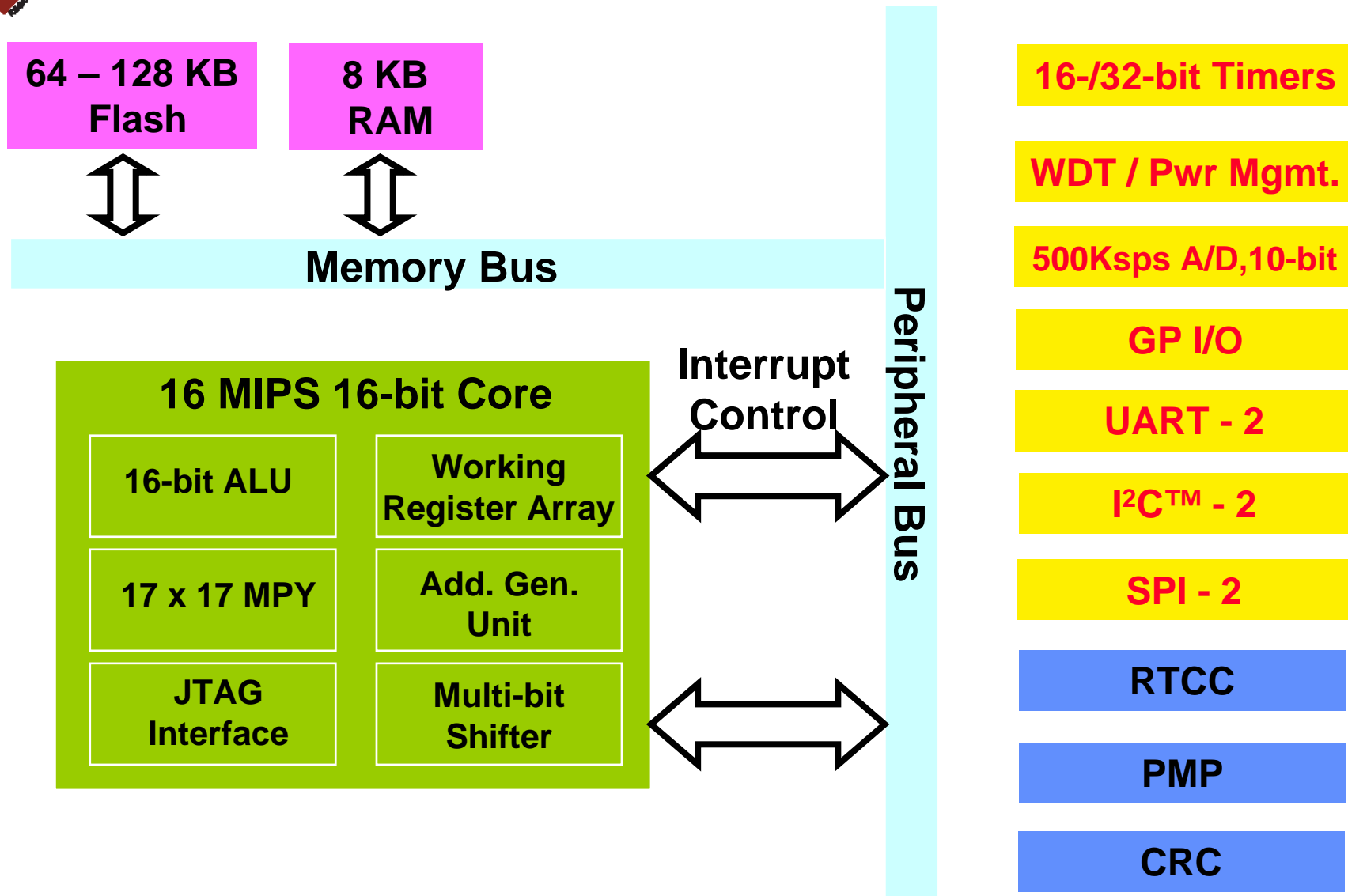
**dsPIC30F**

**dsPIC33F**

- ✓ Same Basic Architecture
- ✓ Same 24-bit Flash Program Memory
- ✓ Same 16-bit Data Memory
- ✓ Instruction Set optimized for C efficiency
- ✓ Same Deterministic Interrupt System
- ✓ Flexible system clock features
- ✓ DSP Performance when you need it
- ✓ Advanced Peripherals
- ✓ Easy Code Migration



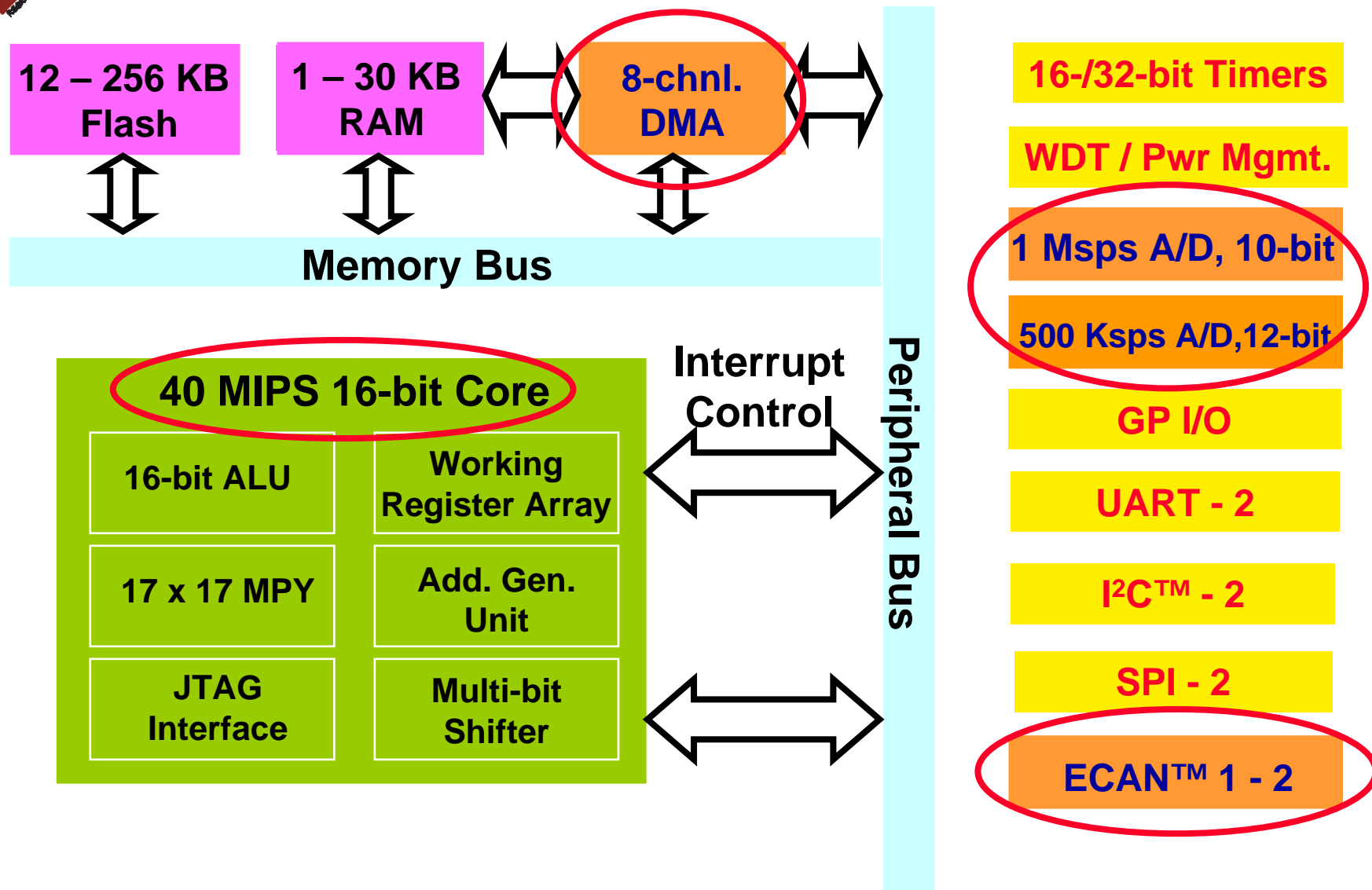
# PIC24F Family Block Diagram





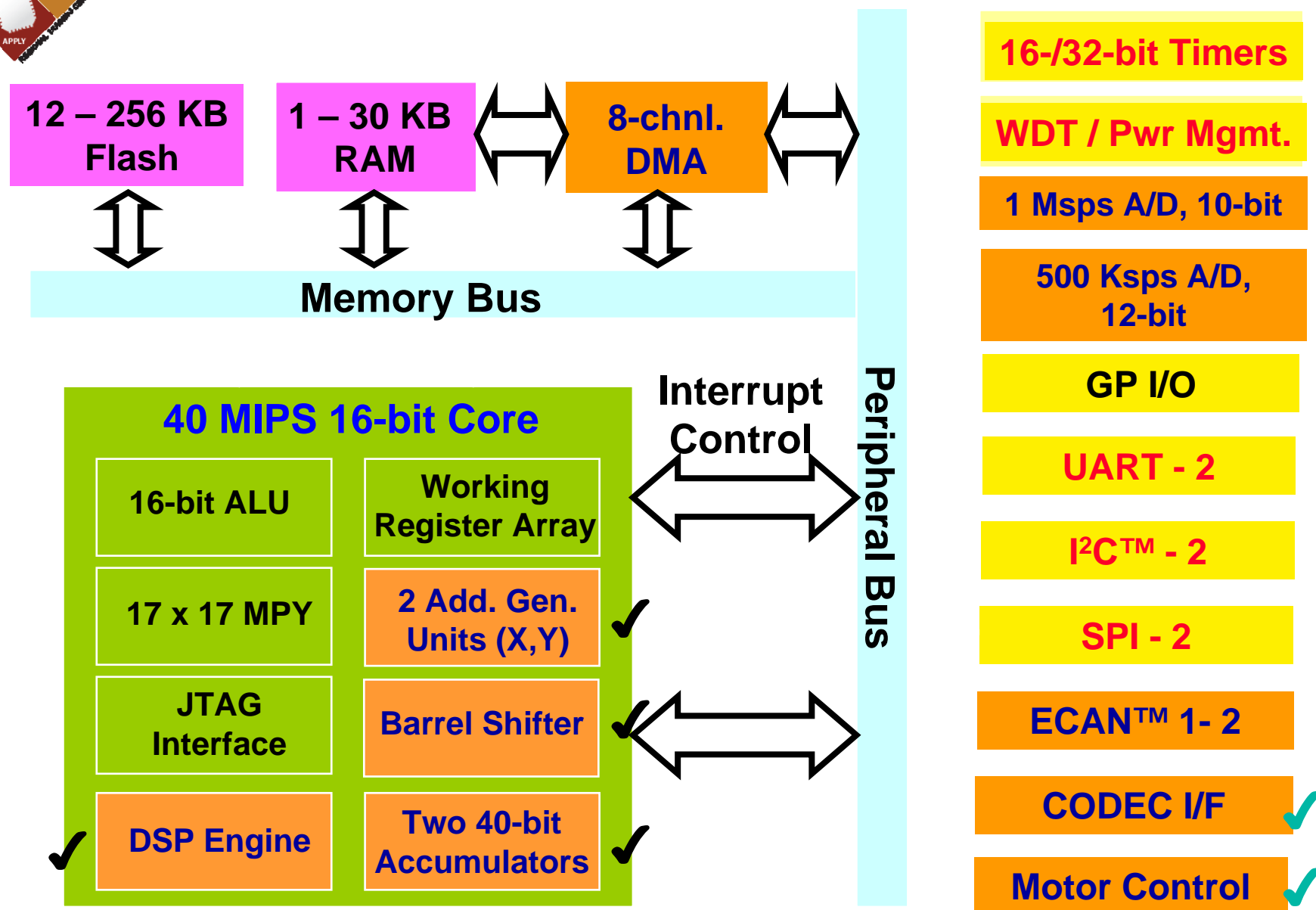


# PIC24H Family Block Diagram





# dsPIC30/33 Family Block Diagram



# HANDS-ON

# Training

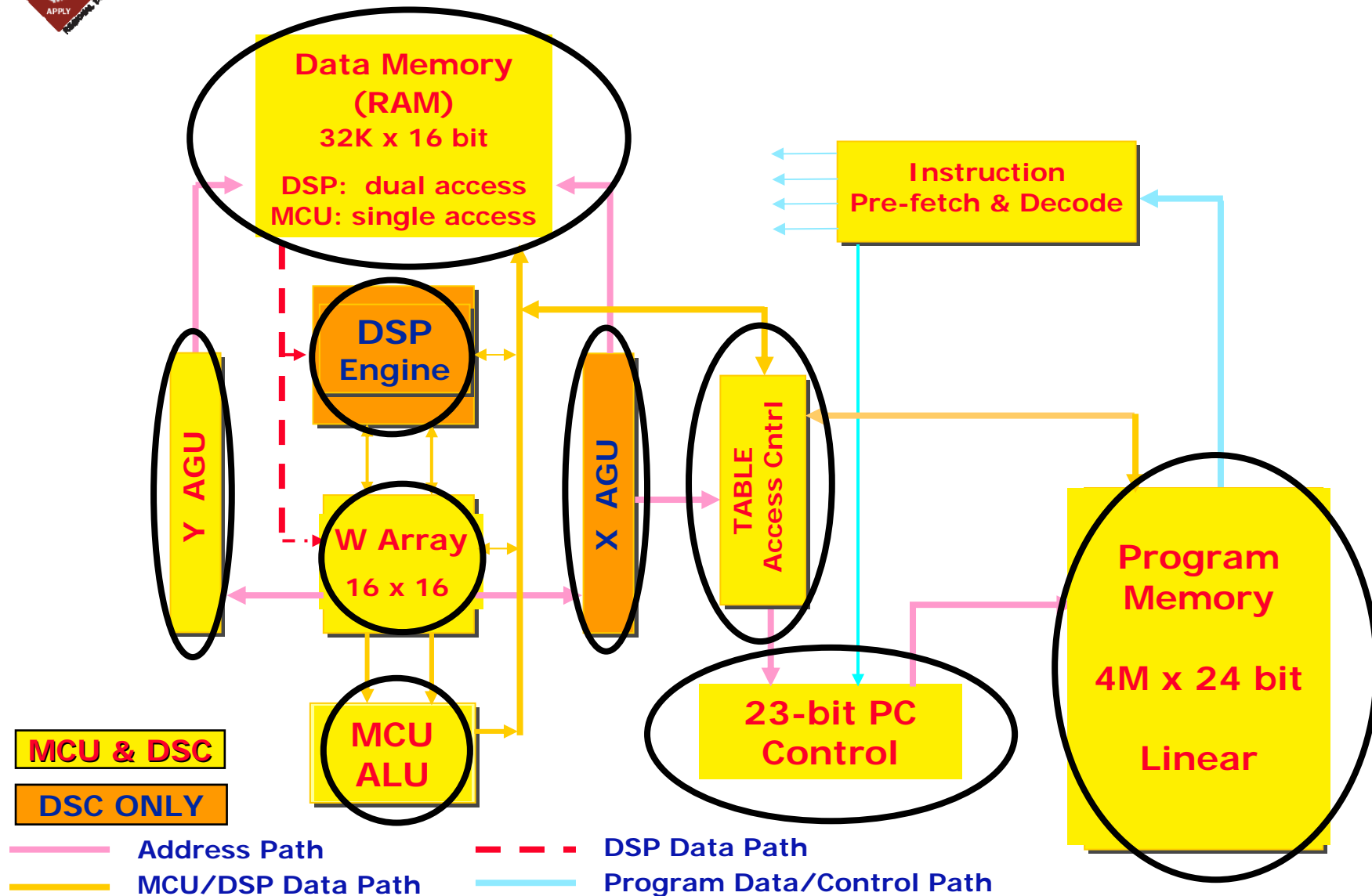
## 16-bit Architecture Basics





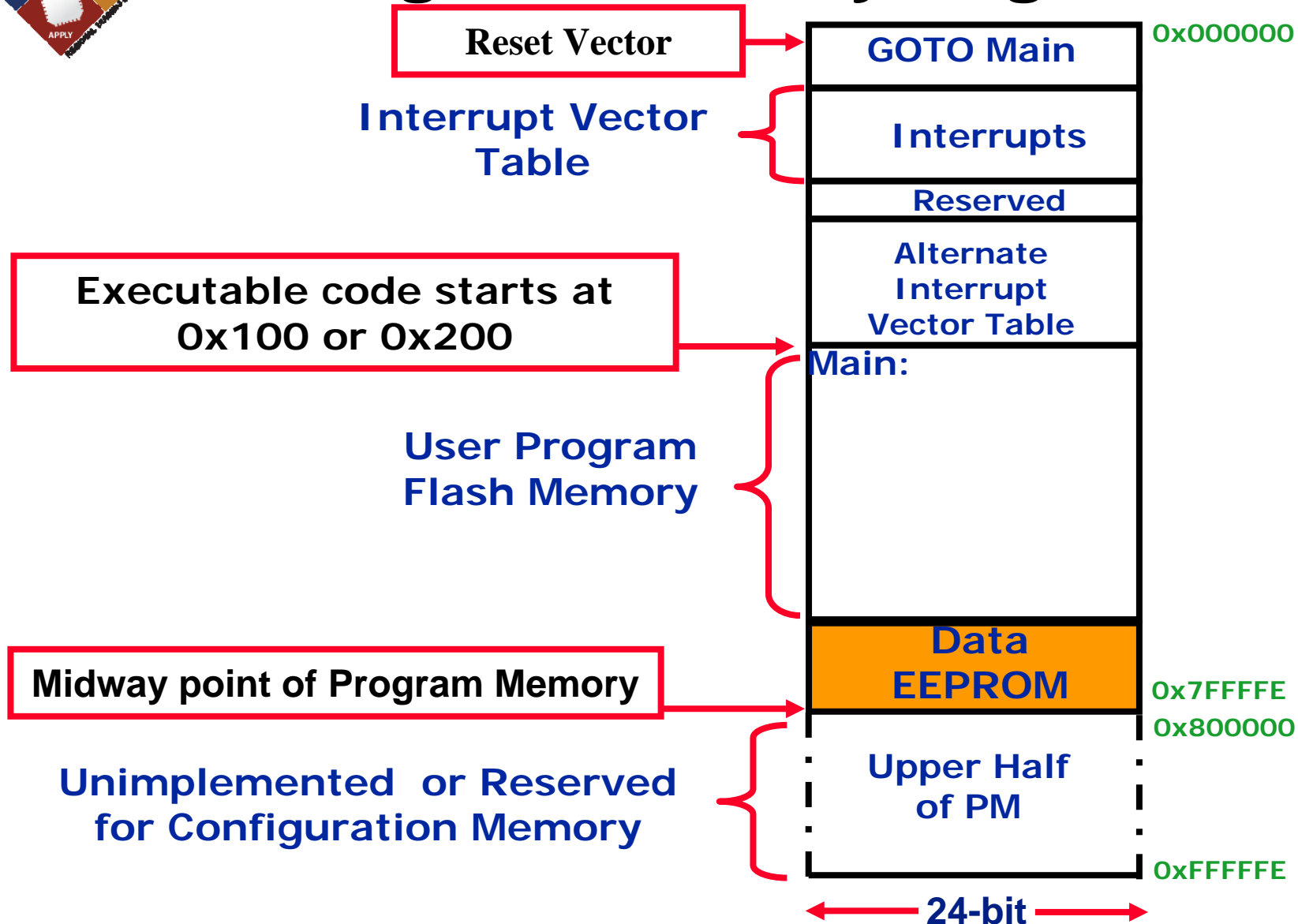


# Architecture Block Diagram



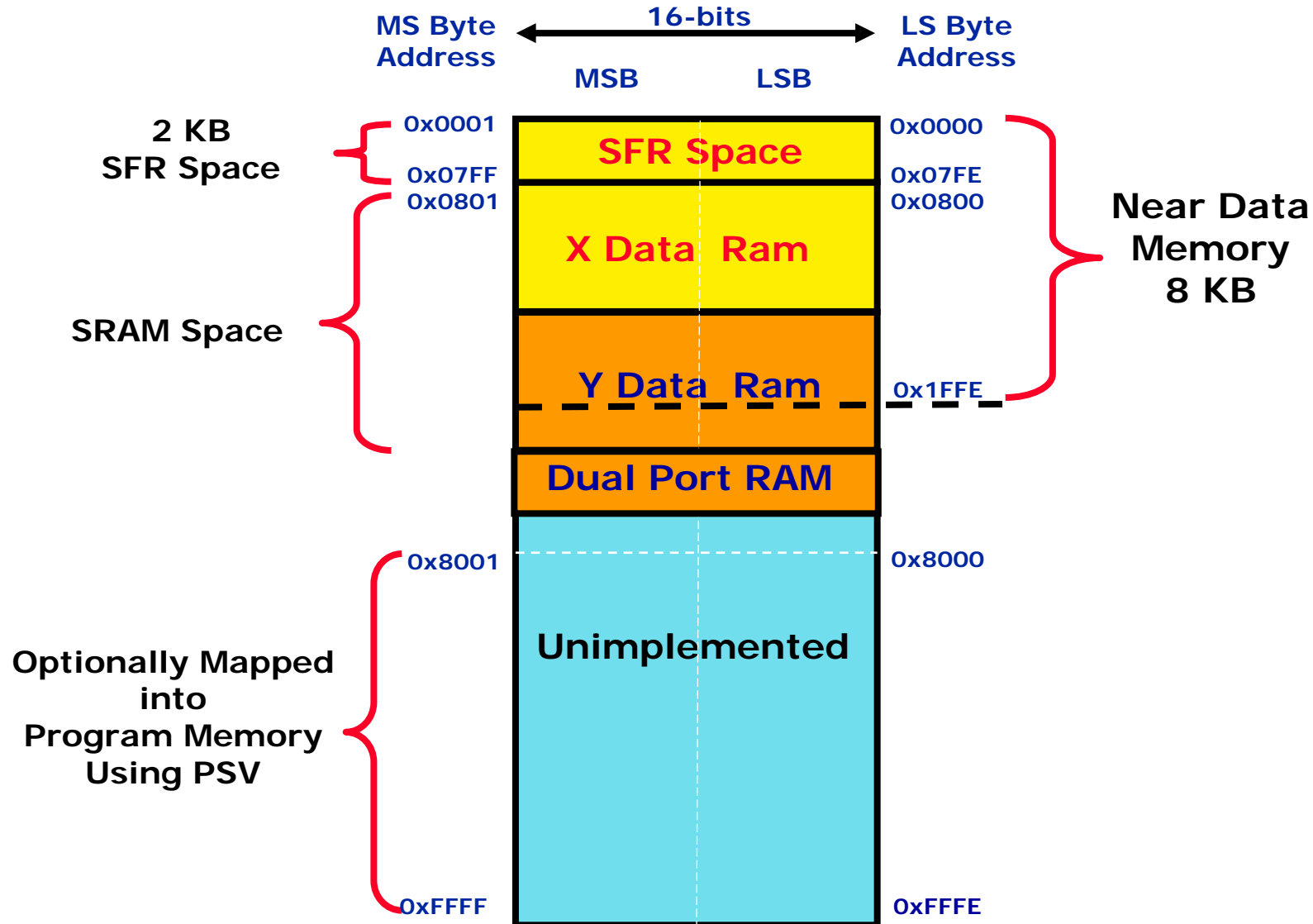


# Program Memory Organization





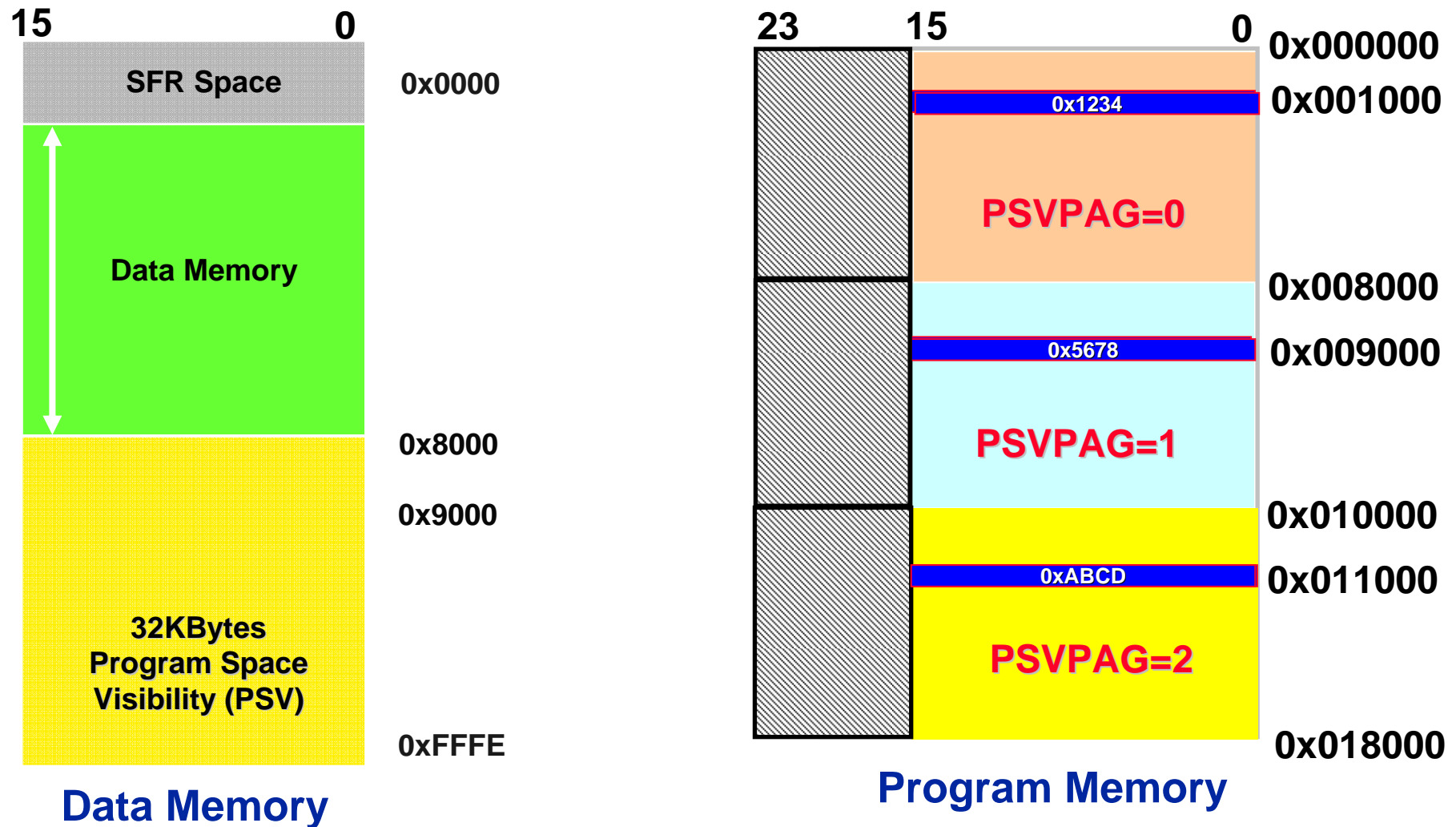
# Data Memory Organization\*





# Program Space Visibility

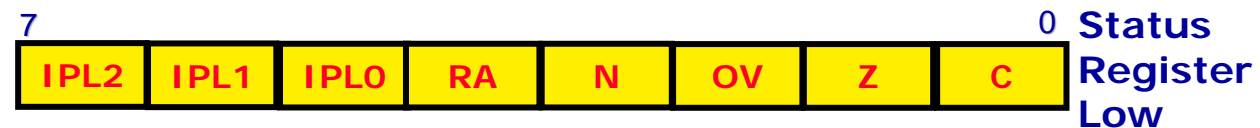
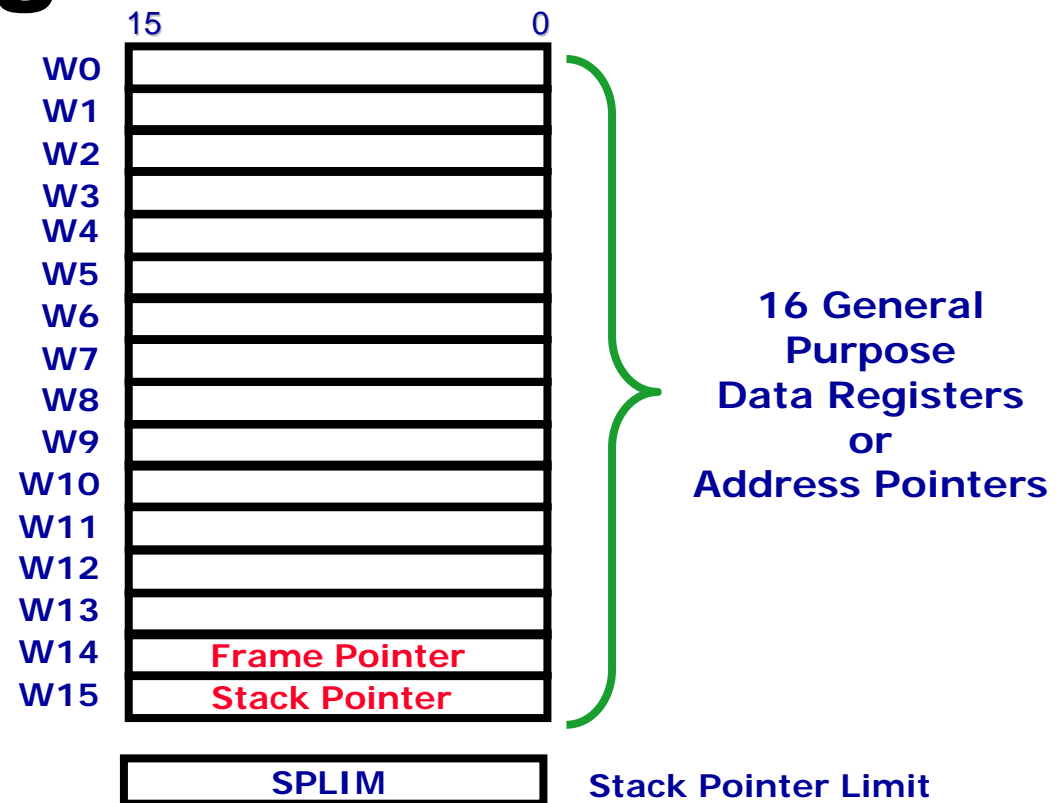
- Program Space Visibility Enabled when **CORCON<PSV> = 1**
- PSVPAG register maps 32Kb program memory segment into data memory



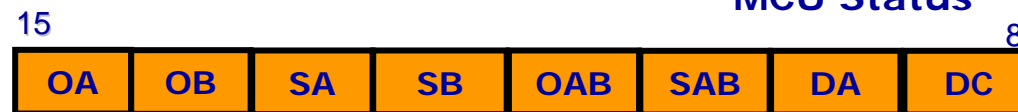




# Programmers Model

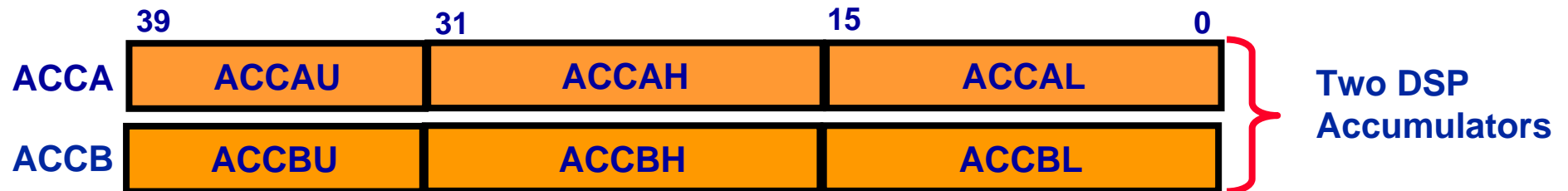
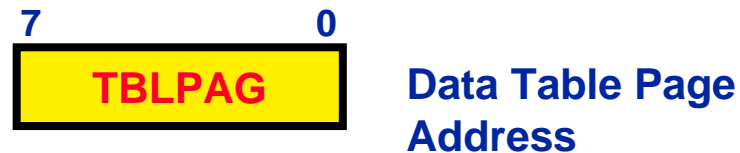


