마이크로프로세서 응용 설계
(**M**icroprocessor **A**pplication **D**esign)
Spring 2019

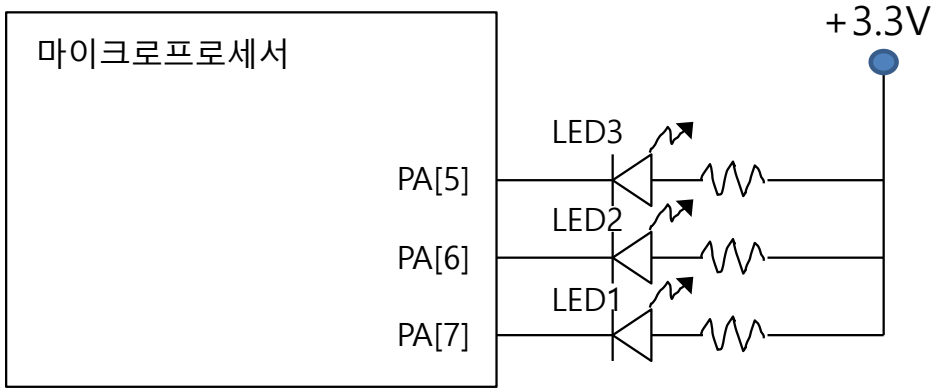
하드웨어 제어 및 최적화

전북대학교 컴퓨터공학부

Memory Mapped I/O

- 마이크로 프로세서는 주변장치를 메모리로 인식
  - 주변장치와 메모리의 구별이 없음
  - 주변장치에 데이터를 보내거나 주변장치로 데이터를 주고 받을 때는 메모리에 읽고 쓰는 것처럼 처리
- Memory mapped I/O vs. I/O mapped I/O
  - Memory mapped I/O
    - 입출력 장치들이 사용하는 메모리가 따로 정해져 있지 않음
    - I/O 장치를 접근하기 위한 명령어가 따로 없음
    - ARM, MIPS, …
  - I/O mapped I/O
    - 입출력 장치들이 사용하는 메모리가 따로 정해져 있음
    - I/O 장치를 접근하기 위한 명령어가 별도로 존재
    - Intel 계열 CPU

Example: LED 제어



Port Address (PA) = 0x3000_0000
PA[n] = 0, LED on

- PA
  - PA port의 주소: 0x3000_0000
  - 출력 전용 포트 PA[7:5]에 LED1~3이 연결
  - LED에 1을 넣어주면 꺼지고, 0을 넣어주면 켜짐
- LED 제어
  - LED1을 제어하려면 PA[7]에 접근해야 하고, 메모리 PA[7]에 원하는 데이터를 써야 함

LED 제어 – 첫 번째 코드

```
char *p = (char *)0x30000000;
*p = 0;
*p |= 0x1 << 6;
```

- 문제점?

- 특정 비트만 제어하기 위해서는?

Bit-wise set/clear

- 변수 A에 0x01010101이 저장되어 있다고 가정

- A의 5번 bit를 1로 set하려면?
  - `A |= 0x00100000; // Version 0.1`
  - `A |= 0x1 << 5;   // Version 0.2`
- A의 2, 3, 5번 bit들만 1로 set하려면?
  - `A |= (0x1<<5) + (0x1<<3) + (0x1<<2);   // Version 0.1`
  - `A |= (0x1<<5) + (0x3<<2);              // Version 0.2`

- A의 2번 bit를 0으로 clear하려면?
  - `A &= ~(0x1<<2);`
- A의 2, 3, 5번 bit들을 0으로 clear하려면?
  - `A &= ~((0x1<<5) + (0x3<<2));`

특정 bit 반전, bit 검사, bit 추출

- A의 5번 bit를 반전시키려면
  - `A ^= 0x1<<5;`
- A의 2, 3, 5번 bit들을 반전시키려면
  - `A ^= (0x1<<5) + (0x3<<2);`

- A의 5번째 bit가 0인지 1인지 검사하려면
  - `A & (0x1<<5)`

- A의 4, 5, 6 번째 bit들을 추출하고 싶으면
  - `A & 0x7`

Bit 연산을 편하게 – 매크로

- 매크로 vs. 함수
  - 각각의 장단점?

