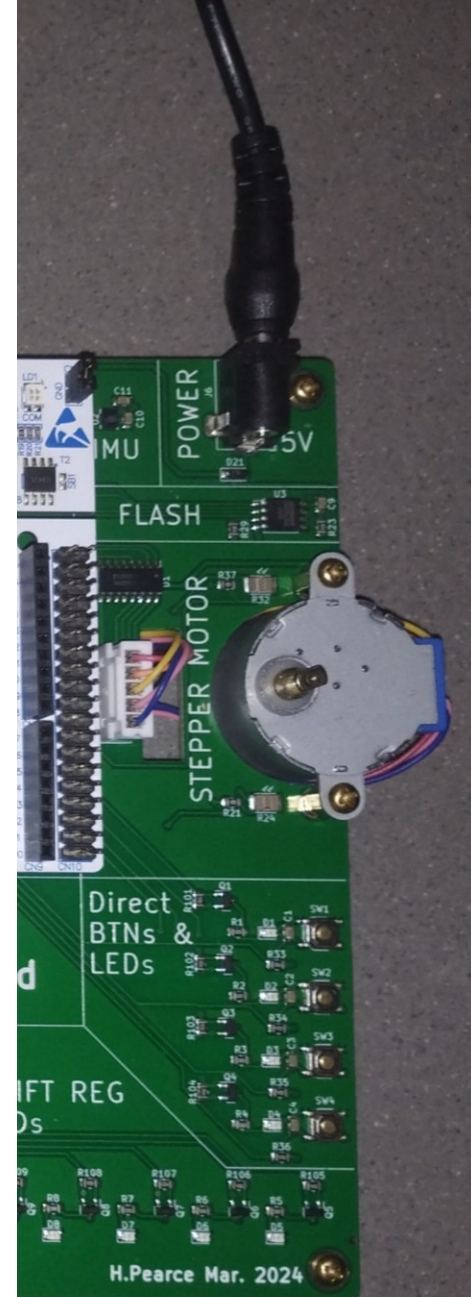


DESN2000
(Computer
Engineering)
2026 T2

Volatile Qualifier in C

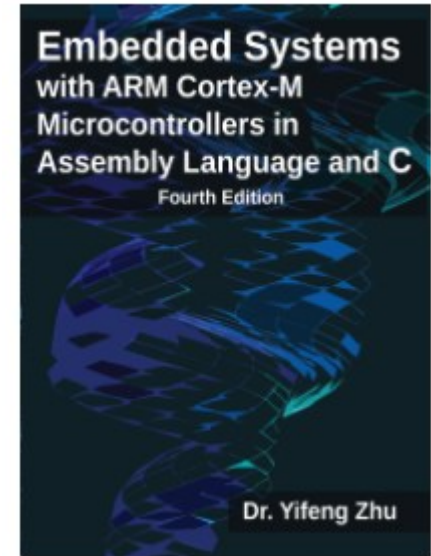
Hasindu Gamaarachchi



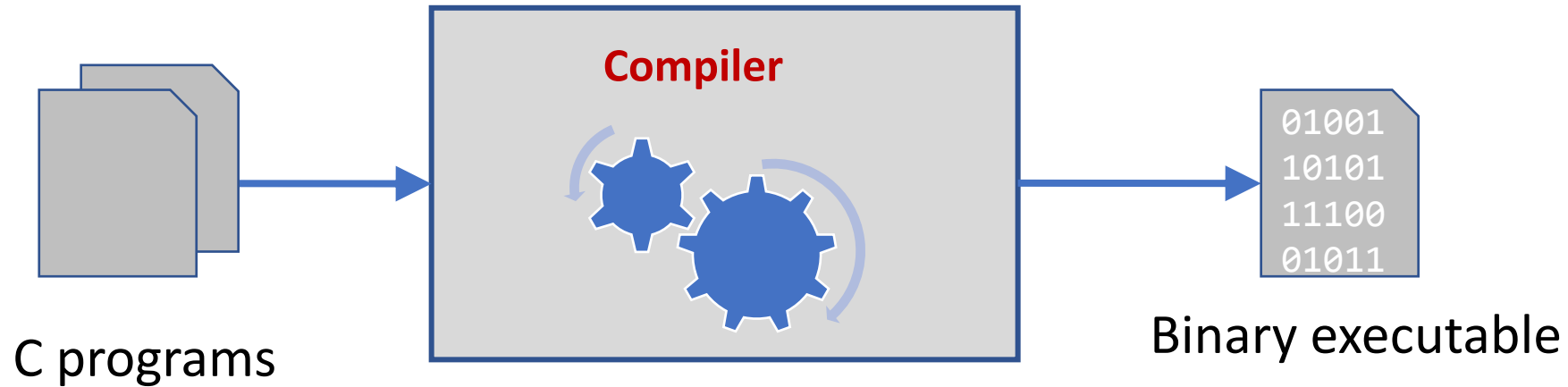
Learning Resources

The upcoming slides are adapted from “Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C (Fourth Edition)” – Yifeng Zhu

- Mixing C and Assembly - chapter 10



Compiler



Performance, Performance, Performance!