Status Register High





# Programmers Model (contd.)





# Programmers Model(contd.)

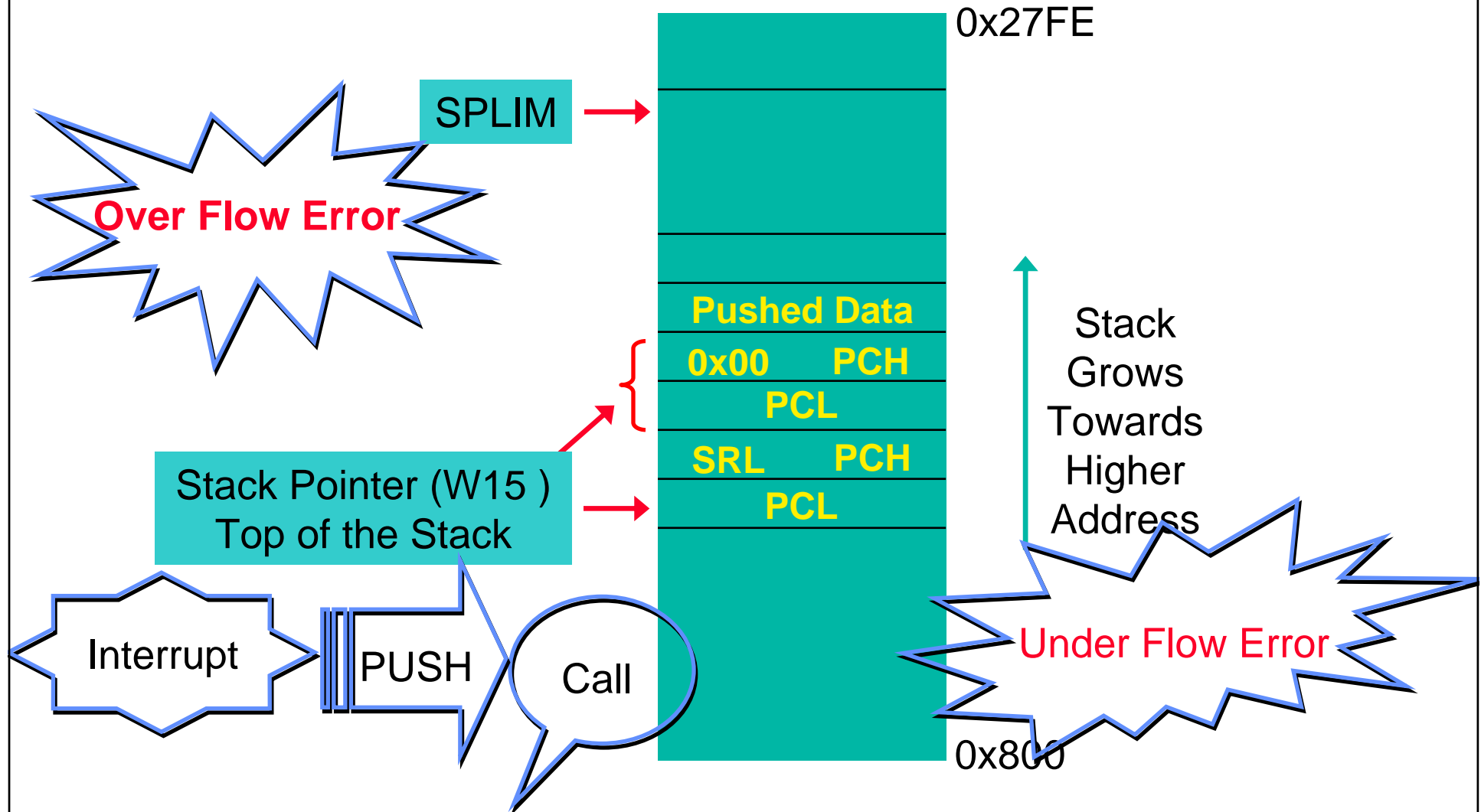


## Do Loop Registers:





# Software Stack in Data RAM

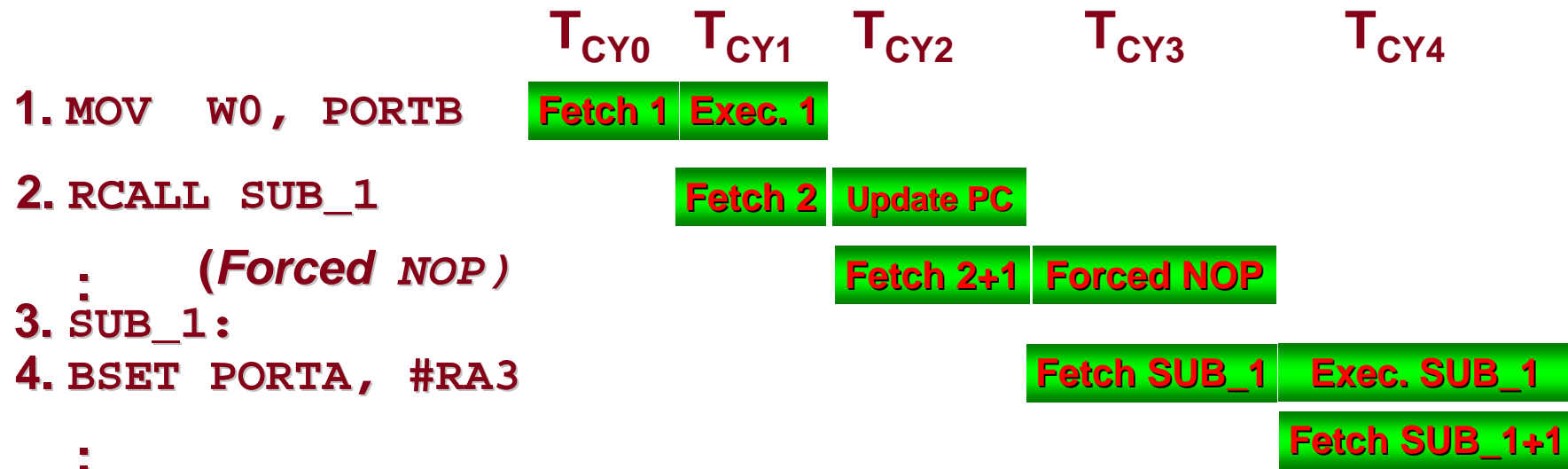






# Instruction Pipeline

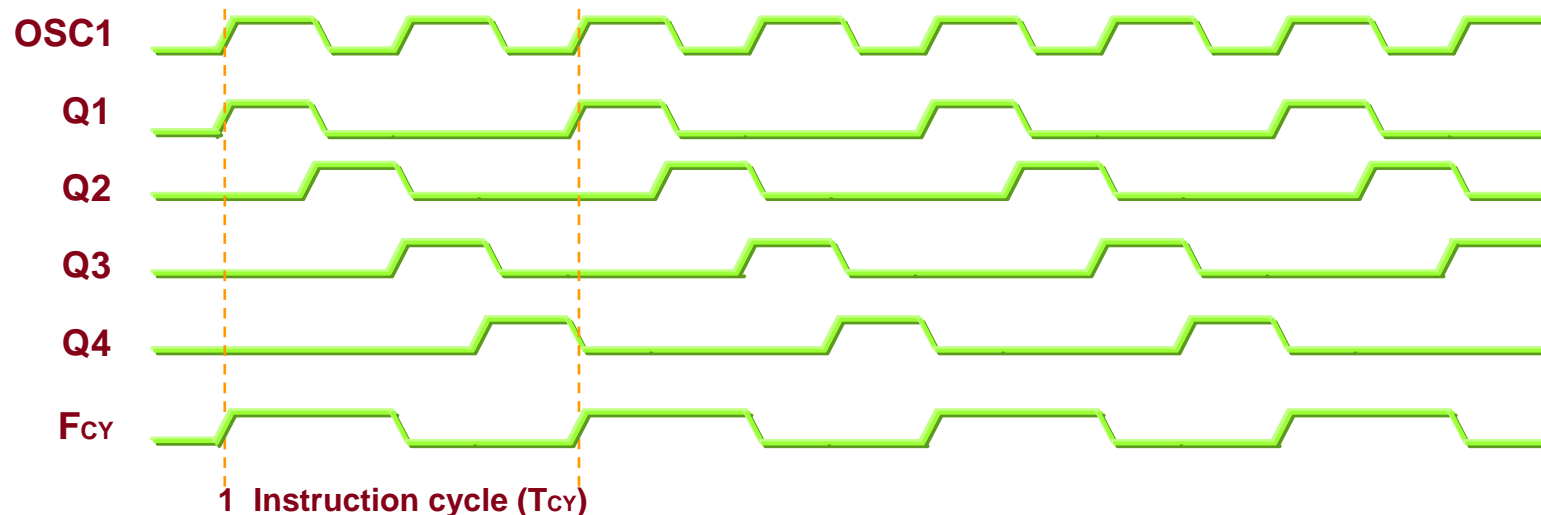
- Allows overlap of execution and fetch
- Makes single cycle execution
- Program branches (e.g. GOTO, CALL) take two cycles





# Clocking Scheme

- **Instruction cycle = 2 Input Clocks**
- **PIC24F**
  - A 32Mhz clk = 16MIPS or  $T_{cy} = 62.5 \text{ nS}$
- **PIC24H/dsPIC33**
  - A 80Mhz clk = 40 MIPS or  $T_{cy} = 25 \text{ nS}$
- **dsPIC30F – Instruction cycle = 4 Input Clocks**
  - A 120Mhz clk = 30MIPS or  $T_{cy} = 33 \text{ nS}$





# Session Agenda

- 16-bit Architecture Basics
- — Some 16-bit Assembly Instructions
- I/O Port Handling
  - **Hands-on Lab #1 Follow along**
    - Basic Assembly language setup
    - Light LED on I/O Port
- Complete 16-bit Assembly Instructions
  - **Hands-on Lab #2**
    - Use Loop Instructions to Blink LED
- Addressing Modes
- Interrupts and Timer1
  - **Hands-on Lab #3**
    - Use Interrupt to Blink LED



# Instruction Set Functional Groups

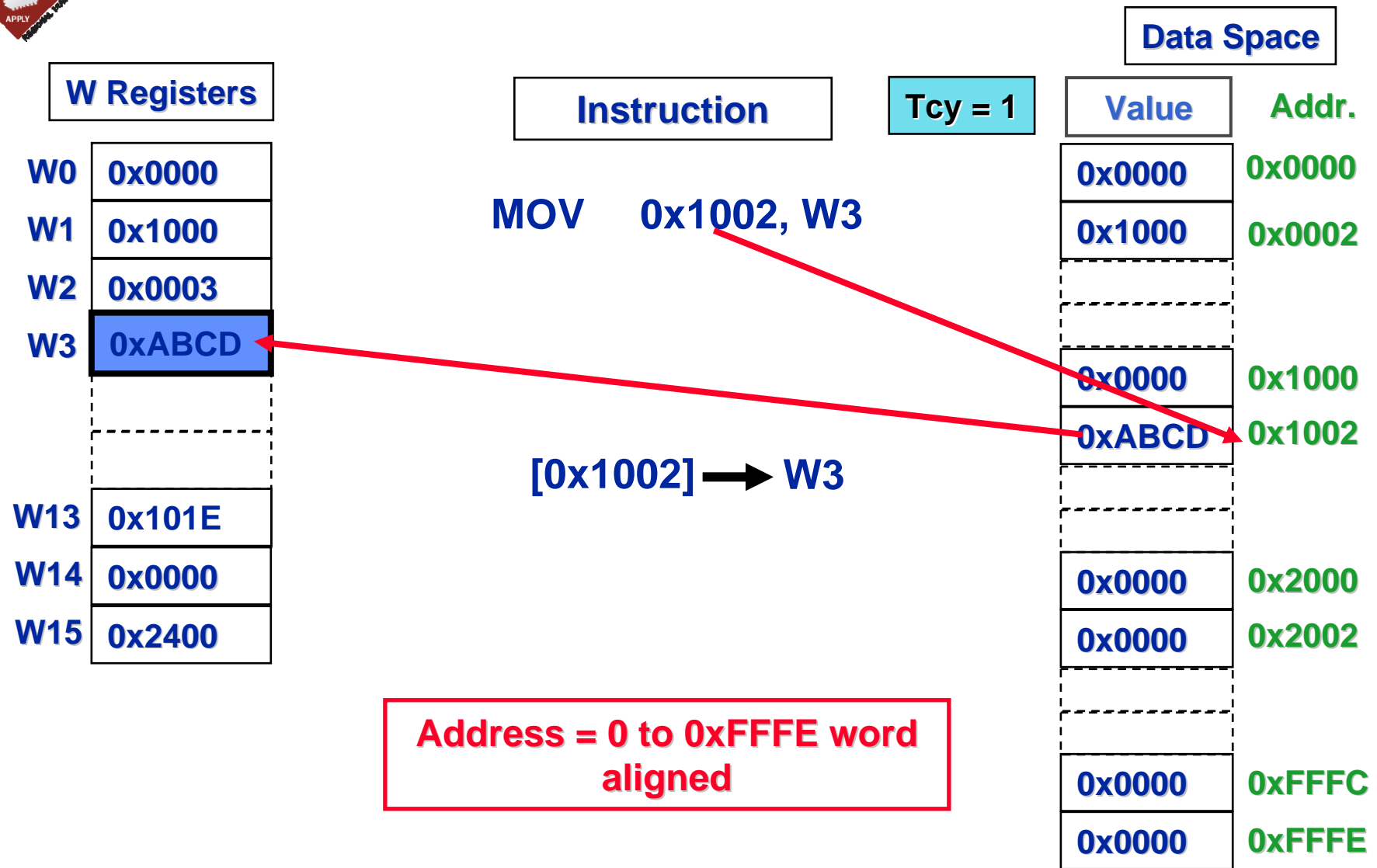
- **16-bit Instructions**

- — **Move Instructions**
- **Bit Instructions**
  - Follow along LAB 1
- **Math and Logic Instructions**
- **Stack Control Instructions**
- **Program Flow Control Instructions**
- **CPU Control Instructions**
  - LAB 2