- 매크로에서 주의할 점

```
#define   ADD(x)        x+x
void main(void) {
  int k;
  k = -ADD(5);   // what happens?
}
```

```
#define   SQUARE(X)     (X*X)
void main(void) {
  int a = 2;
  printf("%d", SQUARE(a+5));   // what's wrong?
}
```

Bit 연산을 편하게 – 매크로 예제

```
#define  clear_bit(data,loc)        ((data) &= ~(0x1<<(loc)))
#define  clear_bits(data,area,loc) ((data) &= ~((area)<<(loc)))

#define set_bit(data, loc)              …
#define set_bits(data, area, loac)      …

#define invert_bit(data, loc)           …
#define invert_bits(data, area, loc)    …
```

LED 제어 프로그램 – Ver. 0

```
char *p = (char*) 0x30000000;
void init(void);
void display(int digit);

void main(void) {
  int i,j;
  init();
  while(1) {
    for (i=0; i<8; i++) {
      display(i);
      for (j=0; j<100000; j++);
      // delay loop
    }
  }
}

void init(void) {
  *P |= (0x7 << 5);
  // 메모리(PA[7:5])를 1로 설정하여 LED를 모두 끔
}
```

```
void display(int digit) {
  init();
  switch(digit) {
    case 1:
      *p &= ~(0x1<<5); // clear PA[5]
       break;
    case 2:
      *p &= ~(0x2<<5); // clear PA[6]
       break;
    case 3:
      *p &= ~(0x3<<5); // clear PA[6:5]
       break;
    case 4:
      *p &= ~(0x1<<7); // clear PA[7]
       break;
    case 5:
      *p &= ~(0x5<<5); // clear PA[7],PA[5]
       break;
    case 6:
      *p &= ~(0x3<<6); // clear PA[7:6]
       break;
    case 7:
      *p &= ~(0x3<<6); // clear PA[7:5]
       break;
  }
}
```

LED 제어 프로그램 Ver. 0의 문제점

- 메모리 0x3000_0000 번지에 100이라는 값을 쓰려면?

```
0x30000000 = 100;                 // Stupid
*(*)0x30000000 = 100;             // what's wrong?
*(int *)0x30000000 = 100;         // not bad
*(unsigned int *)0x30000000 = 100;   // Is this enough?
*(volatile unsigned int*)0x30000000 = 100; // That's it!!!
```

- 매크로를 이용해서 좀더 편하게 메모리 접근을 작성

```
#define PA (*(volatile unsigned char *)0x30000000)
void init(void) {
  PA |= (0x7 << 5);
}
```

LED 제어 프로그램 – Ver. 1

```
#define PA
(*(volatile unsigned char *)0x30000000)

void init(void);
void display(int digit);

void main(void) {
  int i,j;
  init();
  while(1) {
    for (i=0; i<8; i++) {
      display(i);
      for (j=0; j<100000; j++);
      // delay loop
    }
  }
}

void init(void) {
  PA |= (0x7 << 5);
  // 메모리(PA[7:5])를 1로 설정하여 LED를 모두 끔
}
```

```
void display(int digit) {
  init();
  switch(digit) {
    case 1:
      PA &= ~(0x1<<5); // clear PA[5]
      break;
    case 2:
      PA &= ~(0x2<<5); // clear PA[6]
      break;
    case 3:
      PA &= ~(0x3<<5); // clear PA[6:5]
      break;
    case 4:
      PA &= ~(0x1<<7); // clear PA[7]
      break;
    case 5:
      PA &= ~(0x5<<5); // clear PA[7],PA[5]
      break;
    case 6:
      PA &= ~(0x3<<6); // clear PA[7:6]
      break;
    case 7:
      PA &= ~(0x3<<6); // clear PA[7:5]
      break;
  }
}
```

LED 제어 프로그램 Ver. 1의 문제점

- 코드를 간결하게 만들어야 하는 이유
  - 임베디드 시스템에서는 RAM이나 ROM 등의 메모리 자원이 넉넉하지 않은 경우가 많다
  - 조건 분기가 여러 번 발생하는 경우 성능을 떨어뜨릴 수 있다 (why ?)