- Compiler can make aggressive optimization
 - Cache results in register to reduce memory accesses
 - Reorder computations
 - Eliminate useless and redundant computations

```
int flag = 0;

void SysTick_Handler(void){
    flag = 1;
    // Turn on the green LED
    GPIOE->ODR |= GPIO_ODR_ODR_8;
}

int main(void){
    GPIO_init();
    SysTick_init();

    while(flag == 0);

    // Turn on the red LED
    GPIOB->ODR |= GPIO_ODR_ODR_2;

    while(1);
}
```

Without
optimization

```
...
LDR r0, =flag ; Get memory address
loop LDR r1, [r0] ; Read memory
CMP r1, #0
BEQ loop
...
```

With
optimization

```
...
LDR r0, =flag ; Get memory address
LDR r1, [r0] ; Read memory
loop CMP r1, #0
BEQ loop
...
```

A dead loop since flag was 0. Thus, the program is blocked here and failed to turn on the red LED.

Solution

```
volatile int flag = 0;

void SysTick_Handler(void){
    flag = 1;
    // Turn on the green LED
    GPIOE->ODR |= GPIO_ODR_ODR_8;
}

int main(void){
    GPIO_init();
    SysTick_init();

    while(flag == 0);

    // Turn on the red LED
    GPIOB->ODR |= GPIO_ODR_ODR_2;

    while(1);
}
```

With/without
optimization

```
...
LDR r0, =flag ; Get memory address
loop LDR r1, [r0] ; Read memory
CMP r1, #0
BEQ loop
...
```

Volatile variables

- Qualifier **volatile** tells the compiler that this value may change at any time
 - Do not apply any optimizations to remove reads or writes to these variables

Example:

```
volatile int x;  
y = x + x + x + x;
```

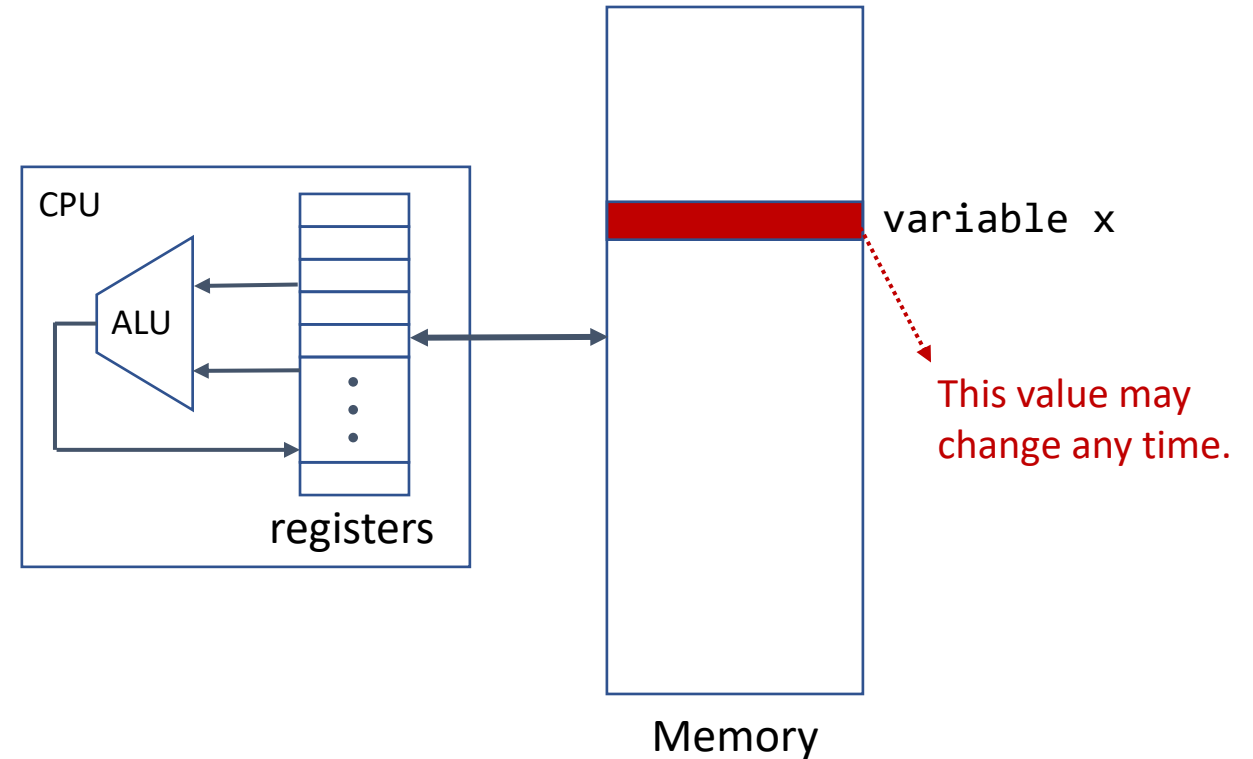
The computation cannot complete within a single clock cycle and x can change any time.