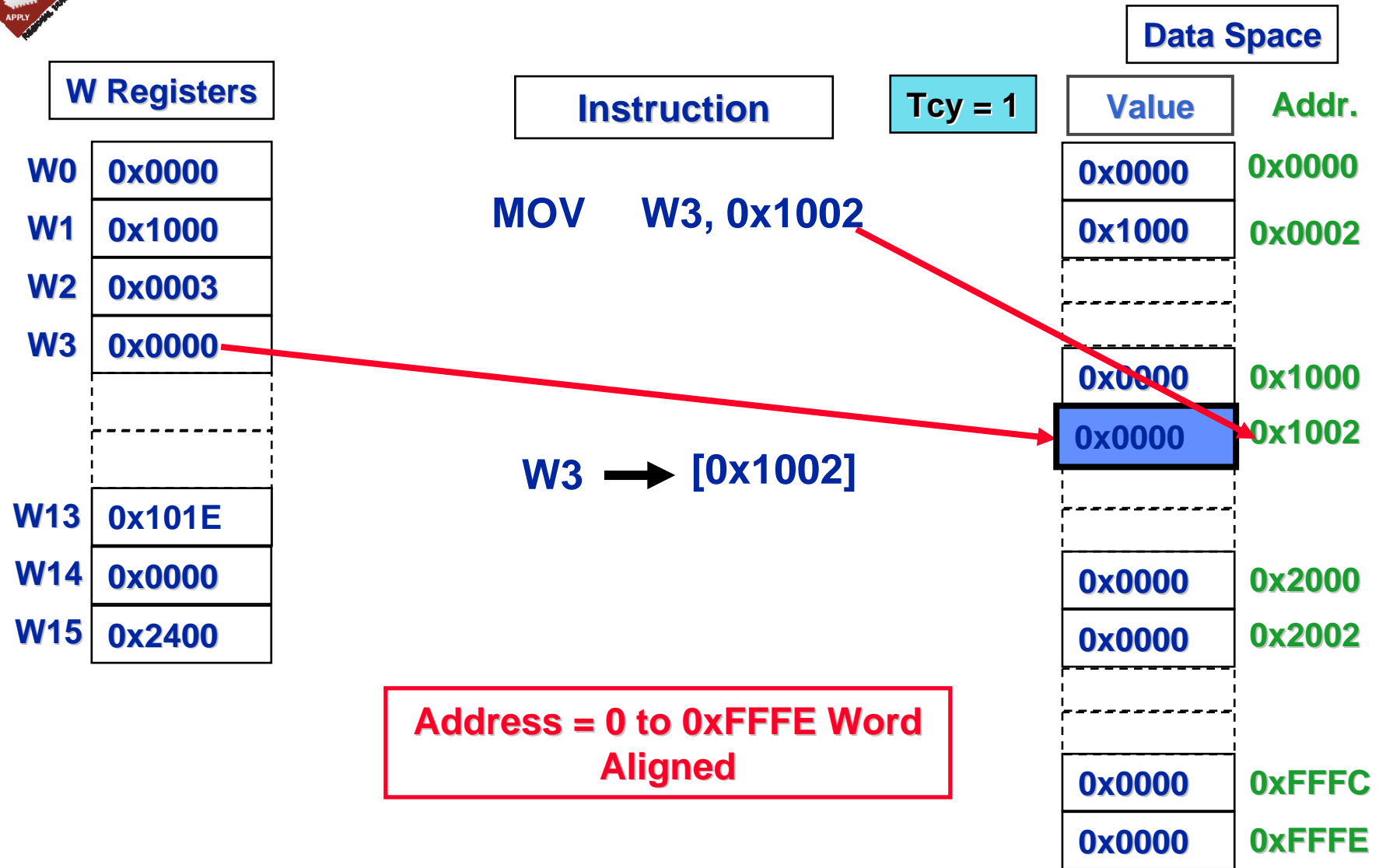


# MOV Instructions



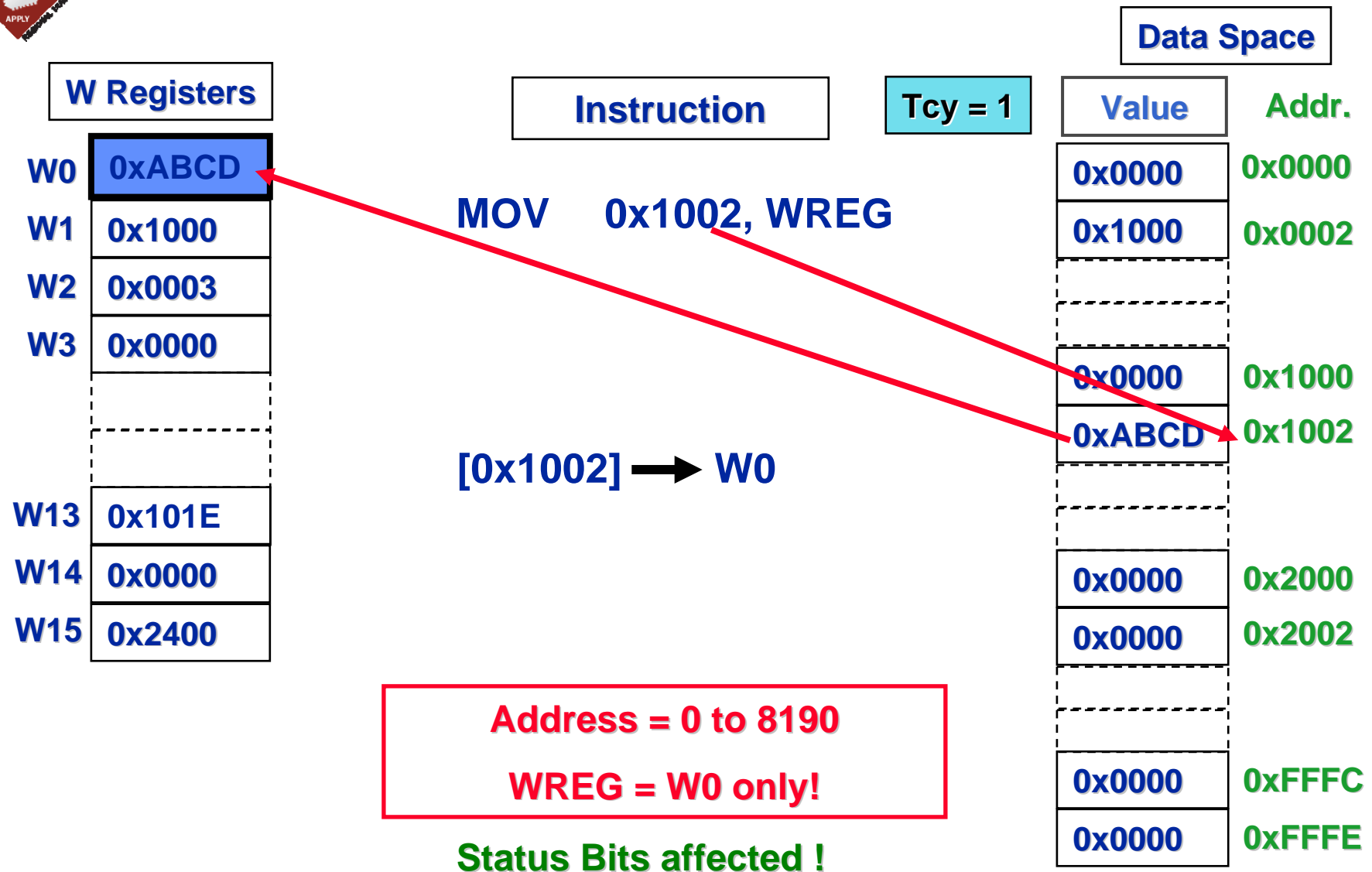


# MOV Instructions



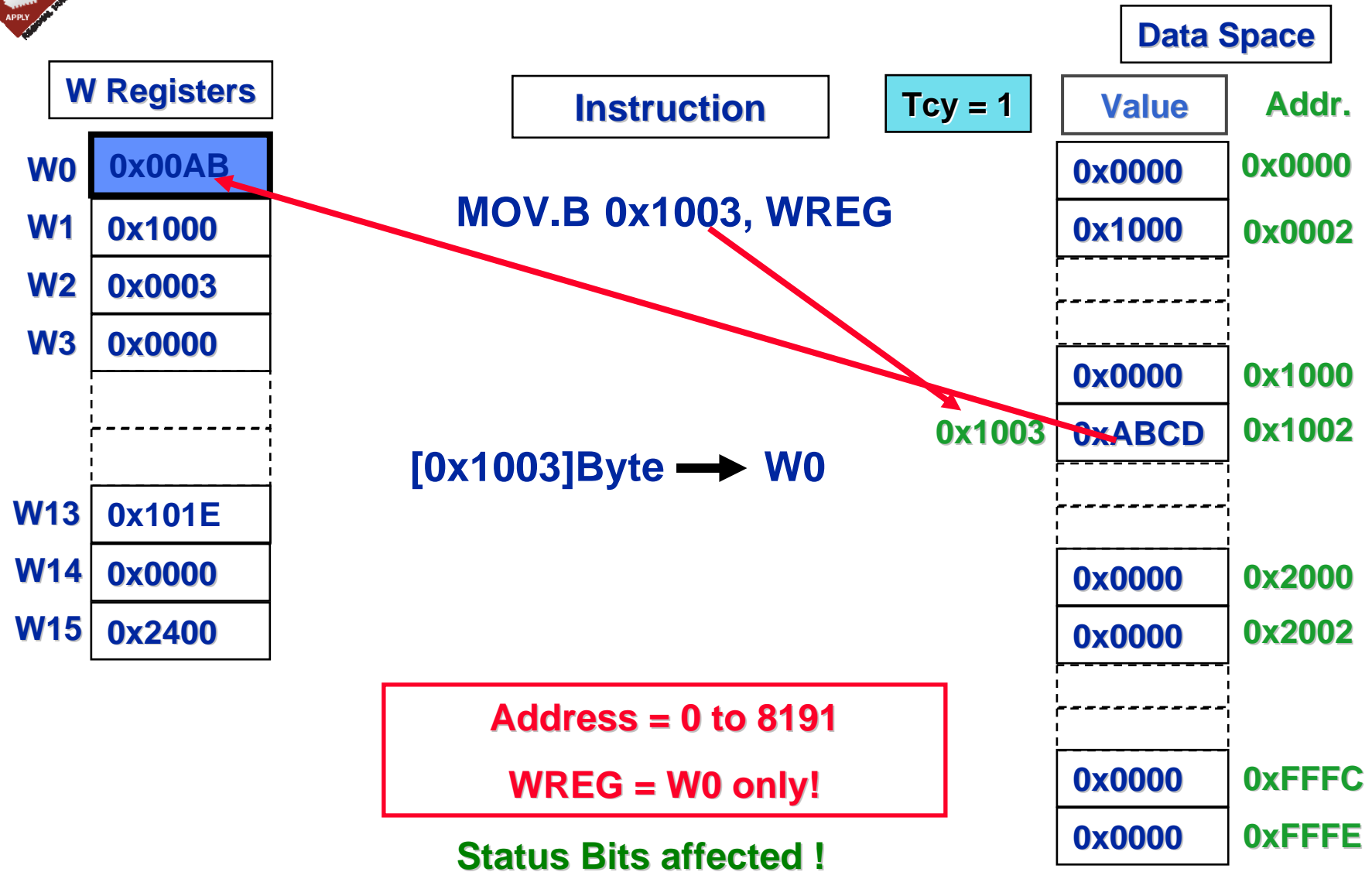


# MOV Instructions





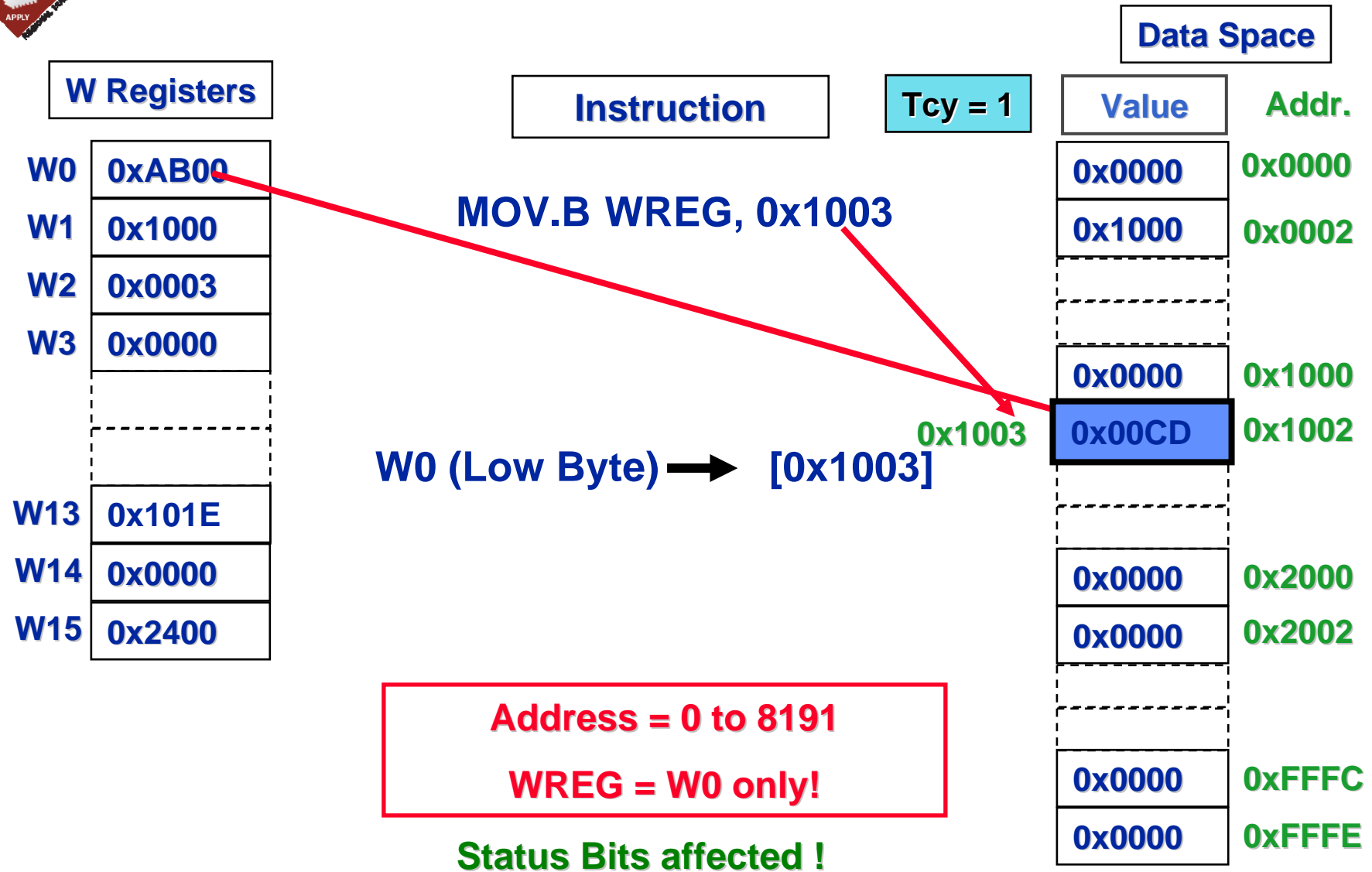
# MOV Instructions





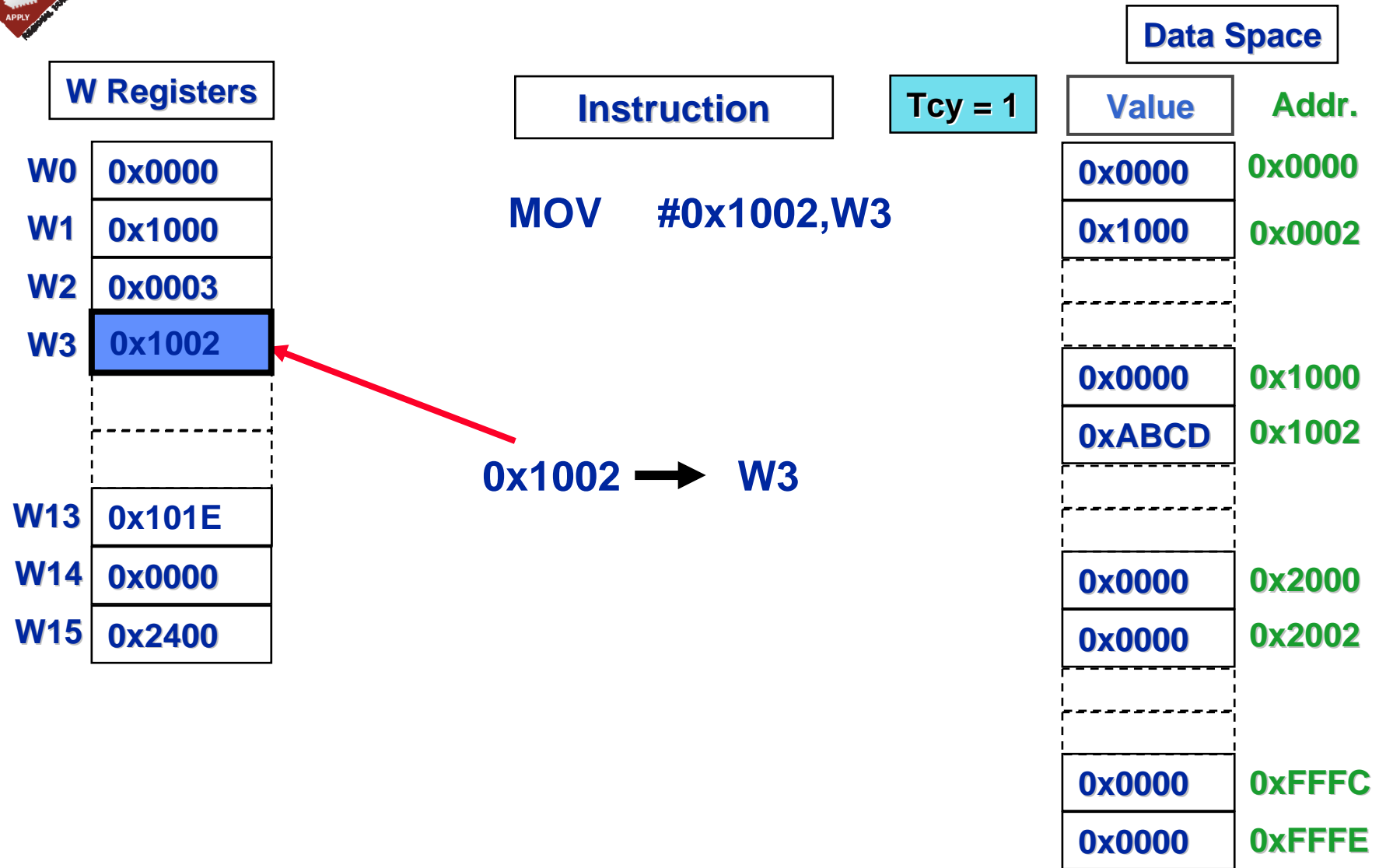


# MOV Instructions



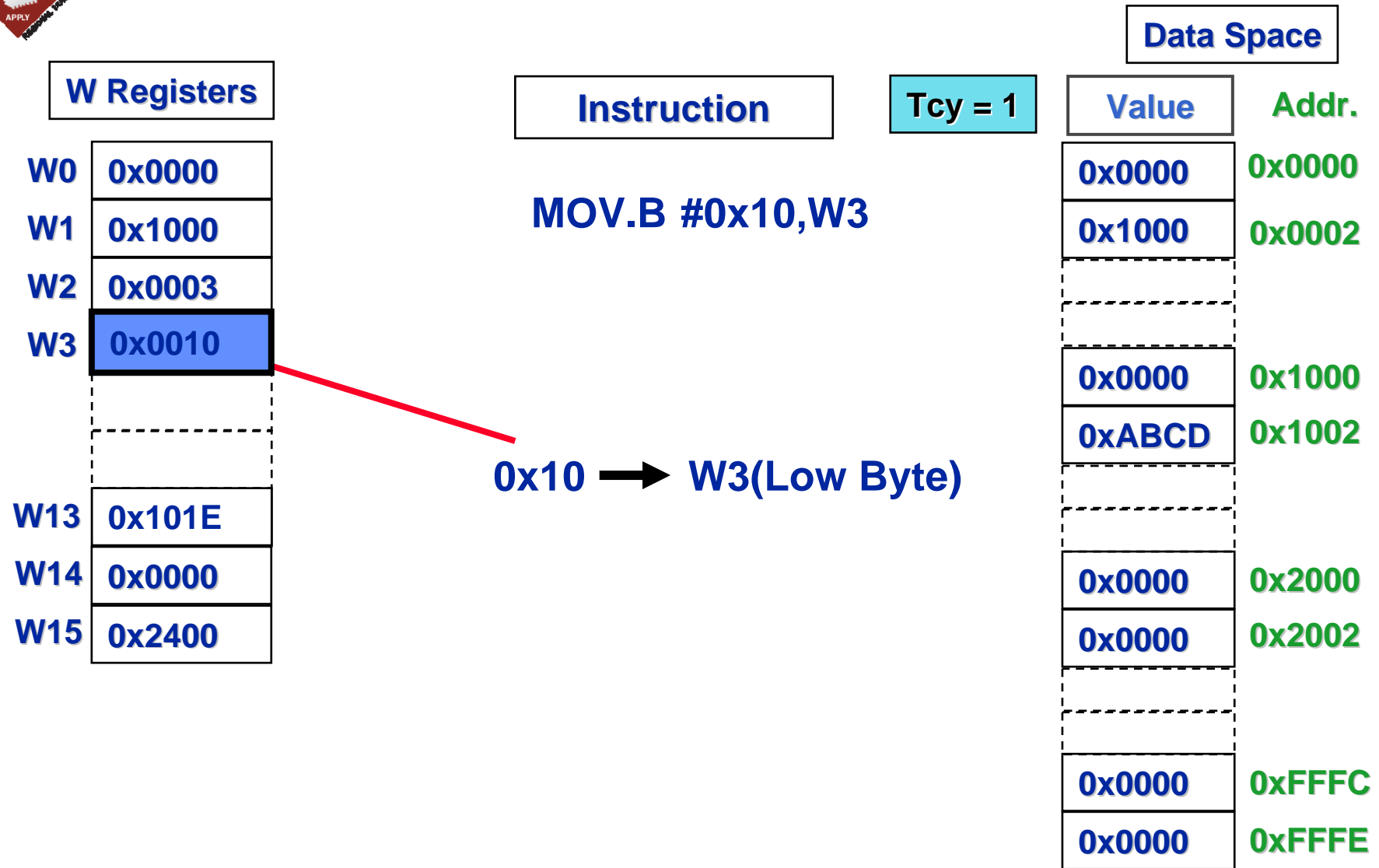


# MOV Instructions



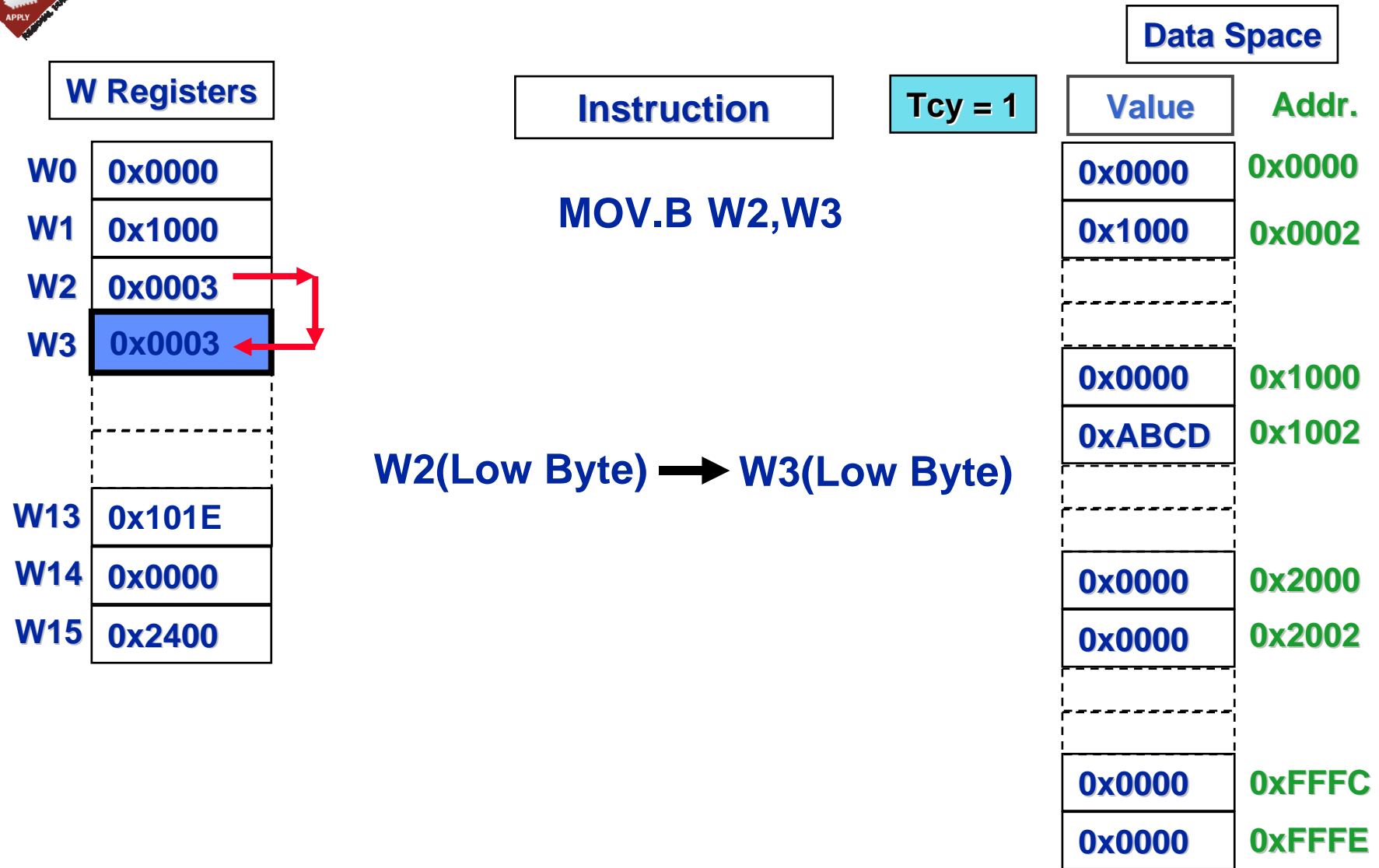


# MOV Instructions



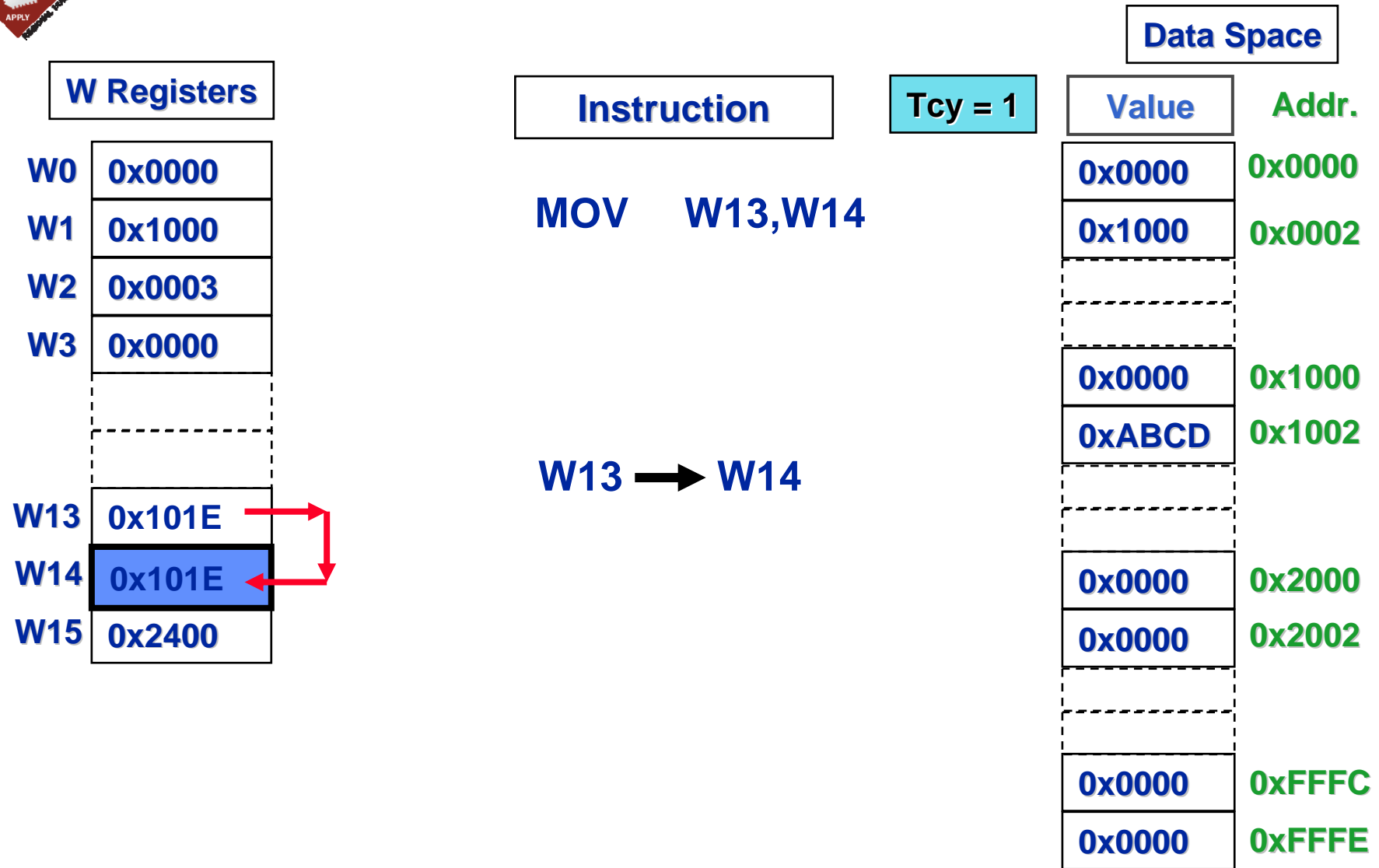


# MOV Instructions



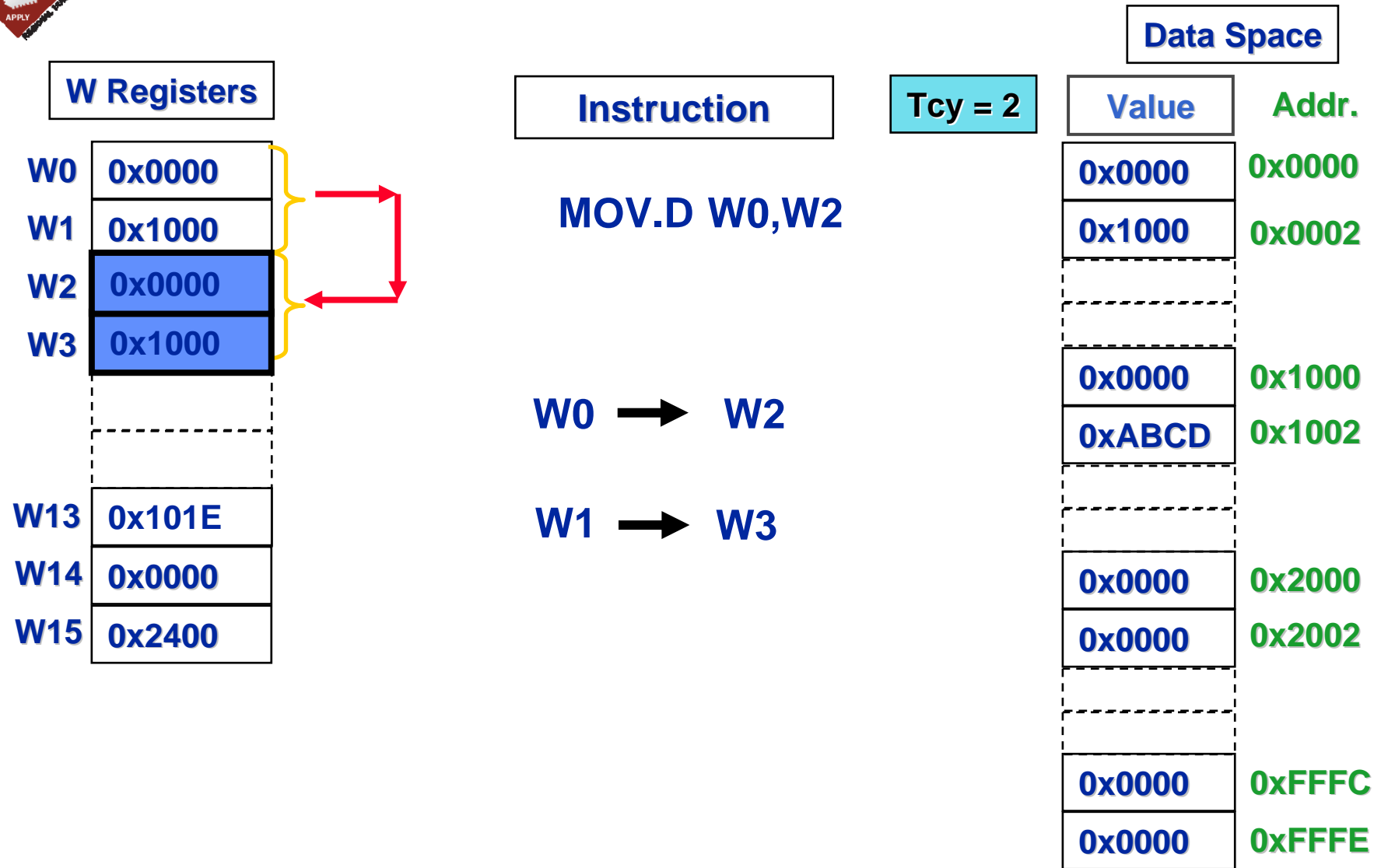


# MOV Instructions





# MOV Instructions





# Instruction Set Functional Groups

- **16-bit Instructions**

- **Move Instructions**

- **Bit Instructions**

- Follow along LAB 1

- **Math and Logic Instructions**

- **Stack Control Instructions**

- **Program Flow Control Instructions**

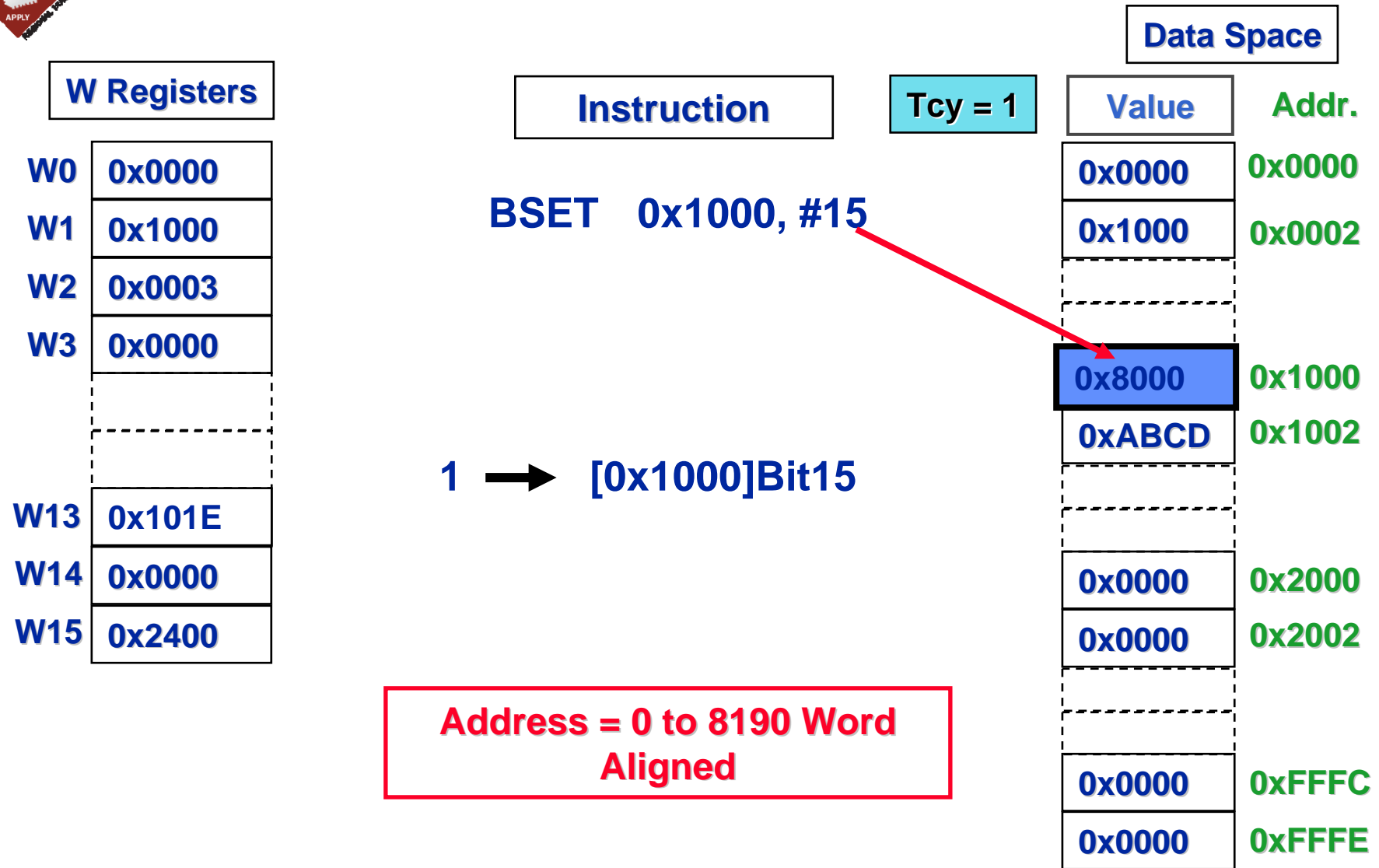
- **CPU Control Instructions**

- LAB 2



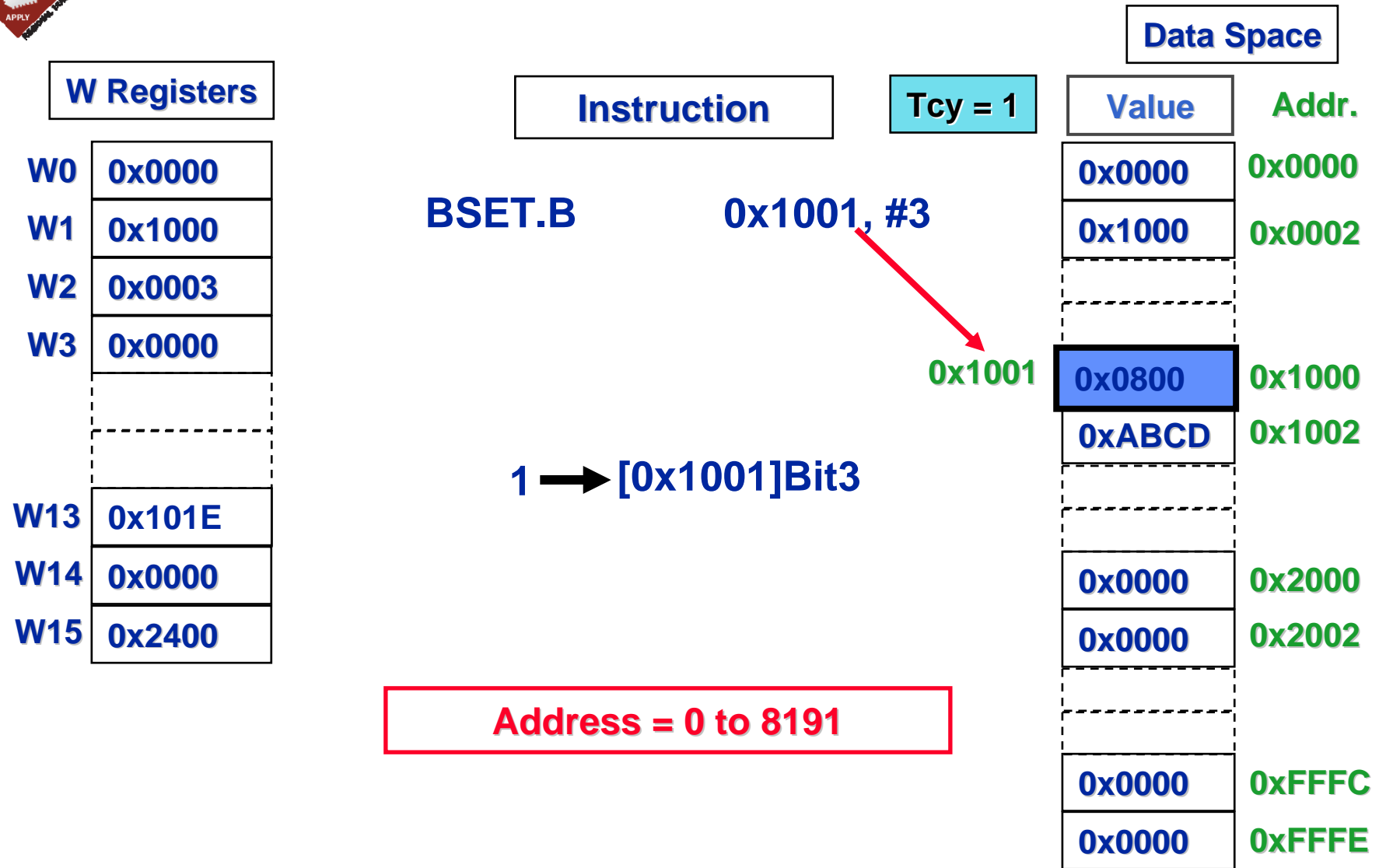


# BIT Instruction



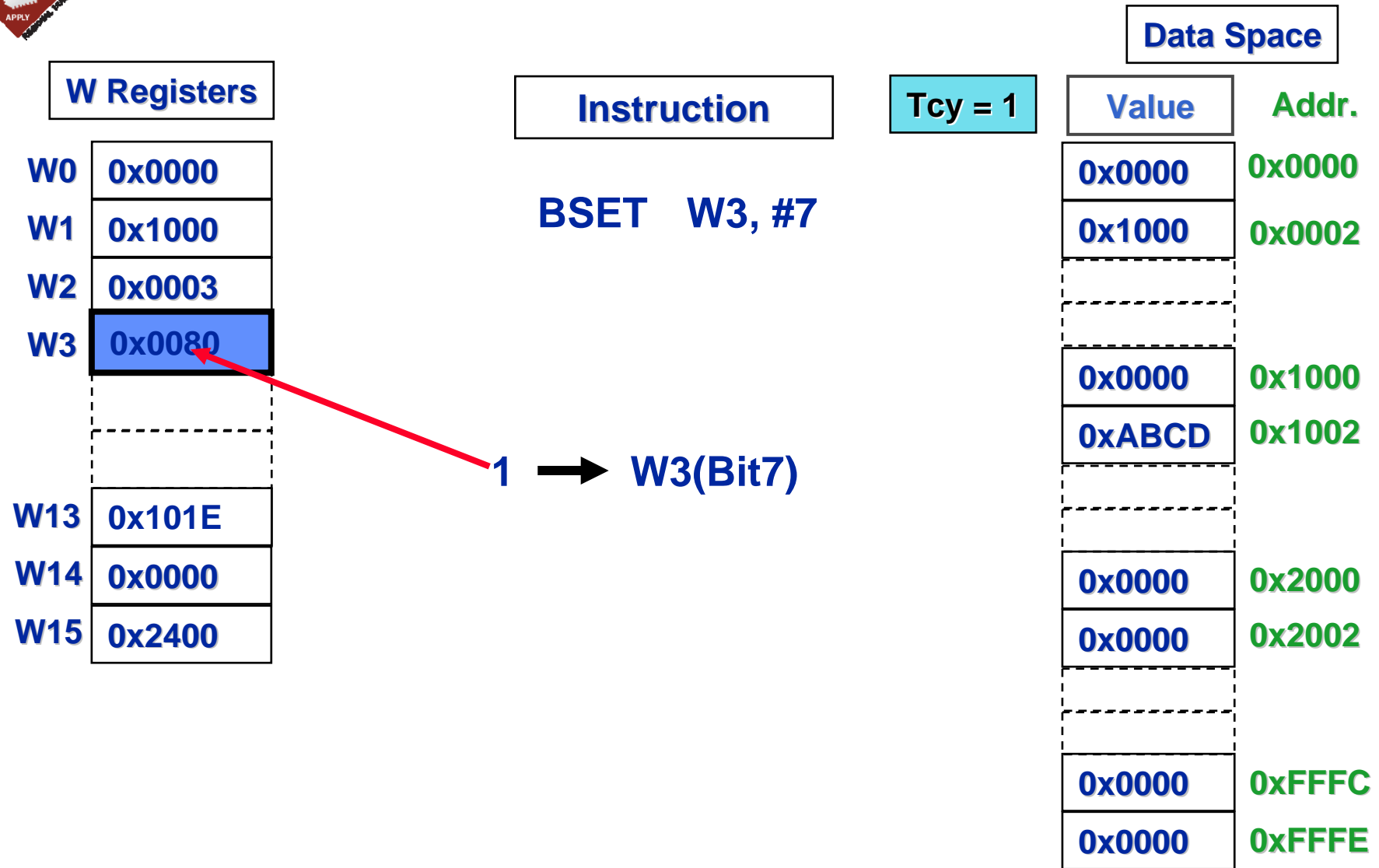


# BIT Instruction



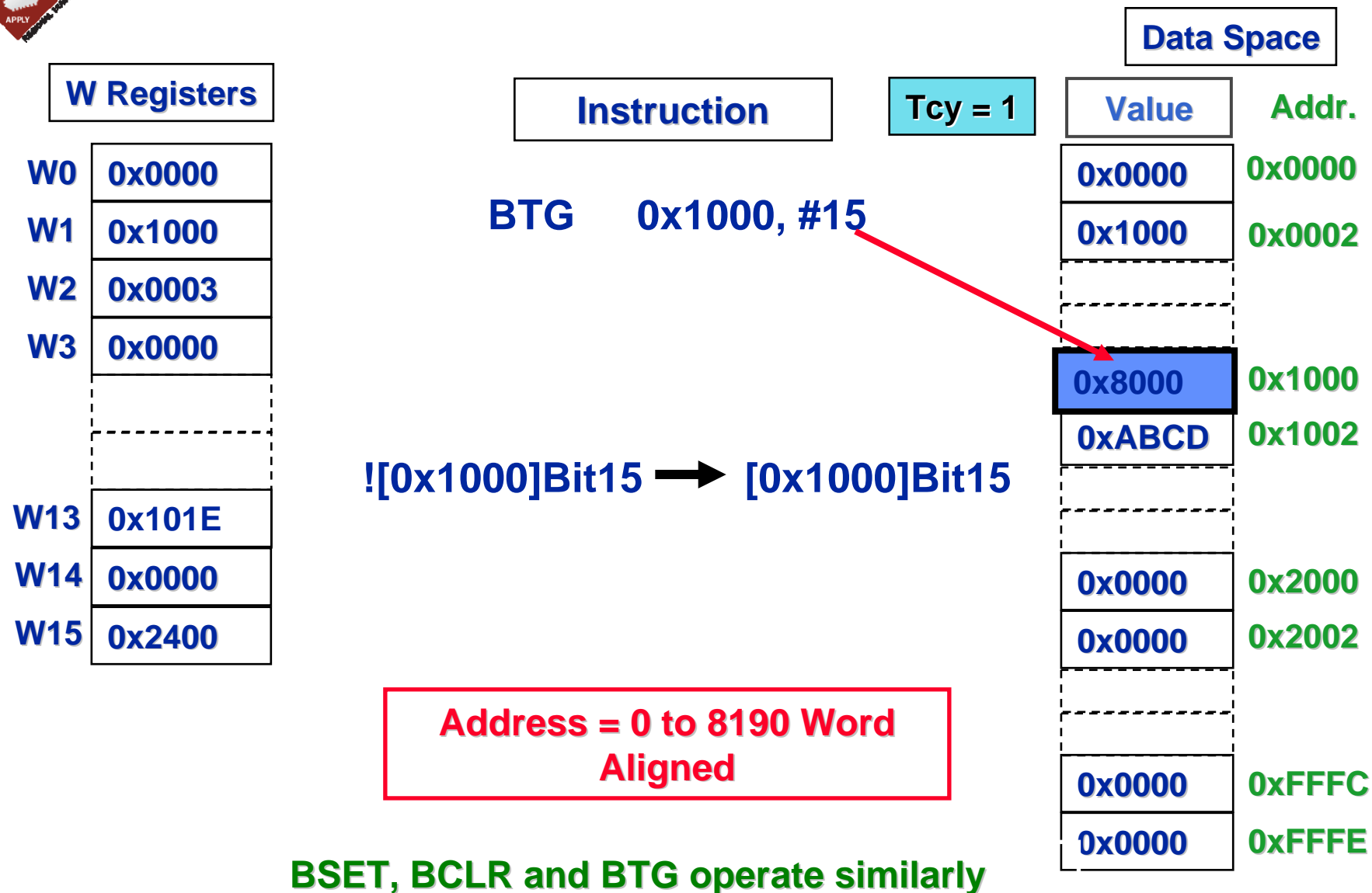


# BIT Instruction



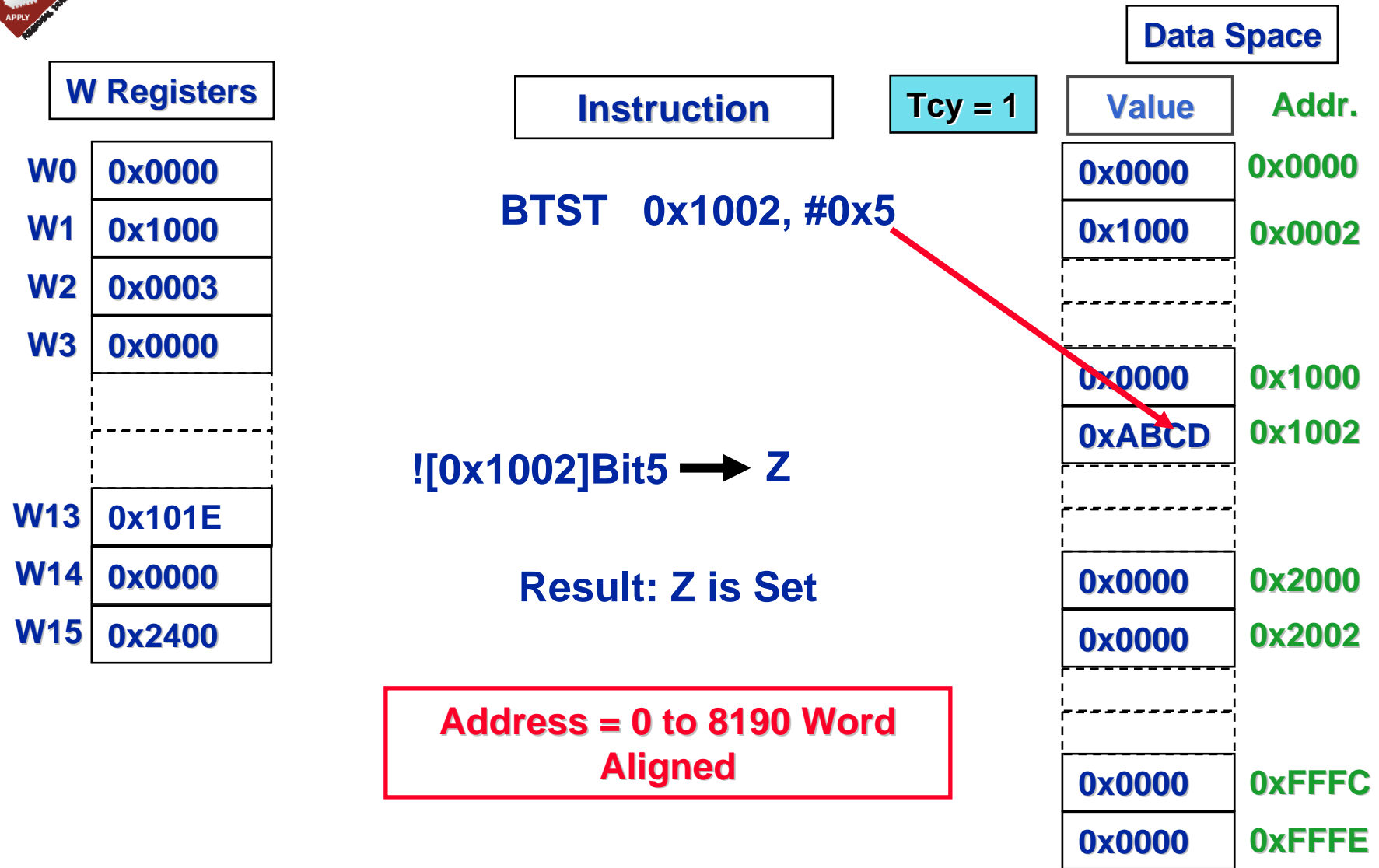


# BIT Instruction



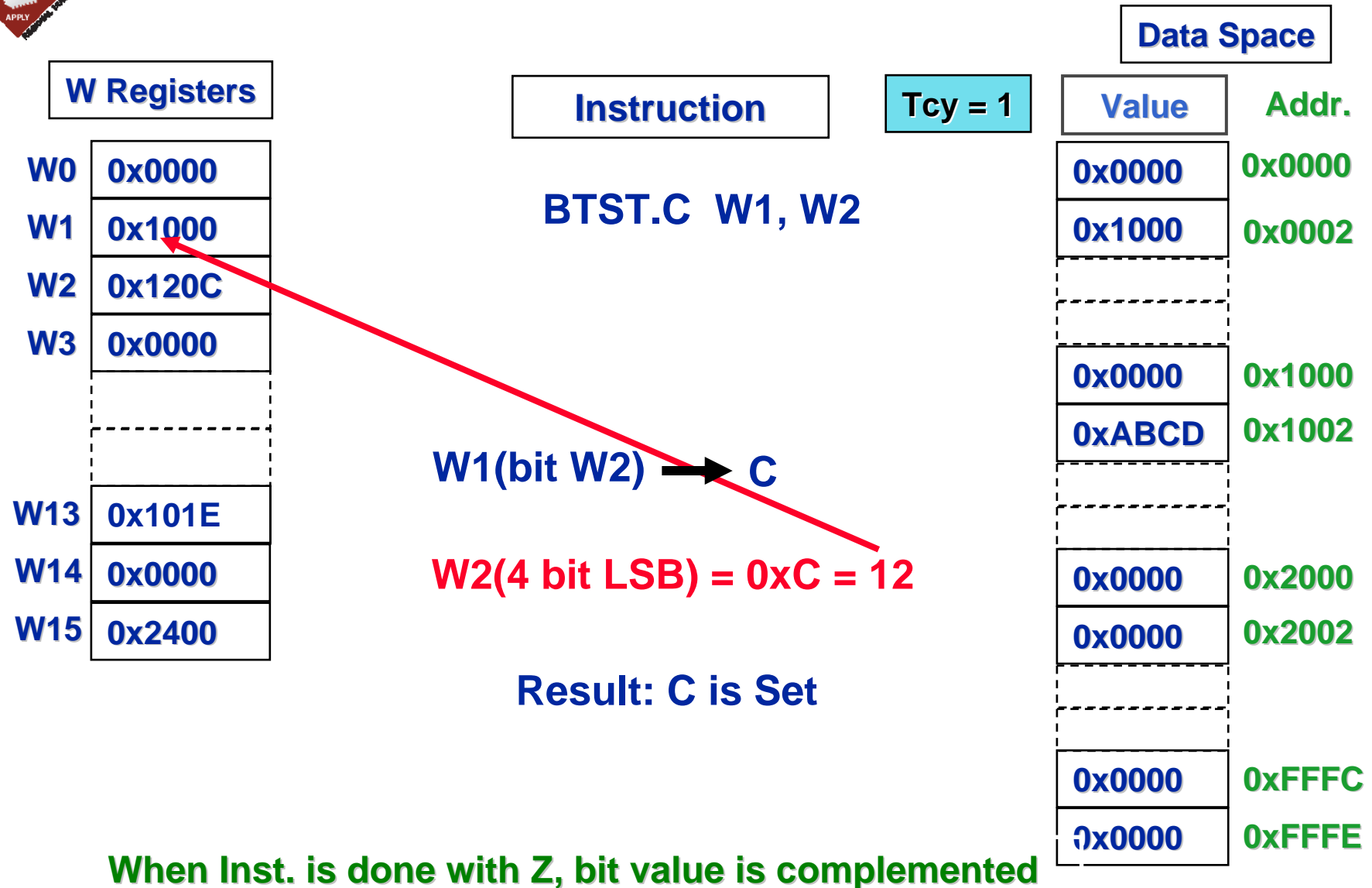


# BIT Instruction



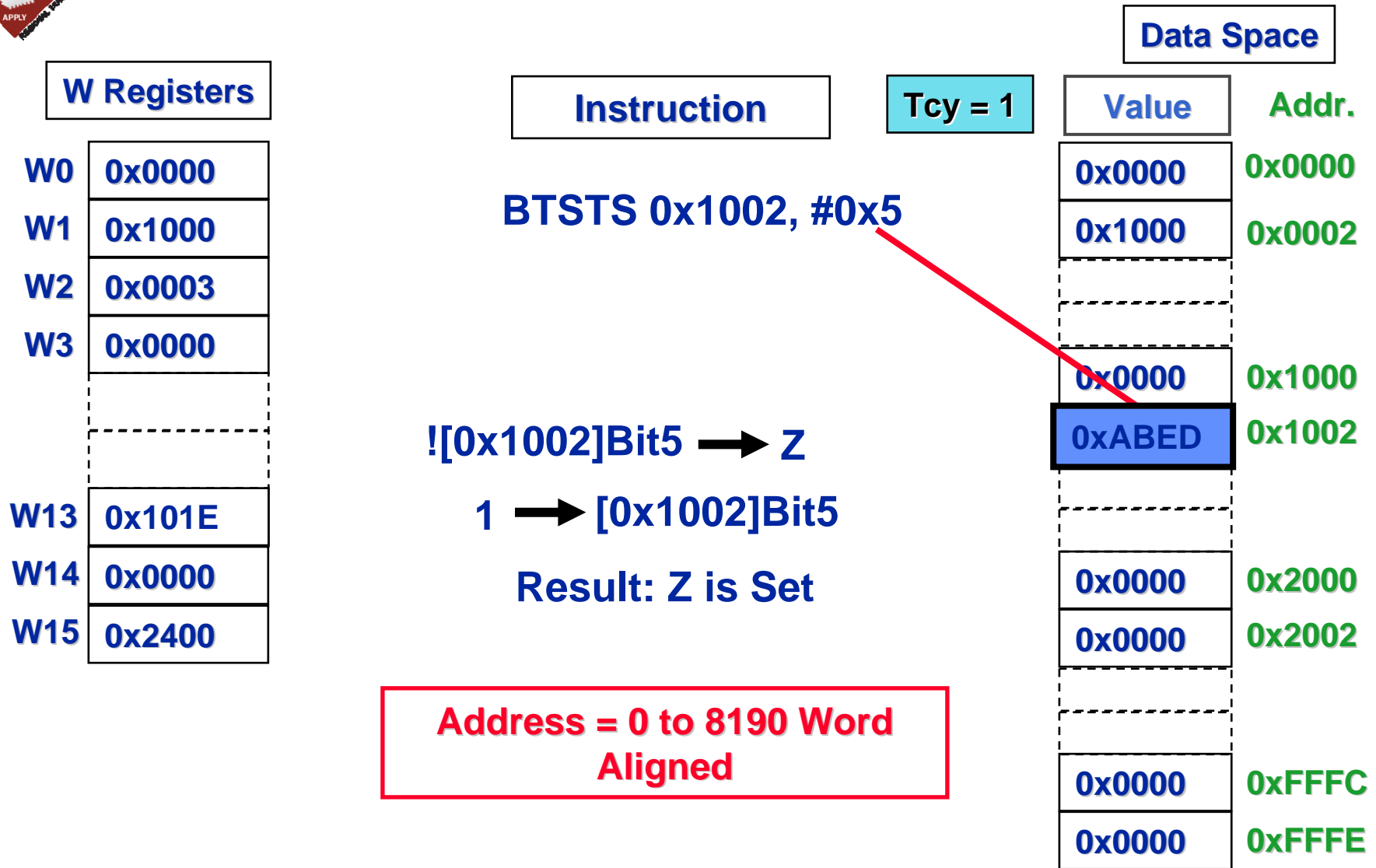


# BIT Instruction





# BIT Instruction







# Instruction Set Functional Groups

- **16-bit Instructions**

- **Move Instructions**

- **Bit Instructions**

- ● Follow along LAB 1

- **Math and Logic Instructions**