LED 제어 프로그램 – Ver. 2

```
#define PA
(*(volatile unsigned char *)0x30000000)

void init(void);
void display(int digit);

void main(void) {
  int i,j;
  init();
  while(1) {
    for (i=0; i<8; i++) {
      display(i);
      for (j=0; j<100000; j++);
      // delay loop
    }
  }
}

void init(void) {
  PA |= (0x7 << 5);
  // 메모리(PA[7:5])를 1로 설정하여 LED를 모두 끔
}
```

```
void display(int digit) {

  // 들어온 이진수의 첫째 자리가 1이면 LED3을 ON, 아니면 OFF
  (digit & 0x1)? (PA &= ~(0x1<<5)) : (PA |= (0x1<<5));

  // 들어온 이진수의 둘째 자리가 1이면 LED2을 ON, 아니면 OFF
  (digit & 0x2)? (PA &= ~(0x1<<6)) : (PA |= (0x1<<6));

  // 들어온 이진수의 셋째 자리가 1이면 LED1을 ON, 아니면 OFF
  (digit & 0x4)? (PA &= ~(0x1<<7)) : (PA |= (0x1<<7));

}
```

LED 제어 프로그램 Ver. 2를 좀더 개선하려면?

- 매크로 !!!

```
#define clear_bit(data, loc)
#define set_bit(data, loc)
#define check_bit(data, loc)
```

`volatile`?

- `volatile` 로 선언된 변수는 외부적인 요인으로 그 값이 언제든지 바뀔 수 있음을 뜻함
- 컴파일러는 `volatile`로 선언된 변수에 대해서는 최적화를 수행하지 않는다.
- `volatile` 변수를 참조할 경우 register에 로드된 값을 사용하지 않고 매번 메모리를 참조한다.

- 주로 사용되는 경우
  - Memory-mapped I/O
  - 인터럽트 서비스 루틴
  - Multi-thread 환경

volatile 키워드의 사용법 (1/2)

- Example 1
  - 잘못된 하드웨어 제어 코드

```
*(unsigned int *)0x8C0F = 0x8001
*(unsigned int *)0x8C0F = 0x8002;
*(unsigned int *)0x8C0F = 0x8003;
*(unsigned int *)0x8C0F = 0x8004;
*(unsigned int *)0x8C0F = 0x8005;
```

같은 주소를 반복적으로 변경하므로 컴파일러 최적화에 의해 제거됨

  - 올바른 하드웨어 제어 코드

```
*(volatile unsigned int *)0x8C0F = 0x8001
*(volatile unsigned int *)0x8C0F = 0x8002;
*(volatile unsigned int *)0x8C0F = 0x8003;
*(volatile unsigned int *)0x8C0F = 0x8004;
*(volatile unsigned int *)0x8C0F = 0x8005;
```

volatile 키워드의 사용법 (2/2)

- Example 2

```
void foo(char *buf, int size)
{
      int i;
      volatile char *p = (volatile char *)0x8C0F;

      for (i = 0 ; i < size; i++)
      {
          buf[i] = *p;
          ...
      }
}
```

같은 주소에서 반복적으로 읽기를 하므로 컴파일러에 의해 최적화되지 않도록 volatile로 선언해야 한다


9/9
0800 Antan started
1000 " stopped - antan ✓
13" uc (032) MP - MC
(033) PRO 2 2.130476415
cond 2.130676415
Relays 6-2 in 033 failed special speed test
in relay " 11.000 test .
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Mult + Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
1630 antangent started.
1700 closed down.

집합원소의 효율적인 저장 방법

- Array index의 활용 1

```
switch (choose)
  case 1:
    text = 'a';
    break;
  case 2:
    text = 'b';
    break;
  case 3:
    text = 'c';
    break;
}
```

```
static char *buf = "abc";
Text = buf[choose];
```

- 오른쪽 코드의 2가지 장점은?

집합원소의 효율적인 저장 방법

- Array index의 활용 2

```
switch (choose)
  case 1:
    f1();
    break;
  case 2:
    f2();
    break;
  case 3:
    f3();
    break;
}
```

```
void (*p[3])() = {f1, f2, f3};
P[choose]();
```

집합원소의 효율적인 저장 방법

- 구조체의 padding bit 줄이기

```
struct Test1 {
  char a;
  int b;
  char c;
  int d;
};

void main() {
  printf("%d\n",
    sizeof(struct Test1));
}
```