Compiler cannot optimize it as:

```
y = 4 * x;
```

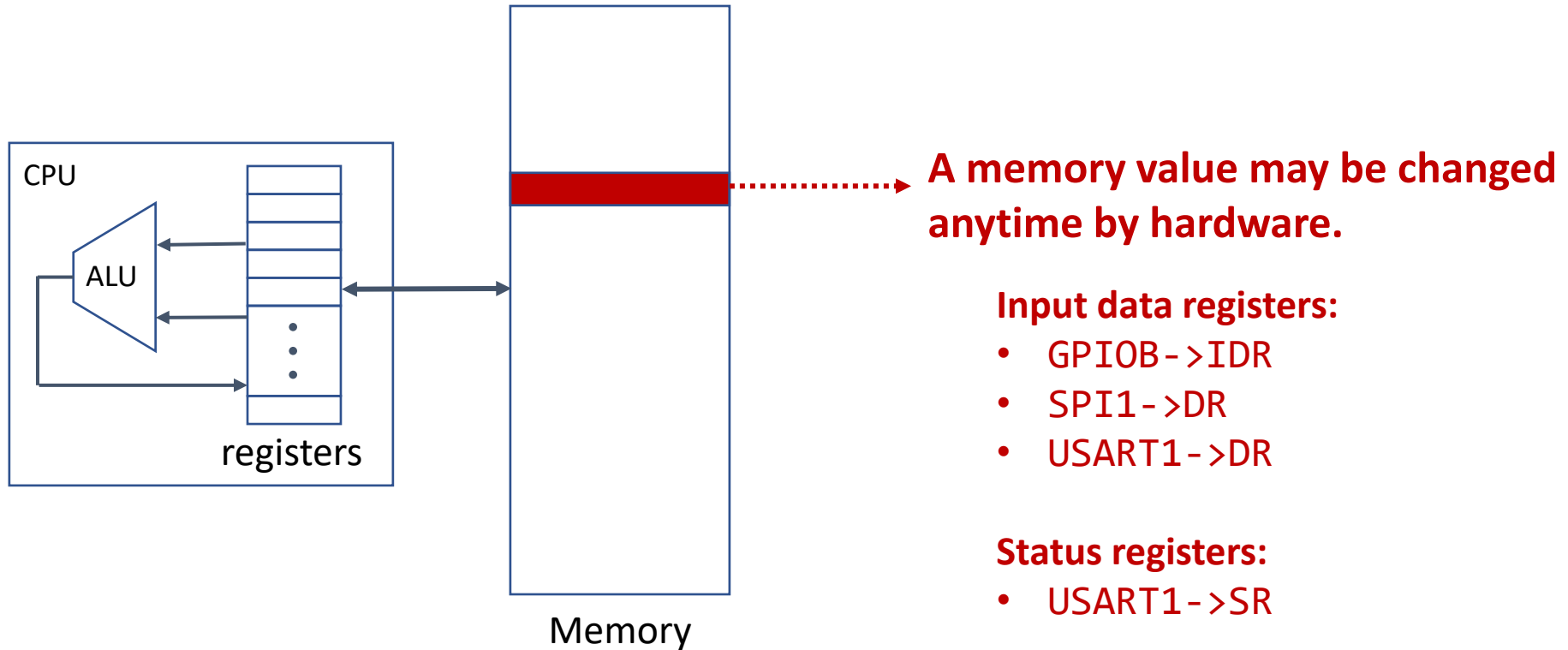
or

```
y = x << 2;
```



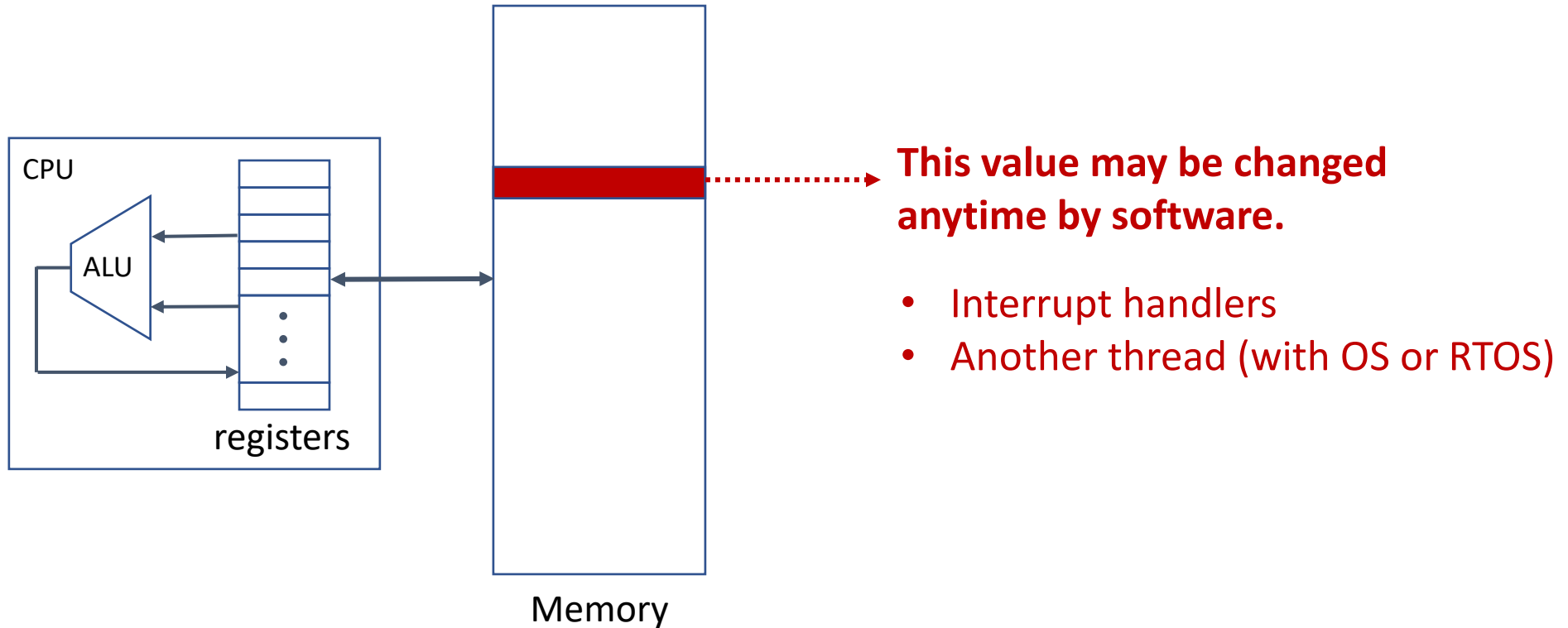
Volatile variables

- Qualifier **volatile** tells the compiler that this value may change at any time
 - Do not apply any optimizations to remove reads or writes to these variables



Volatile variables

- Qualifier **volatile** tells the compiler that this value may change at any time
 - Do not apply any optimizations to remove reads or writes to these variables



Three Common Usages of volatile

- Use volatile on accessing memory-mapped I/O registers

```
#define GPIOA_IDR (*((uint32_t volatile *) 0x48000010))

void main(void){
    ...

    // Read the input on pin 2
    Pin2 = GPIOA_IDR >> 2;

    ...
}
```

0x4800002C	ASCR
0x48000028	BRR
0x48000024	AFR[1]
0x48000020	AFR[0]
0x4800001C	LCKR
0x48000018	BSRR
0x48000014	ODR
0x48000010	IDR
0x4800000C	PUPDR
0x48000008	OSPEEDR
0x48000004	OTYPER
0x48000000	MODER

GPIOA Peripheral Registers

Three Common Usages of volatile

- Use volatile on accessing memory-mapped I/O registers
- Use volatile on global variables accessed in **interrupt service routines**
- Use volatile on global variables shared between **multi-thread tasks**

```
volatile uint32_t counter;  
  
void SysTick_Handler(void){  
    counter++;  
}
```

Summary

Volatile forces compiler to generate executable that performs actual memory reads and writes, instead of caching values in registers!

Read volatile
variable X in C



```
LDR r0, =x  
LDR r1, [r0]
```

Write to volatile
variable X in C



```
LDR r0, =x  
STR r1, [r0]
```

Syntax of volatile keyword

- Declare a variable volatile

```
volatile int x;
```

```
int volatile y;
```

Pointers

```
int* pi;  
int volatile* pvi;  
int* volatile vpi;  
int volatile* volatile vpvi;
```

Trick: Read from left to right

Pointers

- A pointer to an int

```
int* pi;
```

- A pointer to a volatile int

```
int volatile* pvi;
```

- A volatile pointer to an int

```
int* volatile vpi;
```

- A volatile pointer to a volatile int

```
int volatile* volatile vpvi;
```