- **Stack Control Instructions**

- **Program Flow Control Instructions**

- **CPU Control Instructions**

- LAB 2

# HANDS-ON

# Training

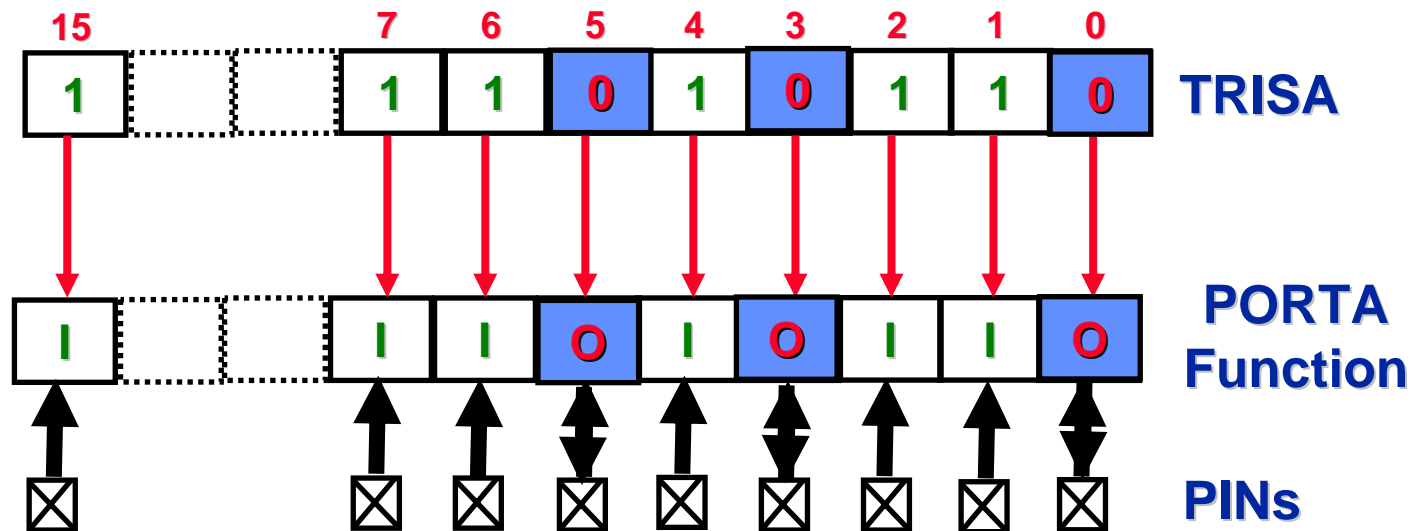
## Follow Along LAB 1

### Setup and Turn ON LED





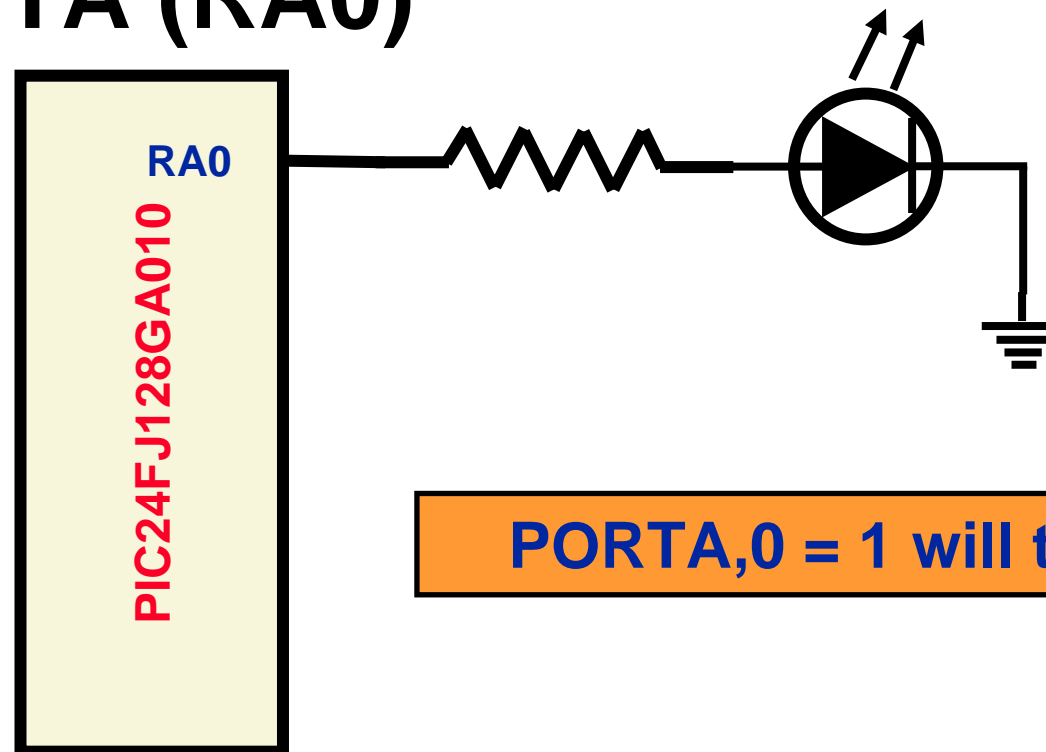
# I/O PORT





# Lab 1: The Task

- Make PORTA, bit 0 as output
- Turn on LED connected to bit 0 of PORTA (RA0)



**PORTA,0 = 1 will turn on LED**



# LAB1

- **Open Lab1.mcw workspace in MPLAB**
  - C:\rtc\103ASP\Lab1.mcw
- **Follow along and edit code:**
  - Setup Stack Pointer and Stack Limit
  - Setup TRISA
  - Bit Set PORTA, bit 0
- **Compile Code**
- **Program Part on Explorer-16 using ICD2**
- **Run Code using ICD2**
- **See the LED light up!!!**

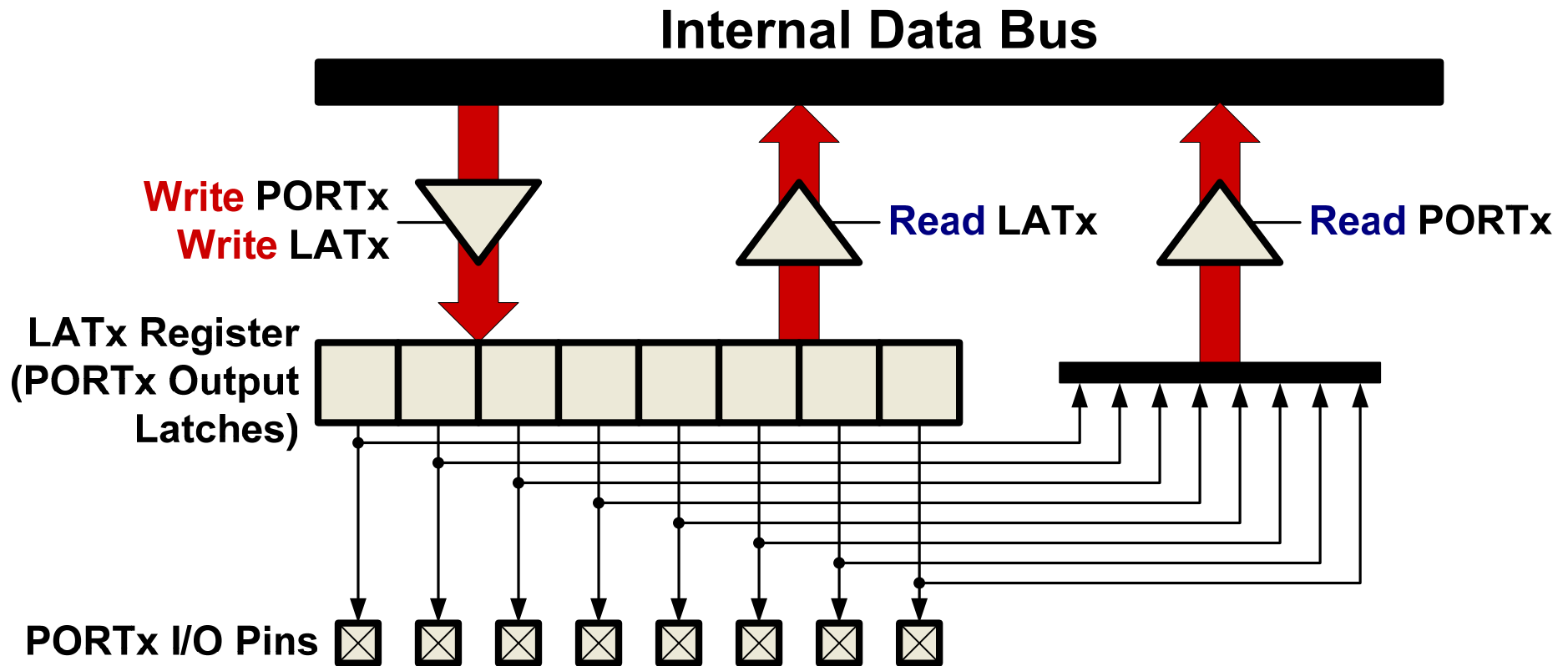


# More Than one LED lights up

- **You may notice:**
  - More than One LED Lights up when RUN is executes
  - WHY?



# Digital I/O Ports – 16-bit



**BSET PORTA,#0 ;Turn LED @ PORTA bit 0 ON**

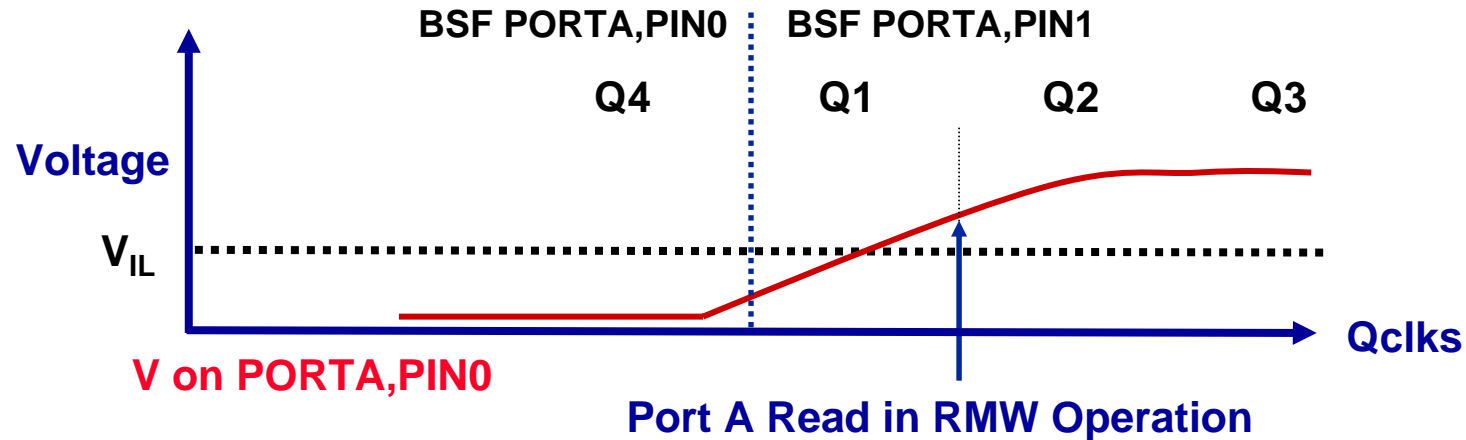
**BSET PORTA,#1 ;Turn LED @ PORTA bit 1 ON**



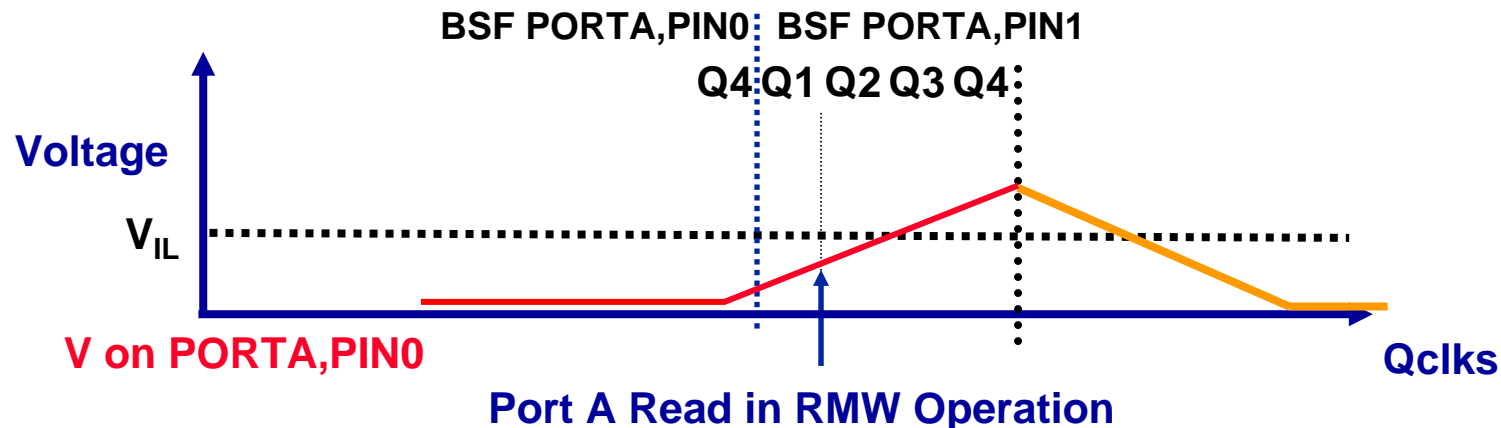


# Read-Modify-Write Issue

## At Low Frequency or Low Capacitive Loading



## At High Frequency or High Capacitive Loading





# Solution for RMW Issue

- **Use RMW instruction on the LATch:**
  - BSF LATA,#0
  - BSF LATA,#1



# LAB1 Summary

- **Wrote our First 16-bit PIC code 😊**
- **Used some of the Assembly Language Inst.**
- **Learned how to program a 16-bit PIC device using ICD2**
- **Got the Program to WORK!!**