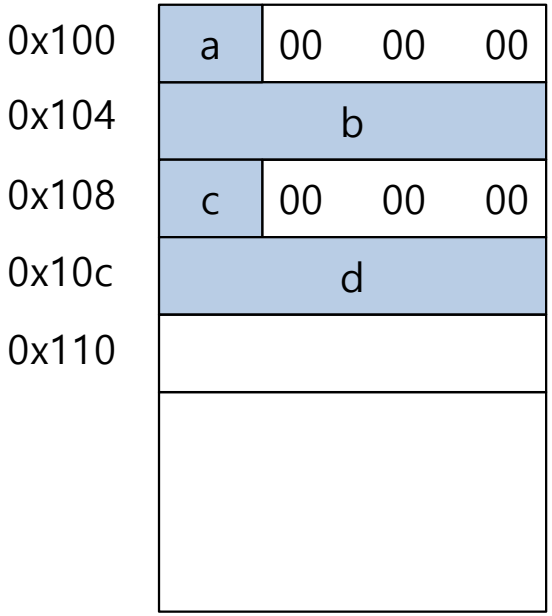
```
struct Test2 {
  char a;
  char c;
  int b;
  int d;
};

void main() {
  printf("%d\n",
    sizeof(struct Test2));
}
```

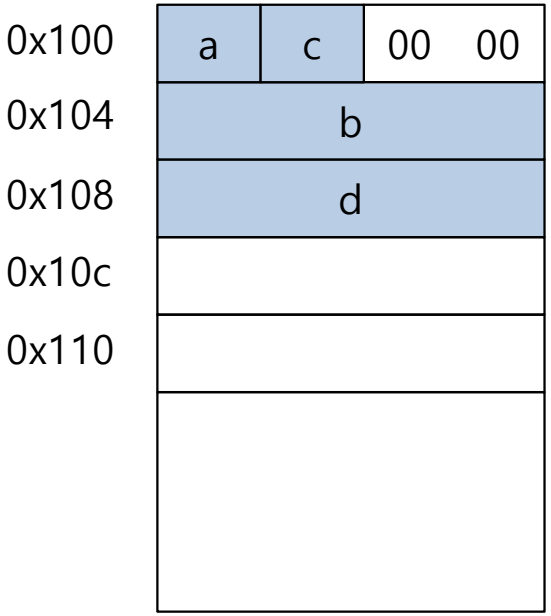
- 두 프로그램의 실행 결과는? 그 이유는?

집합원소의 효율적인 저장 방법

- 구조체의 padding bit 줄이기
  - 데이터 타입 별로 메모리에 저장될 수 있는 주소가 달라진다
  - `char`: 어느 주소나 가능, `short`: 2의 배수, `int`: 4의 배수



`struct Test1`



`struct Test2`

분기문 최적화 – `if`: Example 1 (ver. 0)

```
#include <stdio.h>
void main() {
  int a, b, c, d, x=0;
  a = b = c = 3;
  c = 5;
  d = 6;
  if (((a*c)+b)/d==0)
    x = ((a*c)+b)/d + 5;
  else if (((a*c)+b)/d==1)
    x = ((a*c)+b)/d + 10;
  else if (((a*c)+b)/d==2)
    x = ((a*c)+b)/d + 15;
  else if (((a*c)+b)/d==3)
    x = ((a*c)+b)/d + 20;
  else if (((a*c)+b)/d==4)
    x = ((a*c)+b)/d + 25;
  else
    x = 0;
  printf("%d\n", x);
}
```

- 오른쪽 코드의 문제점?
  - 가독성
  - 성능

분기문 최적화 – `if`: Example 1 (ver. 1)

```
#include <stdio.h>
void main() {
  int a, b, c, d, x=0;
  a = b = c = 3;
  c = 5;
  d = 6;
  t = ((a*c)+b)/d;
  if (t==0)
    x = ((a*c)+b)/d + 5;
  else if (t==1)
    x = ((a*c)+b)/d + 10;
  else if (t==2)
    x = ((a*c)+b)/d + 15;
  else if (t==3)
    x = ((a*c)+b)/d + 20;
  else if (t==4)
    x = ((a*c)+b)/d + 25;
  else
    x = 0;
  printf("%d\n", x);
}
```

- 개선된 코드
- 좀 더 최적화 가능?

분기문 최적화 – `if`: Example 1 (ver. 2)

```
#include <stdio.h>
void main() {
  int a, b, c, d, x=0, t;
  a = b = c = 3;
  c = 5;
  d = 6;
  t = ((a*c)+b)/d;
  if (t==0)
    x = t + 5;
  else if (t==1)
    x = t + 10;
  else if (t==2)
    x = t + 15;
  else if (t==3)
    x = t + 20;
  else if (t==4)
    x = t + 25;
  else
    x = 0;
  printf("%d\n", x);
}
```

- 오른쪽 코드에서 개선된 점?