# Instruction Set Functional Groups

- **16-bit Instructions**

- **Move Instructions**

- **Bit Instructions**

- Follow along LAB 1



- **Math and Logic Instructions**

- **Stack Control Instructions**

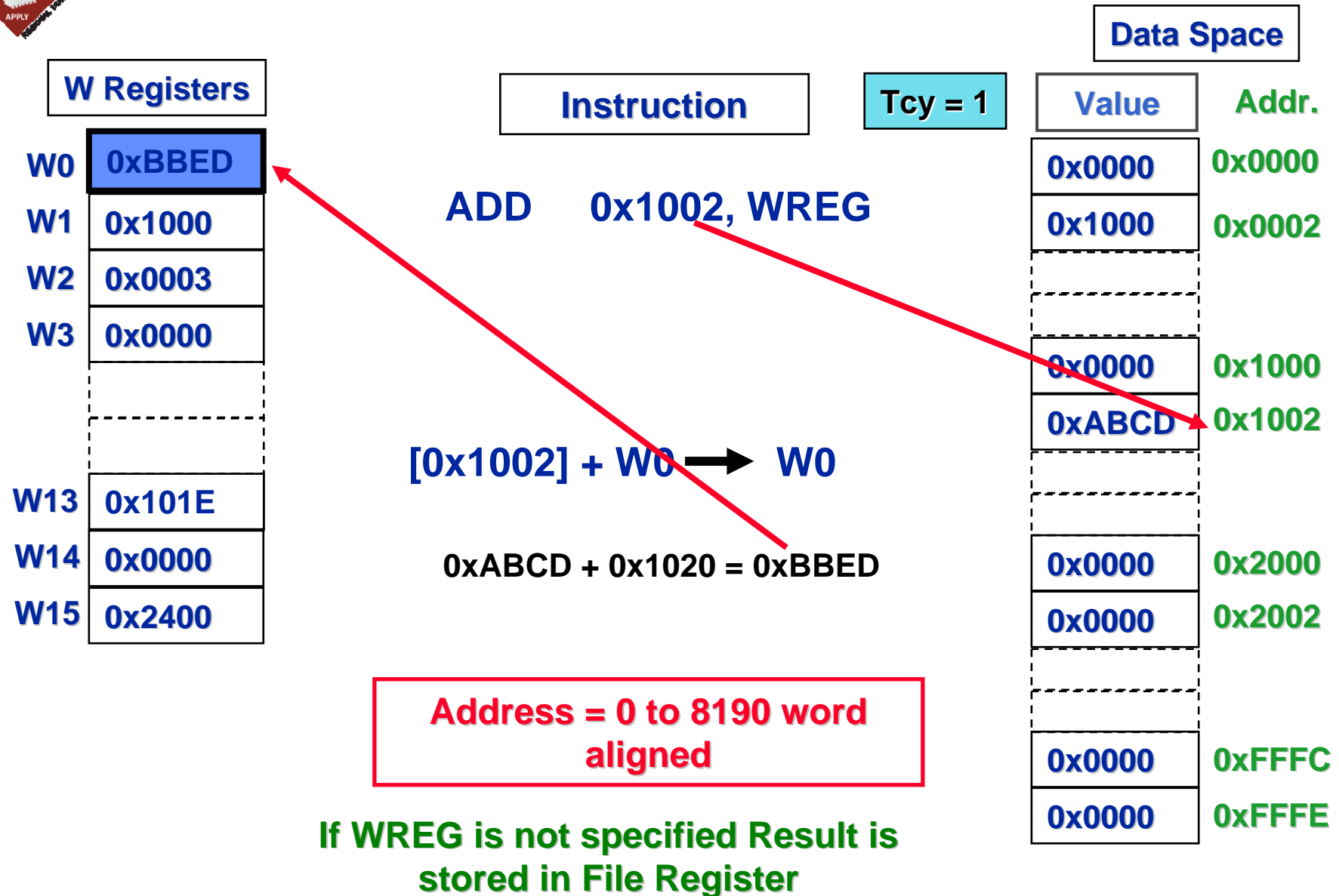
- **Program Flow Control Instructions**

- **CPU Control Instructions**

- LAB 2

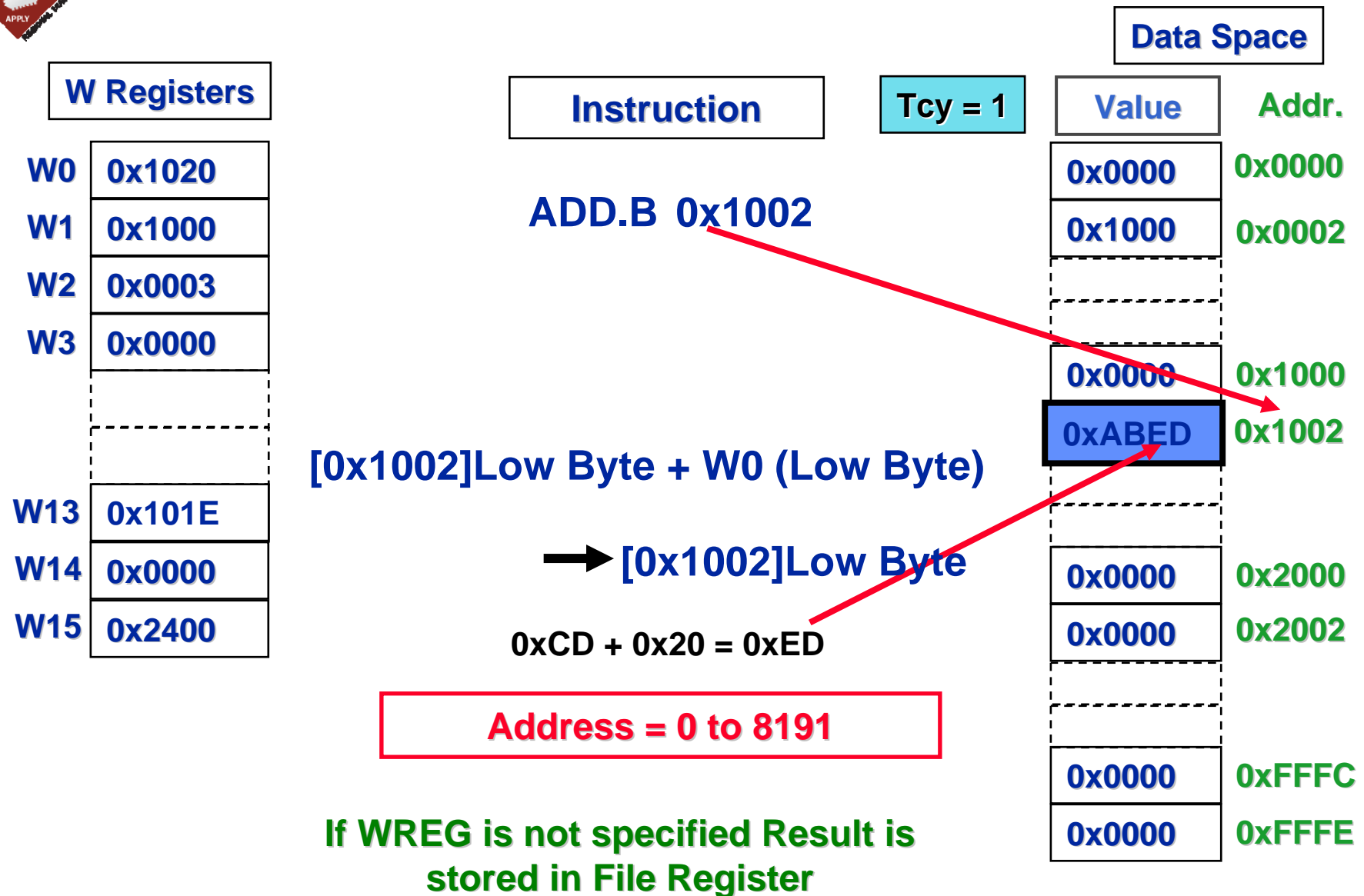


# ADD Instructions



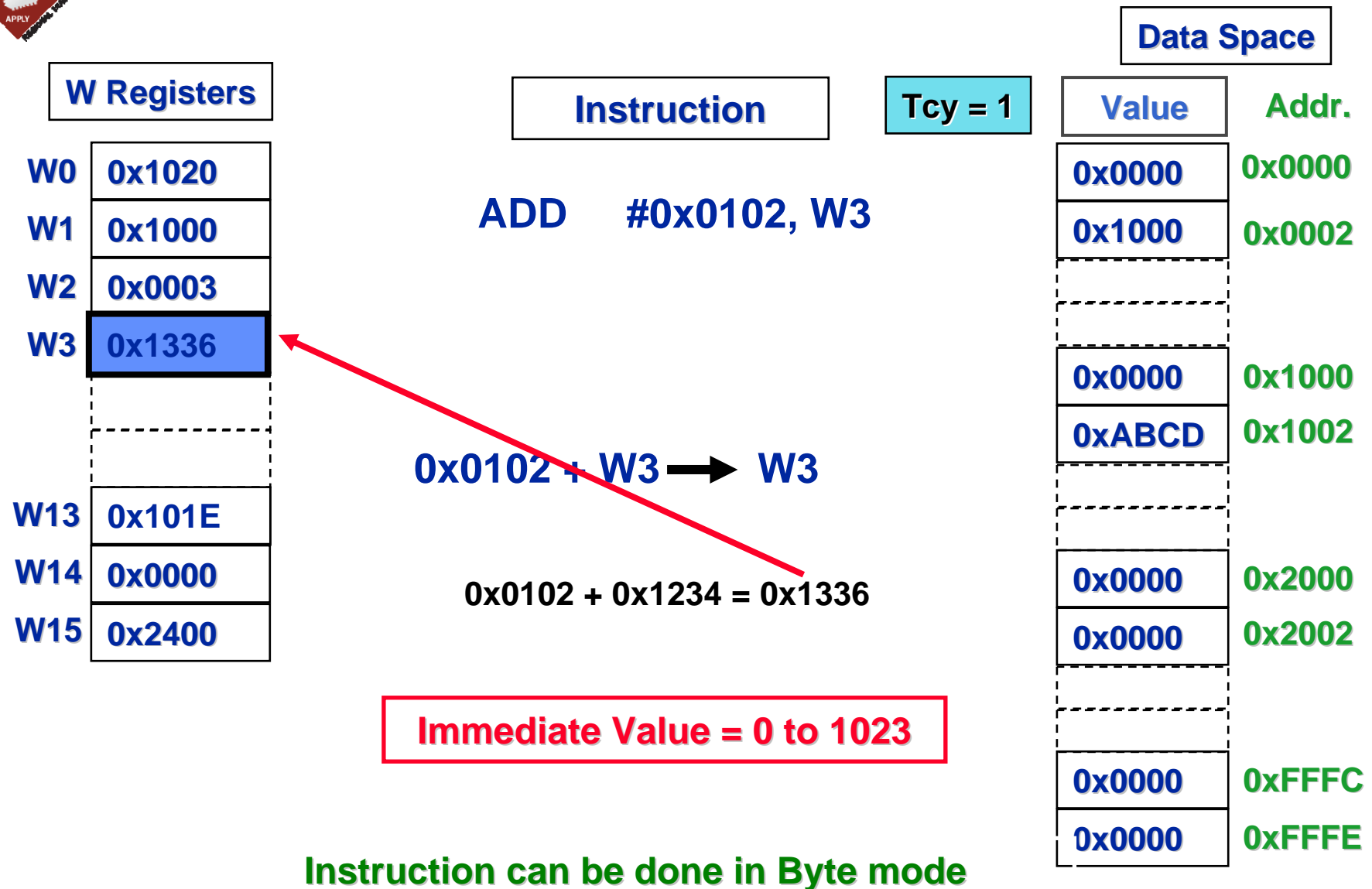


# ADD Instructions





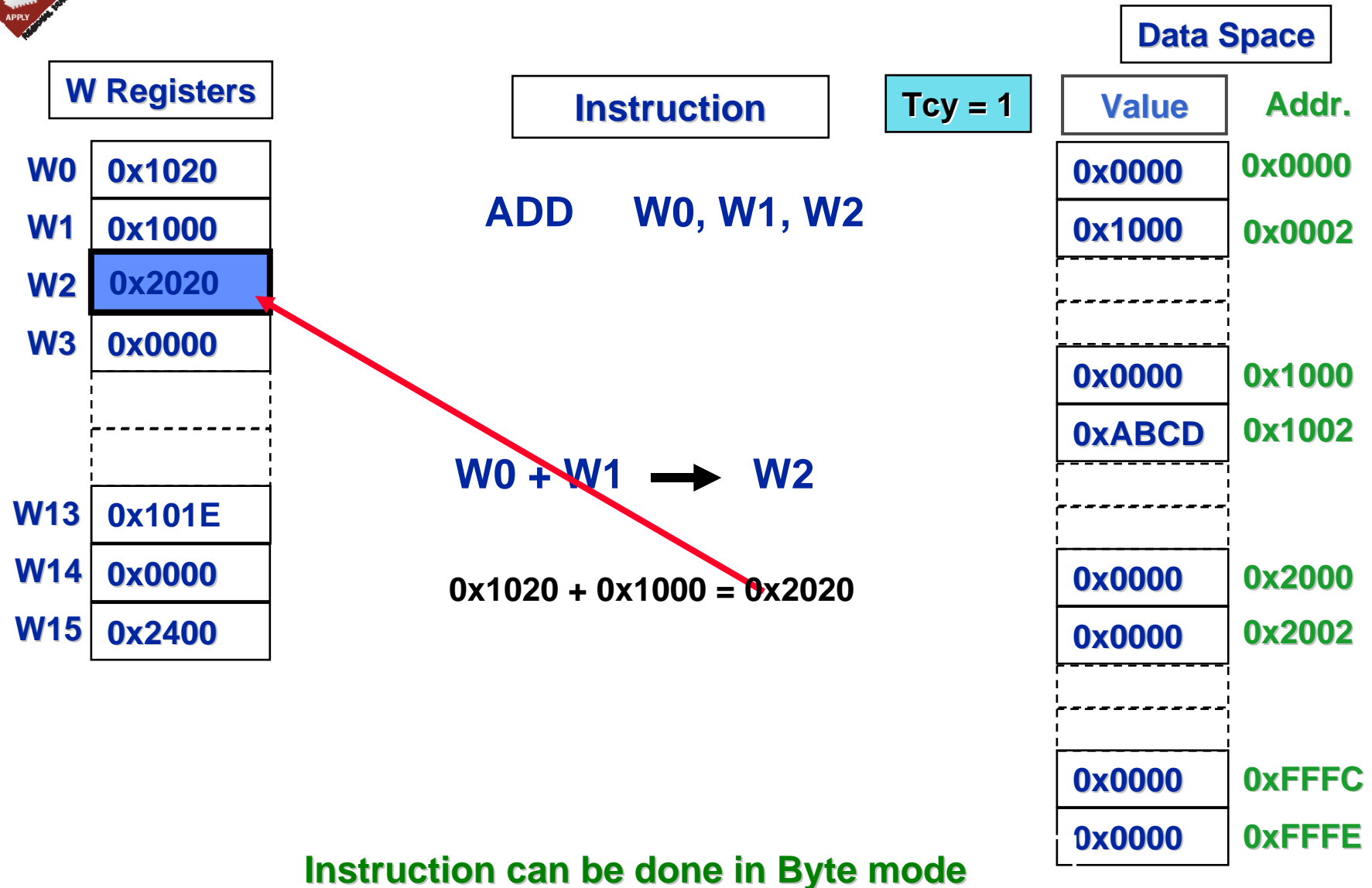
# ADD Instructions







# ADD Instructions



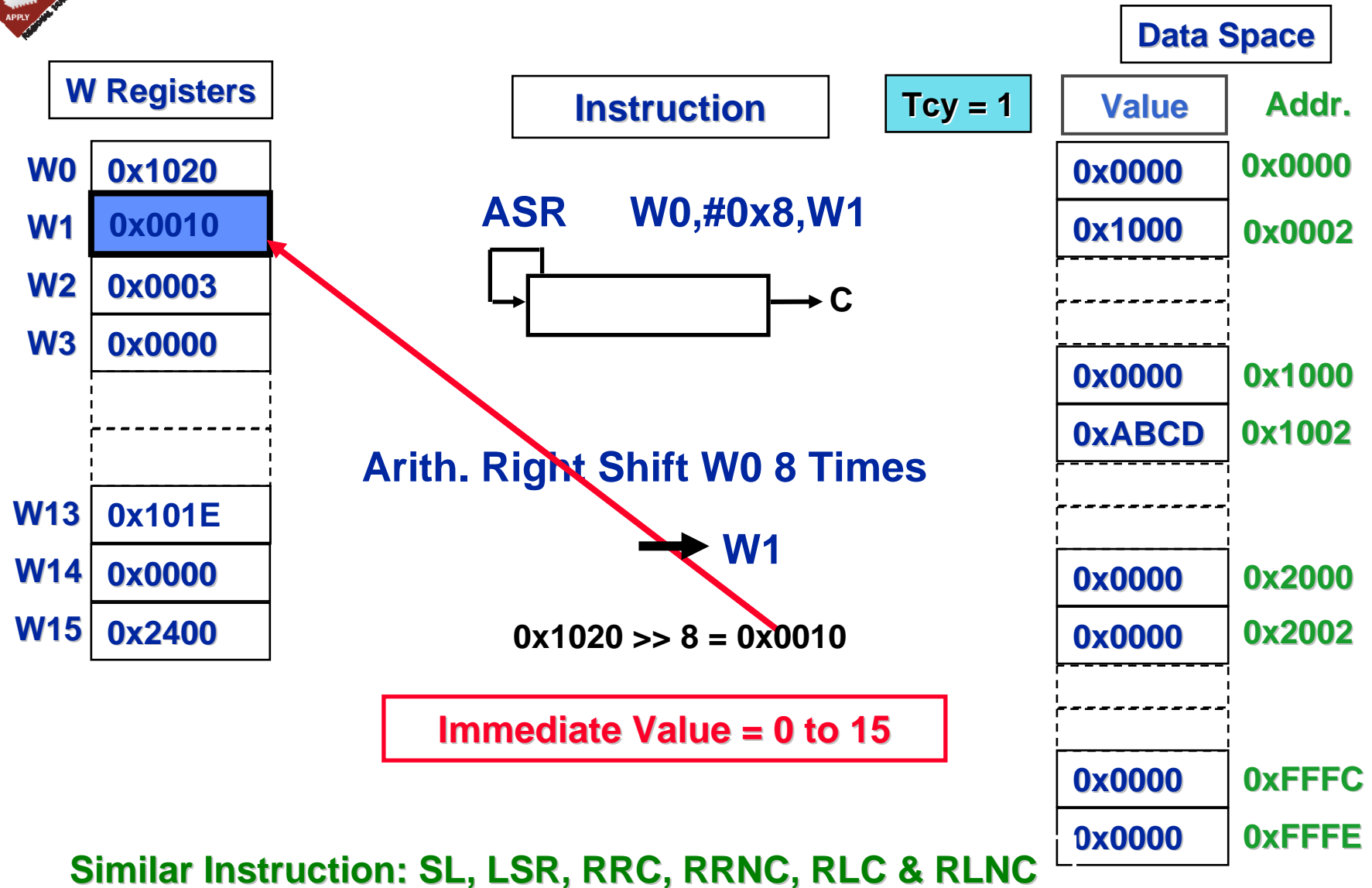


# Similar Instruction to ADD

- **ADDC, SUB and SUBB**
- **INC, INC2, DEC and DEC2**
- **SUBBR and SUBR**

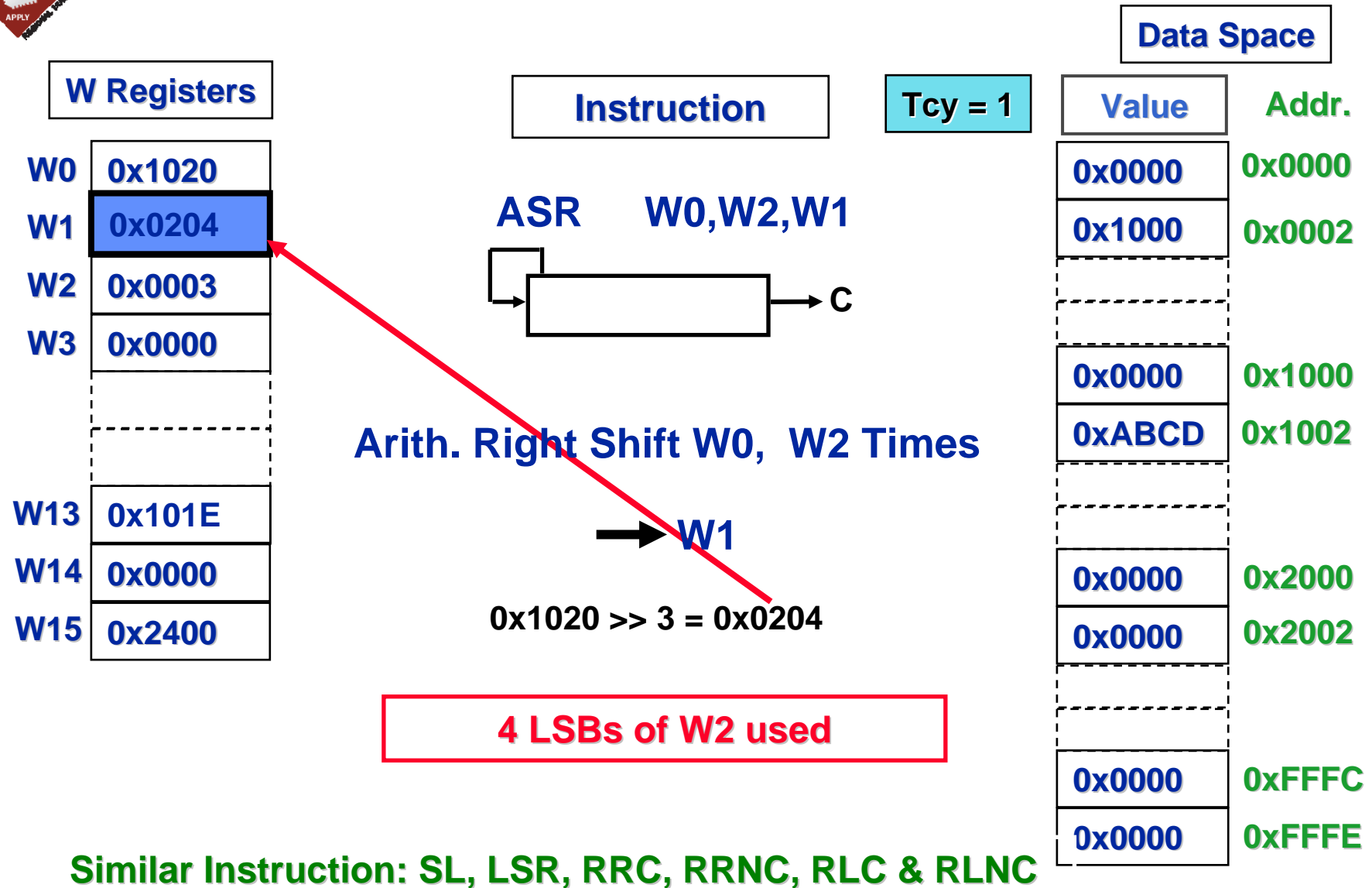


# Shift Instructions





# Shift Instructions





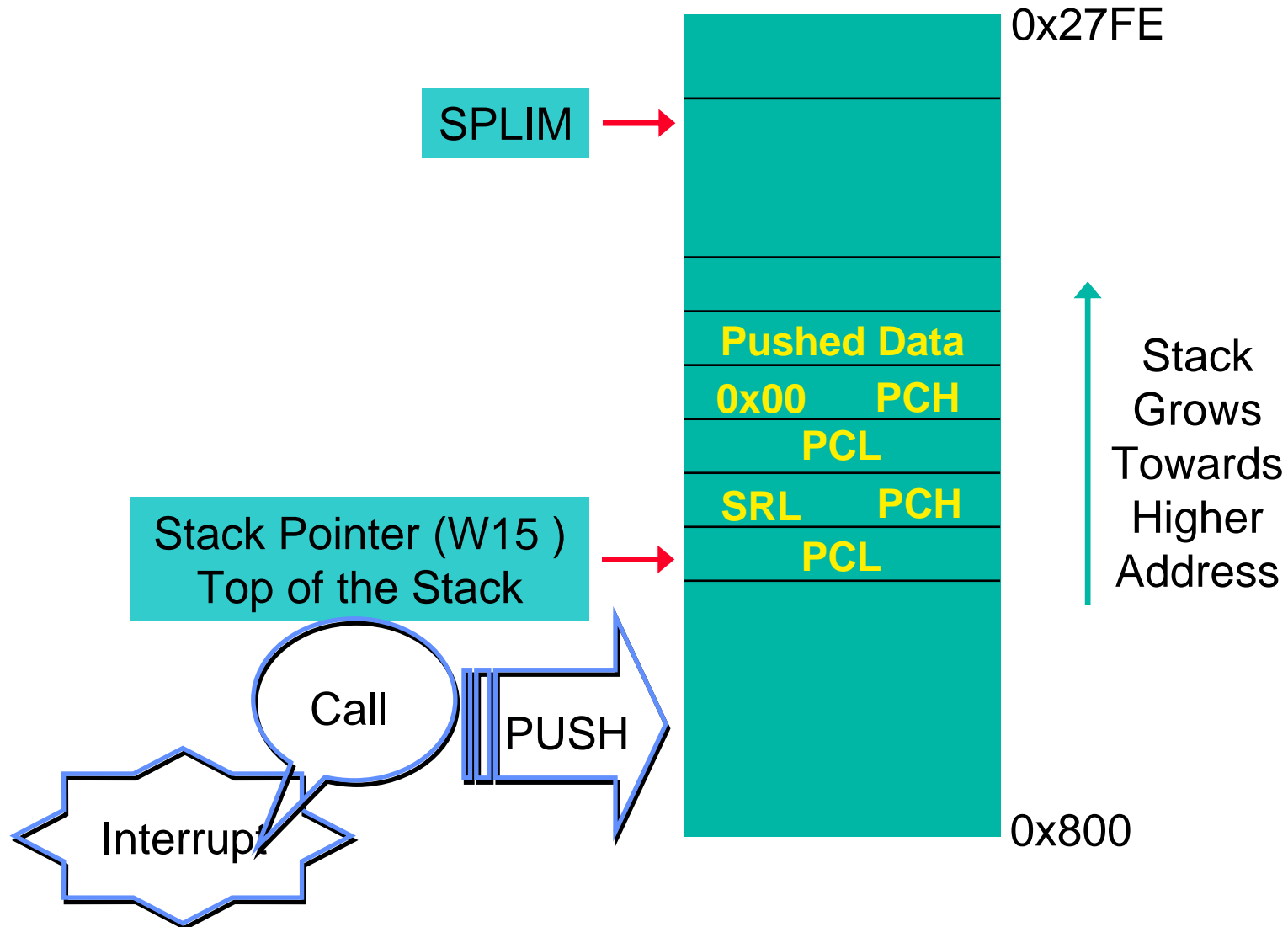
# Instruction Set Functional Groups

- **16-bit Instructions**

- **Move Instructions**
- **Bit Instructions**
  - Follow along LAB 1
- **Math and Logic Instructions**
- **Stack Control Instructions**
- **Program Flow Control Instructions**
- **CPU Control Instructions**
  - LAB 2

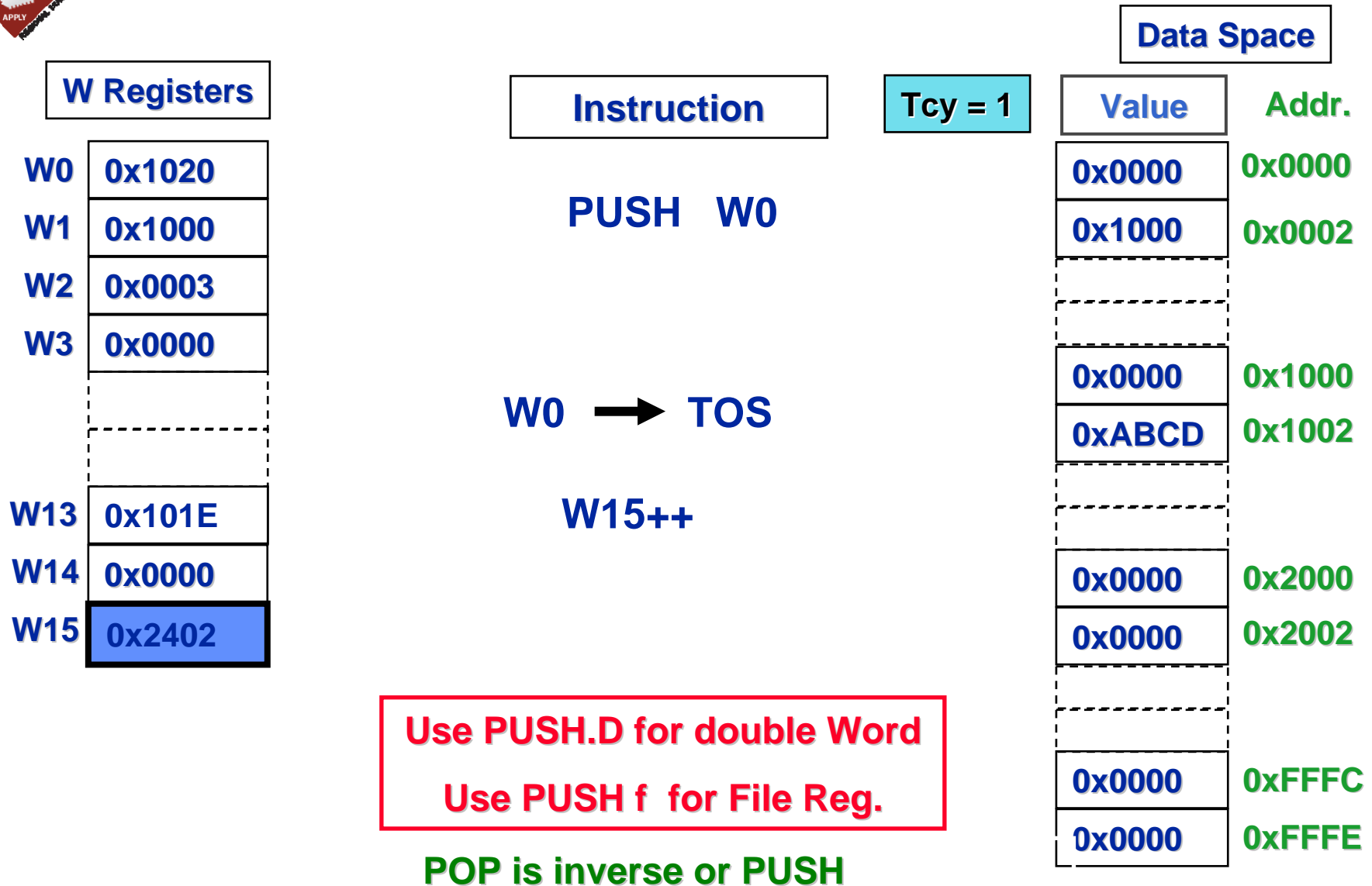


# Software Stack in Data RAM





# PUSH Instructions





# Instruction Set Functional Groups

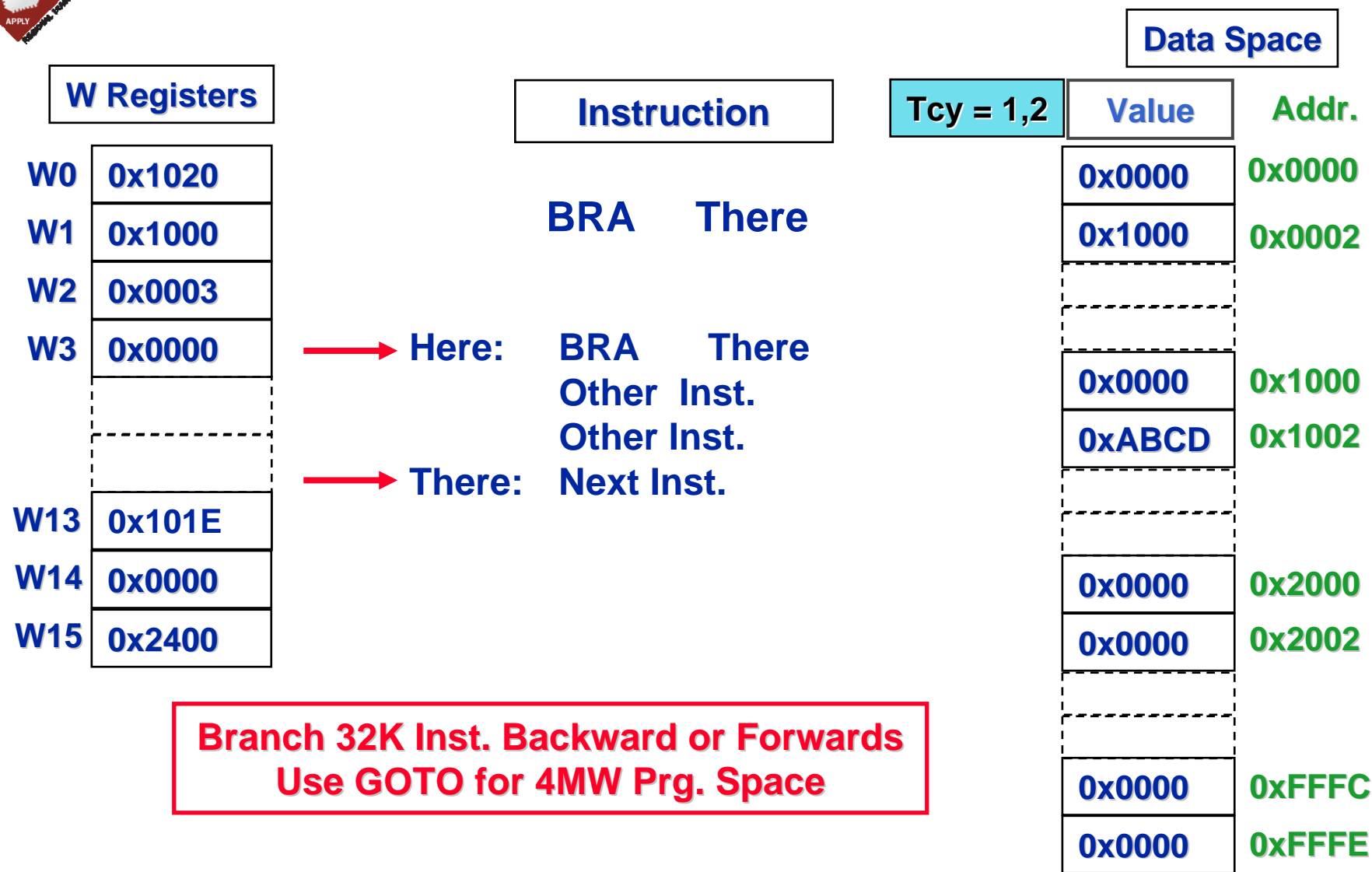
- **16-bit Instructions**

- **Move Instructions**
- **Bit Instructions**
  - Follow along LAB 1
- **Math and Logic Instructions**
- **Stack Control Instructions**
- — **Program Flow Control Instructions**
- **CPU Control Instructions**
  - LAB 2





# Program Flow Instructions



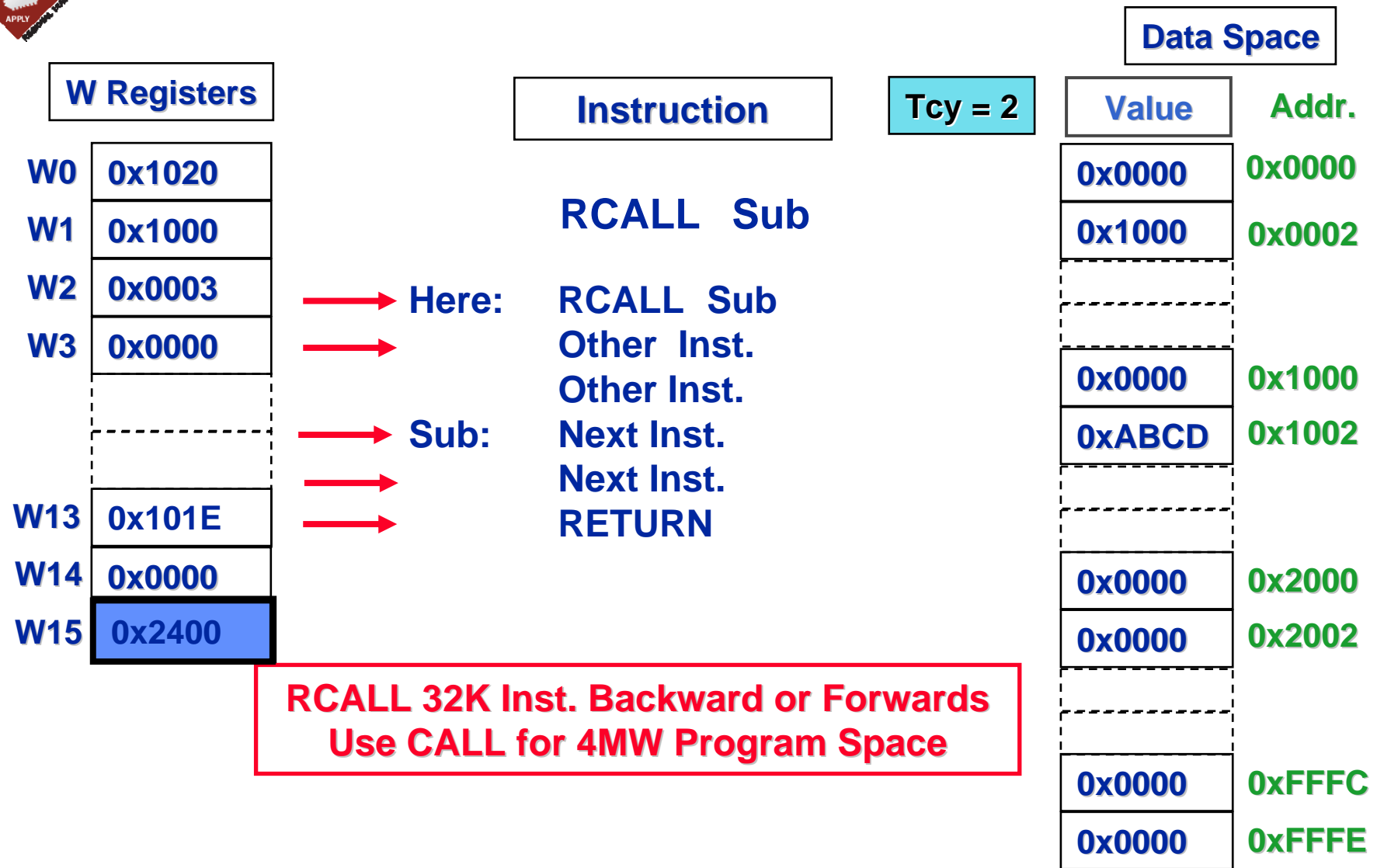


# Conditional Branches

- **BRA C There ; if Carry Set**
- **BRA NC There ; if Not Carry**
- **BRA GT There ; if Signed Greater Than**
- **BRA GTU There ; if UnSigned Greater Than**
- **BRA LT There ; if Signed Less Than**
- **BRA LTU There ; if UnSigned Less Than**
- **BRA Z There ; if Zero**
- **BRA NZ There ; if Not Zero**
- **BRA N There ; if Negative**
- **BRA NN There ; if Not Negative**

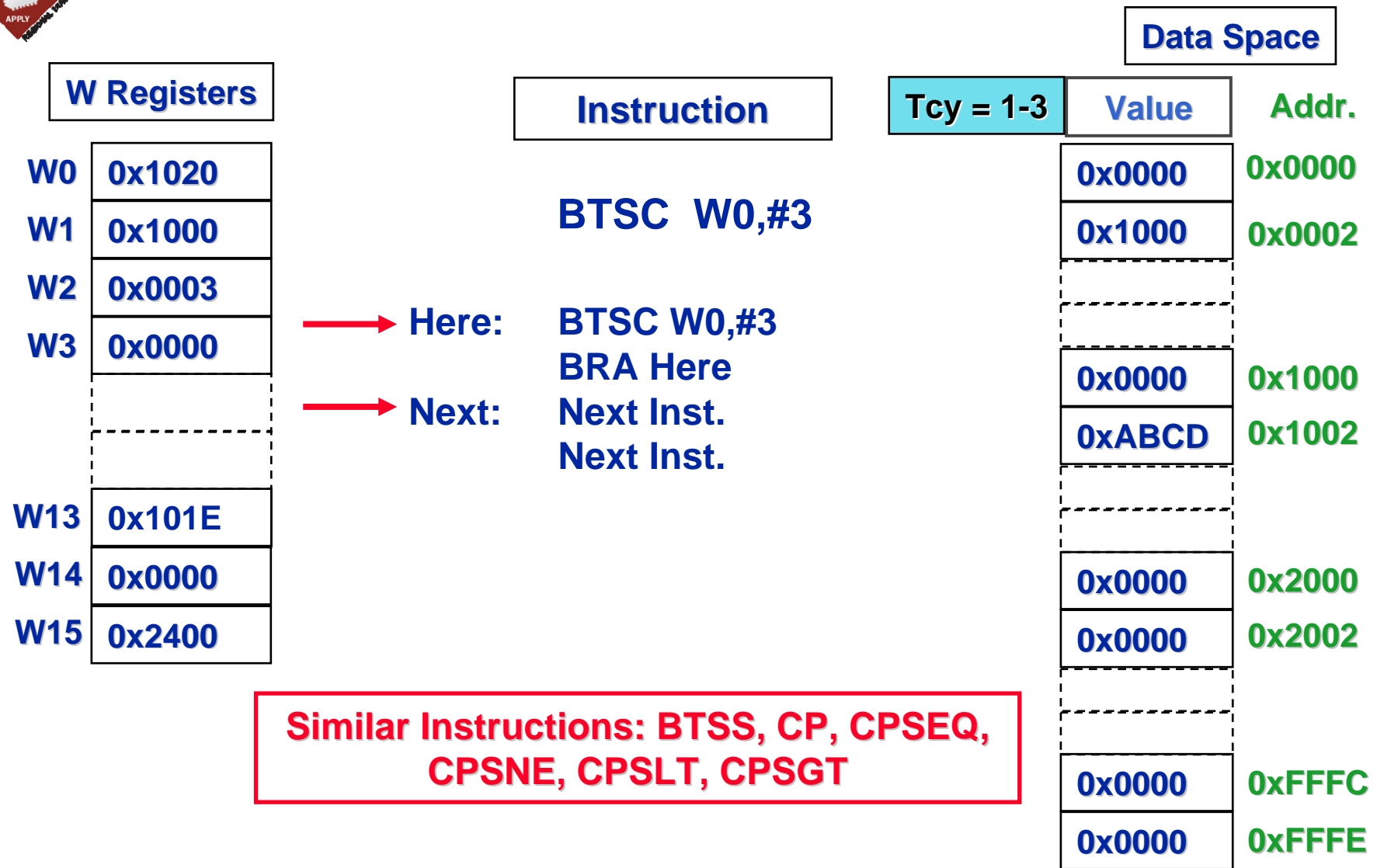


# Program Flow Instructions



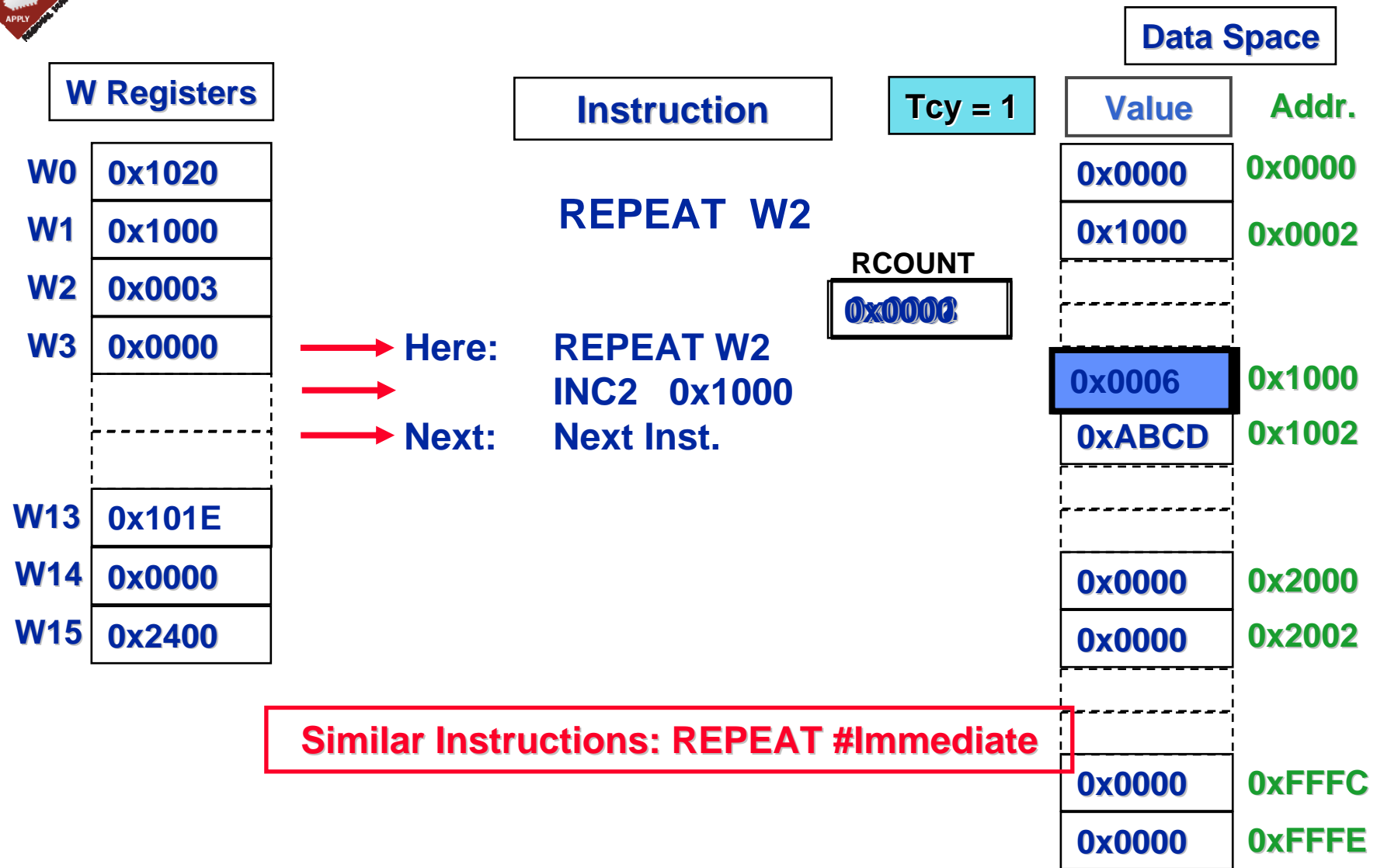


# Program Flow Instructions





# Program Flow Instructions





# Instruction Set Functional Groups

- **16-bit Instructions**

- **Move Instructions**
- **Bit Instructions**
  - Follow along LAB 1
- **Math and Logic Instructions**
- **Stack Control Instructions**
- **Program Flow Control Instructions**
- **CPU Control Instructions**
  - LAB 2



# Control Operations

- **CLRWDT –**
  - Clears the Watch Dog Timer
- **DISI #Lit14 –**
  - Disables Interrupts for 14-bit Literal value \*Tcyc
- **PWRSAV**
  - Use to put CPU in SLEEP or IDLE States
- **RESET**
  - Software Device Reset
- **NOP**



# Session Agenda

- 16-bit Architecture Basics
- Some 16-bit Assembly Instructions
- I/O Port Handling
  - **Hands-on Lab #1 Follow along**
    - Basic Assembly language setup
    - Light LED on I/O Port
- Complete 16-bit Assembly Instructions
  - ● **Hands-on Lab #2**
    - Use Loop Instructions to Blink LED
- Addressing Modes
- Interrupts and Timers
  - **Hands-on Lab #3**
    - Use Interrupt to Blink LED



# HANDS-ON

# Training

## LAB 2

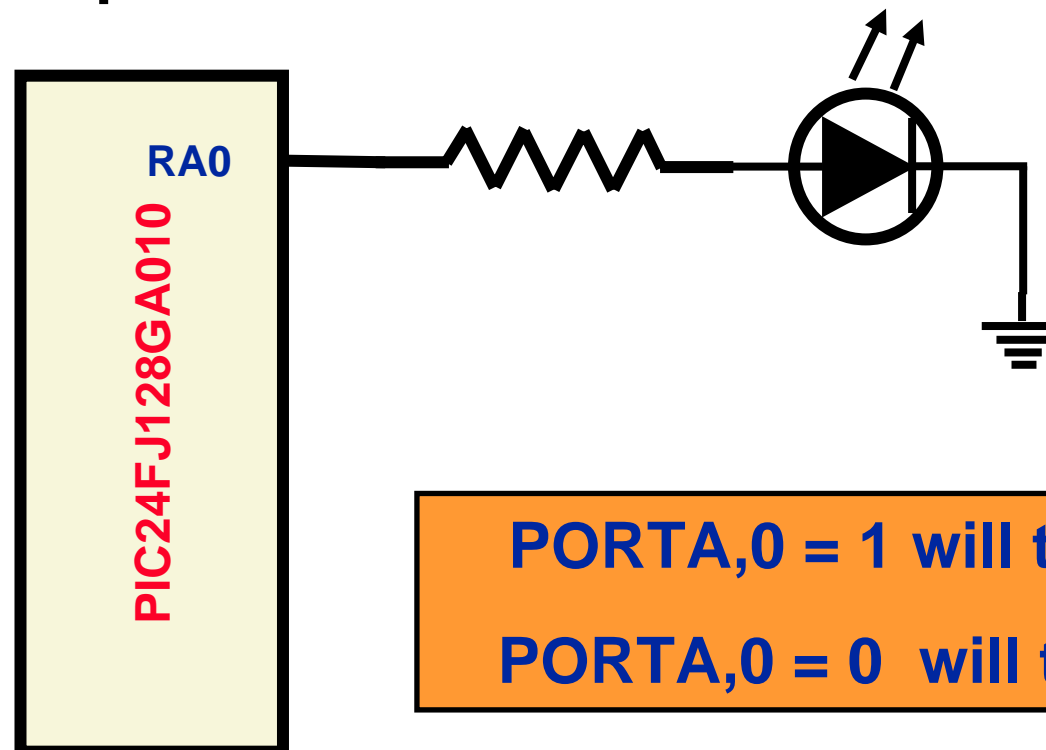
### Blink LED Using a Software Loop





# Lab 2: The Task

- Turn On LED at RA0
- Wait for 0.5 seconds using Software Delay
- Turn OFF LED at RA0
- Wait for 0.5 Seconds using Software Delay
- Repeat



**PORTA,0 = 1 will turn on LED**  
**PORTA,0 = 0 will turn off LED**



# LAB2

- **Open Lab2.mcw workspace in MPLAB**
  - C:\rtc\103\_ASP\Lab2.mcw
- **Edit code:**
  - Write a Software Loop Delay of about 0.5 Second
  - Turn ON LED
  - Wait 0.5 Second
  - Turn OFF LED, delay half second and Repeat
- **Compile Code**
- **Program Part on Explorer-16 using ICD2**
- **Run Code using ICD2**
- **See the LED **BLINK!****



# LAB 2 Clock Speed Hint

- **Use Primary Oscillator Mode, HS Osc**
- **$XT\ Xtal = 8.00\ Mhz = F_{osc}$**
- **$FCY = 4000000\ (F_{osc}/2)$**
- **Instruction Speed = 4 Mhz**
- **So  $T_{cy} = 250\ nS$**
- **0.5 Sec Software Delay = 2Million Instructions**
- **Use w2 register as 16-bit counter**
  - $30\ T_{cy} * 65536(16\text{-bit}) = 1.967\ \text{Million Instructions}$
  - Use Repeat Instruction to get the 30 Instructions



# LAB2 Summary

- **Wrote some more 16-bit PIC code**
- **Used some more Assembly Language Inst.**
  - Branch
  - Repeat Instruction
  - Loop Instructions
- **Got the Program to WORK!!**





# Session Agenda

- 16-bit Architecture Basics
- Some 16-bit Assembly Instructions
- I/O Port Handling
  - **Hands-on Lab #1 Follow along**
    - Basic Assembly language setup
    - Light LED on I/O Port
- Complete 16-bit Assembly Instructions
  - **Hands-on Lab #2**
    - Use Loop Instructions to Blink LED
- — **Addressing Modes**
- Interrupts and Timer1
  - **Hands-on Lab #3**
    - Use Interrupt to Blink LED

# HANDS-ON

# Training

## Addressing Modes





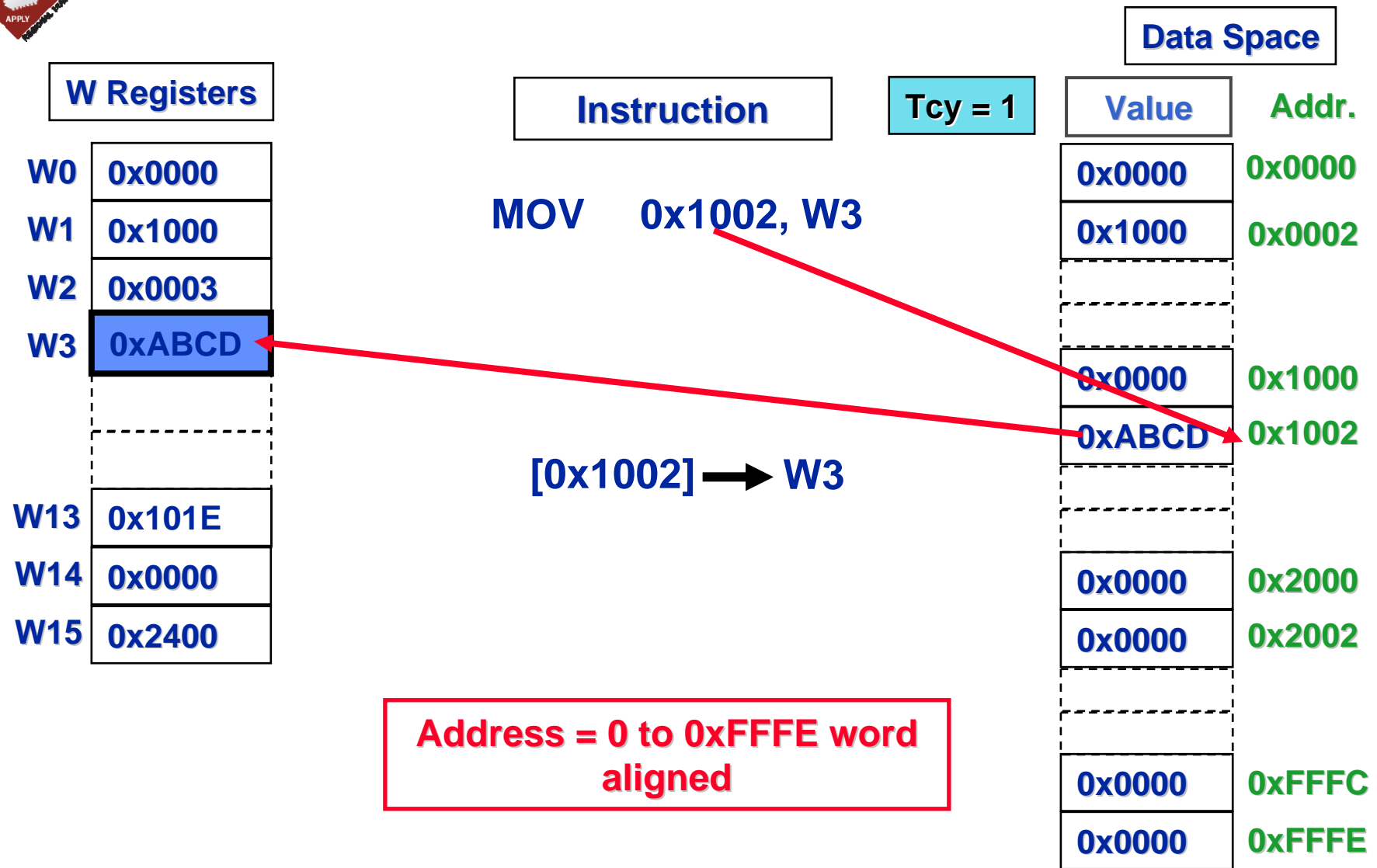
# Basic Addressing Modes

- File Register / Memory Direct
  - Register Direct
  - Register Indirect with optional pre- or post-modification
  - Immediate
  - Register indirect with register or literal offset





# MOV Instructions





# Basic Addressing Modes

- **Basic addressing modes:**

- File Register / Memory Direct

- **Register Direct**

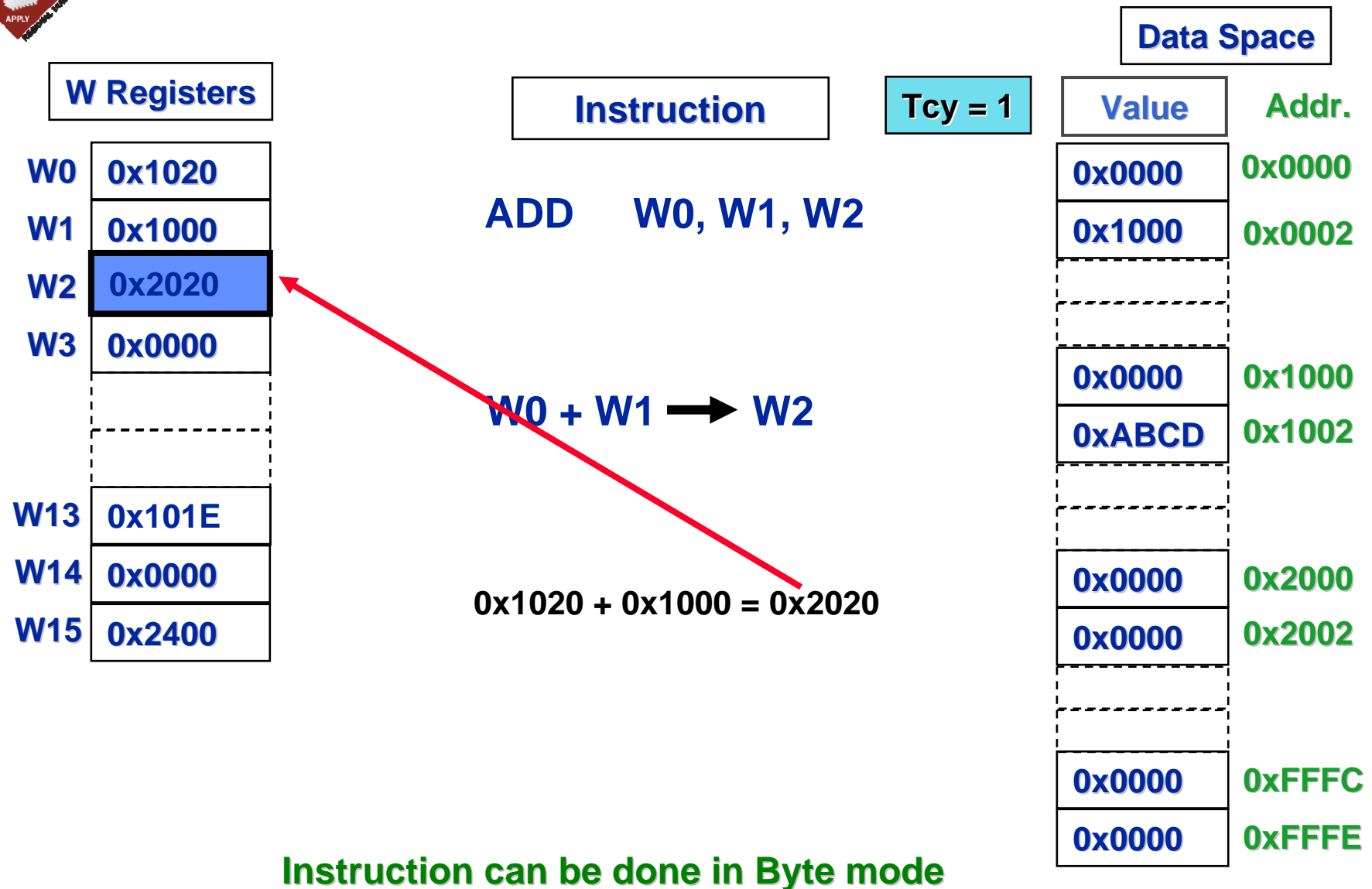
- Register Indirect with optional pre- or post-modification

- Immediate

- Register indirect with register or literal offset



# Register Direct



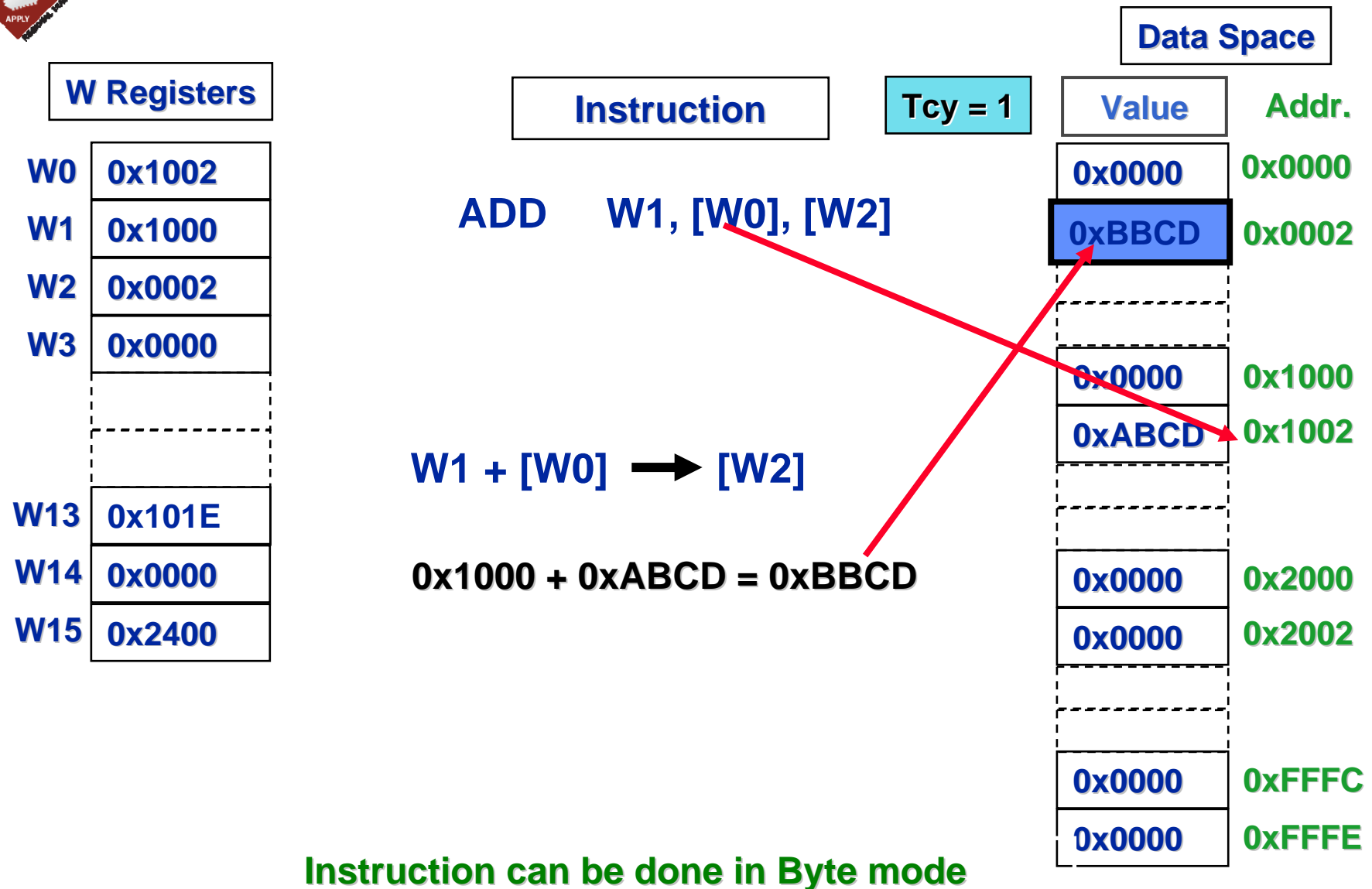


# Basic Addressing Modes

- **Basic addressing modes:**
  - File Register / Memory Direct
  - Register Direct
  - Register Indirect with optional pre- or post-modification
  - Immediate
  - Register indirect with register or literal offset

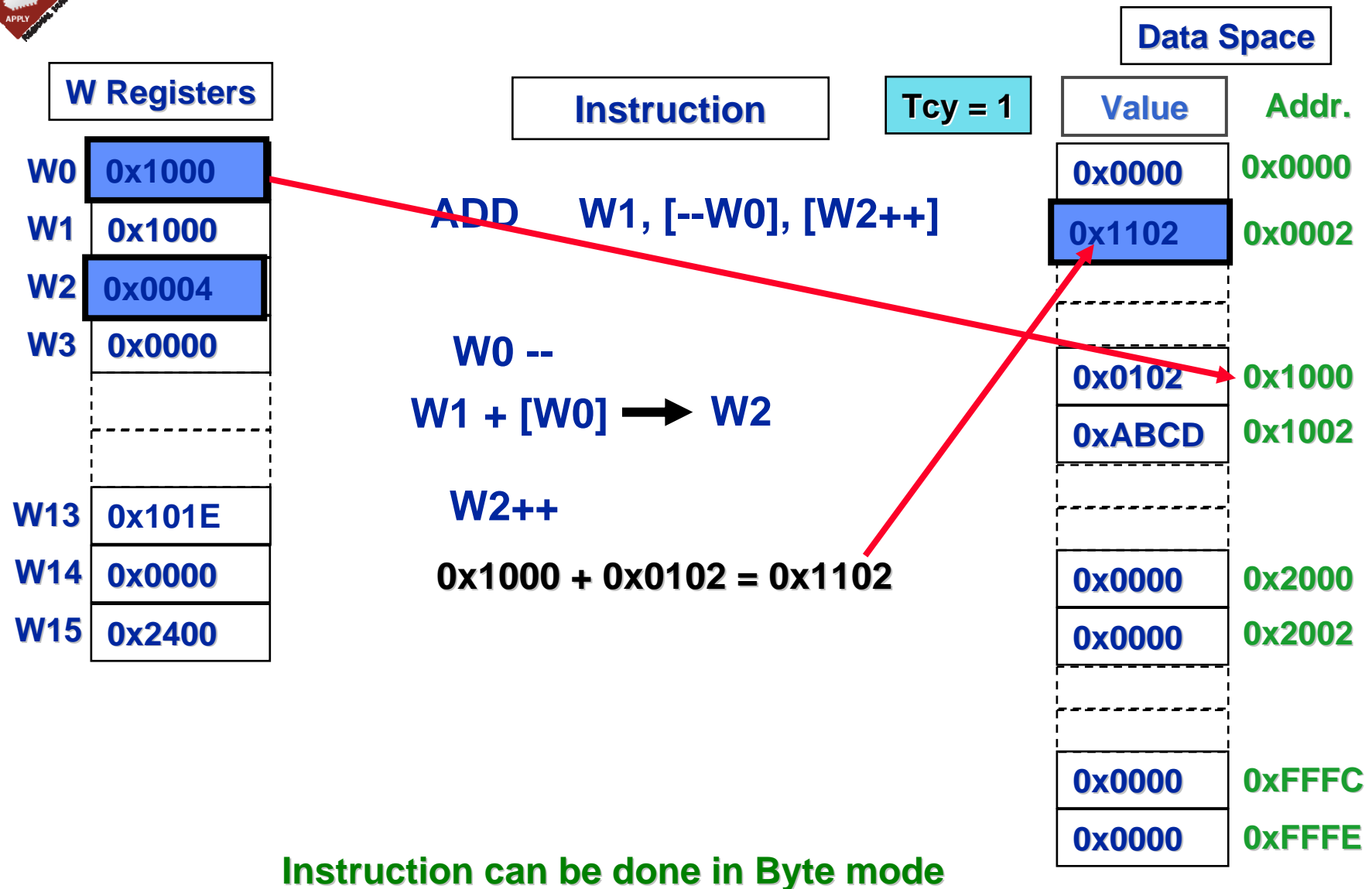


# Register InDirect





# Register InDirect





# Register Indirect Examples

- **With pre/post increment**
  - `MOV [++W4], [W6++]`
- **With pre/post decrement**
  - `MOV [W4--], [W6--]`
- **With register offset**
  - `MOV [W4 + W5], [W6]`
- **With literal offset**
  - `MOV [W4 + #768], [W6]`



# Basic Addressing Modes

- **Basic addressing modes:**

- File Register / Memory Direct
- Register Direct
- Register Indirect with optional pre- or post-modification