분기문 최적화 – `if`: Example 1 (ver. 3)

```
#include <stdio.h>
void main() {
  int a, b, c, d, x=0, t;
  a = b = c = 3;
  c = 5;
  d = 6;
  t = ((a*c)+b)/d;
  switch (t) {
    case 0: x = t + 5; break;
    case 1: x = t + 10; break;
    case 2: x = t + 15; break;
    case 3: x = t + 20; break;
    case 4: x = t + 25; break;
    default: x = 0;
  }
  printf("%d\n", x);
}
```

- switch 문이 if 문보다 성능이 좋은 이유는?
  - 비교, 분기 횟수

분기문 최적화 – 케이스가 많은 if 문

- 최적화이전

```
if (a==1) {
} else if (a==2) {
} else if (a==3) {
} else if (a==4) {
} else if (a==5) {
} else if (a==6) {
} else if (a==7) {
} else if (a==8) {
}
```

- 최적화 이후
  - Binary breakdown

```
if (a<=4) {
  if (a==1) {
  else if (a==2) {
  else if (a==3) {
  else if (a==4) {
  }
} else {
  if (a==5) {
  else if (a==6) {
  else if (a==7) {
  else if (a==8) {
  }
}
```

분기문 최적화 – 케이스가 많은 switch 문

```
if f1(int a)
{
  switch(a) {
    case 0: return 0x3f;
    case 1: return 0x6;
    case 2: return 0x5b;
       .
       .
       .
    case 213: return 0x61;
    default:
  }
}

void main(void)
{
  f1(3);
}
```

- 왼쪽 코드의 문제점?

```
if f1(int a)
{
  int b[] = {0x3f, 0x6, 0x5b, …, 0x61};
  return b[a];
}

void main(void)
{
  f1(3);
}
```

루프 (Loop) 최적화

- 어떤 코드가 더 빠른가? 그 이유는?

```
int i, j=10;
for (i=0; i<20; i++) {
  function(j);
}
```

```
int i, j=10;
for (i=0; i<20; i+=4) {
  function(j);
  function(j);
  function(j);
  function(j);
}
```

Loop Unrolling: Example 1

- 실행되는 loop의 횟수를 줄이는 기법

Without loop unrolling

```
int i, a[100];
for (i=0; i<100; i++) {
  a[i] = i + 1;
}
```

With loop unrolling

```
int i, a[100];
for (i=0; i<100; i+=4) {
  a[i] = i + 1;
  a[i+1] = i + 2;
  a[i+2] = i + 3;
  a[i+3] = i + 4;
}
```

- Loop를 수행할 때 발생하는 비용
  - CPU pipeline stall
  - Loop counter management

```
for (i=0; i<100; i++)   // 비교연산 100회
                        // 카운터 덧셈 100회
```

Loop Unrolling: Example 2

```
int i, a[100];
for (i=0; i<67; i++) {
  function(j);
}
```

```
int i, a[100];
for (i=0; i<66; i+=11) {
  function(j);
  function(j);
  function(j);
  function(j);
  function(j);
  function(j);
  function(j);
  function(j);
  function(j);
  function(j);
  function(j);
}
function(j);
```

- 위의 코드들의 문제점은?
  - Function vs. inline
  - Cache의 영향?

Loop Unrolling: Example 3

```
int cntbit(int n) {
  int count = 0;
  while (n != 0) {
    if (n & 0x1) count++;
    n >>= 1;
  }
  return count;
}
```

```
int cntbit(int n) {
  int count = 0;
  while (n != 0) {
    if (n & 0x1) count++;
    if (n & 0x2) count++;
    if (n & 0x4) count++;
    if (n & 0x8) count++;
    n >>= 4;
  }
  return count;
}
```

Loop Fusion

- 2개의 Loop를 하나로 합침

```
for (i=0; i<N; i++) a[i] = b[i] * 5;
for (j=0; j<N; j++) w[j] = c[j] * d[j];
```

➔

```
for (i=0; i<N; i++) {
    a[i] = b[i] * 5;
    w[i] = c[i] * d[i];
}
```

  - 장점 vs. 단점?