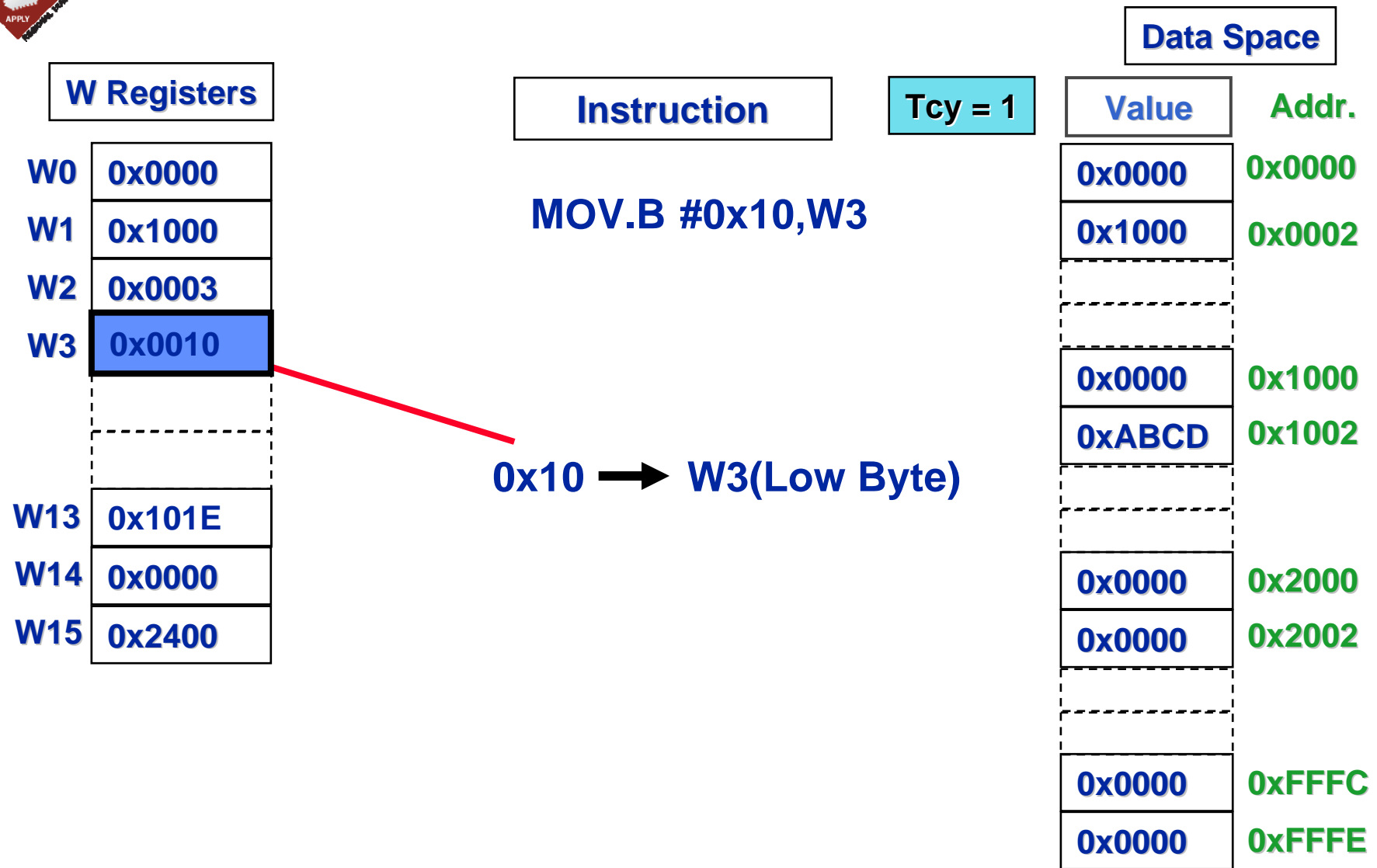
→ **Immediate**

- Register indirect with register or literal offset





# MOV Instructions





# Basic Addressing Modes

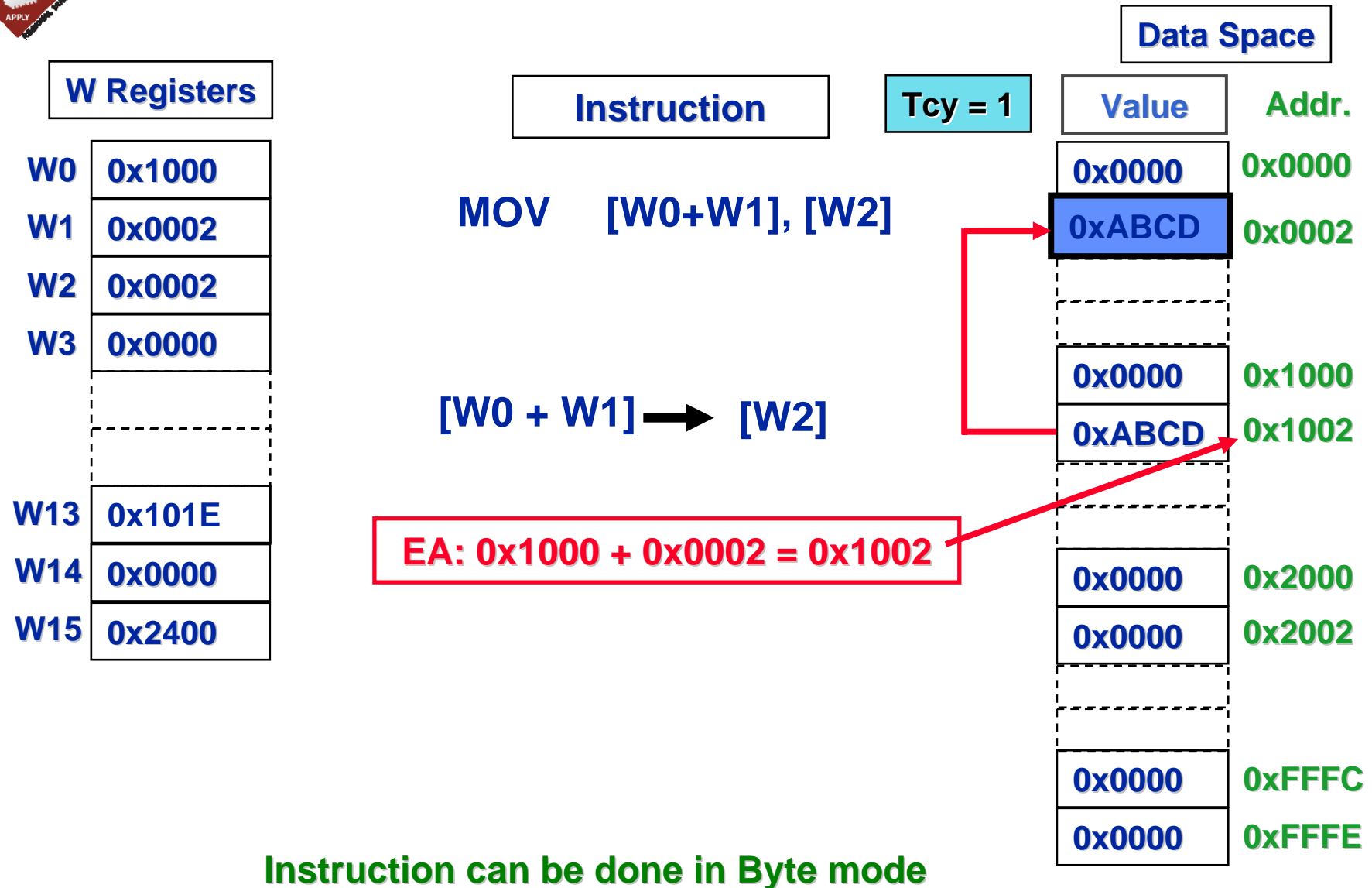
- **Basic addressing modes:**

- File Register / Memory Direct
- Register Direct
- Register Indirect with optional pre- or post-modification
- Immediate

→ Register indirect with register or literal offset



# Register InDirect with Offset





# Session Agenda

- 16-bit Architecture Basics
- Some 16-bit Assembly Instructions
- I/O Port Handling
  - **Hands-on Lab #1 Follow along**
    - Basic Assembly language setup
    - Light LED on I/O Port
- Complete 16-bit Assembly Instructions
  - **Hands-on Lab #2**
    - Use Loop Instructions to Blink LED
- Addressing Modes
- — **Interrupts and Timer1**
  - **Hands-on Lab #3**
    - Use Interrupt to Blink LED

# HANDS-ON

# Training

## Interrupt Control Module





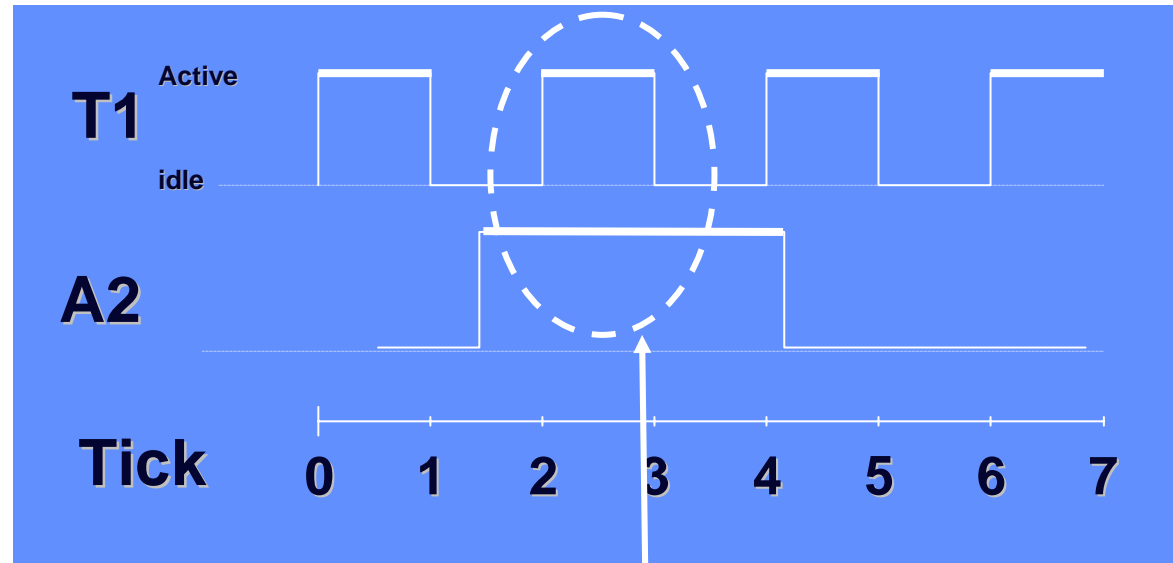


# Interrupt Driven System

```
main
{
  while(1);
}
```

```
T1_ISR
{
}
```

```
A2_ISR
{
}
```



**Two Interrupts are active**

	Interrupt Rate	Interrupt Execution Time
Timer1	50us (20khz)	25uS
A2	Async	65uS



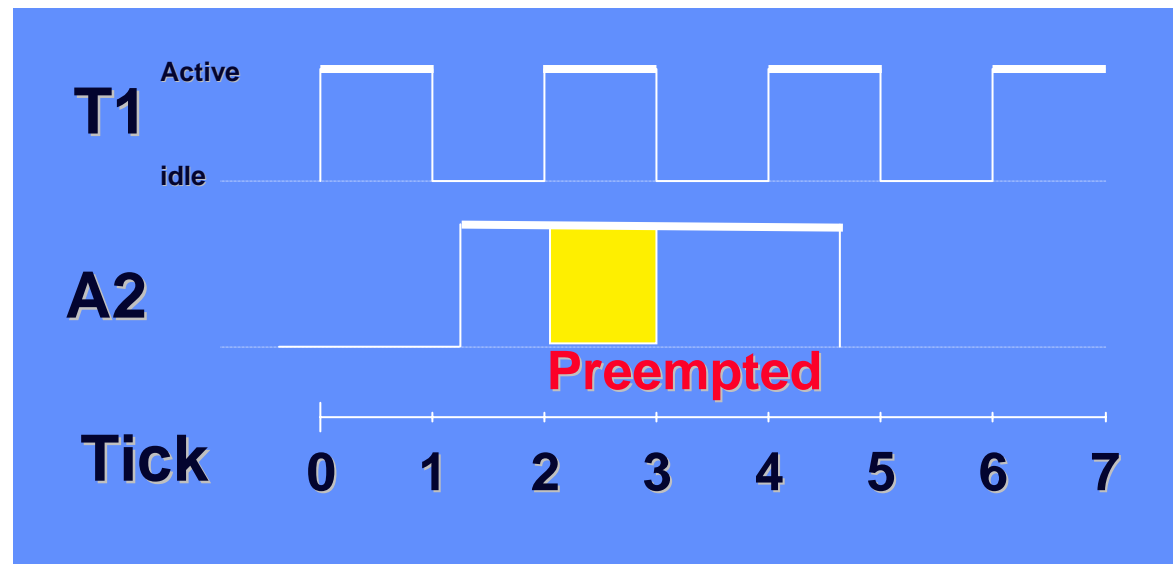
# Interrupt Nesting

- In 16-bit Arch T1 assigned a higher Priority than A2
- When T1 occurs during A2, T1 preempts A2

```
main
{
  while(1);
}
```

```
T1_ISR
{
}
```

```
A2_ISR
{
}
```





# Interrupt Vector Table

**ALTIVT = 0**

Interrupt Vector Table

**ALTIVT = 1**

Alternate Interrupt Vector Table

Reset - GOTO Instruction	0x000000
Reset - GOTO Address	0x000002
Reserved	
Oscillator Fail Trap	
Address Error Trap	
Stack Error Trap	
Arithmetic Error Trap	
Reserved	
Reserved	
Reserved	
Interrupt Vector 0	
Interrupt Vector 1	
Interrupt Vector 2	
•	
•	
Interrupt Vector N	
Reserved	
Reserved	
Reserved	
Oscillator Fail Trap	
•	
•	
•	
•	
Interrupt Vector N	

**Hardware Traps**

**Decreasing Natural Order Priority**

**User Interrupts**



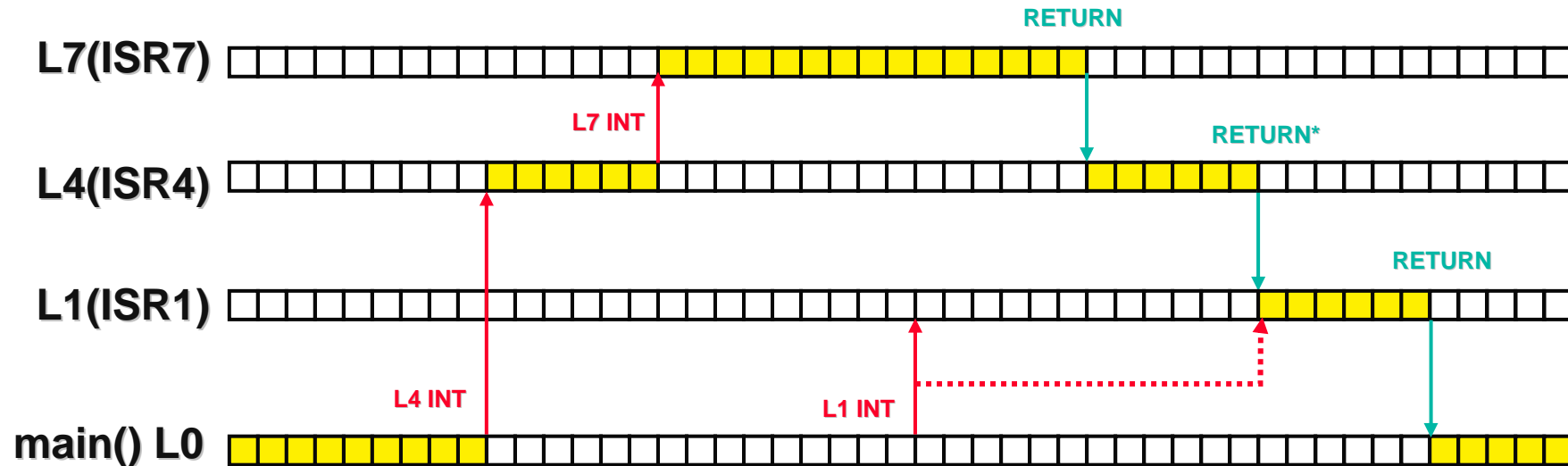


# Interrupt Priority

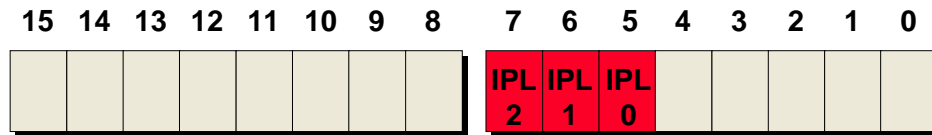
- CPU has 16 priority levels
  - **Level 0 is the default CPU level (main)**
  - **Level 1 - 7 for user interrupts**
  - **Level 8 -15 reserved for traps (NMI)**
- Interrupt with priority level greater than current CPU level ( $IPL<3:0>$ ) can interrupt the CPU
- Interrupts are nested by default, Nesting can be disabled by setting NSTDIS bit in INTCON1 register
- IVT has natural priority to resolve conflicts
- User assigned priority overrides natural priority



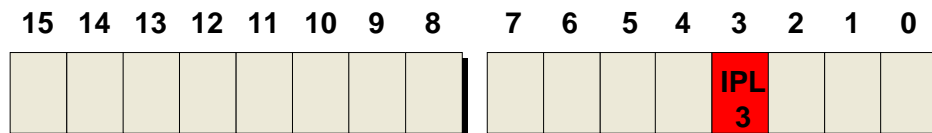
# Interrupt Nesting Example



 CPU EXECUTION TRACE



Status Register



Core Control Register



# Disabling Interrupts

- **DISI Instruction**

- DISI #N instruction disables level 1 - 6 interrupts for (N+1) Instruction cycle

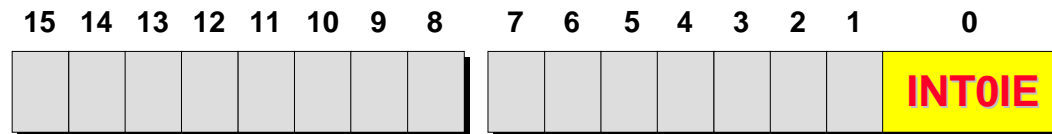
- **Write to CPU **IPL<2:0>** bits to raise CPU priority to level 7**

- MOV.B #0xE0, w0
- MOV.B WREG, SRL

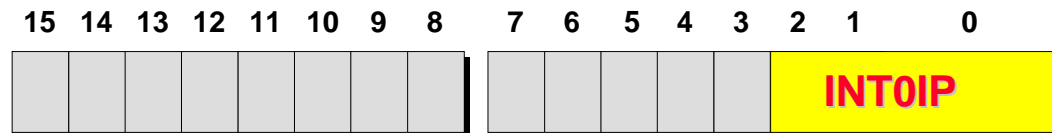


# Configuring Interrupts: Example using INT0

**IEC0: Interrupt Enable Control Register 0 : Enable Interrupt**



**IPC0: Interrupt Priority Control Register 0 : Assign Priority**



000: Disabled  
001: Priority 1 Interrupt  
010: Priority 2 Interrupt  
011: Priority 3 Interrupt  
100: Priority 4 Interrupt  
101: Priority 5 Interrupt  
110: Priority 6 Interrupt  
111: Priority 7 Interrupt

**IFS0: Interrupt Flag Control Register 0 : Clear Interrupt in the ISR**





# Traps for Robust Operation

- **Oscillator Failure Trap (level 14)**
- **Address Error Trap (level 13)**
  - Instruction fetch from illegal program space
  - Data fetch from unimplemented data space
  - Unaligned word access from data space
- **Stack Error Trap (level 12)**
  - Stack overflow or underflow
- **Math Error Trap (level 11)**
  - Divide by Zero
  - Unsaturated Accumulator Overflow (A or B)
  - Catastrophic Accumulator Overflow (either)
  - Accumulator Shift Overflow



# HANDS-ON

# Training

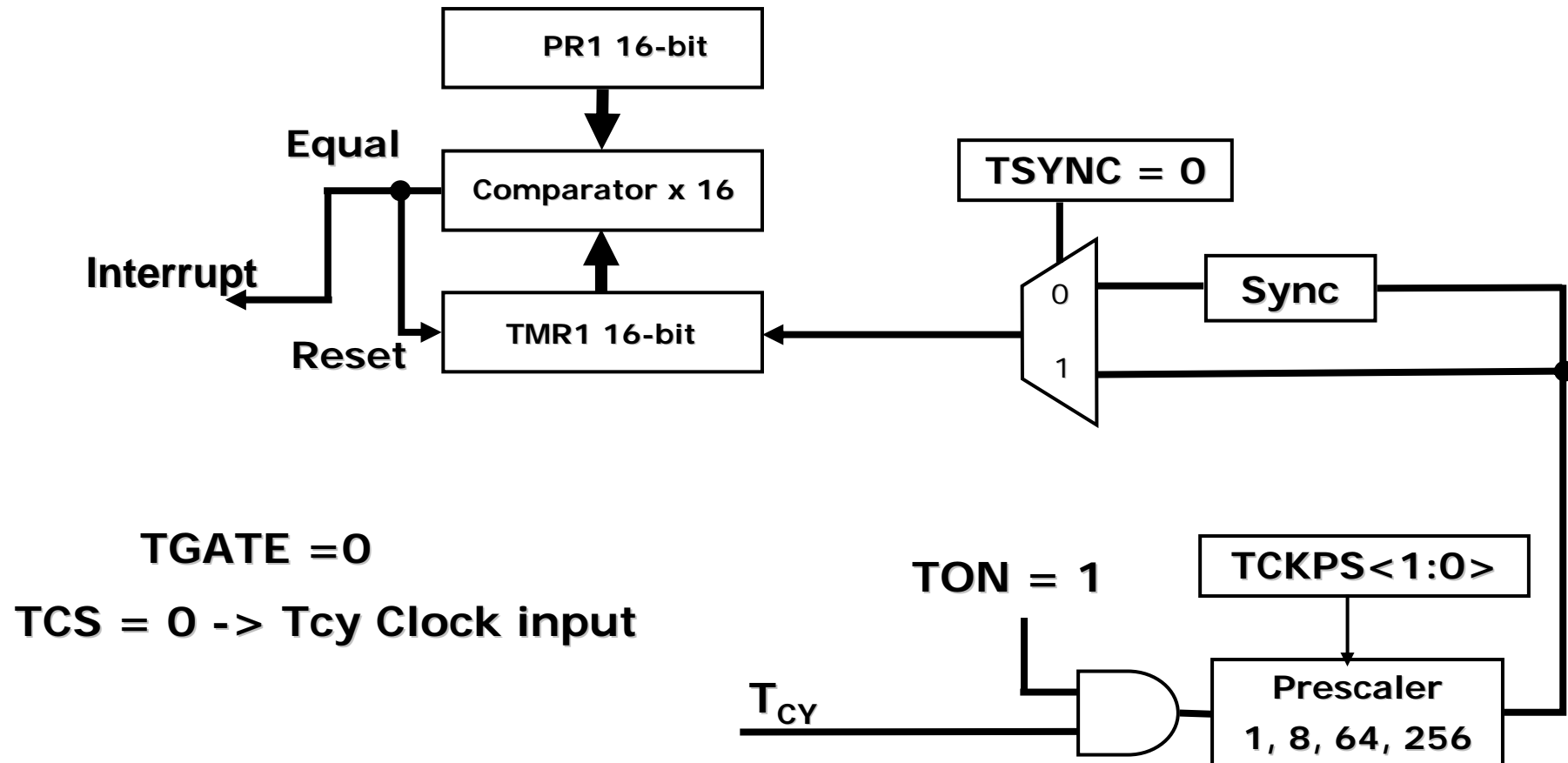
## LAB 3

### Blink LED Using Timer1 and Interrupt Routine





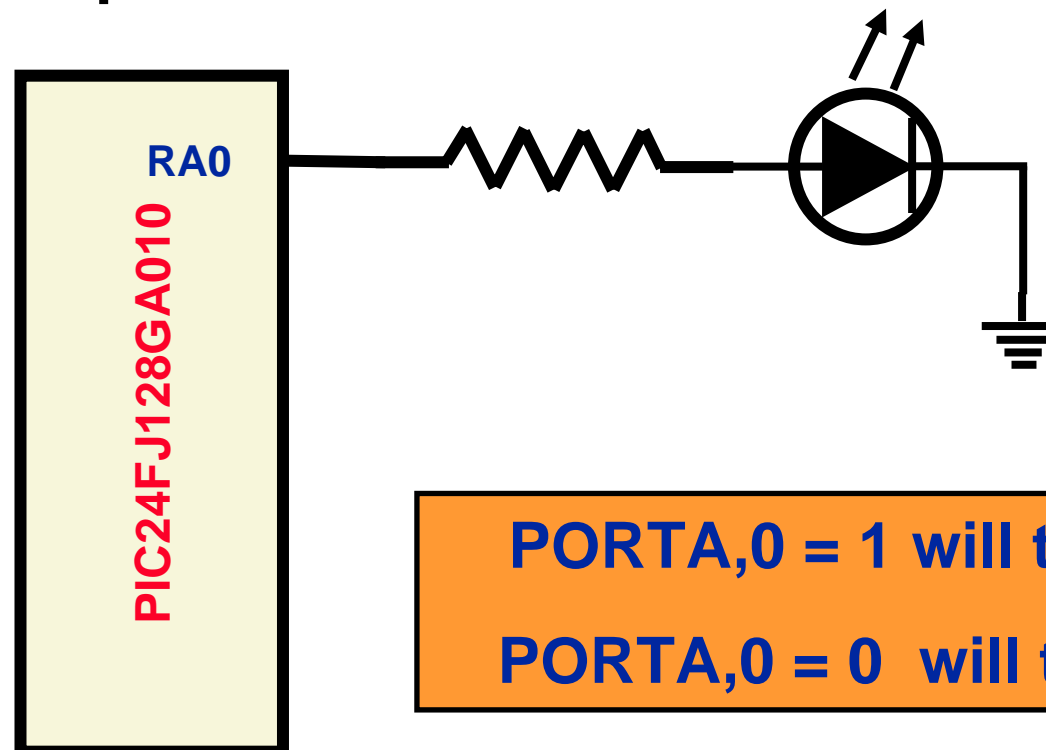
# Simplified TMR1 Diagram





# Lab 3: The Task

- Turn On LED at RA0
- Wait for 0.5 seconds using TMR1 Delay
- Turn OFF LED at RA0
- Wait for 0.5 Seconds using TMR1 Delay
- Repeat



**PORTA,0 = 1 will turn on LED**  
**PORTA,0 = 0 will turn off LED**





# Lab3 Clock Hints

- **Clock TMR1 using internal Tcy**
- **Fcy = 4Mhz**
- **Divide Fcy by 64: Set T1CON,#5**
- **For Half Second Interrupts**
  - PR1 value =  $(Fcy/64) * 0.5$
- **Turn on TMR1: Set T1CON,#TON**
- **Interrupt Service Routine Provided:**
  - Clear the interrupt Flag: T1IF in IFS0 register
  - Remember code gets here ever 0.5 Secs
  - Write code to blink the LED
- **After initialization, do nothing in main program**



# LAB3

- **Open Lab3.mcw workspace in MPLAB**
  - C:\rtc\103\_ASP\Lab3.mcw
- **Edit code:**
  - Initialize Timer 1 for a delay of 500 mS
  - In Timer1 Interrupt Service Routine toggle LED
- **Compile Code**
- **Program Part on Explorer-16 using ICD2**
- **Run Code using ICD2**
- **See the LED **BLINK!****



# LAB3 Summary

- **Learned about Timer1 Peripheral**
- **Wrote code to Initialize Timer1 Peripheral**
- **Wrote an Interrupt Service Routine**
- **Got the Program to WORK!!**

HANDS-ON

Training

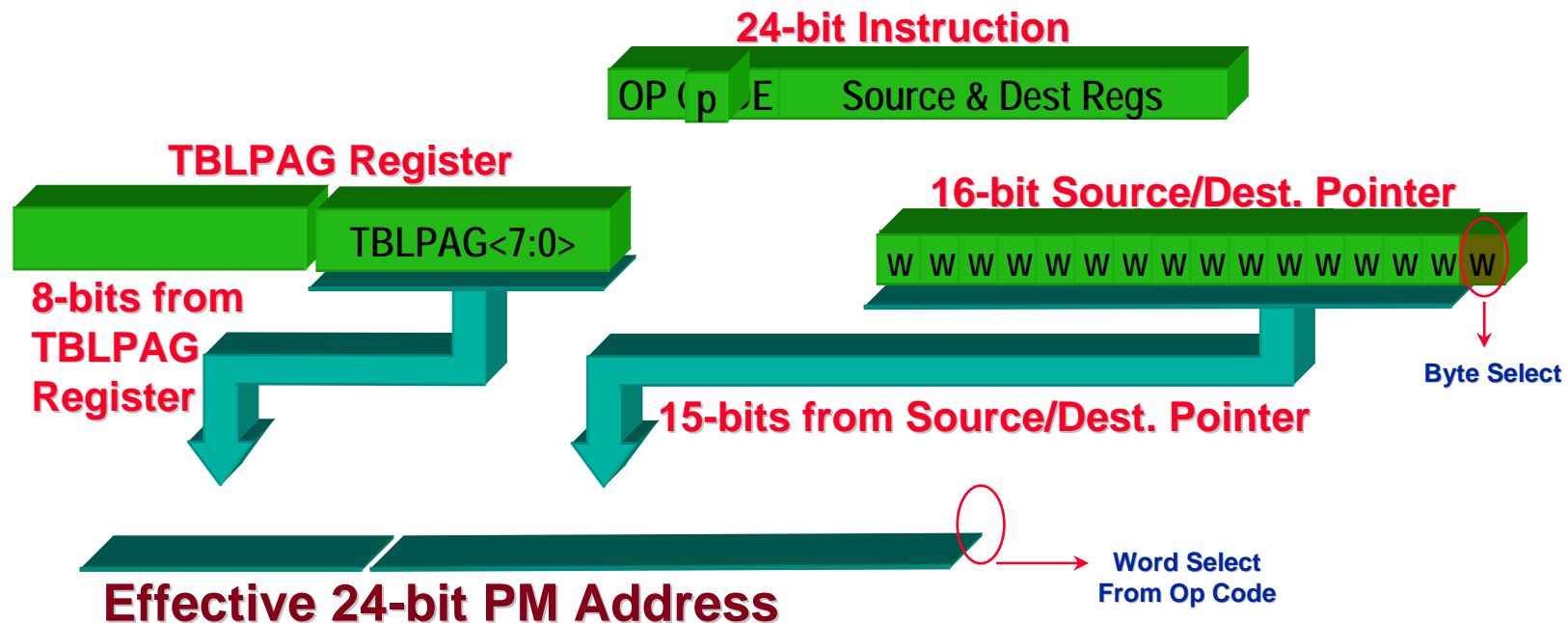
# Program Memory to Data Memory Access





# Data Access from PM Using Table Instructions

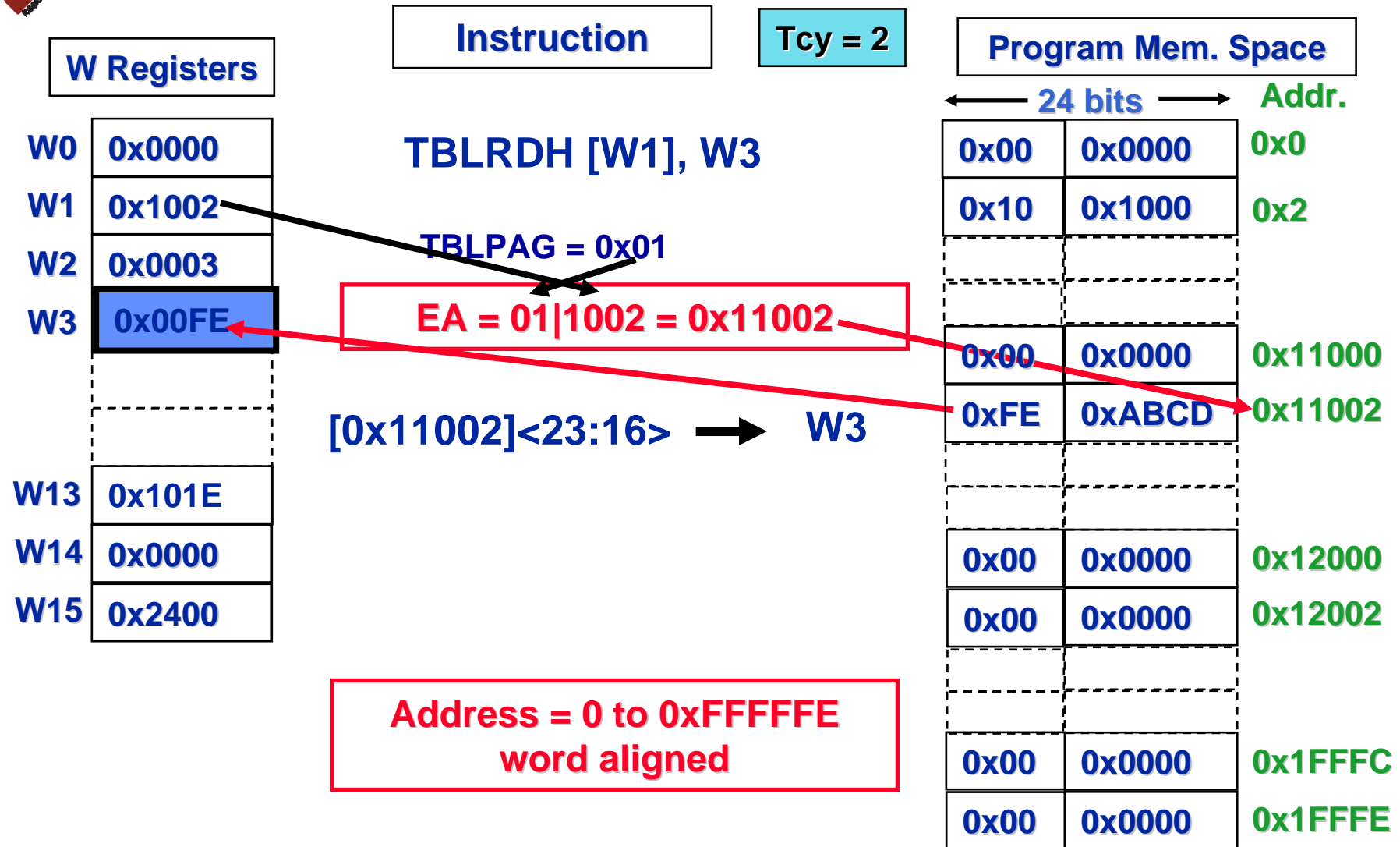
- **24-bit Effective Address(EA) formation**
  - Configuration space is accessed by setting TBLPAG<7> (i.e. EA<23>)
    - Only means of accessing configuration space





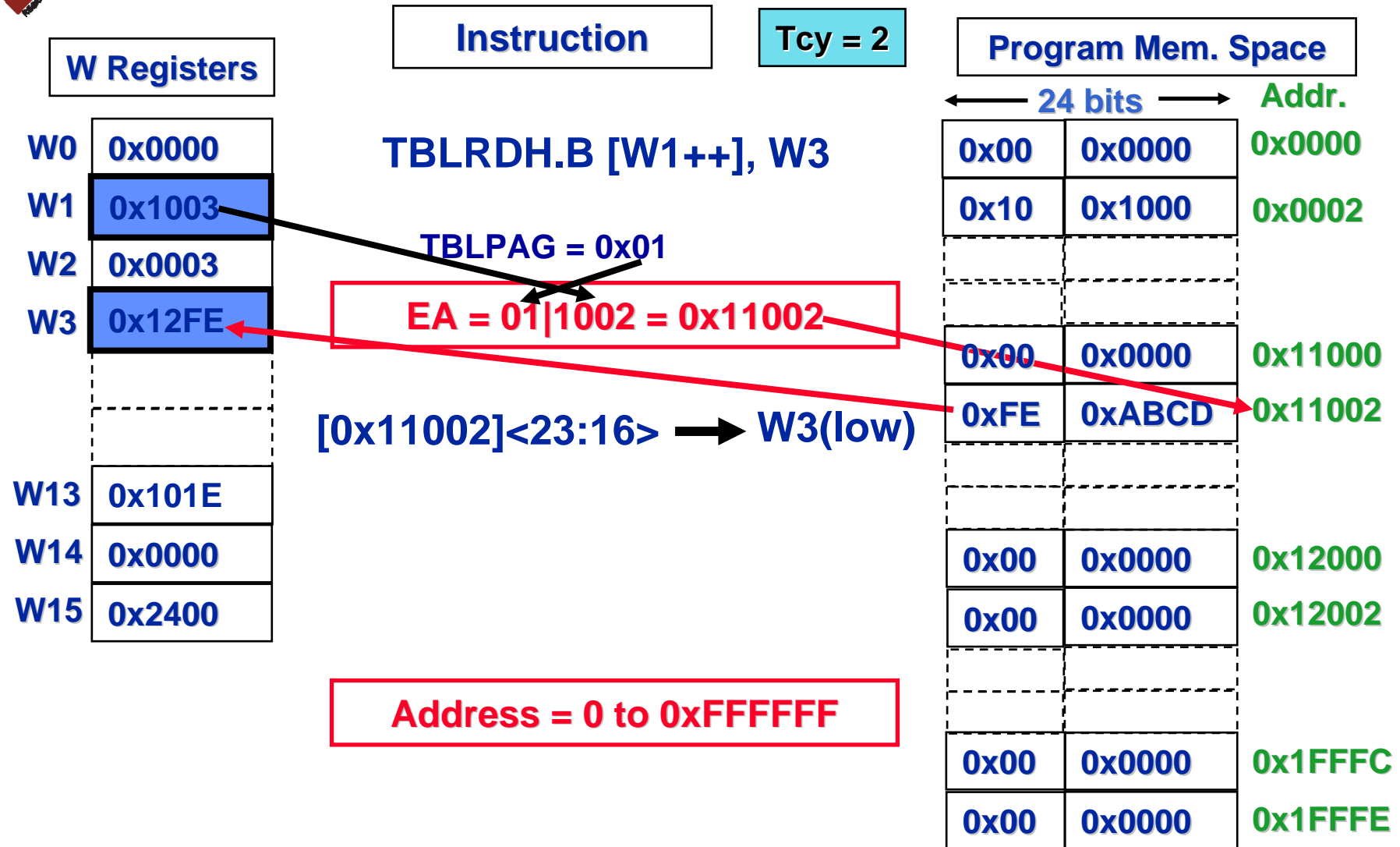


# TBLRDH Instructions



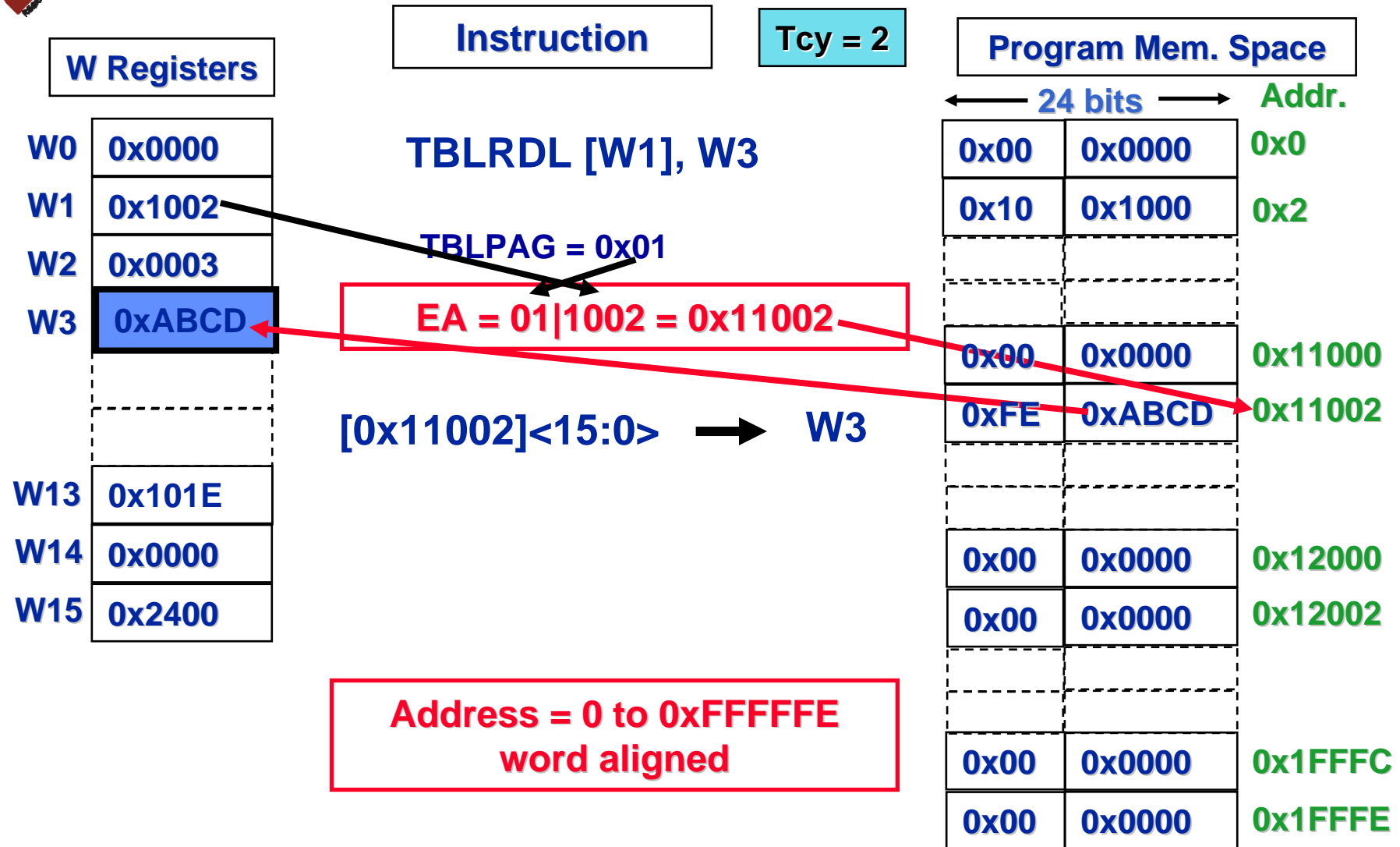


# TBLRDH Instructions





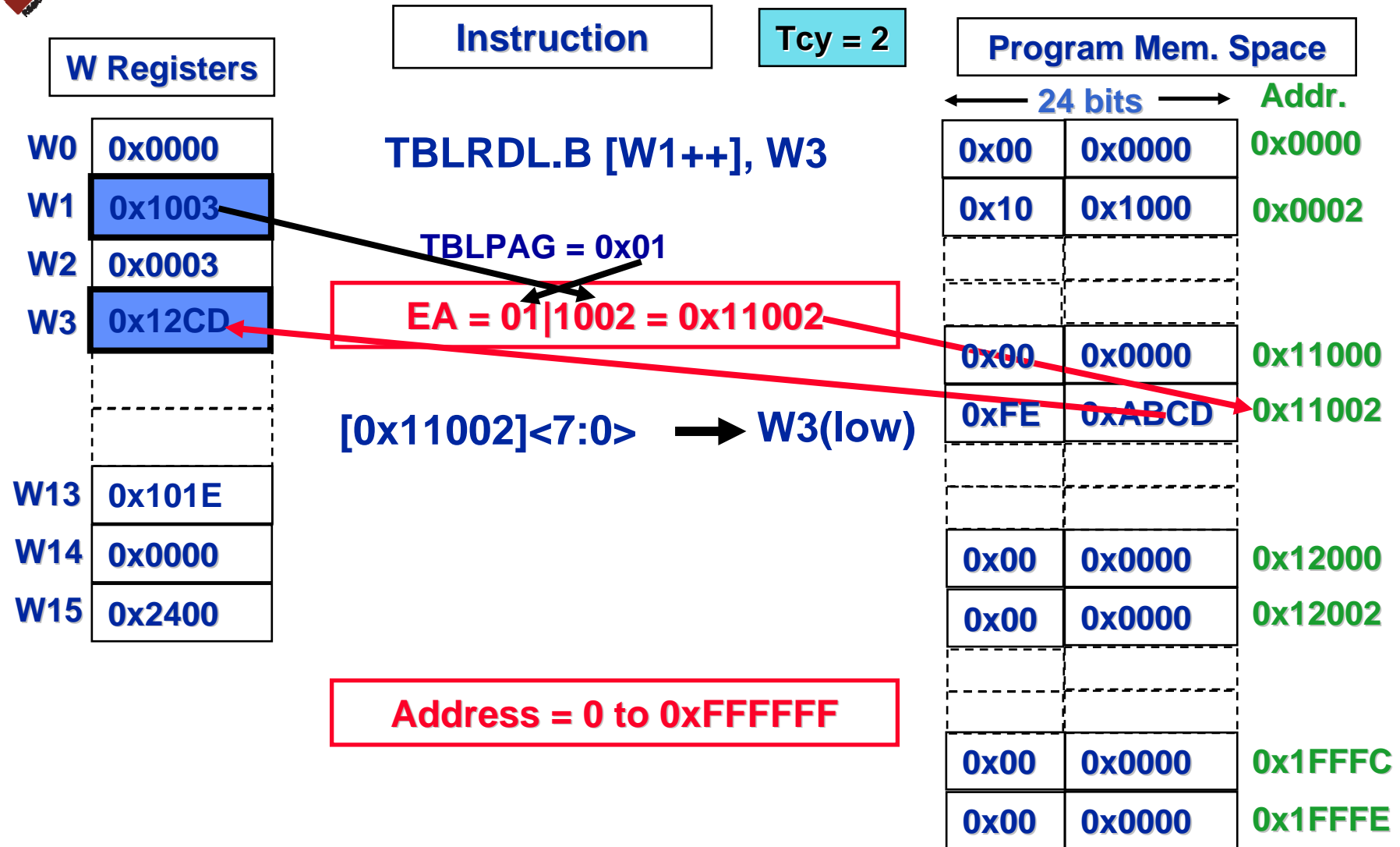
# TBLRDL Instructions







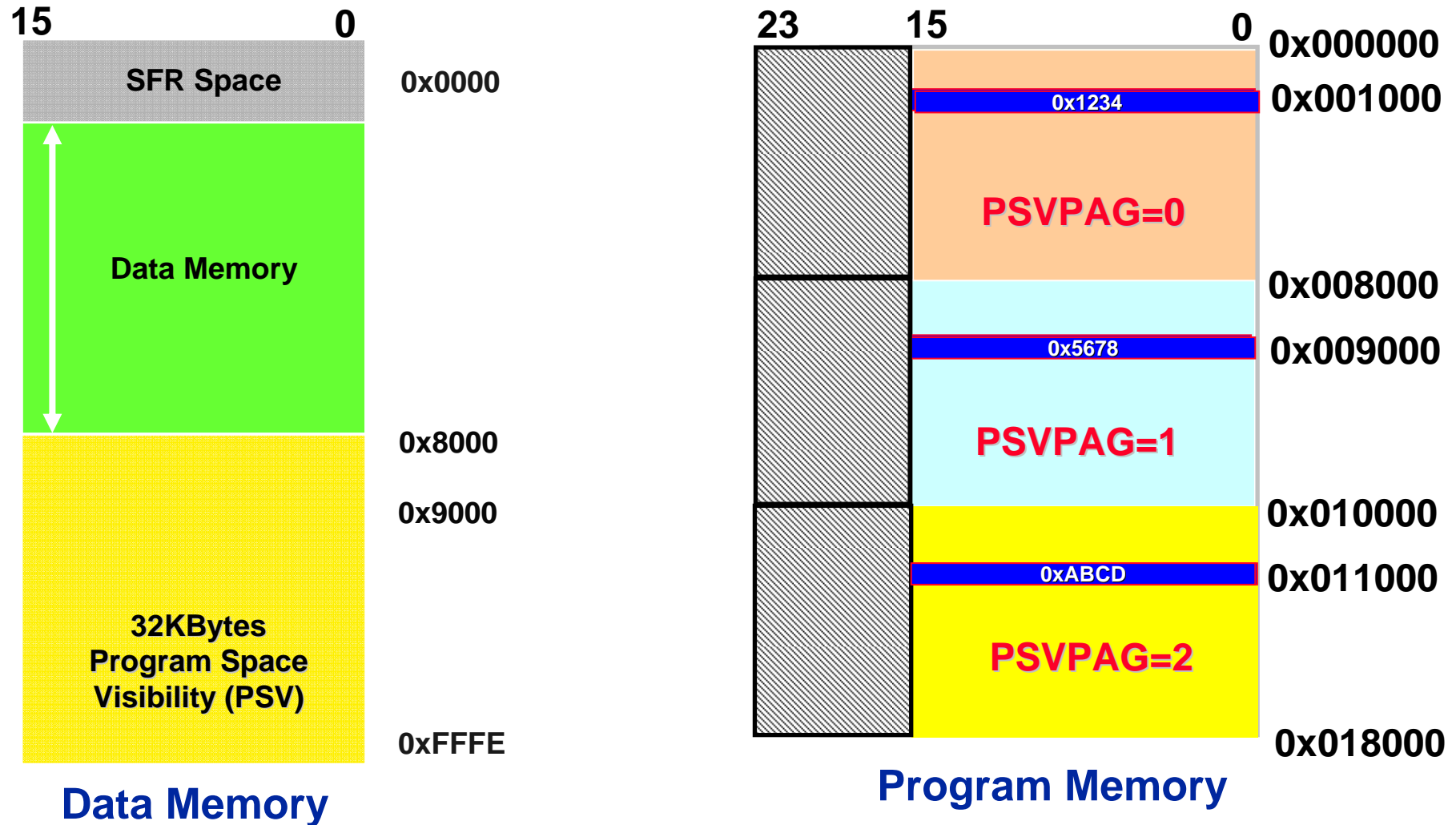
# TBLRDL Instructions





# Program Space Visibility

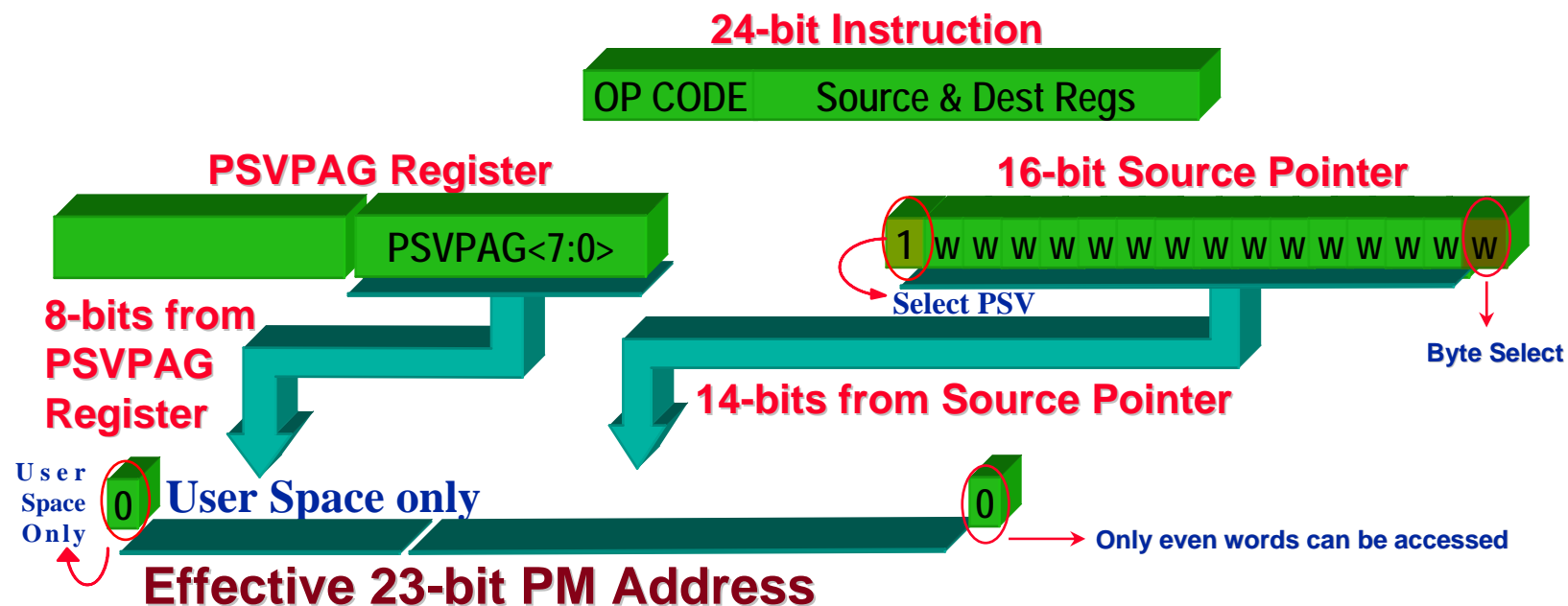
- Program Space Visibility Enabled when **CORCON<PSV> = 1**
- PSVPAG register maps 32Kb program memory segment into data memory





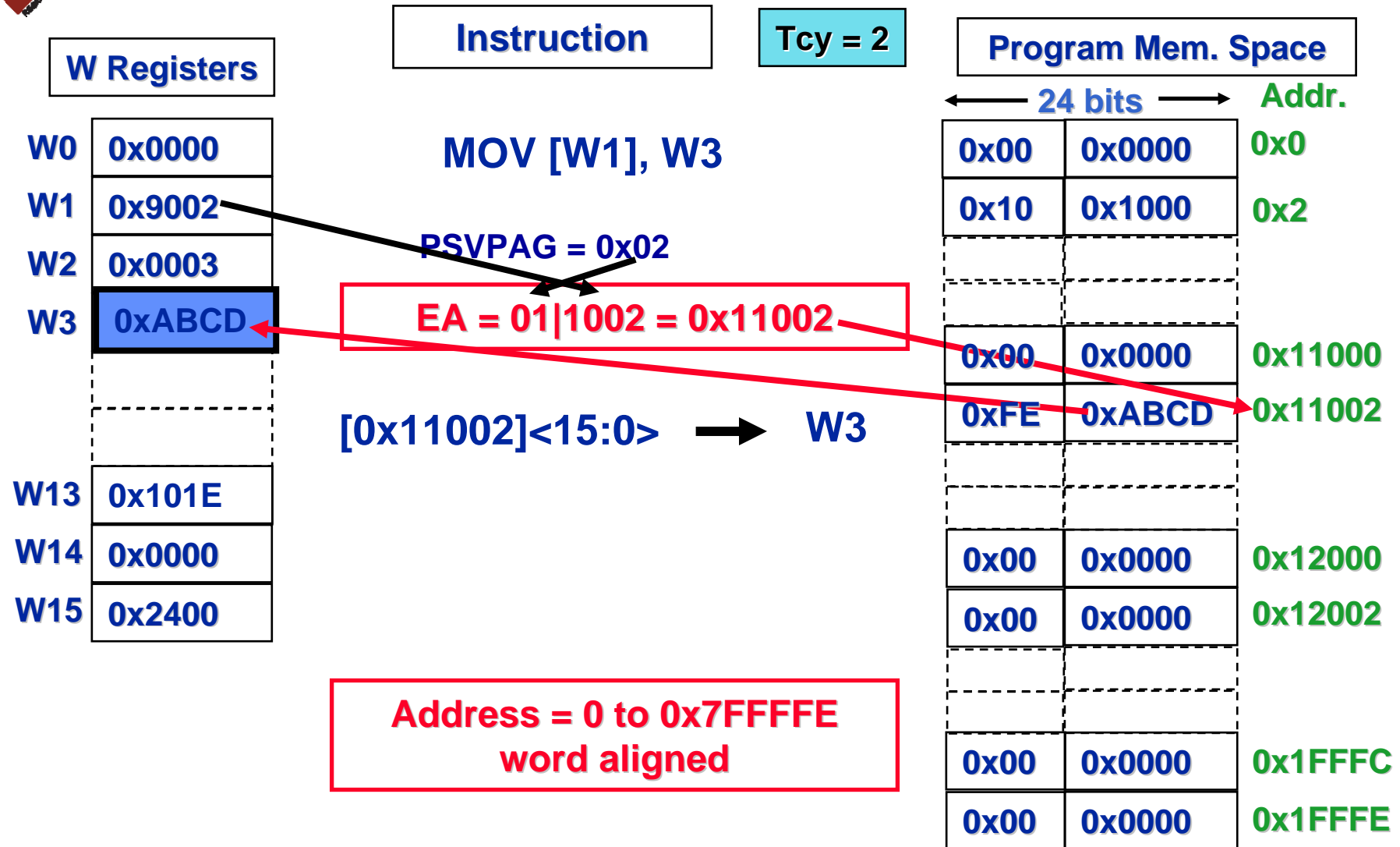
# Data Access from PM Using Program Space Visibility

- 32 KB segment of PM may be mapped into the data memory address space
  - If PSV bit (CORCON<2>) is set and if SrcPointer (DM EA)<15> is '1' then data is accessed from PM



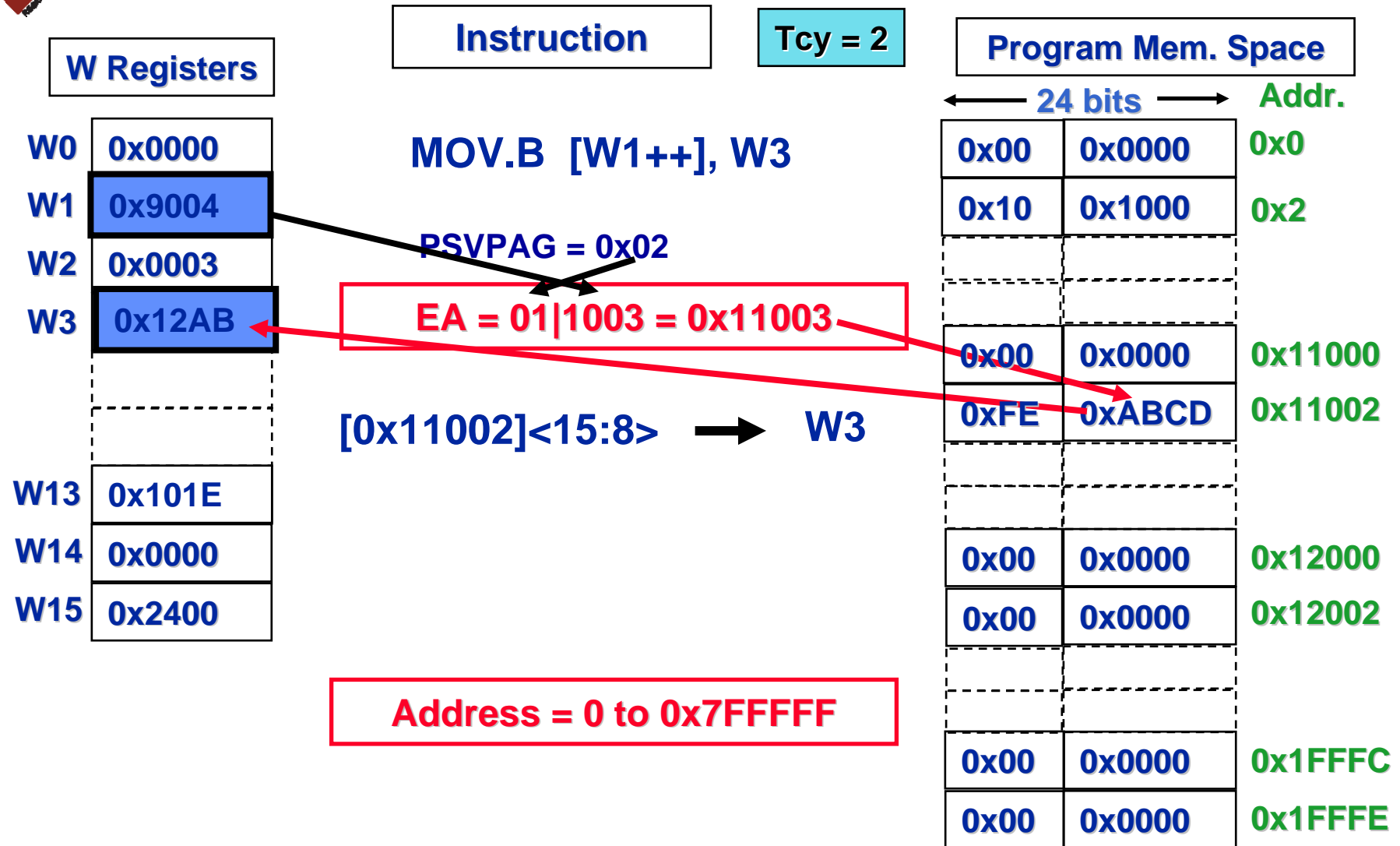


# MOV Instructions (PSV)





# MOV Instructions (PSV)





# Defining/Accessing PSV Constants

- **Defining constants in PM for PSV, in assembly**

```
.section .const, psv  
hello:  
    .ascii "Hello World!\n"
```

- **Accessing PSV in assembly**

```
mov        #psvpage(hello),w0  
mov        wo, PSVPAG      ; set PSVPAG address  
bset.b     CORCONL,#PSV    ; enable PSV operation  
mov        #psvoffset(hello),w0  
mov.b      [w0++],w1       ;get first byte
```



# Usefulness of PSV

- **PSV allows very large tables of data to be stored and accessed quickly & efficiently**
- **PSV provides a bridge to a common data/program address space (Von Neumann) but only when needed**
- **Example:**
  - Large constant data for display
  - Sine Table



# Data Access Overhead Using PSV

- **PSV data fetch overhead:**
  - Outside a REPEAT loop:
    - **Data move ops, overhead = 1 cycle**
    - **ALU based ops, overhead = 2 cycles**
  - Within a REPEAT loop:
    - **Data pipelined, so overhead for all ops = 0 cycles**
    - **First & last iteration - data pipeline fill/flush**
      - Data move ops, overhead = 1 cycle
      - ALU based ops, overhead = 2 cycles



# HANDS-ON

# Training

## LAB 4 Optional

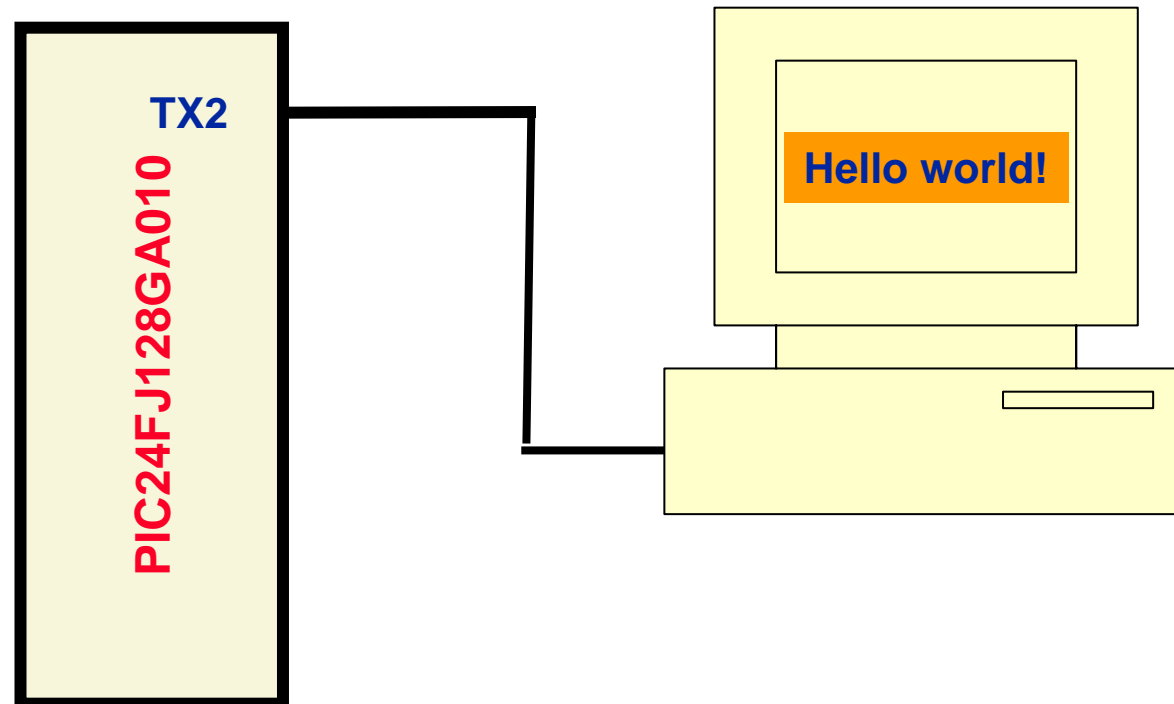
Access Data String in PM using  
PSV





# Lab 4: The Task

- Use PSV to access Data String
- Transmit to Hyper Terminal when S3 pressed
- Data String: “Hello world!”





# Hints for Lab 4

- **Defining constants in PM for PSV, in assembly**

```
.section .const, psv  
hello:  
    .ascii "Hello World!\n"
```

- **Accessing PSV in assembly**

```
mov        #psvpage(hello),w0  
mov        wo, PSVPAG      ; set PSVPAG address  
bset.b     CORCONL,#PSV    ; enable PSV operation  
mov        #psvoffset(hello),w0  
mov.b      [w0++],w1        ;get first byte  
  
mov        w1,U2TXBUF
```



# Summary

- **Learned 16-bit Architecture Basics**
- **Learned the Instruction Set**
- **Wrote three programs using the Assembly Language**
- **Brief overview of the Interrupts and Timer1 peripheral**
- **Learned about Program Memory Access**
- **Ready to take the advanced 16-bit Programming Classes**



# References

- **[www.microchip.com/16bit](http://www.microchip.com/16bit)**
- **dsPIC30F Programmer's Reference Manual – DS70030F**
- **dsPIC30F Family Reference Manual – DS70064C**



# HANDS-ON

# Training

# Thank You

